Draft revision 3 for ECMA Standard 367, Eiffel.
Internal document of ECMA TC49-TG4 group
Not for public use.

This version: 113, 3 November 2025

By: Bertrand Meyer.

For the Record of Changes: see page iii

ecma INTERNATIONAL

Standard ECMA-367

3rd Edition / June 2026

Eiffel: Analysis, Design and Programming Language

Signed by JV - 2026/10/8

Signed by JV - 2026/10/8

Record of Changes towards the completion of the Standard

Informative text

THIS CHANGE LOG IS NOT PART OF THE STANDARD

The present Record of Changes (pages iii to xx) will be **removed in its entirety from the final text of the Standard**.

It is included in draft versions such as the present one for the sole purpose of **supporting the work of the Ecma TC49-TG4 standards** group towards completion of the 367-3 revision of the international Eiffel standard.

The style is informal and the record is incomplete. The systematic part starts with revision 101 (July 2024). In addition, we have included change logs that had been recorded for some earlier versions going back to 2009 (with the most revisions in 2015 and 2020). Even so, the change log does not do justice to the considerable work of the committee in the move than 10 years before, since the publication of revision 367-2 (Ecma and ISO) from 2006, and the earlier 367-1, approved by Ecma in 2005.

Both the PDF of the latest version and the FrameMaker sources can be found at

https://©ln5.sync.com/dl/4dd892ef0/k3jv5z82-9mg6nxqp-vptvtznc-kgkq3iht

(for the sole use of TC49-TG4 members — do not publicize this URL).

Starting in July 2024, revisions come out every Saturday (later changed to Sunday).

Version 113, 3 November 2025

Revision author: Bertrand Meyer.

Version 112, 30 October 2025

Revision author: Bertrand Meyer.

Reworked subclause 8.8, features. Extensive rework, too many changes to be listed here.

In constants, replaced Constant_attribute by Constant_name (that's what it is), reused Constant_attribute for the corresponding variant of Attribute in the Features subclause (in Feature_definition).

Terminology: replaced "explicit attribute" by "defined attribute" (for clarity).

Open question: I think we had this discussion before but I am now uneasy with using, for Variance_mark, the choices **variant** versus **invariant**. I would prefer nothing versus **frozen**. Enlightenment on this matter would be welcome.

Version 111, 25 September 2025

Revision author: Bertrand Meyer.

New pass on subclause 8.7: Clients and exports

I did not yet process the rest of Eric Bezault's earlier corrections.

Important text processing improvement: all "Informative Text" paragraph are now labeled. Normal paragraphs have 4-level numbers, such as 8.7.8.3 (relative to the preceding 2- or 3-level element); bullet points, which are always relative to such a paragraph, have 5-level numbers, such as 8.7.8.3.1. These labels are typeset with a very small font so as not to impair readability (if it is too small, tell me).

Version 110, 16 January 2025

Revision author: Bertrand Meyer (integrating corrections by Éric Bezault).

Revision author: Bertrand Meyer. Finished subsection 8.7 (Clients and exports).

No fundamental language changes but thorough revision of the presentation.

Everything is fully explained in Informative Text.

Simplified the specification of "client" (removing unnecessary variants.)

Version 109, 9 December 2024

Revision author: Bertrand Meyer (integrating corrections by Éric Bezault).

Finished subclause 8.6 (classes).

Integrated comments by Éric Bezault on clauses 8.1 and 8.2.

Version 108, 17 November 2024

Revision author: Bertrand Meyer

Finished subclauses 8.3 (writing conventions), 8.4 (basic concepts), 8.5 (architecture). The end of the last one was done a bit fast.

Turned the letter case conventions (now 8.3.3) into a formal definition, D7LC, page 71, rather than just Informative Text. That does not imply a change of substance, since the conventions remain recommended rather than required; it is their definition that becomes part of the standard.

Clarified c the notion of universe and what it means for a Class_name to be valid.

Version 107, 6 October 2024

Revision author: Bertrand Meyer

Starting with version 106 of last week, major text processing matters have been discharged and as a consequence the work is now on finalizing the actual contents, in the order of the chapters. As a result, this is the first revision for which feedback is again welcome!

For this version: feedback requested for subclauses 8.1 (general organization) and 8.2 (language description conventions).

Work on this version included:

- Finishing 8.2.
- In the process, fixing a mixup (copy-paste error) in the production for Result_type (involving the Precursor_type possibility) in SQFM.

Version 106, 29 September 2024

Revision author: Bertrand Meyer

Apologies for the interruption of more than one month.

I have started going through the entire text (except the Tutorial), finished subclause 8.1 (general organization) and started 8.2 (conventions), up to page 58 (the stopping point is marked). In the process I cleaned up the syntax summary (now clause 8.35, it did not have a number before) and created another one meant to be in alphabetical order of the constructs.

Given the conventions that I used the alphabetical one is not alphabetical yet and I cannot use FrameMaker to produce it even though it has a "Produce alphabetical list of paragraphs" option. (The reasons are complicated.) It is not that hard to write a script, e.g. in Eiffel, which processes the MIF file to produce such a list. Anyone interested in helping can take a look at the file 367_3_STANDARD-ALPHA.mif in the MIF subdirectory. The paragraphs that should be ordered alphabetically are those tagged mm-productionLOP, kept together with the associated ones of tag mm-prodnextLOP. These are the only two paragraph tags, other than the tags for the subclause header and the initial paragraphs. (The rest of the text, mm-productionLOP and mm-prodnextLOP paragraphs, is regenerated automatically each time, so that I can save it to MIF and run the script.) One does not need to learn MIF, just look at the content and figure out what needs to be done. It should take a couple of hours for a good programmer. One should install a trial version of FrameMaker to be able to check the result -- I think they are available for 2 weeks. If you are willing to take care of it please tell me and I will help. Otherwise I will take care of it at some point down the road.

Version 105, 24 August 2024

No record for that version.

Version 104, 17 August 2024

Revision author: Bertrand Meyer

I am circulating this revision in advance since I will not be able to work on it on Aug 16 and 17. It is a fairly small revision but it has a good part of the hyperlinking covered so the subsequent revisions will focus on substance — in other words, the serious work is restarting.

Updated this Record of Changes to include (in a rough format) all the change logs that I could find going back to 2009. (Converted from MS Word.) The record below includes all of them.

Advanced the hyperlinking — the core of it is done now.

Swapped subclauses 8.2 and 8.3 so that the description conventions appear before the language essentials.

Version 103, 10 August 2024

Revision author: Bertrand Meyer

Version 103 essentially introduces many new hyperlinks. I think I will continue the hyperlinking for another week and then move on to content.

Cleaned up many points in the process. Among others, the notion of "root object" was not properly defined.

Signed by JV - 2026/10/8

A change made in version 102 and forgotten in its entry below: introduced the term "Paren_expression" for the form of expression that uses a parenthesis alias, similar to "bracket expressions". Previously such expressions were called "parenthesis expressions", but I found the term subject to confusion with "Parenthesized", which simply covers expressions in parentheses, as in (e). Using the abbreviation "paren", whereas abbreviations are rare in Eiffel and in the Standard, signals something special and should avoid basic confusions.

Updated the notion of "redeclaration" so that it also includes undefinition (not just redefinition and effecting).

Releases from now on will appear on Saturdays (rather than Fridays) to leave more time for finishing up.

Version 102, 2 August 2024

Revision author: Bertrand Meyer

Version 102 is an important text processing milestone: every reference to a syntactical construct, in the (green-colored) font used for that purpose, such as Feature_definition, is now a **hyperlink** to the production defining the corresponding construct. This property has been checked throughout, so at the present state we believe there is no exception whatsoever. **It must be maintained invariant through all subsequent revisions of the standard**.

At the moment, these hyperlinked references are marked not only by their color but also by underlining: Feature_definition. **The underlining is temporary.** It served in the hyperlinking process to distinguish the construct-name occurrences that had already been turned into hyperlinks for this that remained to be processed. It will be removed in the final version, so that the hyperlink will look just like this: Feature_definition — just a construct name, distinguished as such by its color, which happens also to be clickable, leading to the construct definition if one clicks on it.

As as side note, there are an estimated 1500 such links, each one entered manually. The effort took a full week. The total number of manually entered hyperlinks (including those other than construct names) is probably around 3000. It is very tedious work (I feel a certain kindred spirit with medieval monks), although it helps clean up the text.

Indeed in the process I corrected many errors and inconsistencies, and added a few notes of remaining (usually small) corrections that need to be made in the future.

Also in this version: there is now a change log, starting on page iii of the document. It will be updated for every weekly version. It is not part of the Standard, only for the current drafts, and will be removed in due course from the text.

Plan for the next version: I will continue the hyperlinking work to ensure that we have a text-processing-wise impeccable document and can, in the next phases, focus on the contents.

Version 101, 25 July 2024

Revision author: Bertrand Meyer

(The following description is copy-pasted from the email sent to the Technical Group on that date.)

Dear TG4 members,

From now on I am going to try to circulate a version every Friday. (This time I am early by a few hours.)

I am starting the numbering, somewhat arbitrarily but not absurdly given the amount of iterations already executed so far, at 101. It is timed to coincide with the start of the Paris Olympics – appropriately for our Marathon effort.

At this point:

- I am still mostly focusing on the hyperlinking. Someone (I think I know who) suggested that every underlined element should be a hyperlink, which sounds great, but it means entering thousands of cross-references manually. Since the last version I would guess a couple thousand.
- As a result I am not keeping a change log. It would be several pages long and not be interesting at all ("changed all mentions of conforms to to links to 8.18.4 (VNCS) if applied to a signature, to xxx if applied to a type, to zzz if applied to an expression"). When I get to changes of substance I will start including change logs with each version.

If you see mistakes, typos or other anomalies there is no point in sending them to me at this stage where I am just trying to get my head above the water. You can ignore typos and other small matters (as careful proofreading will have to take place in due time) but if you like you can keep a record of important points to be discussed later on.

As you will see the hyperlinking has progressed significantly. When it has reached some 95% of underlined text (or text that should have been underlined) I will move to the serious part; in particular I will resume reading and implementing the earlier meetings' minutes. Since right now I have more time than during the semesters I hope that will be around August 20.

The attached version should still serve as a useful records.

I am still looking for a safe place (in addition to my own paranoid local and cloud-based backups) to keep the FrameMaker sources and other auxiliary materials such as meeting minutes and earlier versions.

I hope you are having a pleasant summer,

-- Bertrand Meyer

Latest book: *Handbook of Requirements and Business Analysis*, Springer, 2022

A treatise and textbook on requirements, see https://se.ethz.ch/requirements

Earlier versions

Below is a copy-paste from the change log of a number of versions in 2020. It is essential copy-pasted from an informal Word document; as a result, the formatting is approximate.

Section numbers without further qualification are to ECMA 367-2. As the language specification

appears in section 8 of the standard, all relevant section numbers start with "8."; for

concision, the "8." part

is omitted in all section numbers of the present document.

Section references preceded by "AM-" refer to the all_minutes document.

Normally page numbers refer to this document as well (not to the standard).

For changes in the "AM-" category, the section numbers have been prepended with the meeting number

followed by a bar, e.g. 19|5.2 appears as just 5.2 in the all_minutes document (as per email sent on 8.11.2009 I hope they can be updated).

Section references preceded by "CO-" are corrections by me, usually without discussion in the committee. This category is used

only for corrections of obvious typos and other non-controversial changes.

Signed by JV - 2026/10/8

Versions appear in reverse chronological order (most recent first).

Version 42, 22 November 2020

Text processing and document organization

Found some redundancies to be cleared later (no difficulty, simply did not have time): PFFI and PQFI are redundant; Alias syntax rule (8.6.22 AND 8.8.3).

Hyperlinked features chapter (F) and syntaxed it (i.e. replaced all references to constructs by hyperlinks). Note that this effort took about 6 hours.

Syntaxed routines chapter (R).

Expressions

Fixed poor wording (and mistakes) in the definition of Operands, DWOP.

Replaced remaining occurrences of Basic_expression by Expression throughout.

Version 41, 8 November 2020

Text processing and document organization

[Change to this Change Log itself: I reversed the order of entries so that from now on they appear in reverse chronological order, most recent first.]

Removed remaining occurrences of the “Times New Roman” font and replaced them by “Arial”, which is the font used by the overwhelming majority of the text (and if I remember right the preferred Ecma font).

Moved the version identification block from the beginning of Partition 8 (where it made sense up to version 39 since it was all I was publishing, but now I am publishing the draft of the entire standard) to the title page. To avoid mistakes and tedium I removed the one other occurrence of the word “draft”.

Fixed the bookmarks and tags in the generated PDF so that there is a proper clickable table of contents (with everything at the right level, including part numbers) in the PDF reader (checked on Acrobat only).

From now on I will use the verb “to hyperlink” (a chapter) to mean that I transformed all underlined text in a chapter to hyperlinks pointing to the corresponding definitions. In this release I hyperlinked the genericity chapter.

“Conventions” chapter (chapter 2)

I cleaned up this chapter and made sure that every single rule is followed by Informative Text. Some of those informative texts are failing long, so the chapter is gaining 2 pages. By and large, applying the same principle (no rule without its immediately following Informative Text) will increase the length of the standard’s text by an estimated 20%..

SCOOP chapter

Added a chapter on concurrency (code 6), currently clause 27. Empty so far (and probably for a while).

Genericity chapter

In preparation of the next meeting, cleaned up and “hyperlinked” the genericity chapter. Added significant Informative Text.

Moved the definition of “features of a type” and “current type” (DGFE and DGCT) to the type chapter. Note two corrections that were made in the FrameMaker text but are not visible in the PDF yet (I was too lazy to regenerate it after noticing the problem, they will be in the next release):

Changed the codes to DTFE and DTCT as a result of this move.

Fixed a typo in the reference to the term “genericity” in the Informative Text after DGFE/ DTFE (currently 8.13.14).

Signed by JV - 2026/10/8

Version 40, 1 November 2020

Precursor

This is a major new release. I worked most of the week on it -- even if there is not that much to show for my efforts (since much of the work was restructuring).

I started out by trying to finish the addition of preconditions etc. to attribute bodies. Little by little I was led to a major text-processing restructuring of the text (see below).

I also took to heart the request to have all underlined elements be hyperlinks to the proper definition. This led to another major restructuring. I spent about three full days on it; it's finished for the first two chapters (clauses): conventions (8.2) and architecture (8.3). To finish it will take about one hour per chapter, so with ca. 30 chapters remaining it will be about four full days of work. I hope it's worth it.

General restructuring

I split the main part (partition 8) into one file per chapter, taking advantage of the FrameMaker book mechanism. (I was using it before but for the whole standard only, partition 8 was a single file, originally generated automatically from ETL but having taken its own flight since). Of course this required updating all the cross-references (although FrameMaker helps here.

For those interested in the FrameMaker sources, the file names start with the corresponding chapter code in upper case (used as the second character of the rule codes, as the F for the feature chapter, e.g. VFDF for its "Feature Definition" Validity rule); for example the file name for the Features chapter is F-feature. All the letters of the alphabet are taken plus digits 1 to 5 (the digit chapters start with lower-case z, e.g. z3-constant).

References to definitions as hyperlinks

I set up the machinery to enable references to definitions to be hyperlinks. I must still process every single one of them (I have processed the targets -- that's the three days mentioned above -- but I will continue chapter by chapter over the coming weeks. For clarity, those that have been processed (turned into a hyperlink) are not only underlined as before but also marked in a special mauvish color, as in:

feature of a class

When they have all been processed, i.e. every reference in a hyperlink, I will switch back to black (fortunately this is the kind of change that is easy to do with FrameMaker, even over 40 files or so) -- consider the (ugly) color just a temporary help during that process.

By the way, there is officially no way in FrameMaker to do what is described above. More precisely, you can do it by adding a marker of the form `**newlink** special_name' to every target position (every definition) and a marker of the form `**gotolink** filename:specialname' to every source position (every reference to a definition). You must type all of this each time. (FrameMaker on Unix used to have a macro facility which would have helped a bit, but it is not available on the Windows version I am using now. Some people sell a scripting facility, but it looks clumsy to use and would help only marginally.) Imagine typing those thousands of times! (I am not exaggerating, look at the number of references in the text and imagine all those that will be added.) Impossible without going mad. I asked on the FrameMaker user list how to achieve the goal and once people understood they either suggested non-applicable suggestions or said it was impossible, which is correct. However I devised a clever or crazy (opinions may vary) scheme which actually achieves it at reasonable effort. (Well, still the equivalent of a week or two full-time in the end, but doable.)

As noted above, you can see the result in chapters (clauses) 8.2 and 8.3.

"Parts"

Signed by JV - 2026/10/8

As a general rule I am trying to make the text more readable and attractive. In particular, I am adding lots of Informative Text. Also, I divided the text into 6 or so “Parts”. I will have to check with Ecma how this extra structure can be accommodated formally.

Removed “attributes” chapter and new “feature names” chapter

In the process of adding facilities to attributes with bodies I realize the attribute chapter (Q) was very short and much of its content was better put together with “features” (F). I moved it there. As a result, a validity constraint changes code: VQMC (Manifest Constant rule) becomes VFCA (I also changed the name to Constant Attribute rule since this is what it is about, hence the new mnemonic CA). [NOTE for version 112, Oct. 2025: I found no VMCA but reintroduced this name for one of the rules.]

The Features chapter was becoming too big; it has a discussion of feature names that detracts from the discussion of features, so I moved it to a separate chapter for which I reused the chapter code Q (freed by the previous change). Again one validity code changes as a result: VFAC becomes VQAC.

Attributes with baggage

I took care of the essentials of adding baggage (preconditions etc.) to attributes. There are still a few details to iron out, and Informative Texts to update.

The syntax description becomes in fact simpler; for example Attribute_or_routine disappears.

I chose (as recommended, see messages by Eric Bezault) the second of the two variants discussed. Note that the keyword attribute **may not** be used for a constant.

Precursor

“Precursor” (SDPR and following paragraphs) covered two different things: the precursor type and its use to represent a routine. I separated them into two parts, Precursor_type and Precursor_call. I correspondingly split the validity rule, VDPR, into two.

Removed use of relative unfolded form -- it seems not to have been used.

Making attributes into full-fledged features (with contracts etc.)

Revised the specification of Feature declarations (SFDE and rest of the section) fairly extensively. The good news is that it’s actually significantly simpler now.

“Features of a class”

There were references to this notion but I could not find a definition. I added it at the beginning of the “features” chapter (“clause”) as DFFC

(Note: I will continue to use “section” rather than the official Ecma “clause” to avoid confusion with “clauses” of Eiffel rules. This convention is only for the present change notes; the standard will of course continue to use official Ecma terminology.)

Assertion adaptation in inheritanc

Moved the corresponding elements from the “feature adaptation” chapter (code D) to the contract chapter (code A). Changed the codes accordingly.

(The choice between these two chapters is somewhat arbitrary but I feel the clauses detract from the general discussion of feature adaptation in D.)

“Precondition-free”

This concept was referenced but not defined. Added it as DAPF.

Contract chapter

Spurred by the preceding changes I thoroughly reorganized the assertion chapter to organize it into four successive marks (delimited by “Informative Text”, maybe I will find a more systematic way of cutting chapters into pieces):

- The basic assertion concepts and constructs (8.

- to 8.).
- The effect of inheritance on contracts (8.
- to 8.).
- Notions of correctness (8.
- to 8.).
- Run-time effect of contracts (from 8.

Initial clauses and conventions chapter

I started to improve the initial partitions (1 to 6). I did not touch the substance of the tutorial (partition 7), however, this will probably come last.

Version 39, 25 October 2020

This is a significant revision, hopefully the start of a new phase with weekly updates, towards finalization of revision 3 of the standard.

I also integrated the comments on revision 36 made by Éric Bezault in an email of 2018.

Integrating Éric Bezault's comments on version 36 (from 2018)

The email was from 20 April 2018 (9:37 at least in some time zone) and was entitled "*Review of the Eiffel standard: chapters 6, 7, 8 and 9*".

I have integrated all of his comments except as follows:

SFDE (8.5.4) I am not sure why we need to wait until chapter 8.8 in order to see the syntax definition of Formal_arguments.

PFAI: Suggestion: it would be interesting if we could specify several aliases for the same feature.

SFNA target-conversion

8.5.27 "or parenthesis alias" – I don't understand this comment, the Parenthesis alias is taken care of in the next condition.

I did nothing for the comments on VLEL – the committee should discuss what the rules should be here.

Comment on DLSP: did nothing yet, needs discussion.

Comments on SRFE require discussion.

Did not process comment on DAUL.

I expanded on some of the comments to update properties of Object_call to general Call (export-valid etc., see change notes).

DGBM: ignored comment, must be discussed in committee.

VGCC and following clauses: I skipped the comments because this must clearly be discussed in the committee.

DIOS and other rules relating to multiple constraints: added that an operator symbol must be non-break.

I ignored the comments on DIFO since we need to review free operators (and operators in general) in light of Unicode.

I ignored comments of the form "is there an Eiffel compiler supporting this"?

stable attributes and corresponding keyword and sections: I ignored all that part since the idea was to use the "then TYPE" syntax instead. I will add it at some point.

Unreported changes

I am making numerous improvements (rephrasings, typo corrections) as I go, adding Informative text boxes (the goal is that every rule, definition etc. should have its Informative text) and improving the existing ones. Most of these changes are not reported here.

Miscellaneous

Added clause (7) to VHPR addressing the issue of anchored types in actual parameters. Updated Informative Text accordingly (and took the opportunity to improve it overall). See email to committee.

In VSRT (Root Type rule), in the process of adding a comment from Éric Bezault, replaced “not deferred” by “effective”). Added Informative Text per his suggestion.

In VSRT, swapped conditions 3 and 4 (for clarity of exposition).

In SFOP (Operator syntax, currently 8.5.21) added, in Informative Text, comment that free operators are less critical today with Unicode.

VFAW conditions 2 and 3: replaced “query” and “routine” by “function”.

DHUI: added a case (now 2) for an Inheritance clause with only Non-conforming parents.

Assertion monitoring

I realized that **the semantics of assertion monitoring appears in two places**: MUGC (General Call Semantics) and MAAM (Assertion Monitoring rule). This is wrong! I replaced the applicable clauses of MUGC by

If postcondition monitoring is applicable per the Assertion monitoring rule, evaluate the postcondtion of df.

and similarly for preconditions and class invariants.

Non-object calls and generalization of the notion of call

Searched for all occurrences of Object_call and either checked that the corresponding comments apply only to this construct or changed the text so that it applies to Call in general.

DUSF (currently 8.23.11): replaced Object_call by Call.

VUTA (definition of “target-valid”): added case for Non_object_call.

VUEX (definition of “export-valid”): replaced Object_call by Call. Checked that the definitions of argument-valid and target-valid are OK in this respect.

Generalized “Operands of a call” so that it applies to non-object calls as well (simply changed “target” to “target, if any”).

Correspondingly improved the definition of “operands of a call” and “operand position” (DPOP, DPPO, codes changed to DUOP and DUPO for consistency with others in the chapter).

Introduced notion of a call being “targeted” so that:

- A qualified Object_call is targeted and qualified.
- A Non_object_call is qualified and untargeted.
- An Unqualified_call is unqualified and targeted.

Swapped SUAA (Syntax: actual arguments) and the definitions of “qualified” (and now“targeted”) for a more logical order.

SUCA (syntax for calls): no change to the syntax, but extensively rewrote the Informative text.

MAAM and MUGC (assertion monitoring): replaced “qualified” by “targeted”.

DTTC (4) (transposition): replaced “qualified call” by “Object call with an explicit Target”.

DDDTR (4) (transposition): same thing.

VBAC (2) (assigner command): replaced

target is a qualified Object_call, Bracket_expression or Parenthesis_expression

by

target is a targeted Call, a Bracket_expression or a Parenthesis_expression

Signed by JV - 2026/10/8

What I didn't do: update VUNO so that it reflects the relaxed conditions for features of non-object-call.

Cosmetics

Replaced uses of "l" in informative texts to denote a tuple label by "x". Simply because in the Arial font the "l" when used by itself is a bit confusing -- could be "1" or "I".

Loosening constant syntax

Loosened the syntax for constants to allow for obsolete, notes, pre- and postconditions as requested. This includes the following.

Syntax

SFDE (Feature_declaration):

- Remove the definition of Feature_value (see next)
- Remove Attribute_or_routine. Its role is now taken over by Feature_value
- Feature bodies (then and currently 8.5.6), syntax code is SFVA:
- Change title of the section to "Feature values"
- Move definition of Feature_value here, merging with the clauses of what was previously called Attribute_or_routine
- In that same definition, make Feature_body also optional (with either it or Explicit_value having to be present, see validity).

Validity

- Old VFFB (Feature Body rule) is now called the Feature Value rule, and has code VFVA. Sorry for this, it may create some confusion, but the previous name was misleading (the rule begins "A Feature_value is valid if and only if...").
- VFFB (1): changed to require exactly one of Explicit_value and

Version 38, February-May 2020

No record for this version

Version 37, 31 March 2018

Updated DWSE and DWET according to resolution of meeting 39: made order of clauses compatible. See email of 16 February 2018 to TC49.

Version 36 (2016-2017)

I have no record for this version. Sorry. In most likelihood this is just a version number I prepared for further work, then I did not do anything, but I do not recall for sure.

Version 35, 18 August 2015

Added numerous "Informative text" entries. My goal is that in the final version every single formal definition or rule will be followed (and sometimes preceded) by Informative Text).

Introduced "feature name of a call":

(1) Replaced the definition of "feature of a call" by "feature name of a call" (new code SUFN).

(2) In Informative Text, removed paragraph saying it's an abuse of language to call it feature of the call.

(3) Introduced notion of static feature of a call.

Signed by JV - 2026/10/8

(4) Simplified the definition of "dynamic feature of a call" (DUDF) to take advantage of the notion of static feature.

(4) Hunted instances of "feature of a call" (cannot guarantee I found them all) and fixed them, replacing by one of "feature name", "static feature" and "dynamic feature" of the call:

- VAPE
- DUDF (dynamic feature of a call)

Reorganized clauses in 8.23 (calls) for more continuity (moved the Non_object_call specs, starting with SUIF, after Object_call).fs

Added feature version theorem: the dynamic feature is a version of its static feature.

Standardized the presentation of "principles", explaining for each of them, in the Informative Text that follows its introduction, whether it is syntax, validity or semantics.

Corrected bracket expression rule, VWBR, to account for an arbitrary number of arguments. (Also changed it to account for parenthesis expressions, see below.)

Added numerous "Informative Text" entries. The goal is that in the Standard every formal element (definition, syntax, validity, semantics) is followed by an Informative Text section.

Gave a name (VUES) to the Export Status principle.

Moved DPPO and DPOP (operands and positions in a call) from the agents chapter to the call chapter, changed codes accordingly (DUPO and DUOP).

Added Informative Text after Descendant Argument rule (VUDA) (after revision, see below).

Added Informative Text after definition of relative unfolded form (DDUR).

In new chapter "Typing-related properties" (will probably be renamed "Safe covariance"), renamed rules transferred from other chapters with chapter code 5:

- DVCC -> D5CC
- VUDA -> V5DA
- VUSC -> V5SC

Defined "tuple type" formally (as a synonym for a type declared as Tuple_type) and underlined it in formal text.

Defined "generic class" and "formal generic paramter" formally and underlined their occurrences in formal text.

Improvements to the clause on tuples (13):

- Added several "Informative text" sections, particularly at the beginning of 13.4
- Improved phrasing of "value sequences" (DJVS) (mathematically more precise, notion of set).

Introduced the parenthesis alias:

- addition to Informative Text in PFAI.
- New constructs Parenthesis and Parenthesis_expression.
- New case in assigner call rule (VBAC).
- New case in Target (SUCA).
- New case in Detachable_expression (SWEX).
- New case in the definition of subexpression: DWSE (9).
- Changed "Bracket Expression rule" (VWBR) to "Bracket and Parenthesis Expression
- rule" and changed its name to VWBP. Updated the Informative Text that follows it.
- Added exclusion of parenthesis case tothe equivalent dot form (DWDO)
- Updated alias validity rule (VFAV) and the following Informative Text.

Signed by JV - 2026/10/8

In DWPA (parenthesized form of an expression), removed "different" before "unary operators" (see email).

Systematized the notion of "actual argument" (it is defined in DRAA, underlined it in subsequent clauses).

Introduced tuple argument unfolding and tuple type unfolding in the Tuples chapter (new end of chapter 13, starting with

8.13.9, DJSR):

- Informative text to introduce the two mechanisms (after MJVA).
- Definition of single-tuple routine and tuple position (DJSR)
- Definition of tuple-argument-unfolded form (DJTA)
- Updated validity rule for calls (argument validity, VUAR) to refer to tuple-argument-unfolded form, as well as V5DA.

Updated corresponding semantic rule: MUGC

- Updated Descendant Argument rule (V5DA, renaming of VUDA) to refer to tuple-argument-unfolded form.
- Principles: PJTA, PJSA
- Mirror of all of this for tuple-type unfolding: DJTT, PJTT, DJST, DJTT, PJTT, updates to VGGD (validity of generic derivation) and MGSB (generic substitution)

Version 33, 17 May 2015

(From now on, using simpler version numbers, incrementing by 1 each time. The firles are correspondingly called 33.pdf and so on.)

Starting to implement decisions and "action items for the editor" of the last meeting.

Version 34, 2 June 2015

Principal change: addition of "Precursor" declarations for arguments and result type of a redeclared feature.

-Added the possibility of "Precursor" for argument and result declarations.

The relevant rules are the following (modifications also affect the Informative Texts that follow them:) SFDE, SRFE, VRFA, new clauses of VDPR, new rules DRUF, DFRT.

I chose to have a single validity rule (VDPR, pre-existing but now extended) for all cases of uses of Precursor.

Consequences: better definition of "formal argument" (see below); "unfolded forms" for Query_mark and Formal_argument_list; precise definition of "result type" (underlined all uses, replaced one instance of "return type".

Other changes:

- - Removed duplicate definition of "effecting" (in 367-2, 8.10.7 and 8.10.22, retained the first as DDEF), checkedthat no property was missed.
- - Changed "declaration for" to underlined style, to indicate that this notion is formally defined, in: DDDE (1) and DDTC (6). Also rephrased VDPR (1) to make it clear that it uses this notion.
- - Numbered the clauses of DTIN.
-
- - Replaced rules on the novelty of formal arguments by the recently introduced notion of "fresh identifier" (see comments on version 30.3, the rules D2SC and D2FI): VRFA (formal arguments), VRLV (local variables), VPIR (inline agents), VOIT (iteration, this one was already taken care of but I must have forgotten to record the change)

Signed by JV - 2026/10/8

-
- - Introduced a better definition for "formal argument" (DRFA), separating it from "actual argument" (DRAA, ex-DRAR), and underlined every occurrence of "formal argument" accordingly.

Version 30.4, 16 May 2015 (Changes performed during meeting 29 in Villebrumier)

Changes in this version were made with the rest of the participants in the TC49 meeting, Villebrumier, 11-13 May 2015]

Removed notion of Parenthesized_target, i.e. (|...|), which was not implemented anyway. replaced its uses by Parenthesized throughout.

Proposed (but did not include yet in the text) slight clarification of Semicolon Optionality

Added Bracket_expression to the possibilities for syntax of Target. As a result, t [i] [j] will officially be allowed.

Also added Precursor. As a result, Precursor [i] will be allowed.

(In progress) introduced distinction between "feature of a call" and "feature name of a call".

Replaced Conditional by Conditional_instruction throughout to make way for Conditional_expression

Introduced Conditional_expression (but did not define it yet)

Introduced the corresponding clauses for DWSE (subexpression) and "type of an expression" (DWET)

Removed the "convert" alias specification in the syntax of Alias (SFNA), keeping the semantics but removing the obligation (not currently implemented by compilers) to specify "convert".

Fixed VGMC (multiple constraints rule): removed second condition, but strengthened the first (the only remaining one)

In SGFG, for Formal_generic_name, replace the optional "detachable" by an optional Attachment_mark, for consistency with other types

DWET: changed the type of Result (clause 4) to be detachable.

Version 30.3, 8 May 2015

In DVOS, corrected mistake: it's not ot that reduces to rt, but the Basic_expression of ot.

Changed DVOS to define the scope of an Object_test (not object-test local). This is because of the general definition of scope.

Correspondingly simplified DVRS.

Introduced general definition of scope (D2SC).

As a consequence, simplified definition of "fresh identifier" (D2FI).

Version 30.1, 3 May 2015

Added syntax, semantics and definition codes (see email to list of this day)

SDUS (was 8.10.18): changed title to "Undefine subclause" rather than "Undefine clause".

Same for SMSE (8.16.6): Select clause -> subclause

In Semantics rules, removed the word "semantics" in the title, for example "Semantics: Check instruction semantics" becomes just "Semantics: Check instruction"

Turned DBAE (8.22.5, def. of "attaching an entity, attached entity") from "semantics" to "definition".

Version 30.2, 6 May 2015

Preparing to update the definition of "properly set" (DEPS, ex-8.19.16) in the form of a calculus asdecided in meeting 29.

Removed definition of "setter instruction" (8.19.5, DESE) since it will no longer be needed.

Version 30.3, 8 May 2015

In DEEP added notion of "start position" (along with "end position")

Also in DEEP, removed notion of "precedes" and "immediately precedes", no longer needed thanks to the new definition of "properly set" through a calculus. This is a considerable simplification. Also removed the corresponding Informative Text.

Rewrote DEPS (definition of "properly set") by defining the "|>" operator as discussed in the meeting 29, for critical cases.

In MEEN (1), replaced "a value" by "the value" (don't understand why it was "a value").

Updated DEVS (1) to account for the removal of the notion of "setter", see 30.2 above. ("Setting" remains.)

Removed VEVI in accordance with the meeting's decision. As a consequence:

- Changed DNEC (2); in fact this should already have been defined as a rule between expressions, not types, since we have the definition of an "attached expression"!
- For calls there was nothing to do since VUTA (2) already said that the target must be attached.

Version 27.1, 8 October 2011

(After meeting 27 held in Orinda, 25-27 August 2011)

CO-27|CHAPS

Introduced page breaks, for readability, before each of the major sections

(I think they are called "Partitions" in Ecma terminology): 1, 2, 3, ...

and also before each subclause of the specification section 8: 8.1,

8.2, 8.3 etc. If this is not compatible with Ecma rules I will remove

it before finalizing the document, but I found them useful to find

one's way through the document.

CO-27|23.22, p.125: Semantics: Once Routine Execution Semantics, Informative text

Fixed typo: An once -> A once

CO-27|2.21: Fresh identifier

Introduced notion of fresh identifier to avoid repetition (the second

time incompletely) between 8.24.23 (Object Test Rule) and 8.17.8

(Iteration rule). Updated these two rules.

Also used this notion in several places where the word

"fresh" was used without a definition: 8.10.4 (covariance-aware form),

8,10.28 (transposition), 8.10.29 (transposition).

CO-27|27 Changed "iteration" in the title to "iterators" (giving "Agents, iterators and introspection"),

to avoid any confusion with the Iteration construct (loops).

Signed by JV - 2026/10/8

CO-27|24 Changed title of section from "Eradicating void calls" to "Void safety".

-> No compelling reason, the simpler title just seems better.

AM-19|5.12, page 296: access to labeled tuple items

-> I now performed these changes. This required some additions to 8.13 (Tuples).

-> Removed 8.13.4 "Value sequences associated with a tuple type" as this notion was not used and it is better (I think) to used the unfolded form

-> Defined the unfolded form for unlabeled tuple types (it was defined only for labeled tuple types but used for unlabeled ones in [reference to be completed]

-> Added Informative Text to various parts of 8.23 (the chapter on calls) to remind the reader that t.l, for t a labeled tuple, is understood as a call thanks to the unfolded form. Affected places: 8.23.7, feature of a call (note was already there for the most part); 8.23.11, Export rule; 8.23.17, Target object; 8.23.28, Call result.

-> Explained things in detail in the Informative Texts.

CO-27|19.4 Changed title from "Object semantics" to "Copy and reference semantics"

CO-27|19.5 Split definition into two definitions and a semantic rule (reused name "Object semantics", see previous entry). There was just too much in this single definition covering the very nature of objects.

CO-27|30 Improved informational text at the top of the chapter on basic types by stating that the specification indicates what the possible values are for each basic type.

-> Improved style of all Semantics subclauses of 8.30, from "class X [e.g. INTEGER etc.] describes..." to "direct instances of class X are ...".

AM-27|5.4

-> Introduced a definition of "once per object", "once per thread" and "once per process") in 8.8.6.

-> Used them to replace longer phrasings in 8.7.16 8.23.20 and 8.23.21

-> Updated VFFD (5) as decided during the meeting.

Version 19, 9 November 2009

AM-19|5.2, pp. 282-283: Assigner commands

-> Not done (could no longer figure out what is needed).

-> To be discussed again

-> DONE IN VERSION 19.1, SEE BELOW

AM-19|5.3, page 283: VAPE and iline agents

-> Done as specified in minutes

Split a clause of 8.28.3 (definition of Subexpression) into two.

AM-19|5.6, page 284: Initialization of attached attributes

-> Not done. I couldn't find a precise enough specification of

Signed by JV - 2026/10/8

what needs to be done.

-> This matter is important and has to be revisited.

AM-19|5.7, page 286: Type of `type'in TYPE

-> Done as specified in minutes.

- Updated ELKS spec for TYPE (not in standard)
- Performed search for TYPE throughout text to look for other
- resulting changes. Found, in 8.27.4 (second bullet point) a
- misleading use of TYPE; replaced TYPE by T. This was the
- only consequence.

AM-19|5.8, page 287: Retry becomes a variable, rather than an instruction

-> Preparatory work

- Removed syntax rule for Retry
- Removed the Retry rule: 8.26.4. Unfortunately this causes
- a renumbering of remaining clauses in 8.26.

-> Done as specified in minutes.

- Updated exception semantics as decided. Because of the renumbering the discussion in AM-5.8 now applies to 8.26.7 (rather than 8.26.8).
- In associated Informative Text, fixed typo that
- hadn't been noticed: "step 1" should have been "step 4".
- Updated associated Informative Text to reflect
- new form of Retry.
- Added, as requested, the comment that Retry
- is not permitted in postconditions. Note, however,
- that this requires a new clause in VEEN as
- described below.

This also implied updating the notion of local variable:

- Updated 8.6.9 (definition of "local variable"); it now
- has three cases.
- Updated preceding Informative Text. (Also removed
- an incorrect mnetion of "the function" which should
- have been "the function or attribute".)
- Added a clause, now 5, to VEEN, to mention where
- Retry can appear. (I also tried updating the next
- clause; it would work but is a bit awkward as the rule
- would seem to allow Retry to appear in a Local_declarations,
- which is OK as this is disallowed by other rules.)
- Updated terminology to refer to both Result and Retry,
- throughout, as "predefined local variables", rather
- than "predefined entities". Current remains mentioned
- as a "predefined entity".

In addition:

- Removed the Retry instruction from 8.8.10.

- Updated Informative Text after 8.26.9 (previously
- 8.26.10); also improved phrasing ("is *f* itself").
- Changed the definition of exception-correct, now 8.26.4. The definition, however, should be revised in light of Martin Nordio's semantics.
- Added a case to type of expressions (8.28.17, newcase 5) and accordingly updated the following case.

AM-19|5.9, page 289: Relaxing inline agent validity rule (NOT!)

-> Updated Informative Text after 8.27.11

-> Please check my added paragraph!

AM-19|5.10, page 289-290: Call-agent equivalent

-> Done as specified in minutes.

AM-19|5.11, pages 290 passim: catcalls

-> No action item for the editor, but this issue must be revisited

AM-19|5.12, page 296: access to labeled tuple items

-> Not done yet, I plan to do it in the next few days

-> Note 8 Oct 2011 (not really "the next few days"): now done, see comments in version 27.1.

AM-19|5.13, page 296: creation availability of root creation procedures

-> Done as specified in minutes

-> Please check Informative Text for 8.3.10.

AM-19|5.14, page 296: conversion rules

-> I see no clear resolution for this issue

-> Did nothing

-> We should revisit the matter!

Version 19.1, 9 November 2009

AM-19|5.2, pp. 282-283: Assigner commands

-> Done, please check change

OBJECT TEST

- Changed to new syntax, systematically through text.
- Updated validity and semantics.

End

PART A: Preliminaries

Signed by JV - 2026/10/8

Signed by JV - 2026/10/8

Standard
ECMA-367

3rd Edition - January 2025

Eiffel:
Analysis, Design and Programming Language

Ecma International Rue du Rhône 114 CH-1204 Geneva T/F: +41 22 849 6000/01 www.ecma-international.org

Signed by JV - 2026/10/8
30/10/25 18:11

Signed by JV - 2026/10/8

Brief history

Eiffel was originally designed, as a method of software construction and a notation to support that method, in 1985. The first implementation, from Eiffel Software (then Interactive Software Engineering Inc.), was commercially released in 1986. The principal designer of the first versions of the language was Bertrand Meyer. Other people closely involved with the original definition included Jean-Marc Nerson. The language was originally described in Eiffel Software technical documents that were expanded to yield Meyer's book *Eiffel: The Language* in 1990-1991. The two editions of *Object-Oriented Software Construction* (1988 and 1997) also served to describe the concepts. (For bibliographical references on the documents cited see 3.6.) As usage of Eiffel grew, other Eiffel implementations appeared, including Eiffel/S and Visual Eiffel from Object Tools, Germany, EiffelStudio and Eiffel Envision from Eiffel Software, and SmartEiffel from LORIA, France.

Eiffel today is used throughout the world for industrial applications in banking and finance, defense and aerospace, health care, networking and telecommunications, computer-aided design, game programming, and many other application areas. Eiffel is particularly suited for mission-critical developments in which programmer productivity and product quality are essential. In addition, Eiffel is a popular medium for teaching programming and software engineering in universities.

In 2002 Ecma International formed Technical Group 4 (Eiffel) of Technical Committee 39 (Programming and Scripting Languages), known as TC39-TG4. The *Eiffel: Analysis, Design and Programming Language* Standard provides a precise definition of the language and ensures interoperability between implementations. The first of these benefits is of particular interest to implementors of Eiffel compilers and environments, who can rely on it as the reference on which to base their work; the second, to Eiffel users, for whom the Standard delivers a guarantee of compatibility between the products of different providers and of trust in the future of Eiffel.

The starting point for this Standard was material from the original and revised versions of the book *Standard Eiffel* (latest revision of *Eiffel: The Language*). The process of preparing the standard relied on automated tools to keep the corresponding elements identical in the Standard and the forthcoming revision of the book.

The First Edition was completed in April of 2005 and approved by the General Assembly of ECMA in June of 2005. As a result, authors of major Eiffel compilers and tools have committed to supporting the language as described in the Standard; compilers are quickly progressing towards full compliance.

The present Second Edition introduces no major change of substance but clarifies and corrects a number of points arising from the experience of using the First Edition and from the feedback of compiler writers and the user community. It was completed in April, 2006 and submitted both to ECMA and to the International Standards Organization (ISO), which has approved it as the ISO standard 25436. The text of this Second Edition is identical to the text of the ISO standard.

To produce this Standard, TC39-TG4 conducted: 18 face-to- face meetings (15 up to the First Edition), most of them over three full days; numerous phone meetings; and extensive technical correspondence (about 1500 email messages by June 2006).

The members of the committee have been: Karine Arnout (Axa Rosenberg, Orinda, previously ETH, Zurich); Éric Bezault (Axa Rosenberg, Orinda); Paul Cohen (Generic, Stockholm), Dominique Colnet (LORIA, Nancy); Mark Howard (Axa Rosenberg, Orinda); Alexander Kogtenkov (Eiffel Software, Moscow); Bertrand Meyer (Eiffel Software, Santa Barbara, and ETH, Zurich); Christine Mingins (Monash University, Melbourne); Roger Osmond (EMC, Boston); Emmanuel Stapf (Eiffel Software, Santa Barbara); Kim Waldén (Generic, Stockholm).

Observers having attended one or more of the meetings include Cyril Adrian (LORIA), Volkan Arslan (ETH), Paul Crismer (Groupe S, Brussels), Jocelyn Fiat (Eiffel Software, France), Randy John (Axa Rosenberg), Ian King (Eiffel Software), Andreas Leitner (ETH), Philippe Ribet (LORIA), Julian Rogers (Eiffel Software), Bernd Schoeller (ETH), David Schwartz (Axa Rosenberg), Zoran Simic (Axa Rosenberg), Raphael Simon (Eiffel Software), Olivier Zendra (LORIA). The committee acknowledges the contributions of many people including David Hollenberg, Marcel Satchell, Richard O'Keefe and numerous others listed in the acknowledgments of the book *Standard Eiffel*.

The editor of the standard is Bertrand Meyer. Emmanuel Stapf is the convener of TG4 (succeeding Christine Mingins, 2002-2003) and was its secretary in 2005. Karine Arnout has been its secretary at all other times.

Signed by JV - 2026/10/8

The final version of the First Edition was prepared by Éric Bezault, Mark Howard, Alexander Kogtenkov, Bertrand Meyer and Emmanuel Stapf, the final version of the Second Edition by Bertrand Meyer and Emmanuel Stapf.

This Ecma Standard has been adopted by the General Assembly of xx 2025.

Signed by JV - 2026/10/8

Table of contents

Record of Changes towards the completion of the Standard iii

PART A: PRELIMINARIES

Brief history 7

1 Scope 25
1.1 Overview 25
1.2 “The Standard” 25
1.3 Aspects covered 25
1.4 Aspects not covered 25
2 Conformance 26
2.1 Definition 26
2.2 Compatibility and non-default options 26
2.3 Departure from the Standard 26
3 Normative references 27
3.1 Earlier Eiffel language specifications 27
3.2 Eiffel Kernel Library 27
3.3 Floating point number representation 27
3.4 Character set: Unicode 27
3.5 Character set: ASCII 27
3.6 Phonetic alphabet 27
4 Definitions 28
5 Notational conventions 29
5.1 Normative elements 29
5.2 Non-normative elements 29
5.3 Kinds of clause 29
5.4 Normative elements 29
5.5 Rules on definitions 30
5.6 Use of defined terms 30
5.7 Unfolded forms 30
5.8 Language description 31
5.9 Validity: “if and only if” rules 31
6 Acronyms and abbreviations 32
6.1 Name of the language 32
6.2 Pronunciation 32

Signed by JV - 2026/10/8

PART B: INTRODUCTION TO EIFFEL

7 An informal introduction to Eiffel **35**
7.1 Design principles **35**
7.2 Object-oriented design **35**
7.3 Classes **36**
7.4 Types **39**
7.5 Assertions **40**
7.6 Exceptions **43**
7.7 Genericity **44**
7.8 Inheritance **45**
7.9 Polymorphism and dynamic binding **47**
7.10 Combining genericity and inheritance **49**
7.11 Deferred classes **50**
7.12 Tuples and agents **51**
7.13 Type- and void-safety **52**
7.14 Putting a system together **53**

PART C: GENERAL CONVENTIONS

8 Language specification **56**
8.1 General organization **57**
8.2 Language description conventions: syntax, validity, semantics **58**
8.2.1 Principle: Subclause classification and codes 58
8.2.2 Principle: Hyperlinking 59
8.2.3 Definition: Syntax, BNF-E 59
8.2.4 Definition: Component, construct, specimen 60
8.2.5 Principle: Construct Specimen convention 60
8.2.6 Principle: Construct Name convention 61
8.2.7 Definition: Terminal, non-terminal, token 61
8.2.8 Definition: Production 63
8.2.9 Definition: Kinds of production 63
8.2.10 Definition: Aggregate production 63
8.2.11 Definition: Choice production 64
8.2.12 Definition: Repetition production, separator 64
8.2.13 Principle: Basic Syntax Description rule 65
8.2.14 Definition: Non-production Syntax rule 65
8.2.15 Principle: Textual Conventions 66
8.2.16 Definition: Validity constraint 67
8.2.17 Definition: Valid 67
8.2.18 Validity: General Validity rule 68
8.2.19 Definition: Semantics 68
8.3 Conventions for Eiffel systems **70**

Signed by JV - 2026/10/8

8.3.1 Definition: Upper name, lower name 70
8.3.2 Semantics: Case Insensitivity principle 70
8.3.3 Definition: Letter case conventions 71
8.3.4 Definition: Scope of an identifier 71
8.3.5 Definition: Fresh identifier 72
8.3.6 Syntax (non-production): Semicolon Optionality rule 73

PART D: SYSTEM STRUCTURING MECHANISMS

8.4 Elementary concepts 76
8.4.1 Definition: Systems and their executions 76
8.4.2 Definition: Execution terminology 76
8.5 The architecture of Eiffel software 77
8.5.1 Definition: Cluster, subcluster, contains directly, contains 77
8.5.2 Definition: Terminal cluster, internal cluster 77
8.5.3 Definition: Universe 78
8.5.4 Syntax : Class names 78
8.5.5 Validity: Class Identification rule 78
8.5.6 Validity: Class Name rule 79
8.5.7 Semantics: Class name semantics 79
8.5.8 Definition: System, root type name, root procedure name 79
8.5.9 Definition: Type dependency 80
8.5.10 Validity: Root Type rule (and definition: root type) 80
8.5.11 Definition: Root class 81
8.5.12 Validity: Root Procedure rule (and definition: root procedure) 81
8.5.13 Semantics: System execution (and definition: root object) 81
8.6 Classes 82
8.6.1 Definition: Current class 82
8.6.2 Syntax : Class declarations 82
8.6.3 Syntax : Notes 83
8.6.4 Semantics: Notes semantics 84
8.6.5 Syntax : Class headers 84
8.6.6 Validity: Class Header rule 85
8.6.7 Semantics: Class Header semantics 85
8.6.8 Definition: Expanded, frozen, deferred, effective class 85
8.6.9 Syntax : Obsolete marks 86
8.6.10 Semantics: Obsolete semantics 86
8.7 Clients and exports 87
8.7.1 Definition: Client relation between classes and types 88
8.7.2 Definition: Client relation between classes 88
8.7.3 Definition: Supplier 88
8.7.4 Definition: Simple client 89
8.7.5 Definition: Expanded client 90
8.7.6 Definition: Generic client, generic supplier 91

Signed by JV - 2026/10/8

8.7.7 Definition: Indirect client 91
8.7.8 Syntax : Clients 92
8.7.9 Definition: Client set of a Clients part or a Feature_clause 93
8.7.10 Syntax : Export adaptation 93
8.7.11 Validity: Export List rule 94
8.7.12 Definition: Client set of a feature 94
8.7.13 Definition: Available for call, available 95
8.7.14 Definition: Exported, selectively available, secret 96
8.7.15 Definition: Secret, public 96
8.7.16 Definition: Incremental contract view, short form 98
8.7.17 Definition: Contract view, flat-short form 98
8.8 Features 99
8.8.1 Definition: Feature of a class 100
8.8.2 Definition: Inherited, immediate, origin, introduce, introduced 100
8.8.3 Syntax : Feature parts 101
8.8.4 Definition: Overview of feature categories 101
8.8.5 Syntax : Feature declarations 102
8.8.6 Syntax : New feature lists 102
8.8.7 Definition: Synonym 103
8.8.8 Definition: Unfolded form of a possibly multiple declaration 103
8.8.9 Definition: Invariant variable 104
8.8.10 Syntax : Feature names and aliases 104
8.8.11 Syntax (non-production): Alias Syntax rule 105
8.8.12 Validity: Alias Validity rule 105
8.8.13 Definition: Operator feature, brackets or parenthesis feature, identifier-only 106
8.8.14 Definition: Identifier of a feature name 107
8.8.15 Principle: Feature Identifier principle 107
8.8.16 Definition: Same feature name, same operator, same alias 107
8.8.17 Syntax : Operators 108
8.8.18 Principle: Feature principle 108
8.8.19 Syntax : Declaration bodies 109
8.8.20 Validity: Declaration Body rule 109
8.8.21 Syntax : Query marks 110
8.8.22 Definition: Result type of a query 110
8.8.23 Definition: Signature, argument signature of a feature 111
8.8.24 Syntax : Assigner marks 112
8.8.25 Validity: Assigner Command rule 113
8.8.26 Syntax : Feature definitions 113
8.8.27 Validity: Feature Definition rule 114
8.8.28 Definition: Attribute 115
8.8.29 Definition: Variable attribute 115
8.8.30 Definition: Constant attribute 116
8.8.31 Syntax : Defined attributes 117

Signed by JV - 2026/10/8

8.8.32 Definition: Routine, function, procedure 117
8.8.33 Definition: Command, query 118
8.8.34 Validity: Constant Attribute Validity rule 118
8.8.35 Validity: Feature Declaration rule 119
8.9 Routines 121
8.9.1 Definition: Actual argument, actual 121
8.9.2 Syntax : Formal argument and entity declarations 121
8.9.3 Definition: Unfolded form of a Formal_arguments 122
8.9.4 Definition: Formal argument, formal 122
8.9.5 Validity: Formal Argument rule 122
8.9.6 Validity: Entity Declaration rule 123
8.9.7 Syntax : Routine bodies 123
8.9.8 Definition: Once routine and their varieties, once keys 123
8.9.9 Syntax : Local variable declarations 123
8.9.10 Validity: Local Variable rule 123
8.9.11 Definition: Local variable, declared local variable 124
8.9.12 Syntax : Instructions 124

PART E: CONTRACTS

8.10 Correctness and contracts 128
8.10.1 Syntax : Assertions 128
8.10.2 Syntax (non-production): Assertion Syntax rule 129
8.10.3 Definition: Contract element 129
8.10.4 Definition: Precondition, postcondition of a feature; invariant of a class 130
8.10.5 Definition: Loop invariant, loop variant 130
8.10.6 Definition: Contract, subcontract of a feature or a class 130
8.10.7 Syntax : Check instructions 130
8.10.8 Syntax : Variants 131
8.10.9 Validity: Variant Expression rule 131
8.10.10 Syntax : “Old” postcondition expressions 131
8.10.11 Validity: Old Expression rule 131
8.10.12 Semantics: Old expression semantics, associated variable, exception marker 131
8.10.13 Definition: Local unfolded form of an assertion 132
8.10.14 Definition: Minimal assertion 133
8.10.15 Definition: Local assertions 133
8.10.16 Validity: Assertion Implication rule 133
8.10.17 Validity: Precondition Export rule 133
8.10.18 Definition: Availability of an assertion clause 134
8.10.19 Definition: Unfolded form of an assertion 134
8.10.20 Definition: Assertion extensions 135
8.10.21 Definition: Covariance-aware form of an assertion extension 135
8.10.22 Definition: Combined precondition, combined postcondition 135

Signed by JV - 2026/10/8

8.10.23 Definition: Precondition-free 136
8.10.24 Definition: Hoare triple notation (total correctness) 136
8.10.25 Definition: Implication correctness 137
8.10.26 Definition: Check-correct 137
8.10.27 Semantics: Class consistency 137
8.10.28 Definition: Loop-correct 137
8.10.29 Definition: Correctness (class) 138
8.10.30 Semantics: Associated variable semantics 138
8.10.31 Semantics: Evaluation of an assertion 138
8.10.32 Semantics: Assertion monitoring 139
8.10.33 Semantics: Assertion violation 139
8.10.34 Semantics: Assertion semantics 139
8.10.35 Semantics: Check instruction semantics 139
8.10.36 Semantics: Assertion monitoring levels 139

PART F: INHERITANCE

8.11 The inheritance relation 143
8.11.1 Syntax : Inheritance parts 143
8.11.2 Syntax (non-production): Feature adaptation 143
8.11.3 Definition: Parent part for a type, for a class 143
8.11.4 Validity: Class *ANY* rule 144
8.11.5 Validity: Universal Conformance principle 144
8.11.6 Definition: Unfolded inheritance part of a class 144
8.11.7 Definition: Multiple, single inheritance 144
8.11.8 Definition: Inherit, heir, parent 145
8.11.9 Definition: Conforming, non-conforming parent 145
8.11.10 Definition: Ancestor types of a type, of a class 145
8.11.11 Definition: Ancestor, descendant 145
8.11.12 Definition: Proper ancestor, proper descendant 146
8.11.13 Validity: Parent rule 146
8.11.14 Syntax : Rename clauses 147
8.11.15 Validity: Rename Clause rule 147
8.11.16 Definition: Final name, extended final name, final name set, renaming 147
8.11.17 Semantics: Renaming principle 148
8.11.18 Definition: Inherited name 148
8.11.19 Definition: Declaration for a feature 148
8.12 Feature adaptation 149
8.12.1 Definition: Redeclare, redeclaration 149
8.12.2 Definition: Inherited as effective, inherited as deferred 149
8.12.3 Definition: Effect, effecting 149
8.12.4 Definition: Redefine, redefinition 149
8.12.5 Definition: Name clash 150
8.12.6 Syntax : Precursor call, precursor signature 150

Signed by JV - 2026/10/8

8.12.7 Validity: Precursor Signature rule and definition: specified parent 150
8.12.8 Definition: Relative unfolded form of a Precursor_call 151
8.12.9 Validity: Precursor Call rule and definitions: precursof feature, applicable call 151
8.12.10 Semantics: Precursor call semantics 152
8.12.11 Syntax : Redefinition 152
8.12.12 Validity: Redefine Subclause rule 152
8.12.13 Semantics: Redefinition semantics 152
8.12.14 Syntax : Undefine subclauses 152
8.12.15 Definition: Undefine, undefinition 152
8.12.16 Validity: Undefine Subclause rule 152
8.12.17 Semantics: Undefinition semantics 153
8.12.18 Definition: Effective, deferred feature 153
8.12.19 Principle: Deferred Class property 153
8.12.20 Principle: Effective Class property 153
8.12.21 Definition: Origin, seed 154
8.12.22 Validity: Redeclaration rule 154
8.12.23 Definition: Precursor (joined features) 155
8.12.24 Definition: Direct transposition 155
8.12.25 Definition: Transposition to a class or type 156
8.12.26 Definition: Unfolded redeclaration 156
8.12.27 Validity: Join rule 157
8.12.28 Semantics: Join Semantics rule 158
8.13 Repeated inheritance 159
8.13.1 Definition: Repeated inheritance, ancestor, descendant 159
8.13.2 Semantics: Repeated Inheritance rule 159
8.13.3 Definition: Sharing, replication 159
8.13.4 Validity: Call Sharing rule 159
8.13.5 Semantics: Replication Semantics rule 159
8.13.6 Syntax : Select subclause 160
8.13.7 Validity: Select Subclause rule 160
8.13.8 Definition: Version 160
8.13.9 Definition: Multiple versions 160
8.13.10 Validity: Repeated Inheritance Consistency constraint 160
8.13.11 Definition: Dynamic binding version 160
8.13.12 Definition: Inherited features 161
8.13.13 Semantics: Join-Sharing Reconciliation rule 161
8.13.14 Definition: Precursor 161
8.13.15 Validity: Feature Name rule 162
8.13.16 Validity: Name Clash rule 162

PART G: THE TYPE SYSTEM

8.14 Types 165

Signed by JV - 2026/10/8

8.14.1 Syntax : Types 165
8.14.2 Semantics: Direct instances and values of a type 165
8.14.3 Semantics: Instance of a type 165
8.14.4 Semantics: Instance principle 166
8.14.5 Definition: Instance, direct instance of a class 166
8.14.6 Principle: Base Class principle 166
8.14.7 Principle: Base Type principle 166
8.14.8 Validity: Class Type rule 167
8.14.9 Semantics: Type Semantics rule 167
8.14.10 Definition: Base class and base type of an expression or entity 167
8.14.11 Definition: Features of a type 167
8.14.12 Semantics: Non-generic class type semantics 168
8.14.13 Definition: Expanded type, reference type 168
8.14.14 Definition: Current type 168
8.14.15 Definition: Features of a type 168
8.14.16 Definition: Basic type 169
8.14.17 Definition: Anchor, anchored type, anchored entity 169
8.14.18 Definition: Anchor set; cyclic anchor 169
8.14.19 Definition: Types and classes involved in a type 169
8.14.20 Definition: Deanchored form of a type 170
8.14.21 Validity: Anchored Type rule 170
8.14.22 Definition: Attached, detachable 171
8.14.23 Semantics: Attached type semantics 171
8.14.24 Definition: Stand-alone type 171
8.15 Genericity 173
8.15.1 Syntax : Actual generic parameters 173
8.15.2 Definition: Actual generic parameter 173
8.15.3 Syntax : Formal generic parameters 174
8.15.4 Definition: Generic class, formal generic parameter, formal parameter 174
8.15.5 Validity: Formal Generic rule 174
8.15.6 Definition: Constrained, unconstrained 175
8.15.7 Definition: Generic derivation, non-generic type 175
8.15.8 Definition: Self-initializing formal generic parameter 175
8.15.9 Definition: Constraint, constraining types of a Formal_generic 176
8.15.10 Syntax : Generic constraints 176
8.15.11 Validity: Generic Constraint rule 176
8.15.12 Definition: Constraining creators 177
8.15.13 Validity: Generic Derivation rule 177
8.15.14 Definition: Generic-creation-ready type 178
8.15.15 Semantics: Generically-derived Class_type semantics 178
8.15.16 Definition: Base type of a single-constrained formal generic 179
8.15.17 Definition: Base type of an unconstrained formal generic 179
8.15.18 Definition: Reference or expanded status of a formal generic 179

Signed by JV - 2026/10/8

8.15.19 Semantics: Generic substitution 179
8.15.20 Principle: Generic Type Adaptation rule 179
8.15.21 Definition: Generically constrained feature name 180
8.15.22 Validity: Multiple Constraints rule 180
8.15.23 Definition: Base type of a multi-constraint formal generic type 180
8.16 Tuples 181
8.16.1 Syntax : Tuple types 181
8.16.2 Syntax : Manifest tuples 181
8.16.3 Definition: Tuple type, type sequence 182
8.16.4 Definition: Value sequences associated with a tuple type 182
8.16.5 Definition: Labeled tuple type, label 182
8.16.6 Validity: Tuple Type rule 183
8.16.7 Definition: Unfolded form of a tuple type 183
8.16.8 Semantics: Tuple type values 184
8.16.9 Definition: Single-tuple routine, tuple position 185
8.16.10 Definition: Tuple-argument-unfolded form of a call 185
8.16.11 Principle: Tuple-Argument-Unfolding property 186
8.16.12 Principle: Tuple-Argument-Unfolding size 186
8.16.13 Definition: Single-tuple class, tuple constraint position 187
8.16.14 Definition: Tuple-type-unfolded form of a type 187
8.16.15 Principle: Tuple-Type-Unfolding property 188
8.16.16 Principle: Tuple-Type-Unfolding Size property 188
8.17 Conformance 189
8.17.1 Definition: Compatibility between types 189
8.17.2 Definition: Compatibility between expressions 189
8.17.3 Definition: Expression conformance 189
8.17.4 Validity: Signature conformance 190
8.17.5 Definition: Covariant argument 190
8.17.6 Validity: General conformance 191
8.17.7 Definition: Conformance path 191
8.17.8 Validity: Direct conformance: reference types 191
8.17.9 Validity: Direct conformance: formal generic 192
8.17.10 Validity: Direct conformance: expanded types 192
8.17.11 Validity: Direct conformance: tuple types 192
8.18 Convertibility 194
8.18.1 Definition: Conversion procedure, conversion type 194
8.18.2 Definition: Conversion query, conversion feature 194
8.18.3 Validity: Conversion principle 194
8.18.4 Validity: Conversion Asymmetry principle 194
8.18.5 Validity: Conversion Non-Transitivity principle 194
8.18.6 Syntax : Converter clauses 194
8.18.7 Validity: Conversion Procedure rule 194
8.18.8 Validity: Conversion Query rule 195

Signed by JV - 2026/10/8

8.18.9 Definition: Converting to a class 195
8.18.10 Definition: Converting to and from a type 195
8.18.11 Definition: Converting “through” 196
8.18.12 Semantics: Conversion semantics 196
8.18.13 Definition: Explicit conversion 196
8.18.14 Validity: Expression convertibility 197
8.18.15 Definition: Statically satisfied precondition 197
8.18.16 Validity: Precondition-free routine 197

PART H: COMPUTATIONAL MECHANISMS

8.19 Feature call 200
8.19.1 Validity: Call Use rule 200
8.19.2 Syntax : Feature calls 200
8.19.3 Definition: Qualified, unqualified call 201
8.19.4 Definition: Targeted, untargeted call 201
8.19.5 Definition: Target of a call 201
8.19.6 Syntax : Actual arguments 202
8.19.7 Definition: Operands of a call 202
8.19.8 Definition: Operand position 202
8.19.9 Definition: Target type of a call 202
8.19.10 Definition: Feature name of a call 203
8.19.11 Validity: Export Status principle 203
8.19.12 Definition: Static feature of a call 203
8.19.13 Definition: Feature of a call 203
8.19.14 Validity: Argument rule 204
8.19.15 Validity: Target rule 204
8.19.16 Definition: Imported form of a Non_object_call 205
8.19.17 Validity: Non-Object Call rule 205
8.19.18 Semantics: Non-Object Call semantics 205
8.19.19 Validity: Export rule 206
8.19.20 Validity: Class-Level Call rule 206
8.19.21 Definition: Target object 206
8.19.22 Definition: Dynamic feature of a call 206
8.19.23 Principle: Feature Version theorem 207
8.19.24 Definition: Void-Unsafe 207
8.19.25 Semantics: Failed target evaluation of a void-unsafe system 207
8.19.26 Definition: Freshness of a once routine call 207
8.19.27 Definition: Latest applicable target and result of a non-fresh call 208
8.19.28 Semantics: Once Routine Execution semantics 208
8.19.29 Semantics: Current object, current routine 208
8.19.30 Semantics: Current semantics 209
8.19.31 Semantics: Non-Once Routine Execution Semantics 209
8.19.32 Semantics: General Call semantics 209

Signed by JV - 2026/10/8

8.19.33 Definition: Expression-call type 209
8.19.34 Semantics: Call result 210
8.19.35 Semantics: Value of a call expression 210
8.20 Agents, iterators and introspection 211
8.20.1 Definition: Construction time, call time 211
8.20.2 Syntax : Agents 211
8.20.3 Syntax (non-production): syntactical forms for a call agent 212
8.20.4 Definition: Target type of a call agent 212
8.20.5 Validity: Call Agent rule 212
8.20.6 Definition: Associated feature of an inline agent 212
8.20.7 Validity: Inline Agent rule 213
8.20.8 Validity: Inline Agent requirements 213
8.20.9 Definition: Call-agent equivalent of an inline agent 213
8.20.10 Semantics: Semantics of inline agents 213
8.20.11 Semantics: Use of **Result** in an inline function agent 213
8.20.12 Definition: Operands of an agent 213
8.20.13 Definition: Open and closed operands of an agent 214
8.20.14 Definition: Open and closed operand positions 214
8.20.15 Definition: Type of an agent expression 214
8.20.16 Semantics: Agent Expression semantics 214
8.20.17 Semantics: Effect of executing *call* on an agent 214
8.21 Expressions 215
8.21.1 Syntax : Expressions 215
8.21.2 Definition: Subexpression 216
8.21.3 Definition: Operands 217
8.21.4 Semantics: Parenthesized semantics 217
8.21.5 Syntax : Operator expressions 217
8.21.6 Syntax (non-production): Operator precedence levels 217
8.21.7 Definition: Parenthesized form of an expression 218
8.21.8 Definition: Target-converted form of a binary expression 218
8.21.9 Validity: Operator Expression rule 218
8.21.10 Syntax : Bracket expressions 219
8.21.11 Syntax : Paren expressions 219
8.21.12 Validity: Bracket and Paren Expression rule 219
8.21.13 Syntax : Conditional expression 219
8.21.14 Definition: Equivalent Dot Form of an expression 220
8.21.15 Validity: Boolean Expression rule 221
8.21.16 Validity: Identifier rule 221
8.21.17 Definition: Expression Type rule 221
8.21.18 Semantics: Expression semantics (strict case) 222
8.21.19 Definition: Semistrict operators 222
8.21.20 Semantics: Operator expression semantics (semistrict case) 222
8.22 Control structures 224

Signed by JV - 2026/10/8

8.22.1 Semantics: Compound (non-exception) semantics 224
8.22.2 Syntax : Conditional instruction 224
8.22.3 Definition: Secondary part 225
8.22.4 Definition: Prevailing immediately 225
8.22.5 Semantics: Conditional instruction semantics 225
8.22.6 Definition: Inspect expression 225
8.22.7 Syntax : Multi-branch instructions 225
8.22.8 Definition: Interval 226
8.22.9 Definition: Unfolded form of a multi-branch 226
8.22.10 Definition: Unfolded form of an interval 226
8.22.11 Validity: Interval rule 227
8.22.12 Definition: Inspect values of a multi-branch 227
8.22.13 Validity: Multi-branch rule 227
8.22.14 Semantics: Matching branch 227
8.22.15 Semantics: Multi-Branch semantics 228
8.22.16 Syntax : Loops 228
8.22.17 Validity: Loop rule 228
8.22.18 Validity: Iteration rule 229
8.22.19 Definition: Unfolded form of a loop 229
8.22.20 Semantics: Loop semantics 230
8.22.21 Syntax : Debug instructions 230
8.22.22 Semantics: Debug semantics 230
8.23 Creating objects 231
8.23.1 Semantics: Creation principle 231
8.23.2 Definition: Creation operation 231
8.23.3 Validity: Creation Precondition rule 231
8.23.4 Syntax : Creators parts 232
8.23.5 Definition: Unfolded Creators part of a class 232
8.23.6 Validity: Creation Clause rule 232
8.23.7 Definition: Creation procedures of a class 233
8.23.8 Principle: Creation Procedure property 233
8.23.9 Definition: Creation procedures of a type 233
8.23.10 Definition: Available for creation; general creation procedure 233
8.23.11 Syntax : Creation instructions 234
8.23.12 Definition: Creation target, creation type of a creation instruction 234
8.23.13 Principle: Creation Type theorem 234
8.23.14 Definition: Unfolded form of a creation instruction 234
8.23.15 Definition: Static and dynamic features of a creation call 234
8.23.16 Validity: Creation Instruction rule 235
8.23.17 Validity: Creation Instruction properties 235
8.23.18 Semantics: Creation instruction Semantics 236
8.23.19 Syntax : Creation expressions 237
8.23.20 Definition: Creation type, unfolded form of a creation expression 237

Signed by JV - 2026/10/8

8.23.21 Validity: Creation Expression rule 237
8.23.22 Validity: Creation Expression Properties 237
8.23.23 Semantics: Creation Expression Semantics 238
8.23.24 Definition: Garbage Collection, not enough memory available 238
8.24 Attaching values to entities 239
8.24.1 Definition: Reattachment, source, target 239
8.24.2 Syntax : Assignments 239
8.24.3 Validity: Assignment rule 239
8.24.4 Semantics: Reattachment principle 240
8.24.5 Definition: Attaching an entity, attached entity 240
8.24.6 Semantics: Reattachment semantics 240
8.24.7 Semantics: Assignment semantics 241
8.24.8 Definition: Dynamic type 241
8.24.9 Definition: Polymorphic; polymorphism; dynamic type and class sets 241
8.24.10 Syntax : Assigner calls 241
8.24.11 Validity: Assigner Call rule 242
8.24.12 Semantics: Assigner Call semantics 242
8.25 Objects, values and entities 243
8.25.1 Semantics: Object, type, generating type and class of an object; generator 243
8.25.2 Definition: Value 243
8.25.3 Definition: Reference, void, attached, attached to 243
8.25.4 Semantics: Object principle 244
8.25.5 Definition: Copy and reference semantics 244
8.25.6 Definition: Non-basic class, non-basic type 244
8.25.7 Definition: Field 244
8.25.8 Semantics: Object semantics 244
8.25.9 Definition: Subobject, composite object 245
8.25.10 Definition: Entity, variable, read-only 245
8.25.11 Syntax : Entities and variables 245
8.25.12 Validity: Entity rule 245
8.25.13 Validity: Variable rule 246
8.25.14 Definition: Self-initializing type 246
8.25.15 Semantics: Default Initialization rule 247
8.25.16 Definition: Self-initializing variable 247
8.25.17 Definition: Evaluation position 248
8.25.18 Definition: Properly set variable 248
8.25.19 Definition: Variable setting and variable value 249
8.25.20 Definition: Execution context 249
8.25.21 Semantics: Variable semantics 249
8.25.22 Semantics: Entity Semantics rule 250
8.26 Comparing and duplicating objects 251
8.27.1 Syntax : Equality expressions 252

Signed by JV - 2026/10/8

8.27.2 Semantics: Equality Expression semantics 252
8.27.3 Semantics: Inequality Expression semantics 252
8.30 Void safety 254
8.30.1 Syntax : Object test 254
8.30.2 Definition: Object-Test Local 254
8.30.3 Validity: Object Test rule 255
8.30.4 Definition: Conjunctive, disjunctive, implicative; term, semistrict term 255
8.30.5 Definition: Expression reduction 255
8.30.6 Definition: Scope of an Object Test 255
8.30.7 Semantics: Object Test semantics 256
8.30.8 Semantics: Object-Test Local semantics 256
8.30.9 Definition: Read-only void test 256
8.30.10 Definition: Scope of a read-only void test 256
8.30.11 Definition: Emergent attribute 257
8.30.12 Definition: Certified Attachment Pattern 257
8.30.13 Definition: Attached expression 257
8.31 Typing-related properties 258
8.31.1 Definition: Catcall 258
8.31.2 Validity: Descendant Argument rule 258
8.31.3 Validity: System-level Call rule 258
8.32 Concurrency: the SCOOP mechanism 259
8.33 Exception handling 260
8.33.1 Definition: Failure, exception, trigger 260
8.33.2 Syntax : Rescue clauses 260
8.33.3 Validity: Rescue Clause rule 260
8.33.4 Definition: Exception-correct 261
8.33.5 Semantics: Default rescue original semantics 261
8.33.6 Definition: Rescue block 261
8.33.7 Semantics: Exception semantics 261
8.33.8 Definition: Type of an exception 262
8.33.9 Semantics: Exception Cases 262
8.33.10 Semantics: Exception Properties 263
8.33.11 Definition: Ignoring, continuing an exception 263

PART I: BASIC PROGRAM COMPONENTS

8.34 Constants 266
8.34.1 Syntax : Constants 266
8.34.2 Validity: Constant Attribute rule 266
8.34.3 Syntax : Manifest constants 266
8.34.4 Syntax (non-production): Sign Syntax rule 266
8.34.5 Syntax (non-production): Character Syntax rule 266
8.34.6 Definition: Type of a manifest constant 267
8.34.7 Validity: Manifest-Type Qualifier rule 267

Signed by JV - 2026/10/8

8.34.8 Semantics: Manifest constant semantics 268
8.34.9 Definition: Manifest value of a constant 268
8.34.10 Syntax : Manifest strings 268
8.34.11 Syntax (non-production): Line_sequence 269
8.34.12 Syntax (non-production): Manifest String rule 269
8.34.13 Syntax (non-production): Line_wrapping_part 269
8.34.14 Semantics: Manifest string semantics 270
8.34.15 Validity: Verbatim String rule 270
8.34.16 Semantics: Verbatim string semantics 270
8.34.17 Definition: Prefix, longest break prefix, left-aligned form 270
8.35 Basic types 271
8.35.1 Definition: Basic types and their sized variants 271
8.35.2 Definition: Sized variants of *STRING* *271*
8.35.3 Semantics: Boolean value semantics 271
8.35.4 Semantics: Character types 271
8.35.5 Semantics: Integer types 271
8.35.6 Semantics: Floating-point types 271
8.35.7 Semantics: Address semantics 272
8.36 Interfacing with C, C++ and other environments 273
8.36.1 Syntax : External routines 273
8.36.2 Validity: Address Expression rule 274
8.36.3 Semantics: Address semantics 274
8.36.4 Syntax : Registered languages 274
8.36.5 Syntax : External signatures 274
8.36.6 Validity: External Signature rule 274
8.36.7 Semantics: External signature semantics 275
8.36.8 Syntax : External file use 275
8.36.9 Validity: External File rule 275
8.36.10 Semantics: External file semantics 276
8.36.11 Syntax : C externals 276
8.36.12 Validity: C External rule 277
8.36.13 Semantics: C Inline semantics 277
8.36.14 Syntax : C++ externals 277
8.36.15 Validity: C++ External rule 277
8.36.16 Semantics: C++ Inline semantics 277
8.36.17 Syntax : DLL externals 278
8.36.18 Validity: External DLL rule 278
8.36.19 Semantics: External DLL semantics 278
8.37 Lexical components 279
8.37.1 Syntax (non-production): Character, character set 279
8.37.2 Definition: Letter, letter case, alpha_betic, numeric, alpha_numeric, printable279
8.37.3 Definition: Break character, break 280

Signed by JV - 2026/10/8

8.37.4 Semantics: Break semantics 280
8.37.5 Definition: Expected, free comment 280
8.37.6 Syntax (non-production): “Blanks or tabs”, new line 280
8.37.7 Syntax : Comments 280
8.37.8 Syntax (non-production): Free Comment rule 281
8.37.9 Principle: Header Comment rule 281
8.37.10 Definition: Symbol, word 281
8.37.11 Syntax (non-production): Break rule 281
8.37.12 Semantics: Letter Case rule 282
8.37.13 Definition: Reserved word, keyword 282
8.37.14 Syntax (non-production): Double Reserved Word rule 282
8.37.15 Definition: Special symbol 282
8.37.16 Syntax (non-production): Identifier 282
8.37.17 Validity: Identifier rule 283
8.37.18 Definition: Predefined operator 283
8.37.19 Definition: Standard operator 283
8.37.20 Definition: Operator symbol 283
8.37.21 Definition: Free operator 283
8.37.22 Syntax (non-production): Manifest character 284
8.37.23 Syntax (non-production): Special characters and their codes 285
8.37.24 Syntax (non-production): Percent variants 285
8.37.25 Semantics: Manifest character semantics 285
8.37.26 Syntax (non-production): String, simple string 285
8.37.27 Semantics: String semantics 286
8.37.28 Syntax : Integers 286
8.37.29 Validity: Integer rule 286
8.37.30 Semantics: Integer semantics 286
8.37.31 Syntax (non-production): Real number 287
8.37.32 Semantics: Real semantics 287

PART J: REFERENCE

8.9 Syntax summary (in order of appearance) 290
8.10 Syntax summary (in alphabetical order of constructs) 299

Signed by JV - 2026/10/8

1 Scope

1.1 Overview

This document provides the full reference for the Eiffel language.

Eiffel is a method of software construction and a language applicable to the analysis, design, implementation and maintenance of software systems. This Standard covers only the language, with an emphasis on the implementation aspects. As a consequence, the word “Eiffel” in the rest of this document is an abbreviation for “the Eiffel language”.

1.2 “The Standard”

The language definition proper — “the Standard” — is contained in Partition 8 of this document, with the exception of text appearing between markers *Informative Text* and *End*, which only plays an explanatory role.

1.3 Aspects covered

The Standard specifies:

- The form of legal basic constituents of Eiffel texts, or *lexical* properties of the language.
- The structure of legal Eiffel texts made of lexically legal constituents, or *syntax* properties.
- Supplementary restrictions imposed on syntactically legal Eiffel texts, or *validity* properties.
- The computational effect of executing valid Eiffel texts, or *semantic* properties.
- Some requirements on a conforming implementation of Eiffel, such as the ability to produce certain forms of automatic documentation.

1.4 Aspects not covered

The Standard does not specify:

- The requirements that a computing environment must meet to support the translation, execution and other handling of Eiffel texts.
- The semantic properties of an Eiffel text if it or its data exceed the capacity of a particular computing environment.
- The mechanisms for translating Eiffel texts into a form that can be executed in a computing environment.
- The mechanisms for starting the execution of the result of such a translation.
- Other mechanisms for handling Eiffel texts and interacting with users of the language.

Signed by JV - 2026/10/8

2 Conformance

2.1 Definition

An implementation of the Eiffel language is **conformant** if and only if in its default operating mode, when provided with a candidate software text, it:

- Can, if the text and all its elements satisfy the lexical, syntax and validity rules of the Standard, execute the software according to the semantic rules of the Standard, or generate code for a computing environment such that, according to the specification of that environment, the generated code represents the semantics of the text according to these rules.
- Will, if any element of the text violates any lexical, syntactical or validity rule of the Standard, report an error and perform no semantic processing (such as generating executable code, or directly attempting to execute the software).

2.2 Compatibility and non-default options

Implementations may provide options that depart in minor ways from the rules of this Standard, for example to provide compatibility with earlier versions of the implementation or of the language itself. Such options are permitted if and only if:

- Per 2.1, they are not the default.
- The implementation includes documentation that states that all such options are nonconformant.

2.3 Departure from the Standard

Material reasons, such as bugs or lack of time in the tool development process, may lead to the release of an implementation that supports most of the Standard but misses a few rules and hence is not yet conformant according to the definition of 2.1. In such a case the implementation shall include documentation that:

- States that the implementation is not conformant.
- Lists all the known causes of non-conformance.
- Provides an estimate of the date or version number for reaching full conformance.

Signed by JV - 2026/10/8

3 Normative references

3.1 Earlier Eiffel language specifications

Bertrand Meyer: *Eiffel: The Language*, Prentice Hall, second printing, 1992 (first printing: 1991).

Bertrand Meyer: *Standard Eiffel* (revision of preceding entry), ongoing, 1997-present, at http://www.inf.ethz.ch/~meyer/ongoing/etl.

Bertrand Meyer: *Object-Oriented Software Construction*, Prentice Hall: first edition, 1988; second edition, 1997.

3.2 Eiffel Kernel Library

The terms “ELKS” and “Kernel Library”, as used in this Standard, refer to the latest version of the Eiffel Library Kernel Standard. A preliminary version is available from the NICE consortium:

NICE consortium: *The Eiffel Library Kernel Standard, 2001 Vintage*.

The Standard assumes that ELKS includes at least the following classes:

- Classes representing fundamental language-related concepts: *ANY*, *DISPOSABLE*, *NONE*, *TYPE*, *TYPED_POINTER*.
- Classes representing basic types and strings: *BOOLEAN*, *CHARACTER*, *CHARACTER_8*, *CHARACTER_32*, *INTEGER*, *INTEGER_8*, *INTEGER_16*, *INTEGER_32*, *INTEGER_64*, *NATURAL*, *NATURAL_8*, *NATURAL_16*, *NATURAL_32*, *NATURAL_64*, *POINTER*, *REAL*, *REAL_32*, *REAL_64*, *STRING*, *STRING_8*, *STRING_32*.
- Classes representing fundamental data structures: *ARRAY*, *TUPLE*.
- Agent-related classes: *FUNCTION*, *PREDICATE*, *PROCEDURE*, *ROUTINE*.
- Exception-related classes: *ASSERTION_VIOLATION*, *ATTACHED_TARGET_VIOLATION*, *EXCEPTION*, *INSPECT_RANGE_VIOLATION*, *INVARIANT_ENTRY_VIOLATION*, *INVARIANT_EXIT_VIOLATION*, *MEMORY_ALLOCATION_FAILURE*, *OLD_VIOLATION*, *POSTCONDITION_VIOLATION*, *PRECONDITION_VIOLATION*, *ROUTINE_FAILURE*.

The clauses referring to these classes list the features they need.

3.3 Floating point number representation

IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems* (previously designated IEC 559:1989). Also known as ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*.

3.4 Character set: Unicode

The Unicode Consortium: *The Unicode Standard*, Version 4.1, at http://www.unicode.org/versions/Unicode4.1.0/.

3.5 Character set: ASCII

ISO 14962:1997: *Space data and information transfer systems*.

3.6 Phonetic alphabet

International Phonetic Association: *International Phonetic Alphabet* (revised 1993, updated 1996).

Signed by JV - 2026/10/8

4 Definitions

All the Eiffel-specific terms used in the Standard are defined in paragraphs labeled “**Definition**”. Like other normative elements, definitions each have an identifying 4-character code, starting (for definitions) with the letter “D”.

5 Notational conventions

5.1 Normative elements

Every normative element of the standard possesses a unique 4-letter code, such as "VTCT" (see VTCT, page 167), built as follows:

- First character: one of the letters S, V, M, D and P, corresponding to the kind of clause as specified in 5.3 below. In this example it is V (used for Validity constraints).
- Second character (upper-case letter, or digit): unique identification of the clause in the standard. In this case the code is T, identifying the clause on types (clause 8.14, starting on page 165).
- Third and fourth characters: upper-case letters serving as a mnemonic for the element. In this example, the element is the Class Type rule and the mnemonic part of the code is CT.

5.2 Non-normative elements

"Informative Text" elements occupy a significant part of the present document. They are not, however (as specified in 1.2 above) part of "the standard". They are merely intended to facilitate reading by providing numerous explanations and examples. To answer any specific question about the Eiffel language, the normative elements are the reference, not the Informative Text.

Informative Text appears at the beginning of every section and after most clauses. As a general convention, explanations about a clause appear in the Informative Text after that clause. (As an occasional exception to this practice, some Informative Text elements end with a sentence or two introducing a transition to the next clause, with phrasing such as "*there is another case, as introduced in the next rule*".)

5.3 Kinds of clause

Every clause of the Standard belongs to one of the following categories:

- **Definition**: introduction of a term defining a certain concept, which may relate to syntax, validity or semantic properties; the rest of the Standard may use the term as an abbreviation for the concept. The code for such clauses starts with "D".
- **Syntax**: rule affecting the structure of Eiffel texts, including lexical properties as well as syntax proper. The conventions for describing syntax appear below. The code for such clauses starts with "S".
- **Validity**: restrictions on syntactically legal texts. The code for such clauses starts with "V".
- **Semantics**: properties of the execution of valid texts. The code for such clauses starts with "M".
- **Principle**: a general language design rule, or a property of the software text resulting from other properties covered by definition and syntax, validity and semantic rules. The code for such clauses starts with "P".

The clauses of the Standard are labeled with the corresponding category. A clause with no label shall be considered as "Definition".

5.4 Normative elements

The rules of syntax, validity and semantics constitute the necessary and sufficient conditions for an implementation to be conformant.

The entries labeled **principle**, addressing validity in most cases and semantics in a few others, express properties that follow from the official rules. For example the Feature Identifier principle (8.5.18) states

> "Given a class *C* and an identifier *f*, *C* contains at most one feature of identifier *f*."

Signed by JV - 2026/10/8

This property (expressing that there must be no overloading of feature identifiers within a class) follows from a variety of validity rules.

Such principles are conceptually redundant and do not add any constraint on an implementation; They provide an interesting angle on the language definition, both for:

- Programmers, by capturing a general property in a concise way.
- Implementers, who when reporting a rule violation in a candidate Eiffel text may choose to refer to the principle rather than the (possibly less immediately clear) rules that underlie it.

5.5 Rules on definitions

In a definition, the first occurrence of each term being defined appears in **bold**.

Some definitions are *recursive*, meaning that the definition of a certain property of certain language elements distinguishes between two or more cases, and in some (but not all) of these cases uses the same property applied to smaller language elements. (The definition then makes sense as all cases eventually reduce to basic, non-recursive cases.) For clarity, the recursive branches are always marked by the word "*recursively*"" in parentheses.

In the text of the Standard, uses of a defined term may appear before the definition. The underlining convention described next helps avoid any confusion in such cases.

5.6 Use of defined terms

In any formal element of the language specification (definition, syntax, validity, semantics), appearance of a word as underlined means that this is a technical term introduced in one of the definitions of the Standard. For example, condition 1 in the Name Clash rule (8.16.16) reads:

> "It is invalid for *C* to introduce two different features with the same name."

The underlining indicates that words like "introduce" are not used in their plain English sense but as Eiffel-related concepts defined precisely elsewhere in the Standard. Indeed, 8.5.1 defines the notion that a class "introduces" a feature. Similarly, the notion of features having the "same name" (actually, "same feature name") is precisely defined in DFSO, page 107, to mean that their identifiers are identical *except* possibly for letter case.

This use of underlining is subject to the following restrictions:

- Underlining applies only to the first occurrence of such a defined term in a particular definition or rule.
- If a single clause defines a group of terms, occurrences of one of these terms in the definition of another are not underlined.
- As a consequence, uses of a term in its own definition are not underlined (they do bear, as noted, the mark "*recursively*").
- In addition, a number of basic concepts, used throughout, are generally not underlined; they include:

Assertion	*Attribute*	*Call*	*Character*	*Class*
Cluster	*Entity*	*Expression*	*Feature*	*Identifier*
Instruction	*Semantics*	*Type*	*Valid*	

5.7 Unfolded forms

The definition of Eiffel in the Standard frequently relies on *unfolded forms* defining certain advanced constructs in terms of simpler ones. For example an "anchored type" of the form **like** *a* has a "deanchored form", which is just the type of *a*. Validity and semantic rules can then rely on the unfolded form rather than the original.

This technique, applied to about twenty constructs of the language, makes the description significantly simpler by identifying a basic set of constructs for which it provides direct validity and semantic specifications, and defining the rest of the language in terms of this core.

5.8 Language description

The Standard follows precise rules and conventions for language description, described in 8.2.

5.9 Validity: "if and only if" rules

A distinctive property of the language description is that the validity constraints, for example type rules, do not just, as in many language descriptions, list properties that a software text *must* satisfy to be valid. They give the rules in an "if and only if" style. For example (8.8.3, Formal Argument rule):

> "Let *fa* be the Formal_arguments part of a routine *r* in a class *C*. Let *formals* be the concatenation of every Entity_list of every Entity_declaration_group in *fa*. Then *fa* is valid if and only if no Identifier *e* appearing in *formals* is the final name of a feature of *C*."

This does not just impose requirements on programmers, but tells them that if they satisfy these requirements they are *entitled* to having the construct (here a "Formal_arguments part") accepted by an Eiffel language processing tool, for example a compiler. The rules, then, are not only necessary but sufficient. This contributes (except of course for the presence of any error or omission in the Standard) to the completeness of the language definition, and reinforces the programmers' trust in the language.

Signed by JV - 2026/10/8

6 Acronyms and abbreviations

6.1 Name of the language

The word “Eiffel” is not an acronym. It is written with a capital initial E, followed by letters in lower case.

6.2 Pronunciation

Verbally, with reference to the International Phonetic Alphabet, the name is pronounced:

- In English: aɪfəl with the stress on the first syllable.
- In French: ɛfɛl with the stress on the second syllable.
- In other languages: the closest approximation to the English or French variant as desired.

Signed by JV - 2026/10/8

PART B: Introduction to Eiffel

Signed by JV - 2026/10/8

Signed by JV - 2026/10/8

7 An informal introduction to Eiffel

The present clause is a tutorial on the Eiffel language. It is informative only.

Informative text

7.1 Design principles

The aim of Eiffel is to help specify, design, implement, reuse and modify quality software. This goal of quality in software is a combination of many factors; the language design concentrated on the three factors which, in the current state of the industry, are in direst need of improvements: reliability, extendibility and reusability. Also important were other factors such as efficiency, openness and portability.

Rreliability is the ability to produce software that is correct and robust — that is to say, bug-free. Eiffel techniques such as static typing, assertions, disciplined exception handling and automatic garbage collection are essential here.

Reusability is the ability to produce components that may be used in many different applications. Central to the Eiffel approach is the presence of widely used libraries complementing the language, and the language's support for the production of new libraries.

Extendibility is the ability to produce easily modifiable software. "Soft" as software is supposed to be, it is notoriously hard to modify software systems, especially large ones.

Among quality factors, reusability and extendibility play a special role: satisfying them means having *less* software to write — and hence more time to devote to other important goals such as efficiency, ease of use and integrity.

Four other factors are also part of Eiffel's principal goals:

- The language enables implementors to produce high **efficiency** compilers, so that systems developed with Eiffel may run under speed and space conditions similar to those of programs written in lower-level languages traditionally focused on efficient implementation.
- Ensuring **openness**, so that Eiffel software may cooperate with programs written in other languages.
- Guaranteeing **portability** by a platform-independent language definition, so that the same semantics may be supported on many different platforms.
- **High-precision language definition**, allowing independent implementers to provide compilers and other tools, and providing Eiffel users the guarantee that their software will work on all the Standard-compliant implementations. The language definition does not favor or refer to any particular implementation.

7.2 Object-oriented design

To achieve reusability, extendibility and reliability, the principles of object-oriented design provide the best known technical answer.

An in-depth discussion of these principles falls beyond the scope of this introduction but here is a short definition:

> *Object-oriented design is the construction of software systems as structured collections of abstract data type implementations, or "classes".*

The following points are worth noting in this definition:

- The emphasis is on structuring a system around the types of objects it manipulates (not the functions it performs on them) and on reusing whole data structures together with the associated operations (not isolated routines).
- Objects are described as instances of abstract data types — that is to say, data structures known from an official interface rather than through their representation.

Signed by JV - 2026/10/8

- The basic modular unit, called the class, describes one implementation of an abstract data type (or, in the case of "deferred" classes, as studied below, a set of possible implementations of the same abstract data type).
- The word *collection* reflects how classes should be designed: as units interesting and useful on their own, independently of the systems to which they belong, and ready to be reused by many different systems. Software construction is viewed as the assembly of existing classes, not as a top-down process starting from scratch.
- Finally, the word *structured* reflects the existence of two important relations between classes: the client and inheritance relations.

Eiffel makes these techniques available to software developers in a simple and practical way.

As a language, Eiffel includes more than presented in this introduction, but not *much* more; it is a small language, not much bigger (by such a measure as the number of keywords) than Pascal. The description as given in the Standard text — excluding "Informative Text" and retaining only the rules — takes up about 80 pages. Eiffel was indeed meant to be a member of the class of languages which programmers can master entirely — as opposed to languages of which most programmers know only a subset. Yet it is appropriate for the development of industrial software systems, as has by now been shown by many full-scale projects, some in the thousands of classes and millions of lines, in companies around the world, for mission-critical systems in many different areas from finance to health care and aerospace.

7.3 Classes

A class, it was said above, is an implementation of an abstract data type. This means that it describes a set of run-time objects, characterized by the **features** (operations) applicable to them, and by the formal properties of these features.

Such objects are called the direct instances of the class. Classes and objects should not be confused: "class" is a compile-time notion, whereas objects only exist at run time. This is similar to the difference that exists in pre-object-oriented programming between a program and one execution of that program, or between a type and a run-time value of that type.

To see what a class looks like, let us look at a simple example, *ACCOUNT*, which describes bank accounts. Before exploring the class itself it is useful to study how it may be used by other classes, called its **clients**.

A class *X* may become a client of *ACCOUNT* by declaring one or more **entities** of type *ACCOUNT*. Such a declaration is of the form:

acc: *ACCOUNT*

An entity such as *acc* that programs can directly modify is called a **variable**. An entity declared of a reference type, such as *acc*, may at any time during execution become "attached to" an object (see figure 1); the type rules imply that this object must be a direct instance of *ACCOUNT* — or, as seen later, of a "descendant" of that class.

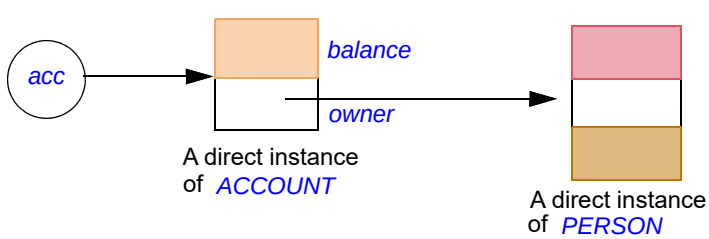


Figure 1: References and objects

Signed by JV - 2026/10/8

An entity is said to be void if it is not attached to any object. As seen below, the type system achieves strict control over void entities, for run-time safety. To obtain objects at run-time, a routine *r* appearing in the client class *X* may use a **creation instruction** of the form

```
create acc
```

which creates a new direct instance of *ACCOUNT*, sets up all its fields for initialization to default values, and attaches *acc* to that instance. A variant of this notation, studied below, makes it possible to override the default initializations.

Once the client has attached *acc* to an object, it may call on this object the features defined in class *ACCOUNT*. Here is an extract with some feature calls using *acc* as their target:

```
acc.open ("Jill")
acc.deposit (5000)
if acc.may_withdraw (3000) then
        acc.withdraw (3000); print (acc.balance)
end
```

These feature calls use dot notation, of the form *target.feature_name*, possibly followed by a list of arguments in parentheses. Features are of two kinds:

- **Routines** (such as *open*, *deposit*, *may_withdraw*, *withdraw*) represent computations applicable to instances of the class.
- **Attributes** represent data items associated with these instances.

Routines are further divided into **procedures** (commands, which do not return a value) and functions (queries, returning a value). Here *may_withdraw* is a function returning a boolean; the other three-routines called are procedures.

The above extract of class *X* does not show whether, in class *ACCOUNT*, *balance* is an attribute or a function without argument. This ambiguity is intentional. A client of *ACCOUNT*, such as *X*, does not need to know how a balance is obtained: the balance could be stored as an attribute of every account object, or computed by a function from other attributes. Choosing between these techniques is the business of class *ACCOUNT*, not anybody else's. Because such implementation choices are often changed over the lifetime of a project, it is essential to protect clients against their effects.

This **Principle of Uniform Access** permeates the use of attributes and functions throughout the language design: whenever one of these feature variants can be used, and the other would make sense too, both are permitted. Descendant classes can for example *redefine* a function into an attribute, and attributes can have *assertions* just like functions. The term query covers both attributes and functions.

The above example illustrates the Eiffel syntax style: focusing on readability, not overwhelming the reader with symbols, and using simple keywords, each based on a single English word (in its simplest form: **feature** in the singular, **require** and not "requires"). The syntax is free-format — spaces, new lines, tabs have the same effect — and yet does not require a separator between successive instructions. You may use semicolons as separators if you wish, but they are optional in *all* possible contexts, and most Eiffel programmers omit them (as an obstacle to readability) except in the rare case of several instructions written on one line.

So much for how client classes will typically use *ACCOUNT*. Next is a first sketch of how class *ACCOUNT* itself might look. Line segments beginning with -- are comments. The class includes two **feature** clauses, introducing its features. The first begins with just the keyword **feature**, without further qualification; this means that the features declared in this clause are available (or "exported") to all clients of the class. The second clause is introduced by **feature** {*NONE*} to indicate that the feature that follows, called *add*, is available to no client. What appears between

Signed by JV - 2026/10/8

the braces is a list of client classes to which the listed features are available; *NONE* is a special class of the Kernel Library, which has no instances, so that *add* is in effect a secret feature, available only locally to the other routines of class *ACCOUNT*. In a client class such as *X*, the call *acc.add* (*–3000*) would be invalid and hence rejected by any Standard-compliant language processing tool.

```
class ACCOUNT feature
        balance: INTEGER
                -- Amount on the account
        owner: PERSON
                -- Account holder
        minimum_balance: INTEGEGER = 1000
                -- Lowest permitted balance

        open (who: PERSON)
                -- Assign the account to owner who.
            do
                owner := who
            end

        deposit (sum: INTEGER)
                -- Add an amount of sum to the account.
            do
                add (sum)
            end

        withdraw (sum: INTEGER)
                -- Remove an amount of sum from the account.
            do
                add (–sum)
            end

        may_withdraw (sum: INTEGER): BOOLEAN
                -- Is there enough money to withdraw sum?
            do
                Result := (balance >= sum + minimum_balance)
            end

feature {NONE}
        add (sum: INTEGER)
                -- Add sum to the balance.
            do
                balance := balance + sum
            end
end
```

Let us examine the features in sequence. The **do** ... **end** distinguishes routines from attributes. So here the class has implemented *balance* as an attribute, although, as noted, a function would also have been acceptable. Feature *owner* is also an attribute.

The language definition guarantees automatic initialization, so that the balance of an account object will be zero the first time it is accessed. Each type has a default initialization value, such as false for *BOOLEAN*, zero for integer and real types, null character for character types and, for a reference type, either a void value or a reference to a new object of that type. The class designer may also provide clients with different initialization options, as will be shown in a revised version of this example.

The other public features, *open*, *deposit*, *withdraw* and *may_withdraw* are straightforward routines. The special variable *Result*, used in *may_withdraw*, denotes the function result; it is initialized on function entry to the default value of the function's result type.

Signed by JV - 2026/10/8

The secret procedure *add* serves for the implementation of the public procedures *deposit* and *withdraw*; the designer of *ACCOUNT* judged it too general to be exported by itself. The clause *= 1000* introduces *minimum_balance* as a constant attribute, which need not occupy any space in instances of the class; in contrast, every instance has a field for every non-constant attribute such as *balance*.

In Eiffel's object-oriented programming style, every operation is relative to a certain object. In a client invoking the operation, this object is specified by writing the corresponding entity on the left of the dot, as *acc* in *acc.open* ("*Jill*"). Within the class, however, the "current" instance to which operations apply usually remains implicit, so that unqualified feature names, such as *owner* in procedure *open* or *add* in *deposit*, mean "the *owner* attribute or *add* routine relative to the current instance".

If you need to denote the current object explicitly, you may use the special entity *Current*. For example the unqualified occurrences of *add* appearing in the above class are essentially equivalent to *Current.add*.

In some cases, infix or prefix notation will be more convenient than dot notation. For example, if a class *VECTOR* offers an addition routine, most people will feel more comfortable with calls of the form *v + w* than with the dot-notation call *v.plus* (*w*). To make this possible it suffices to give the routine a name of the form *plus* **alias** "+"; internally, however, the operation is still a normal routine call. You can also use unary operators for prefix notation. It is also possible to define a *bracket alias*, as in the feature *item* **alias** "[]" of the Kernel Library class *ARRAY*, which returns an array element of given index *i*: in dot notation, you will write this as *your_array.item* (*i*), but the bracket alias allows *your_array* [*i*] as an exact synonym. At most one feature per class may have a bracket alias. These techniques make it possible to use well-established conventions of mathematical notation and traditional programming languages in full application of object-oriented principles.

The above simple example has shown the basic structuring mechanism of the language: the class. A class describes a data structure, accessible to clients through an official interface comprising some of the class features. Features are implemented as attributes or routines; the implementation of exported features may rely on other, secret ones.

7.4 Types

Eiffel is strongly typed for readability and reliability. Every entity is declared of a certain type, which may be either a reference type or an expanded type.

Any type *T* is based on a class, which defines the operations that will be applicable to instances of *T*. The difference between the two categories of type affects the semantics of using an instance of *T* as source of an *attachment*: assignment or argument passing. An attachment from an object of a reference type will attach a new reference to that object; with an expanded type, the attachment will *copy* the contents of the object. Similarly, comparison operations such as *a = b* will compare references in one case and objects contents in the other. (To get object comparison in all cases, use *a ~ b*.) We talk of objects with *reference semantics* and objects with *copy semantics*.

Syntactically, the difference is simple: a class declared without any particular marker, like *ACCOUNT*, yields a reference type. To obtain an expanded type, just start with **expanded class** instead of just **class**.

It may be useful to think of expanded and reference types in terms of figure 2, where we assume that *ACCOUNT* has an extra attribute *exp* of type *EXP*, using a class declared as **expanded class**

Signed by JV - 2026/10/8

EXP. Figure 2 shows the entity *acc* denoting at run time a reference to an instance of *ACCOUNT* and, in contrast, *exp* in that instance denoting a **subobject**, not a reference to another object:

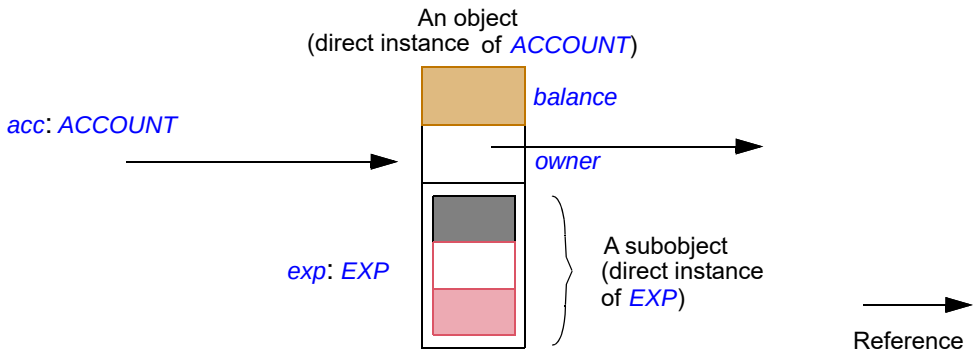


Figure 2: Object and subobject

This is only an illustration, however, and implementations are not required to implement expanded values as subobjects. What matters is the distinction between copy and reference semantics.

An important group of expanded types, based on library classes, includes the basic types *INTEGER*, *REAL*, *CHARACTER* and *BOOLEAN*. Clearly, the value of an entity declared of type *INTEGER* should be an integer, not a reference to an object containing an integer value. Operations on these types are defined by prefix and infix operators such as "+" and "<".

As a result of these conventions, the type system is uniform and consistent: all types, reference or expanded — including the basic types —, are defined from classes.

In the case of basic types, for obvious reasons of efficiency, compilers can and usually do implement the usual arithmetic and boolean operations directly through the corresponding machine operations, not through routine calls. So the performance is the same as if basic types were “magic”, outside of the object-oriented type system. But this is only a compiler optimization, which does not hamper the conceptual homogeneity of the type edifice.

7.5 Assertions

If classes are to deserve their definition as abstract data type implementations, they must be known not just by the available operations, but also by the formal properties of these operations, which did not appear in the above example.

Eiffel encourages software developers to express formal properties of classes by writing **assertions**, which may in particular appear in the following roles:

- Routine **preconditions** express the requirements that clients must satisfy whenever they call a routine. For example the designer of *ACCOUNT* may wish to permit a withdrawal operation only if it keeps the account’s balance at or above the minimum. Preconditions are introduced by the keyword **require**.
- Routine **postconditions**, introduced by the keyword **ensure**, express conditions that the routine (the supplier) guarantees on return, if the precondition was satisfied on entry.
- A class **invariant** must be satisfied by every instance of the class whenever the instance is externally accessible: after creation, and after any call to an exported routine of the class. The invariant appears in a clause introduced by the keyword **invariant**, and represents a general consistency constraint imposed on all routines of the class.

The above class *ACCOUNT* typifies why classes often need such assertions. Implicitly, as suggested by the feature names, the *balance* of an account should always remain at least equal to its *minimum_balance*. But nothing in the class text expresses this fundamental property, or the constraints it implies on the arguments to such features as *deposit* and

Signed by JV - 2026/10/8

withdraw. By adding assertions we make these properties an integral part of the class. We also use this opportunity to make the class more flexible by making *minimum_balance* no longer a constant (1000 above) but a variable attribute to be set by each client when it creates an instance of the class. The revised version is:

```
class ACCOUNT create
        make
feature
        ... Some features as before: balance, owner, open ...

        minimum_balance: INTEGER
                -- Minimum permitted value for balance

        deposit (sum: INTEGER)
                -- Deposit sum into the account.
            require
                positive: sum > 0
            do
                add (sum)
            ensure
                deposited: balance = old balance + sum
            end

        withdraw (sum: INTEGER)
                -- Withdraw sum from the account.
            require
                positive: sum > 0
                sufficient_funds: sum <= balance – minimum_balance
            do
                add (–sum)
            ensure
                withdrawn: balance = old balance – sum
            end

        may_withdraw ...  -- As before

feature {NONE}
        add ... -- As before
        make (initial, min: INTEGER)
                -- Initialize account with balance initial and minimum balance min.
            require
                not_under_minimum: initial >= min
            do
                minimum_balance := min
                balance := initial
            ensure
                balance_initialized: balance = initial
                minimum_initialized: minimum_balance = min
            end

invariant
        sufficient_balance: balance >= minimum_balance
end
```

The notation **old** *expression* may only be used in a routine postcondition. It denotes the value the *expression* had on routine entry.

In its last version above, the class now includes a creation procedure, *make*. With the first version of *ACCOUNT*, clients used creation instructions such as **create** *acc1* to create accounts; but then the default initialization, setting *balance* to zero, violated the invariant. By having one or more creation procedures, listed in the **create** clause at the beginning of the class text, a class offers a way to override the default initializations. The effect of

Signed by JV - 2026/10/8

create *acc1*.*make* (5500, 1000)

is to allocate the object (as with default creation) and to call procedure *make* on it, with the arguments given. This call is correct since it satisfies the precondition; it will ensure that the created object satisfies the invariant.

Note that the same keyword, **create**, serves both to introduce creation instructions and the creation clause listing creation procedures at the beginning of the class.

A procedure listed in the creation clause, such as *make*, otherwise enjoys the same properties as other routines, especially for calls. Here the procedure *make* is secret since it appears in a clause starting with **feature** {*NONE*}; so it would be invalid for a client to include a call such as

acc.*make* (5500, 1000)

To make such a call valid, it would suffice to move the declaration of *make* to the first **feature** clause of class *ACCOUNT*, which carries no export restriction. Such a call does not create any new object, but simply resets the balance of a previously created account.

Syntactically, assertions are boolean expressions, with a few extensions such as the **old** notation. Writing a succession of assertion clauses, as in the precondition to *withdraw*, is equivalent to “and-ing” them, but permits individual identification of the components. (As with instructions you could use a semicolon between assertion clauses, although it is optional and generally omitted.)

Assertions play a central part in the Eiffel method for building reliable object-oriented software. They serve to make explicit the assumptions on which programmers rely when they write software elements that they believe are correct. Writing assertions, in particular preconditions and postconditions, amounts to spelling out the terms of the **contract** which governs the relationship between a routine and its callers. The precondition binds the callers; the postcondition binds the routine.

The underlying theory of *Design by Contract,* the centerpiece of the Eiffel method, views software construction as based on contracts between clients (callers) and suppliers (routines), relying on mutual obligations and benefits made explicit by the assertions.

Assertions are also an indispensable tool for the documentation of reusable software components: one cannot expect large-scale reuse without a precise documentation of what every component expects (precondition), what it guarantees in return (postcondition) and what general conditions it maintains (invariant).

Documentation tools in Eiffel implementations use assertions to produce information for client programmers, describing classes in terms of observable behavior, not implementation. In particular the **contract view** of a class, which serves as its basic documentation, is obtained from the full text by removing all non-exported features and all implementation information such as **do**

Signed by JV - 2026/10/8

clauses of routines, but keeping interface information and in particular assertions. Here is the interface of the above class (with some features omitted):

```
class interface ACCOUNT create
        make
feature
        balance: INTEGER
                -- Amount on the account

        minimum_balance: INTEGER
                -- Lowest permitted balance

        deposit (sum: INTEGER)
                -- Deposit sum into the account.
            require
                positive: sum > 0
            ensure
                deposited: balance = old balance + sum
        withdraw (sum: INTEGER)
                -- Withdraw sum from the account.
            require
                positive: sum > 0
                sufficient_funds: sum <= balance – minimum_balance
            ensure
                withdrawn: balance = old balance – sum
invariant
        sufficient_balance: balance >= minimum_balance
end
```

This is not an Eiffel text, only documentation of Eiffel classes, hence the use of slightly different syntax to avoid any confusion (**class interface** rather than **class**). In accordance with observations made above, the output for *balance* would be the same if this feature were a function rather than an attribute.

Such an interface can be produced by automatic tools from the text of the software. It serves as the primary form of class documentation. A variant of the contract view includes Inherited features along with those introduced in the class itself.

It is also possible to evaluate assertions at run time, to uncover potential errors (“bugs”). The implementation provides several levels of assertion monitoring: preconditions only, postconditions etc. With monitoring on, an assertion that evaluates to true has no further effect on the execution. An assertion that evaluates to false will trigger an exception, as described next; in the absence of a specific exception handler the exception will cause an error message and termination.

This ability to check assertions provides a powerful testing and debugging mechanism, in particular because the classes of widely used libraries are equipped with extensive assertions.

Run-time checking, however, is only one application of assertions, whose role as design and documentation aids, as part of the theory of Design by Contract, exerts a pervasive influence on the Eiffel style of software development.

7.6 Exceptions

Whenever there is a contract, the risk exists that someone will break it. This is where exceptions come in.

Exceptions — contract violations — may arise from several causes. One is assertion violations, if assertions are monitored. Another is the occurrence of a signal triggered by the hardware or operating system to indicate an abnormal condition such as arithmetic overflow or lack of memory to create a new object.

Unless a routine has made specific provision to handle exceptions, it will **fail** if an exception arises during its execution. Failure of a routine is a third cause of exception: a routine that fails triggers an exception in its caller.

Signed by JV - 2026/10/8

A routine may, however, handle an exception through a **rescue** clause. This optional clause attempts to "patch things up" by bringing the current object to a stable state (one satisfying the class invariant). Then it can terminate in either of two ways:

- The **rescue** clause may terminate with the predefined boolean variable **Retry** having the value True; this causes the routine to restart its execution from the beginning, attempting again to fulfil its contract, usually through another strategy. In such a case the instructions of the **rescue** clause, besides setting **Retry** to True, will have attempted to correct the cause of the exception.
- If the **rescue** clause ends with **Retry** False, then the routine fails: it returns to its caller, immediately signaling an exception. (The caller's **rescue** clause will be executed according to the same rules.)

The principle is that **a routine must either succeed or fail**: either it fulfils its contract, or it does not; in the latter case it must notify its caller by triggering an exception.

Usually, only a few routines of a system will include explicit **rescue** clauses. An exception occurring during the execution of a routine with no **rescue** clause will trigger a predefined rescue procedure, which does nothing, and so will cause the routine to fail immediately, propagating the exception to the routine's caller.

An example using the exception mechanism is a routine *attempt_transmission* which tries to transmit a message over a phone line. The actual transmission is performed by an external, low-level routine *transmit*; once started, however, *transmit* may abruptly fail, triggering an exception, if the line is disconnected. Routine *attempt_transmission* tries the transmission at most 50 times; before returning to its caller, it sets a boolean attribute *successful* to **true** or **false** depending on the outcome. Here is the text of the routine:

```
attempt_transmission (message: STRING)
            -- Try to transmit message, at most 50 times.
            -- Set successful accordingly.
        local
            failures: INTEGER
        do
            if failures < 50 then
                transmit (message); successful := true
            else
                successful := false
            end
        rescue
            failures := failures + 1; Retry := True
        end
```

Initialization rules ensure that *failures*, a local variable, is initially zero.

This example illustrates the simplicity of the mechanism: the **rescue** clause never attempts to achieve the routine's original intent; this is the sole responsibility of the body (the **do** clause). The only role of the **rescue** clause is to clean up the objects involved, and then either to fail or to retry.

The Kernel Library provides a class *EXCEPTION* and a number of descendants describing specific kinds of exception. Triggering of an exception produces an instance of one of these types, making it possible, in the **rescue** clause, to perform more specific exception processing.

This disciplined exception mechanism is essential for software developers, who need protection against unexpected events, but cannot be expected to sacrifice safety and simplicity to pay for this protection.

7.7 Genericity

Building software components (classes) as implementations of abstract data types yields systems with a solid architecture but does not in itself suffice to ensure reusability and extendibility. Two key techniques address the problem: genericity (unconstrained or constrained) and inheritance. Let us look first at the unconstrained form.

Signed by JV - 2026/10/8

To make a class generic is to give it **formal generic parameters** representing arbitrary types, as in these examples from typical libraries:

ARRAY [*G*]
LIST [*G*]
LINKED_LIST [*G*]

These classes describe data structures — arrays, lists without commitment to a specific representation, lists in linked representation — containing objects of a certain type. The formal generic parameter *G* represents this type.

Such a class describes a type template. To derive a directly usable type, you must provide a type corresponding to *G*, called an actual generic parameter; this might be a basic type (such as *INTEGER*) or a reference type. Here are some possible generic derivations:

il: *LIST* [*INTEGER*]
aa: *ARRAY* [*ACCOUNT*]
aal: *LIST* [*ARRAY* [*ACCOUNT*]]

As the last example indicates, an actual generic parameter may itself be generically derived.

Without genericity, it would be impossible to obtain static type checking in a realistic object-oriented language.

A variant of this mechanism, *constrained* genericity, introduced below after inheritance, enables a class to place specific requirements on possible actual generic parameters.

7.8 Inheritance

Inheritance, the other fundamental generalization mechanism, makes it possible to define a new class by combination and specialization of existing classes rather than from scratch.

The following simple example describes lists implemented by arrays, combining *LIST* and *ARRAY* through inheritance:

```
class ARRAYED_LIST [G] inherit
        LIST [G]
inherit {NONE}
        ARRAY [G]
                export ... See below ...end
feature
        ... Specific features of lists implemented by arrays ...
end
```

The **inherit**... clauses list all the “parents” of the new class, which is said to be their “heir”. (The “ancestors” of a class include the class itself, its parents, grandparents etc.; the reverse term is “descendant”.) Declaring *ARRAYED_LIST* as shown ensures that all the features and properties of lists and arrays are applicable to arrayed lists as well. Since the class has more than one parent, this is a case of multiple inheritance.

Signed by JV - 2026/10/8

In this case one of the parents is introduced by a different clause, reading **inherit** {*NONE*}; this specifies **non-conforming inheritance**, where it will not be possible to assign values of the new types to variables of the parent type. The other branch, with just **inherit**, is conforming, so we can assign an *ARRAYED_LIST* [*T*] to a *LIST*[*T*]. This reflects the distinction between the "subtyping" and "pure reuse" forms of inheritance.

Standard graphical conventions (figure 3) serve to illustrate such inheritance structures:

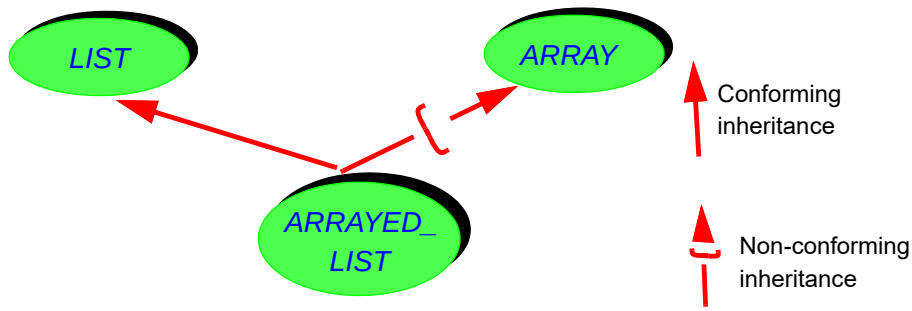


Figure 3: Inheritance

A non-conforming heir class such as *ARRAYED_LIST* needs the ability to define its own export policy. By default, Inherited features keep their export status (publicly available, secret, available to selected classes only); but this may be changed in the heir. Here, for example, *ARRAYED_LIST* will export only the exported features of *LIST*, making those of *ARRAY* unavailable directly to *ARRAYED_LIST*'s clients. The syntax to achieve this is straightforward:

```
class ARRAYED_LIST [G] inherit
        LIST [G]
inherit {NONE}
        ARRAY [G]
                export {NONE} all end
    ... The rest as above ...
```

Another example of multiple inheritance comes from a windowing system based on a class *WINDOW*, as in the EiffelVision graphics library. Windows have **graphical** features: a height, a width, a position, routines to scale windows, move them, and other graphical operations. The system permits windows to be nested, so that a window also has **hierarchical** features: access to subwindows and the parent window, adding a subwindow, deleting a subwindow, attaching to another parent and so on. Rather than writing a complex class that would contain specific implementations for all of these features, it is preferable to inherit all hierarchical features from *TREE*, and all graphical features from a class *RECTANGLE*. In this case both branches are conforming, so a single **inherit** clause listing two parents suffices:

```
class WINDOW inherit
        RECTANGLE
        TREE [WINDOW]
    ...
```

Inheritance yields remarkable economies of effort — whether for analysis, design, implementation or evolution — and has a profound effect on the entire development process.

Signed by JV - 2026/10/8

The very power of inheritance demands adequate means to keep it under control. Multiple inheritance, in particular, raises the question of name conflicts between features inherited from different parents. You may simply remove such a name conflict through renaming, as in

```
class C inherit
        A rename x as x1, y as y1 end
        B rename x as x2, y as y2 end
feature...
```

Here, if both *A* and *B* have features named *x* and *y*, class *C* would be invalid without the renaming.

Renaming also serves to provide more appropriate feature names in descendants. For example, class *WINDOW* may inherit a routine *insert_subtree* from *TREE*. For clients of *WINDOW*, however, such a routine name is no longer appropriate. An application using this class for window manipulation needs coherent window terminology, and should not be concerned with the inheritance structure that led to the implementation of the class. So you may wish to rename *insert_subtree* as *add_subwindow* in the inheritance clause of *WINDOW*.

As a further facility to protect against misusing the multiple inheritance mechanism, the invariants of all parent classes automatically apply to a newly defined class.

7.9 Polymorphism and dynamic binding

Inheritance is not just a module combination and enrichment mechanism. It also enables the definition of flexible entities that may become attached to objects of various forms at run time, a property known as polymorphism.

Complementary mechanisms make this possibility particularly powerful: **feature redefinition** and **dynamic binding**. The first enables a class to redefine some or all of the features which it inherits from its parents. For an attribute or function, the redefinition may affect the type, replacing the original by a descendant; for a routine it may also affect the implementation, replacing the original's routine body by a new one.

Assume for example a class *POLYGON*, describing polygons, whose features include an array of points representing the vertices and a function *perimeter* which computes a polygon's perimeter by summing the successive distances between adjacent vertices. An heir of *POLYGON* may begin:

```
class RECTANGLE inherit
        POLYGON redefine perimeter end
feature -- Specific features of rectangles, such as:
        side1: REAL; side2: REAL
        perimeter: REAL
                        -- Rectangle-specific version
                do
                        Result := 2 * (side1 + side2)
                end
        ... Other RECTANGLE features ...
end
```

Here it is appropriate to redefine *perimeter* for rectangles as there is a simpler and more efficient algorithm. Note the explicit **redefine** subclause (which would come before the **rename** if present).

Signed by JV - 2026/10/8

Other descendants of *POLYGON* may also have their own redefinitions of *perimeter*. The version to use in any call is determined by the run-time form of the target. Consider the following class fragment:

```
p: POLYGON; r: RECTANGLE
... create p; create r; ...
if some_condition then
        p := r
end
print (p.perimeter)
```

The polymorphic assignment *p* := *r* is valid because the type of the source, *RECTANGLE*, conforms, through inheritance, to the type of the target, *POLYGON*. If *some_condition* is false, *p* will be attached to an object of type *POLYGON* for the computation of *p.perimeter*, which will thus use the polygon algorithm. In the opposite case, however, *p* will be attached to a rectangle; then the computation will use the version redefined for *RECTANGLE*. This is known as dynamic binding.

Dynamic binding provides high flexibility. The advantage for clients is the ability to request an operation (such as perimeter computation) without explicitly selecting one of its variants; the choice only occurs at run-time. This is essential in large systems, where many variants may be available; each component must be protected against changes in other components.

This technique is particularly attractive when compared to its closest equivalent in non-object-oriented approaches where you would need records with variant components, and **case** instructions to discriminate between variants. This means that every client must know about every possible case, and that any extension may invalidate a large body of existing software.

Redefinition, polymorphism and dynamic binding support a development mode in which every module is open and incremental. When you want to reuse an existing class but need to adapt it to a new context, you can always define a new descendant of that class (with new features, redefined ones, or both) without any change to the original. This facility is of great importance in software development, an activity which — whether by design or by circumstance — is invariably incremental.

The power of polymorphism and dynamic binding demands adequate controls. First, feature redefinition is explicit. Second, because the language is typed, a compiler can check statically whether a feature application *a.f* is valid, as discussed in more detail below. In other words, the language reconciles dynamic *binding* with static *typing*. Dynamic binding guarantees that whenever more than one version of a routine is applicable the *right* version (the one most directly adapted to the target object) will be selected. Static typing means that the compiler makes sure there is *at least one* such version.

This policy also yields an important performance benefit: the design of the inheritance mechanism makes it possible for an implementation to find the appropriate routine, for a dynamically bound call, in constant time.

Assertions provide a further mechanism for controlling the power of redefinition. In the absence of specific precautions, redefinition may be dangerous: how can a client be sure that evaluation of *p.perimeter* will not in some cases return, say, the area? Preconditions and postconditions provide the answer by limiting the amount of freedom granted to eventual redefiners. The rule is that any redefined version must satisfy a weaker or equal precondition and ensure a stronger or equal postcondition than in the original. In other words, it must stay within the semantic boundaries set by the original assertions.

The rules on redefinition and assertions are part of the Design by Contract theory, where redefinition and dynamic binding introduce subcontracting. *POLYGON*, for example, subcontracts the implementation of perimeter to *RECTANGLE* when applied to any entity that is attached at run-time to a rectangle object. An honest subcontractor is bound to honor the contract accepted

Signed by JV - 2026/10/8

by the prime contractor. This means that it may not impose stronger requirements on the clients, but may accept more general requests, so that the precondition may be weaker; and that it must achieve at least as much as promised by the prime contractor, but may achieve more, so that the postcondition may be stronger.

7.10 Combining genericity and inheritance

Genericity and inheritance, the two fundamental mechanisms for generalizing classes, may be combined in two fruitful ways.

The first technique yields **polymorphic data structures**. Assume that in the generic class *LIST* [*G*] the insertion procedure *put* has a formal argument of type *G*, representing the element to be inserted. Then with a declaration such as

pl: *LIST* [*POLYGON*]

the type rules imply that in a call *pl*.*put* (...) the argument may be not just of type *POLYGON*, but also of type *RECTANGLE* (an heir of *POLYGON*) or any other type conforming to *POLYGON* through inheritance.

The conformance requirement used here is the inheritance-based type compatibility rule: *V* can only conform to *T* if it is a descendant of *T*.

Structures such as *pl* may contain objects of different types, hence the name “polymorphic data structure”. Such polymorphism is, again, made safe by the type rules: by choosing an actual generic parameter (*POLYGON* in the example) based higher or lower in the inheritance graph, you extend or restrict the permissible types of objects in *pl*. A fully general list would be declared as

LIST [*ANY*]

where *ANY*, a Kernel Library class, is automatically an ancestor of any class that you may write.

The other mechanism for combining genericity and inheritance is **constrained genericity**. By indicating a class name after a formal generic parameter, as in

VECTOR [*G* –> *ADDABLE*]

you express that only descendants of that class (here *ADDABLE*) may be used as the corresponding actual generic parameters. This makes it possible to use the corresponding operations. Here, for example, class *VECTOR* may define a routine *plus* **alias** "+" for adding vectors, based on the corresponding routine from *ADDABLE* for adding vector elements. Then by making *VECTOR* itself inherit from *ADDABLE*, you ensure that it satisfies its own generic constraint and enable the definition of types such as *VECTOR* [*VECTOR* [*T*]].

Unconstrained genericity, as in *LIST* [*G*], may be viewed as an abbreviation for genericity constrained by *ANY*, as in

LIST [*G* –> *ANY*]

With these basic forms of genericity, it is not possible to *create* an instance of a formal generic type, for example an object of type *G* in *VECTOR* [*G*]. Indeed without further information we don’t know whether any creation procedures are available. To request specific ones for an actual generic parameter, list them in the class declaration, just after the constraint:

VECTOR [*G* –> *ADDABLE* **create** *make* **end**]

Then for *x* of type *G* you can use the instruction **create** *x*.*make* (*a*), with the appropriate argument type for *a* as specified for *make* in *ADDABLE*, and rely on the guarantee that

Signed by JV - 2026/10/8

when this gets applied to a *VECTOR* [*T*] for a permissible *T* this type will have its own appropriate version of *make*.

7.11 Deferred classes

The inheritance mechanism includes one more major component: deferred routines and classes.

Declaring a routine *r* as deferred in a class *C* expresses that there is no default implementation of *r* in *C*; such implementations will appear in eventual descendants of *C*. A class having one or more deferred routines is itself said to be deferred. A non-deferred routine or class is called **effective**.

For example, a system used by a Department of Motor Vehicles to register vehicles could include a class of the form

```
deferred class VEHICLE feature
        dues_paid (year: INTEGER): BOOLEAN
                do... end
        valid_plate (year: INTEGER): BOOLEAN
                do... end
        register (year: INTEGER)
                        -- Register vehicle for year.
                require
                         dues_paid: dues_paid (year)
                deferred
                ensure
                        valid_plate: valid_plate (year)
                end
        ... Other features, deferred or effective...
end
```

This example assumes that no single registration algorithm applies to all kinds of vehicle; passenger cars, motorcycles, trucks etc. are all registered differently. But the same precondition and postcondition apply in all cases. The solution is to treat *register* as a deferred routine, making *VEHICLE* a deferred class. Descendants of class *VEHICLE*, such as *CAR* or *TRUCK*, **effects** this routine, that is to say, give effective versions (figure 4). An effecting is similar to a redefinition; only here there is no effective definition in the original class, just a specification in the form of a deferred routine. There is no need here for a **redefine** clause; the effective versions simply take over any inherited deferred version. The term **redeclaration** covers both redefinition and effecting.

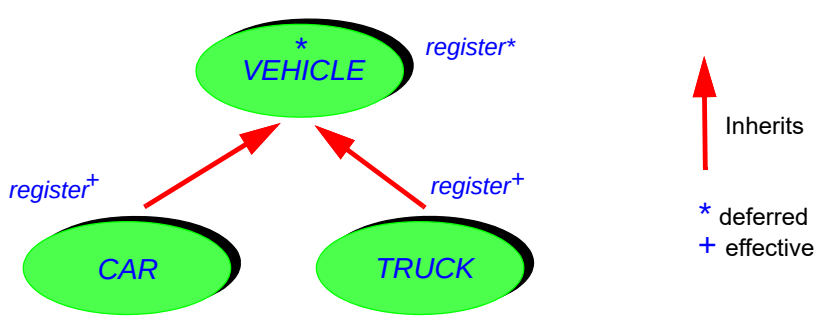


Figure 4: Abstracting variants into a deferred parent

Signed by JV - 2026/10/8

Deferred classes describe a group of implementations of an abstract data type rather than just a single implementation. You may not instantiate a deferred class: **create** *v* is invalid if *v* is a variable declared of type *VEHICLE*. But you may assign to *v* a reference to an instance of a non-deferred descendant of *VEHICLE*. For example, assuming *CAR* and *TRUCK* provide effective definitions for all deferred routines of *VEHICLE*, the following will be valid:

```
v: VEHICLE; c: CAR; t: TRUCK
...
create c ...; create t ...;...
if some_test then v := c else v := t end
v.register (1996)
```

This example fully exploits polymorphism: depending on the outcome of *some_test*, *v* will be treated as a car or a truck, and the appropriate registration algorithm will be applied. Also, "Some test" may depend on some event whose outcome is impossible to predict until run-time, for example a user clicking with the mouse to select one among several vehicle icons displayed on the screen.

Deferred classes are particularly useful at the **design** stage. The first version of a module may be a deferred class, which will later be refined into one or more effective (non-deferred) classes. Particularly important for this application is the possibility of associating a precondition and a postcondition with a routine even though it is a deferred routine (as with *register* above), and an invariant with a class even though it is a deferred class. This enables the designer to attach precise semantics to a module at the design stage, long before making any implementation choices.

These possibilities make Eiffel an attractive alternative to specialized notations, graphical or textual, for design and also for **analysis**. The combination of deferred classes to capture partially understood concepts, assertions to express what is known about their semantics, and the language's other structuring facilities (information hiding, inheritance, genericity) to obtain clear, convincing architectures, yields a higher-level design method. A further benefit, of course, is that the notation is also a programming language, making the development cycle smoother by reducing the gap between design and implementation.

At the analysis stage, deferred classes describe not software objects, but objects from the external reality's model — documents, airplanes, investments. The Eiffel mechanisms are just as attractive for such modeling.

An important property of deferred classes supporting all these lifecycle tasks, as part of a seamless software development cycle, is that they do not have to be *fully* deferred, like pure "interfaces". A class may have a mix of effective features capturing what is known at a certain stage of the development, and deferred ones describing what remains to be refined. This supports a continuous refinement-based process, proceeding smoothly from the abstract to the concrete.

7.12 Tuples and agents

A simple extension to the notion of class is the tuple. The type *TUPLE* [*A*, *B*, *C*] has, as its instances, sequences ("tuples") whose first three elements are of types *A*, *B* and *C* respectively. A tuple expression appears as simply [*a1*, *b1*, *c1*] with elements of the given type. It is also possible in the tuple type declaration to label the components, as in *TUPLE* [*x*: *A*; *y*: *B*; *z*: *C*], making it simpler to access the elements, as in *your_tuple*.*y*, with the proper type, here *B*, rather than *your_tuple*.*item* (2) of type *ANY* by default. Tuples provide a simpler alternative to classes when you don't need specific features, just a sequence of values of given types.

Tuples also help in the definition of **agents**. An agent is a way to define an object that represents a certain routine, ready to be called, possibly with some of its arguments set at the time of the agent definition (**closed operands**) and others to be provided at the time of each actual call (**open**

Signed by JV - 2026/10/8

operands). For example with a routine *r* (*x*: *A*; *y*: *B*) in a class *C*, as well as *a1* of type *A* and *c1* of type *C*, the agent

agent *c1*.*r* (*a1*, **?**)

represents the routine *r* ready to be called on the target *c1*, with the argument *a1* and another argument (corresponding to *y*, of type *B*) to be provided at the time of the call. The question mark **?** represents an open operand. The routine using this agent, for example having received it as an actual argument to a routine, for the formal argument *operation*, can then call the associated routine through

operation.*call* [*b1*]

for some *b1* of type *B*, the only open operand. This will have the same effect as an original call *c1*.*r* (*a1*, *b1*), but the routine executing it does not know that *operation* represents *r* rather than any other routine with the same expected open operands. The argument to *call* is not directly *b1* but a **tuple** with *b1* as its sole item; this is because *call* — a routine of the corresponding general-purpose agent class in the Kernel Library — must be able to accept argument sequences of any length, while ensuring type safety.

Agents add a further level of expressiveness to the mechanisms discussed earlier. They are particularly useful for numerical applications — for example to pass **agent** *f*, where *f* is a mathematical function, to an integration routine — and Graphical User Interface (GUI) applications, where they provide an attractive alternative to techniques such as "function pointers" and the Observer Pattern. For example an application may pass the above **agent** *c1*.*r* (*a1*, **?**) to a GUI routine, telling it to "subscribe" the associated operation to certain event types such as mouse click. When such an event occurs, the GUI will automatically trigger the operation through a *call*.

7.13 Type- and void-safety

As noted, Eiffel puts a great emphasis on reliability through static typing. Polymorphism and dynamic binding in particular are controlled by type rules. The basic requirement is simple: an assignment of the form *a* := *b* is permitted only if *a* and *b* are of reference types *A* and *B*, based on classes *A* and *B* such that *B* is a descendant of *A*. The same applies to argument passing.

This corresponds to the intuitive idea that a value of a more specialized type may be assigned to an entity of a less specialized type — but not the reverse. (As an analogy, consider that if you request vegetables, getting green vegetables is fine, but if you ask for green vegetables, receiving a dish labeled just "vegetables" is not acceptable, as it could include, say, carrots.)

This inheritance-based type system rules out numerous errors, some potentially catastrophic if their detection was left to run time.

Another, just as dangerous type of error has generally eluded static detection: *void calls*. In a framework offering references or pointers, the risk exists that in the execution of a call *x*.*f* (...) the value of *x* will be void rather than attached to an object, leading to a crash. The design of Eiffel treats this case through the type system. Specifically:

- Types are by default **attached**, meaning that they do not permit void values. To support void, a type must be declared as **detachable** *T* rather than just *T*.
- You may use a qualified call *x*.*f* (...) only if the type of *x* is attached.
- As part of the conformance rules, you may assign a *T* source to a **detachable** *T* target but (for risk of losing the characteristic property of attached types) not the other way around.
- All entities, as noted, are initialized to default values. For detachable types the default value is void, but for an attached type it must always be attached to an object. This means that either the type must provide a default creation procedure (a procedure *default_create* from class *ANY* is available for that purpose, which any class can use as creation procedure, after possibly redefining it to suit its needs), or every variable must be explicitly initialized before use.

Signed by JV - 2026/10/8

These simple rules, compiler-enforceable, remove a whole category of tricky and dangerous failures.

7.14 Putting a system together

This discussion has focused so far on individual classes. This is consistent with the Eiffel method, which emphasizes reusability and, as a consequence, the construction of autonomous modules.

To execute software, you will need to group classes into executable compounds. Such a compound is called a **system** — the Eiffel concept closest to the traditional notion of program — and is defined by the following elements:

- A set of classes, called a **universe**.
- The designation of the system's **root type**, based on one of these classes (the "root class").
- The designation of one of the creation procedures of the root class as the **root procedure**.

To execute such a system is to create one direct instance of the root type (the "root object" for the current execution), and to apply to it the root procedure — which will usually create other objects, call other routines and so on.

The method suggests grouping related classes — typically 5 to 40 classes — into collections called **clusters**. A common convention, for the practical implementation on the file system, is to store each class in a file, and associate each cluster with a directory. Then the universe is simply the set of classes stored across a set of directories.

The classes of a system will include its root class and all the classes that it **needs** directly or indirectly, where a class is said to need another if it is one of its heirs or clients.

To specify a system you will need to state, in addition to the list of directories, the root type and the root procedure (which must be one of the creation procedures of the root class). This is achieved through either a graphical interface or a control file.

End

Signed by JV - 2026/10/8

PART C: General conventions

Signed by JV - 2026/10/8

Signed by JV - 2026/10/8

8 Language specification

Signed by JV - 2026/10/8

8.1 General organization

Informative Text

8.1.0.1 After parts A (preliminaries) and B (tutorial), both not formally part of the Standard, the remainder of the text provides the precise specification of Eiffel. After a presentation of the conventions, the overall order of the language description is top-down: it first covers the constructs governing the global structure of software systems, then goes in ever more detail into computational mechanisms.

8.1.0.2 Part C introduces the overall conventions:

8.1.0.2.1 • Conventions for the **language description** (sub-subclause 8.2, page 58),

8.1.0.2.2 • Conventions of the **Eiffel language** itself (8.3, page 70).

8.1.0.3 Part D covers Eiffel's structural mechanisms:

8.1.0.3.1 • Elementary concepts of the language and **architecture** of Eiffel software, including the principal structuring mechanisms: cluster, class, feature, inheritance, client (8.4, page 76 to 8.7, page 87).

8.1.0.3.2 • Key elements of a class: **features** and (specifically) routines (8.8, page 99 to 8.9, page 121).

8.1.0.4 Part E presents a fundamental part of the Eiffel approach: assertions and support for **contracts** (8.10, page 128).

8.1.0.5 Part F is devoted to **inheritance** and the corresponding **feature adaptation** mechanisms, including redeclaration, genericity, tuples, conformance, convertibility and repeated inheritance (8.11, page 143 to 8.13, page 159).

8.1.0.6 Part G describes Eiffel's sophisticated **type system**, largely based on inheritance but also including genericity, tuples, conformance and conversion (8.14, page 165 to 8.18, page 194).

8.1.0.7 Part H goes into Eiffel's **computational mechanisms**:

8.1.0.7.1 • Calls, agents, expressions, control structures (8.19, page 200 to 8.22, page 224).

8.1.0.7.2 • Data structures and dynamic model: the object model, object creation, comparison, void safety (8.23, page 231 to 8.26, page 251)

8.1.0.8 Part I complements the description by presenting:

8.1.0.8.1 • **Advanced** mechanisms: void safety, advanced typing issues, concurrency (SCOOP), exception handling (8.30, page 254 to 8.33, page 260).

8.1.0.8.2 • **Elementary** mechanisms: constants, basic types, lexical elements (8.34, page 266 to 8.37, page 279).

8.1.0.9 Part J is **reference material**: full collected syntax specification, both in the order of the preceding clauses (8.9, page 290) and alphabetically (8.10, page 299), and Index, page 308.

End

Signed by JV - 2026/10/8

8.2 Language description conventions: syntax, validity, semantics

Informative Text

8.2.0.1 *Document terminology note*: the reader not familiar with the conventions of Ecma International standards should note that they do not refer to divisions of documents such as this Standard as "chapters", "sections" etc. but as **clauses** for second-level divisions (such as the one, 8.2, containing the present discussion) and **subclauses** for any of their subdivisions, at the third or higher level (such as the principle 8.2.1 below).

8.2.0.2 The present subclause describes the method and notation that the rest of the Standard uses to define the Eiffel language.

8.2.0.3 It provides no information about the Eiffel language itself, but only the *notation that the Standard uses to specify Eiffel* — which could also be used to specify languages other than Eiffel.

8.2.0.4 Such a language-defining notation is known as a **metalanguage**. For example the terms "*component*", "*construct*" and "*specimen*" introduced in the second definition below (D2CS) describe concepts of the metalanguage, not of Eiffel.

8.2.0.5 The conventions regarding the general properties of the Eiffel language itself are the subject of the following subclause (8.3).

8.2.0.6 *Document terminology note*: in this Standard, the terms "clause" and "subclause" may denote:

8.2.0.6.1 • An Eiffel construct, as in "*The* Rename *clause*", a phrasing used for example in SHRC, page 147 — in meta-langage.

8.2.0.6.2 • A division of this Standard itself, per the Ecma International terminology as noted above (8.2.0.1), as in "*the semantics of renaming is specified by the* Renaming principle*, appearing in subclause* 8.11.17" — a case of what could be called "meta-metalanguage" since it talks about a document that talks about Eiffel.

8.2.0.7 No confusion may result since:

- In the first usage the word "clause" or "subclause" is *always* associated with the name of an Eiffel construct (always a hyperlink to the syntactical definition of that construct), such as Rename in the first example.
- In the second case it is *always* followed by the number of a clause in this Standard, such as 8.11.17 in the second example.

End

8.2.1 Principle: Subclause classification and codes

Code: *P2CL*

8.2.1.1 The core of the language description appears in subclauses with a third-level number (such as the present one 8.2.1) appearing in the main clause of the Standard (the present clause 8).

8.2.1.2 The nature of every such third-level subclause is not arbitrary but strictly codified. There are five categories, comparable to the types of a programming language:

8.2.1.2.1 • Principle.

8.2.1.2.2 • Definition.

8.2.1.2.3 • Syntax (this concept is explained in 8.8.2.3 below).

8.2.1.2.4 • Validity (explained in 8.8.2.16 below). In the computer science and programming language literature, validity is sometimes called "static semantics".

8.2.1.2.5 • Semantics (this concept is explained in 8.8.2.19 below),

8.2.1.3 Each third-level subclause has a **four-character code** identifying it uniquely in the Standard, such as P2CL for the present principle. Other occurrences of such codes, as on the preceding line, are (in the electronic version) hyperlinks to the corresponding sublause. All the codes also appear in the Index starting on page 308.

Signed by JV - 2026/10/8

8.2.1.4 Compilers and other language processing tools may (and should) refer to these codes, in particular for error messages. For example, if a programmer mistakenly chooses for a new class a Class_name that is already used by an existing class in the universe (accounting for a possible variation of letter case), an Eiffel compiler should produce an error message which will refer to the corresponding rule, the Class Name rule (see page 79) by giving its code, VSCN, enabling the author of the code to understand quickly and precisely what property of the language is at play. (The rule states: "It is valid for a universe to include a class if and only if no other class of the universe has the same upper name.").

8.2.1.5 The makeup of the four-letter codes is the following;

8.2.1.5.1 • The first character is one of: P for principle; D for definition; S for syntax; V for validity; M for semantics ("Meaning").

8.2.1.5.2 • The second character identifies the second-level subclause of the Standard in which the rule appears (each such subclause has a single identification, a letter or a digit). In the VSCN example, the clause is part of the subclause on "The architecture of Eiffel Software" (8.5) which has code S (mnemonic for Structure).

8.2.1.5.3 • The last two characters are a mnemonic of the subclause name, such as CN for Class Name in the VSCN example.

8.2.1.5.4 • In that VSCN example, then, the code decomposes as follows: V for Validity; S for the subclause ("structure"); CN as a mnemonic for Class Name.

8.2.1.5.5 • Similarly, the code for the present subclause, P2CL, has P for Principle, 2 as the identifier of the parent second-level subclause (8.2, Language description conventions: syntax, validity, semantics), and CL as a mnemonic for CLassification. (The identifier of a second-level subclause can be any letter or digit, such as C in the previous example and 2 in the present one. That 2 is also the second-level subclause number of subclause 8.2 is just a coincidence.)

8.2.2 Principle: Hyperlinking

Code: *P2HL*

8.2.2.1 The Standard in its electronic version is richly hyperlinked. The following are all hyperlinks, such that clicking on one of them will take the reader to the referenced part of the Standard:

8.2.2.1.1 • Any subclause code, such as P2CL, referencing the corresponding subclause.

8.2.2.1.2 • Any construct name (as defined below), such as Class_declaration, referencing the syntactic definition of the construct, in this case on page 82. (Obvious exception: the occurrence of the construct name in that definition, such as Class_declaration in the syntax rule defining this construct on page 82, is not hyperlinked to itself.)

8.2.2.1.3 • Any underlined text, referencing the subclause in which one may find the definition or explanation of the underlined concept.

Informative Text

8.2.2.2 The Standard contains several thousand hyperlinks. The intent is to facilitate quick retrieval of interconnected definitions and easy navigation through Eiffel's many concepts

End

8.2.3 Definition: Syntax, BNF-E

Code: *D2BN*

Syntax is the set of rules describing the structure of software texts.

The notation used to define Eiffel's syntax is called **BNF-E**.

Informative Text

8.2.3.1 "BNF" is *Backus-Naur Form*, a traditional technique for describing the syntax of a certain category of formalisms ("*context-free languages*"). Originally introduced for the description of the Algol 60 programming languages, BNF has several variants in practice. BNF-E is a form of BNF intended to make descriptions clear and modular, thanks in particular to the following conventions:

8.2.3.1.1 • Every construct is defined by **exactly one rule** (which we call a "production" starting with D2PR, page 63 below).

8.2.3.1.2 • The definition of a construct involving a **repetition** with a given separators (for example Entity_list, describing lists of one or more Identifier separated by commas, such as *a*, *b*, *c*) uses a specific notation designed for clarity and concision, presented in D2RS, page 64 below.

8.2.3.2 The range of formalisms (context-free languages) that can be described by BNF-E is the same as for traditional BNF.

End

8.2.4 Definition: Component, construct, specimen

Code: *D2CS*

Any class text, or syntactically meaningful part of it such as an instruction, an expression or an identifier, is called a **component**.

The structure of all components of a particular kind is described by a **construct**.

A component of a kind described by a certain construct is called a **specimen** of that construct.

Informative Text

8.2.4.1 For example, any particular class text, built according to the rules given in this language description, is a *component*. The *construct* Class_declaration describes the structure of class texts; any class text is a *specimen* of that construct.

8.2.4.2 As another example, covering finer-grain components, an identifier such as *your_variable* is a specimen of the construct Identifier.

8.2.4.3 Although the term "instance" could be used in lieu of "specimen", it could cause confusion with the instances of an Eiffel type — the run-time objects built according to the specification of the type (and its base class). To avoid any ambiguity:

8.2.4.3.1 • A construct belongs to the meta-language (the language description); its *specimens* appear in software texts.

8.2.4.3.2 • A class is one such specimen: a specimen of the construct Class_declaration, appearing in such a software text. Its *instances* will appear during execution.

End

8.2.5 Principle: Construct Specimen convention

Code: *P2SP*

In the Standard, the phrase ''an X", where X is the name of a construct, serves as a shorthand for ''a specimen of X" .

Informative Text

8.2.5.1 This principle applies not to program texts but to the language description in the present Standard. It presents a convention that the Standard uses consistently for specifying language constructs.

8.2.5.2 For example, ''a Class_declaration'' means ''a specimen of construct Class_declaration'': a text built according to the syntactical specification (page 82) of the construct Class_declaration. Another typical example is the Semantics rule MCNO, page 84, which states: "A Notes part has no effect on system execution". Here the phrase "a Notes part" means "a specimen of the Notes construct, that is to say, a part of the class text built according to that specification (which indicates that it consists of the keyword **note** followed by any number of "note entries" separated by optional semicolons).

8.2.5.3 **Every such occurrence of a construct name in the text** is a hyperlink to the production (syntax rule) defining the syntax of the corresponding construct. For example if you click on any of the construct names in the previous paragraph (the names appearing in green and underlined) you will be taken to the syntactical definition of, respectively, class declarations or "notes" parts, in the corresponding subclauses of the Standard.

Signed by JV - 2026/10/8

8.2.5.4 The convention was already used informally above, in the phrase “one or more Identifier separated by commas” in the Informative Text following D2BN, page 59.

End

8.2.6 Principle: Construct Name convention

Code: *P2CN*

Every construct has a name starting with an upper-case letter and continuing with lower-case letters, possibly with underscores (to separate parts of the name if it uses several English words).

Informative Text

8.2.6.1 This principle applies not to program texts but to the language description in the present Standard. It presents a convention that the Standard uses consistently for specifying language constructs.

8.2.6.2 Typesetting conventions complement it: construct names, such as Class_declaration, always appear in Roman and in Green — distinguishing them from the blue of Eiffel text, as in **Result** := *x*.

End

8.2.7 Definition: Terminal, non-terminal, token

Code: *D2TT*

Every construct is either **non-terminal** or **terminal**.

The specimens of a *non-terminal* construct are defined in terms of other constructs.

The specimens of a *terminal* construct, called **tokens** (also, “lexical components”), have no further syntactical structure. Examples of tokens include:

- Reserved words such as **if** and **Result**.
- Manifest constants such as the integer *234*.
- Typographical symbols such as **;** (semicolon) and **+** (plus sign).
- Identifiers (used to denote classes, features, entities) such as *LINKED_LIST* and *put*.

The specimens of terminal constructs are called **tokens**.

Informative Text

8.2.7.1 “Reserved words” include (per their definition on page 282) the special case of *keywords* such as **class** and **if**, which (in contrast with other reserved words such as **Result**), have no meaning of their own but just delimit various parts of the software text.

8.2.7.2 The distinction between non-terminals and terminals reflects the property that program texts, made only of tokens, have a structure, which can be represented by a tree; non-terminal constructs describe the internal nodes of the tree, while the tokens are at the leaves. For example the simple class text

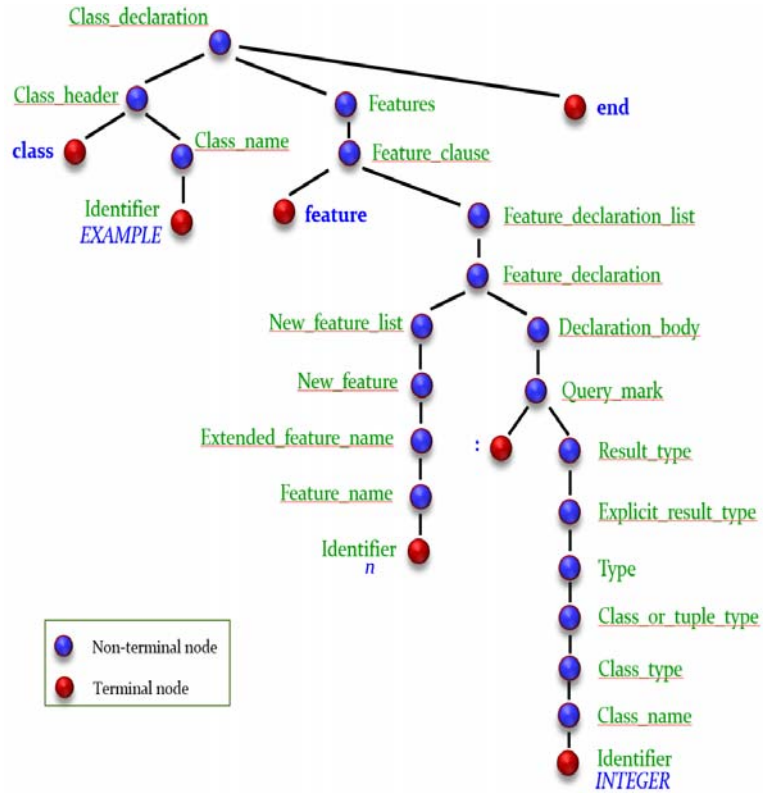
8.2.7.3

```
class EXAMPLE feature
        n: INTEGER
end
```

8.2.7.4 is made of the successive tokens **class** (keyword), *EXAMPLE* (identifier), **feature** (keyword), *n* (identifier), colon (typographical symbol), *INTEGER* (identifier) and **end** (keyword). The tree illustrated above represents the syntax structure underlying this text.

Signed by JV - 2026/10/8

8.2.7.5



8.2.7.6 This tree represents the text as a specimen of the construct Class_declaration built according to the corresponding syntax rule (SCCD, page 82) which specifies that the components of a Class_declaration are, in order:

8.2.7.6.1 • An optional specimen of the Notes construct, not present here.

8.2.7.6.2 • A Class_header which consists, per to its own rule (SCCH, page 84), of the keyword **class** followed by a Class_name, itself defined (SSCN, page 78) as an Identifier, here *EXAMPLE*.

8.2.7.6.3 • Specimens of other constructs such as Formal_generics, all optional and not present in this example (see the full list of possibilities in the rule for Class_declaration).

8.2.7.6.4 • Yet another optional component, present here: a specimen of Features. It involves (per its syntax rule SFPA, page 101) one or more specimens of Feature_clause, here just one, itself starting with the keyword **feature** followed (other than optional components not present here) by a Feature_declaration_list, consisting of one or more Feature_declaration, here just one, the declaration for the attribute *n*, with a further structure that can be seen by following the rule for the components of Feature_declaration (SFDE, page 102) and is shown in the corresponding nodes in the bottom-right part of the tree.

8.2.7.6.5 • Further optional components such as a specimen of Invariant not present here (again, see the rule for Class_declaration, SCCD, page 82, for their list).

8.2.7.6.6 • The keyword **end**.

8.2.7.7 As this example indicates, software texts consist solely of specimens of terminal constructs (tokens); non-terminal constructs serve to describe the underlying syntactical structure.

End

8.2.8 Definition: Production

Code: *D2PR*

A **production** is a formal syntax rule specifying the structure of all specimens of a non-terminal construct. It has the form

Construct ≜ right-side

where right-side describes how to obtain specimens of the Construct under definition.

Informative Text

8.2.8.1 The symbol ≜ may be read aloud as "*is defined as*". It belongs to the metalanguage (the language for describing the properties of Eiffel, specifically in this case the syntax of Eiffel) and has no connection whatsoever to the equal sign as used in the Eiffel language itself (for boolean expressions), written = and appearing in blue in the Standard.

8.2.8.2 BNF-E uses **exactly one production for each non-terminal**. The reason for this convention is explained below.

8.2.8.3 As noted earlier, every construct appearing as part of the right-side is hyperlinked to its own definition — that is to say the production (there is only one in the Standard for every construct) in which it appears on the left side of the "≜".

End

8.2.9 Definition: Kinds of production

Code: *D2PK*

A production defining a construct *C* is of one of the following three kinds, distinguished by the form of its right side:

- **Aggregate**, specifying *C* by concatenation: a specimen of *C* is a sequence of specimens of given constructs, some of them possibly optional.
- **Choice**, specifying *C* by its variants: a specimen of *C* is a specimen of *one* of a set of given constructs.
- **Repetition**: a specimen of *C* is a sequence (of variable length) of specimens of a given construct.

Informative Text

8.2.9.1 The three kinds are defined below. Informally, a specimen of the construct under definition is:

8.2.9.1.1 - In the aggregate case: the concatenation of specimens of specified constructs.

8.2.9.1.2 - In the choice case: a specimen of one of a number of specified constructs.

8.2.9.1.3 - In the repetition case: the concatenation of zero or more (or one or more) specimens of one specified construct, separated by specimens of a specified "separator" construct.

End

8.2.10 Definition: Aggregate production

Code: *D2AG*

An **aggregate** right side is of the form $C_1\ C_2\ ...\ C_n$ ($n > 0$), where every one of the C_i is a construct, with the possibility for any mutually disjoint subsequences to be each enclosed in square brackets, thus appearing as $[C_i\ ...\ C_j]$ for $1 \le i \le j \le n$.

It specifies that every specimen of the corresponding construct consists of a specimen of C_1, followed by a specimen of C_2, ..., followed by a specimen of C_n, with the provision that for any subsequence in brackets the corresponding specimens may be absent.

Informative Text

8.2.10.1 As an example, the production defining Feature_clause (SFPA, page 101) uses the right side

8.2.10.2 **feature** [Clients] [Header_comment] Feature_declaration_list

Signed by JV - 2026/10/8

8.2.10.3 expressing that a specimen of Feature_clause consists, in order, of:

8.2.10.3.1 • The keyword **feature**.

8.2.10.3.2 • Then, optionally, a specimen of the construct Clients.

8.2.10.3.3 • Then, again optionally, a specimen of Header_comment.

8.2.10.3.4 • Finally, a specimen of Feature_declaration_list. This last component, like the initial keyword, is not optional but always required.

8.2.10.4 The above definition of "aggregate production" indicates that optionality applies in the general case not just to one constituent construct, as in [C_i] (for example [Clients]), but to a subsequence of constructs, as [C_i ... C_j]. There is a difference between:

8.2.10.4.1 • [C_i] ... [C_j], prescribing that each of the corresponding specimens may — independently of each other — be present or absent at the corresponding position (between i and j).

8.2.10.4.2 • [C_i ... C_j], (much less common in actual productions of the Standard), prescribing that the corresponding subsequence of components may, in its entirety, be present or absent.

End

8.2.11 Definition: Choice production

Code: *D2CH*

A **choice** right side is of the form C_1 | C_2 | ... | C_n (n > 1), where every one of the C_i is a construct.

It specifies that every specimen of the corresponding construct consists of exactly one specimen of one of the C_i.

Informative Text

8.2.11.1 As an example, the production defining Feature_body (SFDF, page 113) uses the right side

8.2.11.2 Deferred | Effective

8.2.11.3 expressing that a specimen of Feature_body consists of a specimen of either Deferred or Effective.

End

8.2.12 Definition: Repetition production, separator

Code: *D2RS*

A **repetition** right side is of one of the two forms

{C $\S$ ...}*

{C $\S$ ...}+

where C and $\S$ (the **separator**) are constructs. The only difference between these two variants is the final character, an asterisk in the first case and a plus sign in the second one.

It specifies that a specimen of the corresponding construct consists of zero or more (one or more in the second form) specimens of C, each separated from the next, if any, by a specimen of $\S$.

The following abbreviations may be used if the separator $\S$ is empty:

C*

C+

Informative Text

8.2.12.1 As an example, the production defining New_feature_list (SFNL, page 102) uses the right side

8.2.12.2 {New_feature_name "," ...}+

8.2.12.3 expressing that a specimen of New_feature consists of one or more successive specimens of New_feature, separated (if more than one) by commas. Because of the "+" sign, such a list cannot be empty.

Signed by JV - 2026/10/8

8.2.12.4 If a repetition production uses an asterisk "*" instead, the list can be empty, as in the definition of Tag_list (SAAS, page 128), whose right side uses

{Tag "," …}*

8.2.12.5 expressing that a specimen of Tag_list can either be empty or consist of one or more successive specimens of Tag, separated (if more than one) by commas.

8.2.12.6 Traditional BNF specifications tend to rely heavily on recursive definitions for repetitive constructs. For example one might define Tag_list though a choice between three possibilities: Empty; Tag; and Tags, an aggregate construct defined recursively as Tags "," Tag. Thanks to the Repetition production of BNF-E, the language definition avoids such uselessly complicated definitions and only uses recursion for valuable purposes.

8.2.12.7 These remaining uses of recursive definitions (generally involving *indirect* recursion) serve to describe constructs with unbounded nesting potential. A typical example is the set of mutually recursive productions defining forms of Instruction that themselves involve instructions:

8.2.12.7.1 • Instruction is defined by a Choice production, SRIN, page 124, listing a number of kinds of instructions.

8.2.12.7.2 • Among these variants, we find Conditional_instruction (SOCI, page 224) and Loop (SOLO, page 228).

8.2.12.7.3 • Both of them have components of type Instruction, such as the Then_part of a Conditional_instruction or the Loop_body of a Loop, causing indirect recursion in the syntax definition.

8.2.12.8 Such uses of recursion are conceptually inevitable. For specifying repetitive constructs, however, Repetition productions are simpler and clearer.

End

8.2.13 Principle: Basic Syntax Description rule

Code: *P2SD*

Every non-terminal construct is defined by exactly one production.

Informative Text

8.2.13.1 This principle applies not to program texts but to the language description in the present Standard. It presents a convention that the Standard uses consistently for specifying language constructs.

8.2.13.2 Traditional BNF usually does not follow this principle; it allows mixing the different mechanisms in a single production, with a right side such as B | (C ; D) combining choice and aggregation. In contrast with this practice, every BNF-E production always uses **exactly one** of Aggregate, Choice and Repetition, *never* mixing them in the right sides.

8.2.13.3 This convention yields a considerably clearer grammar, even if it implies a few more productions — which in the end is good since they give a more accurate image of the language's structure.

End

8.2.14 Definition: Non-production Syntax rule

Code: *D2NP*

A **non-production syntax rule**, marked "(*non-production*)", is a syntax property expressed outside of the BNF-E formalism. Such rules are marked "Syntax (non-production)".

Informative Text

8.2.14.1 Non-production rules describe syntactical properties that are impossible or inconvenient to specify in BNF style.

8.2.14.2 For example, BNF-E Aggregate productions allow successive right-side components to be separated by an arbitrary break — any sequence of spaces, tabs and "new line" characters. In the case of an Alias declaration (SFNA, page 104), such as **alias** "+", the language *prohibits* breaks between the operator and either one of the double quotes. The specification of this construct still

Signed by JV - 2026/10/8

uses a BNF-E production (an Aggregate production prescribing the keyword **alias** followed by a double quote, an operator and another double quote), but complements it by expressing the extra restriction in natural language: SFAL, page 105, labeled "**Syntax (non-production)**".

8.2.14.3 Unlike validity rules (see next), non-production syntax rules belong to the syntax, that is to say the description of the structure of Eiffel texts, but they capture properties that are not expressible, or not conveniently expressible, through a context-free grammar.

End

8.2.15 Principle: Textual Conventions

Code: *P2TX*

The syntax (BNF-E) productions and other rules of the Standard apply the following conventions:

1 Symbols of BNF-E itself (the "metalanguage"), such as the vertical bars | signaling a choice production, appear in black — non-bold, non-italic.

2 Any construct name appears in dark green (non-bold, non-italic), with a first letter in upper case, as Alias.

3 Any component (Eiffel text element) appears in blue.

4 The double quote, one of Eiffel's special symbols, appears in productions as '"': a double quote character (in blue, as it is a component) enclosed in two single quote characters (in black since they belong to the BNF-E metalanguage, not Eiffel).

5 All other special symbols appear (in blue, since they are components) in double quotes, for example a comma as ",", an assignment symbol as ":=", a single quote as "'" (double quotes black, single quote blue).

6 Keywords and other reserved words, such as **alias** and **Result**, appear in **bold** (in blue, since they are components). They do not require quotes since the conventions avoid ambiguity with construct names: Alias is the name of a construct, **alias** a keyword.

7 Examples of Eiffel comment text appear in non-bold, non-italic (in blue, as they are components), as in -- A comment.

8 Other elements of Eiffel text, such as entities and feature names (including in comments) appear in non-bold *italic* (in blue, as they are components).

The color-related parts of these conventions do not affect the language definition, which remains unambiguous under black-and-white printing (thanks to the letter-case and font parts of the conventions). Color printing is recommended for readability.

Informative Text

8.2.15.1 This principle applies not to program texts but to the language description in the present Standard. It presents a convention that the Standard uses consistently for specifying language constructs.

8.2.15.2 Because of the difference between cases 5 and 1, "**{**" denotes the opening brace as it might appear in an Eiffel class text, whereas { is a symbol of the syntax description, used in Repetition productions.

8.2.15.3 In case 2, the use of an upper-case first letter is a consequence of the "Construct Name convention" seen earlier in this subclause (P2CN, page 61).

8.2.15.4 Special symbols are normally enclosed in double quotes (case 5), except for the double quote itself which, to avoid any confusion, appears enclosed in single quotes (case 4). In either variant, the enclosing quotes — double or single respectively — are not part of the symbol.

8.2.15.5 In some contexts, such as the table of all such symbols, special symbols (cases 4 and 5) appear in bold for emphasis.

8.2.15.6 In application of cases 7 and 8, occurrences of Eiffel entities or feature names in comments appear in italics, to avoid confusion with other comment text, as in a comment

-- Update the value of *value*.

Signed by JV - 2026/10/8

8.2.15.7 where the last word denotes a query of name *value* in the enclosing class.

End

8.2.16 Definition: Validity constraint

Code: *D2VC*

A **validity constraint** on a construct is a requirement that every syntactically well-formed specimen of the construct must satisfy to be acceptable as part of a software text.

Informative Text

8.2.16.1 The concept of validity constraint leads to the definition of what it means for a component of a software text to be valid, as defined immediately next (D2VA). The explanation of *why* we need validity constraints (in addition to syntax productions) appears below after that definition.

End

8.2.17 Definition: Valid

Code: *D2VA*

A construct specimen, built according to the syntax structure defined by the construct's production, is said to be **valid**, and will be accepted by the language processing tool of any Eiffel environment, if and only if it satisfies the validity constraints, if any, applying to the construct.

Informative Text

8.2.17.1 Validity constraints complement the BNF-E syntax descriptions. They play a fundamental role in the definition of the language by capturing rules that Eiffel texts must satisfy to represent meaningful computations, but that context-free rules such as those of BNF-E are unable to specify.

8.2.17.2 As an example, the production defining assignments (Assignment construct, SBAS, page 239) reads

Assignment $\triangleq$ Variable ":=" Expression

8.2.17.3 specifying that an Assignment instruction is of the form *some_variable* := *some_expression*. But writing a Variable followed by a colon, an equal sign then an Expression is not enough to produce a meaningful Assignment; for example, *n* := *3.5* where *n* is an *INTEGER* variable is not acceptable because the type of the Expression (*REAL*) is not permitted for assignment to an integer target (precision would be lost).

8.2.17.4 Such an Assignment is *syntactically correct* but not *valid*. The applicable validity rule, which an Assignment must satisfy in order to be given a meaning (semantics), is the Assignment rule, VBAS, page 239, which states:

> An Assignment is valid if and only if its source Expression is compatible with its target variable.

(where "compatible with" is a notion defined precisely in DNCE, page 189). Many syntax definitions similarly have an associated validity rule.

8.2.17.5 Validity rules define the "static semantics" of the language, so called because it lies between syntax and semantics:

8.2.17.5.1 • Like syntax, it is "*static*", in other words defines properties of software texts, not of their possible executions.

8.2.17.5.2 • Like semantics (which is "*dynamic*", describing execution properties) it defines properties of programs going beyond their mere structure and having to do with their intended meaning.

Signed by JV - 2026/10/8

8.2.17.6 The use of validity rules to complement syntax productions is not a matter of convenience but an absolute necessity: BNF-E, like other BNF variants, only covers certain properties of a language, the so-called context-free properties; it is not powerful enough to capture important practical properties, such as type rules. Validity constraints fill the gap.

End

8.2.18 Validity: General Validity rule

Code: *VBGV*

Every validity constraint relative to a construct is considered to include an implicit supplementary condition stating that every component of the construct satisfies every validity constraint applicable to the component.

Informative Text

8.2.18.1 This rule is a **fundamental convention of the language specification**, keeping it simple and coherent. It avoids expressing conditions on components that follow from the validity of these components and would have to be repeated for every production using them.

8.2.18.2 As an example, consider again the validity rule on Assignment (Assignment rule, VBAS, page 239) cited above:

> An Assignment is valid if and only if its source Expression is compatible with its target variable.

8.2.18.3 In practice, the condition given is not sufficient (for example, for the writer of a compiler) to ascertain that an Assignment *some_variable* := *some_expression* is valid: it must also be the case that *some_variable* is a valid Variable and *some_expression* is a valid Expression, according to the respective validity rules:

8.2.18.3.1 • For the *some_variable* part, the Variable rule, VEVA, page 246, specifying under what conditions an Identifier can be used as a variable.

8.2.18.3.2 • For the *some_expression* part, since Expression is defined (SWEX, page 215) by a Choice production between a number of specific expression constructs, the validity rule for the construct corresponding to the given *some_expression*.

8.2.18.4 Thanks to the Geneval Validity rule, the Assignment rule makes no reference to these conditions. Instead, it automatically includes the validity rules on all of the components of an Assignment. Compilers and other language processing tools must of course check the validity of the components as part of checking the validity of the Assignment as a whole.

End

8.2.19 Definition: Semantics

Code: *D2SM*

The **semantics** of a construct specimen that is syntactically legal and valid is the construct's effect on the execution of a system that includes the specimen.

Informative Text

8.2.19.1 Semantic definitions provide the last and highest of the four levels of description of the language, illustrated below. From the bottom (most elementary properties) up (to the most advanced properties) they are:

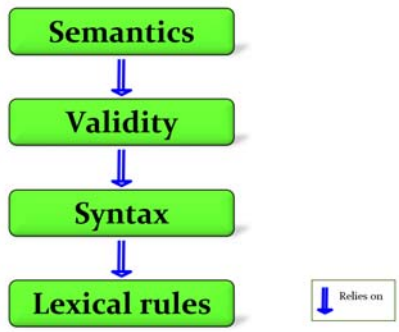
8.2.19.1.1 • **Lexical specification**, specifying the basic components (the tokens).

8.2.19.1.2 • **Syntax**, specifying the structure of software texts.

8.2.19.1.3 • **Validity**, specifying consistency conditions.

Signed by JV - 2026/10/8

8.2.19.1.4 • **Semantics**, specifying the resulting effect (language processing tool behavior).



8.2.19.2 Specifications at the lowest three levels are of a mathematical-style. Semantic specifications, while also precise, generally use a less formal style. They are of course a key element of the language description since they express the effect of software texts.

8.2.19.3 As the arrows in the figure indicate, each level relies on the previous one and hence indirectly on all of the ones below it. This observation means both that:

8.2.19.3.1 • Specifications at each level *refer* to elements introduced at the underlying levels. For example, validity rules express properties of syntactical constructs.

8.2.19.3.2 • Specifications at each level also *assume* the soundness properties at the preceding levels. Syntax rules are only meaningful for syntactical structures made of lexically correct tokens. Validity rules are only meaningful for structures built according to syntax productions (and hence are lexically correct). Semantic rules, which describe how software executes, are only meaningful for structures that satisfy the validity rules (and hence are syntactically and lexically correct).

End

Signed by JV - 2026/10/8

8.3 Conventions for Eiffel systems

Informative Text

8.3.0.1 The present subclause describes basic conventions governing Eiffel texts, such as the influence of letter case, the different forms and general properties of an identifier, and the conditions under which semicolons are optional.

8.3.0.2 The first few rules and definitions rely on the notion of letter case (lower case, upper case), defined in the specification of Eiffel's lexical properties (DILE, page 279). They are relative to the ASCII (7-bit) and extended ASCII (8-bit) character set. 7-bit ASCII covers (among other symbols) the lower- and upper-case forms of the 26 letters of the Roman alphabet. 8-bit ASCII makes it possible to include (among others) other letters with diacritical marks, such as the French letters *à* and *À*.

8.3.0.3 Interpretations of extended-ASCII codes differ depending on the language (in other words, different languages such as French and Swedish may use the same extended-ASCII codes for their own letters with diacritics, different in each case). Correspondingly, DILE, page 279 only defines lower case and upper case for 7-bit-ASCII letters. For identifiers taken from other languages, the definition of lower and upper name should be modeled after the practice of each language; for example, *à* and *À* in French are lower- and upper-case variants of the same letter, although it is also common to omit the accent in the upper-case form.

8.3.0.4 This under-specification of the notion of letter case for non-7-bit-ASCII letters has no damaging consequences since the concept only appears in recommended style rules as given below (particularly D7LC, page 71), not in any rule governing actual syntax, validity or semantics.

End

8.3.1 Definition: Upper name, lower name

Code: *D7UL*

The **lower name** of an Identifier or Operator *i* is *i* written with all letters in lower case; its **upper name**, *i* with all letters in upper case.

Informative Text

8.3.1.1 For the Identifier that would be written as *DocUMent* (valid as such per the language rules, but not conforming to the recommended Eiffel style for any kind of Identifier), the lower name is *document* and the upper name *DOCUMENT*.

8.3.1.2 The definition is mostly useful for identifiers, but the names of some operators, such as **and** and other boolean operators, also contain letters.

8.3.1.3 The definitions of the notions of letter as well as lower-case and upper-case variants of a letter appear in the specification of lexical properties in subclause 0.1 (particularly DILE, page 279).

End

8.3.2 Semantics: Case Insensitivity principle

Code: *M7CI*

In writing the letters of an Identifier serving as name for a class, feature or entity, or a reserved word, using the lower name or upper name has no effect on the syntactic correctness, the validity or the semantics of the software text.

Informative Text

8.3.2.1 You can write a class or feature name as *DOCUMENT*, *document* and even *dOcUmEnT* with exactly the same meaning.

8.3.2.2 There are, however, **recommended style conventions**, presented next.

Signed by JV - 2026/10/8

8.3.2.3 The reason for not letting letter case stand in the way of semantic interpretation (beyond style conventions) is that it is simply too risky to let the meaning of a software text hang on fine nuances of writing, such as changing a letter into its upper-case variant. Such an approach can only cause confusion and errors. Different things should, in reliable and maintainable software, have clearly different names.

8.3.2.4 Letter case is of course significant in “manifest strings”, denoting texts to be taken verbatim, such as error messages or file names. The manifest strings "eiffel" and "Eiffel" are not the same.

End

8.3.3 Definition: Letter case conventions

Code: *D7LC*

An Identifier or Operator *i* appearing in a class text satisfies the letter case conventions if it uses:

1 For a class name, the upper name.

2 For a non-constant feature or entity, the lower name.

3 For a constant attribute or a once function, the lower name with the first letter changed to upper.

Informative Text

8.3.3.1 The following examples satisfy the conventions:

8.3.3.1.1 • A class name *DOCUMENT*.

8.3.3.1.2 • A non-constant *document*.

8.3.3.1.3 • A constant attribute *Document*.

8.3.3.2 While the language definition includes the definition of the letter case conventions, it does not include an obligation to observe them; they are only recommended. If you call a class *Document* rather than *DOCUMENT*, or even sometimes one way and sometimes the other, compilers must still accept all variants. An implementation may include, among its language processing tools, a *style checker* which inspects software texts for their adherence to these rules.

End

8.3.4 Definition: Scope of an identifier

Code: *D7SC*

For any class *C* and any Identifier *x*, the **scope** of *x* in *C* is the part of *C*’s text containing all the following components (deprived in cases 1, 2 and 3 of the declaration of *x* itself):

1 If *x* is the name of a feature of *C*: the entire text of *C*.

2 If *x* is the name of a formal argument of a routine or Inline_agent *r* declared in *C*: the entire Feature_declaration of *r*.

3 If *x* is the name of a declared local variable of a routine or Inline_agent *r* declared in *C*: the Feature_definition part of the Feature_declaration of *r*, deprived of Precondition and Postcondition clauses.

4 If *x* is the name of an object-test local: the scope of the enclosing Object_test.

5 If *x* is the identifier of an Iteration: the enclosing Loop.

Informative Text

8.3.4.1 The scope is the part of the class text in which the name is meaningful for the purpose it was declared: as name of a feature, argument etc. The rules of the language (precluding *overloading* as explained below in the definition of “fresh identifier”, D7FI, page 72) imply that it is not possible, in the scope of *x*, to declare any other entity also called *x*.

8.3.4.2 The discussion of “fresh identifier” will also explain the reason behind the qualification “deprived in cases 1, 2 and 3 of the declaration of *x* itself” in the above definition.

8.3.4.3 In case 1 the "features of a class" include Inherited features. Cases 2 and 3, on the other hand, refer to a feature having a declaration in the class itself, which could be a new ("immediate") declaration or a redeclaration of an inherited feature.

8.3.4.4 In case 3, "declared local variables" are those declared explicitly in a routine with the keyword **local** (part of a Local_declarations). Local variables additionally include two special cases, the predefined locals **Result** and **Retry**, but they are not identifiers so the the notion of scope is not needed for them. A separate rule determines where it is valid to use them: VEEN, page 245.

8.3.4.5 Examples of what the scope of *x* is in each case of the definition:

8.3.4.5.1 • 1 The whole of **class** *NAME* **feature** *x*: *INTEGER*; **...** **end**.

8.3.4.5.2 • 2 All dotted parts in *r* (*x*: *INTEGER*; **...**) **require** **...** **do** **...** **ensure** **...** **rescue** **...** **end**.

8.3.4.5.3 • 3 All dotted parts in *r* (*y*: *INTEGER*) **local** *x*: *INTEGER*; **...** **require** *a* **do** **...** **ensure** *b* **rescue** **...** **end**.

8.3.4.5.4 • 4 The dotted part in **if attached** *e* as *x* **then** **...** **else** *b* **end**.

8.3.4.5.5 • 5 All of **across** *e* **as** *x* **loop** **...** **end**.

8.3.4.6 A consequence of the language rules, none of these scopes (for example, other declarations for formals and locals in cases 2 and 3) may include a declaration for another entity also called *x*. In return you may rest assured that throughout all these scopes the name *x* explicitly and exactly denotes the entity that you have declared (in cases 1, 2 and 3) or introduced in a restricted context (in cases 4 and 5) with the name *x*, to the exclusion of anything else.

End

8.3.5 Definition: Fresh identifier

Code: *D7FI*

At a position in a class text, an Identifier is **fresh** if it is not in the scope of an Identifier with the same lower name.

Informative Text

8.3.5.1 There is no "overloading" in Eiffel. In a given context, one identifier means one thing (or nothing at all). The goal is to help make program texts crystal-clear.

8.3.5.2 We need fresh identifiers for names that are introduced in a certain context, called a "scope" as defined above (D7SC, page 71), and do not conflict with any other name that has a meaning in that context. An identifier is fresh if it does not cause any such confusion. Per the various cases in that definition of "scope", the clash could be with: (1) a feature name; (2) a formal argument; (3) a local variable; (4) an object-test local; or (5) the identifier of an **across** loop.

8.3.5.3 An example of a rule that relies on the concept of a fresh identifier the rule that governs the validity of a formal argument declaration such as *r* (*x*: *INTEGER*; **...**): VRFA, page 122, which states that the name *x* must be fresh. The same principle applies to the declaration of a local variable (VRLV, page 123); the identifier of an across loop (VOIT, page 229); the identifier (Object-test local) of an Object test (VVOT, page 255); a label of a labeled tuple type (VJTT, page 183); the formal arguments and local variables of an inline agent (VPIR, page 213).

8.3.5.4 To make such rules meaningful it is important to make sure that the definition of "scope" (D7SC, page 71) exclude the declaration of *x* itself; otherwise this appearance of *x* would be within its scope and, absurdly, *x* would not be considered fresh! Hence the qualification "deprived in cases 1, 2 and 3 of the declaration of *x* itself" in that definition.

8.3.5.5 Another use of fresh identifiers appears in some "unfolded form" which define the validity and semantics of a construct through a fictitious modification of it, which may involve new identifiers not actually appearing in the program (that is to say, fresh). Examples are: the covariance-aware form of an assertion extension (DACA, page 135); the direct transposition of a component to a class or type (DDTR, page 155) ; and the unfolded form of a tuple type (VJUT, page 183).

8.3.5.6 Note that there is another notion of freshness: the freshness of a routine call (DUFR, page 207), particularly important for once routines. The two are not directly related: freshness of an identifier is a static (in fact, syntactic) concept; freshness of a routine call is a semantic concept having to do with the execution of routines.

End

8.3.6 Syntax (non-production): Semicolon Optionality rule

Code: *S7SC*

In writing specimens of **any** construct defined by a Repetition production specifying the semicolon "**;**" as separator, it is permitted, without any effect on the syntax, validity and semantics of the software, to omit any of the semicolons, or to add a semicolon before the first element of the repetition, after an existing semicolon, or after the last element.

Informative Text

8.3.6.1 This rule applies to instructions in a Compound, declarationsof many different kinds, successive groups of formal arguments, and many other repetition-defined constructs.

8.3.6.2 It does not rely on the *layout* of the software (as in some programming languages which treat passage to a new line as equivalent to inserting a separator): Eiffel's syntax is free-format, so that a return to the next line has the same effect as one or more spaces or any other "break". Rather than relying on line returns, the Semicolon Optionality rule is ensured by the syntax design of the language, which guarantees that omitting a semicolon never creates an ambiguity.

8.3.6.3 The rule also guarantees that an extra semicolon at the end, as in *a*; *b*; instead of just *a*; *b* is harmless; so is adding a semicolon at the beginning of a semicolon-separated list, or duplicating a semicolon. While it is desirable to keep the text clean by avoiding such extra symbols, compilers should not bother programmers with spurious errors for such harmless additions.

8.3.6.4 It is even possible, from the rule, to add a semicolon

8.3.6.5 The **style guidelines** prescribe omitting semicolons (which would only obscure reading) for successive elements appearing on separate lines, as is usually the case for instructions and declarations, and including them to separate elements on a given line.

8.3.6.6 Because the semicolon is still formally in the grammar, programmers used to languages where the semicolon is an instruction *terminator*, who may then out of habit add a semicolon after every instruction, will not suffer any adverse effect, and will get the semantics they expect.

End

Signed by JV - 2026/10/8

PART D: System structuring mechanisms

Signed by JV - 2026/10/8

Signed by JV - 2026/10/8

8.4 Elementary concepts

Informative Text

8.4.0.1 The present clause explains the overall context: systems and their executions.

End

8.4.1 Definition: Systems and their executions

Code: *D2TE*

8.4.1.1 Eiffel is a notation for describing texts, called **systems**. The precise nature of a system is defined as part of the next subclause 8.5 (in DSRT, page 79).

8.4.1.2 The purpose of a system is to describe possible computations, creating and manipulating structures made generator, page 243, to be carried out by computing device such as a computer, a virtual machine (a computer simulated by software), a smartphone, an Internet of Things device, or any other device with capabilities equivalent to Turing-completeness.

8.4.1.3 An **execution** of a system is such a computation, performed according to the description provided by the system. The start of an execution proceeds as defined in MSSE, page 81.

Informative Text

8.4.1.4 A system is also called a *program*. While “program” is the most common term for other programming languages, “system” is preferred for Eiffel as it conveys the Eiffel idea of an executable assembly of software components, which may have their own value beyond their inclusion in any one system, and have the potential to be reused in different systems.

End

8.4.2 Definition: Execution terminology

Code: *D2TE*

- **Run time** is the period during which a system is executed.
- The **machine** is the combination of hardware (one or more computers) and operating system through which you can execute systems.
- The machine type, that is to say a certain combination of computer type and operating system, is called a **platform**.
- **Language processing tools** serve to build, organize, manipulate, modify, explore, analyze, verify, document and execute a system on a machine.

Informative Text

8.4.2.1 The most obvious example of a language processing tool is an Eiffel compiler or interpreter, which you can use to execute a system. But many other tools can manipulate Eiffel texts: Eiffel-aware editors, browsers to explore systems and their properties, documentation tools, debuggers, configuration management systems. Hence the generality of the term “*language processing tool*”.

8.4.2.2 On the side of terminology and proper usage, it is important to note that the word “run time”has some variants, each to be used in a specific context:

8.4.2.2.1
- “**Run time**”, as defined above, is a substantive form (in other words, plays the role of a noun) denoting a period; an example use of this term is “*Classes exist in the Eiffel text; objects exist only at **run time***”. This form does not use a hyphen.

8.4.2.2.2
- “**Run-time**” with a hyphen is the adjectival variant of this substantive form, used in such examples as “*objects are a **run-time** concept*”.

8.4.2.2.3
- The (substantive) term “**runtime**” denotes a program (a language processing tool as defined above) running along Eiffel systems to provide certain facilities necessary to their execution, such as memory management and exception handling. For example: “*the **runtime** includes facilities for garbage collection*”. Since runtimes are software tools specific to each implementation, the term “runtime” does not appear further in the Standard.

End

Signed by JV - 2026/10/8

8.5 The architecture of Eiffel software

Informative Text

8.5.0.1 The constituents of Eiffel software are called **classes**. To keep your classes and your development organized, it is convenient to group classes into **clusters**. By combining classes from one or more clusters, you may build executable **systems**.

8.5.0.2 These three concepts provide the basis for structuring Eiffel software:

8.5.0.2.1 • A class is a modular unit. Classes are studied in the next subclause (SCCD, page 82).

8.5.0.2.2 • A cluster is a logical grouping of classes. Clusters are defined below.

8.5.0.2.3 • A system results from the assembly of one or more classes to produce an executable unit.

8.5.0.3 Of these, only “class”, describing the basic building blocks, corresponds directly to a construct of the language. To build clusters and systems out of classes, you will use not a language mechanism, but language processing tools (software tools of the supporting environment).

8.5.0.4 Clusters provide an intermediate level between classes and systems, indispensable as soon as your systems grow beyond the trivial:

8.5.0.4.1 • At one extreme, a cluster may be a simple group of a few classes.

8.5.0.4.2 • At the other end, a system as a whole is simply a cluster that you have made executable (by selecting a *root class* and a root procedure).

8.5.0.4.3 • In-between, a cluster may be a library which (recursively) groups several clusters (its subclusters), or an existing system that you wish to integrate into a larger system.

8.5.0.5 Clusters also serve to store and group classes using the facilities of the underlying operating system, such as files, folders and directories.

8.5.0.6 After the basic definitions, the language description in this subclause and the following ones will concentrate on classes, indeed the most important concept in the Eiffel method, which views software construction as an industrial production activity: combining components, not writing one-of-a-kind applications.

End

8.5.1 Definition: Cluster, subcluster, contains directly, contains

Code: *DSCL*

A **cluster** is a collection of classes, (recursively) other clusters called its **subclusters**, or both. The cluster is said to **contain directly** these classes and subclusters.

A cluster **contains** a class *C* if it contains directly either *C* or a cluster that (recursively) contains *C*.

Informative Text

8.5.1.1 In the presence of subclusters, several clusters may contain a class, but then exactly one contains it directly.

End

8.5.2 Definition: Terminal cluster, internal cluster

Code: *DSTI*

A cluster is **terminal** if it contains directly at least one class.

A cluster is **internal** if it contains at least one subcluster.

Informative Text

8.5.2.1 From these definitions, it is possible for a cluster to be both terminal and internal.

8.5.2.2 As a general style rule, it is usually better to exclude such a case: some clusters contain only subclusters (they are internal nodes of the cluster tree or forest) and others, at the leaves, contain only classes. While recommended, this practice is not obligatory.

End

8.5.3 Definition: Universe

Code: *DSUV*

A **universe** is a set of classes.

Informative Text

8.5.3.1 A universe provides a reference from which to draw classes of interest for a particular system.

8.5.3.2 Concretely, the universe provides language processing tools, in particular compilers, with an association (mathematically, a function, also called a **mapping**) from class names to classes, specifying a certain set of known class names and, for each of them, a class that it represents. For example a universe might contain three classes and indicate that the first one is known by the Class_name *A*, the second one by *B* and the third one by *C*. This three-element mapping (three names respectively mapped to three different classes) defines the universe.

8.5.3.3 The language processing tools of any Eiffel environment will provide a way to specify a universe.

End

8.5.4 Syntax: Class names

Code: *SSCN*

Class_name $\triangleq$ Identifier

Informative Text

8.5.4.1 As seen in D7LC, page 71, it is customary, and part of the recommended style rules, to use an all-upper-case name for a class, as in *DOCUMENT*.

8.5.4.2 Since this convention is only a style guideline and not a language obligation, all definitions and rules that involve the name of a class refer instead to the upper name of the class, that is to say, its name converted, if necessary, to all-upper-case letters. The first example is the following validity rule.

End

8.5.5 Validity: Class Identification rule

Code: *VSCI*

It is valid for a universe to include a class if and only if no other class of the universe has the same upper name.

Informative Text

8.5.5.1 Eiffel expressly does not include a notion of “namespace” as present in some other languages. Experience with these mechanisms shows that they suffer from two limitations:

8.5.5.1.1 • They only push forward the problem of class name clashes, turning it into a problem of namespace clashes.

8.5.5.1.2 • Even more seriously, they tie a class to a particular context, making it impossible to reorganize (“*refactor*”) the software later without breaking existing code, and hence defeating some of the principal benefits of object technology and modern software engineering.

8.5.5.2 Name clashes, in the current Eiffel view, should be handled by *tools* of the development environment, enabling application writers to combine classes from many different sources, some possibly with clashing names, and resolving these clashes automatically (with the possibility of registering user preferences and remembering them from one release of an acquired external set of classes to the next) while maintaining clarity, reusability and extendibility.

8.5.5.3 The following validity and semantic rules follow directly from the above rule.

End

8.5.6 Validity: Class Name rule

Code: *VSCN*

A Class_name *C* appearing outside of a Class_header is valid if and only if in the enclosing universe includes a class of name *C*.

Informative Text

8.5.6.1 This rules applies in any part of a program text where a Class_name is expected. For example, if you have an Inherit_clause clause, starting with inherit and listing the desired parents of a class, it includes some Parent specimens, each of which starts with a Class_type, based on a Class_name. So in

class *B* **inherit** *A* **feature** *. . .* **end**

it is valid to use A only if it is the name of a class other than B.

8.5.6.2 The reason why the rule includes the qualification "appearing outside of a Class_header" is that the Class_header is the part of a class declaration where we introduce the class name; the Class_header in the above example is "**class** *B*". Clearly, as the Class_header introduces a new class, adding it to the universe (as will be specified in MCCH, page 85), it cannot require that the universe already include the Class_name (that would actually be a violation of the above Class Identification rule, VSCN, page 79). But in all other contexts in which we expect a Class_name, such as *A* in the **inherit** clause, it must be the name of a class already present in the universe.

End

8.5.7 Semantics: Class name semantics

Code: *MSCN*

A Class_name *C* appearing in the text of a class *D* denotes the class called *C* in the enclosing universe.

Informative Text

8.5.7.1 This rule is made meaningful by the two preceding validity rules, Class Identification (VSCI, page 78) and Class Name (VSCN, page 79) which together guarantee that in any class of the universe the name *C* denotes exactly one class (respectively, at most one class and at least one class).

8.5.7.2 *D* can be *C* (with a text referring to *C* itself) or another class.

End

8.5.8 Definition: System, root type name, root procedure name

Code: *DSRT*

A **system** is defined by the combination of:

1 A universe.

2 A type name, called the **root type name**.

3 A feature name, called the **root procedure name**.

Informative Text

8.5.8.1 These notions, developed below, are fundamental to defining how to make up a system. The universe defines where classes will be found, based on their names. The root procedure defines which code must be executed to start the system. The root class defines where to find the root procedure.

Signed by JV - 2026/10/8

8.5.8.2 In the root class, we will generally find class names, which must denote classes of the universe, whose texts again contain class names to which we must be able to associate classes and so on. We need a completeness condition, ensuring that starting from the root class we may find all classes needed directly or indirectly. The following definition helps specify this requirement.

End

8.5.9 Definition: Type dependency

Code: *DSTD*

A type *T* **depends** on a type *R* if any of the following holds:

1 *R* is a parent of the base class *C* of *T*.

2 *T* is a client of *R*.

3 (Recursively) there is a type *S* such that *T* depends on *S* and *S* depends on *R*.

Informative Text

8.5.9.1 This condition relies of the various forms of possible dependency between classes and types specified in later subclauses ot the standard: parent (through inheritance) and client (or its reverse, supplier).

8.5.9.2 It states that *C* depends on *A* if it is connected to *A* directly or indirectly through some combination of these relations.

8.5.9.3 Case 1 relies on the property that every type derives from a class, called its "base class"; for example a generically derived type such as *LIST* [*INTEGER*] has base class *LIST*. Case 3 gives us indirect forms of dependency, derived from the other cases.

End

8.5.10 Validity: Root Type rule (and definition: root type)

Code: *VSRT*

It is valid to designate a type *TN* as root type of a system *S* of universe *U* if and only if it satisfies the following conditions:

1 *TN* is the name of a stand-alone type *T*, called the **root type** of *S*.

2 *T* only involves classes in *U*.

3 The base class of any type on which *T* depends is in *U*.

4 *T*'s base class is effective.

Informative Text

8.5.10.1 These conditions make it possible to create the root object (which, as seen in MSSE, page 81 below, will start the execution of the system). They are expressed in terms of concepts specified precisely in subsquent subclauses:

8.5.10.1.1 • Condition 1: a type is "stand-alone" if it only involves class names; this requirement excludes "anchored" types (**like** *some_entity*) and formal generic parameters, which only mean something in the context of a particular class text. Clearly, if we want to use a type as root for a system, it must have an absolute meaning, independent of any specific context. "Stand-alone type" is defined at the end of the discussion of types (STSA, page 171).

8.5.10.1.2 • Condition 2: every class to which the root type (through its base class, the "root class" as defined next) refers must exist in the universe (the set from which all classes in the system are drawn). A type *T* "involves" a class *C* if it is defined in terms of *C*, meaning that *C* is the base class of *T* or of any of its generic parameters: *U* [*V*, *X* [*Y*, *Z*]] involves *U*, *V*, *X*, *Y* and *Z*. If *T* is a non-generic class used as a type, *T* "involves" only itself.

8.5.10.1.3 • Condition 3: to be able to assemble the system, we must ensure that any class to which the root refers directly or indirectly is also part of the universe.

Signed by JV - 2026/10/8

8.5.10.1.4 • Condition 4: an effective class (the reverse of “deferred”, see PDDE, page 153) is fully implemented, and hence can have direct instances. A deferred class would not work as base class here, since the very purpose of a root type is to be instantiated, as the first event of system execution.

8.5.10.2 Condition 4 also follows from the first condition of a validity rule below, Root Procedure (VSRP, page 81), which requires the class to have a creation procedure and hence not to be deferred.

End

8.5.11 Definition: Root class

Code: *DCRC*

The **base class** of a system is the base class of its root type.

Informative Text

8.5.11.1 In many cases the root type is not generic and is the same as the root class.

8.5.12 Validity: Root Procedure rule (and definition: root procedure)

Code: *VSRP*

It is valid to specify a name *pn* as root procedure name for a system *S* if and only if it satisfies the following conditions:

1 *pn* is the name of a general creation procedure *p* of *S*’s root type. *p* is called the **root procedure** of *S*.

2 *p* has no formal argument.

3 *p* is precondition-free.

Informative Text

8.5.12.1 In condition 1, a creation procedure is *general* if its any client can use it to create objects.

8.5.12.2 Also in condition 1, note that a non-deferred class that does not explicitly list any creation procedures is understood to have a single one, procedure *default_create*, which does nothing by default but may be redefined in any class to carry out specific initializations. See DKPC, page 233 and the surrounding discussion.

8.5.12.3 A routine is *precondition-free* (condition 3) if it has no precondition, or a precondition that evaluates to true. A routine can impose preconditions on its callers if these callers are other routines; but it makes no sense to impose a precondition on the external agent (person, hardware device, other program...) that sets off an entire system execution, since there is no way to ascertain that such an agent, beyond the system’s control, will observe the precondition. Hence the last condition of the rule.

End

8.5.13 Semantics: System execution (and definition: root object)

Code: *MSSE*

To **execute** a system on a machine means to cause the machine to apply a creation instruction (creating a **root object**) to the system’s root type, using the system’s root procedure as the instruction’s creation procedure.

Informative Text

8.5.13.1 This specification describes the entire process of software execution: the creation instruction will (per the specification of creation instruction semantics, MKIN, page 236) create a root object and execute a creation procedure on it — in this case, the root procedure.

8.5.13.2 In any non-trivial case that root procedure will in turn create other objects and call other features on these objects.

8.5.13.3 In other words, the execution of any system is a chain reaction of operations — creations and calls — each one firing off the next; the root procedure is the spark that starts the first step.

End

Signed by JV - 2026/10/8

8.6 Classes

Informative Text

8.6.0.1 Classes are the components used to build Eiffel software.

8.6.0.2 Classes serve two complementary purposes: they are the modular units of software decomposition; they also provide the basis for the type system of Eiffel.

8.6.0.3 About the first role: there is no modular construct in the language at a higher level than classes. "Clusters" serve to group clusters but do not exist as language elements. Routines describe computations but are always part of classes, in line with object-oriented principles.

8.6.0.4 On the second role: every type is based directly or indirectly on a class, called its base class (Base Class principle, page 166). Simple classes are themselves types, describing a set of objects through their applicable operations (features). In more advanced cases the relationship between a type and its base class is more indirect, as with a generically derived class *C* [*T*] whose base class is *C*.

8.6.0.5 This chapter reviews the components making up a class text.

End

8.6.1 Definition: Current class

Code: *DCCC*

The **current class** of a component is the class in whose text it appears.

Informative Text

8.6.1.1 A "component" is any software element, always a specimen of some construct (D2CS, page 60).

8.6.1.2 Every Eiffel component — feature, expression, instruction, … — appears in a class, justifying the definition above.

8.6.1.3 Through this notion of "current class", most language properties refer directly or indirectly, to the class to which a component belongs.

End

8.6.2 Syntax: Class declarations

Code: *SCCD*

Class_declaration ≙
[Notes]
Class_header
[Formal_generics]
[Obsolete]
[Inheritance]
[Creators]
[Converters]
[Features]
[Invariant]
[Notes]
end

Informative Text

8.6.2.1 All the components are optional except the Class_header (defined below as **class** followed by a class name, with an optional initial qualifier) and the final **end**.

8.6.2.2 As a consequence, a Class_declaration in its simplest form would just be **class** *SOME_NAME* **end**.

Signed by JV - 2026/10/8

8.6.2.3 When present, the other components fulfill the following roles, presented here informally as a preview:

8.6.2.3.1 - Notes introduces information about the class that has no effect on its semantics but serves to document it and index it for future retrieval.

8.6.2.3.2 - Formal_generics introduces generic parameters representing types parameterizing the class, as with a class *LIST* [*G*] representing lists of objects of an arbitrary type represented by the formal generic parameter *G*.

8.6.2.3.3 - Obsolete indicates that using the class, while still possible, is no longer recommended. Typically a new, improved class has been made available. An obsolete message included in the component explains the reason for obsolescence and the suggested replacement.

8.6.2.3.4 - Inheritance gives the parent types parent types, specifying existing classes from which the new class will inherit features and invariant clauses.

- Creators lists the procedures of the class (declared in its Features part) that can be used to initialize new instances on creation.

8.6.2.3.5 - Converters lists the functions and procedures of the class (again declared as some of its Features) that can be used to convert instances of this class into instances of others, or conversely.

8.6.2.3.6 - Features includes one or more Feature_declaration clauses, each introducing some of the features of the class.

8.6.2.3.7 - Invariant introduces the class invariant: consistency conditions on instances of the class.

8.6.2.3.8 - A final Notes part, like the one at the top of the class, provides extra information. Typically, the initial clause provides essential information such as the purpose of the class; the final clause provides information that may not be the first that a reader of the class needs, such as copyright information.

End

8.6.3 Syntax: Notes

Code: *SCNO*

Notes ≜ **note** Note_list

Note_list ≜ {Note_entry ";" …}*

Note_entry ≜ Note_name Note_values

Note_name ≜ Identifier ":"

Note_values ≜ {Note_item ","…}+

Note_item ≜ Identifier | Manifest_constant

Informative Text

8.6.3.1 Notes parts (there may be up to two, one at the beginning and one at the end) have no effect on the execution semantics of the class. They serve to associate information with the class, for use in particular by tools for configuration management, documentation, cataloging, archival, and for retrieving classes based on their properties.

8.6.3.2 For convenience, a class text may include a Notes part at the beginning, one at the end, or both (or neither). A short head-of-class Notes (for a *LINKED_LIST* class) reads:

8.6.3.3
```
note
        description: "Sequential, one-way linked lists"
                library: "Free implementation of ELKS library"
                legal: "See notice at end of class."
                status: "See notice at end of class."
```


Signed by JV - 2026/10/8

names: linked_list, sequence
representation: linked
access: index, cursor, membership
contents: generic
date: "$Date$"
revision: "$Revision$"

8.6.3.4 Successive Note_entry components help document and index the class by each giving a general category, the Note_name, such as *description*:,then listing one or more associated values (each a Note_item).

8.6.3.5 Language history note: in some older Eiffel documentation you may encounter the keyword **indexing**, which **note** has since replaced.

End

8.6.4 Semantics: Notes semantics

Code: *MCNO*

A Notes part has no effect on system execution.

Informative Text

8.6.4.1 Notes parts for a class provide information that may not be immediately inferable from the rest of the class text, and can be of use to both human readers and language processing tools (not necessarily compilers) that record information about the class, for example for indexing and keyword-based retrieval.

8.6.4.2 Such information is not part of the execution semantics of the class: it has no effect on creation of its instances and operations on them.

End

8.6.5 Syntax: Class headers

Code: *SCCH*

Class_header ≙ [Header_mark] **class** Class_name

Header_mark ≙ **deferred** | **expanded** | **frozen**

Informative Text

8.6.5.1 The Class_name part gives the name of the class. The recommended convention (here and in any context where a class text refers to a class name) is to use the upper name: call a class not *Person* or *person* but *PERSON*. This style convention (a recommendation, not a language rule) was part of the general letter case conventions presented in D7LC, page 71.

8.6.5.2 The keyword **class** may optionally be preceded by one of the keywords **deferred**, **expanded** and **frozen**, corresponding to variants of the basic notion of class:

8.6.5.2.1 • A **deferred** class describes an incompletely implemented abstraction, which descendants will use as a basis for further refinement. See the discussions of inheritance and of feature adaptation, in particular the Deferred Class property, PDDE, page 153.

8.6.5.2.2 • Declaring a class as **expanded** indicates that entities declared of the corresponding type will denote objects rather than references to objects. See in particular the definition of expanded types, DTEX, page 168.

8.6.5.2.3 • Making a class **frozen** prohibits it from serving as "conforming parent" to other classes. See the definition of "frozen class" below, DCEF, page 85, and clause 2 of the Parent rule, VHPR, page 146.

End

Signed by JV - 2026/10/8

8.6.6 Validity: Class Header rule

Code: *VCCH*

A Class_header appearing in the text of a class *C* is valid if and only if *C* has either no deferred feature or a Header_mark of the **deferred** form.

Informative Text

8.6.6.1 A class is either “effective”, meaning that it is fully implemented, or “deferred”, meaning that it is specified but not necessarily provided with an implementation of all of its features. The key difference is that a class may only, at run time, yield direct instances (objects produced from the bluepring defined by the class) if it is effective.

8.6.6.2 Like classes, features also each have “effective” or “deferred” status. An effective feature is fully implemented; a deferred feature is only specified. A feature is deferred if it is declared as such (see Feature_body, page 114), if the current class undefines it, or if it is inherited as deferred and not effected (provided an implementation) in the current class.

8.6.6.3 As soon as a class has at least one deferred feature, it is considered a deferred class and its Class_header *must* start with the **deferred** Header_mark (per the Deferred Class property, PDDE, page 153). But even if it only has effective (non-deferred) features, you may decide to make the class itself deferred, prohibiting the creation of direct instances of it. To this effect, just start the Class_header with **deferred** (even though the Deferred Class property does not require you to do so in this case: making the class deferred is just a design decision on your part).

8.6.6.4 There is no particular rule on the other possible markers, **expanded** and **frozen**, for a Class_header. Expanded classes often make the procedure *default_create* available for creation, but this is not a requirement since the corresponding entities may be initialized in other ways; they follow the same rules as other “attached” entities.

8.6.6.5 Another important condition governing the validity of a Class_header is that the Class_name should not be the name of a class already present in the universe. That condition, however, is not part of the present rule since it was already specified in the Class Identification rule, VSCI, page 78. As a result of the Class_header, the new class will be added to the universe, as specified in the following semantic rule (MCCH, page 85).

End

8.6.7 Semantics: Class Header semantics

Code: *MCCH*

The appearance of a Class_name in a Class_header results in the addition to the universe of a class of that name.

Informative Text

8.6.7.1 The universe includes all the classes available to build a system (DSUV, page 78). Declaring a class adds it, through its Class_header, to the current universe.

End

8.6.8 Definition: Expanded, frozen, deferred, effective class

Code: *DCEF*

A class is:

- **Expanded** if its Class_header is of the **expanded** form.
- **Frozen** if its Class_header is of the **frozen** or **expanded** form.
- **Deferred** if its Class_header is of the **deferred** form.
- **Effective** if it is not deferred.

Signed by JV - 2026/10/8

Informative Text

8.6.8.1 As the definition implies, an expanded class is also considered frozen (even though the keywords **frozen** and **expanded** are exclusive). The deferred and effective cases are mutually exclusive (in fact, the latter is defined as the opposite of the other). Since the keyword **deferred** is exclusive of **expanded** and **frozen**, a frozen class (including the case of an expanded class) is effective.

8.6.8.2 An expanded class describes a class whose instances (all of which are direct instances) have copy semantics: assignment and argument passing will cause copying objects.

8.6.8.3 Making *C frozen* prohibits it from serving as "conforming parent" (DHCO, page 145) to other classes. The first two cases indicate the two ways of ensuring this:

8.6.8.3.1 • Inheritance from expanded classes, as explained in the discussion of DHCO, is non-conforming.

8.6.8.3.2 • You can explicitly mark a non-expanded class as **frozen**.

8.6.8.4 The other two cases define what makes a class *deferred* or *effective*, as determined by the Deferred Class property, PDDE, page 153 and discussed above:

8.6.8.4.1 • If it has at least one deferred feature, the class itself is deferred. The Class Header rule above (VCCH, page 85) requires it to be marked **deferred** for clarity.

8.6.8.4.2 • If it only has effective features, the class is effective unless you decide otherwise: you can still explicitly mark it **deferred**. This ensures that it will have no direct instances, since one may not apply creation instructions or expressions to a variable whose type is based on a deferred class.

End

8.6.9 Syntax: Obsolete marks

Code: *SCOM*

Obsolete ≜ **obsolete** Message

Message ≜ Manifest_string

Informative Text

8.6.9.1 An Obsolete mark indicates that the class, while still usable, is "deprecated", meaning its usage is no longer recommended. Usually there is a recommended replacement, as specified by the Message, which is just a string that the programmer having decided to make the class obsolete will have devised appropriately to help other programmers with the transition.

8.6.9.2 Besides classes, it is also possible to make individual *features* obsolete, along the same lines. See , the construct for feature declarations (SFDE, page 102), which also includes an optional Obsolete component, with the same properties as the corresponding component for declaring, here, a class obsolete.

End

8.6.10 Semantics: Obsolete semantics

Code: *MCOB*

Specifying an Obsolete mark for a class or feature has no run-time effect.

When encountering such a mark, language processing tools may issue a report, citing the Message and advising software authors to replace the class or feature by a newer version.

Informative Text

8.6.10.1 The obsolescence mechanism, for classes and features, supports *software evolution*: when you realize that a better design can advantageously replace an existing one, you can help programmers move on without breaking existing code by immediately removing the previous version.

End

8.7 Clients and exports

Informative Text

8.7.0.1 Along with inheritance, the client relation is one of the basic mechanisms for structuring software.

8.7.0.2 In broad terms, a class *C* is a client of a type *S* — which is then a *supplier* of *C* — when it can manipulate objects of type *S* and apply *S*'s features to them.

8.7.0.3 IIn this definition, "client" is a relation that holds between a class and a type; since every type is based on a class, the relation immediately extends to one between classes, holding here between *C* and the base class of *S*. This extension to classes is the subject of one of the first definitions below (DLCR, page 88).

8.7.0.4 In the recommended graphical notation for describing system architecture, where classes are represented by ellipses, a client link appears as a double arrow (preferably blue) from the client to the supplier, as in this example:



8.7.0.5 Some discussions of object-oriented methods use the term "delegation" to express that a feature of a class resorts to features of another class to carry out some of its computations; the client relation is the mechanism for achieving this goal.

8.7.0.6 The simplest and most common way for *C* to be a client of *S* is for *C* to contain the declaration of an entity of type *S* (typically declared as *x*: *S)*. In this case *C* is a "simple client" of *S*. Even for simple clients (per the definition below, DLSC, page 89) the dependency may take another form, for example involving expressions or generic parameters.

8.7.0.7 For the unrestricted notion (client, not just "simple" client), inheritance gets into the picture: if a parent of *C*, or more generally one of its proper ancestors, is a simple client of *S*, then *C* is a client of *S*.

8.7.0.8 In addition to simple client, the client relation has the variants expanded client and generic client, each studied below.

8.7.0.9 Underlying the entire discussion of the present subclause is the notion of **information hiding**, a key principle of software development, which helps develop, understand, maintain, verify, extend and reuse software systems, particularly large ones, by limiting the amount of information that various parts of the system (specifically, classes) need to have about other parts on which they rely through the client relation. In application of this principle, a class C may use a supplier S on the basis of only:

8.7.0.9.1 • *Exported features*: a subset of the features of *S*'s base class, excluding features that S only needs for implementation purposes.

8.7.0.9.2 • *Exported properties*: a subset of the exported features' properties, excluding properties of their implementation (but retaining specification properties, known as contracts). Exported properties are also called "public"; this notion is defined precisely below in DLSP, page 96.

8.7.0.10 As a consequence, changes made to *S* as part of the evolution of the software will not affect clients such as *C* if they only involve non-exported features and non-exported properties of exported features. This guarantee is a fundamental advantage for the stability of software architectures.

8.7.0.11 The specification of the client relation in the present subclause successively covers: basic definitions; extension to a class-to-class (not just class-to-type) relation; special cases (expanded client, generic client); Clients parts (specifying availablity of a set of features to specific clients); export status of a feature (public, secret, selectively available); and documentation-oriented forms of a class such as the "Contract View", produced by language processing tools, enabling its use by authors of client classes.

End

Signed by JV - 2026/10/8

8.7.1 Definition: Client relation between classes and types

Code: *DLSS*

A class *C* is a **client** of a type *S* if some ancestor of *C* is a simple client of *S*.

Informative Text

8.7.1.1 The starting point for the client relation is the notion of "simple client", defined below (DLSC, page 89). The more general notion of "client" defined here is based on "simple client", but adds the possibility of *C* becoming a (non-direct) client of *S* through inheritance. It involves all of *C*'s ancestors, which may cause client dependencies caused by inherited features (complementing the direct ones due to to the immediate features of *C*).

8.7.1.2 In other words, while a client dependency between *C* and *S* may come from a declaration appearing in the text of *C* itself, it could also arise from other ("proper") ancestors (parents, their own parents and so on).

8.7.1.3 Assume for example that a routine *r* of *C*, inherited from one of its parents, uses a local variable *x* of type *S*. The declaration *x: S* of that variable makes that parent is a direct client of *S*; it does not make *C* itself a direct client of *S* , but does make it a client of *S* even if the text of *C* does not mention *S*.

8.7.1.4 The next definition generalizes the concept of "client" from types to classes.

End

8.7.2 Definition: Client relation between classes

Code: *DLCR*

A class *C* is a **client of a class** *B* if and only if *C* is a client of a type whose base class is *B*.

The same convention applies to the simple client, expanded client and generic client relations.

Informative Text

8.7.2.1 A reminder on (or preview of) the notion of base class: every type is "based on" a class, its "base class". In simple cases a type is also a class, and then is its own base class; for example, *INTEGER* is both a type and a class. The two concepts become distinct when we consider various advanced mechanisms of the Eiffel type system, particularly genericity (*ARRAY* [*INTEGER*] is not a class by itself but a type, of base class *ARRAY*) and anchored types (a type of the form **like** *x* has the same base class as the type of *x*). The presentation of the various kinds of type, starting in subclause 8.14, gives in each case the precise rule defining the type's base class.

8.7.2.2 The present definition, DLCR, enables us to use the term "client", and its three variants, indifferently between a class and a type or between a class and another class.

8.7.2.3 In addition, this definition makes it possible for the definitions of specific variants (simple client in DLSC, page 89, expanded client in DLEC, page 90 and generic client in DLGC, page 91) to limit themselves to specifying when a class is a client (simple, expanded or generic) of a *type*. The corresponding class-to-class client and supplier relations are a consequencem through DLCR.

End

8.7.3 Definition: Supplier

Code: *DLSU*

A type or class *S* is a **supplier** of a class *C* if *C* is a client of *S*, with corresponding variants: simple, expanded, generic, indirect.

Informative Text

8.7.3.1 In mathematical terms, the relation "supplier" is (for each variant) the inverse of the corresponding "client" relation.

Signed by JV - 2026/10/8

8.7.3.2 The following definitions cover the three ways in which a class may, by listing a type *S* in its own text, enable itself to use features of *S* on targets of type *S*, becoming a *simple*, *expanded* or *generic* client. We start with "simple client".

End

8.7.4 Definition: Simple client

Code: *DLSC*

A class *C* is a **simple client** of a type *S* if and only if *S* is involved in one of the following in *C*:

1 The type of an entity.
2 The type of an Expression.
3 The Explicit_creation_type of a Creation_instruction or Creation_expression.
4 A constraining type of a Formal_generic of *C*.
5 An actual generic parameter of (recursively) a client of *C*.
6 An actual generic parameter of a parent type of *C*.
7 The Type of a Non_object_call.
8 The Type of a Manifest_type.

Informative Text

8.7.4.1 An explanation of "involved in" appears below, at the end of this Informative Text, but as a first approximation you may ignore this phrase and consider that *S* itself simply *is* the type mentioned in the relevant case of the definition.

8.7.4.2 The primary purpose of C being a client of S is to allow, in the text of *C*, a call of the form *x*.*f* (…) (or some variant of that syntax) where *x*, the target of the call, is of type *S* and *f*, the feature of the call, is a feature of *S*. The cases of the definition cover the variants of this basic situation.

8.7.4.3 Case 1 is the simplest and most common: the text of *C* has an entity declaration of the form *x*: *S* for some *x*. The concept of entity is broad (DEEN, page 245), covering attributes, local variables, formal arguments and object-test locals.

8.7.4.4 Case 2 is for an expression, appearing anywhere in *C*, whose type (as determined by the Expression Type rule, DWET, page 221) is *S*.

8.7.4.5 Case 3 addresses the possibility that a creation instruction (or expression) names an explicit type, as in the instruction **create** {*S*} *x* (possibly with a creation procedure as in **create** {*S*} *x*.*make* (…)). The mention {*S*} of an explicit type is optional; if it is absent, the resulting object is of the type of *x*; since that type, say *T,* follows from the declaration of *x* if *x* is an entity, or from the Expression Type rule (DWET, page 221) if it is an expression, we already know that *C* is a client of *S* from the first two cases of the definition. The explicit type variant makes *C* a client of *S* even if no other case applies.

8.7.4.6 Case 4 arises when the class is declared as **class** *C* […, *G* –> *S*, …] …, with a generic parameter *G* "generically constrained" by the type *S*, under the "constrained genericity" mechanism specified in the clause covering genericity, 8.15, page 173 (in particular the subclauses starting with DGCO, page 176). Such a declaration indicates that *C* is a generic class, with a formal generic parameter *G* representing not an arbitrary type but one that must conform to *S*. Then *S* is syntactically a Constraint , part of the Constraining_types for *G*, but the specification of case 4 does not refer to either of these two syntactic types; it uses instead the concept of constraining type (defined in DGCO, page 176) to account for class declarations that use just **class** *C* […, *G*, …] …, without an explicit "–> *S*" Constraint. In such situations, the formal generic parameter *G* still has a constraining type, but an implicit one: **separate detachable** *ANY*, using the class *ANY* which serves as ancestor (implicit or explicit) to all cases. (This role of *ANY* follows from the Universal Conformance principle, VHUC, page 144.) In other words, **class** *C* […, *G*, …] … can be considered an abbreviation for **class** *C* […, *G* –> **separate detachable** *ANY*, …] …. As a result, every generic

Signed by JV - 2026/10/8

class having a non-explicitly-constrained generic parameter — the simplest and most common form of genericity — is a client of **separate detachable** *ANY*. In terms of the client relation between classes (defined above as DLCR, page 88), it is a client of *ANY*.

8.7.4.7 Case 5 is for when, through any of the cases, *C* is a client of a generically derived type *D* […, *S*, …] for some generic class *D* and some type S; it specifies that *C* is then also a client of *S*. This case is recursive, since *S* itself might be generically derived, of the form *E* […, *T*, …], making *C* a client of *T*.

8.7.4.8 In case 6, *C* has an Inheritance clause **inherit** … *D* […, *S*, …] …, making *C* a client of *S*. Note that this inheritance clause does not make *C* a client of the type *D* […, *S*, …] (or of its base class *D*), only an heir of it; but it does make *C* a client of each of the actual generic parameters (and their base classes). By case 5, if *S* itself is generically derived, of the form *E* […, *T*, …], then C is also a client of *T*.

8.7.4.9 In cases 7 and 8, *C* does not need to use an actual entity or expression of type *S*, but does use *S* in another form which also yields a client-to-supplier relationship. In case 7, the reason is a Non_object_call of the form {*S*}.*class_feature* (…), where *class_feature* is a feature defined in the base class of *S* but not necessitating an object as the target of calls to it; for example, it could be a constant declared in *S*. (The discussion of non-object calls is in the subclause on calls: 8.19.)

8.7.4.10 Case 8, also not requiring an entity or expression, happens when *C* uses a “manifest type”, a constant expression of the form {*S*}, denoting at run time an object that represents type *S*, so that the features of *C* can access the properties of that type.

8.7.4.11 In all these cases, the specification (see the first line of the definition) does not say that *S* is the type directly referenced in some component of the text of C (a declaration in case 1, the type of an expression in case 2, a creation instruction or expression in case 3 and so on), but only that this component **involves** *S*. The reason is that the type actually listed, instead of being *S* itself, could also include — as explained in the definition of “involved_in”, DTIN, page 169 — an Attachment_mark, for example by being of the form **detachable** *S*. Then *C* is a client of *S* (as well as of **detachable** *S*).

8.7.4.12 An important property of this definition of “simple client”, DLSC, is that it has **no associated validity constraint**. Any two classes of a system may be simple clients (and hence, more generically, clients) of one another. In particular, **cycles are permitted**: a class may be its own client, both directly (for example a declaration *x*: *C* appearing in the text of *C*) and indirectly (for example *x*: *S* in *C* and *y*: *C* in *S*).

8.7.4.13 The next two definitions introduce important special cases of the simple client relation (and hence of the client relation), extracted from the full list of cases in the above definition.

End

8.7.5 Definition: Expanded client

Code: *DLEC*

A class *C* is an **expanded client** of a type *S* if *S* is an expanded type and some attribute of *C* is of type *S*.

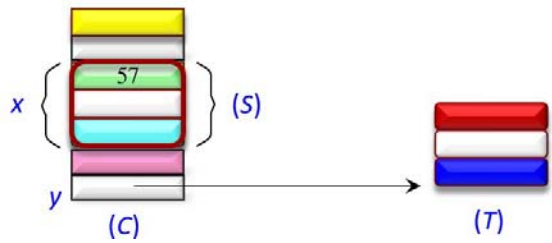
Informative Text

8.7.5.1 “Expanded client” is a special occurrence of case 1 of the above definition of “simple client” (or its transposition to “client” if the attribute declaration appears in a proper ancestor of *C*).

8.7.5.2 The reason for giving a name to this specific case is that it has a notable consequence on the run-time model. If *C* has an attribute declaration *y*: *T*, where *T* is a reference type, every instance of *C* at run time will have a field (corresponding to *y*) containing a reference to an object of type *T*. But with *x*: *S* where *S* is an expanded type, every instance will instead contain a **subobject** of type *S* (corresponding to *x*), as illustrated.

Signed by JV - 2026/10/8

8.7.5.3



8.7.5.4 This second case (*x* in the figure) is the one covered by the definition of “expanded client”.

8.7.5.5 The two declarations in the example make *C* a simple client of (respectively) *T* and *S*, but the second one makes it, in addition, an expanded client of *S*.

End

8.7.6 Definition: Generic client, generic supplier

Code: *DLGC*

A class *C* is a **generic client** of a type *S* if some ancestor of *C* is a simple client of *S* through case 4, 5 or 6 of the definition of that notion (DLSC).

Informative Text

8.7.6.1 This present definition parallels the original definition of “client” (DLSS, page 88), characterizing a “client” of *S* as a class of which an ancestor is a “simple client” of *S*; it specializes it to the case of being a client through a generic parameter. (The referenced definition of simple client, DLSC, appears above on page 89.)

8.7.6.2 The cases listed, cases 4, 5 or 6 from the definition of “simple client”, are those for which *C* becomes a client of *S* by involving *S* in an actual generic parameter of a generic derivation, or a generic constraint for a Formal_generic parameter.

End

8.7.7 Definition: Indirect client

Code: *DLCI*

A class *A* is an **indirect client** of a type *S* of base class *B* if there is a sequence of classes C_1 = *A*, C_2, …, C_n = *B* such that $n > 2$ and every C_i is a client of C_{i+1} for $1 \leq i < n$.

The indirect forms of the simple client, expanded client and generic client relations are defined similarly.

Informative Text

8.7.7.1 In mathematical terms, the “direct or indirect client” relation in its listed variants is the **transitive closure** of the respective variant of “client”. (Not the *reflexive* transitive closure: a class is not necessarily a client of itself. It can be, but only if one of the cases of the general definition, DLSC, page 89, applies. For example a declaration *x*: *C* appearing in the text of a non-generic class *C* makes C is a client of itself through case 1 of DLSC.)

End

Informative Text

8.7.7.2 The following subclauses apply to Clients parts and define Eiffel’s **selective export** mechanism. This mechanim supports a sophisticated form of information hiding, which makes it possible to exports features not just to all classes or no classes at all, but also to a set of specified classes

Signed by JV - 2026/10/8

(and their descendants). Eiffel is geared towards the construction of large systems that must sustain change and reuse. Selective support this goal by allowing a group of classes to share features of joint interest, without making them available to the rest of the universe.

8.7.7.3 A Clients part is a clause that qualifies a Feature_clause (see the specifications starting with SFPA, page 101 in the clause about features, 8.8). For example a sequence of feature declarations can start with:

feature {*A*, *B*, *C*}

… Declarations of some features …

where the Clients part is {*A*, *B*, *C*}, to specify that the features herein declared are available for calls (a notion defined precisely below, in DLES, page 96) to the classes listed (*A*, *B* and *C*) and their descendants. The notion of "client set" defined below formalizes this property.

End

8.7.8 Syntax: Clients

Code: *SLCL*

Clients ≜ "**{**" Class_list "**}**"

Class_list ≜ {Class_name "**,**" …}$^{+}$

Informative Text

8.7.8.1 As previewed above and detailed below, a Feature_clause declaring some features may start with **feature** {*A*, *B*, *C*, …}, where the list in braces is the Clients part, with the effect that calls to these features are only valid in the classes listed (*A*, *B*, *C*, …) and their descendants.

8.7.8.2 There is **no validity constraint** on Clients part. In particular, it is valid for a Clients part both:

8.7.8.2.1 • To list a class that does not belong to the universe (the environment specifying the available class names and associated classes, as defined in DSUV, page 78).

8.7.8.2.2 • To list a class twice.

8.7.8.3 That absence may be surprising at first since the language definition has little tolerance in general for potential inconsistencies (as they may reflect actual mistakes).In this particular context it does not enforce any restriction, for good reasons:

8.7.8.3.1 • The specific inconsistencies (non-existent class, class name listed twice) do not have any apparent potential for causing harm.

8.7.8.3.2 • Demanding that any class name (such as *A* above) listed in a Clients part correspond to an actual class would create unnecessary obstacles to reusability, a key objective of Eiffel and object technology. Specifically: if we have a class *C* with a Feature_clause listing *A* in its Clients part, and want to reuse *C* in a different system with a different universe not containing a class *A*, no damage may result (the Clients part simply says the feature cannot be exported to *A*, which is now pointless but harmless). If the language had a validity constraint requiring class *A* to exist, compiling *C* in the new system would trigger an error message and force a rewrite of *C* just for the purpose of its inclusion in that system in its original form. Variants of such classes would proliferate, differing only by mundane details such as their Clients parts, and defeating the undamental goal of reuse.

8.7.8.4 Even in a single system, this policy means that you can remove *A* — if you find it is no longer needed — without causing compilation errors in the classes that list *A* in their Clients parts. With a stricter rule, you would have to remove *A* from every such Clients part. But then if you later change your mind — as part of the normal back-and-forth of an incremental design process — you would have to put it back in each of these places.

End

Signed by JV - 2026/10/8

8.7.9 Definition: Client set of a Clients part or a Feature_clause

Code: *DLCP*

The **client set** of a Clients part is the set of descendants of every class of the universe whose name it lists. If the Clients part belongs to a Feature_clause, its client set is also called the client set of that Feature_clause.

By convention, the client set of an absent Clients part includes all classes of the system.

Informative Text

8.7.9.1 The descendants of a class (DHAD, page 145) include the class itself.

8.7.9.2 A Clients part such as {*A*, *B*, *C*} may belong to a Feature_clause, starting for example with **feature** {*A*, *B*, *C*}. The definition indicates that the client set of the Clients part (in this example, the set of classes containing the descendants of *A*, *B* and *C*) is also considered the client set of the Feature_clause as a whole.

8.7.9.3 The "convention" of this definition simplifies the following definitions in the case of no Clients part, which should be treated as if there were a Clients part listing just *ANY*, ancestor of all classes. (The actual type to which all types conform is, as seen above, **separate detachable** *ANY*.) This convention is in line with the rule that every class is a descendant of *ANY* (unfolded inheritance part of a class, DHUI, page 144 and the resulting Universal Conformance principle, VHUC, page 144).

8.7.9.4 As noted in the preceding Informative Text (8.7.8.3, 8.7.8.4), no validity *rule* prevents listing in a Clients part a name *n* that does not denote a class of the universe. In this case *n* does not denote any class and hence has no descendants; it does not contribute to the client set.

8.7.9.5 The notion of client set will be extended in DLCF, page 94 below to individual features (from its definition here for entire Clients parts).

End

Informative Text

8.7.9.6 The next subclauses address the New_exports clause, which optionally appears in Inheritance part and makes it possible to extend the export status of some inherited features (from the status they had in the parents from which they are being inherited).

End

8.7.10 Syntax: Export adaptation

Code: *SLEA*

New_exports ≜ **export** New_export_list

New_export_list ≜ {New_export_item ";" …}+

New_export_item ≜ Clients [Header_comment] Feature_set

Feature_set ≜ Feature_list | **all**

Feature_list ≜ {Feature_name "," …}+

Informative Text

8.7.10.1 We may for example start a class with

```
class C inherit
        B export
                {X, Y} f, g
                {Z} all
```


Signed by JV - 2026/10/8

to prescribe the new export status of features inherited from *B*: *f* and *g* are exported to classes *X*, *Y* and their descendants; all features from *B* are exported to *Z*.

End

8.7.11 Validity: Export List rule

Code: *VLEL*

A New_exports clause appearing in class *C* in a Parent part for a parent *B*, of the form

export
{*class_list*$_1$} *feature_set*$_1$
…
{*class_list*$_n$} *feature_set*$_n$

is valid if and only if for every *feature_set*$_i$ (for *i* in the interval *1..n*) that is a Feature_list (rather than **all**):

1 Every element of the list is the final name of a feature of *C* inherited from *B*.

2 No Feature_name appears more than once in any such list.

Informative Text

8.7.11.1 A New_exports clause is meant to adapt the *export status* of inherited features, with the precise effect described next. This validity rule ensures that it makes sense and is unambiguous.

End

Informative Text

8.7.11.2 The export status of a feature f of a class *C* determines whether a class *A* may include a qualified call to that feature: a call *x.f* (…) where x is of a type based on *C* (or a Non_object_call {*T*}*.f* (…) where *T* is *C* or based on *C*). For such a call to be valid, *f* must be "**available to**" *A* according to the rules given below. Defining the export status of a feature consists of specifying the set of classes to which it is available. The following subclauses will define the exact export status of a feature, which depends on whether it is:

8.7.11.2.1 • Immediate (defined in the class itself), in which case what counts is the Feature_clause that introduces it, more specifically its Clients part if present (otherwise the feature is exported to all classes of the universe through *ANY*).

8.7.11.2.2 • Inherited, in which case it retains its original export status unless it appears in a New_exports clause, which defines its new export status in the descendant class. In such a New_exports, the keyword **all** means that the chosen status will apply to all the features inherited from the given parent. If the feature is redeclared in the inheriting class, its new export status can be extended through the Feature_clause in which the redeclaration appears.

8.7.11.3 The definitions below formalize these notions. They start by extending the notion of "client set", defined above (DLCP, page 93) for entire Clients parts, to individual features.

End

8.7.12 Definition: Client set of a feature

Code: *DLCF*

The **client set** of a feature *f* of a class *C*, of final name *fname*, includes the following classes (for all cases that match):

1 If *f* is immediate or redeclared in *C*: the client set of the Feature_clause of the declaration for *f* in *C*.

2 If *f* is inherited: the union of (recursively) the client sets of all its precursors from conforming parents.

3 If the Feature_set of one or more New_exports clauses of *C* includes *fname* or **all**, the union of the client sets of their Clients parts.

Signed by JV - 2026/10/8

Informative Text

8.7.12.1 This definition is the principal rule for determining the export status of a feature. It has two important properties:

8.7.12.1.1 • The different cases are cumulative rather than exclusive. For example a "redeclared" feature (case 1) is also "inherited" (case 2) and the applicable Parent part may have a New_exports (case 3).

8.7.12.1.2 • As a result of case 2, **the client set can never diminish under conforming inheritance**: features can win new clients, but never lose any. This policy is necessary under polymorphism and dynamic binding to avoid certain type of "catcalls", run-time time errors which would lead to crashes.

End

8.7.13 Definition: Available for call, available

Code: *DLAV*

A feature *f* is **available for call**, or just **available** for short, to a class *C* or to a type based on *C*, if and only if *C* belongs to the client set of *f*.

Informative Text

8.7.13.1 There is also a notion of "available for creation", governing whether a **Creation_call** **create** *a*.*f* (…) is valid. "Available" without further qualification means "available for call".

8.7.13.2 In line with others in the present discussion, the definition of "available for call" introduces a notion about *classes* and immediately generalizes it to *types* based on those classes.

8.7.13.3 The notion of "available for call" fundamentally determines the information hiding policy by defining when a feature can be used on an object:

8.7.13.3.1 • A qualified call *x*.*f* (…) can only be valid if *f* is available to the type of *x* (the target of the call). The relevant rule is the key validity constraint on calls, export validity: VUEX, page 206.

8.7.13.3.2 • Similarly, for a Non_object_call {*T*}.*f* (…), *f* must be available to the base class of *T*. The rule here is the Non-Object Call rule, VUEX, page 206.

8.7.13.4 Note that a feature of a class is not necessarily available to the class itself. For example if *f* is declared in a Feature_clause of class *C* that starts **feature** {*NONE*} (or **feature** {*A*, *B*} where neither *A* nor *B* is an ancestor of *C*), then *f* is not available to *C*.

8.7.13.5 The key observation is that the above rules only apply to *qualified* variants of calls (whose default syntax is *x*.*f* (…) or {*T*}.*f* (…), with a period separating the denotation of the target object or type from the feature name and arguments). An Unqualified_call *f* (…) does not depend on the export status of *f*: it is **always** permitted (as long as it observes other conditions not related to information hiding, in particular argument validity, VUAR, page 204) from within the class, including if *f* is inherited from an ancestor (there is no information hiding from descendants). The semantics of such a call is to apply *f* to the current object, with the arguments given.

8.7.13.6 As a consequence it is possible that with *x* of type *C* in *C*, the call *f* (*a*) is valid and *x*.*f* (*a*) is not, *even* if at run time *x* happens to be attached to the current object, so that these two calls would (if both permitted) essentially have the same semantics. That situation will occur if *f* is not available to *C* itself (which is possible as noted above). Then even **Current**.*f* (*a*) is invalid since it is a qualified call.

8.7.13.7 The difference is that the unqualified call is always acceptable, defined as a call on the object itself (if it were not, non-exported features would not be usable at all and hence would not make sense), whereas a qualified call is a call on a *potentially* different object, to which information hiding rules apply (even if in some specific execution that object happens to be the current object).

8.7.13.8 The Unqualified_call *f* (*a*) and its qualified counterpart **Current**.*f* (*a*) (or in the second case *x*.*f* (*a*) where *x* happens to have the same value as **Current**) were characterized above (8.7.13.6) as having, if both valid, "*essentially*" the same semantics. There is in fact a small but significant

Signed by JV - 2026/10/8

difference: the qualified form may assume on entry that the **class invariant** holds, and must satisfy it on exit. No such condition applies to an unqualified call *f* (*a*), for which the notion of class invariant is not relevant. See in particular, in the discussion of contracts (a concept that includes class invariants), clause 3 of the rule on assertion monitoring, MAAM, page 139.

End

Informative Text

8.7.13.9 There are three degrees of availability, as given by the following definition.

End

8.7.14 Definition: Exported, selectively available, secret

Code: *DLES*

The export status of a feature of a class is one of the following:

1 The feature may be available to all classes. It is said to be **exported**, or **generally available**.

2 The feature may be available to specific classes (other than *NONE* and *ANY*) only. In that case it is also available to the descendants of all these classes. Such a feature is said to be **selectively available** to the given classes and their descendants.

3 Otherwise the feature is available only to *NONE*. It is then said to be **secret**.

Informative Text

8.7.14.1 This definition introduces the fundamental terminology for information hiding, which determines when it is possible to call a feature through a qualified call *x.f*. As special cases:

8.7.14.1.1 • A feature introduced by **feature** {*NONE*} (case 3) is available to no useful classes. It is secret.

8.7.14.1.2 • A feature introduced by **feature** {*ANY*}, or just **feature**, is available to all classes and so will be considered to fall under case 1: exported (generally available).

8.7.14.1.3 • A feature introduced by **feature** {*A, B, C*}, where none of {*A, B, C*} is *ANY*, falls under case 2: selectively available to the descendants of *A*, *B* and *C*.

8.7.14.2 A feature available to a class is also available to every proper descendant of that class. As a consequence, selective export is compatible with the goal of reusability: while the features will only be available to certain classes, these may be classes written much later, as long as they are descendants of one of the listed Clients.

End

Informative Text

8.7.14.3 The discussion so far has addressed the export status of features; in particular, it specifies that certain features are "**secret**", meaning informally that they are none of the business of other classes (or more precisely the programmers writing these classes). The following definition generalizes this notion to other properties of a class, such as feature bodies, rescue clauses and Parent parts.

End

8.7.15 Definition: Secret, public

Code: *DLSP*

A property of a class text is **secret** if and only if it involves any of the following, describing information on which client classes may not rely to establish their correctness:

1 Any feature that is not available to the given client, unless it is available for creation to it.

2 Any feature that is not available for creation, unless it is available to it.

3 The Feature_body and Rescue clause of any feature, except for the information that the feature is external or once and, in the last case, its once keys if any.

Signed by JV - 2026/10/8

4 For a query without formal arguments, whether it is implemented as an attribute or a function, except for the information that it is a constant attribute.

5 Any Assertion_clause that (recursively) includes secret information.

6 Any Parent part for a non-conforming parent (and as a consequence the very presence of that parent).

7 The information that a feature is frozen.

Any property of a class text that is not secret is **public**.

Informative Text

8.7.15.1 Software developers must be able to use a class as supplier on the basis of public information only.

8.7.15.2 The key property of the above definition of "secret" properties is the underlying goal: enabling each class to demonstrate its correctness. Classes depend on each other, but a property if secret if it cannot figure in such demonstrations for clients. To establish its correctness, a client of a class *S* may need to rely on properties of *S*, but only its *public* properties.

8.7.15.3 The next definitions will take advantage of the distinction between "public" and "secret" properties, as introduced here, to specify how we may document a class (with the help of language processing tools) to enable programmers to write classes that use it as clients.

8.7.15.4 A feature may be available for call, or for creation, or both (cases 1 and 2). If either of these properties applies, the affected clients must know about the feature, even if they can use it in only one of these two ways.

8.7.15.5 The implementation of a feature (Feature_body and Rescue clause) is one of the most obvious examples of information that must be secret (case 3). Authors of client classes are, however, entitled to know whether the feature is external, and also whether it is "once" (meaning that only an initial call will actually apply the feature's semantics, later ones returning the same result if any). "Once" has several variants, such as "once per thread", specified through "once keys" which must also be public information.

8.7.15.6 Case 4 expresses a major tenet of object technology and information hiding, the **Uniform Access principle**. The principle states that it must be possible to change the implementation of a query from attribute to function or conversely without affecting clients. ("*Clients should not have to know whether information is looked up or computed*.") For example, a feature yielding the balance of a bank account may be a function (computing the current balance from the record of operations on the account) or an attribute (looking up the balance in a field of the account object, where the implementation ensures that every operation updates that field). Uniform Access makes it possible for clients to use the same notation, such as *a.balance* (for *a* of type *ACCOUNT*) in all cases. The notion of query, covering both attributes and functions, is a consequence of the principle. The resulting flexibility only makes sense for features without arguments. As a special qualification to the rule, clients must know whether the feature is a *constant attribute*, since in that case they may use the feature in a Non_object_call. Being a constant attribute is, as a result, a public property.

8.7.15.7 In case 5, assertions characterizing a feature (precondition, postcondition) or a class (class invariant) as part of their contract are normally public information, but not their clauses that include secret information. (For the exact specification of this concept see DAAV, page 134.) For preconditions, a restriction exists, for reasons of soundness, on the inclusion of such secret clauses; see the Precondition Export rule, VAPE, page 133.

8.7.15.8 Case 6 indicates that the inheritance properties of a class are secret information. Clients need to know which features the class offers and with what specification (contracts), but it makes (and should make) no difference to them where each of them originated: in the class or in a proper ancestor.

Signed by JV - 2026/10/8

8.7.15.9 A “frozen” class, in case 7, is specified to prohibit redeclaration in descendants. Clients have no way to redeclare a feature — they just have to call it “as is”, through its contract — and hence have no need for or right to this particular property.

End

8.7.16 Definition: Incremental contract view, short form

Code: *DLIC*

The **incremental contract view** of a class, also called its **short form**, is a text with the same structure as the class but retaining only public properties.

Informative Text

8.7.16.1 This view provides information about the class that programmers can use to write client classes, which may use public properties only. It is “incremental” because it only considers properties expressed in the class text, ignoring inheritance. The features listed, for example, will only be the immediate and redeclared ones; other inherited features, obtained unchanged from parents, will not appear. The next definition introduces a view that takes inheritance into account by “flattening” the class.

8.7.16.2 Eiffel environments usually provide language processing tools which automatically produce the incremental contract view of a class from the class text. Such information provides an excellent form of software documentation: abstract yet precise, and extracted from the program text rather than written and maintained separately.

8.7.16.3 The definition specifies the information that the incremental contract view must retain, but not its exact display format under a particular tool, which typically will be close to Eiffel syntax.

End

8.7.17 Definition: Contract view, flat-short form

Code: *DLCV*

The **contract view** of a class, also called its **flat-short form**, is a text following the same rules as the incremental contract view but extended to include information about inherited as well as immediate features, the resulting combined precondition and postcondition and the unfolded form of the class invariant.

Informative Text

8.7.17.1 The contract view is the full interface information about a class, including everything that clients need to know (but no more) to use it properly.

8.7.17.2 The “combined forms” of preconditions and postconditions take into account parents’ versions as possibly modified by **require else** and **ensure then** clauses, and hence describing features’ contracts as they must appear to the clients. See DACP, page 135.

8.7.17.3 The “unfolded form” of the class invariant includes clauses from proper ancestors (meaning parents, their own parents and so on). In all these, of course, we still eliminate any clause that includes secret information, as with the incremental contract view. See SAUA, page 134.

8.7.17.4 Like its incremental variant above, the contract view should be available through language processing tools of Eiffel environments. It is the principal means of documenting Eiffel software, in particular libraries of reusable components. It provides the right mix of abstraction, clarity and precision, and excludes implementation-dependent properties. Being produced automatically by software tools from the actual text, it does not require extra effort on the part of software developers, and stands a much better chance of remaining accurate when the software changes.

End

Signed by JV - 2026/10/8

8.8 Features

Informative Text

8.8.0.1 A class is characterized by its features. Every feature describes an operation for querying or modifying instances of the class.

8.8.0.2 A feature is either an *attribute*, describing information stored with each instance, or a *routine*, describing an algorithm. Clients of a class *C* may apply *C*'s features to instances of *C* through **call** instructions or expressions.

8.8.0.3 Every feature has an identifier, which identifies it uniquely in its class. In addition, a feature may have an Alias to permit calls using Operator, Bracket or Parenthesis syntax.

8.8.0.4 The present subclause (8.8) introduces the various categories of feature, explains how to write feature declarations, and describes the form of feature names. After an informal introduction to the different kinds of feature (see the figure in 8.8.0.5 below and its explanations), the structure of the remaining part of the subclause is as follows:

8.8.0.4.1 • 8.8.1 to 8.8.5: general concepts (feature of a class, inherited versus introduced status of a feature, structure of a Features part and a Feature_declaration).

8.8.0.4.2 • 8.8.6 to 8.8.25: New_feature_list, the first of the two components of a Feature_declaration, which defines name or names for a feature or group of features with the same properties, and any special characteristic such as an Assigner_mark.

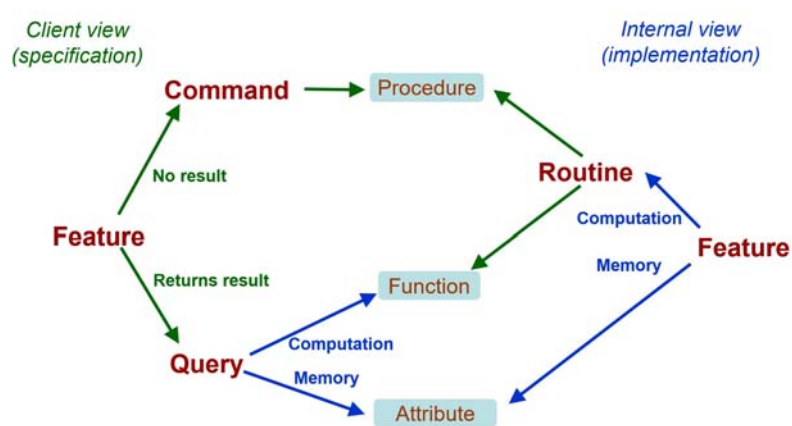
8.8.0.4.3 • 8.8.19 to 8.8.33: Feature_definition, the second component of a Feature_declaration, which defines the feature, including its signature (arguments if any, result type if any) and what it does if it is a routine. This part of the discussion concludes with an exact definition of the various categories of feature: attribute, routine, procedure, command, function, query.

8.8.0.4.4 • 8.8.35: the Feature Declaration Rule, an important validity constraint which brings together all the essential properties of features reviewed in the subclause.

End

Informative Text

8.8.0.5 The discussion in this subclause extensively uses the classification of features into various categories, introduced formally staring with 8.8.28, page 115. To facilitate comprehension, it is useful to consider the following graphical depiction.



8.8.0.6 The figure actually shows two classifications, leading to the same final categories in the center:

8.8.0.6.1 • From the viewpoint of a client (starting from the left), a feature either: does something to an object, and then is a command; or returns information on it, and then is a query. A command

Signed by JV - 2026/10/8

is implemented as a procedure (the terms are synonymous but “procedure” is more implementation-oriented while “command” is more conceptual). A query can be implemented by: computing information at run time from the fields of the object, in which case it is a function; or by storing the required value as a a field in every applicable run-time object (every instance of the class), in which case it is an attribute. (Some attributes are constant and do not need to take up storage in individual instance, but like variable attributes they are values conceptually associated with these instances).

8.8.0.6.2 • From the viewpoint of the implementer (starting from the right), a feature is implemented through either: computation, giving a function; or storage, giving an attribute. A routine can return a result (function) or not (procedure).

8.8.0.7 These two viewpoints are complementary and meet in the middle of the figure.

End

Informative Text

8.8.0.8 The first part of the discussion of features appears next (8.8.1 to 8.8.5), with the overall specification of a Features part and a Feature_declaration.

End

8.8.1 Definition: Feature of a class

Code: *DFFC*

The **features of a class** are its immediate features and its Inherited features.

Informative Text

8.8.1.1 The precise definitions of “immediate” and “inherited” appear next. Informally, a feature of a class *C* is inherited if it comes from a parent, although *C* might provide a *redeclaration*; it is immediate if it is declared in *C* as a new feature.

End

8.8.2 Definition: Inherited, immediate, origin, introduce, introduced

Code: *DFII*

A feature *f* of a class *C* is of one of the following two kinds:

1 If *C* obtains *f* from one of its parents, *f* is an **inherited** feature of *C*.

2 If a declaration appearing in *C* applies to a feature that is not inherited, the feature is **immediate** in *C*, and is also said to be “**introduced**” by *C*. Then *C* is the **origin** (short for “class of origin”) of *f*, and is said to **introduce** *f*.

Informative Text

8.8.2.1 This definition relies on the following concepts. A redeclaration is a declaration appearing in a class *C* for a feature *f* that *C* inherits from a parent (or several parents in the case of repeated inheritance), which changes some of the properties of *f* in the context of *C*. The details of redeclaration appear in the study of inheritance (subclause 8.11, page 143); what is important here is that a declaration in the Features part only introduces a new feature (“immediate” in *C*, or “introduced” by *C*) if it is not a redeclaration of some feature obtained from a parent.

8.8.2.2 Every feature of a class is immediate either in the class or in one of its proper ancestors (parents, grandparents and so on).

8.8.2.3 The definition above is a simple characterization of whether a feature is “inherited”, as opposed to “immediate”. It is complemented by a more detailed definition of “the inherited features of a class”, DMIF, page 161. coming after the full description of inheritance.

8.8.2.4 A related notion is **redeclaration**. If *C* contains a feature declaration for an inherited feature, that declaration is either a redefinition (replacing the version in the corresponding parent class) or an effecting (providing an implementation for a feature that was deferred in the parent). In both cases

the new declaration (redefinition or effecting) is called a redeclaration of the feature; the definition introducing this term officially is SDRD, page 149, part of the subclause on feature adaption (8.12). In addition to these two cases of explicit redeclaration of an inherited feature *f* (through a new declaration in *C*), we also say — again as part of SDRD — that *C* redeclares *f* if it undefines it.

End

8.8.3 Syntax: Feature parts

Code: *SFPA*

Features ≜ Feature_clause+

Feature_clause ≜ **feature** [Clients] [Header_comment] Feature_declaration_list

Feature_declaration_list ≜ {Feature_declaration ";" …}*

Header_comment ≜ Comment

Informative Text

8.8.3.1 The Features part is, typically, the most visible part of a class text (although, like all other parts of a Class_declaration except for the Class_header and the final **end**, it is optional). It consists of one or more Feature_clause. As an example, the following simple class text only includes, along with a simple Creation_clause, a Features part with two Feature_clause, respectively introducing two and one features (note the indentation following the recommended style rules):

8.8.3.2

```
class C create
        set
feature -- Access
        x: INTEGER
        t: REAL
feature -- Element change
        set (a: INTEGER) do x := a end
end
```

8.8.3.3 Within a Feature_clause, as part of a general syntactical convention, semicolons are **optional** between a Feature_declaration and the next; the example omits them between the declarations of *x* and *t*.. The recommended style rule suggests omitting them except in the infrequent case of two successive declarations on a single line.

8.8.3.4 The Features part contains the declarations of all the features of a class. The reason for splitting it into a succession of Feature_clause is to allow Eiffel programmers to maintain well-structured, readable, extendible classes by organizing their features into logical groups, each defined by a Feature_clause.

8.8.3.5 Individual feature declarations each appear in a Feature_declaration. As seen below (SFNL, page 102), such a declaration can actually introduce several features, all with the same properties.

8.8.3.6 The optional Header_comment at the start of a Feature_clause is part of this policy, enabling programmers to label each category. The Eiffel style rules list a number of standard categories, of which the ones in the above example, “Access” and “Element change” are among the most frequently used. Basic libraries systematically use these conventions to organize their classes.

End

8.8.4 Definition: Overview of feature categories

Code: *DFOV*

Every feature of a class is either an *attribute* or a *routine*.

An attribute is either *constant* or *variable*.

A routine is either a *procedure* or a *function*.

Signed by JV - 2026/10/8

Informative Text

8.8.4.1 Successive definitions in the rest of this subclause (8.8) introduce each of these notions precisely, making it possible to recognize easily, from a valid Feature_declaration, which kind of feature it declares.

End

8.8.5 Syntax: Feature declarations

Code: *SFDE*

Feature_declaration ≜ New_feature_list Declaration_body

Informative Text

8.8.5.1 A Feature_declaration introduces a feature description, through its Declaration_body, for at least one feature name, listed in the New_feature_list. If more than one, they all have the same definition.

8.8.5.2 *Validity*: to be valid, a Feature_declaration must satisfy a number of conditions. These conditions depend on the kind of feature (attribute, routine etc.) as reviewed in the rest of this subclause (8.8). The full list of conditions follows from that review and constitutes the conclusion of the subclause: Feature Declaration rule, VFDE, page 119.

8.8.5.3 The two components of a Feature_declaration, New_feature_list and Declaration_body, are reviewed in turn below (respectively . The first one introduces feature names and the second one specifies feature properties.

End

Informative Text

8.8.5.4 The second part of the discussion of features appears next (8.8.6 to 8.8.25), with the specification of the first of the two components of a Feature_declaration, the New_feature_list, which introduces the names of one or more features declared together with their signature (for routines and queries) and special properties such as emergence, variance and any associated assigner command.

End

8.8.6 Syntax: New feature lists

Code: *SFNL*

New_feature_list ≜ {New_feature_name "," …}+

New_feature_name ≜ [Emergence_mark] [Variance_mark] Extended_feature_name

Emergence_mark ≜ **then**

Variance_mark ≜ **variant** | **invariant**

Informative Text

8.8.6.1 Having a list of features, rather than just one, makes it possible for example to declare together several features with the same properties. This possibility is particularly useful in two cases:

8.8.6.1.1
- Declaring two or more attributes of the same type.
- Declaring two or more routines with the same body, making them “synonyms” of each other (DFSY, page 103).

8.8.6.2 Here as a consequence are the three ways of declaring distinct features:

8.8.6.2.1
- Within a given class, declare them in a different Feature_clause (each grouping features of the same general category, such as all features for accessing a certain kind of data structure).

Signed by JV - 2026/10/8

8.8.6.2.2 • Within a given Feature_clause, declare them in a different Feature_declaration (each grouping features with the same Feature_declaration defining their properties, including type for a query and body for a routines).

8.8.6.2.3 • Within a given Feature_declaration, list them with a different New_feature_name in the New_feature_list, making them different features although with the same properties.

8.8.6.3 An Extended_feature_name as defined below (SFNA, page 104) is a feature name possibly equipped with an Alias. Aliases make it possible to use a given feature under different syntax forms: for example a feature with the name *plus* can have "+" as an alias (specified as part of its Extended_feature_name), making it possible to use the feature not only in the standard object-oriented style *a*.*plus* (*b*) but also with operator syntax as *a* + *b*.

8.8.6.4 The optional Variance_mark (either **variant** or **invariant**) and Emergence_mark (simply **then**) are only applicable only to variables (non-constant attributes and local variables).They play different roles as explained now.

8.8.6.5 The Variance_mark governs polymorphism:

8.8.6.5.1 • A variable specified **variant** may also become attached to objects of different types, as long as those types ("dynamic types") conform to the static type. This is the defaul; (in other words if the Variance_mark is absent, the variable will be considered variant).

8.8.6.5.2 • A variable specified **invariant** may only become attached to objects of the exact type declared ("static type"). This restriction is enforced by the rules on attachment from expressions to variables (see clause 3 of expression conformance, DNEC, page 189).

8.8.6.6 The Emergence_mark, consisting simply of the keyword **then**, helps with void safety (the topic of subclause 8.30, page 254). Declared with this mark, an attribute is called "emergent" (DVEA, page 257). Like an attribute of an attached type, an emergent attribute can only be assigned non-void expressions, but unlike attached attributes it does not need to be initialized to an attached value; it can start out void. As a consequence, it can be used in Certified Attachment Patterns (DVCA, page 257). This situation is not the default (attributes declared without **then** are not emergent).

End

8.8.7 Definition: Synonym

Code: *DFSY*

A **synonym** of a feature of a class *C* is a feature with a different Extended_feature_name such that both names appear in the same New_feature_list of a Feature_declaration of *C*.

8.8.8 Definition: Unfolded form of a possibly multiple declaration

Code: *DFUD*

The **unfolded form** of a Feature_declaration listing one or more feature names, as in:

$f_1, f_2, \ldots, f_n$ *declaration_body* ($n \geq 1$)

where each f_i is a New_feature_name, is the corresponding sequence of declarations naming only one feature each, and with identical declaration bodies, as in:

f_1 *declaration_body*

f_2 *declaration_body*

...

f_n *declaration_body*

Informative Text

8.8.8.1 Thanks to the unfolded form, we may always assume, when studying the validity and semantics of feature declarations, that each declaration applies to only one feature name. The language description, throughout, relies on this convention: to define both the validity and the semantics, it simply refers to the unfolded form, which may give several declarations even if they are all grouped in the class text.

8.8.8.2 A multiple declaration introduces the feature names as synonyms. But the synonymy only applies to the enclosing class; there is no permanent binding between the corresponding features. Their only relationship is to have the same Declaration_body at the point of introduction.

8.8.8.3 This means in particular that a proper descendant of the class may rename or redeclare one synonym without affecting the others.

8.8.8.4 Each f_i, being a New_feature_name, may include an **invariant** mark. In the unfolded form this mark only applies to the corresponding f_i and its corresponding declaration (the i-th one).

End

8.8.9 Definition: Invariant variable

Code: *DFIV*

A variable is **invariant** if and only if it is declared with the Variance_mark **invariant**.

Informative Text

8.8.9.1 The definition only applies to "variables" because they are the only features that can be declared **invariant** (condition 9 of VFDE, page 119).

8.8.9.2 An invariant variable can only be assigned an expression of the same exact type. See condition 3 of the definition of "expression conformance", DNEC, page 189, governing the definition of "compatibility" used by in the validity rule for attachment, VBAS, page 239.

End

Informative Text

8.8.9.3 The next definitions and rules describe the mechanism allowing us to may give a feature an "alias", typically an operator, to allow notations such as *a* + *b* or *reverse_lookup* ["123-4567"] or *integral* [*cosine*, *a*, *b*] as alternatives to the default form of feature call.

End

8.8.10 Syntax: Feature names and aliases

Code: *SFNA*

Extended_feature_name ≜ Feature_name [Aliases]

Feature_name ≜ Identifier

Aliases ≜ {Alias "," …}+

Alias ≜ **alias** '"' Alias_name '"'

Alias_name ≜ Operator | Bracket | Parenthesis

Bracket ≜ "**[]**"

Parenthesis ≜ "**()**"

Informative Text

8.8.10.1 Every feature has a Feature_name *f* — an Identifier (this is a formal property appearing below: PFFI, page 107). This identifier, say *f*, makes it possible for clients to call the feature using object-oriented dot notation: *x*.*f* (…).

8.8.10.2 In addition, Aliases, if specified, make it possible to call the feature (if it is a function) in other ways, with a syntax taken from classical mathematical notation. An Alias is one of:

8.8.10.2.1 • An Operator alias, making it possible to call the feature in the form *a* § *b* for some operator §, for example *a* + *b,* with the same semantics as *a*.*f* (*b*). This facility is particularly useful for arithmetic-style expressions.

8.8.10.2.2 • A Bracket alias, making it possible to call the feature in the form *a* [*b*], with the same semantics as *a*.*f* (*b*). A class may have at most one feature with a Bracket alias. This facility is particularly

Signed by JV - 2026/10/8

useful for such indexable data structures as arrays and hash tables, as in *my_array* [*i*] and *my_hash_table* ["*my_key*"] as abbreviations for *my_array*.*item* (i) and *my_hash_table*.*item* ("*my_key*"), assuming that in both cases *item* is the function with the bracket alias in the corresponding class (*ARRAY* or *HASH_TABLE*).

8.8.10.2.3 • A Parenthesis alias, making it possible to call the feature in the form *a* (*b*), with the same semantics as *a*.*f* (*b*). A class may have at most one feature with a Parenthesis alias. This facility is particularly useful for dealing with object representing functions, often represented by Eiffel agents; for example a routine for numerically integrating a function *f* will need to evaluate it at various points x; thanks to the Parenthesis alias in the library classes *FUNCTION* and *PREDICATE* (representing function agents with, in the second case, boolean return values) it may take advantage of the usual mathematical notation *f* (*x*) instead of *f*.*item* (*x*).

8.8.10.3 There is no validity constraint on Extended_feature_name. A feature may have any number of aliases. Any anomaly (such as giving the same alias to different features) will be caught as a violation of the constraint on Alias: Alias Validity rule, VFAV, page 105 below; see the corresponding Informative Text comment, 8.8.12.4.

End

8.8.11 Syntax (non-production): Alias Syntax rule

Code: *SFAL*

The Alias_name in an Alias must immediately follow and precede the enclosing double quote symbols, with no intervening characters (in particular no breaks).

When appearing in such an Alias_name, the two-word operators **and then** and **or else** must be written with exactly one space between the two words.

Informative Text

8.8.11.1 In general, breaks or comment lines may appear between components prescribed by a BNF-E production, making this rule necessary to complement the grammar: you must write **alias "+"**, not **alias " + "**.

8.8.11.2 The two parts of the rule are cumulative, implying that between the double quotes in **alias "and then"** (respectively **alias "or else"**) there may be no other characters than those in the words **and** and **then** (respectively **or** and **else**) and the single space between them.

End

Informative Text

8.8.11.3 Next comes the validity rule for aliases, ensuring that the abbreviated forms for calls in each case — *a* § *b* for an operator §, *f* [*i*], for a feature *f* with a bracket alias, *f* (*i*) for a feature *f* with a parenthesis alias — make sense.

End

8.8.12 Validity: Alias Validity rule

Code: *VFAV*

An Alias clause is **alias-valid** for a feature *f* of a class *C* if and only if it satisfies the following conditions:

1 If it lists an Operator *op*: *f* is a query; no other Alias clause for a query of *C* with the same number of arguments has an Operator alias using the same operator; and either: *op* is a Unary and *f* has no argument, or *op* is a Binary and *f* has one argument.

2 If it lists a Bracket alias: *f* is a function with at least one argument, and no other Alias clause for a feature of *C* has a Bracket alias.

3 If it lists a Parenthesis alias: *f* is a function with at least one argument, and no other Alias clause for a feature of *C* has a Parenthesis alias.

Signed by JV - 2026/10/8

4 If it includes a **convert** mark: it lists an Operator and *f* has one argument.

Informative Text

8.8.12.1 The first three conditions express the uniqueness and signature requirements on operator, bracket and parenthesis aliases:

8.8.12.1.1 • (Condition 1.) An operator feature *plus* **alias** "§" can be either unary (called as § *a*) or binary (called as *a* § *b*), and so must take either zero or one argument. Two features may indeed share the same alias— like *identity* **alias** "+" and *plus* **alias** "+", respectively unary and binary addition in class *INTEGER* and others from the Kernel Library — as long as they have different identifiers (here *identity* and *plus*) and different signatures, one unary and the other binary. In this example the two alias clauses respectively enable notations such as + *a* and *a* + *b*.

8.8.12.1.2 • (Condition 2.) A bracket feature, of which condition 2 tells us a class has at most one, will be called under the form *x* [a_1, …, a_n] with $n \geq 1$, and so must be a function with at least one argument (only routines, including functions, can have arguments). Typical applications include access to an array, as in *my_array* [27], and access to a hash table, as in *my_hash_table* ["*my_key*"].

8.8.12.1.3 • (Condition 3.) A parenthesis feature, of which condition 3 tells us a class also has at most one, will be called under the form *x* (a_1, …, a_n) with $n \geq 1$, and so must be a routine with at least one argument. A typical application would be **Result** := **Result** + *delta* ∗ *f* (*a* + *i* ∗ *delta*) in an integration routine (approximating the integral of *f* on an interval [*a, b*] by a sum of areas of successive rectangles, each with *delta* as its width and , as its height, the value of *f* at the *i*-th subinterval of width *delta*).

8.8.12.2 Note that a bracket feature must be a function (returning a result), whereas a parenthesis feature may be any routine — a procedure or a function.

8.8.12.3 Condition 4 indicates that a **convert** mark, specifying "target conversion" as in *your_integer* + *your_real*, makes sense only for features with one argument, with an Operator which (from condition 1) must be a Binary.

8.8.12.4 The phrasing of the conditions prohibiting two features from having the same alias with the same number of arguments is specific. Condition 1 does not state that, for a given number of arguments, *no other query* can have the same operator as alias, but that *no other* Alias *clause* can use it. This wording prohibits an Extended_feature_name (declaring one or more features for a Feature_declaration) from using the same operator alias twice (which would be probably harmless, but confusing). Conditions 2 and 3 use similar phrasing for the same reason. It avoids the need to specify a validity rule (to forbid repetitions) for New_feature_list (see the corresponding Informative Text comment: 8.8.10.3 above).

End

8.8.13 Definition: Operator feature, brackets or parenthesis feature, identifier-only

Code: *DFOB*

A feature is an **operator feature** if its Extended_feature_name *fn* includes an Operator alias, a **bracket feature** if *fn* includes a Bracket alias, a **parenthesis feature** if *fn* includes a Parenthesis alias. It is **identifier-only** if neither of these cases applies.

Informative Text

8.8.13.1 The most common case is identifier-only. The other kinds provide convenient modes of expression ("*syntactic sugar*") for some cases where a shorter form, compatible with traditional mathematical conventions, is desirable for calling the feature.

8.8.13.2 When referring to feature names, some syntax rules use the Extended_feature_name, and some use the Feature_name, which is just the identifier, dropping the Alias if any. The criterion is simple:

8.8.13.2.1 • To identify the feature, in particular within the class itself and its descendant, the Feature_name is sufficient. As the Feature Identifier principle below expresses formally, the Feature_name uniquely identifies the feature.

8.8.13.2.2 • The Extended_feature_name only serves to give more syntactical flexibility for qualified **calls** in clients of the class. (Occasionally in the class itself, if it calls its own features and hence becomes its own client, but most commonly in other classes.)

8.8.13.3 In the class itself and its proper descendants, the Extended_feature_name is used in only two cases: to declare the feature, in a Feature_declaration, and to change its name for a proper descendant, in a Rename clause (for both inheritance and constrained genericity).

8.8.13.4 This observation also means that in descendants of its original class a feature will retain its Alias, if any, unless a descendant explicitly renames it to a name that may drop the Alias, or provide a new one. In particular, redeclaring a feature does not affect any Alias it may have.

End

8.8.14 Definition: Identifier of a feature name

Code: *DFIN*

The Identifier that starts a Extended_feature_name is called the **identifier of** that Extended_feature_name and, by extension, of the associated feature.

8.8.15 Principle: Feature Identifier principle

Code: *PFFI*

Given a class *C* and an identifier *f*, *C* contains at most one feature identifier of *f*.

Informative Text

8.8.15.1 This validity property is not a separate constraint but follows from other rules.

8.8.15.2 It reflects a critical characteristic of sound object-oriented programming in general and Eiffel in particular: no "*overloading*" of feature names within a class. A class defines a mapping between feature names and features. "Mapping" is just another word for "function" (as this word is used in mathematics): for one feature name, there is at most one feature of that name. Different classes may of course use the same feature name — opening the way for dynamic binding in the presence of polymorphism — but within a class one name means one thing.

End

8.8.16 Definition: Same feature name, same operator, same alias

Code: *DFSO*

Two feature names are considered to be "**the same feature name**" if and only if their identifiers have identical lower names.

Two operators are "**the same operator**" if and only if they have identical lower names.

An Alias in an Extended_feature_name is "**the same alias**" as another if and only if they satisfy the following conditions:

- They are one of: the same operator; both Bracket; both Parenthesis.
- If either has a **convert** mark, so does the other.

Informative Text

8.8.16.1 The "lower name" is the name written in all lower-case letters (D7UL, page 70). Per DFSO, *my_name*, *MY_NAME* and *mY_nAMe* are all considered to be the same feature name.

8.8.16.2 The recommended style uses:

8.8.16.2.1 • A name with an initial capital and the rest in lower case (as in *Planck_constant*) for constant attributes.

8.8.16.2.2 • The lower name, all in lower case (as in *my_name*) for all other features.

8.8.16.3 If letters appear in operator feature names, letter case is also irrelevant when it comes to deciding which feature names are the same and which different.

Signed by JV - 2026/10/8

8.8.16.4 The notions introduced in DFSO are useful in particular to enforce the rule that, in any class, there can be only one feature of a given name (no “overloading”), and to determine what constitutes a “*name clash*” under multiple inheritance. In such cases the language rules simply ignore letter case.

End

Informative Text

8.8.16.5 What symbols are suitable as Operator aliases? The next syntax production gives the answer. It is flexible, since it allows both common arithmetic and logical operators and almost arbitrary symbols that a programmer selects as a “free”operator.

End

8.8.17 Syntax: Operators

Code: *SFOP*

Operator ≜ Unary | Binary

Unary ≜ **not** | "+" | "–" | Free_unary

Binary ≜ "+" | "–" | "*" | "/" | "//" | "\\" | "^" | "**.**" |
"<" | ">" | "<=" | ">=" |
and | **or** | **xor** | **and then** | **or else** | **implies** |
Free_binary

Informative Text

8.8.17.1 Free operators (always binary) enable developers to define their own operators with considerable latitude. This is particularly useful in scientific applications where it is common to define special notations, which Eiffel will render as prefix or infix operators.

8.8.17.2 Eiffel supports the huge Unicode character set, giving programmers a wide choice.

8.8.17.3 You may for example define operators such as **, **|–|** (maybe as an infix alias for a *distance* function), or various forms of arrow such as **<–>**, **–|–>**, **=>**. With the introduction of Unicode symbols such operators made of several simple symbols are less critical than in older versions of Eiffel, which were restricted to shorter characters sets (ASCII or Extended ASCII).

End

8.8.18 Principle: Feature principle

Code: *PFAI*

Every feature has an associated identifier.

Any valid call (qualified or unqualified) to the feature can be expressed through this identifier.

Informative Text

8.8.18.1 This syntactic property follows from syntax rules.

8.8.18.2 The syntactic variants, available through **alias** clauses, offer other ways to express calls, reconciling object-oriented structure with other notations as seen above: operator alias, bracket alias, parenthesis alias. The principle reminds us that these notations are only syntactical variants introduced for convenience, but that the semantic mechanism is always a call:

8.8.18.2.1 • If a feature of identifier *plus* is declared in a class *C* with an operator alias "+" and *a* is of a type *T* based on *C*, then *a* + *b* is permitted, but is just a syntactical variant for *a*.*plus* (*b*), which remains valid.

8.8.18.2.2 • If a feature of identifier *item* is declared in a class *C* with a bracket alias and *your_table* is of a type *T* based on *C*, then *your_table* [*i*] is permitted, but is just a syntactical variant for *your_table*.*item* (*i*), which remains valid.

Signed by JV - 2026/10/8

8.8.18.2.3 • If a feature of identifier *call* is declared in a class *C* with a parenthesis alias and *your_function* is of a type *T* based on *C*, then *your_function* (*i*) is permitted, but is just a syntactical variant for *your_function*.*call* (*i*), which remains valid.

End

Informative Text

8.8.18.3 The third part of the discussion of features appears next (8.8.19 to 8.8.33), with the specification of the second one of the two components of a Feature_declaration, the Declaration_body, which defines what a feature denotes. It is always present for routines, but may also appear for attributes to specify particular properties such as an attribute's initialization.

End

8.8.19 Syntax: Declaration bodies

Code: *SFDB*

Declaration_body ≙ [Formal_arguments]
[Query_mark]
[Obsolete]
[Header_comment]
[Feature_definition]

Informative Text

8.8.19.1 All components of the are optional, but at least one of them must be present (specifically, a Query_mark or a Feature_definition or both), as specified by the validity rule below (VFDB).

8.8.19.2 Two of the components making up a Declaration_body per the syntax are discussed in other subclauses of this Standard:

8.8.19.2.1 • Formal_arguments, only appears for features that are routines, and is presented in the discussion of routines (SRFE, page 121).

8.8.19.2.2 • An Obsolete mark can be applied to classes as well as features, and was presented in SCOM, page 86. Including it in a Declaration_body specifies that the feature or features, while still usable, are "deprecated": using them is no longer advised. Usually there is a recommended replacement, as specified by the Message contained in the Obsolete mark.

8.8.19.3 The other three are presented next.

8.8.19.4 A Query_mark (the syntax appears below in SFQM, page 110) gives a type for the feature, introduced by a colon. It is applicable to attributes and functions (together known as queries), as in the following example declarations:

- *an_attribute*: *T*
- *double* (*n: INTEGER*): *INTEGER* **do Result** := 2 ∗ *n* **end** -- A function

Their Query_mark is respectively *": T"* and *": INTEGER"*.

8.8.19.5 A Query_mark can also include, before the colon, an Assigner_mark (to indicate that the query has a matching assigner command, as specified in VFAC, page 113).

End

8.8.20 Validity: Declaration Body rule

Code: *VFDB*

A Declaration_body is valid if and only if it includes at least one of:

1 A Query_mark.

2 A non-empty Feature_definition.

Signed by JV - 2026/10/8

Informative Text

8.8.20.1 If there is a Query_mark, the declaration introduces a query, either an attribute or a function as in the last two examples. A Query_mark (the syntax appears below in SFQM, page 110) gives a type for the feature, introduced by a colon. It is applicable to attributes and functions (together known as queries), as in the example declarations given above (8.8.19.4 and 8.8.21.4).

8.8.20.2 If there is no Query_mark, there must be something else to define the feature, so the Feature_definition, whose own definition appears below (SFDF, page 113) also consists of all optional components, needs to have at least one.

End

8.8.21 Syntax: Query marks

Code: *SFQM*

Query_mark ≜ [Assigner_mark] ":" Result_type

Result_type ≜ Precursor_type | Explicit_result_type

Explicit_result_type ≜ Type

Informative Text

8.8.21.1 A Query_mark gives a type. It is applicable to attributes and functions, in which return a result. (In contrast with procedures, which do not.) Typical examples using a colon were given above (8.8.19.4):

- *an_attribute*: *T*
- *double* (*n*: *INTEGER*): *INTEGER* **do Result** := 2 ∗ *n* **end**

8.8.21.2 An Assigner_mark, of the form **assign** *c*, is useful for a feature *q* which is a query without arguments, if *c* sets the value of *q* and you want to allow the assignment-like syntax *x*.*q* := *v* (known as an Assigner_call) as a shorthand for a call expressed in standard object-oriented style as *x*.*c* (*v*). The discussion of assigner calls starts with SBAC, page 241.

8.8.21.3 The syntax of the Precursor_type construct is defined in SDPR, page 150. Such a type is written **Precursor** {*Parent_A*} or just **Precursor** {*Parent_A*}. Using such a Precursor_type as type in a declaration is useful in a redeclaration of an inherited feature, and only permitted in that case (condition 1 of VDPS, page 150). The purpose is to avoid declaring a result type explicitly if it is the same as in the original one in the parent, specifying instead that you wish to retain the original type in the parent. If there is any ambiguity as to which parent you mean (because of repeated inheritance), use the full form **Precursor** {*Parent_A*}. In most practical cases there is no ambiguity, so you can just use **Precursor**, but the full form is still recommended for clarity (and to detect error if the inheritance structure of the class changes).

8.8.21.4 Precursor_type is also used in contexts other than feature declarations: for routine arguments (see comments after SRFE, page 121) and in the body of a redefined routine (SDPR, page 150 and following clauses).

End

8.8.22 Definition: Result type of a query

Code: *DFRT*

The **result type** of a query *q* whose Query_mark has the Result_type *t* is, if *t* is an Explicit_result_type, the corresponding Type, and if *t* is of the Precursor_type form the result type (recursively) of *q*'s precursor feature.

Signed by JV - 2026/10/8

Informative Text

8.8.22.1 The Precursor_type form is available for a query being redeclared; the "precursor feature" is the original version; see the Precursor signature rule, VDPS, page 150. The redeclaration rule (VDRD, page 154) and signature conformance rule (VNCS, page 190) imply that the precursor feature of a query is also a query, so that its own result type is defined.

End

8.8.23 Definition: Signature, argument signature of a feature

Code: *DFSA*

The **signature** of a feature *f* is a pair *argument_types*, *result_type* where *argument_types* and *result_type* are the following sequences of types:

- For *argument_types*: if *f* is a routine, the possibly empty sequence of its formal argument types, in the order of the arguments; if *f* is an attribute, an empty sequence.
- For *result_type*: if *f* is a query, a one-element sequence, whose element is the type of *f*; if *f* is a procedure, an empty sequence.

The *argument_types* part is the feature's **argument signature**, and *result_type* the part its **result signature**.

Informative Text

8.8.23.1 Examples (using square brackets for pairs and angle brackets for sequences):

- For *power* (*x*: *INTEGER*; *n*: *REAL*): *REAL* **do** … **end**: [<*INTEGER*, *REAL*>, <*REAL*>] (a function with two arguments).

8.8.23.1.1
- For *merge* (*other*: *PARAGRAPH*) **do** … **end**: [<*PARAGRAPH*>, <>] (a procedure with one argument).
- For *merge* (*other*: **like Current**) **do** … **end**: [<**like Current**>, <>] (similar to previous one, but with an anchored argument type).

8.8.23.1.2
- For *height*: *REAL*: [<>, <*REAL*>] (a query implemented as an attribute).

8.8.23.1.3
- For *height*: *REAL* **do** … **end**: [<>, <*REAL*>] (this time implemented as a function, but the signature is the same).

8.8.23.1.4
- For *hyphenate* **do** … **end**: [<>, <>] (procedure with no argument).

8.8.23.1.5
- For *first_projection* (*x*: *TUPLE* [*A*, *B*]): *A* **do** … **end**: [<*TUPLE* [*A*, *B*]>, <*A*>].

8.8.23.1.6
- For *reversed* (*x*: *TUPLE* [*A*, *B*]): *TUPLE* [*B*, *A*]) **do** … **end**: [<*TUPLE* [*A*, *B*]>, <*TUPLE* [*B*, *A*]>].

8.8.23.2 The argument signature is an empty sequence for attributes and for routines without arguments.

8.8.23.3 The result signature always has either zero element (for a command) or one elements (for a query). To obtain the effect of a query returning several resulting values, use a tuple as the result type, as illustrated in the last example (8.8.23.1.6); the result signature is a sequence with only one element, the result type.

End

Informative Text

8.8.23.4 The next few definitions and rules cover the Assigner_mark which may be included as part of the Query_mark in the declaration of a query. as in

item **assign** *put*: *T*

Signed by JV - 2026/10/8

where the Assigner_mark is **assign** *put*. It assumes that *put* (*a*: *T*) is a command of the same class, which sets the value of the query *item*, and makes it possible to set this value to (say) *v* not just through the usual call syntax *x*.*put* (*v*) but also, for convenience, through the assignment-like syntax *x*.*item* := *v*.

End

8.8.24 Syntax: Assigner marks

Code: *SFAM*

Assigner_mark ≜ **assign** Feature_name

Informative Text

8.8.24.1 An Assigner_mark appears as part of a Query_mark, as in the example Feature_declaration used above, where *item* is an attribute:

item **assign** *put*: *T*

8.8.24.2 or a variant in which *item* is a function rather than an attribute:

item **assign** *put*: *T* **do** … Code computing the value of *item* … **end**

8.8.24.3 In an Assignment *x* := *v* the target *x* must be a variable. If *item* is a query (attribute or function) whose type *T* is the same as the type of *a*, programmers not well-versed in programming methodology may be tempted to write an assignment such as

a.*item* := *v*

to assign directly to the corresponding object field. As an Assignment, this instruction is not permitted as it goes against all the rules of data abstraction and object technology. The normal mechanism is for the author of the base class of *T* to provide a "setter" command (procedure), say *put*, enabling the clients to use *a*.*put* (*v*) to set the value of *item*.

8.8.24.4 Programmers who do understand the principles of programming methodology may, however, have their syntactic cake and eat it without risking semantic poisoning. If you do want to permit *a*.*item* := *v* as a convenient assignment-like shorthand for the call *a*.*put* (*v*), you can. It suffices to mark *put* as an **assigner command** associated with *item*. Then *a*.*item* := *v* is permitted as a synonym for the call.

8.8.24.5 Such an instruction looks like an assignment but it is not. It is just a different notation for a procedure Call and is known as an **assigner call** (syntactically, a specimen of the Assigner_call construct, one of the variants of Instruction, described in the subclauses starting with SBAC, page 241).

8.8.24.6 This scheme, a notational simplification only, is also convenient for features that have a Bracket or Parenthesis alias:

8.8.24.6.1 • In library classes describing arrays, the feature *item* for accessing an element by its index has a bracket alias and an Assigner_mark, allowing full-fledged traditional syntax for both array access and array modification: an assigner call *a* [*i*] := *v* as shorthand for a call *a*.*put* (*v*, *i*), but also *a* [*i*] := *a* [*i*] + 1, formally a shorthand for *a*.*put* (*a*.*item* (*i*) + 1, *i*).

8.8.24.6.2 • This possibility is not limited to integer indexes. We may, for example, using hash tables, write *phone_number* ["John"] := "123-456-7890"

8.8.24.6.3 • Similarly, for *a* of a type *T* offering a query *f* with a parenthesis alias and an associated assigner command *set*, an assigner call *a* (*i*) := *v* is a shorthand for a call *a*.*set* (*v*, *i*).

8.8.24.7 The mechanism is applicable not just to attributes but (in line with the Uniform Access principle) to all queries, including functions with arguments.

8.8.24.8 The following rule defines under what conditions you may, as author of a class, permit such assigner calls from your clients by associating an assigner command with a query.

End

Signed by JV - 2026/10/8

8.8.25 Validity: Assigner Command rule

Code: *VFAC*

An Assigner_mark appearing in the declaration of a query *q* with *n* arguments ($n \geq 0$) and listing a Feature_name *fn*, called the **assigner command** for *q*, is valid if and only if it satisfies the following conditions:

1 *fn* is the identifier of a command *c* of the class.

2 *c* has *n* + 1 arguments.

3 The type of *c*'s first argument and the result type of *q* have an identical deanchored form.

4 For every *i* in 1**..***n*, the type of the *i*+1-st argument of *c* and the type of the *i*-th argument of *q* have an identical deanchored form.

5 *c* is available to any class to which *q* is available.

Informative Text

8.8.25.1 The feature *q* can only be a query since, from the syntax of Declaration_body, an Assigner_mark can only appear as part of a Query_mark, whose presence makes the feature a query. "Query", as a reminder, means attribute or function: a feature whose evaluation produces a value (as opposed to a command, which simply produces an effect on the execution state).

8.8.25.2 "De-anchoring", in cases 3 and 4, takes care of "anchored types", turning them into an explicit form. An anchored type is of the form **like** *x*, for some entity *x* and means "the same type as *x*". If *x* is of type *T*, then the deanchored form of the type **like** *x* is simply *T* (possibly deanchored recursively). See the discussion starting with DTAN, page 169 in the subclause on types.

8.8.25.3 In cases 3 and 4, we require the types (more precisely their deanchored forms) to be **identical**, not just compatible (converting or conforming). To understand why, recall that the assignment *a***.***item* := *y* is only a shorthand for a call *a***.***put* (*x*) implemented, in the simplest case, as:

item: *T* **assign** *put* **do** … **end**

put (*b*: *U*) **do** … *item* := *b* … **end**

8.8.25.4 Now assume that *U* is not identical to *T* but only compatible with it, and consider the procedure call

*a***.***put* (*a***.***item*)

or the equivalent assignment form

*a***.***item* := *a***.***item*

which are in principle useless — they reassign to a field its own value — but have no reason to be prohibited. They become invalid, because the source *a***.***item* (actual argument of the call or right side of the assignment) is of type *T*, the target (the formal argument) of type *U*, and it is generally impossible for two different types to be each compatible with the other.

8.8.25.5 These observations explain clause 3: the first argument of the assigner procedure must have *exactly* the same type as the result of the query (once both have been deanchored). Similar reasoning applied to other arguments, if any, leads to clause 4.

End

Informative Text

8.8.25.6 The following definitions and rules specify how we can express the properties of a feature, through its Feature_definition.

End

8.8.26 Syntax: Feature definitions

Code: *SFDF*

Feature_definition

Signed by JV - 2026/10/8

≜ [Precondition]
[Local_declarations]
[Feature_body]
[Postcondition]
[Rescue]
[**end**]

Feature_body ≜ Deferred | Effective

Effective ≜ Effective_routine | Constant_attribute | Defined_attribute

Constant_attribute ≜ "**=**" Manifest_constant

Informative Text

8.8.26.1 Feature_definition is the basic way of defining object properties and operations in Eiffel.

8.8.26.2 All component are optional, even the final **end**. They can all be absent, as in this exampleof a Feature_declaration:

an_attribute: *T*

The feature declared in this case is, per DFVA, page 115 below, a variable attribute.

8.8.26.3 The Feature_body, if present, determines a form of feature, one of:

8.8.26.3.1 • A deferred function, with no implementation.

8.8.26.3.2 • An effective routine, with an implementation specifying an algorithm.

8.8.26.3.3 • An attribute, indicating that a value is associated with every instance of the class. (The declaration of *an_attribute* above is an abbreviation for a form including a Feature_definition with just a Feature_body of the Defined_attribute form: **attribute end**.) The attribute can be variable, as in the case of *an_attribute*, or it can be constant, using an Constant_attribute, as in

Maximum: *INTEGER* = 100

8.8.26.4 The following discussion covers these various cases. The principal validity rule of this chapter, the Feature Declaration rule (VFDE, page 119) will use their definitions to specify which combinations of the components of a Feature_declaration (of which the Feature_definition is part) permitted by the above syntax are acceptable.

End

Informative Text

8.8.26.5 First comes a simple consistency constraint on the Feature_definition itself.

End

8.8.27 Validity: Feature Definition rule

Code: *VFDF*

Feature Definition rule

A Feature_definition appearing as part of a Declaration_body *db* is valid if and only if it satisfies the following conditions:

1 It has the final **end** if and only if it has at least one of the other components.

2 If it has a Feature_body of the Deferred or Constant_attribute kind, it has no Local_declarations.

3 It has a Rescue if and only if the Feature_body is of the Effective_routine or Defined_attribute kind.

Signed by JV - 2026/10/8

Informative Text

8.8.27.1 The final **end** only makes sense (condition 1) if there is some algorithmic element (code or contract) associated with the feature. It is required in this case, and only permitted in this case. There is no **end**, for example, in a no-frills attribute declaration *x*: *T*.

8.8.27.2 The Local_declarations and Rescue parts only make sense (condition 2) for a feature with an associated algorithm present in the class text itself; this means a routine that is neither deferred nor external, or an attribute with explicit initialization.

8.8.27.3 The Rescue part, specifying exception handling, only makes sense (condition 3) for if there is an algorithmic implementation.

8.8.27.4 As noted, the present rule, VFDF, only defines basic conditions on which components of a Feature_definition can appear together. There are also more elaborate conditions: for example an Constant_attribute only makes sense for a constant attribute and excludes, for example, the feature having any arguments. Such conditions cannot be determined on the basis of the Feature_definition alone; they need the context of the entire Feature_declaration and appear in the final rule of this subclause 8.8: VFDE, page 119.

8.8.27.5 The general idea will be to define categories of feature introduced by a Feature_declaration, based on the components that actually appear (out of all the possible ones) in its components, including not only the Feature_definition but also its Extended_feature_name. These categories, previewed in the informal presentation and figure at the beginning of this subclause (starting with 8.8.0.5, page 99) include attribute (in one of its variants: variable attribute, constant attribute and defined attribute), procedure and function. (More general terms are query, covering attributes and functions, command, covering procedures, and routine, covering procedures and functions.) Then the Feature Declaration rule, VFDE, will simply state that a Feature_declaration must fit one of these kinds.

8.8.27.6 The definitions below are, as a consequence, a little more than definitions: they collectively yield a validity requirement complementing the Feature Definition rule.

End

Informative Text

8.8.27.7 Now for the different kinds of feature, based on what components are present, or not, in a Feature_definition.

End

8.8.28 Definition: Attribute

Code: *DFVA*

A Feature_declaration is an **attribute** declaration if and only if it is the declaration of one of: variable attribute, constant attribute, defined attribute.

Informative Text

8.8.28.1 The three cases are defined next.

End

8.8.29 Definition: Variable attribute

Code: *DFVA*

A Feature_declaration is a **variable attribute** declaration if and only if it satisfies the following conditions:

1 There is no Formal_arguments.

2 There is a Query_mark.

3 There is no Constant_attribute.

Signed by JV - 2026/10/8

4 If there is a Feature_body, it is of the Defined_attribute kind.

Informative Text

8.8.29.1 Variable attributes provide the basic building blocks of run-time data (object) structures. Declaring a variable attribute *a* of type *T* in a class *C* specifies that at run time every instance of *C* will contain a field corresponding to *a*, of type *T* (reference or expanded).

8.8.29.2 The most common way of declaring an attribute is simply to list its type, as in the Feature_declaration

x: T

8.8.29.3 It is also possible to include other parts, such as a Feature_body which specifies an initialization to be performed for the attribute on object creation, or a Postcondition. The following example (an Defined_attribute) has both of these components:

```
Five: INTEGER
        attribute
                Result := 5
        ensure
                Result /= another_attribute
        end
```

8.8.29.4 Note how both the body and the postcondition use the keyword **Result** to denote the value of the attribute (the corresponding object field).

End

8.8.30 Definition: Constant attribute

Code: *DFCA*

A Feature_declaration is a **constant attribute** declaration if and only if it satisfies the following conditions:

1 It has no Formal_arguments.

2 It has a Query_mark.

3 There is a Feature_body of the Constant_attribute kind.

Informative Text

8.8.30.1 A constant attribute is, like a variable attribute, a value associated with every instance of the class where it is declared, but that value is the same for all, so it does not need to occupy space (in the form of a field) in every instance.

8.8.30.2 The declaration of a constant attribute specifies the value, as in

8.8.30.2.1 • *Pi*: *REAL* = 3.141592

8.8.30.2.2 • *Name*: *STRING* = "*HELLO*"

8.8.30.2.3 • *Integer_lists*: *TYPE* = {*LINKED_LIST* [*INTEGER*]}

8.8.30.3 The value coming after the equal sign is a Manifest_constant; the various possibilities for manifest constants are specified after the definition of this construct (S3MA, page 266), in the subclause on constants (8.34). The three examples above are among the most important cases: a constant of a basic type, here REAL; a string constant; and a constant representing a type of the program itself.

8.8.30.4 A constraint governs constant attributes; it applies to the entire Feature_declaration in which a constrant attribute is declared (particularly the type declared in the Query_mark) and appears below as the Constant Attribute Validity rule, VFCA, page 118.

End

Signed by JV - 2026/10/8

8.8.31 Syntax: Defined attributes

Code: *SQBO*

Defined_attribute ≜ **attribute** Compound

Informative Text

8.8.31.1 An attribute is “defined” if it comes with a Feature_body (of the Effective kind which itself is of the Defined_attribute kind).

8.8.31.2 Most commonly, a variable attribute does not have a Feature_body but is simply declared with a type, as in

```
x: T
```

8.8.31.3 It is possible, however, to associate an algorithm (for initialization) as well as contract elements (precondition, postcondition), as in

```
Five: INTEGER
        attribute
                Result := 5
        ensure
                Result /= another_attribute
        end
```

8.8.31.4 This form, using the keyword attribute, is a Defined_attribute declaration.

8.8.31.5 The non-defined form is equivalent to a defined form with an empty compound; an equivalent declaration for x above is

```
x: T attribute end
```

8.8.31.6 The defined form, with a non-empty Compound is required for an attribute of an attached type (including the case of an expanded type) that does not provide *default_create* as a creation procedure; it will then serve to initialize the corresponding field, on first use for any particular object, if that use occurs prior to an explicit initialization. To set that first value, assign to **Result** in the Compound. Such a Compound is executed at most once on any particular object during a system execution. The basic rule specifying this behavior is the Variable Semantics rule, MEVS, page 249, part of the discussion of object semantics in subclause 8.25.

Informative Text

8.8.31.7 After attributes, the discussion now covers the other main category of features, routines and their variants.

End

8.8.32 Definition: Routine, function, procedure

Code: *DFRK*

A Feature_declaration is a **routine** declaration if and only if it satisfies the following condition:

- There is a Feature_body of the Deferred or Effective_routine kind.

If a Query_mark is present, the routine is a **function**; otherwise it is a **procedure**.

Informative Text

8.8.32.1 Example routine declarations (taken from the examples in the Informative text starting with 8.8.23.1) are:

8.8.32.1.1 - *power* (*x*: *INTEGER*; *n*: *REAL*): *REAL* **do** … **end**: a function with two arguments.

8.8.32.1.2 - *merge* (*other*: *PARAGRAPH*) **do** … **end**:): a procedure with one argument.

8.8.32.1.3 - *height*: *REAL* **do** … **end**: a function without arguments.

8.8.32.1.5 - *hyphenate* **do** … **end**: : a procedure without arguments

8.8.32.1.6 - *first_projection* (*x*: *TUPLE* [*A*, *B*]): *A* **do** … **end**: a function with one argument.

Signed by JV - 2026/10/8

8.8.32.2 For a routine, the Formal_arguments (like the Query_mark) may or may not be present.

8.8.32.3 By convention this definition, DFRK, treats a deferred feature as a routine, even though its effectings in proper descendants may be, in the case of a query, attributes as well as functions.

End

8.8.33 Definition: Command, query

Code: *DFCQ*

A **command** is a procedure. A **query** is an attribute or function.

Informative Text

8.8.33.1 These notions underlie two important principles of the Eiffel method:

8.8.33.1.1 • The Command-Query separation principle, which suggests that queries should not change objects.

8.8.33.1.2 • The Uniform Access principle, which enjoins, whenever possible, to make no distinction between attributes and argument-less functions.

8.8.33.2 The terms “command” and “procedure” are synonyms. Elsewhere, this Standard sticks to the discipline of assigning just one name to every relevant concept; here, however, both terms are appropriate, and both are widely used. “Procedure” is more implementation-oriented (denoting a kind of routine, along with “function”); “command” is more abstract (denoting a kind of operation on objects, along with “query”).

End

Informative Text

8.8.33.3 The fourth and last part of the discussion of features appears next with the conditions (rules 8.8.34 and 8.8.35) defining the validity of a Feature_declaration. A previous validity rule, the Feature Definition rule, VFDF, page 114, defined local constraints on a Feature_definition; but the full conditions need to put the Feature_definition in the full context of its enclosing Feature_declaration since they involve all components, such as the Extended_feature_name (particularly if it indicates an Alias), type if any (in a Query_mark), the presence or absence of Assigner_mark and Variance_mark, and the feature category (variable_attribute, constant attribute, defined attribute, function, procedure) as covered by the preceding definitions.

8.8.33.4 The Constant Atrribute rule, VFCA, page 118, covers the validity of constant attributes. The general rule, bringing together all the key properties of features discussed in the present subclause 8.8 on features, is the Feature Declaration rule, VFDE, page 119.

End

8.8.34 Validity: Constant Attribute Validity rule

Code: *VFCA*

A feature *f* introduced in a Feature_declaration defining a constant attribute is **constant-valid** if and only if the Manifest_constant *m* used in the declaration matches the type *T* declared for *f* in one of the following ways:

1 *m* is a Boolean_constant and *T* is *BOOLEAN*.

2 *m* is a Character_constant and *T* is one of the sized variants of *CHARACTER* for which *m* is a valid value.

3 *m* is an Integer_constant and *T* is one of the sized variants of *INTEGER* for which *m* is a valid value.

4 *m* is a Real_constant and *T* is one of the sized variants of *REAL* for which *m* is a valid value.

Signed by JV - 2026/10/8

5 *m* is a Manifest_string and *T* is one of the sized variants of *STRING* for which *m* is a valid value.

6 *m* is a Manifest_type, of the form {*Y*} for some type *Y,* and *T* is *TYPE* [*X*] for some stand-alone type *X* to which *Y* conforms.

Informative Text

8.8.34.1 The "valid values" are determined by each basic type's semantics; for example 1000 is a valid value for *INTEGER_16* but not for *INTEGER_8*.

8.8.34.2 Conditions 1 to 5 apply to constants of the basic types, plus type *STRING* (not considered a "basic type" since it is a library class but sharing many properties of basic types), covered in the corresponding subclause (8.35, page 271): *BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL* and *STRING*.

8.8.34.3 In cases 2 to 5, these types have "sized variants", which denote specific bit lengths making it possible to know the exact mathematical counterparts of the corresponding types. Examples are *CHARACTER_8*, *INTEGER_16*, *REAL_64* and *STRING_32*.

8.8.34.4 Condition 6 requires the type listed in a Manifest_type {*Y*} to be *constant*, meaning that it does not involve any formal generic parameter or anchored type, as these may represent different types in different generic derivations or different descendants of the original class. Such a type would not be suitable for a constant attribute, which must have a single, well-defined value.

8.8.34.5 A companion rule in the subclause on constants, V3CA, page 266, determines the use of a constant attribute in the class text, particularly in expressions.

End

8.8.35 Validity: Feature Declaration rule

Code: *VFDE*

A Feature_declaration appearing in a class *C* is valid if and only if it satisfies all of the following conditions for every declaration of a feature *f* in its unfolded form:

1 The Declaration_body describes a feature which, according to the respective definitions, is one of: attribute, procedure, function.

2 *f* does not have the same feature name as any other feature introduced in *C* (in particular, any of the other features listed in the unfolded form).

3 If *f* has the same feature name as the final name of any inherited feature, the satisfies the Redeclaration rule.

4 If the Declaration_body describes a deferred feature, the feature is not invariant.

5 If the Declaration_body describes a constant attribute, it is constant-valid.

6 If the Declaration_body describes a once function other than once per object, the result type is stand-alone.

7 Any anchored type for an argument is detachable.

8 The Alias clause, if present, is alias-valid for *f*.

9 If a Variance_mark is present, the Variance_mark describes a variable attribute whose type is a Class_type.

10 If an Emergence_mark is present, the Declaration_body describes a variable attribute.

Informative Text

8.8.35.1 As stated at the beginning of the rule, the conditions apply to the *unfolded form* of the declaration. Using the unfolded form — which was defined earlier in the present subclause in DFUD, page 103 — means that the rule treats a multiple declaration $f_1, f_2, \ldots, f_n$ *declaration_body* as a succession of *n* separate declarations with different feature names but the same *declaration_body*.

Signed by JV - 2026/10/8

8.8.35.2 Condition 1 is the fundamental rule taking advantage of the preceding definitions: the Declaration_body must in its Declaration_body, particularly the Feature_definition, define a feature that makes sense for the applicable feature category.

8.8.35.3 Condition 2 is also straightforward: the name or names of the features being introduced must not conflict with those of any other feature introduced in the class. This condition is the key rule chasing away the sinister specter of overloading, threatening the foundations of sound programming.

8.8.35.4 Condition 3 complements condition 2 by prohibiting any name overloading with features inherited from parent classes classes as part of inheritance.

8.8.35.5 In applying conditions 2 and 3, remember that two feature names are "the same feature name" not just if they are written identically, but also if they only differ by letter case. Only the identifiers (Feature_name) of the features play a role in this notion, not any Alias they may have.

8.8.35.6 The discussion of repeated inheritance in subclause 8.13, page 159, will introduce a consequence of the Feature Declaration rule: the Feature Name rule, VMFN, page 162. In the case of reporting a violation (a name clash between two features of a class), compilers may choose between citing clause 2 or 3 or of the present rule, VFDE, or (perhaps more clearly) a clause of VMFN.

8.8.35.7 Condition 4 prohibits an invariant feature from being declared as deferred. A feature was is "invariant" if it has the Variance_mark **invariant** (DFIV, page 104). The two properties are conceptually incompatible since invariant features, by definition, may not be redeclared, whereas the purpose of deferred features is precisely to force redeclaration in every effective proper descendant.

8.8.35.8 Condition 5 governs constant attributes and refers to the preceding rule, VFCA, page 118.

8.8.35.9 Condition 6 applies to once functions. A once routine (see the precise specification of the associated semantics in subclause 8.19 on calls, starting with MUON, page 208) only executes its body on its first call. Further calls have no effect; for a function, they yield the result computed by the first call. Being "once"puts a special requirement on the result type *T* of such a function: if the class is generic, *T* should not depend on any formal generic parameter, since successive calls could then apply to types obtained from different generic derivations; and *T* must not be anchored (that is to say, of the form **like** *x*, denoting the same type as for an entity *x*), as in the context of dynamic binding it could yield incompatible types depending on the type of the target (meaning *x* in the example) of each particular call. The notion of stand-alone type (STSA, page 171) captures these constraints on *T*.

8.8.35.10 Condition 7 addresses delicate cases of polymorphism and dynamic binding, where anchored arguments and their implicit form of "covariance" may cause run-time errors known as "catcalls". It follows from the general rule for signature conformance and is discussed with it, starting with VNCS, page 190.

8.8.35.11 Condition 8 states the condition on Alias clauses, as specified by the earlier validity rule on aliases, VFAV, page 105. An operator, bracket of parenthesis alias may only be given to a query (a function in the last two cases), with the appropriate number of arguments.

8.8.35.12 Condition 9 expresses that a feature may only be declared **variant** or **invariant** (a Variance_mark) if it is a variable attribute. Such marks are also applicable to local variables of routines; see SRFE, page 121, and Informative Text that follows it.

8.8.35.13 Similarly, condition 10 expresses that a feature may only be declared **then** (an Emergence_mark), making it "emergent", if it is a variable attribute. Unlike the preceding one, this qualification is not applicable to local variables.

End

Signed by JV - 2026/10/8

8.9 Routines

Informative Text

8.9.0.1 Routines describe computations.

8.9.0.2 Syntactically, routines are one of the two kinds of feature of a class; the other kind is attributes, which describe data fields associated with instances of the class. When applied to an instance of a class, a routine may query or update some or all fields of the instance, corresponding to attributes of the class.

8.9.0.3 A routine is either a procedure, which does not return a result, or a function, which does. A routine may further be declared as **deferred**, meaning that the class introducing it only gives its specification, leaving it for descendants to provide implementations. A routine that is not deferred is said to be **effective**.

8.9.0.4 An effective routine has a **body**, which describes the computation to be performed by the routine. A body is a Compound, or sequence of instructions; each instruction is a step of the computation.

8.9.0.5 The present discussion explores the structure of routine declarations, ending with the list of possible various forms of instructions.

End

8.9.1 Definition: Actual argument, actual

Code: *DRAA*

A routine call may pass to the routine the values of a list of expressions, the call's **actual arguments** (or just **actuals**).

Informative Text

8.9.1.1 The counterpart in the routine's declaration is a list of formal arguments, a notion defined precisely below. For example the declaration may start with *r* (*x*: *PERSON*; *m*, *n*: *INTEGER*), so that the formal argument are *x*, *m* and *n*. An example call is *r* (*Jane*, 50, *b* + 3), with the actual arguments *Jane*, 50 and *b* + 3.

8.9.1.2 The actual arguments are the elements of the Actual_list in the syntax of Call; see SUAA, page 179.

8.9.1.3 The rules on Call require the number of actual arguments to be the same as the number of formal arguments, and the type of each actual argument to be compatible with (conform or convert to) the type of the formal argument at the same position in the list. (These rules apply after any "tuple argument unfolding", whereby you can drop the brackets around a single actual argument that is a tuple, writing for example *r* (*t*, *u*, [*a*, *b*]) as just *r* (*t*, *u*, *a*, *b*); see DJTA, page 163.)

8.9.1.4 Terminology: Eiffel always uses the term **argument** to refer to the arguments of a routine. The word "parameter" is never used in this context, because it could create confusion with the types that can parameterize *classes*, called **generic parameters**.

End

8.9.2 Syntax: Formal argument and entity declarations

Code: *SRFE*

Formal_arguments ≜ "(" Entity_declarations ")"

Entity_declarations ≜ Precursor_type | Entity_declaration_list

Entity_declaration_list ≜ {Entity_declaration_group ";" …}+

Entity_declaration_group ≜ Entity_list ":" Result_type

Entity_list ≜ {Identifier_with_variance "," …}+

Identifier_with_variance ≜ [Variance_mark] Identifier

Signed by JV - 2026/10/8

Informative Text

8.9.2.1 As with other semicolons, those separating an Entity_declaration_group from the next are optional. The style guidelines suggest including them for successive declarations on a line, as with short formal argument lists, but omitting them between successive lines, as with local variable declarations (also covered by Entity_declaration_group).

8.9.2.2 Usually a Formal_arguments is a list of declarations, such as *p*: *PERSON*; *m*, *n*: *INTEGER*. It is also possible, in the redeclaration of an inherited routine, to use the Precursor_type form, which reads just **Precursor** in simple cases, but may also list a class name, as in **Precursor** {*Parent_A*}, to remove a possible ambiguity in the case of multiple inheritance. Such a form indicates that the redeclaration retains the argument signature of the original redeclaration. The advantage is not only brevity but also the guarantee that if the signature changes in the original declaration the redeclaration' signature will automatically follow.

8.9.2.3 The Precursor_type form is also available for the result of a query: see comments after SFDE, page 82. In addition, you may use a Precursor_call in the body of a redefined routine to denote a call to its original version. The rules on Precursor_call appear in SDPR, page 129 and the clauses that follow it.

8.9.2.4 A The Variance_mark (repeating comments from SFNL, page 83) governs polymorphism:

8.9.2.4.1 • An entity specified as **invariant** may only become attached to objects of the exact type declared ("static type"). This restriction is enforced by the rules on attachment (DVEA, page 232 and DVCA, page 232).

8.9.2.4.2 • An entity specified as **variant** may become attached to objects of different types, as long as those types ("dynamic types") conform to the static type. This is the default (in other words if the Variance_mark is absent, the variable will be considered variant).

End

8.9.3 Definition: Unfolded form of a Formal_arguments

Code: *DRUF*

The unfolded form of a Formal_arguments *fa* is, if *fa* is of the Precursor_type kind, the unfolded form (recursively) of its precursor feature, and otherwise *fa* itself.

Informative Text

8.9.3.1 The "precursor feature" (defined in the Precursor signature rule, VDPS, page 129) is the feature being redeclared. Using Precursor_type in lieu of an explicit argument declaration means that you want to retain the precursor's argument signature. In this case, the unfolded form gives the list of formal arguments (see the following definition).

End

8.9.4 Definition: Formal argument, formal

Code: *DRFA*

The **formal arguments** (or just "formals ") of a routine are the elements of the concatenation of every Entity_declaration_group in the unfolded form of its Formal_arguments, if any.

8.9.5 Validity: Formal Argument rule

Code: *VRFA*

The Formal_arguments part of a routine declaration is valid if and only if every formal argument is a fresh identifier.

Informative Text

8.9.5.1 An identifier is fresh (D7FI, page 66) if it is not already used in the enclosing context; here the rule means a formal argument may not have the same name as a feature of the class.

8.9.5.2 The next rule, Entity Declaration, specifies that the formal arguments must also be all different.

Signed by JV - 2026/10/8

8.9.5.3 Another rule, given later, applies the same conditions to names of local variables. Permitting a formal argument or local variable to bear the same name as a feature could only cause confusion (even if there were a scoping rule removing any ambiguity by specifying that the local name overrides the feature name) and serves no useful purpose.

End

8.9.6 Validity: Entity Declaration rule

Code: *VRED*

Let *el* be an Entity_declaration_list. Let *identifiers* be the concatenation of every Entity_declaration_group in *el*. Then *el* is valid if and only if no Identifier [SHOULD THIS BE ENTITY?] appears more than once in the list *identifiers*.

8.9.7 Syntax: Routine bodies

Code: *SRBO*

Deferred ≜ **deferred**

Effective_routine ≜ Internal | External

Internal ≜ Routine_mark Compound

Routine_mark ≜ **do** | Once

Once ≜ **once** ["("Key_list")"]

Key_list ≜ {Key "," …}+

Key ≜ Manifest_string

8.9.8 Definition: Once routine and their varieties, once keys

Code: *DRON*

A **once routine** is an Internal routine *r* with a Routine_mark of the Once form.

If *r* is a procedure it is also a **once procedure**; if *r* is a function, it is also a **once function**.

The **once keys** of a once routine *r* are the strings appearing in its Key_list. If one of them Local is "*OBJECT*", "*THREAD*" or "*PROCESS*", *r* is (respectively) **once per object**, **once per thread**, **once per process**.

8.9.9 Syntax: Local variable declarations

Code: *SRLV*

Local_declarations ≜ **local** [Entity_declaration_list]

Informative Text

8.9.9.1 For convenience, there is a separate syntactical construct covering entities appearing in the Entity_declaration_list of a Local_declarations list. It is called Local and defined in the clause on objects (SEEN, page 245). It also covers predefined local variables **Result** and **Retry**.

End

8.9.10 Validity: Local Variable rule

Code: *VRLV*

Let *ld* be the Local_declarations part of a routine *r* in a class *C*. Let *locals* be the concatenation of every Entity_declaration_group in *ld*. Then *ld* is valid if and only if every Identifier *e* in *locals* is fresh.

Informative Text

8.9.10.1 An identifier is fresh (D7FI, page 66) if it is not already used in the enclosing context; here the rule means a local variable may not have the same name as a feature of the enclosing class or a formal argument of the enclosing routine.

8.9.10.2 Most of the rules governing the validity and semantics of declared local variables also apply to two special predefined local variables: **Result**, which may only appear in a function or attribute, where it denotes the value to be returned; and **Retry**, used in exception handling. The following definition of "local variable" reflects this similarity.

End

8.9.11 Definition: Local variable, declared local variable

Code: *DRLV*

The **local variables** of a routine (also called just "**locals**") include:

1 All entities declared in its Local_declarations part, if any. They are the **declared local variables** of the routine (also, "declared locals").

2 **Result** (usable in a query).

3 **Retry**.

Informative Text

8.9.11.1 **Result** can appear only in a query: not just in the Compound, Postcondition or Rescue of a function or variable attribute but also in the optional Postcondition of a constant attribute, where it denotes the value of the attribute and makes it possible to state abstract properties of that value, for example after a redefinition. In all these cases the type of **Result** is the type of the enclosing query.

8.9.11.2 **Retry**, of type *BOOLEAN*, supports exception handling, by indicating whether a routine's code is being executed again after an exception triggered a Rescue clause.

End

8.9.12 Syntax: Instructions

Code: *SRIN*

Compound ≜ {Instruction ";" …}*

Instruction ≜ Creation_instruction |
Call |
Assignment |
Assigner_call |
Conditional_instruction |
Multi_branch |
Loop |
Debug |
Precursor_call |
Check

Informative Text

8.9.12.1 A Compound is a possibly empty list of instructions, to be executed in the order given. In the various parts of control structures, such as the branches of a Conditional_instruction or the body of a Loop, the syntax never specifies Instruction but always Compound, so that you can include zero, one or more instructions.

8.9.12.2 A Creation_instruction creates a new object, initializes its fields to default values, calls on it one of the creation procedures of the class (if any), and attaches the object to an entity.

8.9.12.3 A Call executes a routine. For the Call to yield an instruction, the routine must be a procedure.

8.9.12.4 An Assignment changes the value attached to a variable.

8.9.12.5 An Assigner_call is a procedure call written with an assignment-like syntax, as in *x*.*a* := *b*, but with the semantics of a call, as just a notational abbreviation for *x*.*set_a* (*b*) where the declaration of *a* specifies an assigner command *set_a*.

8.9.12.6 Conditional_instruction, Multi_branch, Loop and Compound itself describe control structures, made out of instructions; to execute a control structure is to execute some or all of its constituent instructions, according to a schedule specified by the control structure.

8.9.12.7 Debug, which may also be considered a control structure, is used for instructions that should only be part of the system when you enable the *debug* compilation option.

8.9.12.8 Precursor_call enables you, in redefining a routine, to rely on its original implementation.

8.9.12.9 Check is used to express that certain assertions must hold at certain moments during run time. It many have a **then** part, in which case it can also be considered a control structure.

End

Signed by JV - 2026/10/8

PART E: Contracts

Signed by JV - 2026/10/8

Signed by JV - 2026/10/8

8.10 Correctness and contracts

Informative Text

8.10.0.1 Eiffel software texts — classes and their routines — may be equipped with elements of formal specification, called **assertions**, expressing correctness conditions.

8.10.0.2 Assertions play several roles: they help in the production of correct and robust software, yield high-level documentation, provide debugging support, allow effective software testing, and serve as a basis for exception handling. With advances in formal methods technology, they open the way to proofs of software correctness.

8.10.0.3 Assertions are at the basis of the **Design by Contract** method of Eiffel software construction.

8.10.0.4 The present clause successively presents:

8.10.0.4.1 • The basic assertion concepts and constructs (8.10.1 to 8.10.9).

8.10.0.4.2 • The effect of inheritance on contracts (8.10.10 to 8.10.23).

8.10.0.4.3 • Notions of correctness (8.10.24 to 8.10.29).

8.10.0.4.4 • Run-time effect of contracts (from 8.10.30).

End

8.10.1 Syntax: Assertions

Code: *SAAS*

Precondition ≜ {Individual_precondition …}*

Postcondition ≜ {Individual_postcondition …}*

Invariant ≜ {Individual_invariant …}*

Individual_precondition ≜ **require** [**else**] Assertion

Individual_postcondition ≜ **ensure** [**then**] Assertion

Individual_invariant ≜ **invariant** Assertion

Assertion ≜ [Key] Assertion_clause_list [Implication]

Assertion_clause_list ≜ {Assertion_clause ";" …}*

Assertion_clause ≜ [Tag_mark] Unlabeled_assertion_clause

Unlabeled_assertion_clause ≜ Boolean_expression | Comment

Tag_mark ≜ Tag ":"

Tag ≜ Identifier

Implication ≜ **implies** Implication_list

Implication_list ≜ {Implication_clause ";" …}*

Implication_clause ≜ Tag_list ":" Tag_list

Tag_list ≜ {Tag "," …}*

Informative Text

8.10.1.1 In simple cases a Precondition consists of only one Individual_precondition, and similarly for postconditions and invariants. The reason for allowing more than one “individual” part is to provide more flexibility, with each part preceded by a Key, which is just a Manifest_string. For example a Postcondition could read

```
ensure ("MODEL")
        concatenated: sequence = old sequence + single (x)
```

ensure ("*IMPLEMENTATION*")
resized: *capacity* <= 3 ∗ (**old** *capacity* // 2)

8.10.1.2 to distinguish between two levels of specification (abstract model and concrete implementation). In an Eiffel environment supporting run-time monitoring of assertions, programmers can then turn monitoring on and off for specific assertion keys.

8.10.1.3 The original declaration of a routine uses, for preconditions and postconditions, the simple forms **require** and **ensure**.The **require else** and **ensure then** forms are compulsory in the case of a routine redeclaration.

8.10.1.4 An Implication part lets programmers specify that certain assertion clauses are consequences of others, inviting proof tools to verify this property, and letting compilers generate less run-time assertion-monitoring code. An example would be a Postcondition that reads

ensure ("*MODEL*")
concatenated: … As above …
ensure ("*IMPLEMENTATION*")
resized: … As above …
ensure
a: … *Some property* …
b: … *Some property* …
c: … *Some property* …
implies
a, concatenated: b, c, resized

8.10.1.5 The Implication in this case states that every one of the properties tagged b, c and resized is a logical consequence of the combination of the properties tagged a and concatenated.

End

8.10.2 Syntax (non-production): Assertion Syntax rule

Code: *SASY*

An Assertion_clause without a Tag_mark may not begin with any of the following:

1 An opening parenthesis "**(**".
2 An opening bracket "**[**".
3 A non-keyword Unary operator that is also Binary.

Informative Text

8.10.2.1 This rule participates in the achievement of the general Semicolon Optionality rule. Without it, after an Assertion_clausestarting for example with the Identifier *a*, and continuing (case 2) with [*x*] it is not immediately obvious whether this is the continuation of the same clause, using *a* [*x*] as the application of a bracket feature to *a*, or a new clause that starts by mentioning the Manifest_tuple [*x*]. From the context, the validity rules will exclude one of these possibilities, but a language processing tool should be able to parse an Eiffel text without recourse to non-syntactic information. A similar issue arises with an opening parenthesis (case 1) and also (case 3) if what follows *a* is *–b*, which could express a subtraction from *a* in the same clause, or start a new clause about the negated value of *b*. The Assertion Syntax rule avoids this.

8.10.2.2 The rule does not significantly restrict expressiveness, since violations are rare and will be flagged clearly in reference to the rule, and it is recommended practice anyway to use a Tag_mark, which removes any ambiguity.

End

8.10.3 Definition: Contract element

Code: *DACE*

A **contract element** is a Precondition, Postcondition or Invariant.

Signed by JV - 2026/10/8

8.10.4 Definition: Precondition, postcondition of a feature; invariant of a class

Code: *DAPI*

The **precondition** and **postcondition** of a feature, or the **invariant** of a class, is the unfolded form of, respectively, their Precondition, Postcondition or Invariant if present and non-empty, and otherwise the minimal assertion.

Informative Text

8.10.4.1 This rule allows the Standard to talk about "the precondition" or "the postcondition" of a feature, or "the invariant" of a class to mean the unfolded forms, in other words the "real" assertion, turned into a boolean integration and integrating the properties of inherited versions. They are to be distinguished from "the Precondition", "the Postcondition" and "the Invariant", referring to syntactic constructs and hence to the assertions as written in the declaration of the feature or class.

8.10.4.2 (To avoid any confusion, however, the Standard text still uses "the unfolded precondition" etc. whenever there could be an ambiguity.)

8.10.4.3 The unfolded forms of assertions (SAUA, page 134, see also) turn an assertion into a single boolean expression taking into account the assertions of inherited versions if any. It achieves this form by:

8.10.4.3.1 • Replacing a missing clause by a simple **True** assertion.

8.10.4.3.2 • Removing the asssertion tags.

8.10.4.3.3 • Putting "and then" between successive parts.

8.10.4.3.4 • Integrating the inherited versions of the assertions (recursively "unfolded" according to the same rules).

8.10.4.4 The precise rules appear below.

8.10.4.5 In all three contexts we consider any absent or empty assertion clause as the assertion **True**, satisfied by every state of the computation. Then we can talk, under any circumstance, of "the precondition of a feature" and "the invariant of a class" even if the clauses do not appear explicitly.

8.10.4.6 With these

End

8.10.5 Definition: Loop invariant, loop variant

Code: *DAIV*

The combined assertion of the Assertion introduced by the Invariant clause of a loop is called its **loop invariant**. The Expression introduced by the Variant clause is called its **loop variant**.

8.10.6 Definition: Contract, subcontract of a feature or a class

Code: *DACO*

Let *pre* and *post* be the precondition and postcondition of a feature *f*. The **contract** of *f* is the Assertion pair [*pre*, *post*].

A feature contract [*pre', post'*] is said to be a **subcontract** of [*pre, post*] if and only if *pre* implies *pre'* and *post'* implies *post*.

The contract of a *class* is the collection of all its feature contracts and the invariant of the class.

8.10.7 Syntax: Check instructions

Code: *SACH*

Check ≜ **check** Assertion [Notes] [Check_then] **end**

Check_then ≜ **then** Compound

Informative Text

8.10.7.1 A Check instruction lets you express that a certain property, captured by an assertion, will be satisfied whenever execution reaches the instruction. Often, there is a **then** clause introducing instructions that should only be executed if the assertion is satisfied; the assertion thus acts as a guard for the instructions.

Signed by JV - 2026/10/8

8.10.7.2 There is no **else** clause: if the property is not satisfied, the software is faulty. In an implementation with advanced program-proving capabilities, the program will be rejected at compile time. If this is not possible, any execution of the Check causes evaluation of the property, and triggers an exception if it is not satisfied. Unlike the monitoring of other kinds of assertions, such evaluation and exceptions are not an option that can be turned on or off but part of the semantics of the instruction.

8.10.7.3 The Notes part is intended for expressing a formal or informal justification of the assumption behind the property being asserted.

8.10.7.4 The semantics of Check instructions appears in MACH, page 139 below. The idea is that either a static prover can determine that the Assertion will always hold, or run-time monitoring as presented below will evaluate it at run time.

End

8.10.8 Syntax: Variants

Code: *SAVE*

Variant ≜ **variant** [Tag_mark] Expression

8.10.9 Validity: Variant Expression rule

Code: *VAVE*

A Variant is valid if and only if its loop variant is of type *INTEGER* or one of its sized variants.

8.10.10 Syntax: “Old” postcondition expressions

Code: *SAOL*

Old ≜ **old** Expression

8.10.11 Validity: Old Expression rule

Code: *VAOX*

An Old expression *oe* of the form **old** *e* is valid if and only if it satisfies the following conditions:

1 It appears in a Postcondition part *post* of a feature.

2 It does not involve **Result**.

3 Replacing *oe* by *e* in *post* yields a valid Postcondition.

Informative Text

8.10.11.1 **Result** is otherwise permitted in postconditions, but condition 2 rules it out since its value is meaningless on entry to the routine. Condition 3 simply states that **old** *e* is valid in a postcondition if *e* itself is. The expression *e* may not, for example, involve any local variables (although it might include **Result** were it not for condition 2), but may refer to features of the class and formal arguments of the routine.

End

8.10.12 Semantics: Old expression semantics, associated variable, exception marker

Code: *MAOE*

The effect of including an Old expression *oe* in a Postcondition of an effective feature *f* is equivalent to replacing the semantics of its Feature_body by the effect of a call to a fictitious routine possessing a local variable *av*, called the **associated variable** of *oe*, and semantics defined by the following succession of steps:

1 Evaluate *oe*.

2 If this evaluation triggers an exception, record this event in an **associated exception marker** for *oe*.

3 Otherwise, assign the value of *oe* to *av*.

4 Proceed with the original semantics.

Signed by JV - 2026/10/8

Informative Text

8.10.12.1 The recourse to a fictitious variable and fictitious operations is in the style of "unfolded forms" used throughout the language description. The reason for these techniques is the somewhat peculiar nature of the Old expression, used at postcondition evaluation time, but pre-computed (if assertion monitoring is on for postconditions) on entry to the feature.

8.10.12.2 The matter of exceptions is particularly delicate and justifies the use of "associated exception markers". If an Oldexpression's evaluation triggers an exception, the time of that exception — feature entry — is not the right moment to start handling the exception, because the postcondition might not need the value. For example, a postcondition clause could read

((*x* /= 0) **and** (**old** *x* /= 0)) **implies** (((1 / *x*) + **old** (1 /*x*)) = *y*)

8.10.12.3 If *x* is 0 on entry, **old** *x* /= 0 will be false on exit and hence the postcondition will hold. But there is no way to know this when evaluating the Oldexpressions, particularly **old** (1 /*x*), on entry. We must evaluate this expression anyway, to be prepared for all possible cases. If *x* is zero, this may cause an arithmetic overflow and trigger an exception. This exception should not be processed immediately; instead it should be remembered — hence the associated exception marker — and triggered only if the evaluation of the *postcondition*, on routine exit, attempts to evaluate the associated variable; hence the following rule.

8.10.12.4 The "associated variable" is defined only for effective features, since a deferred feature has no Feature_body. If an Oldexpression appears in the postcondition of a deferred feature, the rule will apply to effectings in descendants through the "unfolded form" of the postconditions, which includes inherited clauses.

8.10.12.5 Like any variable, the associated variable *av* of an Old expression raises a potential initialization problem; but we need not require its type to be self-initializing since the above rule implies that *ov* appears in a Certified Attachment Pattern that assigns it a value (the value of *oe*) prior to use.

End

8.10.13 Definition: Local unfolded form of an assertion

Code: *DAUL*

The **local unfolded form** of a contract element — a Boolean_expression — is the Equivalent Dot Form of the expression obtained by applying to it, in order, the following transformations:

1. Remove every Tag_mark.
2. Replace any clause of the Comment form by **True**.
3. If the resulting list has more than one Unlabeled_assertion_clause, removing the (optional) semicolon separating them if present, and separate them by **and then**.
4. Replace any Old expression by its associated variable.

Informative Text

8.10.13.1 This unfolded form is "local" because it does not take into account any inherited assertion clauses. Integrating inheritance is the business of the full (non-local) notion of unfolded form of an assertion, introduced in the discussion of redeclaration.

8.10.13.2 The unfolded form enables us to understand an assertion, possibly with many clauses, as a single boolean expression. Tag marks play no role in that expression so we remove them. The use of **and then** to separate the clauses indicates that we may, in a later clause, use an expression that is defined only if an earlier clause holds (has value true).

8.10.13.3 The transformations are applied to the "combined assertion" as defined earlier, which concatenates the various individual assertions of the contract element if more than one.

8.10.13.4 The Equivalent Dot Form of an expression removes all operators and replaces them by explicit call, turning for example *a* + *b* into *a*.*plus* (*b*). This puts the result in a simpler form used by later rules.

Signed by JV - 2026/10/8

8.10.13.5 Step 4 refers to the notion of associated variable of an Old expression; as explained below, the associated variable makes it possible to refer to the expression's value on routine entry.

8.10.13.6 The syntax permits a Comment as Unlabeled_assertion_clause. Such clauses are useful for clarity and documentation but, as reflected by step 2, cannot have any effect on run-time monitoring.

End

8.10.14 Definition: Minimal assertion

Code: *DAMA*

The **minimal assertion** is the Assertion consisting of a single Assertion_clause with no Tag_mark and the Boolean_expression **True**.

Informative Text

8.10.14.1 This notion makes it possible — through the following definition of "local" versions of assertions — to consider that every feature has a precondition and postcondition, and every class an invariant, by adding a trivial assertion **True** if the corresponding clause is absent.

End

8.10.15 Definition: Local assertions

Code: *DALA*

The **local precondition**, **local postcondition** of a feature and **local invariant** of a class are:

1 If the respective assertion (Precondition, Postcondition, Invariant) is present in the declaration of the feature: its local unfolded form.

2 If the respective assertion (Precondition, Postcondition, Invariant) is absent from the declaration of the feature: the minimal assertion.

Informative Text

8.10.15.1 With this definition, every feature has a precondition and postcondition, and every class has an invariant, consisting of just the condition **True** if the respective clause is absent.

8.10.15.2 These forms are "local" in that they do not take inheritance into account. The following rules will construct the "unfolded" versions taking into account the original contracts of inherited versions. For example the clauses of preconditions are "or"-ed with those of the inherited ("precursor") version, and those of invariants are "and"-ed. The precise rules follow.

End

8.10.16 Validity: Assertion Implication rule

Code: *VAAI*

An Implication_clause of the form $a_1, \ldots a_m : b_1, \ldots b_n$ appearing in a contract element c is valid if and only if every a_i and b_j is the Tag of exactly one Assertion_clause in c, and the corresponding Unlabeled_assertion_clause is a Boolean_expression.

Informative Text

8.10.16.1 Outside of their uses in Implication, assertion tags are optional, and when present are chosen freely, for documentation purposes; in particular it is not invalid (although obviously not good style) to use the same tag for different clauses within a single contract element. For implication, however, any such repetition would cause ambiguity and is prohibited.

8.10.16.2 In addition, the clause must be a Boolean_expression, not a Comment.

End

8.10.17 Validity: Precondition Export rule

Code: *VAPE*

A Precondition of a feature r of a class S is valid if and only if every feature f appearing in its unfolded form u satisfies the following two conditions for every class C to which r is available:

Signed by JV - 2026/10/8

1 If *f* appears as the static feature of a call in *u* or any of its subexpressions, *f* is available to *C*.

2 If *u* or any of its subexpressions uses *f* as creation procedure of a Creation_expression, *f* is available for creation to *C*.

Informative Text

8.10.17.1 If (condition 1) *r* were available to a class *B* but its precondition involved a feature *f* not available to *B*, *r* would be imposing to *B* a condition that *B* would not be able to check for itself; this would amount to a secret clause in the contract, preventing the designer of *B* from guaranteeing the correctness of calls.

8.10.17.2 Condition 2 places the same obligation on any feature *f* used in a creation expression **create** *a*.*f* (…) appearing in the precondition (a rare but possible case). The requirement in this case is "available for creation".

8.10.17.3 The rule applies to the *unfolded form* of a precondition, which will be defined as the fully reconstructed assertion, including conditions defined by ancestor versions of a feature in addition to those explicitly mentioned in a redeclared version.

8.10.17.4 The unfolded form (by relying on the "Equivalent Dot Form" of the expressions involved) treats all operators as denoting features; for example an occurrence of *a* > *b* in an assertion yields *a*.*greater* (*b*) in the unfolded form, where *greater* is the name of a feature of alias ">". The Precondition Export rule then requires, if the occurrence is in a Precondition, that this feature be available to any classes to which the enclosing feature is available.

End

8.10.18 Definition: Availability of an assertion clause

Code: *DAAV*

An Assertion_clause *a* of a contract element is **available** to a class *B* if and only if all the features involved in its unfolded form are available to *B*.

Informative Text

8.10.18.1 This notion is necessary to define interface forms of a class adapted to individual clients, such as the incremental contract view ("short form").

End

Informative Text

8.10.18.2 The following clauses discuss the effect of inheritance on contracts.

End

8.10.19 Definition: Unfolded form of an assertion

Code: *SAUA*

The **unfolded form** of an assertion *a* of local unfolded form *ua* in a class *C* is the following Boolean_expression:

1 If *a* is the invariant of *C* and *C* has *n* parents for some $n \geq 1$: up_1 **and** … **and** up_n **and then** *ua*, where up_1, … up_n are (recursively) the unfolded forms of the invariants of these parents, after application of any feature renaming specified by *C*'s corresponding Parent clauses.

2 If *a* is the precondition of a redeclared feature *f*: the combined precondition for *a*.

3 If *a* is the postcondition of a redeclared feature *f*: the combined postcondition for *a*.

4 In all other cases: *ua*.

Signed by JV - 2026/10/8

Informative Text

8.10.19.1 The unfolded form of an assertion is the form that will define its semantics. It takes into account not only the assertion as written in the class, but also any applicable property inherited from the parent. The "local unfolded form" is the expression deduced from the assertion in the class itself; for an invariant we "and then" it with the "and" of the parents, and for preconditions and postconditions we use "combined forms", defined next, to integrate the effect of **require else** and **ensure then** clauses, to ensure that things will still work as expected in the context of polymorphism and dynamic binding.

8.10.19.2 The earlier definitions enable us to talk about the "precondition of" and "postcondition "of" a feature and the "invariant of" even in the absence of explicit clauses, by using **True** in such cases. This explains in particular why case 1 can mention "the invariants of" the parents of *C.*

End

8.10.20 Definition: Assertion extensions

Code: *SDAE*

For a feature *f* of a class *C*:

- If *C* redeclares *f* with a non-empty Precondition (starting with **require else**), the **precondition extension** of *f* in *C* is the corresponding Assertion.
- If *C* redeclares *f* with a non-empty Postcondition (starting with **ensure then**), the **postcondition extension** of *f* in *C* is the corresponding Assertion.

In all other cases, the precondition extension of *f* in *C* is **False** and the postcondition extension of *f* in *C* is **True**.

Informative Text

8.10.20.1 These are the forms that routines can use to override inherited specifications while remaining compatible with the original contracts for polymorphism and dynamic binding. **require else** makes it possible to weaken a precondition, **ensure then** to strengthen a postcondition, under the exact interpretation explained next.

End

8.10.21 Definition: Covariance-aware form of an assertion extension

Code: *DACA*

The **covariance-aware form** of an inherited assertion *a* is:

1 If the enclosing routine has one or more arguments x_1, … x_n redefined covariantly to types U_1, … U_n: the assertion
($\{x_1: U_1\}\ y_1$ **and** … **and** $\{x_n: U_n\}\ y_n$) **and then** *a'*
where y_1, … y_n are fresh identifiers and *a'* is the result of substituting y_i for each corresponding x_i in *a*.

2 Otherwise: *a*.

Informative Text

8.10.21.1 A covariant redefinition may make some of the new clauses inapplicable to actual arguments of the old type (leading to "catcalls"). The covariance-aware form avoids this by ignoring the clauses that are not applicable. The rule on covariant redefinition avoid any bad consequences.

End

8.10.22 Definition: Combined precondition, combined postcondition

Code: *DACP*

Consider a feature *f* redeclared in a class *C*. Let f_1, … f_n ($n \geq 1$) be its versions in parents, pre_1, … pre_n the covariance-aware forms of (recursively) the unfolded preconditions of these versions, and $post_1$, … $post_n$ the covariance-aware forms of (recursively) their unfolded postconditions.

Signed by JV - 2026/10/8

Let *pre* be the precondition extension of *f* if defined and not empty, otherwise **False**.

Let *post* be the postcondition extension of *f* if defined and not empty, otherwise **True**.

The **combined precondition** of *f* is the Assertion

(pre_1 **or**… **or** pre_n) **or else** *pre*

The **combined postcondition** of *f* is the Assertion

(**old** pre_1 **implies** $post_1$)

and … **and**

(**old** pre_n **implies** $post_n$)

and then *post*

Informative Text

8.10.22.1 The informal rule is “perform an *or* of the preconditions and an *and* of the postconditions”. This indeed the definition for “combined precondition”. For “combined postconditions” the informal rule is sufficient in most cases, but occasionally it may be too strong because it requires the old postconditions even in cases that do *not* satisfy the old preconditions, and hence only need the new postcondition. The combined postcondition as defined reflects this property.

End

8.10.23 Definition: Precondition-free

Code: *DAPF*

A feature is **precondition-free** if and only if its combined precondition has value **True**.

Informative Text

8.10.23.1 We require the root creation procedure of a system to be precondition-free (see VSRP, page 81).

End

Informative Text

8.10.23.2 The following clauses discuss the notion of correctness associated with contracts. Tools known as program provers, still experimental at the time of writing the Standard, may be able to establish correctness properties (on the sole basis of the class texts, that is to say, without execution).

End

8.10.24 Definition: Hoare triple notation (total correctness)

Code: *SAHT*

In definitions of correctness notions for Eiffel constructs, the notation {*P*} *A* {*Q*} (a mathematical convention, not a part of Eiffel) expresses that any execution of the Instruction or Compound *A* started in a state of the computation satisfying the assertion *P* will terminate in a state satisfying the assertion *Q*.

Informative Text

8.10.24.1 Hoare triples are a fundamental tool of work on program verification, making it possible to express the correctness of a program element *A*, relative to a precondition *P* and a postcondition *Q*, in the form {*P*} *A* {*Q*}. Program verification in this framework, known as *axiomatic semantics*, program verification consists of establishing such properties, either statically (contract-based proofs) or by monitoring the assertions dynamically (contract-based testing).

8.10.24.2 [Note: this notion is not used in the rest of the clause -- but should! FIX]

End

Signed by JV - 2026/10/8

8.10.25 Definition: Implication correctness

Code: *DAIC*

A contract element is **implication-correct** if in any Implication_clause that it contains, of the form $a_1, \ldots a_m : b_1, \ldots b_n$, the logical conjunction of the unfolded forms of the assertion clauses associated with the a_is implies the logical disjunction of those associated with the b_is.

Informative Text

8.10.25.1 As noted, an Implication of the form

implies

a, *concatenated*: *b*, *c*, *resized*

means that each one of the assertion clauses of tags b, c and resized is a logical consequence of the combination of the properties tagged a and concatenated.

8.10.25.2 This definition is the first of a set of definitions and semantic rules, continuing next with various forms of correctness associated with contracts and contract elements.

End

8.10.26 Definition: Check-correct

Code: *DACC*

An effective routine *r* is **check-correct** if, for every Check instruction *c* in *r*, any execution of *c* (as part of an execution of *r*) satisfies its Assertion.

8.10.27 Semantics: Class consistency

Code: *MACO*

A class *C* is **consistent** if and only if it satisfies the following conditions:

1 For every creation procedure *p* of *C*:

$\{pre_p\}\ do_p\ \{INV_C$ **and then** $post_p\}$

1 For every feature *f* of *C* exported generally or selectively:

$\{INV_C$ **and then** $pre_f\}\ do_f\ \{INV_C$ **and then** $post_f\}$

where INV_C is the invariant of *C* and, for any feature *f*, pre_f is the unfolded form of the precondition of *f*, $post_f$ the unfolded form of its postcondition, and do_f its body.

Informative Text

8.10.27.1 Class consistency is one of the most important aspects of the *correctness* of a class: adequation of routine implementations to the specification. The other aspects of correctness, studied below, involve Check instructions, Loop instructions and Rescue clauses.

End

8.10.28 Definition: Loop-correct

Code: *DALC*

8.10.28.1 loop-correct

A routine is **loop-correct** if every loop it contains, with loop invariant *INV*, loop variant *VAR*, Initialization *INIT*, Exit_condition *EXIT* and body (Compound part of the Loop_body) *BODY*, satisfies the following conditions:

1 $\{$**true**$\}\ INIT\ \{INV\}$

2 $\{$**true**$\}\ INIT\ \{VAR \geq 0\}C$

3 $\{INV$ **and then not** $EXIT\}\ BODY\ \{INV\}$

4 $\{INV$ **and then not** $EXIT$ **and then** $(VAR = v)\}\ BODY\ \{0 \leq VAR < v\}$

Signed by JV - 2026/10/8

Informative Text

8.10.28.2 Conditions 1 and 2 express that the initialization yields a state in which the invariant is satisfied and the variant is non-negative. Conditions 3 and 4 express that the body, when executed in a state where the invariant is satisfied but not the exit condition, will preserve the invariant and decrease the variant, while keeping it non-negative. (*v* is an auxiliary variable used to refer to the value of *VAR* before *BODY*'s execution.)

End

8.10.29 Definition: Correctness (class)

Code: *DACS*

A class is **correct** if and only if it is consistent and every routine of the class is check-correct, loop-correct and exception-correct.

Informative Text

8.10.29.1 Program provers may be able to determine that a class is correct.

End

Informative Text

The following clauses describe the run-time effect of contracts, relevant in the absence of a program prover.

End

Informative Text

8.10.30 Semantics: Associated variable semantics

Code: *MAVA*

As part of the evaluation of a postcondition clause, the evaluation of the associated variable of an Old expression:

1 Triggers an exception of type *OLD_EXCEPTION* if an associated exception marker has been recorded.

2 Otherwise, yields the value to which the variable has been set.

Informative Text

8.10.30.1 The notion of associated variable of an Oldexpression captures the semantic purpose of such expressions: to refer, in a routine postcondition — which describes properties that the routine must satisfy on *exit* of every call — to the value that the expression had on *entry* to the call.

End

8.10.31 Semantics: Evaluation of an assertion

Code: *MAEA*

To **evaluate** an assertion consists of computing the value of its unfolded form.

Informative Text

8.10.31.1 This defines the value of an assertion in terms of the value of a boolean expression, as given by the discussion of expressions.

End

Signed by JV - 2026/10/8

8.10.32 Semantics: Assertion monitoring

Code: *MAAM*

The execution of an Eiffel system may evaluate, or **monitor**, specific kinds of assertion, and loop variants, at specific stages:

1 Precondition of a routine *r*: on starting a call to *r*, after argument evaluation and prior to executing any of the instructions in *r*'s body.

2 Postcondition of a routine *r*: on successful (not interrupted by an exception) completion of a call to *r*, after executing any applicable instructions of *r*.

3 Invariant of a class *C*: on termination of a creation instruction whose creation type is based on *C*, and on both start and termination of a targeted and qualified Call to a routine of *C*.

4 Invariant appearing in a Loop: after execution of the Initialization, and after every execution (if any) of the Loop_body.

5 Assertion in a Check instruction: on any execution of that instruction.

6 Variant appearing in a Loop: as with the loop invariant.

8.10.33 Semantics: Assertion violation

Code: *MAAV*

An **assertion violation** is the occurrence at run time, as a result of assertion monitoring, of any of the following:

- An assertion (in the strict sense of the term) evaluating to false.
- A loop variant found to be negative.
- A loop variant found, after the execution of a Loop_body, to be no less than in its previous evaluation.

Informative Text

8.10.33.1 To simplify the discussion these cases are all called "*assertion* violations" even though a loop variant (an integer expression) is not an assertion.

End

8.10.34 Semantics: Assertion semantics

Code: *MAAS*

In the absence of assertion violations, assertions have no effect on system execution other than through their evaluation as a result of assertion monitoring.

An assertion violation causes an exception of type *ASSERTION_VIOLATION* or one of its descendants.

8.10.35 Semantics: Check instruction semantics

Code: *MACH*

In the absence of a formal proof that a Check is check-correct, its execution consists of, in order:

1 Evaluating its Assertion.

2 If the result of the evaluation is false, triggering an exception of type *CHECK_VIOLATION*. The following step does not apply in this case.

3 Executing the Compound of the Check_then, if present.

8.10.36 Semantics: Assertion monitoring levels

Code: *MAAL*

An Eiffel implementation must provide facilities to enable or disable assertion monitoring according to some combinations of the following criteria:

- Statically (at compile time) or dynamically (at run time).
- Through control information specified within the Eiffel text or through outside elements such as a user interface or configuration files.
- For specific kinds as listed in the definition of assertion monitoring: routine preconditions, routine postconditions, class invariants, loop invariants, Check instructions, loop variants.

Signed by JV - 2026/10/8

- For specific Key values (as associated with assertions).
- For specific classes, specific clusters, or the entire system.

The following combinations must be supported:

1 Statically disable all monitoring for the entire system.
2 Statically enable precondition monitoring for an entire system.
3 Statically enable precondition monitoring for specified classes.
4 Statically enable all assertion monitoring for an entire system.

Signed by JV - 2026/10/8

PART F: Inheritance

Signed by JV - 2026/10/8

Signed by JV - 2026/10/8

8.11 The inheritance relation

Informative Text

8.11.0.1 Inheritance is one of the most powerful facilities available to software developers. It addresses two key issues of software development, corresponding to the two roles of classes:

8.11.0.1.1 • As a **module extension** mechanism, inheritance makes it possible to define new classes from existing ones by adding or adapting features.

8.11.0.1.2 • As a **type refinement** mechanism, inheritance supports the definition of new types as specializations of existing ones, and plays a key role in defining the type system.

End

8.11.1 Syntax: Inheritance parts

Code: *SHIP*

Inheritance ≜ Inherit_clause+

Inherit_clause ≜ **inherit** [Non_conformance] Parent_list

Non_conformance ≜ "**{**" *NONE* "**}**"

Parent_list ≜ {Parent ";" …}+

Parent =Class_type [Feature_adaptation]

Feature_adaptation ≜ [Undefine]
[Redefine]
[Rename]
[New_exports]
[Select]
end

Informative Text

8.11.1.1 As with all other uses of semicolons, the semicolon separating successive Parent parts is optional. The style guidelines suggest omitting it between clauses that appear (as they should) on successive lines.

End

8.11.2 Syntax (non-production): Feature adaptation

Code: *SFAD*

A Feature_adaptation part must include at least one of the optional components.

Informative Text

8.11.2.1 This rule removes a potential syntax ambiguity by implying that the **end** in **class** *B* **inherit** *A* **end** closes the class; otherwise it could be understood as closing just the Parent part.

End

8.11.3 Definition: Parent part for a type, for a class

Code: *DHPP*

If a Parent part *p* of an Inheritance part lists a Class_type *T*, *p* is said to be a Parent part **for** *T*, and also for the base class of *T*.

Signed by JV - 2026/10/8

Informative Text

8.11.3.1 So in **inherit** *TREE* [*T*] there is a Parent part for the type *TREE* [*T*] and for its base class *TREE*. For convenience this definition, like those for “parent” and “heir” below, applies to both types and classes.

End

Informative Text

8.11.3.2 An important property of the inheritance structure is that every class inherits, directly or indirectly, from a class called *ANY*, of which a version is provided in the Kernel Library as required by the next rule. The semantics of the language depends on the presence of such a class, whether the library version or one that a programmer has provided as a replacement.

End

8.11.4 Validity: Class *ANY* rule

Code: *VHCA*

Every system must include a non-generic class called *ANY*.

Informative Text

8.11.4.1 The key property of *ANY* is that it is not only an ancestor of all classes and hence types, but that all types **conform** to it, according to the following principle, which is not a separate validity rule (although for reference it has a code of its own) but a consequence of the definitions and rules below.

End

8.11.5 Validity: Universal Conformance principle

Code: *VHUC*

Every type conforms to *ANY*.

Informative Text

8.11.5.1 To achieve the Universal Conformance principle, the semantics of the language guarantees that a class that doesn’t list any explicit Parent is considered to have *ANY* as its parent. This is captured by the following notion: Unfolded Inheritance Part. The above definition of “parent”, and through it the definition of “ancestor”, refer to the Unfolded Inheritance Part of a class rather than its actual Inheritance part.

End

8.11.6 Definition: Unfolded inheritance part of a class

Code: *DHUI*

Any class *C* has an **unfolded inheritance part** defined as follows:

1 If *C* has an Inheritance part with at least one conforming parent: that part.

2 If *C* has an Inheritance part with only non-conforming parent: that part extended with one Parent, of the form *ANY*.

3 Otherwise: an Inheritance part of the form **inherit** *ANY*.

8.11.7 Definition: Multiple, single inheritance

Code: *DHSM*

A class has **multiple inheritance** if it has an unfolded inheritance part with two or more Parent parts. It has **single inheritance** otherwise.

Signed by JV - 2026/10/8

Informative Text

8.11.7.1 What counts for this definition is the number not of parent classes but of Parent parts. If two such parts refer to the same parent class, this is still a case of multiple inheritance, known as **repeated inheritance** and studied later on its own. If there is no Parent part, the class (as will be seen below) has a de facto parent anyway, the Kernel Library class *ANY*.

8.11.7.2 The definition refers to the "Unfolded" inheritance part which is usually just the Inheritance part but may take into account implicit inheritance from *ANY*, as detailed in the corresponding definition below.

8.11.7.3 Multiple inheritance is a frequent occurrence in Eiffel development; most of the effective classes in the widely used EiffelBase library of data structures and algorithms, for example, have two or more parents. The widespread view that multiple inheritance is "bad" or "dangerous" is not justified; most of the time, it results from experience with imperfect multiple inheritance mechanisms, or improper uses of inheritance. Well-applied multiple and repeated inheritance is a powerful way to combine abstractions, and a key technique of object-oriented software development.

End

8.11.8 Definition: Inherit, heir, parent

Code: *DHIH*

A class *C* **inherits** from a type or class *B* if and only if *C*'s unfolded inheritance part contains a Parent part for *B*.

B is then a **parent** of *C* ("**parent type**" or "**parent class**" if there is any ambiguity), and *C* an **heir** (or "heir class") of *B*. Any type of base class *C* is also an heir of *B* ("heir type" in case of ambiguity).

8.11.9 Definition: Conforming, non-conforming parent

Code: *DHCO*

A parent *B* in an Inheritance part is **non-conforming** if and only if every Parent part for *B* in the clause appears in an Inherit_clause with a Non_conformance marker. It is **conforming** otherwise.

[XXXX UPDATE TO INCLUDE CASE OF INHERITING FROM AN EXPANDED CLASS]

8.11.10 Definition: Ancestor types of a type, of a class

Code: *DHAT*

The **ancestor types** of a *type CT* of base class *C* include:

1 *CT* itself.

2 (Recursively) The result of applying *CT*'s generic substitution to the ancestor types of every parent type for *C*.

The ancestor types of a *class* are the ancestor types of its current type.

Informative Text

8.11.10.1 The basic definition covers ancestor types of a *type*; the second part of the definition extends this notion to classes.

8.11.10.2 Case 1 indicates that a type is its own ancestor.

8.11.10.3 Case 2, the recursive case, applies the notion of *generic substitution* introduced in the discussion of genericity. The idea that if we consider the type *C* [*INTEGER*], with the class declaration **class** *C* [*G*] **inherit** *D* [*G*] …, the type to include in the ancestors of *C* [*INTEGER*] as a result of this Inheritance part is not *D* [*G*], which makes no sense outside of the text of *C*, but *D* [*INTEGER*], the result of applying to *D* [*G*] the substitution *G* → *INTEGER*; this is the substitution that yields the type *C* [*INTEGER*] from the class *C* [*G*] and is known as the generic substitution of that type.

End

8.11.11 Definition: Ancestor, descendant

Code: *DHAD*

Class *A* is an **ancestor** of class *B* if and only if *A* is the base class of an ancestor type of *B*.

Signed by JV - 2026/10/8

Class *B* is a **descendant** of class *A* if and only if *A* is an ancestor of *B*.

Informative Text

8.11.11.1 Any class, then, is both one of its own descendants and one of its own ancestors. *Proper* descendants and ancestors exclude these cases.

End

8.11.12 Definition: Proper ancestor, proper descendant

Code: *DHPA*

The **proper ancestors** of a class *C* are its ancestors other than *C* itself. The **proper descendants** of a class *B* are its descendants other than *B* itself.

8.11.13 Validity: Parent rule

Code: *VHPR*

The unfolded inheritance part of a class *D* is valid if and only if it satisfies the following conditions:

1 In every Parent part for a class *B*, *B* is not a descendant of *D*.

2 No conforming parent is a frozen class.

3 If two or more Parent parts are for classes which have a common ancestor *A*, *D* meets the conditions of the Repeated Inheritance Consistency constraint for *A*.

4 At least one of the Parent parts is conforming.

5 No two ancestor types of *D* are different generic derivations of the same class.

6 Every Parent is generic-creation-ready.

7 Any anchored type involved in any actual generic parameter of any Parent is of the form *like* **Current**.

Informative Text

8.11.13.1 Condition 1 ensures that there are no cycles in the inheritance relation.

8.11.13.2 The purpose of declaring a class as **frozen** (case 2) is to prohibit subtyping. We still permit the *non-conforming* form of inheritance, which permits reuse but not subtyping.

8.11.13.3 Condition 3 corresponds to the case of repeated inheritance; the Repeated Inheritance Consistency constraint will guarantee that there is no ambiguity on features that *D* inherits repeatedly from *A*.

8.11.13.4 Condition 4 ensures a central property of the type system: the Universal Conformance principle, stating that all types conform to *ANY*. Without this condition, it would be possible for all Parent parts of a class to be non-conforming and hence to cause violation of the principle. Note that in the Unfolded Inheritance Part there is always at least one Parent part, since the absence of an Inheritance part is a shorthand for **inherit** *ANY*, ensuring that condition 4 holds.

8.11.13.5 The remaining conditions govern Parent parts that are generically derived, as in **inherit** *A* [*T*]:

8.11.13.5.1 • Condition 5 avoids various cases of ambiguity which could arise if we allowed a class *C* to inherit from both *A* [*T*] and *A* [*U*] for different *T* and *U*. For example, if *C* redefines a feature *f* from *A*, the notation **Precursor** {*A*} in the redefinition could refer to either of the parents' generic derivations.

8.11.13.5.2 • Condition 6 also concerns the case of a generically derived Parent *A* [*T*]; requiring it to be "generic-creation-ready" guarantees that creation operations on *D* or its descendants will function properly if they need to create objects of type *T*.

8.11.13.5.3 • Condition 7 says that if *T* (or, indirectly, any type appearing as part of *T*, for example if *T* is itself generally derived) is "anchored", that is to say starting with like, it can only be like Current. This

Signed by JV - 2026/10/8

form can have its usefulness, as in inherit *LIST [like Current]*; but any more general anchored type, of the form *like anchor* for some entity *anchor*, might cause ambiguity if used as a Parent.

End

8.11.14 Syntax: Rename clauses

Code: *SHRC*

Rename ≜ **rename** Rename_list

Rename_list ≜ {Rename_pair "," …}$^{+}$

Rename_pair ≜ Feature_name **as** Extended_feature_name

Informative Text

8.11.14.1 The first component of a Rename_pair is just a Feature_name, the identifier for the feature; the second part is a full Extended_feature_name, which may include an **alias** clause. Indeed:

8.11.14.1.1 • To identify the feature you are renaming, its Feature_name suffices.

8.11.14.1.2 • At the same time you are renaming the feature, you may give it a new operator or bracket alias or parenthesis alias, or remove the alias if it had one.

8.11.14.2 Forms of feature adaptation other than renaming, in particular effecting and redefinition, do not affect the Alias, if any, associated with a Feature_name.

End

8.11.15 Validity: Rename Clause rule

Code: *VHRC*

A Rename_pair of the form *old_name* **as** *new_name*, appearing in the Rename subclause of the Parent part for *B* in a class *C,* is valid if and only if it satisfies the following conditions:

1 *old_name* is the final name of a feature *f* of *B*.

2 *old_name* does not appear as the first element of any other Rename_pair in the same Rename subclause.

3 *new_name* satisfies the Feature Name rule for *C*.

4 The Alias of *new_name*, if present, is alias-valid for the version of *f* in *C*.

Informative Text

8.11.15.1 In condition 4, the “alias-valid” condition captures the signature properties allowing a query to have an operator or bracket aliases. It was enforced when we wanted to give a feature an alias in the first place and, naturally, we encounter it again when we give it an alias through renaming.

End

8.11.16 Definition: Final name, extended final name, final name set, renaming

Code: *DHFI*

Every feature *f* of a class *C* has an **extended final name** in *C* (an Extended_feature_name) and a **final name** (a Feature_name), defined as follows:

1 The final name is the identifier of the extended final name.

2 If *f* is immediate in *C*, its extended final name is the Extended_feature_name under which *C* declares it.

3 If *f* is inherited feature, *f* is obtained from a feature_of a parent *B* of *C*. Let *extended_parent_name* be (recursively) the extended final name of that feature in *B*, and *parent_name* its final name of *f* in *B*. Then the extended final name of *f* in *C* is:

4 If the Parent part for *B* in *C* contains a Rename_pair of the form **rename** *parent_name* **as** *new_name*: *new_name*.

5 Otherwise: *extended_parent_name*.

Signed by JV - 2026/10/8

The final names of all the features of a class constitute the **final name set** of the class.

In case 4, the feature is said to be **renamed** in *C* and the applicable Rename_pair is a **renaming**.

Informative Text

8.11.16.1 Since an inherited feature may be obtained from two or more parent features, case 3 only makes sense if they are all inherited under the same name. This will follow from the final definition of "inherited feature" in the discussion of repeated inheritance.

8.11.16.2 The extended final name is an Extended_feature_name, possibly including an Alias part; the final name is its identifier only, a Feature_name, without the alias. The recursive definition defines the two together.

End

8.11.17 Semantics: Renaming principle

Code: *MHRP*

Renaming does not affect the semantics of an inherited feature.

Informative Text

8.11.17.1 The "positive" semantics of renaming (as opposed to the negative observation captured by this principle) follows from the definition of *final name* and *extended final name* of a feature below.

8.11.17.2 Like other principles, the Renaming principle is not a separate rule but a consequence of validity and semantic rules, particularly those on feature adaptation and repeated inheritance.

End

8.11.18 Definition: Inherited name

Code: *DHIN*

The **inherited name** of a feature obtained from a feature *f* of a parent *B* is the final name of *f* in *B*.

Informative Text

8.11.18.1 In the rest of the language description, references to the "name" of a feature, if not further qualified, always denote the final name.

End

8.11.19 Definition: Declaration for a feature

Code: *DHDF*

A Feature_declaration in a class *C*, listing a Feature_name *fn*, is a **declaration for** a feature *f* if and only if *fn* is the final name of *f* in *C*.

Informative Text

8.11.19.1 Although it may seem almost tautological, we need this definition so that we can talk about a declaration "for" a feature *f* whether *f* is immediate — in which case *fn* is just the name given in its declaration — or inherited, with possible renaming. This will be useful in particular when we look at a *redeclaration*, which overrides a version inherited from a parent.

End

8.12 Feature adaptation

Informative Text

8.12.0.1 A key attraction of the inheritance mechanism is that it lets you tune Inherited features to the context of the new class. This is known as feature adaptation. The present clause covers the principal mechanisms, leaving to a later one some important complements related to repeated inheritance.

End

8.12.1 Definition: Redeclare, redeclaration

Code: *SDRD*

A class **redeclares** an inherited feature if it redefines, effects or undefines it.

A declaration for a feature *f* is a **redeclaration** of *f* if it is either a redefinition or an effecting of *f*.

Informative Text

8.12.1.1 This definition relies on two others, appearing below, for the two cases: *redefinition* and *effecting*.

8.12.1.2 Be sure to distinguish *redeclaration* from *redefinition*, the first of these cases. Redeclaration is the more general notion, redefinition one of its two cases; the other is *effecting*, which provides an implementation for a feature that was deferred in the parent. In both cases, a redeclaration does not introduce a new feature, but simply overrides the parent's version of an inherited feature.

End

8.12.2 Definition: Inherited as effective, inherited as deferred

Code: *DDIN*

An inherited feature is **inherited as effective** if it has at least one precursor that is an effective feature, and the corresponding Parent part does not undefine it.

Otherwise the feature is **inherited as deferred**.

8.12.3 Definition: Effect, effecting

Code: *DDEF*

A class **effects** an inherited feature *f* if and only if it inherits *f* as deferred and contains a redeclaration of *f* which yields an effective feature. That redeclaration is called an **effecting** of *f*.

Informative Text

8.12.3.1 Effecting a feature (making it *effective*, hence the terminology) consists of providing an implementation for a feature that was inherited as deferred. No particular clause (such as **redefine**) will appear in the Inheritance part: the new implementation will without ado subsume the deferred form inherited from the parent.

End

8.12.4 Definition: Redefine, redefinition

Code: *DDRD*

A class **redefines** an inherited feature *f* if and only if it contains a declaration for *f* that is not an effecting of *f*.

Such a declaration is then known as a **redefinition** of *f*

Informative Text

8.12.4.1 Redefining a feature consists of providing a new implementation, specification or both. The applicable Parent clause or clauses must specify **redefine** *f* (with *f*'s original name if the new class renames *f*.)

8.12.4.2 Redefinition must keep the inherited status, deferred or effective, of *f*:

8.12.4.2.1 • It cannot turn a deferred feature into an effective one, as this would be an effecting.

Signed by JV - 2026/10/8

8.12.4.2.2 • It may not turn an effective feature into a deferred one, as there is another mechanism specifically for this purpose, *un*definition. The Redeclaration rule enforces this property.

8.12.4.3 As defined earlier, the two cases, effecting and redefinition, are cases of redeclaration (a term that also covers undefinition).

End

8.12.5 Definition: Name clash

Code: *DDNC*

A class has a **name clash** if it inherits two or more features from different parents under the same final name.

Informative Text

8.12.5.1 Since final names include the identifier part only, aliases if any play no role in this definition.

8.12.5.2 Name clashes would usually render the class invalid. Only three cases may — as detailed by the validity rules — make a name clash permissible:

- At most one of the clashing features is effective.
- The class redefines all the clashing features into a common version.
- The clashing features are really the same feature, inherited without redeclaration from a common ancestor.

End

8.12.6 Syntax: Precursor call, precursor signature

Code: *SDPR*

Precursor_call ≜ Precursor_type [Actuals]

Precursor_type ≜ **Precursor** [Parent_qualification]

Parent_qualification ≜ "**{**" Class_name "**}**"

Informative Text

8.12.6.1 A Precursor_call makes it possible, in the definition of a routine, to call the original version as applicable to the parent.

8.12.6.2 The Precursor_type part is also useful to specify that in such a redefinition, or more generally a redeclaration, an argument or result retains its type from the original.

End

8.12.7 Validity: Precursor Signature rule and definition: specified parent

Code: *VDPS*

A Precursor_type appearing in a class *C* is valid if and only if it satisfies the following conditions:

1 It appears in the redeclaration of a feature.

2 If there is a Parent_qualification, its Class_name is the name of a parent class *P* of *C*, called the **specified parent**.

Informative Text

8.12.7.1 In condition 1, a redeclaration is (as defined in (SFPA, page 101) either the effecting (from a deferred version to an effective one) or the redefinition (from an effective version to another) of a feature. In the case of a Precursor_call the redeclaration can only be a redefinition (per VDPC, page 151 below).

8.12.7.2 Condition 2 states that if there is a Parent_qualification, as in **Precursor** {*P*}, then *P* must be the name of one of the parents of the current class. The associated validity rule (clause VDPC, page 151) makes this qualified form compulsory in case of potential ambiguity, but even in the absence of ambiguity you may use it to state the parent explicitly if you think this improves readability.

End

8.12.8 Definition: Relative unfolded form of a Precursor_call

Code: *DDUR*

Consider a Precursor_call *p* appearing in the redefinition of a routine *r* inherited from a parent class *B*. Its **unfolded form relative to** *B* is an Unqualified_call of the form *s* if *p* has no Actuals part, or *s* (*args*) if *p* has actual arguments *args*, where *s* is a fictitious feature name added, with a **frozen** mark, as synonym for *r* in *B*.

Informative Text

8.12.8.1 The unfolded form is necessary for specifying the semantics of a Precursor_call: the intent of calling **recursor** in the redefinition of a routine *r* is to call the original *r*, which is the same as calling *s* if we had defined *s* as a synonym of *r* in the parent class. Recall that you obtain a "synonym" if instead of a declaration *r feature_declaration* we write *r, s feature_declaration* (DFSY, page 103).

8.12.8.2 To define that semantics we do not use an arbitrary parent *B* but the relevant one, the "applicable parent" as defined next, giving us an "absolute" rather than relative unfolded form. But for that applicable

End

8.12.9 Validity: Precursor Call rule and definitions: precursof feature, applicable call

Code: *VDPC*

A Precursor_call is valid if and only if it satisfies the following conditions:

1. Among the features of *C*'s parents, limited to features of the specified parent if any, exactly one is a feature redeclared by *C* into *f*. This feature is called the **precursor feature** of *f*, and the class to which it belongs is called the **applicable parent** of the Precursor_call.
2. Its unfolded form relative to the applicable parent is, as an Unqualified_call, valid. It is called the **unfolded form** of the Precursor_call.
3. If it is used as an Instruction , the precursor feature is a command.
4. If it is used as an Expression , the precursor feature is a query.
5. The precursor feature is effective.

Informative Text

8.12.9.1 Condition 1 specifies when an explicit parent qualification is required: whenever an ambiguity could arise because the redefinition applies to more than one effective parent version. The phrasing takes care of all the cases in which this could happen, for example as a result of a join.

8.12.9.2 Condition 2 expresses that in this case we understand the Precursor_call specimen as a call to a frozen version of the original routine; we must make sure that such a call would be valid, more precisely "argument-valid", the requirement applicable to such an Unqualified_call.

8.12.9.3 Prior to this rule the "unfolded form" of a Precursor_call was only defined "relative to" a parent class. We are of course interested only in the form relative to the applicable parent. Condition 2 makes it possible from now on to talk of the "unfolded form" — no longer "relative" — of a Precursor_call.

8.12.9.4 A Precursor_call will be used as either an Instruction or an Expression, in the same way as a call to (respectively) a procedure or a function; indeed Precursor_call appears as one of the syntax variants for both of these constructs. So in addition to being valid on its own, it must be valid in the appropriate role. Conditions 3 and 4 take care of this.

Signed by JV - 2026/10/8

8.12.9.5 When we use **Precursor** to call a previous version of a feature, that version must exist; in other words, the original may not have been deferred. Condition 5 specifies this.

End

8.12.10 Semantics: Precursor call semantics

Code: *MDPC*

The effect of a Precursor_call used as in Instruction or Expression is the effect of its unfolded form.

8.12.11 Syntax: Redefinition

Code: *SDRS*

Redefine ≙ **redefine** Feature_list

8.12.12 Validity: Redefine Subclause rule

Code: *VDRS*

A Redefine subclause appearing in a Parent part for a class *B* in a class *C* is valid if and only if every Feature_name *fname* that it lists (in its Feature_list) satisfies the following conditions:

1 *fname* is the final name of a feature *f* of *B*.

2 *f* was not frozen in *B*, and was not a constant attribute.

3 *fname* appears only once in the Feature_list.

4 The Features part of *C* contains one Feature_declaration that is a redeclaration but not an effecting of *f*.

5 If that redeclaration specifies a deferred feature, *C* inherits *f* as deferred.

8.12.13 Semantics: Redefinition semantics

Code: *MDRD*

The effect in a class *C* of redefining a feature *f* in a Parent part for *A* is that the version of *f* in *C* is, rather than its version in *A*, the feature described by the applicable declaration in *C*.

Informative Text

8.12.13.1 This new version will serve for any use of the feature in the class, its clients, its proper descendants (barring further redeclarations), and even ancestors and their clients under dynamic binding.

End

8.12.14 Syntax: Undefine subclauses

Code: *SDUS*

Undefine ≙ **undefine** Feature_list

8.12.15 Definition: Undefine, undefinition

Code: *DDUN*

A class **undefines** an inherited feature *f* if and only if it lists it in an Undefine subclause.

Such a clause is then known as an **undefinition** of *f*.

Informative Text

8.12.15.1 Undefining a feature consists of removing the implementation of a feature that would be otherwise inherited as effective, making it instead inherited as deferred. Undefinition is one of the three cases of redeclaration; the other two are redefinition and effecting. Undefinition can be viewed as the reverse of effecting (since effecting takes a class that was deferred in the parent and makes it effective in the heir).

End

8.12.16 Validity: Undefine Subclause rule

Code: *VDUS*

An Undefine subclause appearing in a Parent part for a class *B* in a class *C* is valid if and only if every Feature_name *fname* that it lists (in its Feature_list) satisfies the following conditions:

1 *fname* is the final name of a feature *f* of *B*.

Signed by JV - 2026/10/8

2 *f* was not frozen in *B*, and was not an attribute.

3 *f* was effective in *B*.

4 *fname* appears only once in the Feature_list.

5 Any redeclaration of *f* in *C* specifies a deferred feature.

8.12.17 Semantics: Undefinition semantics

Code: *MDUD*

The effect in a class *C* of undefining a feature *f* in a Parent part for *A* is to cause *C* to inherit from *A*, rather than the version of *f* in *A*, a deferred form of that version.

8.12.18 Definition: Effective, deferred feature

Code: *DDDE*

A feature *f* of a class *C* is an **effective feature** of *C* if and only if it satisfies either of the following conditions:

1 *C* contains a declaration for *f* whose Feature_body is not of the Deferred form.

2 *f* is an inherited feature, coming from a parent *B* of *C* where it is (recursively) effective, and *C* does not undefine it.

f is **deferred** if and only if it is not effective.

Informative Text

8.12.18.1 As a result of this definition, a feature is deferred in *C* not only if it is introduced or redefined in *C* as deferred, but also if its precursor was deferred and *C* does not redeclare it effectively. In the latter case, the feature is "inherited as deferred".

8.12.18.2 The definition captures the semantics of deferred features and of their effecting. In case 1 it's clear that the feature is effective, since *C* itself declares it as either an attribute of a non-deferred routine. In case 2 the feature is inherited; it was already effective in the parent, and *C* doesn't change that status.

End

8.12.19 Principle: Deferred Class property

Code: *PDDE*

A class that has at least one deferred feature must have a Class_header starting with the keyword **deferred**. The class is then said to be **deferred**.

Informative Text

8.12.19.1 This validity property is not a separate constraint but follows from other rules. It also serves as a definition of "**deferred class**".

End

8.12.20 Principle: Effective Class property

Code: *PDEF*

A class whose features, if any, are all effective, is effective unless its Class_header starts with the keyword **deferred**.

Informative Text

8.12.20.1 This validity property is not a separate constraint but follows from other rules. It also serves as a definition of "**effective class**".

Signed by JV - 2026/10/8

8.12.20.2 It is not an error to declare a class **deferred** if it has no deferred features; the effect is simply that clients are not able to create direct instances. It is indeed sometimes useful to introduce a class that cannot be directly instantiated; for example the designer may intend the class to be used only through inheritance. The technique to achieve this is simply to state the abstract nature of the class by declaring it **deferred** even if all its features are effective.

End

8.12.21 Definition: Origin, seed

Code: *DDOR*

Every feature *f* of a class *C* has one or more features known as its **seeds** and one or more classes known as its **origins**, as follows:

1 If *f* is immediate in *C*: *f* itself as seed; *C* as a origin.

2 If *f* is inherited: (recursively) all the seeds and origins of its precursors.

Informative Text

8.12.21.1 The origin, a class, is “where the feature comes from”, and the seed is the version of the feature from that origin. In the vast majority of cases this is all there is to know. With repeated inheritance and “join”, a feature may result from the merging of two or more features, and hence may have more than one seed and more than one origin. That’s what case 2 is about.

End

8.12.22 Validity: Redeclaration rule

Code: *VDRD*

Let *C* be a class and *g* a feature of *C*. It is valid for *g* to be a redeclaration of a feature *f* inherited from a parent *B* of *C* if and only if the following conditions are satisfied.

1 No effective feature of *C* other than *f* and *g* has the same final name.

2 The signature of *g* conforms to the signature of *f*.

3 The Precondition of *g*, if any, begins with **require else** (not just **require**), and its Postcondition, if any, begins with **ensure then** (not just **ensure**).

4 If the redeclaration is a redefinition (rather than an effecting) the Redefine subclause of the Parent part for *B* lists in its Feature_list the final name of *f* in *B*.

5 If *f* is inherited as effective, then *g* is also effective.

6 If *f* is an attribute, *g* is an attribute, *f* and *g* are both variable, and their types are either both expanded or both non-expanded.

7 *f* and *g* have either both no alias or the same alias.

8 If both features are queries with associated assigner commands *fp* and *gp*, then *gp* is the version of *fp* in *C*.

Informative Text

8.12.22.1 Condition 1 prohibits name clashes between effective features. For *g* to be a redeclaration of *f*, both features must have the same final name; but no other feature of the class may share that name. This is the fundamental rule of **no overloading**.

8.12.22.2 No invalidity results, however, if *f* is deferred. Then if *g* is also deferred, the redeclaration is simply a redefinition of a deferred feature by another (to change the signature or specification). If *g* is effective, the redeclaration is an effecting of *f*. If *g* plays this role for more than one inherited *f*, it both joins and effects these features: this is the case in which *C* kills several deferred birds with one effective stone.

8.12.22.3 Condition 2 is the fundamental type compatibility rule: signature conformance. In the case of a join, *g* may be the redeclaration of more than one *f*; then *g*’s signature must conform to all of the precursors’ signatures.

8.12.22.4 Signature conformance permits *covariant* redefinition of both query results and routine arguments, but for arguments you must make the new type detachable — **detachable** *U* rather than just *U* — to prevent "catcalls".

8.12.22.5 Condition 3 requires adapting the assertions of a redeclared feature, as governed by rules given earlier.

8.12.22.6 Condition 4 requires listing *f* in the appropriate Redefine subclause, but only for a redefinition, not for an effecting. (We have a redefinition only if *g* and the inherited form of *f* are both deferred or both effective.) If two or more features inherited as deferred are joined and then redefined together, **every one of them** must appear in the Redefine subclause for the corresponding parent.

8.12.22.7 Condition 5 bars the use of redeclaration for turning an effective feature into a deferred one. This is because a specific mechanism is available for that purpose: undefinition. It is possible to apply both undefinition and redefinition to the same feature to make it deferred and at the same time change its signature.

8.12.22.8 Condition 6 prohibits redeclaring a constant attribute, or redeclaring a variable attribute into a function or constant attribute. It also precludes redeclaring a (variable) attribute of an expanded type into one of reference type or conversely. You may, however, redeclare a function into an attribute — variable or constant.

8.12.22.9 Condition 7 requires the features, if they have aliases, to have the same ones. If you want to introduce an alias for an inherited feature, change an inherited alias, or remove it, redeclaration is not the appropriate technique: you must rename the feature. Of course you can still redeclare it as well.

8.12.22.10 Condition 8 applies to assigner commands. It is valid for a redeclaration to include an assigner command if the precursor did not include one, or conversely; but if both versions of the query have assigner commands, they must, for obvious reasons of consistency, be the same procedure in *C*.

End

8.12.23 Definition: Precursor (joined features)

Code: *DDPJ*

A **precursor** of an inherited feature is a version of the feature in the parent from which it is inherited.

8.12.24 Definition: Direct transposition

Code: *DDTR*

The **direct transposition** to a class *B* of a component *s* appearing in a parent class *A* of *B* is the component obtained from *s* by replacing every expression by its Equivalent Dot Form, then:

1 Replacing the arguments of any Call by (recursively) their direct transposition to *B*.

2 If *s* is part of the declaration of a feature *g* replicated in *B* along a certain repeated inheritance path, replacing the feature name of any unqualified call by the name of the feature as resulting from any renaming along that path.

3 In every unqualified call of feature *f* whose feature name *fn* appears in a Rename_pair of the form *fn* **as** *gn* in a Parent part for *A*, such that case 2 does not apply, replacing *fn* by the identifier of *gn*.

4 In every Object_call with an explicit Target *t*, replacing *t* by (recursively) its class transposition *t'* to *B* and the dynamic feature of the call by (recursively) its transposition to the type of *t'* in *B*.

5 In every Non_object_call of target type *T*, replacing *T* by (recursively) its class transposition *T'* to *B* and the dynamic feature of the call by (recursively) its transposition to *T'*.

6 For every entity *e*, other than an attribute, such that *s* includes a declaration for *e*, replacing every occurrence of *e* by a fresh identifier not used in *B*.

7 Replacing every occurrence of a formal generic parameter of *A* by the generic substitution of *B*'s parent type of base class *A*.

Signed by JV - 2026/10/8

8.12.25 Definition: Transposition to a class or type

Code: *DDTC*

~~The **transposition** to a class *C* of a component *s* appearing in a ancestor *A* of *C* is the specimen obtained from *s* by replacing every expression by its Equivalent Dot Form, then:~~

1. ~~Replacing the arguments of any Call by (recursively) their transposition to *C*.~~
2. ~~If *s* is part of the declaration of a feature *g* replicated in *C* along a certain repeated inheritance path, replacing any Feature_name used as the feature name of an unqualified call or as anchor of an anchored type by the name resulting from any renaming of the feature along that path.~~
3. ~~Replacing any Feature_name used as the feature name of an unqualified call of an unqualified call or as anchor of an anchored type, if case 2 does not apply, by the result of any renaming along applicable inheritance paths.~~
4. ~~In every Object_call with an explicit Target *t*, replacing *t* by (recursively) its transposition *t'* to *C* and the dynamic feature of the call by (recursively) its transposition to the type of *t'* in *C*.~~
5. ~~In every Non_object_call of target type *T*, replacing *T* by (recursively) its transposition *T'* to *C* and the dynamic feature of the call by (recursively) its transposition to *T'*.~~
6. ~~For every entity *e*, other than an attribute, such that *s* includes a declaration for *e*, replacing every occurrence of *e* by a fresh identifier not used in *C*.~~
7. ~~If an ancestor *B* of *C* has a parent type *P* of base class *A*, replacing every occurrence of any generic parameter *G* of *A* by (recursively) the transposition to *C* of the application to *G* of *P*'s generic substitution.~~

[THE ABOVE IS A DUPLICATE, CHECK FOR INCONSISTENCIES]

The transposition to a type *T* of a component *s* appearing in a ancestor of the base class *C* of *T* is the result of applying the generic substitution of *T* to the class transposition of *s* to *C*.

The **class transposition** to a class *C* of a component *s* appearing in an ancestor *A* of *C* is:

1. If *A* and *C* are the same class: *s*.
2. If *A* is a parent of an ancestor *B* of *C*: (recursively) the transposition to *C* of the direct transposition of *s* to *B*.

The **transposition** to a type *T* of a specimen *s* appearing in a ancestor of the base class *C* of *T* is the result of applying the generic substitution of *T* to the class transposition of *s* to *C*.

Informative Text

8.12.25.1 The first part (cases 3 and 7) defines transposition to an heir (direct descendant). Cases 8 and 9 generalize this to any descendant. Recall that a descendant *C* of *A* is either *A* itself (case 8) or, recursively, a descendant of an heir *B* of *A* (case 9).

8.12.25.2 Case takes care of feature renaming. Because

End

8.12.26 Definition: Unfolded redeclaration

Code: *DDUD*

Consider a feature *f* of a class *A*. The **unfolded redeclaration** of *f* in an heir *C* of *A* is a Feature_declaration defined as follows:

1. If *C* redeclares *f*, the declaration of *f* in *C*.
2. Otherwise, a Feature_declaration for a feature with the same extended name, the same signature as *f* and the same Assigner_mark if any, both transposed to *C*, and a Feature_definition consisting solely of:

- If *f* is deferred, a Feature_body of the Deferred kind.

Signed by JV - 2026/10/8

- If *f* is an effective routine, a **do** clause whose Compound reads just **Precursor** (if *f* is a procedure) or **Result** := **Precursor** (if *f* is a function), followed by the parenthesized list of formal arguments if any.
- If *f* is an attribute, an **attribute** clause whose Compound reads just **Result** := **Precursor**.

8.12.27 Validity: Join rule

Code: *VDJR*

It is valid for a class *C* to inherit two different features under the same final name under and only under the following conditions:

1 If both are inherited as effective, *C* redefines both into a common version.

2 If both are inherited as deferred, the unfolded redeclaration in *C* of each of them is a valid redeclaration of the other.

3 Otherwise, the unfolded redeclaration in *C* of the one inherited as effective is a valid redeclaration of the one inherited as deferred.

Informative Text

8.12.27.1 THE FOLLOWING INFORMATIVE TEXT NEEDS UPDATING. The Join rule indicates that joined features must have exactly the same signature — argument and result types.

8.12.27.2 What matters is the signature after possible redefinition or effecting. So in practice you may join precursor features with different signatures: it suffices to redeclare them using a feature which (as required by point 2 of the Redeclaration rule) must have a signature conforming to all of the precursors' signatures.

8.12.27.3 If the redeclaration describes an effective feature, this is the case of both joining and effecting a set of Inherited features. If the redeclaration describes a feature that is still deferred, it is a redefinition, used to adapt the signature and possibly the specification. In this case, point 4 of the Redeclaration rule requires every one of the precursors to appear in the Redefine subclause for the corresponding parent.

8.12.27.4 Condition 1 mentions "redeclaration or effecting". These two cases are not exclusive: an effecting — turning a feature *f*, inherited as deferred from a parent of *C*, into an effective one — can result from a new declaration of *f* in *C*, but also from a "join" of *f* with an effective feature inherited under the same name from another parent.

8.12.27.5 In any case, nothing requires the precursors' signatures to conform to each other, as long as the signature of the version in *C* conforms to all of them. This means you may write a class inheriting two deferred features of the form

f (p: P): T …

f (t: Q): U …

and redeclare them with

f (*x:* **detachable** *R): V* …

provided *R* conforms to both *P* and *Q* and *V* to both *T* and *U*. No conformance is required between the types appearing in the precursors' signatures (*P* and *Q*, *T* and *U*).

8.12.27.6 The assumption that the features are "different" is important: they could in fact be the same feature, appearing in two parents of *C* that have inherited it from a common ancestor, without any intervening redeclaration. This would be a valid case of repeated inheritance; here the rule that determines validity is the Repeated Inheritance Consistency constraint. The semantic specification (sharing under the Repeated Inheritance rule) indicates that *C* will have just one version of the feature.

8.12.27.7 Conditions 1 and 2 of the Join rule are consistency requirements on aliases and on assigner commands. The condition on aliases is consistent with condition 7 of the Redeclaration rule, which requires a redeclaration to keep the alias if any; it was noted in the comment to that rule that redeclaration is not the appropriate way to add, change or remove an alias (you should use

renaming for that purpose); neither is join. The condition on assigner commands ensures that any Assigner_call has the expected effect, even under dynamic binding on a target declared of a parent type.

End

8.12.28 Semantics: Join Semantics rule

Code: *MDJO*

Joining in a class *C* two or more Inherited features with the same final name under the terms of the Join rule yields a single feature of *C* defined as follows:

1 If at least one of these features is effective: its unfolded redeclaration in *C*.

2 Otherwise: the unfolded redeclaration in *C* of any of them.

Informative Text

8.12.28.1 ****** TO BE REDONE *****The rule covers three cases:

8.12.28.1.1 • An explicit redeclaration, which serves as a redeclaration of all the joined precursors, and gives them a new signature (which must conform to all their signatures per the Join rule), body (since it serves as “unfolded redeclaration” in point ****) and assertions (point ****).

8.12.28.1.2 • No redeclaration, with precursors all deferred, all having the same signature; they are then merged into a single deferred feature.

8.12.28.1.3 • No redeclaration, with one effective feature and the others deferred, all with the same signature; the effective feature then serves as effecting of the others.

8.12.28.2 In the absence of a redeclaration, point****** states that the new feature has no specific precondition and postcondition. It will still, however, have a **combined precondition** and a **combined postcondition** obtained from the precursors’ assertions. In the case of a redeclaration, the combined precondition and postcondition also include the assertions, if any, of the redeclared version.

8.12.28.3 Point ****** leaves the concatenation order unspecified.

8.12.28.4 In point ******, there can be at most one effective precursor because of the Join rule.

8.12.28.5 In point ****** (corresponding to a rare case) language processing tools should produce an obsolescence message for the class performing the join, but the resulting feature is not itself obsolete.

End

8.13 Repeated inheritance

Informative Text

8.13.0.1 Inheritance may be **multiple**: a class may have any number of parents. A more restrictive solution would limit the benefits of inheritance, so central to object-oriented software engineering.

8.13.0.2 Because of multiple inheritance, it is possible for a class to be a descendant of another in more than one way. This case is known as **repeated** inheritance; it raises interesting issues and yields useful techniques, which the following discussion reviews in detail.

End

8.13.1 Definition: Repeated inheritance, ancestor, descendant

Code: *DMRI*

Repeated inheritance occurs whenever (as a result of multiple inheritance) two or more of the ancestors of a class *D* have a common parent *A*.

D is then called a **repeated descendant** of *A*, and *A* a **repeated ancestor** of *D*.

8.13.2 Semantics: Repeated Inheritance rule

Code: *SMRI*

Let *D* be a class and B_1, … B_n ($n \geq 2$) be parents of *D* based on classes having a common ancestor *A*. Let f_1, … f_n be features of these respective parents, all having as one of their seeds the same feature *f* of *A*. Then:

1 Any subset of these features inherited by *D* under the same final name in *D* yields a single feature of *D*.

2 Any two of these features inherited under a different name yield two features of *D*.

Informative Text

8.13.2.1 This is the basic rule allowing us to make sense and take advantage of inheritance, based on the programmer-controlled naming policy: inheriting two features under the same name yields a single feature, inheriting them under two different names yield two features.

End

8.13.3 Definition: Sharing, replication

Code: *DMSH*

A repeatedly inherited feature is **shared** if case 1 of the Repeated Inheritance rule applies, and **replicated** if case 2 applies.

8.13.4 Validity: Call Sharing rule

Code: *VMCS*

It is valid for a feature *f* repeatedly inherited by a class *D* from an ancestor *A*, such that *f* is shared under repeated inheritance and not redeclared, to involve a feature *g* of *A* other than as the feature of a qualified call if and only if *g* is, along the corresponding inheritance paths, also shared.

Informative Text

8.13.4.1 If *g* were duplicated, there would be no way to know which version *f* should call, or evaluate for the assignment. The “selected” version, discussed below, is not necessarily the appropriate one.

End

8.13.5 Semantics: Replication Semantics rule

Code: *DMRP*

Let *f* and *g* be two features both repeatedly inherited by a class *A* and both replicated under the Repeated Inheritance rule, with two respective sets of different names: *f1* and *f2*, *g1* and *g2*.

If the version of *f* in *D* is the original version from *A* and either contains an unqualified call to *g* or (if *f* is an attribute) is the target of an assignment whose source involves *g*, the *f1* version will use *g1* for that call or assignment, and the *f2* version will use *g2*.

Signed by JV - 2026/10/8

Informative Text

8.13.5.1 This rule (which, unlike other semantic rules, clarifies a special case rather than giving the general semantics of a construct) tells us how to interpret calls and assignments if two separate replications have proceeded along distinct inheritance paths.

End

8.13.6 Syntax: Select subclause

Code: *SMSE*

Select ≜ **select** Feature_list

Informative Text

8.13.6.1 The Select subclause serves to resolve any ambiguities that could arise, in dynamic binding on polymorphic targets declared statically of a repeated ancestor's type, when a feature from that type has two different versions in the repeated descendant.

End

8.13.7 Validity: Select Subclause rule

Code: *VMSS*

A Select subclause appearing in the Parent part for a class *B* in a class *D* is valid if and only if, for every Feature_name *fname* in its Feature_list, *fname* is the final name in *D* of a feature that has two or more potential versions in D, and *fname* appears only once in the Feature_list.

Informative Text

8.13.7.1 This rule restricts the use of Select to cases in which it is meaningful: two or more "potential versions", a term which also has its own precise definition. We will encounter next, in the Repeated Inheritance Consistency constraint, the converse requirement that if there is such a conflict a Select *must* be provided.

End

8.13.8 Definition: Version

Code: *DMVE*

A feature *g* from a class *D* is a **version** of a feature *f* from an ancestor of *D* if *f* and *g* have a seed in common.

8.13.9 Definition: Multiple versions

Code: *DMMV*

A class *D* has *n* **versions** ($n \geq 2$) of a feature *f* of an ancestor *A* if and only if *n* of its features, all with different final names in *D*, are all versions of *f*.

8.13.10 Validity: Repeated Inheritance Consistency constraint

Code: *VMRC*

It is valid for a class *D* to have two or more versions of a feature *f* of a proper ancestor *A* if and only if it satisfies one of the following conditions:

1. There is at most one conformance path from *D* to *A*.
2. There are two or more conformance paths, and the Parent clause for exactly one of them in *D* has a Select clause listing the name of the version of *f* from the corresponding parent.

8.13.11 Definition: Dynamic binding version

Code: *DMDV*

For any feature *f* of a type *T* and any type *U* conforming to *T*, the **dynamic binding version** of *f* in *U* is the feature *g* of *U* defined as follows:

1. If *f* has only one version in *U*, then *g* is that feature.
2. If *f* has two or more versions in *U*, then the Repeated Inheritance Consistency constraint ensures that either exactly one conformance path exists from *U* to *T*, in which case *g* is the

Signed by JV - 2026/10/8

version of *f* in *U* obtained along that path, or that a Select subclause name a version of *f*, in which case *g* is that version.

8.13.12 Definition: Inherited features

Code: *DMIF*

Let *D* be a class. Let *precursors* be the list obtained by concatenating the lists of features of every parent of *D*; this list may contain duplicates in the case of repeated inheritance. The list *inherited* of **inherited features** of *D* is obtained from *precursors* as follows:

1 In the list *precursors*, for any set of two or more elements representing features that are repeatedly inherited in *D* under the same name, so that the Repeated Inheritance rule yields sharing, keep only one of these elements. The Repeated Inheritance Consistency constraint (sharing case) indicates that these elements must all represent the same feature, so that it does not matter which one is kept.

2 For every feature *f* in the resulting list, if *D* undefines *f*, replace *f* by a deferred feature with the same signature, specification and header comment.

3 In the resulting list, for any set of deferred features with the same final name in *D*, keep only one of these features, with assertions and header comment joined as per the Join Semantics rule. (Keep the signature, which the Join rule requires to be the same for all the features involved after possible redeclaration.)

4 In the resulting list, remove any deferred feature such that the list contains an effective feature with the same final name. (This is the case in which a feature *f*, inherited as effective, effects one or more deferred features: of the whole group, only *f* remains.)

5 All the features in the resulting list have different names; they are the inherited features of *D* in their parent forms. From this list, produce a new one by replacing any feature that *D* redeclares (through redefinition or effecting) with the result of the redeclaration, and retaining any other feature as it is.

6 The result is the list *inherited* of inherited features of *D*.

Informative Text

8.13.12.1 The presentation of features had a first definition (DFII, page 100) of what it means for a feature to be "inherited" in a class: instead of being introduced in the class itself ("immediate"), it is obtained from at least one Parent. The above definition complements it by specifying precisely, for a given class, what each one of its inherited features actually is in the class:

8.13.12.1.1 • Its version in the class may be kept unchanged from the applicable parent (or parents, in the case of repeated inheritance). But it may also have been subject to adaptations through the Inheritance clause, in one of the following three ways.

8.13.12.1.2 • It may have been effected (transformed from deferred to effective by being given an implementation).

8.13.12.1.4 • It may have been redefined (provided a new implementation or signature or both).

8.13.12.1.5 • It may have beenmerged with others (through effecting or repeated inheritance).

End

8.13.13 Semantics: Join-Sharing Reconciliation rule

Code: *MMJS*

If a class inherits two or more features satisfying both the conditions of sharing under the Repeated Inheritance rule and those of the Join rule, the applicable semantics is the Repeated Inheritance rule.

8.13.14 Definition: Precursor

Code: *DMPR*

A **precursor** of an inherited feature of final name *fname* is any parent feature — appearing in the list *precursors* obtained through case 1 of the definition of "Inherited features" — that the feature mergings resulting from the subsequent cases reduce into a feature of name *fname*.

Signed by JV - 2026/10/8

8.13.15 Validity: Feature Name rule

Code: *VMFN*

It is valid for a feature *f* of a class *C* to have a certain final name if and only if it satisfies the following conditions:

1 No other feature of *C* has that same feature name.

2 If *f* is shared under repeated inheritance, its precursors all have either no Alias or the same alias.

Informative Text

8.13.15.1 Condition 1 follows from other rules: the Feature Declaration rule (VFDE, page 119), the Redeclaration rule and the rules on repeated inheritance. It is convenient to state it as a separate condition, as it can help produce clear error messages in some cases of violation.

8.13.15.2 Two feature names are “the same” if the lower-case version of their identifiers is the same.

8.13.15.3 The important notion in this condition is “**other feature**”, resulting from the above definition of “inherited features”. When do we consider *g* to be a feature “other” than *f*? This is the case whenever *g* has been declared or redeclared distinctly from *f*, unless the definition of inherited features causes the features to be merged into just one feature of *C*. Such merging may only happen as a result of sharing features under repeated inheritance, or of joining deferred features.

8.13.15.4 Also, remember that if *C* redeclares an inherited feature (possibly resulting from the joining of two or more), this does not introduce any new (“other”) feature. This was explicitly stated by the definition of “introducing” a feature.

8.13.15.5 Condition 2 complements these requirements by ensuring that sharing doesn’t inadvertently give a feature more than one alias.

8.13.15.6 The Feature Name rule crowns the discussion of inheritance and feature adaptation by unequivocally implementing the No Overloading Principle: no two features of a class may have the same name. The only permissible case is when the name clash is apparent only, but in reality the features involved are all the same feature under different guises, resulting from a join or from sharing under repeated inheritance.

End

8.13.16 Validity: Name Clash rule

Code: *VMNC*

The following properties govern the names of the features of a class *C*:

1 It is invalid for *C* to introduce two different features with the same name.

2 If *C* introduces a feature with the same name as a feature it inherits as effective, it must rename the inherited feature.

3 If *C* inherits two features as effective from different parents and they have the same name, the class must also (except under sharing for repeated inheritance) remove the name clash through renaming.

Informative Text

8.13.16.1 This is not a new constraint but a set of properties that follow from the Feature Name rule and other rules. Instead of Eiffel’s customary “This is valid if and only if …” style, more directly useful to the programmer since it doesn’t just tell us how to mess things up but also how to produce guaranteeably *valid* software, the Name Clash rule is of the more discouraging form “You may not validly write …”. It does, however, highlight frequently applicable consequences of the naming policy, and compilers may take advantage of it to report naming errors.

End

Signed by JV - 2026/10/8

PART G: The type system

Signed by JV - 2026/10/8

Signed by JV - 2026/10/8

8.14 Types

Informative Text

8.14.0.1 Types describe the form and properties of objects that can be created during the execution of a system. The type system lies at the heart of the object-oriented approach; the use of types to declare all entities leads to more clear software texts and permits compilers to detect many potential errors and inconsistencies before they can cause damage.

End

8.14.1 Syntax: Types

Code: *STTY*

Type ≜ Class_or_tuple_type | Formal_generic_name | Anchored

Class_or_tuple_type ≜ Class_type | Tuple_type

Class_type ≜ [Attachment_mark] Class_name [Actual_generics]

Attachment_mark ≜ **detachable** | **attached**

Anchored ≜ [Attachment_mark] **like** Anchor

Anchor ≜ Feature_name | **Current**

Informative Text

8.14.1.1 The most common and versatile kind is Class_type, covering types described by a class name, followed by actual generic parameters if the class is generic. The class name gives the type's base class. If the base class is expanded, the Class_type itself is an expanded type; if the base class is non-expanded, the Class_type is a reference type.

8.14.1.2 A **detachable** Attachment_mark indicates that the type is **detachable**: its values may be void — not attached to an object. The **attached** mark indicates the reverse: the type is **attached**: its values will always denote an object; language rules, in particular constraints on attachment, guarantee this. No Attachment_mark means the same as **attached**, to ensure that a type, by default, will be attached.

End

8.14.2 Semantics: Direct instances and values of a type

Code: *MTDI*

The **direct instances** of a type *T* are the run-time objects resulting from: representing a manifest constant, Manifest_tuple, Manifest_type, Agent or Address expression of type *T*; applying a creation operation to a target of type *T*; (recursively) cloning an existing direct instance of *T*.

The **values** of a type *T* are the possible run-time values of an entity or expression of type *T*.

Informative Text

8.14.2.1 The word "direct" in "direct instance" indicates that direct instances are objects created exactly (through a creation instruction or expression) from the specification of a type, with one field for every instance of its base class. The more general notion of "instance", defined next, takes inheritance into account: it includes not only direct instances of the type but also those of its conforming types (types based on the descendants of its base class).

8.14.2.2 "Values of a type" as defined here is a specific case of the more general notion of "value" defined in DEVA, page 243,

End

8.14.3 Semantics: Instance of a type

Code: *MTIN*

The **instances** of a type *TX* are the direct instances of any type conforming to *TX*.

Informative Text

8.14.3.1 Since every type conforms to itself, this is equivalent to stating that the instances of *TX* are the direct instances of *TX* and, recursively, the instances of any other type conforming to *TX*.

End

8.14.4 Semantics: Instance principle

Code: *MTIP*

Any value of a type *T* is:

- If *T* is reference, either a reference to an instance of *T* or (unless *T* is attached) a void reference.
- If *T* is expanded, an instance of *T*.

8.14.5 Definition: Instance, direct instance of a class

Code: *DTCI*

An instance of a class *C* is an instance of any type *T* based on *C*.

A direct instance of *C* is a direct instance of any type *T* based on *C*.

Informative Text

8.14.5.1 For non-generic classes the difference between *C* and *T* is irrelevant, but for a generic class you must remember that by itself the class does not fully determine the shape of its direct instances: you need a type, which requires providing a set of actual generic parameters.

End

8.14.6 Principle: Base Class principle

Code: *PTBP*

Any type *T* proceeds, directly or indirectly, from a Class_or_tuple_type called its **base type**, and an underlying class called its **base class**.

The base class of a type is also the base class of its base type.

Informative Text

8.14.6.1 This validity property is not a separate constraint but follows from other rules. For example:

8.14.6.1.1 - A Class_type is its own base type.

8.14.6.1.2 - An anchored type **like** *anchor* with *anchor* having base type *U* also has *U* as its base type.

8.14.6.1.3 - For a formal generic parameter *G* in **class** *C* [*G* –> *T*] … the base type is the constraining type *T*. If the constraint part "–> *T*" is absent, then the base type is *ANY* i (in other words there is an implicit "–> *ANY*").

8.14.6.2 The base class is the class providing the features applicable to instances of the type. If *T* is a Class_type the connection to a class is direct: *T* is either the name of a non-generic class, such as *PARAGRAPH*, or the name of a generic class followed by Actual_generics, such as *LIST* [*WINDOW*]. In both cases the base class of *T* is the class whose name is used to obtain *T*, with any Actual_generics removed: *PARAGRAPH* and *LIST* in the examples. For a Tuple_type, the base class is a fictitious class *TUPLE*, providing the features applicable to all tuples.

8.14.6.3 For types not immediately obtained from a class we obtain the base class by going through base type: for example *T* is an Anchored type of the form **like** *anchor*, and *anchor* is of type *LIST* [*WINDOW*], then the base class of that type, *LIST*, is also the base class of *T*.

End

8.14.7 Principle: Base Type principle

Code: *PTBR*

The base type of any type is a Class_or_tuple_type, with no Attachment_mark.

Signed by JV - 2026/10/8

The base class of any type other than a Class_or_tuple_type is (recursively) the base class of its base type.

The direct instances of a type are those of its base type.

Informative Text

8.14.7.1 This validity property is not a separate constraint but follows from other rules.

8.14.7.2 Why are the notions of base type and base class important? Many of a type's key properties (such as the features applicable to the corresponding entities) are defined by its base class. Furthermore, class texts almost never directly refer to classes: they refer to *types* based on these classes.

End

8.14.8 Validity: Class Type rule

Code: *VTCT*

A Class_type is valid if and only if it satisfies the following two conditions:

1 Its Class_name is the name of a class in the surrounding universe.

2 If it has a **detachable** Attachment_mark, that class is not expanded.

Informative Text

8.14.8.1 The class given by condition 1 will be the type's base class. Regarding condition 2, an expanded type is always attached, so an Attachment_mark would not make sense in that case.

End

8.14.9 Semantics: Type Semantics rule

Code: *MTTY*

To define the semantics of a type *T* it suffices to specify:

1 Whether *T* is expanded or reference.

2 Whether *T*, if reference, is attached or detachable.

3 What is *T*'s base type.

4 If *T* is a Class_or_tuple_type, what are its base class and its type parameters if any.

8.14.10 Definition: Base class and base type of an expression or entity

Code: *DTBE*

Any expression or entity *e* has a **base type** and a **base class**, defined as the base type and base class of the type of *e*.

Informative Text

8.14.10.1 This concept, like the next one, is defined for convenience: it enables language rules and discussions to refer to "the base class of *e*" rather than "the base class of the type of *e*", and similarly for the base type.

End

8.14.11 Definition: Features of a type

Code: *DTFT*

The **features of** a type are the features of its base class.

Informative Text

8.14.11.1 This definition is again a matter of convenience: it enables rules and discussions to use "the features of *T*" rather than "the features of the base class of *T*".

End

8.14.12 Semantics: Non-generic class type semantics

Code: *MTNG*

A non-generic class *C* used as a type (of the Class_type category) has the same expansion status as *C* (i.e. it is expanded if *C* is an expanded class, reference otherwise). It is its own base type (after removal of any Attachment_mark) and base class.

8.14.13 Definition: Expanded type, reference type

Code: *DTEX*

A type *T* is **expanded** if and only if it is not a Formal_generic_name and the base class of its deanchored form is an expanded class.

T is a **reference type** if it is neither a Formal_generic_name nor expanded.

Informative Text

8.14.13.1 This definition characterizes every type as either reference or expanded, except for the case of a Formal_generic_name, which stands for any type to be used as actual generic parameter in a generic derivation: some derivations might use a reference type, others an expanded type.

8.14.13.2 Tuple types are, as a consequence of the definition, reference types.

End

Informative Text

8.14.13.3 The next two definitions generalize to types two notions commonly used for classes: the current class and the features of a class. This generalization only matters for types that are not classes (but are based on classes), particularly generically derived types, formal generic parameters and anchored types.

End

8.14.14 Definition: Current type

Code: *DTCT*

Within a class text, the **current type** is the type obtained from the current class by providing as actual generic parameters, if required, the class's own formal generic parameters.

Informative Text

8.14.14.1 Clearly, the base class of the current type is always the current class.

End

8.14.15 Definition: Features of a type

Code: *DTFE*

The features of a type are the features of its base class.

Informative Text

8.14.15.1 The basic notion is "feature of a class". The features of a *type* are the features applicable to its instances (which are also instances of its base class).

8.14.15.2 The generalization is relevant, in particular, in the context of genericity (8.15):

8.14.15.2.1 • For a non-generic class such as *INTEGER* which is also a type, the features of the type are the same as the features of the class.

8.14.15.2.2 • Consider a generic class such as *LIST* [*G*], with "features of the class" such as *extend (x: G)*, which adds an element to the end of a list of elements of type *G*. They are also the features of any type such as *LIST* [*INTEGER*] obtained ("generically derived") from the class. Note that in this case *extend* applied to a target of type *LIST* [*INTEGER*] expects an argument of type *INTEGER*.

End

Signed by JV - 2026/10/8

8.14.16 Definition: Basic type

Code: *DTBT*

The basic types are *BOOLEAN*, *CHARACTER* and its sized variants, *INTEGER* and its sized variants, *REAL* and its sized variants and *POINTER*.

Informative Text

8.14.16.1 Like most other types, the basic types are defined by classes, found in the Kernel Library. In other words they are not predefined, “magic” types, but fit in the normal class-based type system of Eiffel.

8.14.16.2 Compilers typically know about them, so that they can generate code that performs arithmetic and relational operations as fast as in lower-level languages where basic types are built-in. This is only for efficient implementation: semantically, the basic types are just like other class types.

End

8.14.17 Definition: Anchor, anchored type, anchored entity

Code: *DTAN*

The **anchor** of an anchored type **like** *anchor* is the entity *anchor*. A declaration of an entity with such a type is an **anchored declaration**, and the entity itself is an **anchored entity**.

Informative Text

8.14.17.1 The anchor must be either an entity, or **Current**. If an entity, *anchor* must be the final name of a feature of the enclosing class.

8.14.17.2 The syntax permits *x* to be declared of type **like** *anchor* if *anchor* is itself anchored, of type **like** *other_anchor*. Although most developments do not need such anchor chains, they turn out to be occasionally useful for advanced applications. But then of course we must make sure that an anchor chain is meaningful, by excluding cycles such as *a* declared as **like** *b*, *b* as **like** *c*, and *c* as **like** *a*. The following definition helps.

End

8.14.18 Definition: Anchor set; cyclic anchor

Code: *DTAS*

The **anchor set** of a type *T* is the set of entities containing, for every anchored type **like** *anchor* involved in *T*:

- *anchor*.
- (Recursively) the anchor set of the type of *anchor*.

An entity *a* of type *T* is a **cyclic anchor** if the anchor set of *T* includes *a* itself.

Informative Text

8.14.18.1 The anchor set of *LIST* [**like** *a, HASH_TABLE* [**like** *b, STRING*]] is, according to this definition, the set {*a, b*}.

8.14.18.2 Because of genericity, the cycles that make an anchor “cyclic” might occur not directly through the anchors but through the types they involve, as with *a* of type *LIST* [**like** *b*] where *b* is of type **like** *a*. Here we say that a type “involves” all the types appearing in its definition, as captured by the following definition.

End

8.14.19 Definition: Types and classes involved in a type

Code: *DTIN*

The types **involved** in a type *T* are the following:

1 *T* itself.

2 If *T* is of the form *a T'* where *a* is an Attachment_mark: (recursively) the types involved in *T'*.

3 If *T* is a generically derived Class_type or a Tuple_type: all the types (recursively) involved in any of its actual generic parameters.

The *classes* involved in *T* are the base classes of the types involved in *T*.

Informative Text

8.14.19.1 *A* [*B, C, LIST* [*ARRAY* [*D*]]] involves itself as well as *B*, *C*, *D*, *ARRAY* [*D*] and *LIST* [*ARRAY* [*D*]]. The notion of *cyclic anchor* captures this notion in full generality; the basic rule, stated next, will be that if *a* is a cyclic anchor you may not use it as anchor: the type **like** *a* will be invalid.

End

8.14.20 Definition: Deanchored form of a type

Code: *DTDA*

The **deanchored form** of a type *T* in a class *C* is the type (Class_or_tuple_type or Formal_generic) defined as follows:

1 If *T* is **like Current**: the current type of *C*.

2 If *T* is **like** *anchor* where the type *AT* of *anchor* is not anchored: (recursively) the deanchored form of *AT*.

3 If *T* is **like** *anchor* where the type *AT* of *anchor* is anchored but *anchor* is not a cyclic anchor: (recursively) the deanchored form of *AT* in *C*.

4 If *T* is *a AT*, where *a* is an Attachment_mark: *a DT*, where *DT* is (recursively) the deanchored form of *AT* deprived of its Attachment_mark if any.

5 If none of the previous cases applies: *T* after replacement of any actual parameter by (recursively) its deanchored form.

Informative Text

8.14.20.1 Although useful mostly for anchored types, the notion of “deanchored form” is, thanks to the phrasing of the definition, applicable to *any* type. Informally, the deanchored form yields, for an anchored type, what the type “really means”, in terms of its anchor’s type. It reflects the role of anchoring as what programmers might call a macro mechanism, a notational convenience to define types in terms of others.

8.14.20.2 Case 4 enables us to treat **detachable like** *anchor* as a detachable type whether the type of *anchor* is attached or detachable.

End

8.14.21 Validity: Anchored Type rule

Code: *VTAT*

It is valid to use an anchored type *AT* of the form **like** *anchor* in a class *C* if and only if it satisfies the following conditions:

1 *anchor* is either **Current** or the final name of a query of *C*.

2 *anchor* is not a cyclic anchor.

3 The deanchored form *UT* of *AT* is valid in *C*.

The base class and base type of *AT* are those of *UT*.

Informative Text

8.14.21.1 An anchored type has no properties of its own; it stands as an abbreviation for its unfolded form. You will not, for example, find special conformance rules for anchored type, but should simply apply the usual conformance rules to its deanchored form.

Signed by JV - 2026/10/8

8.14.21.2 Anchored declaration is essentially a syntactical device: you may always replace it by explicit redefinition. But it is extremely useful in practice, avoiding much code duplication when you must deal with a set of entities (attributes, function results, routine arguments) which should all follow suit whenever a proper descendant redefines the type of one of them, to take advantage of the descendant's more specific context.

End

8.14.22 Definition: Attached, detachable

Code: *DTAT*

A type is **detachable** if its deanchored form is a Class_type declared with the **detachable** Attachment_mark.

A type is **attached** if it is not detachable.

Informative Text

8.14.22.1 By taking the "deanchored form", we can apply the concepts of "attached" and "detachable" to an anchored type **like** *a*, by just looking at the type of *a* and finding out whether it is attached or not.

8.14.22.2 As a consequence of this definition, an expanded type is attached.

8.14.22.3 As the following semantic definition indicates, the idea of declaring a type as attached is to guarantee that its values will never be void.

End

8.14.23 Semantics: Attached type semantics

Code: *MTAT*

Every run-time value of an attached type is non-void (attached to an object).

Informative Text

8.14.23.1 In contrast, values of a detachable type may be void.

8.14.23.2 These definitions rely on the run-time notion of a *value* being attached (to an object) or void. So there is a distinction between the *static* property that an entity is attached (meaning that language rules guarantee that its run-time values will never be void) or detachable, and the *dynamic* property that, at some point during execution, its value will be attached or not. If there's any risk of confusion we may say "statically attached" for the entity, and "dynamically attached" for the run-time property of its value.

8.14.23.3 The validity and semantic rules, in particular on attachment operations, ensure that attached types indeed deserve this qualification, by initializing all the corresponding entities to attached values, and protecting them in the rest of their lives from attachment to void.

8.14.23.4 From the above semantics, the **attached** mark appears useless since an absent Attachment_mark has the same effect. The mark exists to ensure a smooth transition: since earlier versions of Eiffel did not guarantee void-safety, types were detachable by default. To facilitate adaptation to current Eiffel and avoid breaking existing code, compilers may offer a compatibility option (departing from the Standard, of course) that treats the absence of an Attachment_mark as equivalent to **detachable**. You can then use **atached** to mark the types that you have moved to the attached world and adapt your software at your own pace, class by class if you wish, to the new, void-safe convention.

End

8.14.24 Definition: Stand-alone type

Code: *STSA*

A Type is **stand-alone** if and only if it involves neither any Anchored type nor any Formal_generic_name.

Informative Text

8.14.24.1 In general, the semantics of a type may be relative to the text of class in which the type appears: if the type involves generic parameters or anchors, we can only understand it with respect to some class context. A stand-alone type always makes sense — and always makes the same sense — regardless of the context.

8.14.24.2 We restrict ourselves to stand-alone types when we want a solidly defined type that we can use anywhere. This is the case in the validity rules enabling creation of a root object for a system, and the definition of a once function.

End

Signed by JV - 2026/10/8

8.15 Genericity

Informative Text

8.15.0.1 The types discussed so far were directly defined by classes. The *genericity* mechanism, while still based on classes, gives us a new level of flexibility through **type parameterization**. You may for example define a class as *LIST* [*G*], yielding not just one type but many: *LIST* [*INTEGER*], *LIST* [*AIRPLANE*] and so on, parameterized by *G*.

8.15.0.2 Parameterized classes such as *LIST* are known as **generic classes**; the resulting types, such as *LIST* [*INTEGER*], are **generically derived**. "Genericity" is the mechanism making generic classes and generic derivations possible.

8.15.0.3 With genericity, the concepts of class and type become distinct, although closely related. A class is a program unit, and may have generic parameters, or not. A type is either:

8.15.0.3.1 • A class without generic parameters (a "**non-generic class**"), such as *INTEGER*. In this case, a class directly yields a type.

8.15.0.3.2 • A generic derivation produced from a generic class such as *LIST* by providing actual generic parameters — as many as required by the class declaration (one in this case). The actual generic parameters are themselves (recursively) types. Using *LIST* as the generic class and *INTEGER* as the actual generic parameter yields the generically derived type *LIST* [*INTEGER*]. Using that type itself as the actual generic parameter yields *LIST* [*LIST* [*INTEGER*]].

8.15.0.4 Two forms of genericity are available:

8.15.0.4.1 • With *unconstrained* genericity, *G* in the declaration of a class *C* [*G*] represents an arbitrary type. The *LIST* example illustrates this case: *LIST* [*T*] is valid for any type *T*, since it is possible to group objects of any kind into lists.

8.15.0.4.2 • With *constrained* genericity, you can demand certain properties of the types represented by *G*, enabling you to do more with *G* in the class text. For example if you declare a class as *SORTABLE* [*G –> COMPARABLE*], a generic derivation *SORTABLE* [*T*] will only be valid if the actual generic parameter *T* conforms to *COMPARABLE*. If *COMPARABLE* describes objects to which a total order relation is applicable, with functions such as *less_than*, then the features of *SORTABLE*, for example a procedure *sort*, may use these functions on targets of type *G*, as in a boolean expression *x*.*less_than* (*y*) (which can also be written *x* < *y* if *less_than* has an alias "<") for *x* and *y* declared of type *G*.

End

8.15.1 Syntax: Actual generic parameters

Code: *SGGP*

Actual_generics ≜ "[" Type_list "]"

Type_list ≜ {Type "," …}$^{+}$

8.15.2 Definition: Actual generic parameter

Code: *DGAP*

The list of **actual generic parameters** of a type *T* is the Type_list of its Actual_generics if *T* is generically derived, its type sequence if it is a tuple type, and empty otherwise.

Informative Text

8.15.2.1 For generality, this definition introduces "actual generic parameters" for any type, but the result is only interesting for a generically derived type *C* [*T, U, …*] or a tuple type *TUPLE* [*a*:*T*; *b*: *U*; …] (where the labels *a, b, …* are optional and the "type sequence" is *T, U, …*). In both cases the actual generic parameters are *T, U, …* For all other kinds of type, the list is empty.

End

Signed by JV - 2026/10/8

8.15.3 Syntax: Formal generic parameters

Code: *SGFG*

Formal_generics ≜ "[" Formal_generic_list "]"

Formal_generic_list ≜ {Formal_generic ","…}+

Formal_generic ≜ [Variance_mark] Formal_generic_name [Constraint]

Formal_generic_name ≜ [**detachable**] Identifier

Informative Text

8.15.3.1 The formal generic parameters represent types which will be specified in each generic derivation.

8.15.3.2 Adding the **variant** qualification to a formal generic, as in *C* [**variant** *G*] rather than just *C* [*G*], means that generic derivations will support conformance on the corresponding actual parameters: *C* [*U*] conforms to *C* [*T*] if *U* conforms to *T*. Without such a mark, conformance requires the actual generic parameter to be exactly the same. To avoid run-time type problems, **variant** parameters imply specific constraints.

8.15.3.3 Adding the **invariant** qualification to a formal generic, as in *D* [**invariant** *G*] rather than just *C* [*G*], means that conformance on the corresponding generically derived classes requires identical actual parameters: whereas *C* [*U*] conforms to *C* [*T*] if *U* conforms to *T*, *D* [*U*] does not conform to *D* [*T*] if *U* is not *T*.

8.15.3.4 Adding the **detachable** mark to a Formal_generic_name, as in **detachable** *G*, means that the class may declare *self-initializing* variables (variables that will be initialized automatically on first use) of type *G*; this requires that any actual generic parameter that is an attached type must also be self-initializing, that is to say, make *default_create* from *ANY* available for creation.

End

8.15.4 Definition: Generic class, formal generic parameter, formal parameter

Code: *DGFP*

A class *C* is **generic** if its declaration includes Formal_generics part.

The list of **formal generic parameters** of a class *C* is its Formal_generic_list if *C* is generic, and empty otherwise.

"**Formal parameter**" is a synonym for "formal generic parameter".

"**Genericity**" denotes the possibility of making classes generic.

Informative Text

8.15.4.1 Unlike "actual generic parameter" in the previous definition (DGAP, page 173), the notion of formal generic parameter only applies to classes, not tuples.

8.15.4.2 As a reminder, the word "parameter" is reserved for type parameters of classes (formal parameters) and types (actual parameters). For values passed to routines, the term is "argument".

End

8.15.5 Validity: Formal Generic rule

Code: *VGFG*

A Formal_generics part of a Class_declaration is valid if and only if every formal generic parameter *G* of Formal_generic_name *GN* satisfies the following conditions:

1 *GN* is different from the name of any class in the universe.

2 *GN* is different from any other Formal_generic_name appearing in the same Formal_genericspart.

Signed by JV - 2026/10/8

Informative Text

8.15.5.1 ~~Adding the **invariant** qualification to a formal generic, as in *D* [**invariant** *G*] rather than just *C* [*G*], means that conformance on the corresponding generically derived classes requires identical actual parameters: whereas *C* [*U*] conforms to *C* [*T*] if *U* conforms to *T*, *D* [*U*] does not conform to *D* [*T*] if *U* is not *T*.~~ CHECK THIS

8.15.5.2 ~~Adding the **detachable** mark to a Formal_generic_name, as in **detachable** *G*, means that the class may declare *self initializing* variables (variables that will be initialized automatically on first use) of type *G*. In this case, ; any actual generic parameter that is an attached type must also be self initializing, that is to say, make *default_create* from *ANY* available for creation.~~ CHECK THIS (REPETION OF PREVIOUS TEXT 8.14.3)

End

8.15.6 Definition: Constrained, unconstrained

Code: *DGGC*

If a formal generic parameter of a generic class is declared with a Constraint, the parameter is **constrained**; if not, it is **unconstrained**.

A generic class is itself **constrained** if it has at least one constrained parameter, **unconstrained** otherwise.

Informative Text

8.15.6.1 An unconstrained formal generic parameter, such as *G* in *LIST* [*G*], signifies that any type is acceptable as actual generic parameter.

8.15.6.2 A constrained generic parameter, as in *SORTABLE* [*G –> COMPARABLE*], indicates that an actual generic parameter must conform to the constraining type, here *COMPARABLE*.

End

Informative Text

8.15.6.3 A generic class does not describe a type but a template for a set of possible types. To obtain an actual type, you must provide an Actual_generics list, whose elements are themselves types. This has a name too, per the following definition.

End

8.15.7 Definition: Generic derivation, non-generic type

Code: *DGGD*

The process of producing a type from a generic class or *TUPLE* by providing one or more actual generic parameters is **generic derivation**.

A type resulting from a generic derivation is a **generically derived type**, or just **generic type**.

A type that is not generically derived is a **non-generic type**.

Informative Text

8.15.7.1 It is preferable to stay away from the term “generic instantiation” (sometimes used in place of “generic derivation”) as it creates a risk of confusion with the normal meaning of “instantiation” in object-oriented development: the *run-time process of obtaining an object from a class.*

End

8.15.8 Definition: Self-initializing formal generic parameter

Code: *DGSI*

A Formal_generic parameter is **self-initializing** if and only if its declaration includes the optional **detachable** mark.

Informative Text

8.15.8.1 The notion of self-initializing formal generic parameter is related to the notion of self-initializing *type* (DEST, page 246): a type which makes *default_create* from *ANY* available for creation. The rule will be that an actual generic parameter corresponding to a self-initializing formal parameter must itself, if attached, be a self-initializing type.

End

Informative Text

8.15.8.2 The next important concept is the *constraint* (short for "generic constraint") of a formal generic parameter. This notion is defined whether or not the formal generic parameter is explicitly specified with a constraining type.

8.15.8.3 If the parameter is explicitly constrained, appearing as *G* –> *T*, the constraint *T* determines what operations are applicable, in the class to an expression of type *G*: the features of *T*. This property will be formalized very simply by defining the base type of *G*, in this case, as being *T*.

8.15.8.4 In the case of unconstrained genericity, we do not know anything about future actual generic parameters: in *C* [*G*], *G* can represent any type. The only operations that we can apply in this case are those of *ANY*, since we know every Eiffel class is a descendant of *ANY*.

8.15.8.5 In addition, *G* could be an attached type but this is not guaranteed; so to cover to all cases we should treat *G* as if it were constrained by **detachable** *ANY* (as attached types conform to their detachable variants but not the other way around).

End

8.15.9 Definition: Constraint, constraining types of a Formal_generic

Code: *DGCO*

The **constraint** of a formal generic parameter is its Constraint part if present, and otherwise **–> separate detachable** *ANY*.

Its **constraining types** are all the types listed in its Constraining_types if present, and otherwise **separate detachable** *ANY*.

8.15.10 Syntax: Generic constraints

Code: *SGGC*

Constraint ≜ "**–>**" Constraining_types [Constraint_creators]

Constraining_types ≜ Single_constraint | Multiple_constraint

Single_constraint ≜ Type [Renaming]

Renaming ≜ Rename **end**

Multiple_constraint ≜ "**{**" Constraint_list "**}**"

Constraint_list ≜ {Single_constraint "**,**" …}$^{+}$

Constraint_creators ≜ **create** Feature_list **end** **--- CHECK THIS**

8.15.11 Validity: Generic Constraint rule

Code: *VGGC*

A Constraint part appearing in the Formal_generics part of a class *C* is valid if and only if it satisfies the following conditions for every Single_constraint listing a type *T* in its Constraining_types:

1 *T* involves no anchored type.

2 If a Renaming clause **rename** *rename_list* **end** is present, a class definition of the form **class** *NEW* **inherit** *BT* **rename** *rename_list* **end** (preceded by **deferred** if the base class of *T* is deferred), where *BT* is the base type of *T*, would be valid.

3 Every name listed in the Constraint_creators is among the generically constrained names of a feature of exactly one of the corresponding constraining types, which is a procedure.

Signed by JV - 2026/10/8

Informative Text

8.15.11.1 There is no requirement here on the Constraint_creators part, although in most cases it will list names (after Renaming) of creation procedures of the constraining types. The precise requirement is captured by other rules.

8.15.11.2 Condition 2 implies that the features listed in the Constraint_creators are, after possible Renaming, names of features of one or more of the constraining types, and that no clash remains that would violate the rules on inheritance. In particular, you can use the Renaming either to merge features if they come from the same seeds, or (the other way around) separate them.

8.15.11.3 If *T* is based on a deferred class the fictitious class *NEW* should be declared as **deferred** too, otherwise it would be invalid if *T* has deferred features. On the other hand, *NEW* cannot be valid if *T* is based on a frozen class; in this case it is indeed desirable to disallow the use of *T* as a constraint, since the purpose of declaring a class **frozen** is to prevent inheritance from it.

End

8.15.12 Definition: Constraining creators

Code: *DGCC*

If *G* is a formal generic parameter of a class, the **constraining creators of** *G* are the features of *G*'s Constraining_types, if any, corresponding after possible Renaming to the feature names listed in the Constraint_creators if present.

Informative Text

8.15.12.1 Constraining creators should be creation procedures, but not necessarily (as seen below) in the constraining types themselves; only their instantiatable descendants are subject to this rule.

End

8.15.13 Validity: Generic Derivation rule

Code: *VGGD*

Let *C* be a generic class. A Class_type *CT* having *C* as base class is valid if and only if its tuple-type-unfolded form satisfies the following conditions for every actual generic parameter *T* and every Single_constraint *U* appearing in the constraint for the corresponding formal generic parameter *G*:

1 The number of Type components in *CT*'s Actual_generics list is the same as the number of Formal_generic parameters in the Formal_generic_list of *C*'s declaration.

2 *T* conforms to the type obtained by applying to *U* the generic substitution of *CT*.

3 If *C* is expanded, *CT* is generic-creation-ready.

4 If *G* is a self-initializing formal parameter and *T* is attached, then *T* is a self-initializing type.

Informative Text

8.15.13.1 The rule applies to the "tuple-type-unfolded form" (DJTT, page 187), a syntactical simplification which in the special case of tuple constraints allows you to drop *TUPLE* and the associated brackets, writing for example *C* [*A*, *TUPLE* [*U*, *V*], *W*] as just *C* [*A*, *U*, *V*, *W*]. This case is rare: in usual conditions the above rule applies directly to *CT*, which is its own tuple-type-unfolded form.

8.15.13.2 In the case of unconstrained generic parameters, only condition 1 applies, since the constraining type in that case is *ANY*, which trivially satisfies the other two conditions.

8.15.13.3 Condition 3 follows from the semantic rule permitting "lazy" creation of entities of expanded types on first use, through *default_create*. Generic-creation-readiness (defined next) is a condition on the actual generic parameters that makes such initialization safe if it may involve creation of objects whose type is the corresponding formal parameters.

8.15.13.4 Condition 4 guarantees that if *C* relies, for some of its variables of type *G*, on automatic initialization on first use, *T* provides it, if attached (remember that this includes the case of expanded types), by making *default_create* from *ANY* available for creation. If *T* is detachable this is not needed, since *Void* will be a suitable initialization value. “Self-initializing formal parameter” is defined in DGSI, page 175.

End

8.15.14 Definition: Generic-creation-ready type

Code: *DGCR*

A type of base class *C* is **generic-creation-ready** if and only if every actual generic parameter *T* of its deanchored form satisfies the following conditions:

1 If there is a corresponding formal generic parameter and its specification includes a Constraint_creators, the versions in *T* of the constraining creators the corresponding formal generic parameter are creation procedures, available for creation to *C*, and *T* is (recursively) generic-creation-ready.

2 If *T* is expanded, it is (recursively) generic-creation-ready.

Informative Text

8.15.14.1 The role of this condition is to make sure that if class *C* […, *G* , …] may cause a creation operation on a target of type *G* — as permitted only if the class appears as *C* […, *G* **–>** *CONST* **create** *cp1*, … **end**, …] — then the corresponding actual parameters, such as *T*, will support the given features — the “constraining creators” — as creation procedures.

8.15.14.2 It might then appear that generic-creation-readiness is a validity requirement on *any* actual generic parameter. But this would be more restrictive than we need. For example *T* might be a deferred type; then it cannot have any creation procedures, but that’s still OK because we cannot create direct instances of *T*, only of its effective descendants. Only if it is possible to **create** an actual object of the type do we require generic-creation-readiness. Overall, we need generic-creation-readiness only in specific cases, including:

8.15.14.2.1 • For the creation type of a creation operation: conditions 4 of the Creation Instruction rule (VKIN, page 235) and 3 of the Creation Expression rule (VKEX, page 237).

8.15.14.2.2 • For a Parent in an Inheritance part: condition 6 of the Parent rule.

8.15.14.2.3 • For an expanded type: condition 3 of the just seen Generic Derivation rule.

8.15.14.3 Although phrased so that it is applicable to any type, the generic-condition-readiness condition is only interesting for generically derived types of the form *C* […, *T*, …]. Indeed DGAP, page 173, defines the list of actual generic parameters for any type, but the list can only be non-empty for generically-derived types, or for tuple types to which the above condition does not apply (since in this case, per DGFP, page 174, there is no “corresponding formal generic parameter” in condition 1 of the above definition).

End

8.15.15 Semantics: Generically-derived Class_type semantics

Code: *MGGD*

A generically derived Class_type of the form *C* […], where *C* is a generic class, is expanded if *C* is an expanded class, reference otherwise. It is its own base type, and its base class is *C*.

Informative Text

8.15.15.1 So *LINKED_LIST* [*POLYGON*] is its own base type, and its base class is *LINKED_LIST*.

End

Signed by JV - 2026/10/8

8.15.16 Definition: Base type of a single-constrained formal generic

Code: *DGBS*

The base type of a constrained Formal_generic_name *G* having as its constraining types a Single_constraint listing a type *T* is:

1 If *T* is a Class_type: *T*.

2 Otherwise (*T* is a Formal_generic_name): the base type of *T* if it can be determined by (recursively) case 1, otherwise *ANY*.

Informative Text

8.15.16.1 The definition is never cyclic since the only recursive part is the use of case 1 from case 2.

8.15.16.2 Case 1 is the common one: for *C* [*G –> T*] we use as base type of *G*, in *C*, the base type of *T*. We need case 2 to make sure that this definition is not cyclic, because we permit cases such as *C* [*G, H–> D* [*G*]], and as a consequence cases such as *C* [*G –> H, H–> G*] or even *C* [*G –> G*] even though they are not useful; both of these examples yield *ANY* as base types for the parameters.

8.15.16.3 As a result of the definition of "constraining types", the base type of an unconstrained formal generic, such as *G* in *C* [*G*], is also *ANY*.

End

8.15.17 Definition: Base type of an unconstrained formal generic

Code: *DGBU*

The base type of an unconstrained Formal_generic_name type is *ANY*.

8.15.18 Definition: Reference or expanded status of a formal generic

Code: *DGRE*

A Formal_generic_name represents a reference type or expanded type depending on the corresponding status of the associated actual generic parameter in a particular generic derivation.

8.15.19 Semantics: Generic substitution

Code: *MGSB*

Every type *T* defines a mapping σ from names to types known as its **generic substitution**:

1 If *T* is generically derived, σ associates to every Formal_generic_name the corresponding actual parameter of *T*'s tuple-type-unfolded form.

2 Otherwise, σ is the identity substitution.

Informative Text

8.15.19.1 The generic substitution defines the mapping from actual to formal parameters induced by any particular generic derivation. For example if a class is declared as *C* [*G, H*] and you use the type *C* [*T, U*], a generic derivation, the generic substitution is the mapping σ that associates *T* to *G* and *U* to *H*.

8.15.19.2 We need this generic substitution to understand the semantics of generic classes: σ tells us that in the context of this particular generic derivation any use of *G* in the text of class *C*, for example in the definition of a routine *r* (*x*: *G*), denotes *T*, and any use of *H* denotes *U*. Another generic derivation, for example *C* [*V, W*], would induce a different σ.

8.15.19.3 The rule applies to the "tuple-type-unfolded form" (DJTT, page 187), a syntactical simplification which in the special case of tuple constraints allows you to drop *TUPLE* and the associated brackets, writing for example *C* [*A*, *TUPLE* [*U*, *V*], *W*] as just *C* [*A*, *U*, *V*, *W*].

End

8.15.20 Principle: Generic Type Adaptation rule

Code: *PGTA*

The signature of an entity or feature *f* of a type *T* of base class *C* is the result of applying *T*'s generic substitution to the signature of *f* in *C*.

Signed by JV - 2026/10/8

Informative Text

8.15.20.1 The signature include both the type of an entity or query, and the argument types for a routine; the rule is applicable to both parts.

End

8.15.21 Definition: Generically constrained feature name

Code: *DGFN*

Consider a generic class *C*, a constrained Formal_generic_name *G* of *C*, a type *T* appearing as one of the Constraining_typesfor *G*, and a feature *f* of name *fname* in the base class of *T*. The **generically constrained names** of *f* for *G* in *C* are:

1 If one or more Single_constraintclauses for *T* include a Renaming part with a clause *fname* **as** *ename*, where the Feature_name part of *ename* (an Extended_feature_name) is *gname*: all such *gname*.

2 Otherwise: just *fname*.

8.15.22 Validity: Multiple Constraints rule

Code: *VGMC*

A feature of name *fname* is applicable in a class *C* to a target *x* whose type is a Formal_generic_name *G* constrained by two or more types *CONST1*, *CONST2*,…, if and only if exactly one of the $CONST_i$ has a feature such that *fname* is one of the generically constrained names for *G* in *C*, and that feature is available to *C*.

Informative Text

8.15.22.1 COMPLETE.

8.15.22.2 Explain that when there is more than one the programmer should use renaming.

End

8.15.23 Definition: Base type of a multi-constraint formal generic type

Code: *DGBM*

The base type of a multiply constrained Formal_generic_name is a type generically derived, with the same actual parameters as the current class, from a fictitious class with none of the optional parts except for Formal_generics and an Inheritance clause that lists all the constraining types as parents, with the given Renaming clause if any, and resolves any conflicts between potentially ambiguous features by further renaming them to new names not available to developers and adding Select clauses (arbitrarily chosen) to resolve any repeated-inheritance conflicts.

Informative Text

8.15.23.1 class C [G -> {U, V}]

8.15.23.2

8.15.23.3 class U inherit ANY rename is_equal as is_equal_U redefine is_equal_U end

8.15.23.4

8.15.23.5

Signed by JV - 2026/10/8

8.16 Tuples

Informative Text

8.16.0.1 Based on a bare-bones form of class — with no class names — tuple types provide a concise and elegant solution to a number of issues:

8.16.0.1.1 • Writing functions with multiple results, ensuring complete symmetry with multiple arguments.

8.16.0.1.2 • Describing sequences of values of heterogeneous types, or "tuples", such as [*some_integer*, *some_string*, *some_object*], convenient for example as arguments to printing routines.

8.16.0.1.3 • Achieving the effect of routines with a variable number of arguments.

8.16.0.1.4 • Achieving the effect of generic classes with a variable number of generic parameters.

8.16.0.1.5 • Using simple classes, defined by a few attributes and the corresponding assigner commands — similar to the "structures" or "records" of non-O-O languages, but in line with O-O principles — without writing explicit class declarations.

8.16.0.1.6 • Making possible the agent mechanism through which you can handle routines as objects and define higher-order routines.

End

8.16.1 Syntax: Tuple types

Code: *SJTU*

Tuple_type ≜ *TUPLE* [Tuple_parameter_list]

Tuple_parameter_list ≜ "**[**" Tuple_parameters "**]**"

Tuple_parameters ≜ Type_list | Entity_declaration_list

Informative Text

8.16.1.1 In its simplest form, a Tuple_type is just *TUPLE* possibly followed by a list of types in brackets.

8.16.1.2 The Tuple_parameter_list is optional: *TUPLE* just by itself is a valid Tuple_type.

8.16.1.3 If the Tuple_parameter_list is present, it can be of two forms, unlabeled and labeled:

- The unlabeled form contains types separated by semicolons, as in *TUPLE* [*INTEGER*; *INTEGER*; *REAL*; *POLYGON*].
- For the Tuple_parameters, the second possibility is Entity_declaration_list, meaning that you can also use a form such as *TUPLE* [*i*, *j*: *INTEGER*; *r*: *REAL*; *p*: *POLYGON*], where each type has an associated label, which will be used to identify components of the corresponding tuples (for example, the components of a manifest tuple, as introduced next).

8.16.1.4 You cannot mix the two styles: labels must be present either for all Tuple_parameters or for none of them.

End

8.16.2 Syntax: Manifest tuples

Code: *SJMT*

Manifest_tuple ≜ "**[**" Expression_list "**]**"

Expression_list ≜ {Expression "**,**" …}

8.16.2.1 A Manifest_tuple gives a tuple through the enumeration of its elements, as in [0, 1, 3.5, *pol*].

8.16.2.2 Tuple type labels do not appear in manifest tuples: this example (with *pol* of type *POLYGON*) is an instance of *TUPLE* [*INTEGER*; *INTEGER*; *REAL*; *POLYGON*] but also of such labeled tuple types as *TUPLE* [*i*, *j*: *INTEGER*; *r*: *REAL*; *p*: *POLYGON*].

End

Signed by JV - 2026/10/8

8.16.3 Definition: Tuple type, type sequence

Code: *DJTS*

A type defined as a Tuple_type is all called a **tuple type**.

The **type sequence** of a tuple type is the sequence of types obtained by listing its parameters, if any, in the order in which they appear, every labeled parameter being listed as many times as it has labels.

Informative Text

8.16.3.1 The type sequence for *TUPLE* is empty; the type sequence for both *TUPLE* [*INTEGER*; *INTEGER*; *REAL*; *POLYGON*] and *TUPLE* [*i*, *j*: *INTEGER*; *r*: *REAL*; *p*: *POLYGON*] is *INTEGER*, *INTEGER*, *REAL*, *POLYGON*, where *INTEGER* appears twice — in the second case because of the two labels *i, j*.

8.16.3.2 The elements of the type sequence are known as the tuple type's "actual generic parameters": see DGAP, page 173.

8.16.3.3 The "if any" part of the definition reminds us that a tuple type need not have actual generic parameters: *TUPLE* without further qualification is a valid type (covering tuples with zero or more elements). In this case the type sequence is empty.

End

8.16.4 Definition: Value sequences associated with a tuple type

Code: *DJVS*

The set of **value sequences** associated with a tuple type *T* has as its elements all sequences of values of which each appears at the corresponding position in *T*'s type sequence.

Informative Text

8.16.4.1 The set of value sequences for *TUPLE* contains a single element, the empty sequence.

8.16.4.2 The set of value sequences for *TUPLE* [*NTEGER*; *INTEGER*; *REAL*; *POLYGON*] and *TUPLE* [*i*, *j*: *INTEGER*; *r*: *REAL*; *p*: *POLYGON*] contains all 4-element sequences of which the first two are integers, the third a *REAL*, and the last a reference to an object of type *POLYGON*.

8.16.4.3 As the example indicates, such a sequence is not an Eiffel specimen but a sequence of run-time values (objects or references depending on whether the corresponding element of the type sequence is an expanded or reference type). To denote such a sequence in Eiffel, the program may use a manifest tuple, such as [0, 1, 3.5, *pol*], or some other kind of expression whose type is a tuple type.

End

8.16.5 Definition: Labeled tuple type, label

Code: *DJLB*

A Tuple_type is **labeled** if its Tuple_parameters are given by an Entity_declarations (rather than the alternative, a Type_list).

The identifiers in the Entity_list specimens of a labeled tuple type's Entity_declaration_group are its **labels**.

Informative Text

8.16.5.1 Labels, when present, play no role in the semantics of tuples and their conformance properties. They never intervene in manifest tuples (such as [25, –8.75, *pol*]). Their only use is to allow name-based access to tuple fields, as *your_tuple*.*label*, guaranteeing statically the type of the result.

Signed by JV - 2026/10/8

8.16.5.2 More precisely, for a tuple *t: TUPLE* [*a*: *S*; *b*: *T*; …], where *l* is one of the labels, you may access the corresponding field using a simple notation, *t.x* where *x* is one of the labels *a*, *b*, …. You may do without this notation by using for example *t.item* (*x*), but this is of type **detachable** *ANY* and hence requires an Object_test to yield the desired type, here *T*. With *t.x* you get an expression of the appropriate type.

8.16.5.3 You may also use this notation to *change* the value of the field, through an Assigner_call *t.x := some_value*.

8.16.5.4 These mechanisms are similar to those available for classes; the notion of unfolded form of a tuple, introduced below, makes them officially possible through the following definition.

End

8.16.6 Validity: Tuple Type rule

Code: *VJTT*

A labeled Tuple_type is valid if and only if all its labels are fresh.

Informative Text

8.16.6.1 A label, or generally any identifier, is “fresh” (D7FI, page 72) if it does not conflict with an identifier already used for some other purpose in the current context.

8.16.6.2 The label-freshness requirement, which only applies to the labeled variant, is the only constraint on tuple types. There is no restriction on the type sequence: you may use tuples of anything.

End

8.16.7 Definition: Unfolded form of a tuple type

Code: *VJUT*

The unfolded form of a labeled Tuple_type *TT* is a fictitious class, consisting solely of the following generally available feature declarations, two for every label *x* associated in *TT* with a type *T*:

1 An attribute declared as
x: *T* **assign** *set_x*

2 A procedure declared as
set_x (*v*: *T*) **do** *x* := *v* **end**

and, in addition, two functions and a procedure with the following contract views, where *N* is the number of elements in *TT*’s type sequence:

```
item (i: NATURAL) alias "[]": ANY assign put
        -- Value of the i-th tuple element.
    require
        not_too_small: i >= 1
        not_too_big: i <= N

valid_type (v: ANY ; i: NATURAL): BOOLEAN
        -- Is v of a type that conforms to the type of the i-th tuple element?

put (v: ANY; i: NATURAL)
        -- Replace by v the value of the i-th tuple element?
    require
        good_type: valid_type (v, i)
        not_too_small: i >= 1
        not_too_big: i >= N
    ensure
        item (i) = v
```


Signed by JV - 2026/10/8

The unfolded form of a non-labeled Tuple_type is (recursively) the unfolded form of a labeled Tuple_type obtained by equipping every type in its Tuple_parameters with a label, chosen as an arbitrary fresh identifier.

Informative Text

8.16.7.1 The unfolded form serves only to define properties of tuple field access and modification. It makes it possible in particular to access the element of label *x* in a labeled tuple *t* through the notation *t*.*x*, and to set its value through the assigner command *t*.*x* := *v*.

8.16.7.2 Any call to *item* or *put* (including under the forms *t* [*i*] and *t* [*i*] := *v*, available thanks to the bracket alias and **assign** clause) requires in practice an Object_test since these features use *ANY* as the type of (respectively) their result and first argument. These features are, as a consequence, intended for unlabeled tuples only, since in the labeled case the *t*.*x* forms are simpler and can be type-checked statically rather than through a precondition.

8.16.7.3 The definition of the unfolded form yields a very simple class, made only of attribute and procedure declarations, one each for every label, plus the three features *item*, *valid_index* and *put*. In particular there is no Inheritance clause; it could be useful for defining conformance properties of tuple types, but these are described directly, without reference to the unfolded form.

8.16.7.4 In the case of an unlabeled tuple type, any property deduced from the unfolded form must be independent of the choice of labels, since it is arbitrary. This is indeed the case with all validity and semantic rules in the presentat Standard that are applicable to tuples and rely on the notion of unfolded form.

End

8.16.8 Semantics: Tuple type values

Code: *MJVA*

The possible **values** of a Tuple_type are the direct instances of its unfolded form.

Informative Text

8.16.8.1 This semantic rule also implies that the “generator” of a tuple object (a notion defined for every object) is the unfolded form of its type.

End

Informative Text

8.16.8.2 The present Standard does not contain any specific rule (syntax, validity or semantics) on how to access the elements of a tuple *t*. The reason is that tuple access is covered by mechanisms applicable to any class, and applied here to the unfolded form:

8.16.8.2.1 • If *t* is a labeled tuple and *x* one of its labels, *t*.*x*, is an attribute call on *t* (since *x* is an attribute of the fictitious class defined by the unfolded form) and *t*.*x* := *v* is a procedure call (since *x* has an associated assigner command *set_x*). The validity and semantic rules of these constructs are in each case those of an Object_call on the unfolded form.

8.16.8.2.2 • For any tuple, labeled or not, access through *item* and setting through *put* (under any of the syntactical variants) are also defined as cases of Object_call on the unfolded form.

End

Informative Text

8.16.8.3 The next few clauses introduce two related syntactical mechanism, with the common aim of simplifying the use of tuples by dropping brackets when there is no ambiguity:

8.16.8.3.1 • **Tuple-argument unfolding** applies to routine calls, allowing you to write a call *r* (*t*, *u*, [*a*, *b*]) as just *r* (*t*, *u*, [*a*, *b*]).

Signed by JV - 2026/10/8

8.16.8.3.2 • **Tuple-type unfolding** applies to generic derivations, allowing you to write a generically derived type *C* [*T*, *U*, *TUPLE* [*A*, *B*]] as just *C* [*T*, *U*, *A*, *B*].

End

Informative Text

8.16.8.4 First, tuple-argument unfolding. Consider a routine *r* (*x*: *T*; *y*: *U*; *z*: *TUPLE* [*A*, *B*]). With *t* of type *T* and so on, you can have calls such as *r* (*t*, *u*, [*a*, *b*]) (or the qualified variant: *z*.*r* (*t*, *u*, [*a*, *b*])). Tuple-argument unfolding allows you to write such a call simply as *r* (*t*, *u*, *a*, *b*) if there is no ambiguity — as is usually the case, thanks to conditions specified in the rules below.

8.16.8.5 Since the type *TUPLE* [*A*, *B*] covers all tuples with two *or more* elements of which the first is of type *A* and the second of type *B*, the call *r* (*t*, *u*, [*a*, *b*, *c*]) is also valid. Tuple-argument unfolding allows you to write it as just *r* (*t*, *u*, *a*, *b*, *c*), giving the effect of a variable number of arguments.

8.16.8.6 Tuple-argument unfolding is particularly useful for agents, since the *call* and *item* library routines apply an agent to a list of arguments given as a tuple. For example if a is an agent associated with a routine *f* (*x*: *T*; *y*: *U*), then *a*.*call* ([*t*, *u*]) has the same effect as if you called the routine directly in *f* (*t*, *u*). Thanks to tuple-argument unfolding, you can write it as just *a*.*call* (*t*, *u*) without brackets.

8.16.8.7 For agents, the simplification yields an extra simplification: since *call* is a parenthesized feature (having the parenthesis alias "**()**", DFOV, page 101), you can write the call *a*.*call* (*t*, *u*) as just *a* (*t*, *u*), treating the agent *a* as if it were the routine *f* (known only at run time) that it represents.

End

8.16.9 Definition: Single-tuple routine, tuple position

Code: *DJSR*

A routine with *N* arguments (*N* ≥ 1) is a **single-tuple routine** if exactly one of its formal arguments, *fa*, is of a tuple type. In this case, the routine's **tuple position** *i* (1 ≤ *i* ≤ *N*) is the position of *fa* in the list of formal arguments.

Informative Text

8.16.9.1 The notion of "list of formal arguments" is defined precisely in DRFA, page 122. The "position" of an element in a list is its rank in the list, numbered from 1.

8.16.9.2 The routine *r* of the preceding Informative Text is single-tuple, with tuple position 3.

8.16.9.3 The idea of tuple-argument unfolding is that if you have a single-tuple routine you may drop the brackets around the tuple actual, as when we write *r* (*t*, *u*, *a*, *b*) for *r* (*t*, *u*, [*a*, *b*]).

End

8.16.10 Definition: Tuple-argument-unfolded form of a call

Code: *DJTA*

The **tuple-argument-unfolded form** of a call *c* with *n* actual arguments (*n* ≥ 0), where the feature *f* of *c* has *N* arguments (*N* ≥ 0), is *c* adapted through any of the following transformations that may be applicable, if *f* is a single-tuple routine of tuple position *i* (1 ≤ *i* ≤ *N*), to its Actuals *a*:

1 If *N* = 1 and *c* has no Actuals (*n* = 0, *i* = 1): add the following Actuals: ([]).

2 If 0 < *n* = *N* – 1: insert an empty tuple [] at position *i* in the Actual_list.

3 If 0 < *n* = *N* and the *i*-th element *e* of the Actual_list is not compatible with the corresponding formal argument of *f*: replace that element by [*e*].

4 If *n* > *N*: in the Actual_list, replace all elements at positions *k* for *i* ≤ *k* ≤ *i* + *N* – *n* by a single Manifest_tuple made of these elements.

Signed by JV - 2026/10/8

Informative Text

8.16.10.1 The tuple-argument-unfolded form is the default form of the argument list, including brackets around tuple arguments. This notion enables you to drop brackets for convenience; the rules governing calls refer to the tuple-argument-unfolded form in lieu of the Actual_list as written: see VUAR, page 204 and V5DA, page 258 for validity and MUGC, page 209 (step 4) for semantics.

8.16.10.2 In case 1, you have written the call as just *r*, an abbreviation for *r* ([]), with a single tuple argument which happens to be empty.

8.16.10.3 In case 2, you have written the call for example as just *r* (*a*, *c*), an abbreviation for *r* (*a*, [], *c*), where the second argument is a tuple which happens to be empty.

8.16.10.4 In case 3, you have written the call for example as just *r* (*a*, *b*, *c*), an abbreviation for *r* (*a*, [*b*], *c*), where the second formal argument is of a tuple type such as *TUPLE* [*T*]. Other validity rules (see VUAR, page 204) will ensure that *b* is of type *T* or compatible (conforming or converting). To avoid any ambiguity, the present rule forbids the — devious and unlikely — case in which *b* is itself a tuple of a type compatible with *TUPLE* [*T*]: the interpretation would be ambiguous since we could understand the argument as *b*, a tuple, or, by unfolding, [*b*], a tuple containing a tuple.

8.16.10.5 In case 4, you have included too many actuals, as in *r* (*a*, *b*, *c*, *d*, *e*) where the routine has three formal arguments, the second one of type *TUPLE* [*B*, *C*, *D*]. Then the three middle actuals just represent the tuple [*B*, *C*, *D*]. There is no ambiguity since *r* is, as in all other cases, a single-tuple routine.

8.16.10.6 A property that also applies to all cases, but is particularly relevant in the last one, helps guarantee that tuple-argument unfolding causes no ambiguity, for the human reader as well as for the compiler: if any of these transformations is applicable, *the call without unfolding would not be valid*. The following principle expresses this property.

End

8.16.11 Principle: Tuple-Argument-Unfolding property

Code: *PJTA*

If a call is argument-valid, it is identical to its tuple-argument-unfolded form.

Informative Text

8.16.11.1 This validity property is a result of the preceding definition of the tuple-argument-unfolded form and the definition of an argument-valid call (VUAR, page 204).

8.16.11.2 The result of the unfolding always has the right number of arguments, as expressed by the following property.

End

8.16.12 Principle: Tuple-Argument-Unfolding size

Code: *PJSA*

The cases of the definition of tuple-argument-unfolding are mutually exclusive. If any of them applies to a Call, the Actual_list resulting from the corresponding transformation has *N* elements where *N*, as in the definition, is the number of formal arguments.

Informative Text

8.16.12.1 So if tuple unfolding has any effect we are ready to pass on the resulting Actual_list to argument validity, for full type validation.

End

Signed by JV - 2026/10/8

Informative Text

8.16.12.2 The preceding definitions and principles complete the specification of tuple-argument unfolding, which lets you simplify *actual argument lists* in calls by dropping brackets around *tuples*. The specification of the corresponding mechanism for dropping brackets around *tuple types* in generically derived types, writing for example *C* [*T*, *U*, *TUPLE* [*A*, *B*]] as just *C* [*T*, *U*, *A*, *B*], occupies the remainder of 8.16. The clauses are directly rephrased from the preceding ones.

End

8.16.13 Definition: Single-tuple class, tuple constraint position

Code: *DJST*

A class A routine with *N* formal generic parameters (*N* ≥ 1) is a **single-tuple class** if exactly one of these parameters, *GP*, has a tuple type as its Constraint. In this case, the class's **tuple constraint position** *i* (1 ≤ *i* ≤ *N*) is the position of *GP* in the list of formal generic parameters.

Informative Text

8.16.13.1 The "position" of an element in a list is its rank in the list, numbered from 1.

8.16.13.2 The class *C* of the preceding Informative Text is single-tuple, with tuple position 3.

8.16.13.3 The idea of tuple-argument unfolding is that if you have a single-tuple routine you may drop the brackets around the tuple actual, as when we write *C* [*T, U, A, B*] for *C* [*T*, *U*, [*A*, *B*]).

End

8.16.14 Definition: Tuple-type-unfolded form of a type

Code: *DJTT*

The **tuple-type-unfolded form** of a type *T* with *n* actual generic parameters (*n* ≥ 0), where the base class *C* of *T* has *N* formal generic parameters (*N* ≥ 0), is *T* adapted through any of the following transformations that may be applicable, if *C* is a single-tuple class of tuple constraint position *i* (1 ≤ *i* ≤ *N*), to its Tuple_parameter_list:

1 If *N* = 1 and *T* has no Actual_generics (*n* = 0, *i* = 1): add the following Actual_generics: [*TUPLE*].

2 If 0 < *n* = *N* – 1: insert the type *TUPLE* at position *i* in the Type_list of the Actual_generics.

3 If 0 < *n* = *N* and the *i*-th element *E* of the Type_list of the Actual_generics is not compatible with the corresponding formal parameter of *C*: replace that element by *TUPLE* [*E*].

4 If *n* > *N*: in the Type_list of the Actual_generics, replace all elements at positions *k* for *i* ≤ *k* ≤ *i* + *N* – *n* by the single type *TUPLE* [*s*] where *s* is the sequence of these elements.

Informative Text

8.16.14.1 The tuple-argument-unfolded form is the default form of the actual parameter list, where tuple-type parameters are written in the usual form *TUPLE* [*A*, *B*, …]. This notion enables you to drop *TUPLE* and the brackets for convenience; the rules governing generic derivations refer to the tuple-argument-unfolded form in lieu of the Tuple_parameter_list as written: see the rule on generic derivation, VGGD, page 177 for validity, and the definition of generic substitution, MGSB, page 179, for semantics.

8.16.14.2 In case 1, you have written the generic derivation as just *C*, an abbreviation for *C* [*TUPLE*], with a single tuple tuple type, itself without parameters.

8.16.14.3 In case 2, you have written the generic derivation for example as just *C* [*T*, *V*], an abbreviation for *C* [*T*, *TUPLE*, *V*], where the second formal parameter of *C* is constrained by *TUPLE*, and you want to use just *TUPLE* (with no actual parameters of its own) as actual generic parameter, without bothering to write it in the absence of ambiguity.

Signed by JV - 2026/10/8

8.16.14.4 In case 3, you have written the call for example as just *C* [*T*, *U*, *V*], an abbreviation for *C* [*T*, *TUPLE* [*U*], *V*], where the second formal parameter of *C* is constrained by a tuple type such as *TUPLE* [*U*].

8.16.14.5 In case 4, you have included too many actual parameters, as in *T* [*U*, *V*, *W*, *X*, *Y*] where the class has three formal generic parameters, the second one constrained by *TUPLE* [*V*, *W*, *X*]. Then the three middle actual parameters just represent this tuple type *TUPLE* [*V*, *W*, *X*]. There is no ambiguity since *C* is, as in all other cases, a single-tuple class.

8.16.14.6 A property that also applies to all cases, but is particularly relevant in the last one, helps guarantee that tuple-argument unfolding causes no ambiguity, for the human reader as well as for the compiler: if any of these transformations is applicable, *the generic derivation without unfolding would not be valid*. The following principle expresses this property.

End

8.16.15 Principle: Tuple-Type-Unfolding property

Code: *PJTT*

If a generic derivation is valid, it is identical to its tuple-type-unfolded form.

Informative Text

8.16.15.1 This validity property is a result of the preceding definition of the tuple-argument-unfolded form and the definition of an argument-valid call (VUAR, page 204).

8.16.15.2 The result of the unfolding always has the right number of arguments, as expressed by the following property.

End

8.16.16 Principle: Tuple-Type-Unfolding Size property

Code: *PJST*

The cases of the definition of tuple-type-unfolding are mutually exclusive. If any of them applies to a class, the Actual_list resulting from the corresponding transformation has *N* elements where *N*, as in the definition, is the number of formal generic parameters.

Informative Text

8.16.16.1 So if tuple-type unfolding has any effect we are ready to pass on the resulting Tuple_parameter_list to type validity, for full type validation.

End

8.17 Conformance

Informative Text

8.17.0.1 Conformance is the most important characteristic of the Eiffel type system: it determines when a type may be used in lieu of another.

8.17.0.2 The most obvious use of conformance is to make assignment and argument passing type-safe: for *x* of type *T* and *y* of type *V*, the instruction *x* := *y*, and the call *some_routine* (*y*) with *x* as formal argument, will only be valid if *V* is compatible with *T*, meaning that it either *conforms* or *converts to T*. Conformance also governs the validity of many other constructs, as discussed below.

8.17.0.3 Conformance, as the rest of the type system, relies on inheritance. The basic condition for *V* to conform to *T* is straightforward:

8.17.0.3.1 • The base class of *V* must be a descendant of the base class of *T*.

8.17.0.3.2 • If *V* is a generically derived type, its actual generic parameters must conform to the corresponding ones in *T*: *B* [*Y*] conforms to *A* [*X*] only if *B* conforms to *A* and *Y* to *X*.

8.17.0.3.3 • If *T* is expanded, inheritance is not involved: *V* can only be *T* itself.

8.17.0.4 A full understanding of conformance requires the formal rules explained below, which take into account the details of the type system: constrained and unconstrained genericity, special rules for predefined arithmetic types, tuple types, anchored types.

8.17.0.5 The following discussion introduces the various conformance rules of the language as “definitions”. Although not validity constraints themselves, these rules play a central role in many of the constraints, so that language processing tools such as compilers may need to refer to them in their error messages. For that reason each rule has a validity code of the form VNCx.

End

8.17.1 Definition: Compatibility between types

Code: *DNCT*

A type is **compatible** with another if it either conforms or converts to it.

8.17.2 Definition: Compatibility between expressions

Code: *DNCE*

An expression *b* **is compatible with** an expression *a* if and only if *b* either conforms or converts to *a*.

Informative Text

8.17.2.1 Definition DNCE is not redundant with the previous one, DNCT: both conformance and convertibility are defined first for types, and then, as a consequence, for expressions. Roughly, an expression *b* conforms to another *a* if the type of *b* conforms to the type of *a*, and similarly for convertibility. The details appear next for conformance and in the next chapter (in VYEC, page 197) for convertibility.

End

8.17.3 Definition: Expression conformance

Code: *DNEC*

An expression *exp* of type *SOURCE* **conforms to** an expression *ent* of type *TARGET* if and only if they satisfy the following conditions:

1 *SOURCE* conforms to *TARGET*.

2 If *TARGET* is attached or *ent* is emergent, *exp* is an attached expression.

3 If *ent* is an Invariant variable, *SOURCE* and *TARGET* are the same type.

4 If *SOURCE* is expanded, its version of the function *cloned* from *ANY* is available to the current class.

Signed by JV - 2026/10/8

Informative Text

8.17.3.1 So conformance of expressions is more than conformance of their types. Both conditions 2 and 3 are essential. Condition 2 guarantees that execution will never attach a void value to an entity declared of an attached type — a declaration intended precisely to rule out that possibility, so that the entity can be used as target of calls. Condition 3 allows us, in the semantics of attachment, to use a cloning operation when attaching an object with "copy semantics", without causing inconsistencies.

8.17.3.2 *ADD SOMETHING ABOUT THE EMERGENT CASE*

8.17.3.3 A later definition will state what it means for an expression *b* to *convert* to another *a*. These properties also apply to entities (since an entity is an expression).

8.17.3.4 Conformance and convertibility are exclusive of each other, so we study the two mechanisms separately. The rest of the present discussion is devoted to conformance.

End

8.17.4 Validity: Signature conformance

Code: *VNCS*

A signature $t = [B_1, \ldots B_n], [S]$ **conforms to** a signature $s = [A_1, \ldots A_n], [R]$ if and only if it satisfies the following conditions:

1. Each of the two components of *t* has the same number of elements as the corresponding component of *s*.
2. Each type in each of the two components of *t* conforms to the corresponding type in the corresponding component of *s*.
3. Any B_i not identical to the corresponding A_i is detachable.

Informative Text

8.17.4.1 For a signature to conform: the argument types must conform (for a routine); the two signatures must both have a result type or both not have it (meaning they are both queries, or both procedures); and if there are result types, they must conform.

8.17.4.2 Condition 3 adds a particular rule for "covariant redefinition" of arguments as defined next.

End

8.17.5 Definition: Covariant argument

Code: *DNCA*

In a redeclaration of a routine, a formal argument is **covariant** if its type differs from the type of the corresponding argument in at least one of the parents' versions.

Informative Text

8.17.5.1 From the preceding signature conformance rule, the type of a covariant argument will have to be declared as *detachable*: you cannot redefine *f* (*x*: *T*) into *f* (*x*: *U*) even if *U* conforms to *T*; you may, however, redefine it to *f* (*x*: **detachable** *U*). This forces the body of the redefined version, when applying to *x* any feature of *T*, to ensure that the value is indeed attached to an instance of *U* by applying an Object_test, for example in the form

if attached {*U*} *x* **as** *y* **then** *y*.*feature_of_U* **else** … **end**

8.17.5.2 This technique protects the program from *catcalls* — wrongful uses, of a redefined feature, through polymorphism and dynamic binding, to an actual argument of the original, pre-covariant type.

8.17.5.3 The rule only applies to *arguments*, not results, which do not pose a risk of catcall.

Signed by JV - 2026/10/8

8.17.5.4 This rule is the reason why the Feature Declaration rule (VFDE, page 119, condition ?????) requires that if any routine argument is of an anchored type, that type must be detachable, since anchored declaration is a shorthand for explicit covariance.

End

8.17.6 Validity: General conformance

Code: *VNCC*

Let *T* and *V* be two types. *V* **conforms to** *T* if and only if one of the following conditions holds:

1 *V* and *T* are identical.

2 *V* conforms directly to *T*.

3 *V* is *NONE* and *T* is a detachable reference type.

4 *V* is *B* [Y_1,… Y_n] where *B* is a generic class, *T* is *B* [X_1,… X_n], and for every X_i the corresponding Y_i is identical to X_i or, if the corresponding formal generic parameter does not specify **frozen**, conforms (recursively) to X_i.

5 For some type *U* (recursively), *V* conforms to *U* and *U* conforms to *T*.

6 *T* or *V* or both are anchored types appearing in the same class *C*, and the deanchored form of *V* in *C* (recursively) conforms to the deanchored form of *T*.

Informative Text

8.17.6.1 Cases 1 and 2 are immediate: a type conforms to itself, and direct conformance is a case of conformance.

8.17.6.2 Case 3 introduces the class *NONE* describing void values for references. You may assign such a value to a variable of a reference type not declared as attached (as the role of such declarations is precisely to exclude void values); an expanded target is also excluded since it requires an object.

8.17.6.3 Case 4 covers the replacement of one or more generic parameters by conforming ones, keeping the same base class: *B* [*Y*] conforms to *B* [*X*] if *Y* conforms to *X*. (This does not yet address conformance to *B* [*Y1*, … *Yn*] of a type *CT* based on a class *C* different from *B*.) Also note that the **frozen** specification is precisely intended to preclude conformance other than from the given type to itself.

8.17.6.4 Case 5 is indirect conformance through an intermediate type *U*.

8.17.6.5 Finally, case 6 allows us to treat any anchored type, for conformance as for its other properties, as an abbreviation — a “macro” in programmer terminology — for the type of its anchor.

8.17.6.6 Thanks to this definition of conformance in terms of direct conformance, the remainder of the discussion of conformance only needs to define **direct** conformance rules for the various categories of type.

End

8.17.7 Definition: Conformance path

Code: *DNCP*

8.17.7.1 conformance path

A **conformance path** from a type *U* to a type *T* is a sequence of types T_0, T_1, … T_n ($n \geq 1$) such that T_0 is *U*, T_n is *T*, and every T_i (for $0 \leq i < n$) conforms to T_{i+1}. This notion also applies to **classes** by considering the associated base classes.

8.17.8 Validity: Direct conformance: reference types

Code: *VNCN*

A Class_type *CT* of base class *C* **conforms directly** to a reference type *BT* if and only if it satisfies the following conditions:

1 Applying *CT*’s generic substitution to one of the conforming parents of *C* yields *BT*.

2 If *BT* is attached, so is *CT*.

Signed by JV - 2026/10/8

Informative Text

8.17.8.1 The restriction to a reference type in this rule applies only to the target of the conformance, *BT*. The source, *CT*, may be expanded.

8.17.8.2 The basic condition, 1, is inheritance. To handle genericity it applies the "generic substitution" associated with every type: for example with a class *C* [*G*, *H*] inheriting from *D* [*G*], the type *C* [*T*, *U*] has a generic substitution associating *T* to *G* and *U* to *H*. So it conforms to the result of applying that substitution to the Parent *D* [*G*]: the type *D* [*T*].

8.17.8.3 Condition 2 guarantees that we'll never attach a value of a detachable type — possibly void — to a target declared of an attached type; the purpose of such a declaration is to avoid this very case. The other way around, an attached type may conform to a detachable one.

8.17.8.4 This rule is the foundation of the conformance mechanism, relying on the inheritance structure as the condition governing attachments and redeclarations. The other rules cover refinements (involving in particular genericity), iterations of the basic rule (as with "general conformance") and adaptations to special cases (such as expanded types).

End

8.17.9 Validity: Direct conformance: formal generic

Code: *VNCF*

Let *G* be a formal generic parameter of a class *C*, which in the text of *C* may be used as a Formal_generic_name type. Then:

1 No type **conforms directly** to *G*.

2 *G* **conforms directly** to every type listed in its constraint, and to no other type.

8.17.10 Validity: Direct conformance: expanded types

Code: *VNCE*

No type **conforms directly** to an expanded type.

Informative Text

8.17.10.1 From the definition of general conformance, an expanded type *ET* still conforms, of course, to itself. *ET* may also conform to reference types as allowed by the corresponding rule (VNCN); the corresponding assignments will use copy semantics. But no other type (except, per General Conformance, for *e* of type *ET*, the type **like** *e*, an abbreviation for *ET*) conforms to *ET*.

8.17.10.2 This rule might seem to preclude mixed-type operations of the kind widely accepted for basic types, such as *f (3)* where the routine *f* has a formal argument of type *REAL*, or *your_integer_64 := your_integer_16* with a target of type *INTEGER_64* and a source of type *INTEGER_16*. Such attachments, however, involve **conversion** from one type to another. What makes them valid is not conformance but **convertibility**, which does support a broad range of safe mixed-type assignments.

End

8.17.11 Validity: Direct conformance: tuple types

Code: *VNCT*

A Tuple_type *U*, of type sequence *us*, **conforms directly** to a type *T* if and only if *T* satisfies the following conditions:

1 *T* is a tuple type, of type sequence *ts*.

2 The length of *us* is greater than or equal to the length of *ts*.

3 For every element *X* of *ts*, the corresponding element of *us* conforms to *X*.

No type conforms directly to a tuple type except as implied by these conditions.

Informative Text

8.17.11.1 Labels, if present, play no part in the conformance.

End

Signed by JV - 2026/10/8

8.18 Convertibility

Informative Text

8.18.0.1 Complementing the conformance mechanism of the previous discussion, convertibility lets you perform assignment and argument passing in cases where conformance does not hold but you still want the operation to succeed after adapting the source value to the target type.

End

8.18.1 Definition: Conversion procedure, conversion type

Code: *DYPT*

A procedure whose name appears in a Converters clause is a **conversion procedure**.

A type listed in a Converters clause is a **conversion type**.

8.18.2 Definition: Conversion query, conversion feature

Code: *DYQF*

A query whose name appears in a Converters clause is a **conversion query**.

A feature that is either a conversion procedure or a conversion query is a **conversion feature**.

8.18.3 Validity: Conversion principle

Code: *DYCV*

No type may both conform and convert to another.

8.18.4 Validity: Conversion Asymmetry principle

Code: *DYCA*

No type *T* may convert to another through both a conversion procedure and a conversion query.

8.18.5 Validity: Conversion Non-Transitivity principle

Code: *DYNT*

That *V* converts to *U* and *U* to *T* does not imply that *V* converts to *T*.

8.18.6 Syntax: Converter clauses

Code: *SYCV*

Converters ≜ **convert** Converter_list

Converter_list ≜ {Converter","…}+

Converter ≜ Conversion_procedure | Conversion_query

Conversion_procedure ≜ Feature_name "**(**" "**{**" Type_list "**}**" "**)**"

Conversion_query ≜ Feature_name "**:**" "**{**" Type_list "**}**"

8.18.7 Validity: Conversion Procedure rule

Code: *VYCP*

A Conversion_procedure listing a Feature_name *fn* and appearing in a class *C* with current type *CT* is valid if and only if it satisfies the following conditions, applicable to every type *SOURCE* listed in its Type_list:

1 *fn* is the name of a creation procedure *cp* of *C*.

2 If *C* is not generic, *SOURCE* does not conform to *CT*.

3 If *C* is generic, *SOURCE* does not conform to the type obtained from *CT* by replacing every formal generic parameter by its constraining type.

4 *SOURCE*'s base class is different from the base class of any other conversion type listed for a Conversion_procedurein the Converters clause of *C*.

5 The specification of the base class of *SOURCE* does not list a conversion query specifying a type of base class *C*.

6 *cp* has exactly one formal argument, of a type *ARG*.

7 *SOURCE* conforms to *ARG*.

8 *SOURCE* involves no anchored type.

9 *SOURCE* is an attached type.

Signed by JV - 2026/10/8

Informative Text

8.18.7.1 Conditions 2 and 3 (the second one covering generic classes) express the crucial requirement, ensuring the Conversion principle: no type that conforms to the current type may convert to it.

8.18.7.2 In many practical uses of conversion, the target class *CX* is expanded; this is the case with *REAL_64*, and with *REAL_32*, to which *INTEGER* also converts. Such cases satisfy condition 2 almost automatically since essentially no other type conforms to an expanded type. But the validity of a conversion specification does not require the enclosing class to be expanded; all that condition 2 states is that the conversion types must not conform to it (more precisely, to the current type).

End

8.18.8 Validity: Conversion Query rule

Code: *VYCQ*

A Conversion_query listing a Feature_name *fn* and appearing in a class *C* with current type *CT* is valid if and only if it satisfies the following conditions, applicable to every type *TARGET* listed in its Type_list:

1 *fn* is the name of a query *f* of *C*.

2 If *C* is not generic, *CT* does not conform to *TARGET*.

3 If *C* is generic, the type obtained from *CT* by replacing every formal generic parameter by its constraining type does not conform to *TARGET*.

4 *TARGET*'s base class is different from the base class of any other conversion type listed for a Conversion_query in the Converters clause of *C*.

5 The specification of the base class of *TARGET* does not list a conversion procedure specifying a type of base class *C*.

6 *f* has no formal argument.

7 The result type of *f* conforms to *TARGET*.

8 *TARGET* involves no anchored type.

9 *TARGET* is an attached type.

Informative Text

8.18.8.1 Condition 5 is redundant with condition 5 of the Conversion Procedure rule but is included anyway for symmetry. In case of violation, a compiler may refer to either rule.

End

8.18.9 Definition: Converting to a class

Code: *DYCL*

A type *T* of base class *CT* **converts to** a class *C* if either:

- The deanchored form of *T* appears as conversion type for a procedure in the Converters clause of *C*.
- A type based on *C* appears as conversion type for a query in the Converters clause of *CT*.

8.18.10 Definition: Converting to and from a type

Code: *DYCT*

A type *U* of base class *D* **converts to** a Class_type *T* of base class *C* if and only if either:

1 The deanchored form of *U* is the result of applying the generic substitution of the deanchored form of *T* to a conversion type for a procedure *cp* appearing in the Converters clause of *C*.

2 The deanchored form of *T* is the result of applying the generic substitution of the deanchored form of *U* to a conversion type for a query *cq* appearing in the Converters clause of *D*.

Signed by JV - 2026/10/8

A Class_type *T* **converts from** a type *U* if and only if *U* converts to *T*.

8.18.11 Definition: Converting "through"

Code: *DYTH*

A type *U* that converts to a type *T*:

1 Converts to *T* **through a procedure** *cp* if case 1 of the definition of "converting to a type" applies.

2 Converts to *T* **through a query** *cq* if case 2 of the definition applies.

These terms also apply to "**converting from**" specifications.

Informative Text

8.18.11.1 From the definitions and validity rules, it's clear that if *U* converts to *T* then it's either — but not both — "through a procedure" or "through a query", and that exactly one routine, *cp* or *f*, meets the criteria in each case.

End

8.18.12 Semantics: Conversion semantics

Code: *MYCV*

Given an expression *e* of type *U* and a variable *x* of type *T*, where *U* converts to *T*, the effect of a **conversion attachment** of source *e* and target *x* is to attach to *x* the value of the following expression:

1 If *U* converts to *T* through a procedure *cp*: **create** *{U} cp (e)*.

2 If *U* converts to *T* through a query *cq*: *e*.*cq*.

Informative Text

8.18.12.1 This is an "unfolded form" specification expressing the semantics of an operation (conversion attachment) in terms of another: either a creation or a query call. Both of these operations involve an attachment (argument passing or assignment) and so may trigger one other conversion.

End

8.18.13 Definition: Explicit conversion

Code: *DYEX*

The Kernel Library class *TYPE* [*G*] provides a function

adapted **alias** "[]" (*x: G*): *G*

which can be used for any type *T* and any expression *exp* of a type *U* compatible with *T* to produce a *T* version of *exp*, written

{*T*} [*exp*]

If *U* converts to *T*, this expression denotes the result of converting *exp* to *T*, and is called an **explicit conversion**.

Informative Text

8.18.13.1 Explicit conversion involves no new language mechanism, simply a feature of a Kernel Library class and the notion of bracket alias.

8.18.13.2 For example, assuming a tuple type that converts to *DATE*, you may use

{*DATE*} [[20, "*April*", 2005]]

The Expression [20, "*April*", 2005] is a Manifest_tuple. Giving it — through the outermost brackets — as argument to the *adapted* function of {*DATE*} turns it into an expression of type *DATE*. This is permitted for example if class *DATE* specifies a conversion procedure from *TUPLE* [*INTEGER, STRING, INTEGER*].

End

Signed by JV - 2026/10/8

8.18.14 Validity: Expression convertibility

Code: *VYEC*

An expression *exp* of type *U* **converts to** an entity *ent* of type *T* if and only if *U* converts to *T* through a conversion feature *conv* satisfying either of the following two conditions:

1 *conv* is precondition-free.

2 *exp* statically satisfies the precondition.

8.18.15 Definition: Statically satisfied precondition

Code: *DYSS*

A feature precondition is **statically satisfied** if it satisfies any of the following conditions:

1 It applies to a boolean, character, integer or real expression involving only constants, states that the expression equals a specific constant value or (in the last three cases) belongs to a specified interval, and holds for that value or interval.

2 It applies to the type of an expression, states that it must be one of a specified set of types, and holds for that type.

Informative Text

8.18.15.1 The "constants" of the expression can be manifest constants, or they can be constant actual arguments to a routine — possibly the unfolded form of an assignment, as in *of_type_NATURAL_8* := 1, whose semantics is that of **create** *of_type_natural*.*from_INTEGER* (1). Without the notion of "statically satisfied precondition" such instructions would be invalid because *from_INTEGER* in class *NATURAL_8* has a precondition (not every integer is representable as a *NATURAL_8*), and arbitrary preconditions are not permitted for conversion features. This would condemn us to the tedium of writing {*NATURAL_8*} 1 and the like for every such case, and would be regrettable since 1 *is* as a matter of fact acceptable as a *NATURAL_8*. So the definition of expression convertibility permits a "statically satisfied" precondition, making such cases valid.

8.18.15.2 It would be possible to generalize the definition by making permissible any precondition that can be assessed statically. But this would leave too much initiative to individual compilers: a "smarter" compiler might accept a precondition that another rejects, leading to incompatibilities. It was judged preferable to limit the rule to the two cases known to be important in practice; if others appear in the future, the rule will be extended.

End

8.18.16 Validity: Precondition-free routine

Code: *VYPF*

A feature *r* of a class *C* is **precondition-free** if it is either:

1 Immediate in *C*, with either no Precondition clause or one consisting of a single Assertion_clause (introduced by **require**) whose Boolean_expression is the constant *True*.

2 Inherited, and such that every precursor of *r* is (recursively) precondition-free, or *r* is redeclared in *C* with a Precondition consisting of a single Assertion_clause (introduced by **require else**) whose Boolean_expression is the constant **True**.

Informative Text

8.18.16.1 A feature is "immediate" if it is declared in the class itself. In the other case, "inherited" feature, it's OK if the feature had a precondition in the parent, but then the class must redeclare it with a clause **require else** *True*. A simple **require** without the **else** is not permitted in this case.

8.18.16.2 A "precursor" of an inherited routine is its version in a parent; there may be more than one as a result of feature merging and repeated inheritance.

End

PART H: Computational mechanisms

Signed by JV - 2026/10/8

Signed by JV - 2026/10/8

8.19 Feature call

Informative Text

8.19.0.1 In Eiffel's model of computation, the fundamental way to do something with an object is to apply to it an operation which — because the model is class-based, and behind every run-time object lurks some class of the system's text — must be a feature of the appropriate class.

8.19.0.2 This is feature call, central to the object-oriented style of programming. The present chapter reviews its three forms (including one that does not need a target object, only a target type) and their properties.

End

8.19.1 Validity: Call Use rule

Code: *VUCN*

A Call of feature *f* denotes:

1 If *f* is a query (attribute or function): an expression.

2 If *f* is a procedure: an instruction.

8.19.2 Syntax: Feature calls

Code: *SUCA*

```
Call ≜ Object_call | Non_object_call
Object_call ≜ [Target "." ] Unqualified_call
Unqualified_call ≜ Feature_name [Actuals]
Target ≜ Local |
         Read_only |
         Call |
         Parenthesized |
         Bracket_expression |
         Paren_expression |
         Precursor_call
Non_object_call ≜ "{" Type "}" "." Unqualified_call
```

Informative Text

8.19.2.1 The three possible forms of a call are:

8.19.2.1.1 • The most powerful form, an Object_call with an explicit Target, such as

```
x.f (a)                    [1]
x.a.b.c                    [2]
x.u (v, w).y (z)           [3]
(a (b, c) + x ).d          [4]
```

Such a call applies to an object, denoted (in all three examples) by *x*.

8.19.2.1.2 • The simplest form, an Unqualified_call, with arguments as in *u* (*v*, *w*) (with arguments) or just *a* (without arguments), where *u* and *a* are features. Such a call applies to the current object; the effect is (other than effect on class invariant checking) the same as that of an Object_call whose target would be **Current**.

8.19.2.1.3 • Finally, a Non_object_call, which does not apply to any specific object but simply calls a feature obtained from a specified Type — more precisely, its base class.

Signed by JV - 2026/10/8

8.19.2.2 In the first case, Object_call, Target allows an Object_call to apply to an explicit target object (rather than the current object). The target can itself be a Call, allowing multidot calls, as in example [3] where the target is *x*.*u* (*v*, *w*). Other possible targets are a local variable (which *x* could be in example [1]), a Read_only (including formal arguments and **Current**) a Non_object_call, or a complex expression written as a Parenthesized (…) as in example [4] where the target is (*a* (*b*, *c*) + *x*).

End

Informative Text

8.19.2.3 The following two definitions introduce two properties deduced from the syntax: qualified and targeted. The two are distinct:

- An Object_call with a Target, such as *x*.*f* (*a*), is qualified and targeted.
- An Unqualified_call is targeted (the target is the current object) and not qualified.
- A Non_object_call {*T*}.*f* (*a*) (for some type *T*) is not targeted (that is the basic idea: that it does not need an object), but qualified (by a type).

End

8.19.3 Definition: Qualified, unqualified call

Code: *SUQU*

A Call is **unqualified** if it is an Unqualified_call and **unqualified** otherwise.

Informative Text

8.19.3.1 In equivalent terms, a call is qualified if and only if it either an Object_call with a Target or a Non_object_call. So:

- *f* (*a*) is unqualified (as one would expect since it is syntatically a specimen of Unqualified_call),
- *x*.*f* (a) , an Object_call, is qualified.
- So is {*T*}.*f* (*a*), a Non_object_call. It is qualified not by a target expression (such as *x* in the previous case) but by a type *T*.

End

8.19.4 Definition: Targeted, untargeted call

Code: *SUTU*

A Call is **targeted** if it is an Object_call and **untargeted** otherwise.

Informative Text

8.19.4.1 “Targeted call” is simply a synonym for a call that is syntactically an Object_call, that is to say, not a Non_object_call.

8.19.4.2 The purpose of this definition is to simplify the terminology in introducing the concept of a “target of a call” next: the target is defined if the call is “targeted”.

End

8.19.5 Definition: Target of a call

Code: *SUTA*

A targeted call has a **target**, defined as follows:

1 If it is qualified: its Target component.

2 If it is unqualified: **Current**.

Informative Text

8.19.5.1 Per the preceding definition (SUTU), a targeted call is an Object_call, with or without a Target component. But unlike a Non_object_call it always has a target; if there is no explicit Target component, the target denotes the current object.

Signed by JV - 2026/10/8

8.19.5.2 Syntactically the target is an expression:

- In *a* (*b*, *c*)**.***d* the target is *a* (*b*, *c*).
- In (*a* (*b*, *c*) + *x*)**.***d* the target is *a* (*b*, *c*) + *x*.
- In a multidot case the target includes the Call deprived of its last part. For example in
  *x***.***f* (*args*)**.***g***.***h* (*args1*)
  the target is
  *x***.***f* (*args*)**.***g*

8.19.5.3 {*T*}**.***f* (*a*) is a Non_object_call, hence untargeted (from SUTU above), and does not have a target.

8.19.5.4 Terminology: the term "target" is also used for reattachments, meaning assignments and argument passing (DBRE, page 239), and for creation instructions (DKTT, page 234).

End

8.19.6 Syntax: Actual arguments

Code: *SUAA*

Actuals ≙ "(" Actual_list ")"

Actual_list ≙ {Expression "," …}$^{+}$

Informative Text

8.19.6.1 The elements of the Actual_list are the "actual arguments" of the call per DRAA, page 121.

End

8.19.7 Definition: Operands of a call

Code: *DUOP*

The **operands** of a call include its target if the call is targeted, and its arguments if any.

Informative Text

8.19.7.1 The notion of operand also exists for agents (DPOP, page 213) and operator expressions (DWOP, page 217).

End

8.19.8 Definition: Operand position

Code: *DUPO*

The **position** of an operand of a Call is 0 if it is the Call's target, and *i* if it is the Call's *i*-th actual argument, for any applicable *i*.

Informative Text

8.19.8.1 Positions of arguments start from 1.

8.19.8.2 The operand at position 0 is the target, and is defined only for a targeted call (Object_call), not for a Non_object_call.

End

8.19.9 Definition: Target type of a call

Code: *DUTT*

Any Call has a **target type**, defined as follows:

1 For an Unqualified_call: the current type.

2 For an Object_call: the type of its target if it is not a Tuple_type; if if is a Tuple_type, the type based on its unfolded form.

3 For a Non_object_call having a type *T* as its Type part: *T*.

Informative Text

8.19.9.1 There is a corresponding definition for the target type of *agents*: DPTT, page 212. See also the *creation type* of a creation instruction: DKTT, page 234.

End

8.19.10 Definition: Feature name of a call

Code: *SUFN*

For any Call the **feature name of the call** is defined as follows:

1 For an Unqualified_call: its Feature_name.

2 For a qualified call or Non_object_call: (recursively) the feature name of its Unqualified_call part.

Informative Text

8.19.10.1 Case 1 tells us that the feature name of *f* (*args*) is *f* and the feature of *g*, an Unqualified_call to a feature without arguments, is *g*.

8.19.10.2 It follows from case 2 that the feature name of a qualified call *x*.*f* (*args*) is *f*. The recursive phrasing addresses the multidot case: the feature name of *x*.*f* (*args*).*g*.*h* (*args1*) is *h*.

8.19.10.3 Also covered by case 2 is a call of the form *t*.*b* where *t* is a labeled tuple and *b* is one of the labels: using the "unfolded form" of the Tuple_type makes it possible to consider this case as if it were just a call to an attribute of a class.

End

8.19.11 Validity: Export Status principle

Code: *VUES*

The export status of a feature *f*:

- Constrains all qualified calls *x*.*f* (…), including those in which the type of *x* is the current type, or is **Current** itself.
- Does not constrain unqualified calls.

Informative Text

8.19.11.1 This is a validity property, but with no new information; it simply restates, for emphasis, condition 2 of the Export rule (VUEX).

End

8.19.12 Definition: Static feature of a call

Code: *DUSF*

The **static feature** of an export-valid Call of feature name *fname* is the feature of name *fname* in the call's target type.

Informative Text

8.19.12.1 The Export Rule, VUEX (1), tells us that *fn*, in a call *x*.*fn* (…), is the name of a feature *f* of the type of the target *x*. Then *f* is the static feature of the call. Because of dynamic binding, an execution of the call will not necessarily use *f* but the *dynamic* feature of the call (see DUDF, page 206 below), which is a *version* of *f*: either *f* itself or a redeclaration in a descendant.

End

8.19.13 Definition: Feature of a call

Code: *DUSF*

The **feature of a call** is its static feature.

Signed by JV - 2026/10/8

Informative Text

8.19.13.1 Along with a dynamic feature (defined in the program text) a call has, during one of its execution, a dynamic feature as well. It is better to use the full terminology (static feature, dynamic feature), but for discussions that clearly involve the program text only, not its execution, so that no ambiguity arises, this definition allows the Standard to use occasionally, for conciseness, "feature" as an abbreviation for "static feature".

End

8.19.14 Validity: Argument rule

Code: *VUAR*

An export-valid Call of target type *ST* and feature *sf* is argument-valid if and only if its tuple-argument-unfolded form satisfies the following conditions:

1 The number of actual arguments is the same as the number of formal arguments declared for *sf*.

2 Every actual argument of the call is compatible with the corresponding formal argument of *sf*.

3 If compatibility in clause 2 does not involve conversion, the call is conforming-argument-valid.

Informative Text

8.19.14.1 Condition 2 is the fundamental type rule on argument passing, which allowed the discussion of direct reattachment to treat Assignment and actual-formal association in the same way. An expression is *compatible* (DNCE, page 189) with an entity if its type either conforms or converts to the entity's type.

8.19.14.2 The "tuple-argument-unfolded" form takes into account the possible dropping of brackets around a tuple argument; see DJTA, page 185.

End

8.19.15 Validity: Target rule

Code: *VUTA*

An Object_call is **target-valid** if and only if it satisfies one of the following conditions:

1 It is an Non_object_call.

2 It is an Object_call without a Target.

3 It is an Object_call with a Target which is an attached expression.

Informative Text

8.19.15.1 Non-object and unqualified calls (cases 1 and 2) are always target-valid since they are applied to the current object, which by construction is not void.

8.19.15.2 For the target expression *x* to be "*attached*", in case 2, means that the program text guarantees — statically, that is to say through rules enforced by FIX THIS compileFeature Value ruleFeature Value rulers — that *x* will never be void at run time. This may be because *x* is an entity declared as attached (so that the validity rules ensure it can never be attached a void value) or because the context of the call precludes voidness, as in if *x* /= *Void* **then** *x*.*f* (…) **end** for a local variable *x*. The precise definition will cover all these cases.

End

Informative Text

8.19.15.3 The preceding specifications take care of the most important form of call, Object_call. The next few ones deal with the case of a Non_object_call, which does not use a target object but directly calls a feature from a class, as in {*MATH_CLASS*} *square_root* (*x*).

End

8.19.16 Definition: Imported form of a Non_object_call

Code: *SUIF*

The **imported form** of a Non_object_call of Type *T* and feature *f* appearing in a class *C* is the Unqualified_call built from the original Actuals if any and, as static feature of the call, a fictitious new feature added to *C* and consisting of the following elements:

1 A name different from those of other features of *C* .

2 A obtained from the of *f* by replacing every type by its deanchored form, then applying the generic substitution of *T*.

Informative Text

8.19.16.1 This definition in “unfolded” style allows us to view {*T*}.*f* (*args*) appearing in a class *C* as if it were just *f* (*args*), an Unqualified_call, but appearing in *C* itself, assuming we had moved *f* over — “imported” it — to *C*.

8.19.16.2 In item 2 we use the “deanchored form” of the argument types and result, since a type **like** *a* that makes sense in *T* would be meaningless in *C*. As defined in the discussion of anchored types, the deanchored version precisely removes all such local dependencies, making the type understandable instead in any other context.

End

8.19.17 Validity: Non-Object Call rule

Code: *VUNO*

A Non_object_call of Type *T* and feature *fname* in a class *C* is valid if and only if it satisfies the following conditions:

1 *fname* is the final name of a feature *f* of *T*.

2 *f* is available to *C*.

3 *f* is either a constant attribute or an external feature whose assertions, if any, use neither **Current** nor any unqualified calls.

4 The call’s imported form is a valid Unqualified_call.

Informative Text

8.19.17.1 Condition 2 requires *f* to have a sufficient export status for use in *C*; there will be a similar requirement for Object_call. Condition 3 is the restriction to constants and externals. Condition 4 takes care of the rest by relying on the rules for Unqualified_call.

End

8.19.18 Semantics: Non-Object Call semantics

Code: *MUNC*

The effect of a Non_object_call is that of its imported form.

Informative Text

8.19.18.1 *Need explanation here*

End

8.19.19 Validity: Export rule

Code: *VUEX*

A Call appearing in a class *C*, with *fname* as the feature name of the call, is **export-valid** for *C* if and only if it satisfies the following conditions.

1 *fname* is either the final name of a feature of the target type of the call or an entity whose type includes a routine with a parenthesized alias.

2 If the Object_call is qualified, that feature is available to *C*.

Informative Text

8.19.19.1 For an unqualified call *f* or *f (args)*, only condition 1 is applicable, requiring simply (since the target type of an unqualified call is the current type) that *f* be a feature, immediate or inherited, of the current class.

8.19.19.2 For a qualified call *x.f* with *x* of type *T,* possibly with arguments, condition 2 requires that the base class of *T* make the feature available to *C*: export it either generally or selectively to *C* or one of its ancestors. (Through the Non-Object Call rule this also governs the validity of a Non_object_call *{T}.f*.)

8.19.19.3 As a consequence, *s* (…) might be permitted and *x.s* (…) invalid, even if *x* is **Current**. The semantics of qualified and unqualified calls is indeed slightly different; in particular, with invariant monitoring on, a qualified call will — even with **Current** as its target — check the class invariant, but an unqualified call won't.

8.19.19.4 The Export rule also determines the validity of expressions denoting components of labeled tuples: thanks to the notion of unfolded form of a Tuple_type, the expression *t.b*, where the type of *t* is a labeled tuple type *TT*, is export-valid if and only if *b* is one of the labels of *TT*.

End

8.19.20 Validity: Class-Level Call rule

Code: *VUCC*

A Call of target type *ST* is **class-valid** if and only if it is export-valid, argument-valid and target-valid.

8.19.21 Definition: Target object

Code: *DUTO*

The **target object** of an execution of an Object_call is:

1 If the call is qualified: the object attached to its target.

2 If it is unqualified: the current object.

Informative Text

8.19.21.1 In the case of labeled tuple access, *t.b* where the type of *t* is a Tuple_type, the target object is an instance of a fictitious class as defined by the unfolded form.

End

8.19.22 Definition: Dynamic feature of a call

Code: *DUDF*

Consider an execution of a call of dynamic feature *fname* and target object O. Let *DT* be the type of O. The **dynamic feature** of the call is the dynamic binding version in *DT* of *f*.

Informative Text

8.19.22.1 Behind the soundness of this definition stands a significant part of the validity machinery of the language:

8.19.22.1.1 • The rules on reattachment imply that *DT* conforms to *ST*.

8.19.22.1.2 • The Export rule imply that *fname* is the name of a feature of *ST* (meaning, per DTFT, page 167, a feature of the base class of *ST*).

Signed by JV - 2026/10/8

8.19.22.1.3 • As a consequence, this feature has a version in *DT*; it might have several, but the definition of "dynamic binding version" removes any ambiguity.

8.19.22.2 Combining the last two semantic definitions enables the rest of the semantic discussion to take for granted, for any execution of a qualified call, that we know both the target object and the feature to execute. In other words, we have taken care of the two key parts of Object_call semantics, although we still have to integrate a few details and special cases.

End

8.19.23 Principle: Feature Version theorem

Code: *PUFV*

The dynamic feature of a callis a version of its dynamic feature.

Informative Text

8.19.23.1 This semantic property follows from the definition of "dynamic feature". A "version" of a feature (the precise definition is in DMVE, page 160) is either the feature itself or its redeclaration in a proper descendant of its original class. The Feature Version theorem captures a fundamental property of call semantics, characterized by dynamic binding: when you write *x*.*f* (…), the execution will use either *f* itself or one of its descendant redeclarations.

End

8.19.24 Definition: Void-Unsafe

Code: *DUVU*

A language processing tool may, as a temporary migration facility, provide an option that waives the target validity requirement in class validity. Systems processed under such an option are **void-unsafe**.

Informative Text

8.19.24.1 Void-unsafe systems are not valid Eiffel systems. Since void safety was not enforced by previous versions of Eiffel, compilers may need, all the same, to provide an option that temporarily lifts this requirement. Including the notion of "void-unsafe" in the language definition enforces a consistent way for various compilers to provide this transition facility.

End

8.19.25 Semantics: Failed target evaluation of a void-unsafe system

Code: *MUVU*

In the execution of an (invalid) system compiled in void-unsafe mode through a language processing tool offering such a migration option, an attempt to execute a call triggers, if it evaluates the target to a void reference, an exception of type *VOID_TARGET*.

8.19.26 Definition: Freshness of a once routine call

Code: *DUFR*

During execution, a call whose feature is a once routine *r* is **fresh** if and only if every feature call started so far satisfies any of the following conditions:

1 It did not use *r* as dynamic feature.

2 It was in a different thread, and *r* is once per thread or has no once key.

3 Its target object was not the current object, and *r* is a once per object routine.

4 After it was started, a call was executed to one of the refreshing features ******* CHECK ******of *onces* from *ANY*, including among the keys to be refreshed at least one of the once keys of *r*.

Informative Text

8.19.26.1 Case 2 indicates that "once per thread" is the default in the absence of an explicit once key.

8.19.26.2 Note that there is another, unrelated notion of “freshness”: the concept of a fresh identifier.

End

8.19.27 Definition: Latest applicable target and result of a non-fresh call

Code: *DULA*

The **latest applicable target** of a non-fresh call to a once routine *df* to a target object O is the last value to which it was attached in the call to *df* most recently started on:

1 If *df* is once per object: O.

2 Otherwise, if *df* is once per thread or has no once key: any target in the current thread.

3 Otherwise: any target in any thread.

If *df* is a function, the **latest applicable result** of the call is the last value of **Result** computed by the most recent fresh call.

8.19.28 Semantics: Once Routine Execution semantics

Code: *MUON*

The effect of executing a once routine *df* on a target object O is:

1 If the call is fresh: that of a non-once call made of the same elements, as determined by Non-once Routine Execution Semantics.

2 If the call is not fresh and the last execution of *f* on the latest applicable target triggered an exception: to trigger again an identical exception. The remaining cases do not then apply.

3 If the call is not fresh and *df* is a procedure: no further effect.

4 If the call is not fresh and *df* is a function: to attach the local variable **Result** to the latest applicable result of the call.

Informative Text

8.19.28.1 Case 2 is known as “*once a once exception, always a once exception*”. If a call to a once routine yields an exception, then all subsequent calls for the same applicable target, which would normally yield no further effect (for a procedure, case 3) or return the same value (for a function, case 4) should follow the same general idea and, by re-triggering the exception, repeatedly tell the client — if the client is repeatedly asking — that the requested effect or value is impossible to provide.

8.19.28.2 There is a little subtlety in the definition of “latest applicable target” as used in case 4. For a once function that has already been evaluated (is not fresh), the specification does not state that subsequent calls return the result of the first, but that they yield the value of the predefined local variable **Result**. Usually this is the same, since the first call returned its value through **Result**. But if the function is **recursive**, a new call may start before the first one has terminated, so the “result of the first call” would not be a meaningful notion. The specification states that in this case the recursive call will return whatever value the first call has obtained so far for **Result** (starting with the default initialization). A recursive once function is a bit bizarre, and of little apparent use, but no validity constraint disallows it, and the semantics must cover all valid cases.

End

8.19.29 Semantics: Current object, current routine

Code: *MUCD*

At any time during the execution of a system there is a **current object** CO and a **current routine** *cr* defined as follows:

1 At the start of the execution: CO is the root object and *cr* is the root procedure.

2 If *cr* executes a qualified call: the call’s target object becomes the new current object, and its dynamic feature becomes the new current routine. When the qualified call terminates, the earlier current object and routine resume their roles.

3 If *cr* executes an unqualified call: the current object remains the same, and the dynamic feature of the call becomes the current routine for the duration of the call as in case 2.

Signed by JV - 2026/10/8

4 If *cr* starts executing any construct whose semantics does not involve a call: the current object and current routine remain the same.

8.19.30 Semantics: Current semantics

Code: *MUCU*

The value of the predefined entity **Current** at any time during execution is the current object if the current routine belongs to an expanded class, and a reference to the current object otherwise.

8.19.31 Semantics: Non-Once Routine Execution Semantics

Code: *MUNO*

The effect of executing a non-once routine *df* on a target object O is the effect of the following sequence of steps:

1 If *df* has any local variables, including **Result** if *df* is a function and **Retry** in a Rescue clause, save their current values if any call to *df* has been started but not yet terminated.

2 Execute the body of *df*.

3 If the values of any local variables have been saved in step 1, restore the variables to their earlier values.

8.19.32 Semantics: General Call semantics

Code: *MUGC*

The effect of an Object_call of feature *sf* is, in the absence of any exception, the effect of the following sequence of steps:

1 Determine the target object O through the applicable definition.

2 Attach **Current** to O.

3 Determine the dynamic feature *df* of the call through the applicable definition.

4 Replace the Actuals part, if any, by its tuple-argument-unfolded form.

5 For every actual argument *a*, if any, in the order listed: obtain the value *v* of *a*; then if the type of *a* converts to the type of the corresponding formal in *sf*, replace *v* by the result of the applicable conversion. Let *arg_values* be the resulting sequence of all such *v*.

6 Attach every formal argument of *df* to the corresponding element of *arg_values* by applying the Reattachment Semantics rule.

7 If the call is targeted and class invariant monitoring is on, evaluate the class invariant of O's base type on O.

8 If precondition monitoring is applicable per the Assertion monitoring rule, evaluate the precondition of *df*.

9 If *df* is not an attribute, not a once routine and not external, apply Non-Once Routine Execution Semantics to O and *df*.

10 If *df* is a once routine, apply the Once Routine Execution Semantics to O and *df*.

11 If *df* is an external routine, execute that routine on the actual arguments given, if any, according to the rules of the language in which it is written.

12 If *df* is a self-initializing attribute and has not yet been initialized, initialize it through the Default Initialization rule.

13 If invariant monitoring is applicable per the Assertion monitoring rule, evaluate the class invariant of O's base type on O.

14 If postcondition monitoring is applicable per the Assertion monitoring rule, evaluate the postcondtion of *df*.

An exception occurring during any of these steps causes the execution to skip the remaining parts of this process and instead handle the exception according to the Exception Semantics rule.

8.19.33 Definition: Expression-call type

Code: *DUCT*

Consider a call denoting an expression. Its **expression-call type** with respect to a type *CT* of base class *C* is:

1 For an unqualified call, its feature *f* being a query of *CT*: the result type of the version of *f* in *C*, adapted through the generic substitution of *CT*.

2 For a qualified call *a.e* of Target *a*: (recursively) the type of *e* with respect to the type of *a*.

3 For a Non_object_call: (recursively) the type of its imported form.

8.19.34 Semantics: Call result

Code: *MURE*

Consider a Call *c* whose feature is a query. An execution of *c* according to the General Call Semantics yields a **call result** defined as follows, where *O* is the target object determined at step 1 of the rule and *df* the dynamic feature determined at step 3:

1 If *df* is a non-external, non-once function: the value attached to the local variable **Result** of *df* at the end of step 2 of Non-Once Routine Execution Semantics.

2 If *df* is a once function: the value attached to **Result** as a result of the application of Once Routine Execution Semantics.

3 If *df* is an attribute: the corresponding field in *O*.

4 If *df* is an external function: the result returned by the function according to the external language's rule.

Informative Text

8.19.34.1 A labeled tuple access, *t.b* where the type of *t* is a Tuple_type, falls under case 3 as a result of the notion of unfolded form of tuple types.

End

8.19.35 Semantics: Value of a call expression

Code: *MUVA*

The **value** of a Call *c* used as an expression is, at any run-time moment, the result of executing *c*.

8.20 Agents, iterators and introspection

Informative Text

8.20.0.1 Objects represent information equipped with operations. These are clearly defined concepts; no one would mistake an operation for an object.

8.20.0.2 For some applications — graphics, numerical computation, iteration, writing contracts, building development environments, "*reflection*" (a system's ability to explore its own properties) — you may find the operations so interesting that you will want to define objects to represent them, and pass these objects around to software elements, which can use these objects to execute the operations whenever they want. Because this separates the place of an operation's *definition* from the place of its *execution*, the definition can be incomplete, since you can provide any missing details at the time of any particular execution.

8.20.0.3 You can create **agent** objects to describe such partially or completely specified computations. Agents combine the power of higher-level functionals — operations acting on other operations — with the safety of Eiffel's static typing system.

End

8.20.1 Definition: Construction time, call time

Code: *DPCC*

The **construction time** of an agent object is the time of evaluation of the agent expression defining it.

Its **call time** is when a call to its associated operation is executed.

8.20.2 Syntax: Agents

Code: *SPAG*

Agent ≜ Call_agent | Inline_agent

Call_agent ≜ **agent** Call_agent_body

Inline_agent ≜ **agent**
[Formal_arguments]
[Result_type]
[Feature_definition]
[Agent_actuals]

Call_agent_body ≜ Agent_qualified | Agent_unqualified

Agent_qualified ≜ Agent_target "**.** " Agent_unqualified

Agent_unqualified ≜ Feature_name [Agent_actuals]

Agent_target ≜ Entity | Parenthesized | Manifest_type

Agent_actuals ≜ "**(**" Agent_actual_list "**)**"

Agent_actual_list ≜ {Agent_actual "**,**" …}$^{+}$

Agent_actual ≜ Expression | Placeholder

Placeholder ≜ [Manifest_type] "**?**"

Informative Text

8.20.2.1 The next specification gives a more informal view of these syntax definitions.

End

Signed by JV - 2026/10/8

8.20.3 Syntax (non-production): syntactical forms for a call agent

Code: *SPFO*

A call agent is of the form

agent *agent_body*

where *agent_body* is a Call, qualified (as in *x.r* (…)) or unqualified (as in *f* (…)) with the following possible variants:

- Replace any argument by a question mark **?**, making the argument open.
- Replace the target, by {*TYPE*} where *TYPE* is the name of a type, making the target open.
- Remove the argument list (…) altogether, making all arguments open.

Informative Text

8.20.3.1 This list of possible forms is not a formal syntax definition, but a summary of the available forms permitted by the preceding syntax (SPAG, page 211) and the following more formal definitions and validity rules.

End

8.20.4 Definition: Target type of a call agent

Code: *DPTT*

The **target type** of a Call_agent is:

1 If there is no Agent_target, the current type.

2 If there is an Agent_target and it is an Entity or Parenthesized, its type.

3 If there is an Agent_target and it is a Manifest_type, the type that it lists (in braces).

Informative Text

8.20.4.1 See the corresponding definition for calls: DUTT, page 202. (Also the *creation type* of a creation instruction: DKTT, page 234.)

End

8.20.5 Validity: Call Agent rule

Code: *VPCA*

A Call_agent involving a Feature_name *fn*, appearing in a class *C*, with target type *T0*, is valid if and only if it satisfies the following conditions:

1 *fn* is the name of a feature *f* of *T0*.

2 If there is an Agent_target, *f* is export-valid for *T0* in *C*.

3 If the Agent_actual part is present, the number of elements in its Agent_actual_list is equal to the number of formals of *f*.

4 Any Agent_actual of the Expression kind is of a type compatible_with the type of the corresponding formal in *f*.

8.20.6 Definition: Associated feature of an inline agent

Code: *DPAF*

Every inline agent *ia* of a class *C* has an **associated feature**, defined as a fictitious routine *f* of *C*, such that:

1 The name of *f* is chosen not to conflict with any other feature name in *C* and its descendants.

2 The formal arguments of *f* are those of *ia*.

3 *f* is secret (available for call to no class).

4 Its Result_type is the Result_type of *ia* if pressent; otherwise it has no Result_type and is a command.----- THIS TO BE CHECKED, see point 6 below---

5 The Feature_definition part of *f* is defined by the Feature_definition part of *ia*.

6 *f* is a function if *ia* has a Result_type (its result type being given by the Type in that Result_type), a procedure otherwise.

8.20.7 Validity: Inline Agent rule

Code: *VPIA*

An Inline_agent *a* of associated feature *f*, is valid in the text of a class *C* if and only if it satisfies the following conditions:

1 *f*, if added to *C*, would be valid.

2 *f* is not deferred.

8.20.8 Validity: Inline Agent requirements

Code: *VPIR*

An Inline_agent *a* must satisfy the following conditions:

1 Its formal arguments and local variables, if any, are fresh.

2 Every entity appearing in the Feature_definition part of *a* is the name of one of: a formal argument of *a*; a local variable of *a*; a feature of the enclosing class; **Current**.

3 The Feature_body of *a*'s Feature_definition is not of the Deferred form.

Informative Text

8.20.8.1 These conditions are stated as another validity rule permitting compilers to issue more understandable error messages. It is not in the usual "if and only if" form (since the preceding rule, the more official one, takes care of this), but the requirements given cover the most obvious possible errors.

8.20.8.2 In condition 1, an identifier is fresh (D7FI, page 72) if it is not already used in the enclosing context; here the rule means that the names of an inline agent's formals and locals may not have the same name as a feature of the class, or a formal or local of an enclosing routine or inline agent.

8.20.8.3 A consequence of the rules on inline agents is that it is not possible in an inline agent to refer to the local variables of the enclosing routine or inline agent; this property is enforced on purpose, for simplicity, as any other policy would require special disambiguation techniques (think for example of referring to the **Result** of both the inline agent and its enclosing routine). The simple solution, to use the context's local variables, is to pass them as arguments to the inline agent.

End

8.20.9 Definition: Call-agent equivalent of an inline agent

Code: *DPCE*

The **call-agent equivalent** of an inline agent *ia* is

agent *f*

where *f* is the associated feature of *ia*.

8.20.10 Semantics: Semantics of inline agents

Code: *MPIA*

The semantic properties of an inline agent are those of its call-agent equivalent.

8.20.11 Semantics: Use of Result in an inline function agent

Code: *MPRE*

In an agent of the Inline_agent form denoting a function, the local variable **Result** denotes the result of the agent itself.

8.20.12 Definition: Operands of an agent

Code: *DPOP*

The **operands** of an agent include its Agent_target if any and its arguments if any.

Signed by JV - 2026/10/8

Informative Text

8.20.12.1 The notion of operand also exists for calls (DUOP, page 202) and operator expressions (DWOP, page 217).

End

8.20.13 Definition: Open and closed operands of an agent

Code: *DPOC*

The **open operands** of a Call_agent include:

1 Any Agent_actual that is a Placeholder.

2 The Agent_target if it is present and is a Manifest_type.

The **closed operands** include all non-open operands.

8.20.14 Definition: Open and closed operand positions

Code: *DPPS*

The **open operand positions** of an Agent are the operand positions of its open operands, and the **closed operand positions** those of its closed operands.

8.20.15 Definition: Type of an agent expression

Code: *SPET*

Consider a Call_agent *a*, of target type *T0*. Let *i1*, …, *im* ($m \geq 0$) be its open operand positions, if any, and let T_{i1}, .., T_{im} be the types of *f*'s formal arguments at positions *i1*, …, *im* (taking T_{i1} to be *T0* if *i1* = 0).

The type of *a* is:

- *PROCEDURE* [*T0, TUPLE* [T_{i1}, .., T_{im}]] if *f* is a procedure.
- *FUNCTION* [*T0, TUPLE* [T_{i1}, .., T_{im}], *R*] if *f* is a function of result type *R* other than *BOOLEAN*.
- *PREDICATE* [*T0, TUPLE* [T_{i1}, .., T_{im}]] if *f* is a function of result type *BOOLEAN*.

8.20.16 Semantics: Agent Expression semantics

Code: *MPAE*

The value of an agent expression *a* at a certain *construction time* yields a reference to an instance D0 of the type of *a*, containing information identifying:

- The associated feature of *a*.
- Its open operand positions.
- The values of its closed operands at the time of evaluation.

8.20.17 Semantics: Effect of executing *call* on an agent

Code: *MPCA*

Let D0 be an agent object with associated feature *f* and open positions *i1*, …, *im* ($m \geq 0$). The information in D0 enables a call to the procedure *call*, executed at any **call time** posterior to D0's construction time, with (if required) actual arguments a_{i1}, .., a_{im}, to perform the following:

- Produce the same effect as a call to *f*, using the closed operands at the closed operand positions and a_{i1}, .., a_{im}, evaluated at call time, at the open operand positions.
- In addition, if *f* is a function, setting the value of the query *last_result* for D0 to the result returned by such a call.

Signed by JV - 2026/10/8

8.21 Expressions

Informative Text

8.21.0.1 Through the various forms of Expression, software texts can include denotations of run-time values — objects and references.

8.21.0.2 Previous discussions have already introduced some of the available variants of the construct: Local, Call, Old, Manifest_tuple, Agent. The details of such others as Read_only appear in later chapters. The present one gives the full list of permissible expressions and the precise form of all but one of the remaining categories: operator expressions, equality and locals. The last category, constants, has its own separate presentation, just after this one. Other important notions covered below are the type of an expression and its semantics.

End

8.21.1 Syntax: Expressions

Code: *SWEX*

Expression ≜ Detachable_expression | Attached_expression | **Void**

Detachable_expression ≜
Read_only |
Local |
Call |
Precursor_call |
Parenthesized |
Old |
Operator_expression |
Bracket_expression |
Paren_expression |
Conditional_expression

Attached_expression ≜
Manifest_constant |
Manifest_tuple |
Equality |
Agent |
Creation_expression |
Object_test |
Manifest_string |
Address

Parenthesized ≜ "**(**" Expression "**)**"

Address ≜ "**$**" Variable

Once_string ≜ **once** Manifest_string

Boolean_expression ≜ **once** Manifest_string

8.21.1.1

8.21.1.2

8.21.1.3

8.21.1.4

Informative Text

8.21.1.5 Do not confuse:

Signed by JV - 2026/10/8

- A Parenthesized, of the form (*e*), which simply represents *e* parenthesized (as the name suggests) to remove possible syntactical ambiguities or supersede default operator precedence rules. More details below ADD CROSS-REFERENCE
- A Paren_expression, of the form *f* (e), similar to a Bracket_expression *f* [e] but with parentheses instead of brackets, representing a call to the feature which has been given a "parenthesis alias" in the current class.

8.21.1.6 The name of the latter construct uses "Paren" rather than "Parenthesis" (as a departure from the general practice, in Eiffel and in this Standard, to avoid abbreviations) to allay the risk of confusion.

End

8.21.2 Definition: Subexpression

Code: *DWSE*

[OPERATOR_EXPRESSION IS MISSING!!!] The **subexpressions** of an Expression or Boolean_expression *e* are *e* itself and (recursively) all the following expressions, for applicable cases depending on the form of *e*:

1 Parenthesized (*a*): the subexpressions of *a*.

2 Equality or Binary_expression or Unary_expression: the subexpressions of the operands.

3 Bracket_expression *f* [*a*1, … a_n] or Paren_expression *f* (*a*1, … a_n): the subexpressions of *f* and those of all of *a*1, … a_n.

4 Call: the subexpressions of both Actuals part, if any, of the Unqualified_call, and the Target in the case of an Object_call.

5 Qualified Call: the subexpressions of its target.

6 Precursor_call: the subexpressions of its unfolded form.

7 Call_agent: the subexpression of its Agent_actuals if any.

8 Inline_agent: the subexpressions of all expressions that appear in it.

9 Old expression **old** *a*: *a*.

10 Manifest_tuple [*a*1, … a_n]: the subexpressions of all of *a*1, … a_n.

11 Conditional_expression: the subexpressions of every Boolean_expression in every Then_case of its Then_case_list.

12 Boolean_expression of the Loop form, any of the subexpressions of every Expression and Boolean_expression appearing in its components.

Informative Text

8.21.2.1 The various cases list all the places in an expression *e* where components and sub-components of *e* may be expressions. The clauses are not exclusive but additive: they all add potential subexpressions to *e* itself, and case 5 adds to case 4 since a qualified call is also a Call.

8.21.2.2 Case 4 uses "the Unqualified_call part of a Call":

8.21.2.2.1 - Both of the available variants, Object_call and Non_object_call, indeed include an Unqualified_call. That Unqualified_call may have arguments, whose subexpressions are subexpressions of the call.

8.21.2.2.2 - In the case of an Object_call there is a Target, the *x* in *x.f* (…); the subexpressions of *x* are subexpressions of the call.

8.21.2.3 In case 12, the relevant components of the Loop are its Iteration, Invariant, Exit_condition and Loop_body parts.

8.21.2.4 Rule DWET, page 221, which defines the type of an expression, follows a similar structure to this DWSE since an expression's type often depends on the type of its subexpressions.

End

8.21.3 Definition: Operands

Code: *DWOP*

The **operands** of an Expression or Boolean_expression, defined only in the following cases, are:

1 For a Unary_expression ◊ *a*, where ◊ is a Unary operator: *a*.

2 For an Equality *a* § *b* where § is a Comparison operator: *a* and *b*.

3 For a Binary_expression *a* § *b* , where § is a Binary operator:*a* and *b*.

Informative Text

8.21.3.1 The notion of operand also exists for calls (DUOP, page 202) and agents (DPOP, page 213).

End

8.21.4 Semantics: Parenthesized semantics

Code: *SWPE*

If *e* is an expression, the value of the Parenthesized (*e*) is the value of *e*.

8.21.5 Syntax: Operator expressions

Code: *SWOP*

Operator_expression ≜ Unary_expression | Binary_expression

Unary_expression ≜ Unary Expression

Binary_expression ≜ Expression Binary Expression

8.21.6 Syntax (non-production): Operator precedence levels

Code: *SWPR*

13	**.**	(Dot notation, in qualified and non-object calls)
12	**old** **not + –** All free unary operators	(In postconditions) (Used as unary)
11	All free binary operators.	
10	^	(Used as binary: power)
9	* / // \\	(Used as binary: multiplicative arithmetic operators)
8	**+ –**	(Used as binary)
7	**. .**	(To define an interval)
6	**= /= ~ /~ < > <= >=**	(Comparison and relational operators)
5	**and and then**	(Conjunctive boolean operators)
4	**or or else xor**	(Disjunctive boolean operators)
3	**implies**	(Implicative boolean operator)
2	**[ ]**	(Manifest tuple delimiter)
1	**;**	(Optional semicolon between an Assertion_clause and the next)

Informative Text

8.21.6.1 This precedence table includes the operators that may appear in an Operator_expression, the equality and inequality symbols used in Equality expressions, as well as other symbols and keywords which also occur in expressions and hence require disambiguating: the semicolon in its role as separator for Assertion_clause; the **old** operator which may appear in an Old expression as part of a Postcondition; the dot **.** of dot notation, which binds tighter than any other operator.

Signed by JV - 2026/10/8

8.21.6.2 The operators listed include both standard operators and predefined operators (**=**, **/=**, **~**, **/~**). For a free operator, you cannot set the precedence: all free unaries appear at one level, and all free binaries at another level.

End

8.21.7 Definition: Parenthesized form of an expression

Code: *DWPA*

The **parenthesized form** of an expression is the result of rewriting every subexpression of one of the forms below, where § and ‡ are different binary operators, ◊ and ♣ are unary operators, and *a*, *b*, *c* arbitrary operands, as follows:

1 For *a* § *b* § *c* where § is not the power operator ^: (*a* § *b*) § *c* (left associativity).

2 For *a* ^ *b* ^ *c* : *a* ^ (*b* ^ *c*) (right associativity).

3 For *a* § *b* ‡ *c*: (*a* § *b*) ‡ *c* if the precedence of ‡ is lower than the precedence of § or the same, and *a* § (*b* ‡ *c*) otherwise.

4 For ◊ ♣ *a*: ◊ (♣ *a*)

5 For ◊ *a* § *b*: (◊ *a*) § *b*

6 For *a* § ◊ *b*: *a* § (◊ *b*)

7 For a subexpression *e* to which none of the previous patterns applies: *e* unchanged.

8.21.8 Definition: Target-converted form of a binary expression

Code: *DWTC*

The **target-converted form** of a Binary_expression *x* § *y*, where the one-argument feature of alias § in the base class of *x* has the Feature_name *f*, is:

1 If the type *TY* of *y* is not compatible with the type of the formal argument of *f*, and the type *TX* of *x* converts to *TY*: (({*TY*}) [*x*]) § *y*.

2 Otherwise: the original expression, *x* § *y*.

Informative Text

8.21.8.1 (({*TY*}) [*x*]) denotes the result of converting the value of *x* to type *TY*. {*TY*} is the object of type *TYPE* denoting the type *TY*; class *TYPE* has a bracket feature *adapted* which, when applied to a value *x*, converts it, if possible, to the current type, in this case *TY*.

8.21.8.2 This definition allows us, if the feature from *x's* type *TX* cannot accept a *TY* argument, but, the other way around, the value of *x* can be turned into an object of type *TY*, and *TY* has a version of the operator, to find out if that inerpretation is valid and if so to use it.

8.21.8.3 The archetypal example is *your_integer* + *your_real*, which we can interpret as (({*REAL*}) [*your_integer*]) + *your_real*, where "+" represents the *plus* feature from *REAL*. Rather than introducing special rules for such classic cases involving traditional numeric types, the language definition provides a general mechanism applicable to any such situation.

8.21.8.4 [PARKED TEXT, NOT SURE WHERE IT BELONGS, PROBABLY OBSOLETE, RELATING TO AN EARLIER VERSION] The optional **[FIXME]**, supports mixed-type expressions causing a conversion of the target, as in the expression *your_integer* + *your_real*, which should use the "+" operation from *REAL*, not *INTEGER*, for compatibility with ordinary arithmetic practice. See the presentation of conversions.

8.21.8.5 We don't need any specific semantic rule either: the normal semantic rules, applied to the target-converted form, yield exactly what we need. FIXFIXFIXFIXFIX

End

8.21.9 Validity: Operator Expression rule

Code: *VWOP*

A Unary_expression§ *x* or *Binary_expressionx* § *y*, for some operator §, is valid if and only if it satisfies the following conditions:

1 A feature of the base class of *x* is declared as **alias** "§".

2 The expression's Equivalent Dot Form is a valid Call.

8.21.10 Syntax: Bracket expressions

Code: *SWBR*

Bracket_expression ≜ Bracket_or_paren_target "**[**" Actuals "**]**"

Bracket_or_paren_target ≜ Target | Once_string | Manifest_constant | Manifest_tuple

Informative Text

8.21.10.1 Bracket expressions are of the form *x* [*e*], a syntactical simplification for *x***.***f* (*e*) where f is the feature that has been given the "bracket alias" in the base class of the type of *x*. ADD CROSS-REFERENCE

8.21.10.2 Bracket_or_paren_target covers the target of bracket expressions as well as paren expressions (next syntax element).

8.21.10.3 Target (SUCA, page 200) covers every kind of expression that can be used as target of a call, including simple variants like Local variables and formal arguments, as well as Call representing the application of a query to a target that may itself be the result of applying calls.

End

8.21.11 Syntax: Paren expressions

Code: *SWBR*

Paren_expression ≜ Bracket_or_paren_target "**(**" Actuals "**)**"

Informative Text

8.21.11.1 Paren expressions are of the form *x* (*e*), a syntactical simplification for *x***.***f* (*e*) where *f* is the feature that has been given the "parenthesis alias" in the base class of the type of *x*. ADD CROSS-REFERENCE

End

8.21.12 Validity: Bracket and Paren Expression rule

Code: *VWBP*

A Bracket_expression *x* [a_1, … a_n] or Paren_expression *x* (a_1, … a_n) is valid if and only if it satisfies the following conditions:

1 A feature of the base class of *x* is declared is, respectively, a bracket feature or parenthesis feature.

2 The expression's Equivalent Dot Form is a valid Call.

Informative Text

8.21.12.1 A feature *f* is a "bracket feature" if it is declared with **alias** "[]", allowing you to write a call *x***.***f* (a_1, … a_n) as *x* [a_1, … a_n], and similarly *x* (a_1, … a_n) for a "parenthesis feature" (DFOB, page 106). The feature *f* must be a routine with at least one argument and, in the bracket case, a function.

End

8.21.13 Syntax: Conditional expression

Code: *SOCE*

Conditional_expression ≜ **if** Then_case_list [Else_case] **end**

Then_case_list ≜ {Then_case **elseif** …}$^+$

Then_case ≜ Boolean_expression **then** Expression

Else_case ≜ **else** Expression

Signed by JV - 2026/10/8

Informative Text

8.21.13.1 The simplest form of a Conditional_instruction is

8.21.13.2 **if** *cond* **then** *iinst*$_1$; *inst*$_2$; … **end**

8.21.13.3 involving a condition (boolean expression) *cond* and a compound made of any number of instructions. It will execute the compound if the condition evaluates to true, and do nothing otherwise.

8.21.13.4 You may add an Else_case, as in

8.21.13.5 **if** *cond* **then** *inst*$_1$; *inst*$_2$; … **else** *other_inst*$_1$ …**end**

8.21.13.6 so that if *cond* evaluates to false the instruction will (instead of doing nothing) execute the second compound.

8.21.13.7 If you want more than one condition and hence more than two cases, you may add any number of **elseif** parts, as in

8.21.13.8 **if** *m* = 1 **then** *n* := *q* **elseif** *m* = 2 **then** *n* := *r* **else** *n* := *s* **end**

8.21.13.9 although when the number of cases grows other techniques are usually preferable to conditional instructions: multi-branch (discussed below) or dynamic binding.

8.21.13.10 If, as in the last example, the various branches are all assignments (of different source values) to the same target, you may also use a conditional *expression (ADD REFERENCE)*, as in

8.21.13.11 *n* := (**if** *m* = 1 **then** *q* **elseif** *m* = 2 **then** *r* **else** *s* **end**) -- Parentheses for clarity, not required.

End

8.21.14 Definition: Equivalent Dot Form of an expression

Code: *DWDO*

Any Expression *e* has an **Equivalent Dot Form**, not involving (in any of its subexpressions) any Bracket_expression, Paren_expression or Operator_expression, and defined as follows, where *C* denotes the base class of *x*, *pe* denotes the Parenthesized Form of *e*, and *x'*, *y'*, *c'* denote the Equivalent Dot Forms (recursively) of *x*, *y*, *c*:

1 If *pe* is a Unary_expression § *x*: *x'***.***f*, where *f* is the Feature_name of the no-argument feature of alias § in *C*.

2 If *pe* is a Binary_expressionof target-converted form *x* § *y*: *x'***.***f* (*y'*) where *f* is the Feature_name of the one-argument feature of alias § in *C*.

3 If *pe* is a Bracket_expression *x* [*y*]: *x'***.***f* (*y'*) where *f* is the Feature_name of the feature declared as **alias** "[]" in *C*.

4 If *pe* is a Paren_expression *x* (*y*): *x'***.***f* (*y'*) where *f* is the Feature_name of the feature declared as **alias** "()" in *C*.

5 If *pe* has no subexpression other than itself: *pe*.

6 In all other cases: (recursively) the result of replacing every subexpression of *e* by its Equivalent Dot Form.

Informative Text

8.21.14.1 For constructs such as assertions it is useful to define a notion of Boolean_expression. The syntax gave two variants: Expression, in which case the type must obviously be *BOOLEAN*; but also Loop. This second case covers “quantifier” expressions of one of the forms

8.21.14.1.1 • **across** *s* **as** *c* **all** *boolexp* **end**, expressing that all *items* in the structure *s* satisfy *boolexp*, itself a boolean expression relative to a typical element accessible through *c*.

8.21.14.1.2 • **across** *s* **as** *c* **some** *boolexp* **end**, expressing that *at least one* of the items in the structure *s* satisfies *boolexp*, itself a boolean expression relative to a typical element known through *c*.

End

8.21.15 Validity: Boolean Expression rule

Code: *VWBE*

A Boolean_expression is valid if and only if it satisfies one of the following conditions:

1 It is of the Expression kind, and the type of the expression is *BOOLEAN*.

2 It is of the Loop kind, made of an Iteration of the All_body or Some_body kind.

Informative Text

8.21.15.1 In case 1, he "type" of an Expression is the result of applying the definiton of "type of an expression" appearing below.

8.21.15.2 Case 2 makes it possible to use conditions of the form **across** *s* **as** *c* **all** *boolexp* **end** and **across** *s* **as** *c* **some** *boolexp* **end**. The validity rule on loops take care of ruling out any improper clause (such as **from**) for such loops.

End

8.21.16 Validity: Identifier rule

Code: *VWID*

An Identifier appearing in an expression in a class *C*, other than as the static feature of a qualified Call, must be the name of a feature of *C*, or a local variable of the enclosing feature or inline agent if any, or a formal argument of the enclosing reature or inline agent if any, or the object-test local of an Object_test, or the Identifier of an Iteration.

Informative Text

8.21.16.1 The restriction "other than as the feature of a qualified Call" excludes an identifier appearing immediately after a dot to denote a feature being called on a target object: in *a* + *b*.*c* (*d*), the rule applies to *a*, *b* (target of a Call) and *d* (actual argument), but not to *c* (feature of a qualified Call). For *c* the relevant constraint is the Call rule, which among other conditions requires *c* to be a feature of the base class of *b* (in other words, of its type).

8.21.16.2 The Identifier rule is not a full "if and only if" rule; in fact it is conceptually superfluous since it follows from earlier, more complete constraints. language processing tools may find it convenient as a simple criterion for detecting the most common case of invalid Identifier in expression.

End

8.21.17 Definition: Expression Type rule

Code: *DWET*

The type of an Expression *e* is:

1 For a Parenthesized (*f*): (recursively) the type of *f*.

2 For an Equality: *BOOLEAN*.

3 For an Operator_expression, Bracket_expression or Parenthesized: (recursively) the type of the Equivalent Dot Form of *e*.

4 For a Call: the type of *e* as determined by the Expression-Call Type definition with respect to the current type.

5 For a Precursor_call: (recursively) the type of its unfolded form.

6 For an Agent: as given by the definition of the type of an agent expression.

7 For **old** *f*: (recursively) the type of *f*.

8 For a Manifest_tuple [a_1, … a_n] ($n \geq 0$): *TUPLE* [T_1, … T_n] where each T_i is (recursively) the type of a_i.

Signed by JV - 2026/10/8

9 For a Manifest_tuple: the type of the Expression in the **then** part.

10 For a Boolean_expression: *BOOLEAN*.

11 For the predefined Read_only **Current**:: the current type.

12 For a formal argument of a routine: the type declared for *e*.

13 For an object-test local: its declared type.

14 For **Result**, appearing in the text of a query *f*: the detachable version of the result type of *f*.DUCT, page 209

15 For **Retry**: *BOOLEAN*.

16 For a declared local variable: the type declared for *e*.

17 For a Manifest_constant: as given by the definition of the type of a manifest constant.

18 For an Object_test: *BOOLEAN*.

19 For a Once_string: *STRING*.

20 For an Address **$***v*: *TYPED_POINTER* [*T*] where *T* is (recursively) the type of *v*.

21 For a Creation_expression: the Explicit_creation_type.

22 For a Loop: *BOOLEAN*

23 For the Identifier of an Iteration, using an Expression of type *ET*: the result type of *ET*'s version of the query *new_cursor* from the Kernel Library class *ITERABLE*.

24 For **Void**: **detachable** *NONE*.

8.21.18 Semantics: Expression semantics (strict case)

Code: *MWES*

The **value** of an Expression, other than a Binary_expression whose Binary is semistrict, is the value of its Equivalent Dot Form.

Informative Text

8.21.18.1 This semantic rule and the preceding validity constraint make it possible to forego any specific semantics for operator expressions (except in one special case) and define the value of any expression through other semantic rules of the language, in particular the rules for calls and entities.

8.21.18.2 This applies in particular to arithmetic and relational operators (for which the feature declarations are in basic classes such as *INTEGER* and *REAL*) and to boolean operators (class *BOOLEAN*): in principle, although not necessary as implemented by compilers, *a* + *b* is just a feature call like any other.

8.21.18.3 The excluded case — covered by a separate rule — is that of a binary expression using one of the three **semistrict** operators: **and then**, **or else**, **implies**. This is because the value of an expression such as *a* **and then** *b* is not entirely defined by its Equivalent Dot Form *a*.*conjuncted_semistrict (b)*, which needs to evaluate *b*, whereas the **and then** form explicitly ignores *b* when *a* has value *False*, as the value of the whole expression is *False* even if *b* does not have a defined value, a case which should not be treated as an error.

End

8.21.19 Definition: Semistrict operators

Code: *DWNS*

A **semistrict operator** is any one of the three operators **and then, or else** and **implies**, applied to operands of type *BOOLEAN*.

8.21.20 Semantics: Operator expression semantics (semistrict case)

Code: *MWEN*

For *a* and *b* of type *BOOLEAN*:

- The value of *a* **and then** *b* is: if *a* has value false, then false; otherwise the value of *b*.

- The value of *a* **or else** *b* is: if *a* has value true, then true; otherwise the value of *b*.
- The value of *a* **implies** *b* is: if *a* has value false, then true; otherwise the value of *b*.

Informative Text

8.21.20.1 The semantics of other kinds of expression, and Eiffel constructs in general, is **compositional**: the value of an expression with subexpressions *a* and *b*, for example *a* + *b* (where *a* and *b* may themselves be complex expressions), is defined in terms of the values of *a* and *b*, obtained from the same set of semantic rules, and of the connecting operators, here +. Among expressions, those involving semistrict operators are the only exception to this general style. The above rule is not strictly compositional since it tells us that in certain cases of evaluating an expression involving *b* we should not consider the value of *b*. It's not just that we *may* ignore the value of *b* in some cases — which would also be true of *a* **and** *b* (strict) when *a* is false — but that we *must* ignore it lest it prevents us from evaluating the expression as a whole.

8.21.20.2 It's this lack of full compositionality that makes the above rule more **operational** than the semantic specification of other kinds of expression. Their usual form is "*the value of an expression of the form X is Y*", where *Y* only refers to values of subexpressions of *X*. Such rules normally don't mention order of execution. They respect compositionality and leave compilers free to choose any operand evaluation order, in particular for performance. Here, however, order matters: the final requirement of the rule *requires* that the computation first evaluate *a*. We need this operational style to reflect the special nature of nonstrict operators, letting us sometimes get a value for an expression whose second operand does not have any.

End

8.22 Control structures

Informative Text

8.22.0.1 The previous discussions have described the “bones” of Eiffel software: the module and type structure of systems. Here we begin studying the “meat”: the elements that govern the execution of applications.

8.22.0.2 Control structures are the constructs used to schedule the run-time execution of instructions. There are four of them: sequencing (compound), conditional, multi-branch choice and loop. A complementary construct is the Debug instruction.

8.22.0.3 As made clear by the definition of “non-exception semantics” in the semantic rule for Compound, which indirectly governs all control structures (since al instructions are directly or indirectly part of a Compound), the default semantics assumes that none of the instructions executed as part of a control structure triggers an exception. If an exception does occur, the normal flow of control is interrupted, as described by the rules of exception handling in the discussion of this topic.

End

8.22.1 Semantics: Compound (non-exception) semantics

Code: *MOCO*

The effect of executing a Compound is:

- If it has zero instructions: to leave the state of the computation unchanged.
- If it has one or more instructions: to execute the first instruction of the Compound, then (recursively) to execute the Compound obtained by removing the first instruction.

This specification, the **non-exception semantics** of Compound, assumes that no exception is triggered. If the execution of any of the instructions triggers an exception, the Exception Semantics rule takes effect for the rest of the Compound’s instructions.

Informative Text

8.22.1.1 Less formally, this means executing the constituent instructions in the order in which they appear in the Compound, each being started only when the previous one has been completed.

8.22.1.2 Note that a Compound can be empty, in which case its execution has no effect. This is useful for examples when refactoring the branches of a Conditional_instruction: you might temporarily remove all the instructions of the Else_part, but not the Else_part itself yet as you think it may be needed later.

End

8.22.2 Syntax: Conditional instruction

Code: *SOCI*

Conditional_instruction ≜ **if** Then_part_list [Else_part] **end**

Then_part_list ≜ {Then_part **elseif** …}$^{+}$

Then_part ≜ Boolean_expression **then** Compound

Else_part ≜ **else** Compound

Informative Text

8.22.2.1 The simplest form of a Conditional_instruction is

if *cond* **then** *iinst*$_1$; *inst*$_2$; … **end**

involving a condition (boolean expression) *cond* and a compound made of any number of instructions.

8.22.2.2 It will execute the compound if the condition evaluates to true, and do nothing otherwise.

8.22.2.3 You may add an Else_part, as in

if *cond* **then** $inst_1$; $inst_2$; … **else** $other\_inst_1$ …**end**

so that if *cond* evaluates to false the instruction will (instead of doing nothing) execute the second compound.

8.22.2.4 If you want more than one condition and hence more than two cases, you may add any number of **elseif** clauses, as in

if *m* = 1 **then** *n* := *q* **elseif** *m* = 2 **then** *n* := *r* **else** *n* := *s* **end**

although when the number of cases grows other techniques are usually preferable to conditional instructions: multi-branch (discussed below) or dynamic binding.

8.22.2.5 If, as in the last example, the various branches are all assignments (of different source values) to the same target, you may also use a conditional *expression*, as in

n := (**if** *m* = 1 **then** *q* **elseif** *m* = 2 **then** *r* **else** *s* **end**) -- Parentheses for clarity, not required.

End

8.22.3 Definition: Secondary part

Code: *DOSP*

The **secondary part** of a Conditional_instruction possessing at least one **elseif** is the Conditional_instruction obtained by removing the initial "**if** Then_part_list" and replacing the first **elseif** of the remainder by **if**.

8.22.4 Definition: Prevailing immediately

Code: *DOPI*

The execution of a Conditional_instruction starting with **if** $condition_1$ is said to **prevail immediately** if $condition_1$ has value true.

8.22.5 Semantics: Conditional instruction semantics

Code: *MOCI*

The effect of a Conditional_instruction is:

- If it prevails immediately: the effect of the first Compound in its Then_part_list.
- Otherwise, if it has at least one **elseif**: the effect (recursively) of its secondary part.
- Otherwise, if it has an Else_part: the effect of the Compound in that Else_part.
- Otherwise: no effect.

Informative Text

8.22.5.1 Like the instruction studied next, the Conditional_instruction is a "multi-branch" choice instruction, thanks to the presence of an arbitrary number of **elseif** clauses. These branches do not have equal rights, however; their conditions are evaluated in the order of their appearance in the text, until one is found to evaluate to true. If two or more conditions are true, the one selected will be the first in the syntactical order of the clauses.

End

8.22.6 Definition: Inspect expression

Code: *DOIS*

The **inspect expression** of a Multi_branch is the expression appearing after the keyword **inspect**.

8.22.7 Syntax: Multi-branch instructions

Code: *SOMB*

Multi_branch ≜ **inspect** Expression [When_part_list] [Else_part] **end**

When_part_list ≜ When_part+

When_part ≜ **when** Choices **then** Compound

Choices ≜ {Choice "," …}+

Choice ≜ Constant | Manifest_type | Constant_interval| Type_interval

Constant_interval ≜ Constant "**..**" Constant

Signed by JV - 2026/10/8

Type_interval ≜ Manifest_type ".." Manifest_type

8.22.8 Definition: Interval Code: *DOIN*

An **interval** is a Constant_interval or Type_interval.

8.22.9 Definition: Unfolded form of a multi-branch Code: *DOUM*

To obtain the **unfolded form** of a Multi_branch instruction, apply the following transformations in the order given:

1 Replace every constant inspect value by its manifest value.

2 If the type *T* of the inspect expression is any sized variant of *CHARACTER*, *STRING* or *INTEGER*, replace every inspect value *v* by {*T*} *v*.

3 Replace every interval by its unfolded form.

Informative Text

8.22.9.1 Step 2 enables us, with an inspect expression of a type such as *INTEGER_8*, to use constants in ordinary notation, such as 1, rather than the heavier {*INTEGER_8*} 1. Unfolded form constructs this proper form for us. The rules on constants make this convention safe: a value that doesn't match the type, such as 1000 here, will cause a validity error.

End

8.22.10 Definition: Unfolded form of an interval Code: *DOUI*

The **unfolded form** of an interval *a*..*b* is the following (possibly empty) list:

1 If *a* and *b* are constants, both of either a character type, a string type or an integer type, and of manifest values *va* and *vb*: the list made up of all values *i*, if any, such that *va* ≤ *i* ≤ *vb*, using character, integer or lexicographical order respectively.

2 If *a* and *b* are both of type *TYPE* [*T*] for some *T*, and have manifest values *va* and *vb*: the list containing every Manifest_type of the system conforming to *vb* and to which *va* conforms.

3 If neither of the previous two cases apply: an empty list.

Informative Text

8.22.10.1 The "manifest value" of a constant is the value that has been declared for it, ignoring any Manifest_type: for example both 1 and {*INTEGER_8*} 1 have the manifest value 1.

8.22.10.2 The symbol .. is not a special symbol of the language but an alias for a feature of the Kernel Library class *PART_COMPARABLE*, which for any partially or totally ordered set and yielding the set of values between a lower and an upper bound. Here, the bounds must be constant.

8.22.10.3 A note for implementers: type intervals such as {*U*}..{*T*}, denoting all types conforming to *T* and to which *U* conforms, may seem to raise difficult implementation issues: the set of types, which the unfolded form seems to require that we compute, is potentially large; the validity (Multi-Branch rule) requires that all types in the unfolded form be distinct, which seems to call for tricky computations of intersections between multiple sets; and all this may seem hard to reconcile with incremental compilation, since a type interval may include types from both our own software and externally acquired libraries, raising the question of what happens on delivery of a new version of such a library, possibly without source code. Closer examination removes these worries:

8.22.10.3.1 • There is no need actually to compute entire type intervals as defined by the unfolded form. Listing {*U*}..{*T*} simply means, when examining a candidate type *Z*, finding out whether *Z* conforms to *T* and *U* to *Z*.

8.22.10.3.2 • To ascertain that such a type interval does not intersect with another {*Y*}..{*X*}, the basic check is that *Y* does not conform to *T* and *U* does not conform to *X*.

Signed by JV - 2026/10/8

8.22.10.3.3 • If we add a new set of classes and hence types to a previously validated system, a new case of intersection can only occur if either: a new type inherits from one of ours, a case that won't happen for a completely external set of reusable classes and, if it happens, should require re-validating since existing Multi_branch instructions may be affected; or one of ours inherits from a new type, which will happen only when we modify our software *after* receiving the delivery, and again should require normal rechecking.

End

8.22.11 Validity: Interval rule

Code: *VOIN*

An interval is valid if and only if its unfolded form is not empty.

Informative Text

8.22.11.1 This rule applies to both forms of interval (per the definition of this concept above, DOIN, page 226): Constant_interval such as 1**..**10 and Type_interval such as {*FIGURE*} **..** {*SQUARE*}.

End

8.22.12 Definition: Inspect values of a multi-branch

Code: *DOIV*

The **inspect values** of a Multi_branch instruction are all the values listed in the Choices parts of the instruction's unfolded form.

Informative Text

8.22.12.1 The set of inspect values may be infinite in the case of a string interval, but this poses no problem for either programmers or compilers, meaning simply that matches will be determined through lexicographical comparisons.

End

8.22.13 Validity: Multi-branch rule

Code: *VOMB*

A Multi_branch instruction is valid if and only if its unfolded form satisfies the following conditions.

1 Inspect values are all valid.

2 Inspect values are all constants.

3 The manifest values of any two inspect values are different.

4 If the inspect expression is of type *TYPE* [*T*] for some type *T*, all inspect values are types.

5 If case 4 does not apply, the inspect expression is one of the sized variants of *INTEGER*, *CHARACTER* or *STRING*.

8.22.14 Semantics: Matching branch

Code: *MOMA*

During execution, a **matching branch** of a Multi_branch is a When_part *wp* of its unfolded form, satisfying either of the following for the value *val* of its inspect expression:

1 *val* **~** *i*, where *i* is one of the non-Manifest_type inspect values listed in *wp*.

2 *val* denotes a Manifest_type listed among the choices of *wp*.

Informative Text

8.22.14.1 The Multi-branch rule is designed to ensure that in any execution there will be at most one matching branch.

8.22.14.2 In case 1, we look for object equality, as expressed by **~**. Strings, in particular, will be compared according to the function *is_equal* of *STRING*. A void value, even if type-wise permitted by the inspect expression, will never have a matching branch.

Signed by JV - 2026/10/8

8.22.14.3 In case 2, we look for an exact type match, not just conformance. For conformance, we have type intervals: to match types conforming to some *T*, use {*NONE*}**..**{*T*}; for types to which *T* conforms, use {*T*}**..**{*ANY*}.

End

8.22.15 Semantics: Multi-Branch semantics

Code: *MOMB*

Executing a Multi_branch with a matching branch consists of executing the Compound following the **then** in that branch. In the absence of matching branch:

1 If the Else_part is present, the effect of the Multi_branch is that of the Compound appearing in its Else_part.

2 Otherwise the execution triggers an exception of type *BAD_INSPECT_VALUE*.

8.22.16 Syntax: Loops

Code: *SOLO*

Loop ≜
[Iteration]
[Initialization]
[Invariant]
[Exit_condition]
Loop_body
[Variant]
end

Iteration ≜ **across** Expression **as** Identifier

Initialization ≜ **from** Compound

Exit_condition ≜ **until** Boolean_expression

Loop_body ≜ Do_body | All_body | Some_body

Do_body ≜ **loop** Compound

All_body ≜ **all** Boolean_expression

Some_body ≜ **some** Boolean_expression

Informative Text

8.22.16.1 There are two basic forms of loop:

8.22.16.1.1 • The **from** form, as in **from** *s*`.`*start* **until** *s*`.`*after* **loop** *s*`.`*action; s*`.`*move* **end**.

8.22.16.1.2 • The **across** form, more abstract as it hides the need to advance a cursor, as in **across** *s* **as** *c* **loop** *c*`.`*action* **end**. The cursor *c* is provided automatically (it need not be declared except for appearing in the **as** clause) provides local access to every element of the structure *s* in turn. This form is applicable to structures *s* whose type conforms to *ITERABLE* [*T*] for some *T*, where *ITERABLE* is a Kernel Library class providing iteration facilities. All library classes describing traversable data structures are expected to be descendants of *ITERABLE*.

8.22.16.2 The two forms are exclusive but can share many of the other components, such as an Exit_condition, as made precise by the following validity rule.

End

8.22.17 Validity: Loop rule

Code: *VOLO*

A Loop instruction is valid if and only if it satisfies the following conditions.

1 At least one of Initialization and Iteration is present.

2 At least one of Iteration and Exit_condition is present.

3 If the Loop_body is of the All_body or Some_body form, there is no Initialization part.

Signed by JV - 2026/10/8

Informative Text

8.22.17.1 The Iteration form requires conditions of its own, as expressed next.

End

8.22.18 Validity: Iteration rule

Code: *VOIT*

An Iteration instruction is valid if and only if it satisfies the following conditions.

1 The type of the Expression conforms to *ITERABLE* [*T*], based on the Kernel Library class *ITERABLE*, for some type *T*. (By VHPR (5), this *T* is unique.)

2 The Identifier is a fresh identifier.

Informative Text

8.22.18.1 For programmers, the **across** form, involving an iteration, is often the simpler. For the language definition, however, the explicit form with an Initialization (**from** …) and an Exit_condition (**until** …) is the basic one, serving as a basis for the definition of the **across** variant through the following "unfolded form". The definition is a bit heavy, but is necessary to keep the semantic specification simple

End

8.22.19 Definition: Unfolded form of a loop

Code: *SOUL*

Every Loop has an **unfolded form** with no Iteration, All_body or Some_body. To obain it, apply the following transformations in the order given:

1 In the case of a Loop_body of the form All_body or Some_body, assume a fictitious boolean attribute *b*, distinct from all other identifiers in the class. Several of the subsequent steps refer to this *b*.

2 For a Loop_body of the form All_body or Some_body, there is no Initialization (by the Loop rule). Add the Initialization clause:

from *b* := **True** (for an All_body), or
from *b* := **False** (for a Some_body)

Then replace the All_body or Some_body, of expression *e*, with the Do_body

loop *b* := (*b* **and then** *e*) (for an All_body), or
loop *b* := (*b* **or else** *e*) (for a Some_body)

3 If there is no Initialization clause, add an empty one. The following steps apply only if there is an Iteration clause **across** *s* **as** *c*.

4 Introduce a fictitious attribute *ic* with a name that does not conflict with any identifier, and The same type as the result type *U* of *new_cursor* in the type of *s*. *U* conforms to *ITERABLE* [*T*], for some *T* as specified in the Iteration rule VOIT (1). In the rest of this definition, any occurrence of a name of a feature *f* of *ITERATION_CURSOR* (such as *start*, *forth*, *item*, *after*) is an abbreviation for "the version of *f* in *U*".

5 Replace any occurrence of *c* in the Initialization, the Exit_condition and the Loop_body by *ic*.

6 At the beginning of the Initialization, add the instructions: ic := *s*.*new_cursor*; *ic*.*start*

7 If there an Exit_condition **until** *e*, replace it by **until** *ic*.*after* **or else** (*e*); if not add the following one: **until** *ic*.*after*.

8 At the end of the Loop_body (which is now of theDo_body form), add the instruction *ic*.*forth*.

9 Remove the Iteration.

Signed by JV - 2026/10/8

Informative Text

8.22.19.1 The resulting unfolded form does not include specimens of any of the constructs Iteration, All_body and Some_body, but has the same semantics as the original form, translated into a more explicit variant that spells out every step of the iteration and the exit condition.

End

8.22.20 Semantics: Loop semantics

Code: *MOLO*

The effect of executing a Loop *l* is the effect of executing its unfolded form *ul*. The effect of executing *ul* is the effect of executing the Compound of its Initialization, then its Loop_body.

The effect of executing a Loop_body is:

- If the Boolean_expression of the Exit_condition evaluates to true: no effect (leave the state of the computation unchanged).
- Otherwise: the effect of executing the Compound clause, followed (recursively) by the effect of executing the Loop_body again in the resulting state.

8.22.21 Syntax: Debug instructions

Code: *SODB*

Debug ≙ **debug** ["**(**"Key_list "**)**"] Compound **end**

8.22.22 Semantics: Debug semantics

Code: *MODB*

A language processing tool must provide an option that makes its possible to enable or disable Debug instructions, both globally and for individual keys of a Key_list. Such an option may be settable for an entire system, or for individual classes, or both.

Letter case is not significant for a debug key.

The effect of a Debug instruction depends on the mode that has been set for the current class:

- If the Debug option is on generally, or if the instruction includes a Key_list and the option is on for at least one of the keys in the list, the effect of the Debug instruction is that of its Compound.
- Otherwise the effect is that of a null instruction.

Signed by JV - 2026/10/8

8.23 Creating objects

Informative Text

8.23.0.1 The dynamic model, whose major properties were reviewed in the preceding presentations, is highly flexible; your systems may create objects and attach them to entities at will, according to the demands of their execution. The following discussion explores the two principal mechanisms for producing new objects: the Creation_instruction and its less frequently encountered sister, the Creation_expression.

8.23.0.2 A closely related mechanism — **cloning** — exists for duplicating objects. This will be studied separately, with the mechanism for copying the contents of an object onto another.

8.23.0.3 The creation constructs offer considerable flexibility, allowing you to rely on language-defined initialization mechanisms for all the instances of a class, but also to override these defaults with your own conventions, to define any number of alternative initialization procedures, and to let each creation instruction provide specific values for the initialization. You can even instantiate an entity declared of a generic type — a non-trivial problem since, for *x* declared of type *G* in a class *C* [*G*], we don't know what actual type *G* denotes in any particular case, and how one creates and initializes instances of that type.

8.23.0.4 In using all these facilities, you should never forget the methodological rule governing creation, as expressed by the following principle.

End

8.23.1 Semantics: Creation principle

Code: *MKIP*

Any execution of a creation operation must produce an object that satisfies the invariant of its generating class.

Informative Text

8.23.1.1 Such is the theoretical role of creation: to make sure that any object we create starts its life in a state satisfying the corresponding invariant. The various properties of creation, reviewed next, are designed to ensure this principle.

End

8.23.2 Definition: Creation operation

Code: *DKOP*

A **creation operation** is a creation instruction or creation expression. -- TODO: USE THIS NOTION MORE OFTEN INSTEAD OF "CREATION INSTRUCTION OR CREATION EXPRESSION"

8.23.3 Validity: Creation Precondition rule

Code: *VKCP*

A Precondition of a routine *r* is **creation-valid** if and only if its unfolded form *uf* satisfies the following conditions:

1 The predefined entity **Current** does not appear in *uf*.

2 No Unqualified_call appears in *uf*.

3 Every feature whose final name appears in *uf* is available to every class to which *r* is available for creation.

Informative Text

8.23.3.1 This definition is not itself a validity constraint, but is used by condition 5 of the Creation Clause rule below; giving it a code as for a validity constraint enables compilers to provide a precise error message in case of a violation.

Signed by JV - 2026/10/8

8.23.3.2 Requiring preconditions to be creation-valid will ensure that a creation procedure does not try to access, in the object being created, fields whose properties are not guaranteed before initialization.

8.23.3.3 The definition relies on the "unfolded form" of an assertion, which reduces it to a boolean expression with clauses separated by **and then**. Because the unfolded form uses the Equivalent Dot Form, condition 3 also governs the use of operators: with *plus* **alias** "+", the expression *a + b* will be acceptable only if the feature *plus* is available for creation as stated.

End

8.23.4 Syntax: Creators parts

Code: *SKPA*

Creators ≜ Creation_clause+

Creation_clause ≜ **create** [Clients] [Header_comment] Creation_procedure_list

Creation_procedure_list ≜ {Creation_procedure ","…}+

Creation_procedure ≜ Feature_name

8.23.5 Definition: Unfolded Creators part of a class

Code: *DKUP*

The **unfolded creators part** of a class *C* is a Creators defined as:

1 If *C* has a Creators part *c*: *c*.

2 If *C* is deferred: an empty Creators part.

3 Otherwise, a Creators part built as follows, *dc_name* being the final name in *C* of its version of *default_create* from *ANY*:

```
create
    dc_name
```

Informative Text

8.23.5.1 For generality the definition is applicable to any class, even though for a deferred class (case 2) it would be invalid to include a Creators part. This causes no problem since the rules never refer to a deferred class actually extended with its unfolded creators part.

8.23.5.2 Case 3 reflects the convention that an absent Creators part stands for **create** *dc_name* — normally **create** *default_create*, but *dc_name* may be another name if the class or one of its proper ancestors has renamed *default_create*.

End

8.23.6 Validity: Creation Clause rule

Code: *VKCC*

A Creation_clause in the unfolded creators part of a class *C* is valid if and only if it satisfies the following conditions, the last four for every Feature_name *cp_name* in the clause's Feature_list:

1 *C* is effective.

2 *cp_name* appears only once in the Feature_list.

3 *cp_name* is the final name of some procedure *cp* of *C*.

4 *cp* is not a once routine.

5 The precondition of *cp*, if any, is creation-valid.

Informative Text

8.23.6.1 As a result of conditions 1 and 4, a creation procedure may only be of the **do** form (the most common case) or External.

Signed by JV - 2026/10/8

8.23.6.2 The prohibition of **once** creation procedures in condition 4 is a[FIX FOR ONCE CLASSES****]consequence of the Creation principle: with a once procedure, the first object created would satisfy the invariant (assuming the creation procedure is correct), but subsequent creation instructions would not execute the call, and hence would limit themselves to the default initializations, which might not ensure the invariant.

8.23.6.3 As a corollary of condition 4, a class that has no explicit Creators part may not redefine *default_create* into a once routine, or inherit *default_create* as a once routine from one of its deferred parents. (Effective parents would themselves violate the condition and hence be invalid.)

End

8.23.7 Definition: Creation procedures of a class

Code: *DKPC*

The **creation procedures** of a class are all the features appearing in any Creation_clause of its unfolded creators part.

Informative Text

8.23.7.1 If there is an explicit Creators part, the creation procedures are the procedures listed there. Otherwise there is only one creation procedure: the class's version of *default_create*.

8.23.7.2 The following property is a consequence of the definitions of "unfolded creators part" and "creation procedures of a class".

End

8.23.8 Principle: Creation Procedure property

Code: *PKCP*

An effective class has at least one creation procedure.

Informative Text

8.23.8.1 This validity property is not a separate constraint but follows from the rules on creation.

8.23.8.2 The creation procedures of an effective class are those explicitly listed if any, otherwise *default_create*.

End

8.23.9 Definition: Creation procedures of a type

Code: *DKPT*

The **creation procedures** of a type *T* are:

1 If *T* is a Formal_generic_name, the constraining creators for *T*.

2 Otherwise, the creation procedures of *T*'s base class.

Informative Text

8.23.9.1 The definition of case 2 is not good enough for case 1, because in the scheme **class** *D [G –> CONST* **create** *cp1, cp2,* … **end***]* it would give us, as creation procedures of *G*, the creation procedures of *CONST*, and what we want is something else: the set of procedures *cp1, cp2,* … specifically listed after *CONST* — the "*constraining creators for G*". These are indeed procedures of *CONST*, but they are not necessarily *creation* procedures of *CONST*, especially since *CONST* can be deferred. What matters is that they must be creation procedures in any instantiatable descendant of *CONST* used as actual generic parameter for *G*.

End

8.23.10 Definition: Available for creation; general creation procedure

Code: *DKAV*

A creation procedure of a class *C*, listed in a Creation_clause *cc* of *C*'s unfolded creators part, is **available for creation** to the descendants of the classes given in the Clients restriction of *cc*, if present, and otherwise to all classes.

If there is no Clients restriction, the procedure is said to be a **general creation procedure**.

8.23.11 Syntax: Creation instructions

Code: *SKIN*

Creation_instruction ≜ **create** [Explicit_creation_type] Creation_call

Explicit_creation_type ≜ "**{**" Type "**}**"

Creation_call ≜ Variable [Explicit_creation_call]

Explicit_creation_call ≜ "**.**" Unqualified_call

8.23.12 Definition: Creation target, creation type of a creation instruction

Code: *DKTT*

The **creation target** (or just "target" if there is no ambiguity) of a Creation_instruction is the Variable of its Creation_call.

The **creation type** of a creation instruction, denoting the type of the object to be created, is:

- The Explicit_creation_type appearing (between braces) in the instruction, if present.
- Otherwise, the type of the creation target.

Informative Text

8.23.12.1 The definition of the creation type reflects the notion of "target type" for calls; see DUTT, page 202. (Also for agents: DPTT, page 212.)

8.23.12.2 The term "target" is also used for reattachments, meaning assignments and argument passing (DBRE, page 239), and for calls (SUTA, page 201). In the present discussion of creation, "target" means "creation target". In any case, for a creation instruction including a Creation_call, such as **create** *x*.*make* (*a*), the creation target, *x*, is also the target (in the sense of call target) of its Creation_call part, *x*.*make* (*a*).

End

8.23.13 Principle: Creation Type theorem

Code: *PKCT*

The creation type of a creation instruction is always effective.

Informative Text

8.23.13.1 This validity property is not a separate constraint but follows from the rules on creation.

End

8.23.14 Definition: Unfolded form of a creation instruction

Code: *DKUI*

Consider a Creation_instruction *ci* of creation type *CT*. The **unfolded form** of *ci* is a creation instruction defined as:

1 If *ci* has an Explicit_creation_call, then *ci* itself.

2 Otherwise, a Creation_instruction obtained from *ci* by making the Creation_call explicit, using as feature name the final name in *CT* of *CT*'s version of *ANY*'s *default_create*.

8.23.15 Definition: Static and dynamic features of a creation call

Code: *DKCF*

The **static feature** and **dynamic feature** of a Creation_call are (respectively) the static feature and dynamic feature of the Unqualified_call of the Explicit_creation_call in its unfolded form.

Informative Text

8.23.15.1 For example, in **create** *x*.*cp* (…), the static feature is *cp* and the dynamic feature is the version of *cp* for the dynamic (run-time) type of *x*.

End

Signed by JV - 2026/10/8

8.23.16 Validity: Creation Instruction rule

Code: *VKIN*

A Creation_instruction of creation type *CT*, appearing in a class *C*, is valid if and only if it satisfies the following conditions:

1 *CT* conforms to the creation target's type.

2 The static feature of the Creation_call of the instruction's unfolded form is available for creation to *C*.

3 That Creation_call is argument-valid.

4 *CT* is generic-creation-ready.

Informative Text

8.23.16.1 In spite of its compactness, the Creation Instruction rule suffices to capture all properties of creation instructions thanks to the auxiliary definitions of "*creation type*", "*unfolded form*" of both a Creation_instruction and a Creators part, "*available for creation*" and others. The rule captures in particular the following cases:

8.23.16.1.1 • The procedure-less form **create** *x* is valid only if *CT*'s version of *default_create* is available for creation to *C*; this is because in this case the unfolded form of the instruction is **create** *x*.*dc_name*, where *dc_name* is *CT*'s name for *default_create*. On *CT*'s side the condition implies that there is either no Creators part (so that *CT*'s own unfolded form lists *dc_name* as creation procedure), or that it has one making it available for creation to *C* (through a Creation_clausewith either no Clients specification or one that lists an ancestor of *C*).

8.23.16.1.2 • If *CT* is a Formal_generic_name, its creation procedures are those listed in the **create** subclause after the constraint. So **create** *x* is valid if and only if the local version of *default_create* is one of them, and **create** *x*.*cp* (…) only if *cp* is one of them.

8.23.16.1.3 • If *CT* is generically derived, and its base class needs to perform creation operations on targets of some of the formal generic types, the last condition (generic-creation readiness) ensures that the corresponding actual parameters are equipped with appropriate creation procedures.

8.23.16.2 The very brevity of this rule may make it less suitable for one of the applications of validity constraints: enabling compilers to produce precise diagnostics in case of errors. For this reason a complementary rule, conceptually redundant since it follows from the Creation Instruction rule, but providing a more explicit view, appears next. It is stated in "*only if*" style rather than the usual "*if and only if*" of other validity rules, since it limits itself to a set of necessary validity conditions.

End

8.23.17 Validity: Creation Instruction properties

Code: *VKIP*

A Creation_instruction *ci* of creation type *CT*, appearing in a class *C*, is valid only if it satisfies the following conditions, assuming *CT* is not a Formal_generic_name and calling *BCT* the base class of *CT* and *dc* the version of *ANY*'s *default_create* in *BCT*:

1 *BCT* is an effective class.

2 If *ci* includes a Type part, the type it lists (which is *CT*) conforms_to the type of the instruction's creation target.

3 If *ci* has no Creation_call, then *BCT* either has no Creators part or has one that lists *dc* as one of the procedures available for creation to *C*.

4 If *BCT* has a Creators part which doesn't list *dc*, then *ci* has a Creation_call.

5 If *ci* has a Creation_call whose feature *f* is not *dc*, then *BCT* has a Creators part which lists *f* as one of the procedures available for creation to *C*.

6 If *ci* has a Creation_call, that call is argument-valid.

If *CT* is a Formal_generic_name, the instruction is valid only if it satisfies the following conditions:

1 *CT* denotes a constrained generic parameter.

Signed by JV - 2026/10/8

2 The Constraint for *CT* specifies one or more procedures as constraining creators.

3 If *ci* has no Creation_call, one of the constraining creators is the Constraint's version of *default_create* from *ANY*.

4 If *ci* has a Creation_call, one of the constraining creators is the static feature of the Creation_call.

Informative Text

8.23.17.1 Compiler writers may refer, in error messages, to either these "Creation Instruction Properties" or the earlier "Creation Instruction rule" (VKIN, page 235) of which they are consequences. For the language definition, **the official rule is the Creation Instruction rule**, which provides a necessary and sufficient set of validity conditions.

End

8.23.18 Semantics: Creation instruction Semantics

Code: *MKIN*

The effect of a creation instruction of creation target *x* and creation type *TC* is the effect of the following sequence of steps, in order:

1 If there is not enough memory available for a new direct instance of *TC*, trigger an exception of type *NO_MORE_MEMORY* in the routine that attempted to execute the instruction. The remaining steps do not apply in this case.

2 Create a new direct instance of *TC*, with reference semantics if *CT* is a reference type and copy semantics if *CT* is an expanded type.

3 Call, on the resulting object, the dynamic feature of the Unqualified_call of the instruction's unfolded form.

4 Attach *x* to the object.

Informative Text

8.23.18.1 The rules requires the *effect* described by this sequence of steps; it does not require that the implementation literally carry out the steps. In particular, if the creation target is expanded and has already been set to an object value, the implementation (in the absence of cycles in the client relation between expanded classes) may **not have to allocate new memory**; instead, it may be able simply to reuse the memory previously allocated to that object. (Because only expanded types conform to an expanded type, no references may exist to the previous object, and hence it is not necessary to preserve its value.) In that case, there will always at step 1 be "enough memory available for a new direct instance" — the memory being reused — and so the exception cannot happen.

8.23.18.2 One might expect, between steps 2 and 3, a step of *default initialization* of the fields of the new object, since this is the intuitive semantics of the language: integers initialized to zero, detachable references to void etc. There is no need, however, for such a step since the Variable Semantics rule implies that an attribute or other variable, unless previously set by an explicit attachment, is automatically set on first access. The rule implies for example that an integer field will be set to zero. More generally, the semantics of the language guarantees that in every run-time circumstance any object field and local variable, even if never explicitly assigned to yet, always has a well-defined value when the computation needs it.

8.23.18.3 About step 3, remember that the notion of "unfolded form" allows us to consider that every creation instruction has an Unqualified_call; in the procedure-less form **create** *x*, this is a call to *default_create*.

8.23.18.4 Also note the order of steps: attachment to the creation target *x* is the last operation. Until then, *x* retains its earlier value, void if *x* is a previously unattached reference.

8.23.18.5 In step 2, “not enough memory available” is a precise notion (the definition appears below); it means that even after possible *garbage collection* the memory available for the system’s execution is not sufficient for the requested object creation.

End

8.23.19 Syntax: Creation expressions

Code: *SKEX*

Creation_expression ≜ **create** Explicit_creation_type [Explicit_creation_call]

8.23.20 Definition: Creation type, unfolded form of a creation expression

Code: *DKEX*

The **creation type** and **unfolded form** of a creation expression are defined as for a creation instruction.

Informative Text

8.23.20.1 Unlike a creation instruction, a creation expressions has no “creation target”. The other properties (creation type — always given explicitly, as in *{T}.creator (x)* — and unfolded form) are the same.

End

8.23.21 Validity: Creation Expression rule

Code: *VKEX*

A Creation_expression of creation type *CT*, appearing in a class *C*, is valid if and only if it satisfies the following conditions:

1 The static feature of the Creation_call of the expression’s unfolded form is available for creation to *C*.

2 That Creation_call is argument-valid.

3 *CT* is generic-creation-ready.

8.23.22 Validity: Creation Expression Properties

Code: *VKEP*

A Creation_expression *ce* of creation type *CT*, appearing in a class *C*, is valid only if it satisfies the following conditions, assuming *CT* is not a Formal_generic_name and calling *BCT* the base class of *CT* and *dc* the version of *ANY*’s *default_create* in *BCT*:

1 *BCT* is an effective class.

2 If *ce* has no Explicit_creation_call, then *BCT* either has no Creators part or has one that lists *dc* as one of the procedures available for creation to *C*.

3 If *BCT* has a Creators part which doesn’t list *dc*, then *ce* has an Explicit_creation_call.

4 If *ce* has an Explicit_creation_call whose feature *f* is not *dc*, then *BCT* has a Creators part which lists *f* as one of the procedures available for creation to *C*.

5 If *ce* has an Explicit_creation_call, that call is argument-valid.

If *CT* is a Formal_generic_name, the expression is valid only if it satisfies the following conditions:

1 *CT* denotes a constrained generic parameter.

2 The Constraint for *CT* specifies one or more procedures as constraining creators.

3 If *ce* has no Creation_call, one of the constraining creators is the Constraint’s version of *default_create* from *ANY*.

4 If *ce* has a Creation_call, one of the constraining creators is the static feature of the Creation_call.

Signed by JV - 2026/10/8

Informative Text

8.23.22.1 As with the corresponding “Creation Instruction Properties”, this is not an independent rule but a set of properties following from previous constraints, expressed with more detailed requirements that may be useful for error reporting by compilers.

End

8.23.23 Semantics: Creation Expression Semantics

Code: *MKEX*

The value of a creation expression of creation type *TC* is — except if step 1 below triggers an exception, in which case the expression has no value — a value attached to a new object as can be obtained through the following sequence of steps:

1 If there is not enough memory available for a new direct instance of *TC*, trigger an exception of type *NO_MORE_MEMORY* in the routine that attempted to execute the expression. In this case the expression has no value and the remaining steps do not apply.

2 Create a new direct instance of *TC*, with reference semantics if *CT* is a reference type and copy semantics if *CT* is an expanded type.

3 Call, on the resulting object, the dynamic feature of the Unqualified_call of the expression’s unfolded form.

Informative Text

8.23.23.1 The notes appearing after the Creation Instruction Semantics rule also apply here.

End

8.23.24 Definition: Garbage Collection, not enough memory available

Code: *DKGC*

Authors of Eiffel implementation are required to provide **garbage collection**, defined as a mechanism that can reuse for allocating new objects the memory occupied by unreachable objects, guaranteeing the following two properties:

1 *Consistency*: the garbage collector never reclaims an object unless it is unreachable.

2 *Completeness*: no allocation request for an object of a certain size *s* will fail if there exists an unreachable object of size >= *s*.

Not enough memory available for a certain size *s* means that even after possible application of the garbage collection mechanism the memory available to the program is not sufficient for allocating an object of size *s*.

8.24 Attaching values to entities

Informative Text

8.24.0.1 At any instant of a system's execution, every entity of the system has a certain **attachment status**, one of:

- Attached to a certain object.
- Void (attached to no object).

8.24.0.2 ===== NO LONGER TRUE, FIX ====== Initially, all entities of reference types are void; one of the effects of a Creation_instruction is to attach its creation target to an object.

8.24.0.3 The attachment status of an entity may change one or more times during system execution through a **attachment** operations, in particular:

- The association of an actual argument of a routine to the corresponding formal argument at the time of a call.
- The Assignment instruction, which may attach an entity to a new object, or remove the attachment.

8.24.0.4 The validity and semantic properties of these two mechanisms are essentially the same; we study them jointly here.

End

8.24.1 Definition: Reattachment, source, target

Code: *DBRE*

A **reattachment** operation is one of:

1. An Assignment *x* := *y*; then y is the attachment's **source** and x its **target**.
2. The run-time association, during the execution of a routine call, of an actual argument (the source) to the corresponding formal argument (the **target** in this case).

Informative Text

8.24.1.1 We group assignment and argument passing into the same category, reattachment, because their validity and semantics are essentially the same:

8.24.1.1.1
- Validity in both cases is governed by the type system: the source must conform to the target's type, or at least convert to it. The Conversion principle guarantees that these two cases are exclusive.

8.24.1.1.2
- The semantics in both cases is to attach the target to the value of the source or a copy of that value.

8.24.1.2 Terminology: the term "target" is also used for qualified calls (SUTA, page 201), and for creation instructions (DKTT, page 234).

End

8.24.2 Syntax: Assignments

Code: *SBAS*

Assignment $\triangleq$ Variable ":=" Expression

8.24.3 Validity: Assignment rule

Code: *VBAS*

An Assignment is valid if and only if its source Expression is compatible with its target variable.

Informative Text

8.24.3.1 To be "compatible" means to conform or convert.

Signed by JV - 2026/10/8

8.24.3.2 This also applies to actual-formal association: the actual argument in a call must conform or convert to the formal argument. The applicable rule is **argument validity** (VUAR, page 204), part of the general discussion of call validity.

End

8.24.4 Semantics: Reattachment principle

Code: *MBRP*

After a reattachement to a target entity *t* of type *TT*, the object attached to *t*, if any, is of a type conforming to *TT*.

8.24.5 Definition: Attaching an entity, attached entity

Code: *DBAE*

Attaching an entity *e* to an object O is the operation ensuring that the value of *e* becomes attached to **O**.

Informative Text

8.24.5.1 Although it may seem tautological at first, this definition simply relates the two terms “attach”, denoting an operation that can change an entity, and “attached to an object”, denoting the state of such an entity — as determined by such operations. These are key concepts of the language since:

8.24.5.1.1 • A reattachment operation (see next) may “*attach*” its target to a certain object as defined by the semantic rule; a creation operation creates an object and similarly “*attaches*” its creation target to that object.

8.24.5.1.2 • Evaluation of an entity, per the Entity Semantics rule, uses (partly directly, partly by depending on the Variable Semantics rule and through it on the definition of “value of a variable setting”) the object *attached* to that entity. This is only possible by ensuring, through other rules, that prior to any such attempt on a specific entity there will have been operations to “attach” the entity or make it void.

8.24.5.2 If the entity is of an expanded type, it will always be attached at run time. If it is of a reference type, its value, a reference, may be “attached” or “void” per DERV, page 243. The entity is attached if and only if the reference is.

End

8.24.6 Semantics: Reattachment semantics

Code: *MBRE*

The effect of a reattachment of source expression *source* and target entity *target* is the effect of the first of the following steps whose condition applies:

1 If *source* converts to *target*: perform a conversion attachment from *source* to *target*.

2 If the value of *source* is a void reference: make *target*’s value void as well.

3 If the value of *source* is attached to an object with copy semantics: create a clone of that object, if possible, and attach *target* to it.

4 If the value of *source* is attached to an object with reference semantics: attach *target* to that object.

Informative Text

8.24.6.1 As with other semantic rules describing the “effect” of a sequence of steps, only that effect counts, not the exact means employed to achieve it. In particular, the creation of a clone in step 3 is — as also noted in the discussion of creation — often avoidable in practice if the target is expanded and already initialized, so that the instruction can reuse the memory of the previous object.

8.24.6.2 Case 1 indicates that a conversion, if applicable, overrides all other possibilities. In those other cases, if follows from the Assignment rule that *source* must conform (in the sense of expressions) to *target*.

Signed by JV - 2026/10/8

8.24.6.3 Case 2 is, from the validity rules, possible only if both *target* and *source* are declared of *detachable* types.

8.24.6.4 In case 3, a "clone" of an object is obtained by application of the function *cloned* from *ANY*; expression conformance ensures that *cloned* is available (exported) to the type of *target*; otherwise, cloning could produce an inconsistent object.

8.24.6.5 The cloning might be impossible for lack of memory, in which case the semantics of the cloning operation specifies triggering an exception, of type *NO_MORE_MEMORY*. As usual with exceptions, the rest of case 3 does not then apply.

8.24.6.6 In case 4 we simply reattach a reference. Because of the validity rules (no reference type conforms to an expanded type), the target must indeed be of a reference type.

8.24.6.7 This rule defines the *effect* of a construct through a sequence of cases, looking for the first one that matches. As usual with semantic rules, this only specifies the result, but does not imply that the implementation must try all of them in order.

End

8.24.7 Semantics: Assignment semantics

Code: *MBAS*

The effect of an assignment *x* := *y* is determined by the Reattachment Semantics rule, with source *y* and target *x*.

Informative Text

8.24.7.1 The other cases where Reattachment Semantics applies is actual-formal association, per step 5 of the General Call semantics rule (MUGC, page 209).

8.24.7.2 On the other hand, the semantics of Object_test, a construct which also allows a Read_only entity to denote the same value as an expression, is simple enough that it does not need to refer to reattachment.

End

8.24.8 Definition: Dynamic type

Code: *DBDT*

The **dynamic type** of an expression *x*, at some instant of execution, is the type of the object to which *x* is attached, or *NONE* if *x* is void.

8.24.9 Definition: Polymorphic; polymorphism; dynamic type and class sets

Code: *DBDS*

An expression that has two or more possible dynamic types is said to be **polymorphic**. This possibility is called **polymorphism**.

The set of possible dynamic types for an expression *x* is called the **dynamic type set** of *x*. The set of base classes of these types is called the **dynamic class set** of *x*.

8.24.10 Syntax: Assigner calls

Code: *SBAC*

Assigner_call ≜ Expression ":=" Expression

Informative Text

8.24.10.1 What appears to the left of the ":=" is is surprisingly general: any expression. The following validity rule (VBAC) will constrain it to be of a form that can be interpreted as a qualified call to a query, such as *x*.*a*, or *x*.*f* (*i*, *j*); but the syntactic form can be different, using for example bracket syntax as in *a* [*i*, *j*] := *x*.

8.24.10.2 Operator syntax (as in *a* + *b* := *c*) would be possible in principle; for the sake of simplicity, however, it is not permitted.

End

8.24.11 Validity: Assigner Call rule

Code: *VBAC*

An Assigner_call of the form *target* := *source*, where *target* and *source* are expressions, is valid if and only if it satisfies the following conditions:

1 *source* is compatible with *target*.

2 *target* is a targeted Call, a Bracket_expression or a Paren_expression such that the static feature of its Equivalent Dot Form has an assigner command.

8.24.12 Semantics: Assigner Call semantics

Code: *MBAC*

The effect of an Assigner_call *target* := *source*, where the Equivalent Dot Form of *target* is *x*.*f* or *x*.*f* (*args*) and *f* has an assigner command *p*, is, respectively, *x*.*p* (*source*) or *x*.*p* (*source*, *args*).

Informative Text

8.24.12.1 This rule confirms that the construct is just an abbreviation for a procedure call.

End

Signed by JV - 2026/10/8

8.25 Objects, values and entities

Informative Text

8.25.0.1 The execution of an Eiffel system consists of creating, accessing and modifying **objects**.

8.25.0.2 The following presentation discusses the structure of objects and how they relate to the syntactical constructs that denote objects in software texts: **expressions**. At run time, an expression may take on various *values*; every value is either an object or a reference to an object.

8.25.0.3 Among expressions, **entities** play a particular role. An entity is an identifier (name in the software text), meant at execution time to denote possible values. Some entities are **read-only**: the execution can't change their initial value. Others, called **variables**, can take on successive values during execution as a result of such operations as creation and assignment.

8.25.0.4 The description of objects and their properties introduces the *dynamic model* of Eiffel software execution: the run-time structures of the data manipulated by an Eiffel system.

End

8.25.1 Semantics: Object, type, generating type and class of an object; generator

Code: *MEGE*

During run time, the execution of a system creates and manipulates an **object structure**, a collection of interlinked components called **objects**.

Every run-time object is a direct instance of exactly one stand-alone type of the system, called the **generating type** of the object, or just "the type of the object" if there is no ambiguity.

The base class of the generating type is called the object's **generating class**, or **generator** for short.

Informative Text

8.25.1.1 What is an object? The following definitions and semantic specifications answer this question; they involve recursion since the basic idea is that an *object* (if it is not a "basic" object such as an integer or real number) is made of a sequence of *fields*, each of which is a *value*, but then a value itself is an *object* or a *reference* to an object.

8.25.1.2 As a result the definitions of these three concepts of object, value and reference are closely interlinked, together with the definitions of associated concepts.

End

8.25.2 Definition: Value

Code: *DEVA*

During run time, a **value** is either a reference or an object.

Informative Text

8.25.2.1 An object, as defined above, is a collection of fields.

8.25.2.2 A reference, as defined next, either is "void" or **identifies** an object.

8.25.2.3 A field of an object is itself a value, that is to say, again either an object (a "sub-object" of the enclosing object) or a reference. So what we are dealing with at run time, called above the "object structure", is a collection of objects, linked to each other by references.

8.25.2.4 An earlier definition, MTDI, page 165, defined a specific case of the notion of "value": the values *of a type*.

End

8.25.3 Definition: Reference, void, attached, attached to

Code: *DERV*

A **reference** is a value that is either:

- **Void**, in which case it provides no more information.

- **Attached**, in which case it gives access to an object. The reference is said to be **attached to** that object, and the object attached to the reference.

Informative Text

8.25.3.1 "Attached" is also defined for entities (DBAE, page 240). An entity of reference type is, at some run-time instant, "attached" if its value, a reference, is attached in the sense of the present definition".

8.25.3.2 The term "attached to" actually denotes, per this definition two relations: from a reference to an object, and from that object back to the reference. We need not fear ambiguity since it will always be clear whether one is talking about a reference or about an object.

End

8.25.4 Semantics: Object principle

Code: *MEOP*

Every non-void value is either an object or a reference attached to an object. In both cases the value is said to be **attached to** the corresponding object.

Informative Text

8.25.4.1 The Object principle introduces no new concept since it follows from the preceding definitions, but it brings in a new variant of the notion of being "attached to" an object: now this phrase also applies to a value, in all cases except if the value is a void reference.

End

8.25.5 Definition: Copy and reference semantics

Code: *DECR*

Every run-time object has either **copy semantics** or **reference semantics**.

An object has copy semantics if and only if its generating type is an expanded type.

Informative Text

8.25.5.1 This property determines the role of the object when used as source of an assignment: with copy semantics, it will be copied onto the target; with reference semantics, a reference will be reattached to it.

End

8.25.6 Definition: Non-basic class, non-basic type

Code: *DENB*

Any class other than the basic types is said to be a **non-basic class**. Any type whose base class is non-basic is a **non-basic type**, and its instances are **non-basic objects**.

8.25.7 Definition: Field

Code: *DEFI*

If *T* is a Type, a **field** of type *T* is a value of type *T*.

8.25.8 Semantics: Object semantics

Code: *MEOB*

A direct instance of a non-basic type is a sequence of zero or more fields: one for every attribute of the type's base class, and with the same type as that attribute.

Informative Text

8.25.8.1 This rule defines the fundamental data structure on which the execution of an Eiffel program relies: a set of objects, each made of fields. According to the Object principle, each one of these fields is either itself an object (which can be either a simple value, such as an integer or boolean, or a subobject, as discussed next) or a reference to an object.

8.25.8.2 The rule only defines direct instances of non-basic types. Basic types have their own separate discussion (section XXXX), which for each of them specifies the direct instances.

End

8.25.9 Definition: Subobject, composite object

Code: *DESO*

Any expanded field of an object is a **subobject** of that object.

An object that has a non-basic subobject is said to be **composite**.

8.25.10 Definition: Entity, variable, read-only

Code: *DEEN*

An **entity** is an Identifier, or one of three reserved words (**Current**, **Result** and **Retry**), used in one of the following roles:

1 Final name of an attribute of a class.

2 Local variables of a routine or Inline_agent, including **Result** and **Retry**.

3 Formal argument of a routine or inline agent.

4 Object-test local.

5 **Current**, the predefined entity used to represent a reference to the current object (the target of the latest not yet completed routine call).

Names of non-constant attributes and local variables are **variable** entities, also called just **variables**. Constant_attributes, formal arguments, object-test_locals and **Current** are **read-only** entities.

Informative Text

8.25.10.1 Two kinds of operation, creation and reattachment, may modify the value of a variable (a non-constant attribute, part of category 1, or local variable, category 2. In the other four cases — constant attributes, formal arguments (3), object-test locals (4) and **Current** (5) — you may not directly modify the entities, hence the name *read-only* entity.

8.25.10.2 The term "*constant* entity" wouldn't do, not so much because you can modify the corresponding objects but because read-only entities (other than constant attributes) do change at run time: a qualified call reattaches **Current**, and any routine call reattaches the formal arguments.

8.25.10.3 **Result** appearing in the Postcondition of a constant attribute cannot be changed at execution time, but for simplicity is considered part of local variables in all cases anyway.

End

8.25.11 Syntax: Entities and variables

Code: *SEEN*

Entity ≜ Variable | Read_only

Variable ≜ Variable_attribute | Local

Variable_attribute ≜ Feature_name

Local ≜ Identifier | **Result** | **Retry**

Read_only ≜ Formal | Constant_name | **Current**

Formal ≜ Identifier

8.25.12 Validity: Entity rule

Code: *VEEN*

An occurrence of an entity *e* in the text of a class *C* (other than as the static feature of a qualified call) is valid if and only if it satisfies one of the following conditions:

1 *e* is **Current**.

2 *e* is the final name of an attribute of *C*.

Signed by JV - 2026/10/8

3 *e* is the local variable **Result**, and the occurrence is in a Feature_body, Postcondition or Rescue part of an Feature_definition text for a query or an Inline_agent whose signature includes a result type.

4 *e* is **Result** appearing in the Postcondition of a constant attribute's declaration.

5 *e* is the local variable **Retry** appearing in a Feature_body or Rescue part of a feature or Inline_agent.

6 *e* is listed in the Entity_list of an Entity_declaration_group in a Local_declarations part of a feature or Inline_agent *fa*, and the occurrence is in a Local_declarations, Feature_body or Rescue part for *fa*.

7 *e* is listed in the Entity_list of an Entity_declaration_group in a Formal_arguments part for a routine *r*, and the occurrence is in a declaration for *r*.

8 ~~*e* is listed in the Entity_list of an Entity_declaration_group in the Agent__arguments part of a Call_agent *a*, and the occurrence is in the Agent__body of *a*.~~ TO BE CHECKED

9 *e* is the object-test local of an Object_test, and the occurrence is in its scope.

Informative Text

8.25.12.1 This VEEN rule is well known to Eiffel programmers — although not necessarily in a kind way — since it figures in compiler error messages that you will receive in many cases of using a non-recognized entity, for example if you have misspelled a variable. The compiler does not recognize the name as denoting one of the legitimate (valid) cases listed above, and hence can do little better than to ask you to check the name and refer you to these cases as stipulated in VEEN.

8.25.12.2 "Other than as feature of a qualified call" excludes from the rule any attribute, possibly of another class, used as feature of a qualified call: in *a*.*b* the rule applies to *a* but not to *b*.

8.25.12.3 For the feature of a qualified call, such as *b* in *a*.*b*, the applicable constraint is the General Call semantics rule (MUGC, page 209), requiring *b* to be the name of a feature in *D*'s base class.

End

8.25.13 Validity: Variable rule

Code: *VEVA*

A Variable entity *v* is valid in a class *C* if an only if it satisfies one of the following conditions:

1 *v* is the final name of a variable attribute of *C*.

2 *v* is the final name of a local variable of the immediately enclosing routine or agent.

8.25.14 Definition: Self-initializing type

Code: *DEST*

A type is **self-initializing** if it is one of:

1 A detachable type.

2 A self-initializing formal parameter.

3 An attached type (including expanded types and, as a special case of these, basic types) whose creation procedures include a version of *default_create* from *ANY* available for creation to *C*.

Informative Text

8.25.14.1 A self-initializing type enables us to define a default initialization value:

- Use *Void* for a detachable type (case 1, the easiest but also the least interesting)
- Execute a creation instruction with the applicable version of *default_create* for the most interesting case: 3, attached types, including expanded types. This case also covers basic types, which all have a default value given by the following rule.

8.25.14.2 A "self-initializing formal parameter" (case 2, see DGSI, page 175) is a generic parameter, so we don't exactly know which one of these three semantics will apply; but we do require, through the Generic Derivation rule, that any attached type used as actual generic parameter be self-initializing, meaning in this case that it will provide *default_create*.

8.25.14.3 In the definition, the "creation procedures" of a *type* are (DKPT, page 233) the creation procedures of its base class or, for a formal generic parameter, its "constraining creators", the features listed as available for creation in its constraining type.

8.25.14.4 The more directly useful notion is that of a self-initializing *variable*, appearing below.

8.25.14.5 The term "self-initializing" is justified by the following semantic rule, specifying the actual initialization values for every self-initializing type.

End

8.25.15 Semantics: Default Initialization rule

Code: *MEDI*

Every self-initializing type *T* has a **default initialization value** as follows:

1 For a detachable type: a void reference.
2 For a self-initializing attached type: an object obtained by creating an instance of *T* through *default_create*.
3 For a self-initializing formal parameter: for every generic derivation, (recursively) the default initialization value of the corresponding actual generic parameter.
4 For *BOOLEAN*: the boolean value false.
5 For a sized variant of *CHARACTER*: null character.
6 For a sized variant of *INTEGER*: integer zero.
7 For a sized variant of *REAL*: floating-point zero.
8 For *POINTER*: a null pointer.
9 For *TYPED_POINTER*: an object representing a null pointer.

Informative Text

8.25.15.1 This rule is the reason why everyone loves self-initializing types: whenever execution catches an entity that hasn't been explicitly set, it can (and, thanks to the Entity Semantics rule, will) set it to a well-defined default value. This idea gains extra flexibility, in the next definition, through the notion of attributes with an explicit initialization.

End

8.25.16 Definition: Self-initializing variable

Code: *DESV*

A variable is **self-initializing** if one of the following holds:

1 Its type is a self-initializing type.
2 It is an attribute declared with an Defined_attribute part such that the entity **Result** is properly set at the end of its Compound.

Informative Text

8.25.16.1 If a variable is self-initializing, we don't need to worry about finding it with an undefined value at execution time: if it has not yet been the target of an attachment operation, automatic initialization can take over and set it to a well-defined default value. That value is, in case 1, the default value for its type, and in case 2 the result of the attribute's own initialization. That initialization must ensure that **Result** is "properly set" as defined next (which is in part mutually recursive with the above definition) .

End

8.25.17 Definition: Evaluation position

Code: *DEEP*

An **evaluation position** is one of:

- In a Compound, one of its Instruction components.
- In an Assertion, one of its Assertion_clause components.
- In either case, the **start position** or **end position**.

Informative Text

8.25.17.1 The notion of evaluation position is needed to ensure that entities are properly set before use.

8.25.17.2 In a compound *i1; i2; i3* we have five positions including the start and end positions.

End

8.25.18 Definition: Properly set variable

Code: *DEPS*

The set of **properly set** variables contains, for a start position, the self-initializing attributes of the class, and at any other evaluation position following an instruction i is S , i as defined by the following equations, S being the set of properly set variables at the evaluation position before i:

#	Case	Value	Comments
1	S , (*var* := *exp*)	$S \cup var$ if *exp* $\in S$ or *exp* is attached $S - \{x\}$ otherwise	Assignment ("—" is set difference)
2	S , (**create** *var*)	$S \cup var$	
3	S , (p ; q)	(S , p), q	
4	S , (**if** c **then** p **else** q **end**)	(S , (**check** c **end** ; p)) $\cap$ (S , (**check** c **end** ; p))	
5	S , (**from** p **until** c **loop** q **end**)	(Lim A_i) , (**check** c **end**) where $A_0 = S$, q $A_{i+1} = A_i \cap (A_i$, (**check not** c **end** ; p))	Non-increasing sequence in finite set: fixpoint exists
6	S , (**inspect** *exp* … **when** v_i **then** p_i … **else** p_0 **end**)	$\bigcap_i$ (S , p_i)	$\cap$ covers all i including 0
7	S , (**check** c **then** p **end**)	(S , (**check** c **end**)) , p	
8	S , (**check** *var* /= **Void end**) S , (**check not** (*var* = **Void**) **end**) S , (**require** *var* /= **Void**) S , (**require not** (*var* = **Void**))	$S \cup var$	
9	S , (**check attached** *var* **and then** c **end**) S , (**require attached** *var* **and then** c)	($S \cup var$) , (**check** c **end**)	
10	S , (**check** c **end**) S , (**require** c)	S	For c other than above cases

Signed by JV - 2026/10/8

#	Case	Value	Comments
11	*S* , (“ *call r*”)	**check False end** if *r* has postcondition **False** *S* — *NS* otherwise where *NS* is the set of non-emergent detachable attributes	Call, qualified or not, to routine *r*

Informative Text

8.25.18.1 Case 1 .

End

8.25.19 Definition: Variable setting and variable value

Code: *DEVS*

A **setting** for a variable *x* is any one of the following run-time events, defining in each case the **value** of the setting:

1 Execution of a creation instruction of creation target *x* or an assignment of assignment target *x*. (*Value*: the object attached to *x* by the instruction, or a void reference if none.)

2 If *x* is a variable attribute with an Defined_attribute part: evaluation of that part, implying execution of its Compound. (*Value*: the object attached to **Result** at the end position of that Compound, or a void reference if none.)

3 If the type *T* of *x* is self-initializing: assignment to *x* of *T*'s default initialization value. (*Value*: that initialization value.)

Informative Text

8.25.19.1 As a consequence of case 2, an attribute *a* that is self-initializing through an Defined_attribute part *ap* is *not* set until execution of *ap* has reached its end position. In particular, it is not invalid (although definitely unusual and perhaps strange) for the instructions *ap* to use the value *a*: as with a recursive call in a routine, this will start the computation again at the beginning of *ap*. For attributes as for routines, this raises the risk of infinite recursion (perhaps higher for attributes since they have no arguments) and it is the programmer's responsibility to avoid this by ensuring that before a recursive call the context will have sufficiently changed to ensure eventual termination. No language rule can ensure this (in either the routine or attribute cases) since this would amount to solving the “halting problem”, a provably impossible task.

8.25.19.2 Another consequence of the same observation is that if the execution of *ap* triggers an exception, and hence does not reach its end position, any later attempt to access *a* will also restart the execution of *ap* from the beginning. This might trigger the same exception, or succeed if the conditions of the execution have changed.

End

8.25.20 Definition: Execution context

Code: *DEEC*

At any time during execution, the current **execution context** for a variable is the period elapsed since:

1 For an attribute: the creation of the current object.

2 For a local variable: the start of execution of the current routine.

8.25.21 Semantics: Variable semantics

Code: *MEVS*

The value produced by the run-time evaluation of a variable *x* is:

1 If the execution context has previously executed at least one setting for *x*: the value of the latest such setting.

Signed by JV - 2026/10/8

2 Otherwise, if the type *T* of *x* is self-initializing: assignment to *x* of *T*'s default initialization value, causing a setting of *x*.

3 Otherwise, if *x* is a variable attribute with an Defined_attribute part: evaluation of that part, implying execution of its Compound and hence a setting for *x*.

4 Otherwise, if *x* is **Result** in the Postcondition of a constant attribute: the value of the attribute.

Informative Text

8.25.21.1 This rule is phrased so that the order of the first three cases is significant: if there's already been an assignment, no self-initialization is possible; and if *T* has a default value, the Defined_attribute part won't be used.

8.25.21.2 The Variable Initialization rule ensures that one of these cases will apply, so that *x* will always have a well-defined result for evaluation. This property was our main goal, and its achievement concludes the discussion of variable semantics.

End

8.25.22 Semantics: Entity Semantics rule

Code: *MEEN*

Evaluating an entity yields a **value** as follows:

1 For **Current**: if C is expanded, the current object; if not, a reference attached to the current object.

2 For a formal argument of a routine or Inline_agent: the value of the corresponding actual at the time of the current call.

3 For a constant attribute: the value of the associated Manifest_constant as determined by the Manifest Constant Semantics rule.

4 For an object-test local: as determined by the Object-Test Local Semantics rule.

5 For a variable: as determined by the Variable Semantics rule.

Informative Text

8.25.22.1 This rule concludes the semantics of entities by gathering all cases. It serves as one of the cases of the semantics of expressions, since an entity can be used as one of the forms of Expression.

8.25.22.2 The Object-Test Local Semantics rule appears in the discussion of the Object_test construct.

End

8.26 Comparing and duplicating objects

Informative Text

8.26.0.1 Creation operations (that is to say creation instructions and expressions), reviewed in clause 8.23, page 231, are the basic language mechanisms for obtaining new objects at run time; they produce new direct instances of a given class, initialized from scratch.

8.26.0.2 Sometimes you will need instead to copy the contents of an *existing* object onto those of another. This is the **copying** operation.

8.26.0.3 A variant of copying is **cloning**, which produces a new object by duplicating an existing one.

8.26.0.4 For both copying and cloning, the default variants are “shallow”, affecting only one object, but **deep** versions are available to duplicate an object structure recursively.

8.26.0.5 A closely related problem is that of *comparing* two objects for shallow or deep equality.

8.26.0.6 The copying, cloning and comparison operations rely on only one language construct (the object equality operator **~**) and are entirely defined through language constructs but through routines that developer-defined classes inherit from the universal class ANY. This makes it possible, through feature redefinitions, to adapt the semantics of copying, cloning and comparing objects to the specific properties of any class.

End

8.8.27 Library: Object comparison features from *ANY*

Code: *L1CD*

The features whose contract views appear below are provided by class *ANY*.

```
default_is_equal (other: like Current)
                -- Is other attached to object field-by-field equal
                -- to current object?
        ensure
                same_type: Result implies same_type (other)
                symmetric: Result = other.default_is_equal (Current)
                consistent: Result implies is_equal (other)
is_equal (other: detachable like Current)
                -- Is other attached to object considered equal
                -- to current object?
        ensure
                same_type: Result implies same_type (other)
                symmetric: Result = other.is_equal (Current)
                consistent: default_is_equal (other) implies Result
```

The original version of *is_equal* in *ANY* has the same effect as *default_is_equal*.

Informative Text

8.27.0.1 These are the two basic object comparison operations. The difference is that *default_is_equal* is frozen, always returning the value of field-by-field identity comparison (for non-void *other*); any class may, on the other hand, redefine *is_equal*, in accordance with the pre- and postcondition, to reflect a more specific notion of equality.

8.27.0.2 Both functions take an argument of an attached type, so there is no need to consider void values.

End

Signed by JV - 2026/10/8

8.27.1 Syntax: Equality expressions

Code: *S1EE*

Equality ≜ Expression Comparison Expression

Comparison ≜ "**=**" | "**/=**" | "**~**" | "**/~**"

8.27.2 Semantics: Equality Expression semantics

Code: *M1EE*

The Boolean_expression *e* **~** *f* has value true if and only if the values of *e* and *f* are attached to objects of the same type and *e*.*is_equal* (*f*) holds.

The Boolean_expression *e* = *f* has value true if and only if the values of *e* and *f* are one of:

1 Both void.

2 Both attached to the same object with reference semantics.

3 Both attached to objects with copy semantics, and such that *e* **~** *f* holds.

Informative Text

8.27.2.1 The form with **~** always denotes object equality. The form with **=** denotes reference equality if applicable, otherwise object equality. Both rely, for object equality, on function *is_equal* — the version that can be redefined locally in any class to account for a programmer-defined notion of object equality adapted to the specific semantics of the class.

8.27.2.2 *e* **~** *f* can only hold if *e* and *f* are both attached (not void), and the types of the corresponding objects are identical. Then they must be equal in the sense of *is_equal*.

8.27.2.3 It is preferable to use **~** rather than calling *is_equal* explicitly, since **~** avoids any type error (potential catcall).

End

8.27.3 Semantics: Inequality Expression semantics

Code: *M1IE*

The expression *e* /= *f* has value true if and only if *e* = *f* has value false.

The expression *e* /~ *f* has value true if and only if *e* ~ *f* has value false.

8.8.28 Library: Copying and cloning features from *ANY*

Code: *L1CC*

The features whose contract views appear below are provided by class *ANY* as secret features.

```
copy (other: detachable like Current)
        -- Update current object using fields of object
        -- attached to other, to yield equal objects.
    require
        exists: other /= Void
        same_type: other.same_type (Current)
    ensure
        equal: is_equal (other)
frozen default_copy (other: detachable like Current)
        -- Update current object using fields of object
        -- attached to other, to yield identical objects.
    require
        exists: other /= Void
        same_type: other.same_type (Current)
    ensure
        equal: default_is_equal (other)
```


Signed by JV - 2026/10/8

```
frozen cloned: like Current
        -- New object equal to current object
        -- (relies on copy)
    ensure
        equal: is_equal (Result)
frozen default_cloned: like Current
        -- New object equal to current object
        -- (relies on default_copy)
    ensure
        equal: default_is_equal (Result)
```

The original versions of *copy* and *cloned* in *ANY* have the same effect as *default_copy* and *default_cloned* respectively.

Informative Text

8.28.0.1 Procedure *copy* is called in the form *x*.*copy* (*y*) and overrides the fields of the object attached to *x*. Function *cloned* is called as *x*.*cloned* and returns a new object, a "clone" of the object attached to *x*. These features can be adapted to a specific notion of copying adapted to any class, as long as they produce a result equal to the source, in the sense of the — also redefinable — function *is_equal*. You only have to redefine *copy*, since *cloned* itself is frozen, with the guarantee that it will follow any redefined version of *copy*; the semantics of *cloned* is to create a new object and apply *copy* to it.

8.28.0.2 In contrast, *default_copy* and *default_cloned*, which produce field-by-field identical copies of an object, are frozen and hence always yield the original semantics as defined in *ANY*.

8.28.0.3 All these features are **secret in their original class** *ANY*. The reason is that exporting copying and cloning may violate the intended semantics of a class, and concretely its invariant. For example the correctness of a class may rely on an invariant property such as

some_circumstance **implies** (*some_attribute* = *Current*)

stating that under *some_circumstance* (a boolean property) the field corresponding to *some_attribute* is cyclic (refers to the current object itself). Copying or cloning an object will usually not preserve such a property. The class should then definitely not export *default_copy* and *default_cloned*, and should not export *copy* and *cloned* unless it redefines *copy* in accordance with this invariant; such redefinition may not be possible or desirable. Because these features are secret by default, software authors must decide, class by class, whether to re-export them.

End

8.8.29 Library: Deep equality, copying and cloning

Code: *L1DE*

The feature *is_deep_equal* of class *ANY* makes it possible to compare object structures recursively; the features *deep_copy* and *deep_cloned* duplicate an object structure recursively.

Informative Text

8.29.0.1 The default versions of the earlier features — *default_is_equal*, *default_copy*, *default_cloned* and the original versions of their non-*default* variants — are "shallow": they compare or copy only one source object. The "deep" versions recursively compare or copy entire object structures.

8.29.0.2 Detailed descriptions of the "deep" features appear in the specification of ELKS.

End

Signed by JV - 2026/10/8

8.30 Void safety

Informative Text

8.30.0.1 In the object-oriented style of programming the basic unit of computation is a qualified feature call

x.f (args)

which applies the feature *f*, with the given arguments *args*, to the object attached to *x*. But *x* can be a reference, and that reference can be void. Then there is *no* object attached to *x*. An attempt to execute the call would fail, triggering an exception.

8.30.0.2 If permitted to occur, void calls are a source of instability and crashes in object-oriented programs. For other potential run-time accidents such as type mismatches, the compilation process spots the errors and refuses to generate executable code until they've all been corrected. Can we do the same for void calls?

8.30.0.3 Eiffel indeed provides a coordinated set of techniques that guarantee the absence of void calls at execution time. The actual rules are specific conditions of more general validity constraints — in particular on attachment and qualified calls — appearing elsewhere; in the following discussion we look at them together from the viewpoint of ensuring their common goal: precluding void calls.

8.30.0.4 The basic idea is simple. Its the combination of three rules:

8.30.0.4.1 • As defined in VUTA, page 204, a qualified call x**.***f (args)* is target-valid — a required part of being plain valid — if the type of *x* is **attached**, "Attached" is here a static property, deduced from the declaration of *x* (or, if it is a complex expression, of its constituents).

8.30.0.4.2 • A reference type with a name in the usual form, *T*, is attached. To obtain a detachable type — meaning that *Void* is a valid value — use **detachable** *T*.

8.30.0.4.3 • The validity rules ensure that attached types — those without a **detachable** mark — deserve their name: an entity declared as *x*: *T* can never take on a void value at execution time. In particular, you may not assign to *x* a detachable value, or if *x* is a formal argument to a routine you may not call it with a detachable actual. (With a detachable target, the other way around, you are free to use an attached or detachable source.)

End

8.30.1 Syntax: Object test

Code: *SVOT*

Object_test ≜ **attached** ["**{**" Type "**}**"] [**as** Identifier] Expression

Informative Text

8.30.1.1 An Object_test of the form **attached** {*T*} *exp* **as** *x*, where *exp* is an expression, *T* is a type and *x* is a name different from those of all entities of the enclosing context, is a boolean-valued expression; its value is true if and only *exp* is attached to an instance of *T* (hence, non-void). In addition, evaluating the expression has the effect of letting *x* denote that value of *exp*, captured at the time of its evaluation, throughout the execution of a neighboring part of the program text known as the **scope** of the Object_test. For example, in **if attached** {*T*} *exp* **as** *x* **then** *c1* **else** *c2* **end** the scope of the Object_test is the compound in the **then** part, *c1*. Within *c1*, you may use *x* as a Read_only entity, knowing that it has the value *exp* had on evaluation of the Object_test, that this value is of type *T*, and that it cannot be changed during the execution of *c1*.

8.30.1.2 The syntax includes two optional elements:

8.30.1.3 The following rules define these notions precisely.

End

8.30.2 Definition: Object-Test Local

Code: *DVOL*

The **object-test local** of an Object_test, defined only if the **as** Identifier component is present, is the listed Identifier.

8.30.3 Validity: Object Test rule

Code: *VVOT*

An Object_test *ot* of the form **attached** {*T*} *exp* **as** *x* (with either or both of the {*T*}and **as** *x* parts possibly absent) is valid if and only if it satisfies the following conditions:

1 If the **as** *x* part is present, *x* is a fresh identifier.

2 If the {*T*} part is present, *T* is an attached type.

Informative Text

8.30.3.1

8.30.3.2 Condition 2 reflects the intent of an Object_test: to test whether the value of an expression is attached to an instance of a given type. It would make no sense then to use a detachable type.

End

8.30.4 Definition: Conjunctive, disjunctive, implicative; term, semistrict term

Code: *DVSS*

Consider an Operator_expression *e* of boolean type, which after resolution of any ambiguities through precedence rules can be expressed as a_1 § a_2 §… § a_n for $n \geq 1$, where § represents boolean operators and every a_i, called a **term**, is itself a valid Boolean_expression. Then *e* is:

- **Conjunctive** if every § is either **and** or **and then**.
- **Disjunctive** if every § is either **or** or **or else**.
- **Implicative** if $n = 2$ and § is **implies**.

A term a_i is **semistrict** if in the corresponding form it is followed by a semistrict operator.

Informative Text

8.30.4.1 The definition of the scope of an object test requires one more auxiliary notion, making the definition immune to simple syntactical differences.

End

8.30.5 Definition: Expression reduction

Code: *DVER*

An expression *e* **reduces to** an expression *f* if it can be turned into *f* by zero or more applications of any of the following transformations:

1 Removal of a parenthesis pair if, according to the rules of operator precedence, this does not affect the semantics of the expression.

2 Replacement of a *a* **§** *b* by *b* **§** *a* where **§** is one of: **=**, **/=**, **~**, **/~**.

Informative Text

8.30.5.1 The following definition of the scope of an object test takes advantage of the just introduced auxiliary definitions.

End

8.30.6 Definition: Scope of an Object Test

Code: *DVOS*

The scope of an Object_test *ot* includes any applicable program element from the following, where *rt* is any expressions to which the Expression of *ot* reduces:

1 If *rt* is a semistrict term of a conjunctive expression: any subsequent terms.

2 If *rt* is a term of an implicative expression: the next term.

3 If **not** *rt* is a semistrict term of a disjunctive expression *e*: any subsequent terms.

4 If *rt* is a term of a conjunctive expression serving as the Boolean_expression in the Then_part in a Conditional_instruction: the corresponding Compound.

5 If **not** *rt* is a term of a disjunctive expression serving as the Boolean_expression in the Then_part in a Conditional_instruction: any subsequent Then_part and Else_part.

6 If **not** *rt* is a term of a disjunctive expression serving as the Exit_condition in a Loop: the Loop_body.

7 If *rt* is a term of a conjunctive expression appearing in the unfolded form of an Unlabeled_assertion_clause in a Precondition: the subsequent components of the Feature_definition.

8 If *rt* is a term of a conjunctive expression used as Unlabeled_assertion_clause in a Check: the subsequent components of its enclosing Compound.

Informative Text

8.30.6.1 The scope of an Object_test is also (from the general definition of scope of an identifier, *D2SC*) the scope of its object-test local.

8.30.6.2 The definition ensures that, for an Object_test **attached** {*T*} *exp* **as** *x*, we can rest assured that, throughout its scope, *x* will never at run time have a void value, and hence can be used as the target of a call.

End

8.30.7 Semantics: Object Test semantics

Code: *MVOT*

The value of an Object_test **attached** {*T*} *exp* **as** *x* (with either or both of the {*T*}and **as** *x* parts possibly absent) is true if and only if the value of *exp* satisfies the following conditions:

1 It is attached to an object.

2 If If the {*T*} part is present, that object is an instance of *T*.

Informative Text

8.30.7.1 In particular, if *exp* is void (which is possible only if *T* is a detachable type), the result will be false.

End

8.30.8 Semantics: Object-Test Local semantics

Code: *MVOL*

For an Object_test **attached** {*T*} *exp* **as** *x* (with the {*T*} part possibly absent), the value of *x*, defined only over its scope, is the value of *exp* at the time of the Object_test's evaluation.

8.30.9 Definition: Read-only void test

Code: *DVRO*

A **read-only void test** is a Boolean_expression of one of the forms *e* = **Void** and *e* /= **Void**, where *e* is a read-only entity.

8.30.10 Definition: Scope of a read-only void test

Code: *DVRS*

The **scope** of a read-only void test *vt* is the scope of an Object_test which would be written in place of *vt* as one of the following, where *T* is the type of *e*:

1 For *vt* of the form *e* = **Void**: **not** (**attached** { *T*} *e*).

2 For *vt* of the form *e* /= **Void**: **attached** {*T*} *e*.

Informative Text

8.30.10.1 This is useful if *T* is a detachable type, providing a simple way to generalize the notion of scope to common schemes such as **if** *e* /= *Void* **then** …, where we know that *e* cannot be void in the Then_part. Note that it is essential to limit ourselves to read-only entities; for a variable, or an expression involving a variable, anything could happen to the value during the execution of the scope even if *e* is initially not void.

Signed by JV - 2026/10/8

8.30.10.2 Of course one could always write an Object_test instead, but the void test is a common and convenient form, if only because it doesn't require repeating the type *T* of *e*, so it will be important to handle it as part of the Certified Attachment Patterns discussed next.

End

8.30.11 Definition: Emergent attribute

Code: *DVEA*

An attribute declared by a New_feature_name with the **then** qualifier is called **emergent**.

Informative Text

8.30.11.1 An emergent attribute

End

8.30.12 Definition: Certified Attachment Pattern

Code: *DVCA*

A **Certified Attachment Pattern** (or **CAP**) for an expression *exp* whose type is detachable or emergent is an occurrence of *exp* in one of the following contexts:

1 *exp* is an object-test local and the occurrence is in its scope.

2 *exp* is a read-only entity and the occurrence is in the scope of a void test involving *exp*.

Informative Text

8.30.12.1 A CAP is a scheme that has been proved, or certified by sufficiently many competent people (or computerized proof tools), to ensure that *exp* will never have a void run-time value in the covered scope.

8.30.12.1.1 • The CAPs listed here are the most frequently useful and seem beyond doubt. Here too compilers could be "smart" and find other cases making *exp*`.f` safe. The language specification explicitly refrains, however, from accepting such supposed compiler improvements: other than the risk of mistake in the absence of a public discussion, this would result in some Eiffel texts being accepted by certain compilers and rejected by others. Instead, a compiler that *accepts* a call to a detachable target that is not part of one of the official CAPs listed above is **non-conformant**.

8.30.12.1.2 • The list of CAPs may grow in the future, as more analysis is applied to actual systems, leading to the identification, and certification by human or automatic means, of safe patterns for using targets of detachable types.

End

8.30.13 Definition: Attached expression

Code: *DVAE*

An expression *exp* of type *T* is **attached** if it satisfies any of the following conditions:

1 *T* is attached.

2 *T* is expanded.

3 *exp* appears in a Certified Attachment Pattern for *exp*.

Informative Text

8.30.13.1 This is the principal result of this discussion: the condition under which an expression is *target-valid*, that is to say, can be used as target of a call because its value is guaranteed never to be void at any time of evaluation. It is in an Expanded type's nature to abhor a void; attached types are devised to avoid void too; and Certified Attachment Patterns catch a detachable variable when it is provably not detached.

End

8.31 Typing-related properties

Informative Text

8.31.0.1 This Part does not introduce any new rules, only a few definitions that facilitate discussion of type issues.

End

8.31.1 Definition: Catcall

Code: *D5CC*

A **catcall** is a run-time attempt to execute a Call, such that the dynamic feature of the call is not applicable to the target of the call.

Informative Text

8.31.1.1 The role of the type system is to ensure that a valid system can never, during its execution, produce a catcall.

8.31.1.2 "Cat" is an abbreviation for "Changed Availability or Type", two language mechanisms that, if not properly controlled by the type system, could cause catcalls.

End

8.31.2 Validity: Descendant Argument rule

Code: *V5DA*

Consider a Call of target type *ST* and feature *fname* appearing in a class *C*. Let *sf* be the feature of final name *fname* in *ST*. Let *DT* be a type conforming to *ST*, and *df* the version of *sf* in *DT*. The call is **conforming-argument-valid** for *DT* if and only if its tuple-argument-unfolded form satisfies the following conditions:

1 The call is argument-valid.

2 Every actual argument conforms, after conversion to the type of the corresponding formal argument *sf* if applicable, replacement of *sf* by *df* and replacement of the target's type by *DT*, to the corresponding formal argument of *df*.

Informative Text

8.31.2.1 A call satisfying this property will be applicable under dynamic binding to a target of the chosen descendant type DT.

8.31.2.2 The rule uses the "tuple-argument-unfolded form", which reintroduces tuples even if the program text has dropped brackets around them when permitted for brevity (DJTA, page 185).

End

8.31.3 Validity: System-level Call rule

Code: *V5SC*

A call of target *x* is **system-valid** if for any element *D* of the dynamic type set of *x* it is export-valid for *D* and conformant-argument-valid for *D*.

Informative Text

8.31.3.1 The common goal of the type mechanisms and rules of the language is to ensure that every call is both class-valid and system-valid.

End

Signed by JV - 2026/10/8

8.32 Concurrency: the SCOOP mechanism

Informative Text

8.32.0.1 The

End

Signed by JV - 2026/10/8

8.33 Exception handling

Informative Text

8.33.0.1 During the execution of an Eiffel system, various abnormal events may occur. A hardware or operating system component may be unable to do its job; an arithmetic operation may result in overflow; an improperly written software element may produce an unacceptable outcome.

8.33.0.2 Such events will usually trigger a signal, or **exception**, which interrupts the normal flow of execution. If the system's text does not include any provision for the exception, execution will terminate. The system may, however, be programmed so as to *handle* exceptions, which means that it will respond by executing specified actions and, if possible, resuming execution after correcting the cause of the exception.

End

8.33.1 Definition: Failure, exception, trigger

Code: *DXFA*

Under certain circumstances, the execution or evaluation of a construct specimen may be unable to proceed as defined by the construct's semantics. It is then said to result in a **failure**.

If, during the execution of a feature, the execution of one of its components fails, this prevents continuing its execution normally; such an event is said to **trigger** an **exception**.

Informative Text

8.33.1.1 Examples of exception causes include:

8.33.1.1.1 • Assertion violation (in an assertion monitoring mode).

8.33.1.1.2 • Failure of a called routine.

8.33.1.1.3 • Impossible operation, such as a Creation_instruction (or evaluation of a Creation_expression) attempted when not enough memory is available, or an arithmetic operation which would cause an overflow or underflow in the platform's number system.

8.33.1.1.4 • Interruption signal sent by the machine for example after a user has hit the "break" key or the window of the current process has been resized.

8.33.1.1.5 • An exception explicitly raised by the software itself.

8.33.1.2 Common exception types that do *not* arise in Eiffel, other than through mistakes in the definition of the language as specified by the Standard, are "void calls" (attempts to execute a feature on a void target) and "catcalls" (attempt to execute a feature on an unsuitable object).

End

8.33.2 Syntax: Rescue clauses

Code: *SXRC*

Rescue ≙ **rescue** Compound

8.33.3 Validity: Rescue Clause rule

Code: *VXRC*

It is valid for a Feature_definition to include a Rescue clause if and only if its Feature_body is an Defined_attribute or an Effective_routine of the Internal form.

Informative Text

8.33.3.1 An Internal body is one which begins with either **do** or **once**. The other possibilities are Deferred, for which it would be improper to define an exception handler since the body does not specify an algorithm, and an External body, where the algorithm is specified outside of the Eiffel system, which then lacks the information it would need to handle exceptions.

End

8.33.4 Definition: Exception-correct

Code: *DXCO*

A routine is **exception-correct** if any branch of the Rescue clause terminating with **Retry** having value False ensures the class invariant.

8.33.5 Semantics: Default rescue original semantics

Code: *MXRO*

Class *ANY* introduces a non-frozen procedure *default_rescue* with no argument and a null effect.

Informative Text

8.33.5.1 As the following semantic rules indicate, an exception not handled by an explicit Rescue clause will cause a call to *default_rescue*. Any class can redefine this procedure to implement a default exception handling policy for routines of the class that do not have their own Rescue clauses.

End

8.33.6 Definition: Rescue block

Code: *DXRB*

Any Internal or Defined_attribute feature *f* of a class *C* has a **rescue block**, a Compound defined as follows, where *rc* is *C*'s version of *ANY*'s *default_rescue*:

1 If *f* has a Rescue clause: the Compound contained in that clause.

2 If *r* is not *rc* and has no Rescue clause: a Compound made of a single instruction: an Unqualified_call to *rc*.

3 If *r* is *rc* and has no Rescue clause: an empty Compound.

Informative Text

8.33.6.1 The semantic rules rely on this definition to define the effect of an exception as if every routine had a Rescue clause: either one written explicitly, or an implicit one calling *default_rescue*. To this effect they refer not to Rescue clauses but to rescue blocks.

8.33.6.2 Condition 3 avoids endless recursion in the case of *default_rescue* itself.

End

8.33.7 Semantics: Exception semantics

Code: *MXEX*

An exception triggered during an execution of a feature *f* causes, if it is neither ignored nor continued, the effect of the following sequence of events.

1 Attach the value of *last_exception* from *ANY* to a direct instance of a descendant of the Kernel Library class *EXCEPTION* corresponding to the type of the exception.

2 Unlike in the non-exception semantics of Compound, do not execute the remaining instructions of *f*.

3 If the recipient of the exception is *f*, execute the rescue block of *f*.

4 If case 3 applies and the value of the **Retry** local variable is True on termination of the rescue block's execution, this terminates the processing of the exception. Execution continues with a new execution of the Compound in the Feature_body of *f*.

5 If neither case 3 nor case 4 applies (in particular in case 3 if the final value of **Retry** is False), this terminates the processing of the current exception and the current execution of *f*, causing a failure of that execution. If the execution of *f* was caused by a call to *f* from another feature, trigger an exception of type *ROUTINE_FAILURE* in the calling routine, to be handled (recursively) according to the present rule. If there is no such calling feature, *f* is the root procedure; terminate its execution as having failed.

Informative Text

8.33.7.1 As usual in rules specifying the "effect" of an event in terms of a sequence of steps, all that counts is that effect; it is not required that the execution carry out these exact steps, or carry them in this exact order.

8.33.7.2 In most cases, the "recipient" of the exception (case 3) is the current routine, *f*. For exception occurring in special places, such as when evaluating an assertion, the next rule, Exception Cases, tells us whether *f* or its caller is the "recipient".

8.33.7.3 In step 4, the new execution of the Feature_body of *r*, resulting from a rescue block ending with a True value for **Retry**, will start with all entities set to their current value; it does **not** repeat argument passing and local variable initialization. This property may be used to ensure that the execution takes a different path on a new attempt.

8.33.7.4 The routine's postcondition may not refer to **Retry**; this requirement is specified in the entity rule, which precludes local variables other than **Result** in postconditions.

8.33.7.5 In the case of a Feature_body of the Once form, the above semantics only applies to the first call to every applicable target, where the body may be executed two or more times, as a result of an exception whose rescue block ends with **Retry** having value True. If that first call fails, triggering a routine failure exception, the applicable rule for subsequent calls is not the above Exception Semantics (since the routine will not execute again) but the Once Routine Execution Semantics, which specifies that any such calls must trigger the exception again.

End

8.33.8 Definition: Type of an exception

Code: *DXET*

The **type** of a triggered exception is the generating type of the object to which the value of *last_exception* is attached per step 1 of the Expression Semantics rule.

8.33.9 Semantics: Exception Cases

Code: *MXEC*

The triggering of an exception in a feature *f* called by a feature *caller* results in the setting of the following properties, accessible through features applicable to the exception class instance to which the value of *last_exception* is attached, as per the following table, where:

- The **Recipient** is either *f* or *caller*.
- "**Type**" indicates the type of the exception (a descendant of *EXCEPTION*).
- If *f* is the root procedure, executed during the original system creation call, the value of *caller* as given below does not apply.

	Recipient	**Type**
Exception during evaluation of invariant on entry	*caller*	[Type of exception as triggered]
Invariant violation on entry	*caller*	*INVARIANT_ENTRY_VIOLATION*
Exception during evaluation of precondition	*caller*	[Type of exception as triggered]
Exception during evaluation of Old expression on entry	See Old Expression Semantics	
Precondition violation	*caller*	*PRECONDITION_VIOLATION*
Exception in body	*f*	[Type of exception as triggered]
Exception during evaluation of invariant on exit	*f*	[Type of exception as triggered]
Invariant violation on exit	*f*	*INVARIANT_EXIT_VIOLATION*
Exception during evaluation of postcondition on exit	*f*	[Type of exception as triggered]

Postcondition violation	*f*	*POSTCONDITION_VIOLATION*

Informative Text

8.33.9.1 This rule specifies the precise effect of an exception occurring anywhere during execution (including some rather extreme cases, such as the occurrence of an exception in the evaluation of an assertion). Whether the "recipient" is *f* or *caller* determines whether the execution of the current routine can be "retried": as specified for case 3 of the Exception Semantics rule, retrying is only applicable if the recipient is *f* itself. Otherwise a *ROUTINE_FAILURE* will be triggered in the *caller*.

8.33.9.2 In the case of an Old expression, a special rule, given earlier, requires the exception to be remembered, during evaluation of the expression on entry to the routine, for re-triggering during evaluation of the postcondition on exit, but only if the expression turns out to be needed then.

End

8.33.10 Semantics: Exception Properties

Code: *MXEP*

The value of the query *original* of class *EXCEPTION*, applicable to *last_exception*, is an *EXCEPTION* reference determined as follows after the triggering of an exception of type *TEX*:

1 If *TEX* does not conform to *ROUTINE_FAILURE*: a reference to the current *EXCEPTION* object.

2 If *TEX* conforms to *ROUTINE_FAILURE*: the previous value of *original*.

Informative Text

8.33.10.1 The reason for this query is that when a routine fails, because execution of a routine *f* has triggered an exception and has not been able to handle it through retrying, the consequence, per case 5 of the Exception Semantics rule, is to trigger a new exception of type *ROUTINE_FAILURE*, to which *last_exception* now becomes attached. Without a provision for *original*, the "real" source of the exception would be lost, as *ROUTINE_FAILURE* exceptions get passed up the call chain. Querying *original* makes it possible, for any other routine up that chain, to find out the *Ur*-exception that truly started the full process.

End

8.33.11 Definition: Ignoring, continuing an exception

Code: *DXIG*

It is possible, through routines of the Kernel Library class *EXCEPTION*, to ensure that exceptions of certain types be:

- **Ignored**: lead to no change of non-exception semantics.
- **Continued**: lead to execution of a programmer-specified routine, then to continuation of the execution according to non-exception semantics.

Informative Text

8.33.11.1 The details of what types of exceptions can be ignored and continued, and how to achieve these effects, belong to the specification of class *EXCEPTION* and its descendants.

End

PART I: Basic program components

Signed by JV - 2026/10/8

Signed by JV - 2026/10/8

8.34 Constants

Informative Text

8.34.0.1 Expressions, just studied, include the special case of constants, whose values cannot directly be changed by execution-time actions. This discussion goes through the various kinds. Particular attention will be devoted to the various forms, single- and multi-line, of *string* constant.

8.34.0.2 Along with constants proper, we will study two notations for “manifest” objects given by the list of their items: manifest tuples and manifest arrays, both using the syntax [$item_1$, … $item_n$].

End

8.34.1 Syntax: Constants

Code: *S3CO*

Constant ≜ Manifest_constant | Constant_name

Constant_name ≜ Feature_name

8.34.2 Validity: Constant Attribute rule

Code: *V3CA*

A Constant_name appearing in a class *C* is valid if and only if its Feature_name is the final name of a constant attribute of *C*.

Informative Text

8.34.2.1 The feature must have been declared by a constant attribute declaration, satisfying the Constant Attribute rule (VFCA, page 118), which also determines the type of the constant attribute.

End

8.34.3 Syntax: Manifest constants

Code: *S3MA*

Manifest_constant ≜ [Manifest_type] Manifest_value

Manifest_type ≜ "**{**" Type "**}**"

Manifest_value ≜ Boolean_constant |
Character_constant |
Integer_constant |
Real_constant |
Manifest_string |
Manifest_type

Sign ≜ "**+**" | "**–**"

Integer_constant ≜ [Sign] Integer

Character_constant ≜ "**'**" Character "**'**"

Boolean_constant ≜ **True** | **False**

Real_constant ≜ [Sign] Real

8.34.4 Syntax (non-production): Sign Syntax rule

Code: *S3SG*

If present, the Sign of an Integer_constant or Real_constant must immediately precede the associated Integer or Real, with no intervening tokens or other components (such as breaks or comments).

8.34.5 Syntax (non-production): Character Syntax rule

Code: *S3CH*

The quotes of a Character_constant must immediately precede and follow the Character, with no intervening tokens or other components (such as breaks or comments).

Signed by JV - 2026/10/8

Informative Text

8.34.5.1 In general, breaks or comment lines may appear between components prescribed by a BNF-E production, making the last two rules necessary to complement the grammar: for signed constants, you must write –5, not – 5 etc. This helps avoid confusion with operators in arithmetic expressions, which may of course be followed by spaces, as in *a – b*. Similarly, you must write a character constant as '*A*', not ' *A* '.

8.34.5.2 To avoid any confusion about the syntax of Character_constant, it is important to note that a character code such as %N (New Line) constitutes a single Character token (even though its notation uses two characters).

End

8.34.6 Definition: Type of a manifest constant

Code: *D3MT*

The type of a Manifest_constant of Manifest_value *mv* is:

1 For {*T*} *mv*, with the optional Manifest_type present: *T*. The remaining cases assume this optional component is absent, and only involve *mv*.
2 If *mv* is a Boolean_constant: *BOOLEAN*.
3 If *mv* is a Character_constant: *CHARACTER*.
4 If *mv* is an Integer_constant: *INTEGER*.
5 If *mv* is a Real_constant: *REAL*.
6 If *mv* is a Manifest_string: *STRING*.
7 If *mv* is a Manifest_type {*T*}: *TYPE* [*T*].

Informative Text

8.34.6.1 As a consequence of cases 3 to 6, the type of a character, string or numeric constant is never one of the sized variants but always the fundamental underlying type (*CHARACTER*, *INTEGER*, *REAL*, *STRING*). Language mechanisms are designed so that you can use such constants without hassle — for example, without explicit conversions — even in connection with specific variants. For example:

- You can assign an integer constant such as 10 to a target of a type such as *INTEGER_8* as long as it fits (as enforced by validity rules).
- You can use such a constant for discrimination in a Multi_branch even if the expression being discriminated is of a specific sized variant; here too the compatibility is enforced statically by the validity rules.

8.34.6.2 Case 7 involves the Kernel Library class *TYPE*.

End

8.34.7 Validity: Manifest-Type Qualifier rule

Code: *V3MQ*

It is valid for a Manifest_constant to be of the form {*T*} *v* (with the optional Manifest_type qualifier present) if and only if the type *U* of *v* (as determined by cases 2 to 7 of the definition of the type of a manifest constant) is one of *CHARACTER*, *STRING*, *INTEGER* and *REAL*, and *T* is one of the sized variants of *U*.

Informative Text

8.34.7.1 The rule states no restriction on the value, even though an example such as {*INTEGER_8*} 256 is clearly invalid, since 256 is not representable as an *INTEGER_8*. The Manifest Constant rule addresses this.

End

8.34.8 Semantics: Manifest constant semantics

Code: *M3MC*

The **value** of a Manifest_constant *c* listing a *Manifest_value v* is:

1 If *c* is of the form {*T*} *v* (with the optional Manifest_type qualifier present): the value of type *T* denoted by *v*.

2 Otherwise (*c* is just *v*): the value denoted by *v*.

8.34.9 Definition: Manifest value of a constant

Code: *D3MC*

The **manifest value** of a constant is:

1 If it is a Manifest_constant: its value.

2 If it is a constant attribute: (recursively) the manifest value of the Manifest_constant listed in its declaration.

Informative Text

8.34.9.1 As the following syntax indicates, there are two ways to write a manifest string:

- A Basic_manifest_string, the most common case, is a sequence of characters in double quotes, as in "*This text*". Some of the characters may be special character codes, such as %N representing a new line. This variant is useful for such frequent applications as object names, texts of simple messages to be displayed, labels of buttons and other user interface elements, generally using fairly short and simple sequences of characters. You may write the string over several lines by ending an interrupted line with a percent character % and starting the next one, after possible blanks and tabs, by the same character.
- A Verbatim_string is a sequence of lines to be taken exactly as they are (hence the name), bracketed by "{ at the end of the line that precedes the sequence and }" at the beginning of the line that follows the sequence (or "[and "] to left-align the lines). No special character codes apply. This is useful for embedding multi-line texts; applications include *description* entries of Notes clauses, inline C code, SQL or XML queries to be passed to some external program.

End

8.34.10 Syntax: Manifest strings

Code: *S3MS*

Manifest_string ≜ Basic_manifest_string| Verbatim_string

Basic_manifest_string ≜ ' " ' String_content ' " '

String_content ≜ {Simple_string Line_wrapping_part …}+

Verbatim_string ≜ Verbatim_string_opener Line_sequence Verbatim_string_closer

Verbatim_string_opener ≜ ' " ' [Simple_string] Open_bracket

Verbatim_string_closer ≜ Close_bracket [Simple_string] ' " '

Open_bracket ≜ "[" | "{"

Close_bracket ≜ "]" | "}"

Signed by JV - 2026/10/8

Informative Text

8.34.10.1

8.34.10.2 In the “basic” case, covering most uses of strings, specimens of String_content involve just one Simple_string (a sequence of printable characters, with no new lines, as defined in the description of lexical components). For generality, however, String_content is defined as a repetition, with successive Simple_string components separated by Line_wrapping_part to allow writing a string on several lines. Details below.

8.34.10.3 In the “verbatim” case, Line_sequence is a lexical construct denoting a sequence of lines with arbitrary text. The reason for the Verbatim_string_opener and the Verbatim_string_closer is to provide an escape sequence for an extreme case (a Line_sequence that begins with **]"**), but most of the time the opener is just **"[** or **"{** and the closer **]"** or **"}**. The difference between brackets and braces is that with **"{** … **}"** the Line_sequenceis kept exactly as is, whereas with **"[** … **]"** the lines are left-aligned (stripped of any common initial blanks and tabs). Details below.

End

8.34.11 Syntax (non-production): Line_sequence

Code: *S3LS*

A specimen of Line_sequenceis a sequence of one or more Simple_string components, each separated from the next by a single New_line.

8.34.12 Syntax (non-production): Manifest String rule

Code: *S3MR*

In addition to the properties specified by the grammar, every Manifest_string must satisfy the following properties:

1 The Simple_stringcomponents of its String_content or Line_sequencemay not include a double quote character except as part of the character code **%"** (denoting a double quote).

2 A Verbatim_string_opener or Verbatim_string_closer may not contain any break character.

Informative Text

8.34.12.1 Like other “non-production” syntax rules, the last two rules capture simple syntax requirements not expressible through BNF-E productions.

8.34.12.2 Because a Line_sequence is made of simple strings separated by a single New_line in each case, a line in a Verbatim_string that looks like a comment is not a comment but a substring of the Verbatim_string.

End

8.34.13 Syntax (non-production): Line_wrapping_part

Code: *S3LW*

A Line_wrapping_part is a sequence of characters consisting of the following, in order: **%** (percent character); zero or more blanks or tabs; New_line; zero or more blanks or tabs; **%** again.

Informative Text

8.34.13.1 This construct requires such a definition since it cannot be specified through a context-free syntax formalism such as BNF-E.

8.34.13.2 The use of Line_wrapping_part as separator between a Simple_string and the next in a Basic_manifest_string allows you to split a string across lines, with a % at the end of an interrupting line and another one at the beginning of the resuming line. The definition allows blanks and tabs before the final **%** of a Line_wrapping_part although they will not contribute to the contents of the string. This makes it possible to apply to the Basic_manifest_stringthe same indentation as to the neighboring elements. The definition also permits blanks and tabs after the

Signed by JV - 2026/10/8

initial % of a Line_wrapping_part, partly for symmetry and partly because it doesn't seem justified to raise an error just because the compiler has detected such invisible but probably harmless characters.

End

8.34.14 Semantics: Manifest string semantics

Code: *M3MS*

The value of a Basic_manifest_string is the sequence of characters that it includes, in the order given, excluding any line_wrapping_parts, and with any character_code replaced by the corresponding character.

8.34.15 Validity: Verbatim String rule

Code: *V3VS*

A Verbatim_string is valid if and only if it satisfies the following conditions, where α is the (possibly empty) Simple_string appearing in its Verbatim_string_opener:

1 The Close_bracket is **]** if the Open_bracket is **[**, and **}** if the Open_bracket is **{**.

2 Every character in α is printable, and not a double quote **"**.

3 If α is not empty, the string's Verbatim_string_closer includes a Simple_string identical to α.

8.34.16 Semantics: Verbatim string semantics

Code: *M3VS*

8.34.16.1 Verbatim string semantics

The value of a Line_sequence is the string obtained by concatenating the characters of its successive lines, with a "new line" character inserted between any adjacent ones.

The value of a Verbatim_string using braces **{ }** as Open_bracket and Close_bracketis the value of its Line_sequence.

The value of a Verbatim_string using braces **[ ]** as Open_bracket and Close_bracketis the value of the left-aligned form of its Line_sequence.

Informative Text

8.34.16.2 This semantic definition is **platform-independent**: even if an environment has its own way of separating lines (such as two characters, carriage return %R and new line %N, on Windows) or represents each line as a separate element in a sequence (as in older operating systems still used on mainframes), the semantics yields a single string — a single character sequence — where each successive group of characters, each representing a line of the original, is separated from the next one by a single %N.

End

8.34.17 Definition: Prefix, longest break prefix, left-aligned form

Code: *D3PR*

A **prefix** of a string *s* is a string *p* of some length *n* ($n \geq 0$) such that the first *n* characters of *s* are the corresponding characters of *p*.

The **longest break prefix** of a sequence of strings *ls* is the longest string *bp* containing no characters other than spaces and tabs, such that *bp* is a prefix of every string in *ls*. (The longest break prefix is always defined, although it may be an empty string.)

The **left-aligned form** of a sequence of strings *ls* is the sequence of strings obtained from the corresponding strings in *ls* by removing the first *n* characters, where *n* is the length of the longest break prefix of *ls* ($n \geq 0$).

8.35 Basic types

Informative Text

8.35.0.1 The term “basic type” covers a number of expanded class types describing elementary values: booleans, characters, integers, reals, machine-level addresses. The corresponding classes — *BOOLEAN*; *CHARACTER*, *INTEGER*, *REAL* and variants specifying explicit sizes; *POINTER* — are part of ELKS, the Eiffel Library Kernel Standard.

8.35.0.2 The following presentation explains the general concepts behind the design and use of these classes and specifies the possible values (direct instances) of each of them.

End

8.35.1 Definition: Basic types and their sized variants

Code: *D4BT*

A **basic type** is any of the types defined by the following ELKS classes:

- *BOOLEAN*.
- *CHARACTER*, *CHARACTER_8*, *CHARACTER_32*, together called the “**sized variants** of *CHARACTER*”.
- *INTEGER*, *INTEGER_8*, *INTEGER_16*, *INTEGER_32*, *INTEGER_64*, *NATURAL*, *NATURAL_8*, *NATURAL_16*, *NATURAL_32*, *NATURAL_64*, together called the “**sized variants** of *INTEGER*”.
- *REAL*, *REAL_32*, *REAL_64*, together called the “**sized variants** of *REAL*”.
- *POINTER*.

8.35.2 Definition: Sized variants of *STRING*

Code: *D4SS*

The sized variants of *STRING* are *STRING*, *STRING_8* and *STRING_32*.

8.35.3 Semantics: Boolean value semantics

Code: *M4BO*

Class *BOOLEAN* covers the two truth values.

The reserved words *True* and *False* denote the corresponding constants.

8.35.4 Semantics: Character types

Code: *M4CO*

The reference class *CHARACTER_GENERAL* describes properties of characters independently of the character code.

The expanded class *CHARACTER_32* describes Unicode characters; the expanded class *CHARACTER_8* describes 8-bit (ASCII-like) characters.

Direct instances of the expanded class *CHARACTER* are characters with a length and encoding settable through a compilation option. The recommended default is Unicode.

8.35.5 Semantics: Integer types

Code: *M4IN*

Direct instances of the reference class *INTEGER_GENERAL* are integers, signed or not, of arbitrary length. Direct instances of the expanded classes *INTEGER_xx*, for *xx* = 8, 16, 32 or 64, are signed integers stored on *xx* bits. The expanded classes *NATURAL_xx*, for *xx* = 8, 16, 32 or 64, describe unsigned integers stored on *xx* bits.

The expanded classes *INTEGER* and *NATURAL* describe integers, respectively signed and unsigned, with a length settable through a compilation option. The recommended default is 64 bits in both cases.

8.35.6 Semantics: Floating-point types

Code: *M4RE*

Direct instances of the reference class *REAL_GENERAL* are floating-point numbers with arbitrary precision. Direct instances of the expanded classes *REAL_xx*, for *xx* = 32 or 64, are IEEE floating-point numbers with *xx* bits of precision.

Direct instances of the expanded class *REAL* are floating-point numbers with a precision settable through a compilation option. The recommended default is 64 bits.

8.35.7 Semantics: Address semantics

Code: *M4AD*

Direct instances of the expanded class *POINTER* are addresses of data beyond the control of Eiffel systems.

8.36 Interfacing with C, C++ and other environments

Informative Text

8.36.0.1 Object technology as realized in Eiffel is about **combining components**. Not all of these components are necessarily written in the same language; in particular, as organizations move to Eiffel, they will want to reuse their existing investment in components from other languages, and make their Eiffel systems interoperate with non-Eiffel software.

8.36.0.2 Eiffel is a "pure" O-O language, not a hybrid between object principles and earlier approaches such as C, and at the same time an **open** framework for combining software written in various languages. These two properties might appear contradictory, as if consistent use of object technology meant closing oneself off from the rest of the programming world. But it's exactly the reverse: a hybrid approach, trying to be O-O as well as something completely different, cannot succeed at both since the concepts are too distant. Eiffel instead strives, by providing a coherent object framework — with such principles as Uniform Access, Command-Query Separation, Single Choice, Open-Closed and Design by Contract — to be a *component combinator* capable of assembling software bricks of many different kinds.

8.36.0.3 The following presentation describes how Eiffel systems can integrate components from other languages and environments.

End

8.36.1 Syntax: External routines

Code: *SZER*

External ≜ **external** External_language [External_name]

External_language ≜ Unregistered_language | Registered_language

Unregistered_language ≜ Manifest_string

External_name ≜ **alias** Manifest_string

Informative Text

8.36.1.1 The External clause is the mechanism that enables Eiffel to interface with other environments and serve as a "component combinator" for software reuse and particularly for taking advantage of legacy code.

8.36.1.2 By default the mechanism assumes that the external routine has the same name as the Eiffel routine. If this is not the case, use an External_name of the form **alias** "ext_name". The name appears as a Manifest_string, in quotes, not an identifier, because external languages may have different naming conventions; for example an underscore may begin a feature name in C but not in Eiffel, and some languages are case-sensitive for identifiers whereas Eiffel is not.

8.36.1.3 Instead of calling a pre-existing foreign routine, it is possible to include **inline** C or C++ code; the **alias** clause will host that code, which can access Eiffel objects through the arguments of the external routine.

8.36.1.4 The language name (External_name) can be an Unregistered_language: a string in quotes such as **"Cobol"**. Since the content of the string is arbitrary, there is no guarantee that a particular Eiffel environment will support the corresponding language interface. This is the reason for the other variant, Registered_language: every Eiffel compiler must support the language names **"C"**, **"C++"** and **dll**. Details of the specific mechanisms for every such Registered_language appear below.

8.36.1.5 Some of the *validity* rules below include a provision, unheard of in other parts of the language specification, allowing Eiffel language processing tools to rely on *non-Eiffel tools* to enforce some conditions. A typical example is a rule that requires an external name to denote a suitable foreign function; often, this can only be ascertained by a compiler for the foreign language. Such rules should be part of the specification, but we can't impose their enforcement on an Eiffel compiler without asking it also to become a compiler of C, C++ etc.; hence this special tolerance.

Signed by JV - 2026/10/8

8.36.1.6 The general *semantics* of executing external calls appeared as part of the general semantics of calls. The semantic rules of the present discussion address specific cases, in particular inline C and C++.

End

Informative Text

8.36.1.7 Address expressions of the form **$***v* where *v* is a variable denote the address of the corresponding variable; not necessarily the physical address, but a value enabling foreign software to access the variable. As the following validity rule expresses, that value must be defined in the context of the expression.

End

8.36.2 Validity: Address Expression rule

Code: *VZAE*

An Address expression is valid if and only if its Variable is properly set at the evaluation position to which it belongs.

8.36.3 Semantics: Address semantics

Code: *MZAE*

The value of an Address expression is an address enabling foreign software to access the associated Variable.

Informative Text

8.36.3.1 The manipulations that the foreign software can perform on such addresses depend on the foreign programming language. It is the implementation's responsibility to ensure that such manipulations do not violate Eiffel semantic properties.

End

8.36.4 Syntax: Registered languages

Code: *SZRL*

Registered_language ≜ C_external | C++_external | DLL_external

8.36.5 Syntax: External signatures

Code: *SZES*

External_signature ≜ **signature** [External_argument_types] [**:** External_type]

External_argument_types ≜ "**(**" External_type_list "**)**"

External_type_list ≜ {External_type "**,**" …}*

External_type ≜ Simple_string

8.36.6 Validity: External Signature rule

Code: *VZES*

An External_signature in the declaration of an external routine *r* is valid if and only if it satisfies the following conditions:

1 Its External_type_list contains the same number of elements as *r* has formal arguments.

2 The final optional component (**:** External_type) if present if and only if *r* is a function.

A language processing tool may delegate enforcement of these requirements to non-Eiffel tools on the chosen platform.

Informative Text

8.36.6.1 The rule does not prescribe any particular relationship between the argument and result types declared for the Eiffel routine and the names appearing in the External_type_list and the final External_type if any, since the precise correspondence depends on foreign language properties beyond the scope of Eiffel rules.

8.36.6.2 The specification of a non-external routine never includes C-style empty parenthesization: for a declaration or call of a routine without arguments you write *r*, not *r* (). The syntax of External_argument_types, however, permits **()** for compatibility with other languages' conventions.

8.36.6.3 The last part of the rule allows Eiffel tools to rely on non-Eiffel tools if it is not possible, from within Eiffel, to check the properties of external routines. This provision also applies to several of the following rules.

End

8.36.7 Semantics: External signature semantics

Code: *MZES*

An External_signature specifies that the associated external routine:

- Expects arguments of number and types as given by the External_argument_types if present, and no arguments otherwise.
- Returns a result of the External_type appearing after the colon, if present, and otherwise no result.

8.36.8 Syntax: External file use

Code: *SZEF*

External_file_use ≜ **use** External_file_list

External_file_list ≜ {External_file "**,**" …}$^{+}$

External_file ≜ External_user_file | External_system_file

External_user_file ≜ ' **"** 'Simple_string ' **"** '

External_system_file ≜ "**<**"Simple_string "**>**"

Informative Text

8.36.8.1 As the syntax indicates, you may specify as many external files as you like, preceded by **use** and separated by commas. You may specify two kinds of files:

8.36.8.1.1 - "System" files, used only in a C context, appear between angle brackets <> and refer to specific locations in the C library installation.

8.36.8.1.2 - The name of a "user" file appears between double quotes, as in "/path/user/her_include**.**h", and will be passed on literally to the operating system. Do not forget, when using double quotes, that this is all part of an Eiffel Manifest_string: you must either code them as %" or, more conveniently, write the string as a Verbatim_string, the first line preceded by **"[** and the last line followed by **]"**.

End

8.36.9 Validity: External File rule

Code: *VZEF*

An External_fileis valid if and only if its Simple_string atisfies the following conditions:

1. When interpreted as a file name according to the conventions of the underlying platform, it denotes a file.
2. The file is accessible for reading.
3. The file's content satisfies the rules of the applicable foreign language.

A language processing tool may delegate enforcement of these conditions to non-Eiffel tools on the chosen platform.

Informative Text

8.36.9.1 Condition 3 means for example that if you pass an include file to a C function the content must be C code suitable for inclusion by a C "include" directive. Such a requirement may be beyond the competence of an Eiffel compiler, hence the final qualification enabling Eiffel tools to rely, for example, on compilation errors produced by a C compiler.

8.36.9.2 The "conventions of the underlying platforms" cited in condition 1 govern the rules on file names (in particular the interpretation of path delimiters such as **/** and **** on Unix and Windows) and, for an External_system_file name of the form **<***some_file***.**h**>**, the places in the file system where *some_file***.**h is to be found.

End

8.36.10 Semantics: External file semantics

Code: *MZEF*

An External_file_use in an external routine declaration specifies that foreign language tools, to process the routine (for example to compile its original code), require access to the listed files.

8.36.11 Syntax: C externals

Code: *SZEC*

C_external ≜ '**"** ' **C**
'[**inline**]
[External_signature] [External_file_use]
' **"** '

Informative Text

8.36.11.1 The C_external mechanism makes it possible, from Eiffel, to use the mechanisms of C. The syntax covers two basic schemes:

8.36.11.1.1 • You may rely on an existing C function. You will not, in this case, use **inline**. If the C function's name is different from the lower name of the Eiffel routine, specify it in the **alias** (External_name) clause; otherwise you may just omit that clause.

8.36.11.1.2 • You may also write C code *within* the Eiffel routine, putting that code in the **alias** clause and specifying **inline**.

8.36.11.2 In the second case the C code can directly manipulate the routine's formal arguments and, through them, Eiffel objects. The primary application (rather than writing complex processing in C code in an Eiffel class, which would make little sense) is to provide access to existing C libraries without having to write and maintain any new C files even if some "glue code" is necessary, for example to perform type adaptations. Such code, which should remain short and simple, will be directly included and maintained in the Eiffel classes providing the interface to the legacy code.

8.36.11.3 The **alias** part is a Manifest_string of one of the two available forms:

8.36.11.3.1 • It may begin and end with a double quote **"**; then any double quote character appearing in it must be preceded by a percent sign, as **%"**; line separations are marked by the special code for "new line", **%N**.

8.36.11.3.2 • If the text extends over more than one line, it is more convenient to use a Verbatim_string: a sequence of lines to be taken exactly as they are, preceded by **"[** at the end of a line and followed by **]"** at the beginning of a line.

8.36.11.4 In this Manifest_string, you may refer to any formal argument *a* of the external routine through the notation **$***a* (a dollar sign immediately followed by the name of the argument). For *a* you may use either upper or lower case, lower being the recommended style as usual.

End

Signed by JV - 2026/10/8

8.36.12 Validity: C External rule

Code: *VZEC*

A C_external for the declaration of an external routine *r* is valid if and only if it satisfies the following conditions:

1 At least one of the optional **inline** and External_signature components is present.

2 If the **inline** part is present, the external routine includes an **External_name** component, of the form **alias** *C_text*.

3 If case 2 applies, then for any occurrence in *C_text* of an Identifier *a* immediately preceded by a dollar sign **$** the lower name of *a* is the lower name of a formal argument of *r*.

8.36.13 Semantics: C Inline semantics

Code: *MZIC*

In an external routine *er* of the **inline** form, an External_nameof the form **alias** *C_text* denotes the algorithm defined, according to the semantics of the C language, by a C function that has:

- As its signature, the signature specified by *er*.
- As its body, *C_text* after replacement of every occurrence of **$***a*, where the lower name of *a* is the lower name of one of the formal arguments of *er*, by *a*.

8.36.14 Syntax: C++ externals

Code: *SZC+*

C++_external ≙ ' **"** ' **C++**
inline
[External_signature]
[External_file_use]
' **"** '

Informative Text

8.36.14.1 As in the C case, you may directly write C++ code which can access the external routine's argument and hence Eiffel objects. Such code can, among other operations, create and delete C++ objects using C++ constructors and destructors.

8.36.14.2 Unlike in the C case, this inline facility is the *only* possibility: you cannot rely on an existing function. The reason is that C++ functions — if not "static" — require a target object, like Eiffel routines. By directly writing appropriate inline C++ code, you will take care of providing the target object whenever required.

End

8.36.15 Validity: C++ External rule

Code: *VZC+*

A C++_external part for the declaration of an external routine *r* is valid if and only if it satisfies the following conditions:

1 The external routine includes an External_name component, of the form **alias** *C++_text*.

2 For any occurrence in *C++_text* of an Identifier *a* immediately preceded by a dollar sign **$**, the lower name of *a* is the lower name of a formal argument of *r*.

8.36.16 Semantics: C++ Inline semantics

Code: *MZC+*

In an external routine *er* of the C++_external form, an External_name of the form **alias** *C++_text* denotes the algorithm defined, according to the semantics of the C++ language, by a C++ function that has:

- As its signature, the signature specified by *er*.
- As its body, *C++_text* after replacement of every occurrence of **$***a*, where the lower name of *a* is the lower name of one of the formal arguments of *er*, by *a*.

Signed by JV - 2026/10/8

8.36.17 Syntax: DLL externals

Code: *SZDL*

DLL_external ≜ ' " ' **dll**
[**windows**]
DLL_identifier
[DLL_index]
[External_signature]
[External_file_use]
' " '

DLL_identifier ≜ Simple_string

DLL_index ≜ Integer

8.36.17.1 Through a DLL_external you may define an Eiffel routine whose execution calls an external mechanism from a Dynamic Link Library, not loaded until first use.

8.36.17.2 The mechanism assumes a dynamic loading facility, such as exist on modern platforms; it is specified to work with any such platform.

End

8.36.18 Validity: External DLL rule

Code: *VZDL*

A DLL_external of DLL_identifier *i* is valid if and only if it satisfies the following conditions:

1 When interpreted as a file name according to the conventions of the underlying platform, *i* denotes a file.

2 The file is accessible for reading.

3 The file's content denotes a dynamically loadable module.

8.36.19 Semantics: External DLL semantics

Code: *MZDL*

The routine to be executed (after loading if necessary) in a call to a DLL_external is the dynamically loadable routine from the file specified by the DLL_identifier and, within that file, by its name and the DLL_index if present.

8.37 Lexical components

Informative Text

8.37.0.1 The discussions in the previous subclauses have covered the syntax, validity and semantics of software systems. At the most basic level, the texts of these systems are made of **lexical** components, playing for Eiffel classes the role that words and punctuation play for the sentences of human language.

8.37.0.2 All construct descriptions relied on lexical components — identifiers, reserved words, special symbols … — but their structure has not been formally defined yet. It is time now to cover this aspect of the language, affecting its most elementary components.

End

8.37.1 Syntax (non-production): Character, character set

Code: *SICH*

An Eiffel text is a sequence of **characters**. Characters are either:

- All 32-bit, corresponding to Unicode and to the Eiffel type *CHARACTER_32*.
- All 8-bit, corresponding to 8-bit extended ASCII and to the Eiffel type *CHARACTER_8*.

Compilers and other language processing tools must offer an option to select one **character set** from these two. The same or another option determines whether the type *CHARACTER* is equivalent to *CHARACTER_32* or *CHARACTER_8*.

Informative Text

8.37.1.1 In manifest strings and character constants, characters can be coded either directly, as a single-key entry, or through a multiple-key character code such as %N (denoting a new-line character) or %/59/. The details appear below.

End

8.37.2 Definition: Letter, letter case, alpha_betic, numeric, alpha_numeric, printable

Code: *DILE*

A **letter** is any character belonging to one of the following categories:

1 Any of the following fifty-two, each a lower-case (first row) or upper-case second row) element of the Roman alphabet:

a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

2 If the underlying character set is 8-bit extended ASCII, the characters of codes 192 to 255 in that set.

3 If the underlying character set is Unicode, all characters defined as letters in that set.

An **alpha_betic character** is a letter or an underscore **_**.

A **numeric character** is one of the ten characters *0 1 2 3 4 5 6 7 8 9*.

An **alpha_numeric character** is alpha_betic or numeric.

A **printable character** is any of the characters listed as printable in the definition of the character set (Unicode or extended ASCII).

Informative Text

8.37.2.1 In common English usage, “alphabetic” and “alphanumeric” characters do not include the underscore. The spellings “*alpha_betic*” and “*alpha_numeric*” are a reminder that we accept underscores in both identifiers, as in *your_variable*, and numeric constants, as in 8_961_226.

8.37.2.2 “*Printable*” characters exclude such special characters as New Line and Backspace.

Signed by JV - 2026/10/8

8.37.2.3 Case 2 of the definition of “letter” refers to the 8-bit extended ASCII character set. Only the 7-bit ASCII character set (as used for English) is universally defined; the 8-bit extension has variants corresponding to alphabets used in various countries. Codes 192 to 255 generally cover letters equipped with *diacritical marks* (accents, umlauts, cedilla). As a result, if you use an 8-bit letter not in the 7-bit character set, for example to define an identifier with a diacritical mark, it may — without any effect on its Eiffel semantics — display differently depending on the “*locale*” settings of your computer.

8.37.2.4 Correspondingly, lower and upper case are defined for the 7-bit ASCII code. In the case of identifiers using extended ASCII, for example accented letters such as *à* and *À*, the style rules on letter case for identifiers (D7LC, page 71), suggesting upper case for classes and lower case for feature names (other than constant attributes and once functions) and local variables, the practice of the corresponding language should be applied. (In French, for example, *à* and *À* are lower- and upper-case variants of the same letter, although it is also common to omit the accent in the upper-case form.)

End

8.37.3 Definition: Break character, break

Code: *DIBR*

A **break character** is one of the following characters:

- Blank (also known as space).
- Tab.
- New Line (also known as Line Feed).
- Return (also known as Enter and as Carriage Return).

A **break** is a sequence of one or more break characters that is not part of a Character_constant, of a Manifest_string or of a Simple_string component of a Comment.

8.37.4 Semantics: Break semantics

Code: *MIBR*

Breaks serve a purely syntactical role, to separate tokens. The effect of a break is independent of its makeup (its precise use of spaces, tabs and newlines). In particular, the separation of a class text into lines has no effect on its semantics.

Informative Text

8.37.4.1 Because the above definition of “break” excludes break characters appearing in Character_constant, Manifest_string and Comment components, the semantics of these constructs may take such break characters into account.

End

8.37.5 Definition: Expected, free comment

Code: *DICO*

A comment is **expected** if it appears in a construct as part of the style guidelines for that construct. Otherwise it is **free**.

8.37.6 Syntax (non-production): “Blanks or tabs”, new line

Code: *SIBT*

A specimen of Blanks_or_tabs is any non-empty sequence of characters, each of which is a blank or a tab.

A specimen of New_line is a New Line.

8.37.7 Syntax: Comments

Code: *SICO*

Comment ≙ "– –" {Simple_string Comment_break …}*

Comment_break ≙ New_line [Blanks_or_tabs] "– –"

Signed by JV - 2026/10/8

Informative Text

8.37.7.1 This syntax implies that two or more successive comment lines, with nothing other than new lines to separate them, form a single comment.

End

8.37.8 Syntax (non-production): Free Comment rule

Code: *SIFC*

It is permitted to include a free comment between any two successive components of a specimen of a construct defined by a BNF-E production, except if excluded by specific syntax rules.

Informative Text

8.37.8.1 An example of construct whose specimens may not include comments is Line_sequence, defined (in S3LS, page 269) not by a BNF-E production but by another “non-production” syntax rule: no comments may appear between the successive lines of such a sequence — or, as a consequence, of a Verbatim_string.

8.37.8.2 Similarly, the Alias Syntax rule (SFAL, page 105) excludes any characters — and hence comments — between an Alias_name and its enclosing quotes.

End

8.37.9 Principle: Header Comment rule

Code: *PIHC*

A feature’s Header_comment is an abbreviation for a Notes clause of the form

note

what: *Explanation*

where *Explanation* is a Verbatim_string with **[** and **]** as Open_bracket and Close_bracket and a a Line_sequence made up of the successive lines (each a Simple_string) of the comment, each deprived of its first characters up to and including the first two consecutive dash characters, and of the space immediately following them if any.

Informative Text

8.37.9.1 This principle is a semantic rule. Per the syntax, a comment is a succession of Simple_string components, each prefixed by "**--**" itself optionally preceded, in the second and subsequent lines if any, by a Blanks_or_tabs. To make up the Blanks_or_tabs we remove the Blanks_or_tabs and dashes; we also remove one immediately following space, to account for the common practice of separating the dashes from the actual comment text, as in

```
-- A comment.
```

End

8.37.10 Definition: Symbol, word

Code: *DISY*

A **symbol** is either a special symbol of the language, such as the semicolon “**;**” and the “**.**” of dot notation, or a standard operator such as “**+**” and “*****”.

A **word** is any token that is not a symbol. Examples of words include identifiers, keywords, free operators and non-symbol operators such as **or else**.

8.37.11 Syntax (non-production): Break rule

Code: *SIBR*

It is permitted to write two adjacent tokens without an intervening break if and only if they satisfy one of the following conditions:

1 One is a word and the other is a symbol.

2 They are both symbols, and their concatenation is not a symbol.

Signed by JV - 2026/10/8

Informative Text

8.37.11.1 Without this rule, adjacent words not separated by a break — as in *ifxthen* — or adjacent symbols would be ambiguous.

End

8.37.12 Semantics: Letter Case rule

Code: *MILC*

Letter case is significant for the following constructs: Character_constant and Manifest_string except for special character codes, Comment.

For all other constructs, letter case is not significant: changing a letter to its lower-case or upper-case counterpart does not affect the semantics of a specimen of the construct.

8.37.13 Definition: Reserved word, keyword

Code: *DIKE*

The following names are **reserved words** of the language.

across	**agent**	**alias**	**all**	**and**	**as**	**assign**
attribute	**check**	**class**	**convert**	**create**	**Current**	**debug**
deferred	**do**	**else**	**elseif**	**end**	**ensure**	**expanded**
export	**external**	**False**	**feature**	**from**	**frozen**	**if**
implies	**inherit**	**inspect**	**invariant**	**like**	**local**	**loop**
not	**note**	**obsolete**	**old**	**once**	**or**	**Precursor**
redefine	**rename**	**require**	**rescue**	**Result**	**Retry**	**select**
separate	**some**	**then**	**True**	*TUPLE*	**undefine**	**until**
variant	**Void**	**when**	**xor**			

The reserved words that serve as purely syntactical markers, not carrying a direct semantic value, are called **keywords**; they appear in the above list in all lower-case letters.

Informative Text

8.37.13.1 The non-keyword reserved words, such as **True**, have a semantics of their own (**True** denotes one of the two boolean values).

8.37.13.2 The Letter Case rule applies to reserved words, so the decision to write keywords in all lower case is simply a style guideline. Non-keyword reserved words are most closely related to constants and, like constants, have — in the recommended style — a single upper-case letter, the first; *TUPLE* is most closely related to types and is all upper-case.

End

8.37.14 Syntax (non-production): Double Reserved Word rule

Code: *SIDR*

The reserved words **and then** and **or else** are each made of two components separated by one or more blanks (but no other break characters). Every other reserved word is a sequence of letters with no intervening break character.

8.37.15 Definition: Special symbol

Code: *DISP*

A **special symbol** is any of the following character sequences:

-- : ; , ? ' " $. -> :=
= /= ~ /~ ( ) [ ] { }

8.37.16 Syntax (non-production): Identifier

Code: *SIID*

An Identifier is a sequence of one or more alpha_numeric characters of which the first is a letter.

Signed by JV - 2026/10/8

8.37.17 Validity: Identifier rule

Code: *VIID*

An Identifier is valid if and only if it is not one of the language's reserved words.

8.37.18 Definition: Predefined operator

Code: *DIPO*

A **predefined operator** is one of:

= /= ~ /~

Informative Text

8.37.18.1 These operators — all "special symbols" — appear in Equality expressions. Their semantics, reference or object equality or inequality, is defined by the language (although you can adapt the effect of **~** and **/~** since they follow redefinitions of *is_equal*). As a consequence you may not use them as Alias for your own features.

End

8.37.19 Definition: Standard operator

Code: *DISO*

A **standard unary operator** is one of:

+ –

A **standard binary operator** is any one of the following one- or two-character symbols:

+ – * / ^ < >

<= >= // \\ ..

Informative Text

8.37.19.1 All the standard operators appear as Operator aliases for numeric and relational features of the Kernel Library, for example *less_than* **alias** "<" in *INTEGER* and many other classes. You may also use them as Alias in your own classes.

End

8.37.20 Definition: Operator symbol

Code: *DIOS*

An **operator symbol** is any non-alpha_numeric, non-break, printable character that satisfies any of the following properties:

1 It does not appear in any of the special symbols.

2 It appears in any of the standard (unary or binary) operators but is neither a dot **.** nor an equal sign **=**.

3 It is a tilde **~**, percent **%**, or question mark **?**.

Informative Text

8.37.20.1 Condition 1 avoids ambiguities with special symbols such as quotes. Conditions 2 and 3 override it when needed: we do for example accept as operator symbols **+**, a standard operator, and **** which appears in a standard operator — but not a dot or an equal sign, which have a set meaning.

End

8.37.21 Definition: Free operator

Code: *DIFO*

A **free operator** is a sequence of one or more characters satisfying the following properties:

1 It is not a special symbol, standard operator or predefined operator.

2 Every character in the sequence is an operator symbol.

3 Every subsequence that is not a standard operator or predefined operator is distinct from all special symbols.

Signed by JV - 2026/10/8

A Free_unary is a free operator that is distinct from all standard unary operators.

A Free_binary is a free operator that is distinct from all standard binary operators.

Informative Text

8.37.21.1 Condition 3 gives us maximum flexibility without ambiguity; for example:

8.37.21.1.1 - You may **not** use **---** as an operator because, its subsequence **--** clashes with the special symbol introducing comments.

8.37.21.1.2 - You may similarly **not** use **--** because the full sequence (which of course is a subsequence too) could still be interpreted as making the rest of the line a comment.

8.37.21.1.3 - You **may**, however, use a single **-**, or define a free operator such as **-*** which does not cause any such confusion.

8.37.21.1.4 - You may **not** use **?**, **=** or **~**, but you **may** use operators containing these characters, for example **?=**.

8.37.21.1.5 - You **may** use a percent character **%** by itself or in connection with other operator symbols. No confusion is possible with character codes such as **%**B and **%**/123/. (If you use a percent character in an Alias specification, its occurrences in its Alias_name string must be written as **%%** according to the normal rules for special characters in strings. For example you may define a feature *remainder* **alias "%%"** to indicate that it has **%** as an Operator alias. But any use of the operator outside of such a string is written just **%**, for example in the expression *a* **%** *b* which in this case would be a shorthand for *a*.*remainder* (*b*).)

8.37.21.2 Alpha_numeric characters are not permitted. For example, you may not use *+b* as an operator: otherwise *a+b* could be understood as consisting of one identifier and one operator.

End

8.37.22 Syntax (non-production): Manifest character

Code: *SIMC*

A **manifest character** — specimen of construct Character — is one of the following:

1. Any key associated with a printable character, except for the percent key **%**.
2. The sequence **%***k*, where *k* is a one-key code taken from the list of special characters.
3. The sequence **%**/*code*/, where *code* is an unsigned integer in any of the available forms — decimal, binary, octal, hexadecimal — corresponding to a valid character code in the chosen character set.

Informative Text

8.37.22.1 Form 1 accepts any character on your keyboard, provided it matches the character set you have selected (Unicode or extended ASCII), with the exception of the percent, used as special marker for the other two cases.

8.37.22.2 Form 2 lets you use predefined percent codes, such as **%B** for backspace, for the most commonly needed special characters. The set of supported codes follows.

8.37.22.3 Form 3 allows you to denote any Unicode or Extended ASCII character by its integer code; for example **%/**59/ represents a semicolon (the character of code 59). Since listings for character codes — for example in Unicode documentation — often give them in base 16, you may use the **0x***NNN* convention for hexadecimal integers: the semicolon example can also be expressed as **%/0x**3B/, where 3B is the hexadecimal code for 59.

8.37.22.4 Since the three cases define all the possibilities, a percent sign is illegal in a context expecting a Character unless immediately followed by one of the keys of the following table or by /*code*/ where *code* is a legal character code. For example **%?** is illegal (no such special character); so is **%0x**/FFFFFF/ (not in the Unicode range).

End

8.37.23 Syntax (non-production): Special characters and their codes

Code: *SISP*

Character	*Code*	*Mnemonic name*
@	**%A**	***A**t-sign*
BS	**%B**	***B**ackspace*
^	**%C**	***C**ircumflex*
$	**%D**	***D**ollar*
FF	**%F**	***F**orm feed*
****	**%H**	*BackslasH*
~	**%L**	*Ti**L**de*
NL (LF)	**%N**	***N**ewline*
`	**%Q**	*Back**Q**uote*
CR	**%R**	*Carriage **R**eturn*
#	**%S**	***S**harp*
HT	**%T**	*Horizontal **T**ab*
NUL	**%U**	*N**U**ll*
|	**%V**	***V**ertical bar*
%	**%%**	*Percent*
'	**%'**	*Single quote*
"	**%"**	*Double quote*
[	**%(**	*Opening bracket*
]	**%)**	*Closing bracket*
{	**%<**	*Opening brace*
}	**%>**	*Closing brace*

Informative Text

8.37.23.1 A few of these codes, such as the last four, are present on many keyboards, but sometimes preempted to represent letters with diacritical marks; using **%(** rather than **[** guarantees that you always get a bracket.

End

8.37.24 Syntax (non-production): Percent variants

Code: *SIPC*

The percent forms of Character are available for the manifest characters of a Character_constant and of the Simple_string components of a Manifest_string, but not for any other token.

Informative Text

8.37.24.1 The characters “of” such a constant do not include the single **'** or double **"** quotes, which you must enter as themselves.

End

8.37.25 Semantics: Manifest character semantics

Code: *MIMC*

The value of a Character is:

1. If it is a printable character *c* other than **%**: *c*.
2. If it is of the form **%***k* for a one-key code *k*: the corresponding character as given by the table of special characters.
3. If it is of the form *%/code/*: the character of code *code* in the chosen character set.

8.37.26 Syntax (non-production): String, simple string

Code: *SIST*

A **string** — specimen of construct String — is a sequence of zero or more manifest characters.

A **simple string** — specimen of Simple_string — is a String consisting of at most one line (that is to say, containing no embedded new-line manifest character).

Signed by JV - 2026/10/8

8.37.27 Semantics: String semantics

Code: *MIST*

The value of a String or Simple_string is the sequence of the values of its characters.

8.37.28 Syntax: Integers

Code: *SIIN*

Integer ≜ [Integer_base] Digit_sequence

Integer_base ≜ "**0**" Integer_base_letter

Integer_base_letter ≜ "**b**" | "**c**" | "**x**" | "**B**" | "**C**" | "**X**"

Digit_sequence ≜ Digit^{+}

Digit ≜ "**0**" | "**1**" | "**2**" | "**3**" | "**4**" | "**5**" | "**6**" | "**7**" | "**8**" | "**9**" | "**a**" | "**b**" | "**c**" | "**d**" | "**e**" | "**f**" | "**A**" | "**B**" | "**C**" | "**D**" | "**E**" | "**F**" | "**_**"

Informative Text

8.37.28.1 To introduce an integer base, use the digit **0** (zero) followed by a letter denoting the base: **b** for binary, **c** for octal, **x** for hexadecimal. Examples are:

- [COMPLETE]

8.37.28.2 Per the Letter_Case_rule, the upper-case versions of these letters are permitted, although lower-case is the recommended style.

8.37.28.3 Similarly, you may write the hexadecimal digits of the last two lines in lower or upper case. Here upper case is the recommended style, as in **0xA5**.

End

8.37.29 Validity: Integer rule

Code: *VIIN*

An Integer is valid if and only if it satisfies the following conditions:

1 It contains no breaks.

2 Neither the first nor the last Digit of the Digit_sequence is an underscore "**_**".

3 If there is no Integer_base (decimal integer), every Digit is either one of the decimal digits **0** to **9** (zero to nine) or an underscore.

4 If there is an Integer_base of the form **0b** or **0B** (binary integer), every Digit is either **0**, **1** or an underscore.

5 If there is an Integer_base of the form **0c** or **0C** (octal integer), every Digit is either one of the octal digits **0** to **7** or an underscore.

Informative Text

8.37.29.1 The rule has no requirement for the hexadecimal case, which accepts all the digits permitted by the syntax.

8.37.29.2 Integer is a purely lexical construct and does not include provision for a sign; the construct Integer_constant denotes possibly signed integers.

End

8.37.30 Semantics: Integer semantics

Code: *MIIN*

The value of an Integer is the integer constant denoted in ordinary mathematical notation by the Digit_sequence, without its underscores if any, in the corresponding base: binary if the Integer starts with **0b** or **0B**, octal if it starts with **0c** or **0C**, hexadecimal if it starts with **0x** or **0X**, decimal otherwise.

Signed by JV - 2026/10/8

Informative Text

8.37.30.1 This definition always yields a well-defined mathematical value, regardless of the number of digits. It is only at the level of Integer_constant that the value may be flagged as invalid, for example {*NATURAL_8*} 256, or 999 … 999 with too many digits to be representable as either an *INTEGER_32* or an *INTEGER_64*.

8.37.30.2 The semantics ignores any underscores, which only serve to separate groups of digits for clarity. With decimal digits, the recommended style, if you include underscores, is to use groups of three from the right.

End

8.37.31 Syntax (non-production): Real number

Code: *SIRE*

A **real** — specimen of Real — is made of the following elements, in the order given:

- An optional decimal Integer, giving the integral part.
- A required "**.** " (dot).
- An optional decimal Integer, giving the fractional part.
- An optional exponent, which is the letter *e* or *E* followed by an optional Sign (+ or –) and a decimal Integer.

No intervening character (blank or otherwise) is permitted between these elements. The integral and fractional parts may not both be absent.

Informative Text

8.37.31.1 As with integers, you may use underscores to group the digits for readability. The recommended style uses groups of three in both the integral and decimal parts, as in *45_093_373.567_21*. If you include an exponent, *E*, rather than *e*, is the recommended form.

End

8.37.32 Semantics: Real semantics

Code: *MIRE*

The value of a Real is the real number that would be expressed in ordinary mathematical notation as $i.f\ 10^{e}$, where *i* is the integral part, *f* the fractional part and *e* the exponent (or, in each case, zero if the corresponding part is absent).

PART J: Reference

Signed by JV - 2026/10/8

Signed by JV - 2026/10/8

8.9 Syntax summary (in order of appearance)

This subclause includes all syntax productions appearing in the other clauses of the Standard, in their order of appearance.

They are collected automatically and hence identical to the originals, each from the page indicated before each production. In the electronic version, these page numbers are hyperlinks to the original appearance.

For the same information in the alphabetical order of construct names, see the next subclause (8.10).

.

78 Class_name ≜ Identifier

82 Class_declaration ≜
[Notes]
Class_header
[Formal_generics]
[Obsolete]
[Inheritance]
[Creators]
[Converters]
[Features]
[Invariant]
[Notes]
end

83 Notes ≜ **note** Note_list

83 Note_list ≜ {Note_entry ";" …}*

83 Note_entry ≜ Note_name Note_values

83 Note_name ≜ Identifier ":"

83 Note_values ≜ {Note_item ","…}+

83 Note_item ≜ Identifier | Manifest_constant

84 Class_header ≜ [Header_mark] **class** Class_name

84 Header_mark ≜ **deferred** | **expanded** | **frozen**

86 Obsolete ≜ **obsolete** Message

86 Message ≜ Manifest_string

92 Clients ≜ "**{**" Class_list "**}**"

92 Class_list ≜ {Class_name "**,**" …}+

93 New_exports ≜ **export** New_export_list

93 New_export_list ≜ {New_export_item "**;**" …}+

93 New_export_item ≜ Clients [Header_comment] Feature_set

93 Feature_set ≜ Feature_list | **all**

93 Feature_list ≜ {Feature_name "**,**" …}+

101 Features ≜ Feature_clause+

101 Feature_clause ≜ **feature** [Clients] [Header_comment] Feature_declaration_list

Signed by JV - 2026/10/8

101 Feature_declaration_list ≜ {Feature_declaration ";" …}*

101 Header_comment ≜ Comment

102 Feature_declaration ≜ New_feature_list Declaration_body

102 New_feature_list ≜ {New_feature_name "," …}+

102 New_feature_name ≜ [Emergence_mark] [Variance_mark] Extended_feature_name

102 Emergence_mark ≜ **then**

102 Variance_mark ≜ **variant** | **invariant**

104 Extended_feature_name ≜ Feature_name [Aliases]

104 Feature_name ≜ Identifier

104 Aliases ≜ {Alias "," …}+

104 Alias ≜ **alias** "'" Alias_name "'"

104 Alias_name ≜ Operator | Bracket | Parenthesis

104 Bracket ≜ "**[]**"

104 Parenthesis ≜ "**()**"

108 Operator ≜ Unary | Binary

108 Unary ≜ **not** | "+" | "–" | Free_unary

108 Binary ≜ "+" | "–" | "*" | "/" | "//" | "\\" | "^" | "**..**" |
"<" | ">" | "<=" | ">=" |
and | **or** | **xor** | **and then** | **or else** | **implies** |
Free_binary

109 Declaration_body ≜ [Formal_arguments]
[Query_mark]
[Obsolete]
[Header_comment]
[Feature_definition]

110 Query_mark ≜ [Assigner_mark] "**:**" Result_type

110 Result_type ≜ Precursor_type | Explicit_result_type

110 Explicit_result_type ≜ Type

112 Assigner_mark ≜ **assign** Feature_name

113 Feature_definition ≜ [Precondition]
[Local_declarations]
[Feature_body]
[Postcondition]
[Rescue]
[**end**]

114 Feature_body ≜ Deferred | Effective

114 Effective ≜ Effective_routine | Constant_attribute | Defined_attribute

114 Constant_attribute ≜ "**=**" Manifest_constant

117 Defined_attribute ≜ **attribute** Compound

121 Formal_arguments ≜ "(" Entity_declarations ")"

Signed by JV - 2026/10/8

121 Entity_declarations ≜ Precursor_type | Entity_declaration_list
121 Entity_declaration_list ≜ {Entity_declaration_group ";" …}+
121 Entity_declaration_group ≜ Entity_list ":" Result_type
121 Entity_list ≜ {Identifier_with_variance "," …}+
121 Identifier_with_variance ≜ [Variance_mark] Identifier
123 Deferred ≜ **deferred**
123 Effective_routine ≜ Internal | External
123 Internal ≜ Routine_mark Compound
123 Routine_mark ≜ **do** | Once
123 Once ≜ **once** ["("Key_list")"]
123 Key_list ≜ {Key "," …}+
123 Key ≜ Manifest_string
123 Local_declarations ≜ **local** [Entity_declaration_list]
124 Compound ≜ {Instruction ";" …}*
124 Instruction ≜ Creation_instruction |
Call |
Assignment |
Assigner_call |
Conditional_instruction |
Multi_branch |
Loop |
Debug |
Precursor_call |
Check
128 Precondition ≜ {Individual_precondition …}*
128 Postcondition ≜ {Individual_postcondition …}*
128 Invariant ≜ {Individual_invariant …}*
128 Individual_precondition ≜ **require** [**else**] Assertion
128 Individual_postcondition ≜ **ensure** [**then**] Assertion
128 Individual_invariant ≜ **invariant** Assertion
128 Assertion ≜ [Key] Assertion_clause_list [Implication]
128 Assertion_clause_list ≜ {Assertion_clause ";" …}*
128 Assertion_clause ≜ [Tag_mark] Unlabeled_assertion_clause
128 Unlabeled_assertion_clause ≜ Boolean_expression | Comment
128 Tag_mark ≜ Tag ":"
128 Tag ≜ Identifier
128 Implication ≜ **implies** Implication_list
128 Implication_list ≜ {Implication_clause ";" …}*
128 Implication_clause ≜ Tag_list ":" Tag_list
128 Tag_list ≜ {Tag "," …}*

130 Check ≜ **check** Assertion [Notes] [Check_then] **end**

130 Check_then ≜ **then** Compound

131 Variant ≜ **variant** [Tag_mark] Expression

131 Old ≜ **old** Expression

143 Inheritance ≜ Inherit_clause$^+$

143 Inherit_clause ≜ **inherit** [Non_conformance] Parent_list

143 Non_conformance ≜ "**{**" *NONE* "**}**"

143 Parent_list ≜ {Parent ";" …}$^+$

143 Parent =Class_type [Feature_adaptation]

143 Feature_adaptation ≜
[Undefine]
[Redefine]
[Rename]
[New_exports]
[Select]
end

147 Rename ≜ **rename** Rename_list

147 Rename_list ≜ {Rename_pair "," …}$^+$

147 Rename_pair ≜ Feature_name **as** Extended_feature_name

150 Precursor_call ≜ Precursor_type [Actuals]

150 Precursor_type ≜ **Precursor** [Parent_qualification]

150 Parent_qualification ≜ "**{**" Class_name "**}**"

152 Redefine ≜ **redefine** Feature_list

152 Undefine ≜ **undefine** Feature_list

160 Select ≜ **select** Feature_list

165 Type ≜ Class_or_tuple_type | Formal_generic_name | Anchored

165 Class_or_tuple_type ≜ Class_type | Tuple_type

165 Class_type ≜ [Attachment_mark] Class_name [Actual_generics]

165 Attachment_mark ≜ **detachable** | **attached**

165 Anchored ≜ [Attachment_mark] **like** Anchor

165 Anchor ≜ Feature_name | **Current**

173 Actual_generics ≜ "**[**" Type_list "**]**"

173 Type_list ≜ {Type "**,**" …}$^+$

174 Formal_generics ≜ "[" Formal_generic_list "]"

174 Formal_generic_list ≜ {Formal_generic ","…}$^+$

174 Formal_generic ≜ [Variance_mark] Formal_generic_name [Constraint]

174 Formal_generic_name ≜ [**detachable**] Identifier

176 Constraint ≜ "**–>**" Constraining_types [Constraint_creators]

176 Constraining_types ≜ Single_constraint | Multiple_constraint

Signed by JV - 2026/10/8

176 Single_constraint ≜ Type [Renaming]
176 Renaming ≜ Rename **end**
176 Multiple_constraint ≜ "**{**" Constraint_list "**}**"
176 Constraint_list ≜ {Single_constraint "**,**" …}+
176 Constraint_creators ≜ **create** Feature_list **end** **--- CHECK THIS**
181 Tuple_type ≜ *TUPLE* [Tuple_parameter_list]
181 Tuple_parameter_list ≜ "**[**" Tuple_parameters "**]**"
181 Tuple_parameters ≜ Type_list | Entity_declaration_list
181 Manifest_tuple ≜ "**[**" Expression_list "**]**"
181 Expression_list ≜ {Expression "**,**" …}
194 Converters ≜ **convert** Converter_list
194 Converter_list ≜ {Converter"**,**"…}+
194 Converter ≜ Conversion_procedure | Conversion_query
194 Conversion_procedure ≜ Feature_name "**(**" "**{**" Type_list "**}**" "**)**"
194 Conversion_query ≜ Feature_name "**:**" "**{**" Type_list "**}**"
200 Call ≜ Object_call | Non_object_call
200 Object_call ≜ [Target "**.**"] Unqualified_call
200 Unqualified_call ≜ Feature_name [Actuals]
200 Target ≜ Local |
Read_only |
Call |
Parenthesized |
Bracket_expression |
Paren_expression |
Precursor_call
200 Non_object_call ≜ "**{**" Type "**}**" "**.**" Unqualified_call
202 Actuals ≜ "**(**" Actual_list "**)**"
202 Actual_list ≜ {Expression "**,**" …}+
211 Agent ≜ Call_agent | Inline_agent
211 Call_agent ≜ **agent** Call_agent_body
211 Inline_agent ≜ **agent**
[Formal_arguments]
[Result_type]
[Feature_definition]
[Agent_actuals]
211 Call_agent_body ≜ Agent_qualified | Agent_unqualified
211 Agent_qualified ≜ Agent_target "**.** " Agent_unqualified

211 Agent_unqualified ≜ Feature_name [Agent_actuals]

211 Agent_target ≜ Entity | Parenthesized | Manifest_type

211 Agent_actuals ≜ "**(**" Agent_actual_list "**)**"

211 Agent_actual_list ≜ {Agent_actual "**,**" …}$^{+}$

211 Agent_actual ≜ Expression | Placeholder

211 Placeholder ≜ [Manifest_type] "**?**"

215 Expression ≜ Detachable_expression | Attached_expression | **Void**

215 Detachable_expression ≜ Read_only |
Local |
Call |
Precursor_call |
Parenthesized |
Old |
Operator_expression |
Bracket_expression |
Paren_expression |
Conditional_expression

215 Attached_expression ≜ Manifest_constant |
Manifest_tuple |
Equality |
Agent |
Creation_expression |
Object_test |
Manifest_string |
Address

215 Parenthesized ≜ "**(**" Expression "**)**"

215 Address ≜ "**$**" Variable

215 Once_string ≜ **once** Manifest_string

215 Boolean_expression ≜ **once** Manifest_string

217 Operator_expression ≜ Unary_expression | Binary_expression

217 Unary_expression ≜ Unary Expression

217 Binary_expression ≜ Expression Binary Expression

219 Bracket_expression ≜ Bracket_or_paren_target "**[**" Actuals "**]**"

219 Bracket_or_paren_target ≜ Target | Once_string | Manifest_constant | Manifest_tuple

219 Paren_expression ≜ Bracket_or_paren_target "**(**" Actuals "**)**"

219 Conditional_expression ≜ **if** Then_case_list [Else_case] **end**

219 Then_case_list ≜ {Then_case **elseif** …}$^{+}$

219 Then_case ≜ Boolean_expression **then** Expression

219 Else_case ≜ **else** Expression

224 Conditional_instruction ≜ **if** Then_part_list [Else_part] **end**

224 Then_part_list ≜ {Then_part **elseif** …}$^{+}$

224 Then_part ≜ Boolean_expression **then** Compound

Signed by JV - 2026/10/8

224 Else_part ≜ **else** Compound

225 Multi_branch ≜ **inspect** Expression [When_part_list] [Else_part] **end**

225 When_part_list ≜ When_part^{+}

225 When_part ≜ **when** Choices **then** Compound

225 Choices ≜ {Choice "**,**" …}$^{+}$

225 Choice ≜ Constant | Manifest_type | Constant_interval| Type_interval

225 Constant_interval ≜ Constant "**..**" Constant

226 Type_interval ≜ Manifest_type "**..**" Manifest_type

228 Loop ≜
[Iteration]
[Initialization]
[Invariant]
[Exit_condition]
Loop_body
[Variant]
end

228 Iteration ≜ **across** Expression **as** Identifier

228 Initialization ≜ **from** Compound

228 Exit_condition ≜ **until** Boolean_expression

228 Loop_body ≜ Do_body | All_body | Some_body

228 Do_body ≜ **loop** Compound

228 All_body ≜ **all** Boolean_expression

228 Some_body ≜ **some** Boolean_expression

230 Debug ≜ **debug** ["**(**"Key_list "**)**"] Compound **end**

232 Creators ≜ Creation_clause^{+}

232 Creation_clause ≜ **create** [Clients] [Header_comment] Creation_procedure_list

232 Creation_procedure_list ≜ {Creation_procedure "**,**"…}$^{+}$

232 Creation_procedure ≜ Feature_name

234 Creation_instruction ≜ **create** [Explicit_creation_type] Creation_call

234 Explicit_creation_type ≜ "**{**" Type "**}**"

234 Creation_call ≜ Variable [Explicit_creation_call]

234 Explicit_creation_call ≜ "**.**" Unqualified_call

237 Creation_expression ≜ **create** Explicit_creation_type [Explicit_creation_call]

239 Assignment ≜ Variable ":=" Expression

241 Assigner_call ≜ Expression ":=" Expression

245 Entity ≜ Variable | Read_only

245 Variable ≜ Variable_attribute | Local

245 Variable_attribute ≜ Feature_name

245 Local ≜ Identifier | **Result** | **Retry**

245 Read_only ≜ Formal | Constant_name | **Current**

Signed by JV - 2026/10/8

245 Formal ≜ Identifier

252 Equality ≜ Expression Comparison Expression

252 Comparison ≜ "**=**" | "**/=**" | "**~**" | "**/~**"

254 Object_test ≜ **attached** ["**{**" Type "**}**"] [**as** Identifier] Expression

260 Rescue ≜ **rescue** Compound

266 Constant ≜ Manifest_constant | Constant_name

266 Constant_name ≜ Feature_name

266 Manifest_constant ≜ [Manifest_type] Manifest_value

266 Manifest_type ≜ "**{**" Type "**}**"

266 Manifest_value ≜ Boolean_constant |
Character_constant |
Integer_constant |
Real_constant |
Manifest_string |
Manifest_type

266 Sign ≜ "**+**" | "**–**"

266 Integer_constant ≜ [Sign] Integer

266 Character_constant ≜ "**'**" Character "**'**"

266 Boolean_constant ≜ **True** | **False**

266 Real_constant ≜ [Sign] Real

268 Manifest_string ≜ Basic_manifest_string| Verbatim_string

268 Basic_manifest_string ≜ ' **"** ' String_content ' **"** '

268 String_content ≜ {Simple_string Line_wrapping_part …}$^{+}$

268 Verbatim_string ≜ Verbatim_string_opener Line_sequence Verbatim_string_closer

268 Verbatim_string_opener ≜ ' **"** ' [Simple_string] Open_bracket

268 Verbatim_string_closer ≜ Close_bracket [Simple_string] ' **"** '

268 Open_bracket ≜ "**[**" | "**{**"

268 Close_bracket ≜ "**]**" | "**}**"

273 External ≜ **external** External_language [External_name]

273 External_language ≜ Unregistered_language | Registered_language

273 Unregistered_language ≜ Manifest_string

273 External_name ≜ **alias** Manifest_string

274 Registered_language ≜ C_external | C++_external | DLL_external

274 External_signature ≜ **signature** [External_argument_types] [**:** External_type]

274 External_argument_types ≜ "**(**" External_type_list "**)**"

274 External_type_list ≜ {External_type "**,**" …}*

274 External_type ≜ Simple_string

275 External_file_use ≜ **use** External_file_list

Signed by JV - 2026/10/8

275 External_file_list ≜ {External_file "," …}+

275 External_file ≜ External_user_file | External_system_file

275 External_user_file ≜ ' " 'Simple_string ' " '

275 External_system_file ≜ "<"Simple_string ">"

276 C_external ≜ ’' " ' **C**
’[**inline**]
[External_signature] [External_file_use]
' " '

277 C++_external ≜ ' " ' **C++**
inline
[External_signature]
[External_file_use]
' " '

278 DLL_external ≜ ' " ' **dll**
[**windows**]
DLL_identifier
[DLL_index]
[External_signature]
[External_file_use]
' " '

278 DLL_identifier ≜ Simple_string

278 DLL_index ≜ Integer

280 Comment ≜ "– –" {Simple_string Comment_break …}*

280 Comment_break ≜ New_line [Blanks_or_tabs] "– –"

286 Integer ≜ [Integer_base] Digit_sequence

286 Integer_base ≜ "**0**" Integer_base_letter

286 Integer_base_letter ≜ "**b**" | "**c**" | "**x**" | "**B**" | "**C**" | "**X**"

286 Digit_sequence ≜ Digit +

286 Digit ≜ "**0**" | "**1**" | "**2**" | "**3**" | "**4**" | "**5**" | "**6**" | "**7**" | "**8**" | "**9**" |
"**a**" | "**b**" | "**c**" | "**d**" | "**e**" | "**f**" |
"**A**" | "**B**" | "**C**" | "**D**" | "**E**" | "**F**" | "**_**"

Signed by JV - 2026/10/8

8.10 Syntax summary (in alphabetical order of constructs)

This subclause includes all syntax productions appearing in the other clauses of the Standard, in alphabetical order of constructed names.

The productions are collected automatically and hence identical to the originals, each from the page indicated before each production. In the electronic version, these page numbers are hyperlinks to the original appearance.

For the same information in the order of appearance of constructs in the rest of the text, see the preceding subclause (8.9).

[Draft note: for the moment the productions are in the original order, not in alphabetical order. They need to be sorted for the final version.]

78 Class_name ≜ Identifier

82 Class_declaration ≜
[Notes]
Class_header
[Formal_generics]
[Obsolete]
[Inheritance]
[Creators]
[Converters]
[Features]
[Invariant]
[Notes]
end

83 Notes ≜ **note** Note_list

83 Note_list ≜ {Note_entry ";" …}*

83 Note_entry ≜ Note_name Note_values

83 Note_name ≜ Identifier ":"

83 Note_values ≜ {Note_item ","…}$^{+}$

83 Note_item ≜ Identifier | Manifest_constant

84 Class_header ≜ [Header_mark] **class** Class_name

84 Header_mark ≜ **deferred** | **expanded** | **frozen**

86 Obsolete ≜ **obsolete** Message

86 Message ≜ Manifest_string

92 Clients ≜ "**{**" Class_list "**}**"

92 Class_list ≜ {Class_name "**,**" …}$^{+}$

93 New_exports ≜ **export** New_export_list

93 New_export_list ≜ {New_export_item "**;**" …}$^{+}$

93 New_export_item ≜ Clients [Header_comment] Feature_set

93 Feature_set ≜ Feature_list | **all**

93 Feature_list ≜ {Feature_name "**,**" …}$^{+}$

101 Features ≜ Feature_clause^{+}

Signed by JV - 2026/10/8

101 Feature_clause ≜ **feature** [Clients] [Header_comment] Feature_declaration_list

101 Feature_declaration_list ≜ {Feature_declaration **";"** …}*

101 Header_comment ≜ Comment

102 Feature_declaration ≜ New_feature_list Declaration_body

102 New_feature_list ≜ {New_feature_name "," …}+

102 New_feature_name ≜ [Emergence_mark] [Variance_mark] Extended_feature_name

102 Emergence_mark ≜ **then**

102 Variance_mark ≜ **variant** | **invariant**

104 Extended_feature_name ≜ Feature_name [Aliases]

104 Feature_name ≜ Identifier

104 Aliases ≜ {Alias "," …}+

104 Alias ≜ **alias** **'"'** Alias_name **'"'**

104 Alias_name ≜ Operator | Bracket | Parenthesis

104 Bracket ≜ "**[]**"

104 Parenthesis ≜ "**()**"

108 Operator ≜ Unary | Binary

108 Unary ≜ **not** | "**+**" | "**–**" | Free_unary

108 Binary ≜ "**+**" | "**–**" | "*****" | "**/**" | "**//**" | "****" | "**^**" | "**. .**" |
"**<**" | "**>**" | "**<=**" | "**>=**" |
and | **or** | **xor** | **and then** | **or else** | **implies** |
Free_binary

109 Declaration_body ≜
[Formal_arguments]
[Query_mark]
[Obsolete]
[Header_comment]
[Feature_definition]

110 Query_mark ≜ [Assigner_mark] "**:**" Result_type

110 Result_type ≜ Precursor_type | Explicit_result_type

110 Explicit_result_type ≜ Type

112 Assigner_mark ≜ **assign** Feature_name

113 Feature_definition ≜
[Precondition]
[Local_declarations]
[Feature_body]
[Postcondition]
[Rescue]
[**end**]

114 Feature_body ≜ Deferred | Effective

114 Effective ≜ Effective_routine | Constant_attribute | Defined_attribute

114 Constant_attribute ≜ "**=**" Manifest_constant

117 Defined_attribute ≜ **attribute** Compound

Signed by JV - 2026/10/8

121 Formal_arguments ≜ "(" Entity_declarations ")"

121 Entity_declarations ≜ Precursor_type | Entity_declaration_list

121 Entity_declaration_list ≜ {Entity_declaration_group ";" ...}+

121 Entity_declaration_group ≜ Entity_list ":" Result_type

121 Entity_list ≜ {Identifier_with_variance "," ...}+

121 Identifier_with_variance ≜ [Variance_mark] Identifier

123 Deferred ≜ **deferred**

123 Effective_routine ≜ Internal | External

123 Internal ≜ Routine_mark Compound

123 Routine_mark ≜ **do** | Once

123 Once ≜ **once** ["("Key_list")"]

123 Key_list ≜ {Key "," ...}+

123 Key ≜ Manifest_string

123 Local_declarations ≜ **local** [Entity_declaration_list]

124 Compound ≜ {Instruction ";" ...}*

124 Instruction ≜ Creation_instruction |
Call |
Assignment |
Assigner_call |
Conditional_instruction |
Multi_branch |
Loop |
Debug |
Precursor_call |
Check

128 Precondition ≜ {Individual_precondition ...}*

128 Postcondition ≜ {Individual_postcondition ...}*

128 Invariant ≜ {Individual_invariant ...}*

128 Individual_precondition ≜ **require** [**else**] Assertion

128 Individual_postcondition ≜ **ensure** [**then**] Assertion

128 Individual_invariant ≜ **invariant** Assertion

128 Assertion ≜ [Key] Assertion_clause_list [Implication]

128 Assertion_clause_list ≜ {Assertion_clause ";" ...}*

128 Assertion_clause ≜ [Tag_mark] Unlabeled_assertion_clause

128 Unlabeled_assertion_clause ≜ Boolean_expression | Comment

128 Tag_mark ≜ Tag ":"

128 Tag ≜ Identifier

128 Implication ≜ **implies** Implication_list

128 Implication_list ≜ {Implication_clause ";" ...}*

128 Implication_clause ≜ Tag_list ":" Tag_list

128 Tag_list ≜ {Tag "," ...}*

130 Check ≜ **check** Assertion [Notes] [Check_then] **end**

130 Check_then ≜ **then** Compound

131 Variant ≜ **variant** [Tag_mark] Expression

131 Old ≜ **old** Expression

143 Inheritance ≜ Inherit_clause+

143 Inherit_clause ≜ **inherit** [Non_conformance] Parent_list

143 Non_conformance ≜ "**{**" *NONE* "**}**"

143 Parent_list ≜ {Parent ";" ...}+

143 Parent =Class_type [Feature_adaptation]

143 Feature_adaptation ≜
[Undefine]
[Redefine]
[Rename]
[New_exports]
[Select]
end

147 Rename ≜ **rename** Rename_list

147 Rename_list ≜ {Rename_pair "," ...}+

147 Rename_pair ≜ Feature_name **as** Extended_feature_name

150 Precursor_call ≜ Precursor_type [Actuals]

150 Precursor_type ≜ **Precursor** [Parent_qualification]

150 Parent_qualification ≜ "**{**" Class_name "**}**"

152 Redefine ≜ **redefine** Feature_list

152 Undefine ≜ **undefine** Feature_list

160 Select ≜ **select** Feature_list

165 Type ≜ Class_or_tuple_type | Formal_generic_name | Anchored

165 Class_or_tuple_type ≜ Class_type | Tuple_type

165 Class_type ≜ [Attachment_mark] Class_name [Actual_generics]

165 Attachment_mark ≜ **detachable** | **attached**

165 Anchored ≜ [Attachment_mark] **like** Anchor

165 Anchor ≜ Feature_name | **Current**

173 Actual_generics ≜ "**[**" Type_list "**]**"

173 Type_list ≜ {Type "," ...}+

174 Formal_generics ≜ "[" Formal_generic_list "]"

174 Formal_generic_list ≜ {Formal_generic ","...}+

174 Formal_generic ≜ [Variance_mark] Formal_generic_name [Constraint]

174 Formal_generic_name ≜ [**detachable**] Identifier

176 Constraint ≜ "**–>**" Constraining_types [Constraint_creators]

Signed by JV - 2026/10/8

176 Constraining_types ≜ Single_constraint | Multiple_constraint
176 Single_constraint ≜ Type [Renaming]
176 Renaming ≜ Rename **end**
176 Multiple_constraint ≜ "**{**" Constraint_list "**}**"
176 Constraint_list ≜ {Single_constraint "**,**" …}$^{+}$
176 Constraint_creators ≜ **create** Feature_list **end** **--- CHECK THIS**
181 Tuple_type ≜ *TUPLE* [Tuple_parameter_list]
181 Tuple_parameter_list ≜ "**[**" Tuple_parameters "**]**"
181 Tuple_parameters ≜ Type_list | Entity_declaration_list
181 Manifest_tuple ≜ "**[**" Expression_list "**]**"
181 Expression_list ≜ {Expression "**,**" …}
194 Converters ≜ **convert** Converter_list
194 Converter_list ≜ {Converter"**,**"…}$^{+}$
194 Converter ≜ Conversion_procedure | Conversion_query
194 Conversion_procedure ≜ Feature_name "**(**" "**{**" Type_list "**}**" "**)**"
194 Conversion_query ≜ Feature_name "**:**" "**{**" Type_list "**}**"
200 Call ≜ Object_call | Non_object_call
200 Object_call ≜ [Target "**.**"] Unqualified_call
200 Unqualified_call ≜ Feature_name [Actuals]
200 Target ≜ Local |
Read_only |
Call |
Parenthesized |
Bracket_expression |
Paren_expression |
Precursor_call
200 Non_object_call ≜ "**{**" Type "**}**" "**.**" Unqualified_call
202 Actuals ≜ "**(**" Actual_list "**)**"
202 Actual_list ≜ {Expression "**,**" …}$^{+}$
211 Agent ≜ Call_agent | Inline_agent
211 Call_agent ≜ **agent** Call_agent_body
211 Inline_agent ≜ **agent**
[Formal_arguments]
[Result_type]
[Feature_definition]
[Agent_actuals]
211 Call_agent_body ≜ Agent_qualified | Agent_unqualified

211 Agent_qualified ≜ Agent_target "**.** " Agent_unqualified

211 Agent_unqualified ≜ Feature_name [Agent_actuals]

211 Agent_target ≜ Entity | Parenthesized | Manifest_type

211 Agent_actuals ≜ "**(**" Agent_actual_list "**)**"

211 Agent_actual_list ≜ {Agent_actual "**,**" …}$^{+}$

211 Agent_actual ≜ Expression | Placeholder

211 Placeholder ≜ [Manifest_type] "**?**"

215 Expression ≜ Detachable_expression | Attached_expression | **Void**

215 Detachable_expression ≜ Read_only |
Local |
Call |
Precursor_call |
Parenthesized |
Old |
Operator_expression |
Bracket_expression |
Paren_expression |
Conditional_expression

215 Attached_expression ≜ Manifest_constant |
Manifest_tuple |
Equality |
Agent |
Creation_expression |
Object_test |
Manifest_string |
Address

215 Parenthesized ≜ "**(**" Expression "**)**"

215 Address ≜ "**$**" Variable

215 Once_string ≜ **once** Manifest_string

215 Boolean_expression ≜ **once** Manifest_string

217 Operator_expression ≜ Unary_expression | Binary_expression

217 Unary_expression ≜ Unary Expression

217 Binary_expression ≜ Expression Binary Expression

219 Bracket_expression ≜ Bracket_or_paren_target "**[**" Actuals "**]**"

219 Bracket_or_paren_target ≜ Target | Once_string | Manifest_constant | Manifest_tuple

219 Paren_expression ≜ Bracket_or_paren_target "**(**" Actuals "**)**"

219 Conditional_expression ≜ **if** Then_case_list [Else_case] **end**

219 Then_case_list ≜ {Then_case **elseif** …}$^{+}$

219 Then_case ≜ Boolean_expression **then** Expression

219 Else_case ≜ **else** Expression

224 Conditional_instruction ≜ **if** Then_part_list [Else_part] **end**

224 Then_part_list ≜ {Then_part **elseif** …}$^{+}$

Signed by JV - 2026/10/8

224 Then_part ≜ Boolean_expression **then** Compound

224 Else_part ≜ **else** Compound

225 Multi_branch ≜ **inspect** Expression [When_part_list] [Else_part] **end**

225 When_part_list ≜ When_part^{+}

225 When_part ≜ **when** Choices **then** Compound

225 Choices ≜ {Choice "**,**" …}$^{+}$

225 Choice ≜ Constant | Manifest_type | Constant_interval| Type_interval

225 Constant_interval ≜ Constant "**..**" Constant

226 Type_interval ≜ Manifest_type "**..**" Manifest_type

228 Loop ≜
[Iteration]
[Initialization]
[Invariant]
[Exit_condition]
Loop_body
[Variant]
end

228 Iteration ≜ **across** Expression **as** Identifier

228 Initialization ≜ **from** Compound

228 Exit_condition ≜ **until** Boolean_expression

228 Loop_body ≜ Do_body | All_body | Some_body

228 Do_body ≜ **loop** Compound

228 All_body ≜ **all** Boolean_expression

228 Some_body ≜ **some** Boolean_expression

230 Debug ≜ **debug** ["**(**"Key_list "**)**"] Compound **end**

232 Creators ≜ Creation_clause^{+}

232 Creation_clause ≜ **create** [Clients] [Header_comment] Creation_procedure_list

232 Creation_procedure_list ≜ {Creation_procedure "**,**"…}$^{+}$

232 Creation_procedure ≜ Feature_name

234 Creation_instruction ≜ **create** [Explicit_creation_type] Creation_call

234 Explicit_creation_type ≜ "**{**" Type "**}**"

234 Creation_call ≜ Variable [Explicit_creation_call]

234 Explicit_creation_call ≜ "**.**" Unqualified_call

237 Creation_expression ≜ **create** Explicit_creation_type [Explicit_creation_call]

239 Assignment ≜ Variable ":=" Expression

241 Assigner_call ≜ Expression ":=" Expression

245 Entity ≜ Variable | Read_only

245 Variable ≜ Variable_attribute | Local

245 Variable_attribute ≜ Feature_name

245 Local ≜ Identifier | **Result** | **Retry**

Signed by JV - 2026/10/8

245 Read_only ≜ Formal | Constant_name | **Current**

245 Formal ≜ Identifier

252 Equality ≜ Expression Comparison Expression

252 Comparison ≜ "**=**" | "**/=**" | "**~**" | "**/~**"

254 Object_test ≜ **attached** ["**{**" Type "**}**"] [**as** Identifier] Expression

260 Rescue ≜ **rescue** Compound

266 Constant ≜ Manifest_constant | Constant_name

266 Constant_name ≜ Feature_name

266 Manifest_constant ≜ [Manifest_type] Manifest_value

266 Manifest_type ≜ "**{**" Type "**}**"

266 Manifest_value ≜ Boolean_constant |
Character_constant |
Integer_constant |
Real_constant |
Manifest_string |
Manifest_type

266 Sign ≜ "**+**" | "**–**"

266 Integer_constant ≜ [Sign] Integer

266 Character_constant ≜ "**'**" Character "**'**"

266 Boolean_constant ≜ **True** | **False**

266 Real_constant ≜ [Sign] Real

268 Manifest_string ≜ Basic_manifest_string| Verbatim_string

268 Basic_manifest_string ≜ ' **"** ' String_content ' **"** '

268 String_content ≜ {Simple_string Line_wrapping_part …}$^{+}$

268 Verbatim_string ≜ Verbatim_string_opener Line_sequence Verbatim_string_closer

268 Verbatim_string_opener ≜ ' **"** ' [Simple_string] Open_bracket

268 Verbatim_string_closer ≜ Close_bracket [Simple_string] ' **"** '

268 Open_bracket ≜ "**[**" | "**{**"

268 Close_bracket ≜ "**]**" | "**}**"

273 External ≜ **external** External_language [External_name]

273 External_language ≜ Unregistered_language | Registered_language

273 Unregistered_language ≜ Manifest_string

273 External_name ≜ **alias** Manifest_string

274 Registered_language ≜ C_external | C++_external | DLL_external

274 External_signature ≜ **signature** [External_argument_types] [**:** External_type]

274 External_argument_types ≜ "**(**" External_type_list "**)**"

274 External_type_list ≜ {External_type "**,**" …}*

274 External_type ≜ Simple_string

Signed by JV - 2026/10/8

275 External_file_use ≜ **use** External_file_list

275 External_file_list ≜ {External_file "**,**" …}+

275 External_file ≜ External_user_file | External_system_file

275 External_user_file ≜ ' **"** 'Simple_string ' **"** '

275 External_system_file ≜ "**<**"Simple_string "**>**"

276 C_external ≜ ’' **"** ' **C**
’[**inline**]
[External_signature] [External_file_use]
' **"** '

277 C++_external ≜ ' **"** ' **C++**
inline
[External_signature]
[External_file_use]
' **"** '

278 DLL_external ≜ ' **"** ' **dll**
[**windows**]
DLL_identifier
[DLL_index]
[External_signature]
[External_file_use]
' **"** '

278 DLL_identifier ≜ Simple_string

278 DLL_index ≜ Integer

280 Comment ‘ ≜ "**– –**" {Simple_string Comment_break …}*

280 Comment_break ≜ New_line [Blanks_or_tabs] "**– –**"

286 Integer ≜ [Integer_base] Digit_sequence

286 Integer_base ≜ "**0**" Integer_base_letter

286 Integer_base_letter ≜ "**b**" | "**c**" | "**x**" | "**B**" | "**C**" | "**X**"

286 Digit_sequence ≜ Digit +

286 Digit ≜ "**0**" | "**1**" | "**2**" | "**3**" | "**4**" | "**5**" | "**6**" | "**7**" | "**8**" | "**9**" |
"**a**" | "**b**" | "**c**" | "**d**" | "**e**" | "**f**" |
"**A**" | "**B**" | "**C**" | "**D**" | "**E**" | "**F**" | "**_**"

Signed by JV - 2026/10/8

Index

Page numbers in *italics* refer to the (informative only) introduction of clauses 1 to 7.

A
abstract data type ***35***
actual generic parameter ***45***
alias ***39***, ***49***
all ***46***
ancestors of a class ***45***
as ***47***
assertion ***40–43***
  weakening and strengthening in descendants ***49***
attached object ***36***
attribute ***37***

B
basic types ***40***

C
class ***36***, ***36–39***
  ancestors ***45***
  clients ***36***
  descendants ***45***
  heirs ***45***
  needs another ***53***
  parents ***45***
class ***38***
class invariant ***40***
client ***36***
cluster ***53***
constrained, see genericity
contract ***42***, ***43***
creation ***41***
creation instruction ***37***
Current (entity denoting the current object) ***39***
current object ***39***

D
default value for each type ***38***
deferred ***50***
deferred class ***50***–??
delegation ***87***
descendants of a class ***45***
design ***51***
Design by Contract ***42–43***, ***48***
  subcontracting ***48***
direct instance ***36***
do ***38***
dynamic binding ***47–49***
  definition ***48***

E
effective ***50***
efficiency of Eiffel ***35***
Eiffel
  organization of Eiffel software ***53***
Eiffel language ??–***57***
else ***44***
end ***37***
ensure ***41***
entity ***36***
exception ***43–44***
export ***45***
extendibility ***35***

F
failure of a routine ***43***
false ***44***
feature ***37***
feature ***37***
formal generic parameter ***45***
function ***37***

G
genericity ***44***, ***49***
  combined with inheritance ***49***
  constrained ***49***
  unconstrained ***45***

H
heir ***45***

I
if ***37***
infix operator ***39***
inherit ***45***
inheritance ***45***–??
  combined with genericity ***49***
initialization ***38***
instance ***35***, ***36***, ***37***, ***38***, ***39***, ***40***, ***51***
  current ***39***
invariant ***40***
invariant ***40***

L
Lace ***57***
local ***44***

N
needed class ***53***
needs (relation between classes) ***53***
NONE ***37***, ***38***

O
object-oriented design ***35–36***, ***51***
old ***41***
openness ***35***
operator, prefix or infix ***39***

P
parameter, see formal generic parameter, actual generic parameter (for routines the terminology is “argument”)
parent ***45***
polymorphic data structure ***49***
polymorphism ***47–49***
portability ***35***
postcondition ***40***
precondition ***40***
prefix operator ***39***
procedure ***37***
program, see system

R
redeclaration ***50***
redefine ***47***
redefinition ***47***
reference type ***39***
rename ***47***

Signed by JV - 2026/10/8

require *41*
rescue *44*
Result *38*
retry *44*
reusability *35*, *92*
root class *53*
root creation procedure *53*
root object *53*
routine *37*

S
short form *43*
strong typing *35*, *48*
subcontracting *48*
system *53–57*

T
then *37*
true *44*
type *39–40*
  basic *40*
  expanded, see expanded type
  reference, see reference type

U
unconstrained, see genericity
universe *53*

Signed by JV - 2026/10/8

Signed by JV - 2026/10/8

This generated list helps for the draft (it makes it possible to find all hyperlinked concepts very easily).

IT IS NOT PART OF THE STANDARD AND WILL BE REMOVED FROM THE FINAL VERSION

-- END compatibility for expressions 189
-- END compatibility for types 189
-- START compatibility for expressions 189
-- START compatibility for types 189
xx
Actual 121
actual 121
Actual argument 121
actual argument 121
Actual arguments 121
actual arguments 121
actual generic parameter 173
Actual generic parameters 173
actual generic parameters 173
Actuals 121
actuals 121
Address Expression rule 274
Address semantics 272
Address semantics 274
agent 211
agent call semantics 214
agents 211
aggregate 63
Aggregate production 63
Alias Validity rule 105
alias-valid 105
alpha_betic 279
alpha_betic characters 279
alpha_numeric 279
alpha_numeric characters 279
ancestor 145
ancestor type 145
ancestor type of a class 145
ancestor type of a type 145
Ancestor types 145
ancestor types 145
ancestor types of a class 145
ancestor types of a type 145
Ancestors 145
ancestors 145
anchor 169
anchor set 169
anchored 169
anchored declaration 169
anchored entity 169
anchored entity 169
anchored type 169
Anchored Type rule 170
Anchored types 169
anchored types 169
applicable parent 151
Argument rule 204
argument signature 111
argument signature of a feature 111
argument-valid 204
as deferred 149
as effective 149
assertion evaluation 138
assertion extension 135
Assertion Implication rule 133
assertion monitoring 139
assertion monitoring level 139
assertion monitoring levels 139
assertion semantics 139
Assertion semantics rule 139
Assertion Syntax rule 129
assertion violation' 139
Assigner Call rule 242
Assigner Call semantics 242
Assigner command 113
assigner command 113
Assigner Command rule 113
assigner commands 113
ASSIGNMENT RULE 239
Assignment semantics 241
associated feature 212
Associated feature of an inline agent 212
associated feature of an inline agent 212
associated variable 131
associated variable semantics 138
attached 171
attached 240
attached 243
attached 244
attached 257
Attached entity 240
attached entity 240
attached expression 257
attached reference 243
attached references 243
attached to (THIS IS FOR A REFERENCE) 243
attached to (THIS IS FOR A VALUE) 244
Attached to 240
attached to 240
attached to 240
Attached to 243
attached to 243
Attached to 244
attached to 244
attached to an object 240
attached type 171
Attached type semantics 171
Attached types 171
attached types 171
attaches 240
attaching 240
Attaching an entity 240
attaching an entity 240
attaching the entity 240
Attribute 115
attribute 115
Attributes 115
attributes 115
Availability 95
availability 95
availability for creation 233
available 134
available 233
available 95
available 95
available assertion 134
available for a call 95
available for call 95
available for calls 95
available for creation 233
available for creation to 233
available to 134
available to 95
base class 166
base class 166
base class 167
base class of a type 166
base class of a type 166
base class of an entity 167
base class of an expression 167
base class of an expression 167
Base Class principle 166
Base classes 166
base classes 166
base type 166
base type 167
base type 167
base type 179
base type 179
base type 180
base type of a multi-constraint formal generic 180
base type of a multi-constraint formal generic type 180
base type of a single-constrained formal generic 179
base type of a type 166
base type of an entity 167
base type of an expression 167
base type of an unconstrained formal generic 179
Base Type principle 166
Base types 166
base types 166
based on 166
basic 169
basic 169
Basic Syntax Description rule 65
basic type 169
basic type 169
basic type 271
Basic types 169
Basic types 169
basic types 169
basic types 169
Basic types 271
basic types 271
Blank 280
blank 280
Blanks or tabs 280
Blanks_or_tabs 280
BNF-E 59
Boolean Expression rule 221
Boolean value semantics 271
bracket 106
Bracket and Parenthesis Expression rule 219
bracket feature 106
Bracket features 106
bracket features 106
break 280
break character 280
Break characters 280
break characters 280
Break semantics 280
Breaks 280
breaks 280
C External rule 277
C Inline semantics 277
C++ External rule 277
C++ Inline semantics 277
Call agent 211
call agent 211
Call Agent rule 212
Call agents 211
call agents 211
call result 210
Call Sharing rule 159
call time 211
Call Use rule 200
call-agent equivalent 213
call-agent equivalent of an inline agent 213
CAP 257
CAPS 257
Carriage Return 280
case insensitivity 70
Case Insensitivity principle 70
cases of exception 262
Catcall 258
catcall 258
Catcalls 258
catcalls 258
Certified Attachment Pattern (CAP) 257
Certified Attachment Pattern 257
certified attachment pattern 257
Certified Attachment Patterns (CAPs) 257
character 279
Character 284
character code 285
character codes 285
character codes 285
character semantics 285
character set 279
Character Syntax rule 266
Character types semantics 271
Characters 279
characters 279
Check instruction semantics 139
check-correct 137
choice 64
Choice production 64
Clashes 150
clashes 150
Class 82
class 82
Class ANY rule 144
class consistency 137
class correctness 138
Class Header rule 85
Class Header rule 85
Class Header rule 85
Class Header rule 85
Class Identification rule 78
Class Identification rule 79
Class invariant 130
class invariant 130
class invariant 130
class name semantics 79
class transposition 155
Class Type rule 167
Class_declaration 82
classes involved in a type 169
classes it involves 169
Class-Level Call rule 206
class-transposed 155
Client 88
client 88
client of 88
client of 88
client of a class 88
client of a class 88
client of a type 88
client set 93
Client set 94
client set 94
client set OF A CLIENTS PART 93
client set OF A FEATURE 94
Clients 88
clients 88
clients 88
Clients of 88
clients of 88
closed operand 214
closed operand position 214
closed operand positions 214
closed operands 214
cluster 77
Clusters 77
clusters 77
code 285
codes 285
Combatible types 189
combined 135
combined postcondition 135
combined postcondition for 135
combined precondition 135
combined precondition and postcondition 135
combined precondition for 135
combined preconditions 135
combined preconditions and postconditions 135
command 118
Command-Query separation principle 118
Commands 118
commands 118
compatibility 189
compatibility between expressions 189
compatibility with 189
compatible 189
compatible 189
compatible expressions 189
compatible type 189
compatible types 189
compatible with 189
compatible with 189
complete garbage collector 238
completeness 238
completeness of a garbage collector 238
component 60
Components 60
components 60
composite 245
composite object 245
Compound semantics 224
Conditional instruction semantics 225
conform (in the sense of expressions) 189
conform 189
conform 190
conform 191
conform to 189
conform to 190
conform to 191
conformance 191
conformance between expressions 189
conformant-argument-valid 258
conforming 145
conforming 191
conforming parent 145
Conforming parents 145
conforming parents 145
conforming to 191
conforms 189
conforms 190
conforms 191
conforms 191
conforms 192
conforms directly 191
conforms directly 192
conforms directly 192
conforms to (as defined for expressions) 189
conforms to (expressions) 189
conforms to 189
conforms to 190
conforms to 191
conforms to 192
conjunctive 255
consistency 238
consistency of a garbage collector 238
consistent 137
consistent garbage collector 238
Constant 116
constant 116
Constant attribute 116
constant attributes 116
Constant Attribute Validity rule 118
Constant attributes 116
constant attributes 116
constant-attribute validity 118
Constants 116
constant-valid 118
constrained 175
constrained formal generic parameter 175
constrained generic class 175
constrained parameter 175
constrained parameters 175
constraining create procedures 177
constraining creation procedure 177
constraining creator 177
constraining creators 177
constraining type 176
constraining types 176
constraining types of a Formal_generic 176
constraint 176
constraint of a Formal_generic 176
constraint position 187
Constraints 176
constraints 176
construct 60
construct name convention 61
construct specimen convention 60
construction time 211
Constructs 60
constructs 60
contains 77
contains directly 77
context 249
continue 263
continued 263
continuing 263
contract 130
contract element 129
contract of a class 130
contract of a feature 130
contract of the class 130
contract of the feature 130
contract view 98
contract views 98
Conversion Asymmetry principle 194
conversion feature 194
conversion features 194
conversion from a type 195
Conversion principle 194
conversion procedure 194
Conversion Procedure rule 194
conversion procedures 194
conversion queries 194
conversion query 194
Conversion Query rule 195
Conversion semantics 196
conversion through 196
conversion through a procedure 196
conversion through a query 196
conversion to a class 195
conversion to a type 195
conversion to the type 195
conversion type 194
conversion types 194
convertibility (for expressions) 197
converting through 196
converting to a class 195
converts 195
converts 197

Signed by JV - 2026/10/8

converts explicitly 196
converts from 195
converts from a type 195
converts through 196
converts through 196
converts through a procedure 196
converts through a query 196
converts to 195
converts to 195
converts to 197
converts to a class 195
converts to a type 195
copy 244
copy semantics 244
correct 138
correct class 138
Covariance 190
covariance 190
covariance-aware form 135
Covariant 190
covariant 190
covariant argument 190
Covariant arguments 190
covariant arguments 190
creation 231
Creation Clause rule 232
Creation Clause rule 232
Creation Clause rule 232
Creation expression 237
creation expression 237
Creation Expression properties 237
creation expression properties 237
Creation Expression rule 237
Creation Expression semantics 238
Creation expressions 237
creation expressions 237
Creation instruction 234
creation instruction 234
creation instruction 237
Creation Instruction properties 235
Creation Instruction rule 235
Creation instruction semantics 236
Creation instructions 234
creation instructions 234
creation instructions 237
Creation operation 231
creation operation 231
creation operation 231
Creation operations 231
creation operations 231
Creation Precondition rule 231
Creation principle 231
creation procedure 233
creation procedure 233
creation procedure of a type 233
Creation Procedure property 233
creation procedures 233
creation procedures 233
creation procedures of a class 233
creation procedures of a type 233
creation target 234
creation target 234
creation type 234
creation type 237
creation type of a creation expression 237
Creation Type theorem 234
current class 82
current object 208
current routine 208
Current semantics 209
Current type 168
current type 168
cyclic 169
cyclic anchor 169
D2AG 63
D2BN 59
D2CH 64
D2CS 60
D2NP 65
D2PK 63
D2PR 63
D2RS 64
D2SM 68
D2TE 76
D2TE 76
D2TT 61
D2VA 67
D2VC 67
D3MC 268
D3MT 267
D3PR 270
D4BT 271
D4SS 271
D5CC 258
D7FI 72
D7LC 71
D7SC 71
D7UL 70
DAAV 134
DACA 135
DACC 137
DACE 129
DACO 130
DACP 135
DACS 138
DAIC 137
DAIV 130
DALA 133
DALC 137
DAMA 133
DAPF 136
DAPI 130
DAUL 132
DBAE 240
DBDS 241
DBDT 241
DBRE 239
DCCC 82
DCEF 85
DCRC 81
DDDE 153
DDEF 149
DDIN 149
DDNC 150
DDOR 154
DDPJ 155
DDRD 149
DDTC 156
DDTR 155
DDUD 156
DDUN 152
DDUR 151
deanchored 170
deanchored form 170
Deanchored forms 170
deanchored forms 170
Debug semantics 230
Declaration Body rule 109
declaration for 148
declaration for a feature 148
Declarations for 148
declared local 124
declared local variable 124
Declared local variables 124
declared local variables 124
Declared locals 124
declared locals 124
DECR 244
DEEC 249
DEEN 245
DEEP 248
default initialization 247
Default Initialization rule 247
default rescue original semantics 261
deferred 153
deferred 153
deferred class 153
Deferred Class property 153
deferred feature 153
Deferred features 153
deferred features 153
DEFI 244
Defined attribute 117
defined attribute 117
Defined attributes 117
defined attributes 117
DENB 244
dependency 80
depends 80
DEPS 248
DERV 243
descendant 145
Descendant Argument rule 258
Descendants 145
descendants 145
DESO 245
DEST 246
DESV 247
detachable 171
detachable type 171
Detachable types 171
detachable types 171
DEVA 243
DEVS 249
DFCA 116
DFCQ 118
DFFC 100
DFII 100
DFIN 107
DFIV 104
DFOB 106
DFOV 101
DFRK 117
DFRT 110
DFSA 111
DFSO 107
DFSY 103
DFUD 103
DFVA 115
DFVA 115
DGAP 173
DGBM 180
DGBS 179
DGBU 179
DGCC 177
DGCO 176
DGCR 178
DGFN 180
DGFP 174
DGGC 175
DGGD 175
DGRE 179
DGSI 175
DHAD 145
DHAT 145
DHCO 145
DHDF 148
DHFI 147
DHIH 145
DHIN 148
DHPA 146
DHPP 143
DHSM 144
DHUI 144
DIBR 280
DICO 280
DIFO 283
DIKE 282
DILE 279
DIOS 283
DIPO 283
direct conformance (reference types) 191
direct conformance 191
direct conformance for reference types 191
Direct conformance: expanded types 192
Direct conformance: formal generic 192
direct conformance: tuple types 192
Direct instance 165
direct instance 165
direct instance of a class 166
Direct instances 165
direct instances 165
Direct instances 166
direct instances 166
DIrect instances of a class 166
direct instances of a class 166
direct instances of a type 165
direct transposition 155
directly transposed 155
disjunctive 255
DISO 283
DISP 282
DISY 281
DJLB 182
DJSR 185
DJST 187
DJTA 185
DJTS 182
DJTT 187
DJVS 182
DKAV 233
DKCF 234
DKEX 237
DKGC 238
DKOP 231
DKPC 233
DKPT 233
DKTT 234
DKUI 234
DKUP 232
DLAV 95
DLCF 94
DLCI 91
DLCP 93
DLCR 88
DLCV 98
DLEC 90
DLES 96
DLGC 91
DLIC 98
DLSC 89
DLSP 96
DLSS 88
DLSU 88
DMDV 160
DMIF 161
DMMV 160
DMPR 161
DMRI 159
DMRP 159
DMSH 159
DMVE 160
DNCA 190
DNCE 189
DNCP 191
DNCT 189
DNEC 189
DOIN 226
DOIS 225
DOIV 227
DOPI 225
DOSP 225
double reserved word 282
Double Reserved Word rule 282
double reserved words 282
DOUI 226
DOUM 226
DPAF 212
DPCC 211
DPCE 213
DPOC 214
DPOP 213
DPPS 214
DPTT 212
DRAA 121
DRFA 122
DRLV 124
DRON 123
DRUF 122
DSCL 77
DSRT 79
DSTD 80
DSTI 77
DSUV 78
DTAN 169
DTAS 169
DTAT 171
DTBE 167
DTBT 169
DTCI 166
DTCT 168
DTDA 170
DTEX 168
DTFE 168
DTFT 167
DTIN 169
DUCT 209
DUDF 206
DUFR 207
DULA 208
DUOP 202
DUPO 202
DUSF 203
DUSF 203
DUTO 206
DUTT 202
DUVU 207
DVAE 257
DVCA 257
DVEA 257
DVER 255
DVOL 254
DVOS 255
DVRO 256
DVRS 256
DVSS 255
DWDO 220
DWET 221
DWNS 222
DWOP 217
DWPA 218
DWSE 216
DWTC 218
DXCO 261
DXET 262
DXFA 260
DXIG 263
DXRB 261
DYCA 194
DYCL 195
DYCT 195
DYCV 194
DYEX 196
dynamic 241
dynamic binding 160
dynamic binding version 160
dynamic class set 241
dynamic feature 206
dynamic feature 234
dynamic feature of 206
dynamic feature of 234
dynamic feature of a call 206
dynamic feature of a creation call 234
dynamic feature of the call 206
dynamic type 241
dynamic type set 241
dynamic types 241
DYNT 194
DYPT 194
DYQF 194
DYSS 197
DYTH 196
effect 149
effect of executing an agent call 214
effected 149
effecting 149
effectings 149
effective 153
effective 153
effective 85
effective class 153
effective class 85
Effective Class property 153
Effective classes 85
effective feature 153
Effective features 153
effective features 153
effects 149
Emergence 257
emergence 257
emergent 257
emergent attribute 257
Emergent attributes 257
emergent attributes 257
END_MARKER xx
Enter 280
Entities 245

Signed by JV - 2026/10/8

entities 245
entity 245
Entity Declaration rule 123
Entity rule 245
Entity Semantics rule 250
Equality Expression semantics 252
Equivalent Dot Form 220
equivalent dot form 220
Equivalent Dot Form of an expression 220
Equivalent Dot Forms 220
equivalent dot forms 220
evaluate 138
evaluating 138
evaluation 138
evaluation of an assertion 138
evaluation position 248
exception 260
exception cases 262
Exception handling 260
exception handling 260
exception marker 131
exception of type 262
exception semantics 261
exception trigger 260
exception type 262
exception-correct 261
exceptions 260
Execution 81
execution 81
execution context 249
execution contexts 249
expanded 168
expanded 168
expanded 85
expanded 90
expanded class 85
Expanded classes 85
expanded classes 85
expanded client 90
Expanded clients 90
expanded type 168
expanded type 168
Expanded types 168
expanded types 168
expected 280
expected comment 280
explicit 196
explicit conversion 196
explicitly 196
Export rule 206
Export rule 94
export status 96
Export Status principle 203
Exported 96
exported 96
export-valid 206
expression conformance 189
expression convertibility 197
expression reduction 255
Expression semantics (strict case) 222
Expression semantics (strict) 222
expression's type 221
expression-call type 209
expression-call types 209
expression-compatible 189
extended final name 147
Extended final names 147
External DLL rule 278
External DLL semantics 278
External File rule 275
External file semantics 276
External Signature rule 274
External signature semantics 275
failed target evaluation 207
failure 260
failures 260
Feature Declaration rule 119
Feature Definition rule 114
Feature Identifier principle 107
feature name 203
feature name of 203
feature name of a call 203
feature name of the call 203
Feature Name rule 162
feature of 100
feature of 167
feature of 168
feature of 203
feature of a call 203
feature of a class 100
feature of a type 167
feature of the call 203
feature of the class 100
Feature principle 108
Feature Version theorem 207
features of 100
Features of 167
features of 167
features of 168
features of a class 100
Features of a type 167
features of a type 167
features of a type 168
features of the class 100
field 244
Fields 244
fields 244
Final 147
final 147
Final name 147
final name 147
final name set 147
Final names 147
final names 147
flat-short form 98
flat-short forms 98
floating-point type 271
Floating-point types 271
floating-point types 271
for 143
for 148
formal 122
Formal argument 122
formal argument 122
Formal Argument rule 122
Formal arguments 122
formal arguments 122
formal generic parameter 174
Formal generic parameters 174
formal generic parameters 174
Formal Generic rule 174
Formals 122
formals 122
free 280
free comment 280
Free Comment rule 281
free operator 283
Free operators 283
free operators 283
Free_binary 283
Free_unary 283
fresh 207
fresh 72
fresh call to once routine 207
fresh identifier 72
fresh identifiers 72
fresh once routine call 207
freshness of a once routine call 207
frozen 85
frozen class 85
Frozen classes 85
function 117
Functions 117
functions 117
garbage collection 238
general 233
General Call semantics 209
General conformance 191
general conformance 191
general creation 233
general creation procedure 233
General Validity rule 68
Generally available 96
generally available 96
generating class 243
generating type 243
generating type of an object 243
generator 243
generators 243
generic 174
generic class 174
Generic client 91
generic client 91
Generic clients 91
generic clients 91
generic constraint 176
Generic Constraint rule 176
generic derivation 175
Generic Derivation rule 177
Generic derivations 175
generic derivations 175
generic substitution 179
generic type 175
Generic Type Adaptation rule 179
generically constrained feature name 180
generically constrained name 180
generically constrained name 180
generically constrained names 180
generically derived 175
generically derived type 175
Generically-derived Class_type semantics 178
generic-creation-ready 178
generic-creation-ready type 178
generic-creation-ready types 178
Genericity 174
genericity 174
Header Comment rule 281
heir 145
Heirs 145
Hoare triple 136
identifier 107
Identifier 282
identifier 282
identifier of 107
identifier of a feature name 107
Identifier rule 221
Identifier rule 283
identifier-only 106
identifier-only feature 106
Identifier-only features 106
Identifiers 282
identifiers 282
identifiers of 107
ignore 263
ignored 263
ignoring 263
Immediate 100
immediate 100
immediate feature 100
immediate features 100
implication correctness 137
implication-correct 137
implicative 255
imported form 205
imported form of a non-object call 205
imported forms 205
incremental contract view 98
indirect 91
indirect client 91
Indirect clients 91
indirect clients 91
Inequality Expression semantics 252
Information Hiding 87
Information hiding 87
information hiding 87
inherit 145
inherit as deferred 149
inherit as effective 149
Inherited 100
inherited 100
inherited 148
inherited 161
inherited as deferred 149
inherited as effective 149
inherited feature 100
inherited feature 161
inherited feature list 161
inherited features 100
Inherited features 161
inherited features 161
inherited name 148
Inherited names 148
inherits as deferred 149
inherits as effective 149
Inline agent 211
inline agent 211
Inline Agent requirements 213
Inline Agent rule 213
Inline agents 211
inline agents 211
inspect expression 225
instance 165
instance 166
instance of a class 166
instance of a type 165
Instance principle 166
instances 165
instances 166
Instances of a class 166
instances of a class 166
instances of a type 165
instruction 237
Integer rule 286
Integer semantics 286
Integer types semantics 271
internal cluster 77
interval 226
introduce 100
introduce 100
introduced 100
introduced 100
invariance 104
invariant 104
invariant 130
invariant 130
invariant of a class 130
Invariant variable 104
invariant variable 104
Involved 169
involved 169
involved classes 169
involved in 169
involved types 169
involves 169
involving 169
Iteration rule 229
its base class 166
its class invariant 130
its features 100
its features 168
its generating type 243
its identifier 107
its invariant 130
its postcondition 130
its precondition 130
its static feature 203
its type 221
its type 243
its unfolded form 103
its value 210
Join rule 157
Join Semantics rule 158
Join-Sharing Reconciliation rule 161
keyword 282
Keywords 282
keywords 282
kinds of production 63
L1CC 252
L1CD 251
L1DE 253
label 182
labeled 182
labeled tuple type 182
labeled tuple types 182
Labels 182
labels 182
language processing tool 76
language processing tool 76
Language processing tools 76
Language processing tools 76
language processing tools 76
language processing tools 76
latest applicable target 208
left-aligned form 270
letter 279
Letter case 279
letter case 279
Letter case conventions 71
letter case conventions 71
Letter Case rule 282
Letters 279
letters 279
level 139
levels 139
levels of assertion monitoring 139
Line Feed 280
line sequence 269
line wrapping part 269
Line_sequence 269
Line_wrapping_part 269
list of basic types and sized variants 271
list of formal arguments 122
list of formal arguments of a routine 122
list of inherited features 161
local 124
local assertion 133
local invariant 133
local postcondition 133
local postconditions 133
local precondition 133
local preconditions 133
local unfolded form 132
local variable 124
Local Variable rule 123
Local variables 124
local variables 124
Locals 124
locals 124
longest break prefix 270
loop invariant 130
Loop rule 228
Loop semantics 230
loop variant 130
Lower case 279
lower case 279
lower name 70
lower names 70
Lower-case 279
lower-case 279
M1EE 252
M1IE 252
M3MC 268
M3MS 270
M3VS 270
M4AD 272
M4BO 271
M4CO 271
M4IN 271
M4RE 271
M7CI 70
MAAL 139
MAAM 139
MAAS 139
MAAV 139
MACH 139
machine 76
machine 76
MACO 137
MAEA 138
manifest 284
manifest character 284
Manifest character semantics 285
manifest characters 284
Manifest constant 266
manifest constant 266
Manifest constant semantics 268
manifest constant type 267
Manifest constants 266
Manifest String rule 269
manifest string semantics 270
manifest value 268
manifest value of a constant 268
Manifest-Type Qualifier rule 267
MAOE 131
MAVA 138
MBAC 242
MBAS 241
MBRE 240
MBRP 240
MCCH 85
MCNO 84
MCOB 86
MDJO 158
MDPC 152
MDRD 152
MDUD 153
MEDI 247
MEEN 250
MEGE 243
MEOB 244
MEOP 244
MEVS 249
MGGD 178
MGSB 179
MHRP 148
MIBR 280
MIIN 286
MILC 282
MIMC 285
minimal assertion 133
MIRE 287
MIST 286
MJVA 184
MKEX 238

Signed by JV - 2026/10/8

MKIN 236
MKIP 231
MMJS 161
MOCI 225
MOCO 224
MODB 230
MOLO 230
MOMA 227
MOMB 228
monitor 139
monitored 139
monitoring 139
MPAE 214
MPCA 214
MPIA 213
MPRE 213
MSCN 79
MSSE 81
MTAT 171
MTDI 165
MTIN 165
MTIP 166
MTNG 168
MTTY 167
MUCD 208
MUCU 209
MUGC 209
multiple 144
Multiple Constraints rule 180
multiple inheritance 144
multiple versions 160
MUNC 205
MUNO 209
MUON 208
MURE 210
MUVA 210
MUVU 207
MVOL 256
MVOT 256
MWEN 222
MWES 222
MXEC 262
MXEP 263
MXEX 261
MXRO 261
MYCV 196
MZAE 274
MZC+ 277
MZDL 278
MZEF 276
MZES 275
MZIC 277
name clash 150
Name Clash rule 162
Name clashes 150
name clashes 150
New Line 280
new line 280
new line 280
New_line 280
non-basic class 244
non-basic type 244
non-basis 244
non-conforming parent 145
Non-conforming parents 145
non-conforming parents 145
Non-generic class semantics 168
Non-generic class type semantics 168
non-generic type on 175
Non-Object Call rule 205
Non-Object Call semantics 205
Non-Once Routine Execution semantics 209
Non-production Syntax rule 65
non-terminal 61
not enough memory 238
not enough memory available 238
not enough memory is available 238
Notes semantics 84
numeric 279
Object principle 244
Object semantics 244
Object Test rule 255
Object Test semantics 256
Object-test local 254
object-test local 254
Object-Test Local semantics 256
Object-Test Local Semantics rule 256
Object-test locals 254
object-test locals 254
Obsolete semantics 86
of identifier 107
of type 262
Old Expression rule 131
Old expression semantics 131
once 123
once function 123
once functions 123
once key 123
Once keys 123
once keys 123
once per object 123
once per process 123
once per thread 123
once procedure 123
once procedures 123
once routine 123
Once Routine Execution semantics 208
Once routines 123
once routines 123
open and closed operands 214
open operand 214
open operand position 214
open operand positions 214
open operands 214
operand 202
operand 213
operand 216
operand 217
operand of a call 202
operand of an agent 213
operand position 202
Operand positions 202
operand positions 202
operands 202
operands 213
operands 217
operands of a call 202
operands of an agent 213
Operator Expression rule 218
Operator expression semantics (semistrict case) 222
Operator expression semantics (semistrict) 222
operator feature 106
Operator features 106
operator precedence 217
operator symbol 283
operator symbols 283
origin 100
origin 100
origin 154
origin of a feature 100
origin of a feature 100
original 263
original query 263
origins 154
P2CL 58
P2CN 61
P2HL 59
P2SD 65
P2SP 60
P2TX 66
parent 145
parent class 145
parent class for 145
Parent classes 145
parent classes 145
parent classes for 145
Parent part 143
Parent part for 143
Parent part for a class 143
Parent part for a type 143
Parent parts 143
Parent rule 146
parent type 145
parent type for 145
Parent types 145
parent types 145
parent types for 145
parenthesis feature 106
Parenthesis features 106
parenthesis features 106
parenthesized 218
Parenthesized expression semantics 217
parenthesized form 218
parenthesized form of an expression 218
Parents 145
parents 145
part for 143
PDDE 153
PDEF 153
percent 285
percent variants 285
PFAI 108
PFFI 107
PGTA 179
PIHC 281
PJSA 186
PJST 188
PJTA 186
PJTT 188
PKCP 233
PKCT 234
platform 76
platform 76
Polymorphic 241
polymorphic 241
polymorphic expression 241
Polymorphic expressions 241
Polymorphism 241
polymorphism 241
position 185
position 202
position in a tuple 185
position of an operand 202
positions 202
postcondition 130
postcondition 135
postcondition extension 135
postcondition of a feature 130
Postconditions 130
postconditions 130
precedence 217
precondition 130
Precondition Export rule 133
precondition extension 135
precondition of a feature 130
precondition-free 136
precondition-free 197
Precondition-free routine 197
precondition-free routine 197
preconditions 130
Precursor (joined features) 155
Precursor 155
Precursor 161
precursor 161
Precursor Call rule 151
Precursor call semantics 152
precursor feature 151
precursor feature 161
Precursor features 151
precursor features 151
precursor of a feature 161
precursor of the feature 161
Precursor SIgnature 150
Precursor Signature rule 150
Precursors 161
precursors 161
predefined 283
predefined operator 283
predefined operators 283
prefix 270
prevail immediately 225
Prevailing immediately 225
prevails immediately 225
printable 279
printable characters 279
procedure 117
Procedures 117
production 63
Proper 146
proper 146
proper ancestor 146
Proper ancestors 146
proper ancestors 146
proper descendant 146
Proper descendants 146
proper descendants 146
properly set 248
properly set variable 248
PTBP 166
PTBR 166
Public 96
public 96
Public properties 96
public properties 96
public property 96
PUFV 207
Qualified 201
qualified 201
Qualified call 201
qualified call 201
Qualified calls 201
qualified calls 201
Queries 118
queries 118
query 118
read-only 245
read-only 256
read-only entities 245
read-only entity 245
read-only void test 256
Real 287
real 287
Real number 287
real number 287
Real semantics 287
reattach 239
reattachment 239
Reattachment principle 240
Reattachment semantics 240
recipient 262
redeclaration 149
redeclaration 149
Redeclaration rule 154
Redeclarations 149
redeclare 149
redeclared 149
redeclares 149
Redeclaring 149
redeclaring 149
redefine 149
Redefine Subclause rule 152
redefined 149
redefines 149
Redefining 149
redefining 149
Redefinition 149
redefinition 149
redefinition 149
Redefinition semantics 152
Redefinitions 149
redefinitions 149
reduces 255
reduces to 255
reference 168
reference 168
reference 243
reference 243
reference 244
reference or expanded 179
reference or expanded status 179
reference or expanded status of a formal generic 179
reference semantics 244
reference type 168
reference type 168
Reference types 168
reference types 168
References 243
References 243
references 243
references 243
relative unfolded form 151
relative unfolded form of a Precursor_call 151
Rename 147
rename 147
Rename Clause rule 147
Renamed 147
renamed 147
Renaming 147
renaming 147
Renaming principle 148
Renamings 147
renamings 147
repeated 159
repeated ancestor 159
repeated descendant 159
repeated inheritance 159
Repeated Inheritance Consistency constraint 160
Repeated Inheritance rule 159
repeated-inheritance 159
repeatedly 159
repetition 64
Repetition production 64
replicated 159
Replication 159
replication 159
Replication Semantics rule 159
rescue block 261
Rescue Clause rule 260
Reserved 282
reserved 282
reserved word 282
Reserved words 282
reserved words 282
result 208
result of a call 210
result of a non-fresh call 208
result of non-fresh call 208
result of the call 210
result of the non-fresh call 208
result signature 111
result signature of a feature 111
Result type 110
result type 110
Result type of a query 110
result types 110
Return 280
Root class 81
root class 81
Root classes 81
root classes 81
Root object 81
root object 81
Root procedure 81
root procedure 81
Root procedure name 79
root procedure name 79
Root Procedure rule 81
root procedures 81
Root type 80
root type 80
Root type name 79
root type name 79
Root Type rule 80
root types 80
routine 117
Routines 117
routines 117
Run time 76
Run time 76
run time 76
run time 76
Run-time 76
Run-time 76
Runtime 76
Runtime 76
run-time 76
run-time 76
runtime 76
runtime 76
S1EE 252
S3CH 266
S3CO 266
S3LS 269
S3LW 269
S3MA 266
S3MR 269
S3MS 268
S3SG 266
S7SC 73
SAAS 128
SACH 130
SAHT 136
same 107
same alias 107
same feature name 107
same feature names 107
same name 107
same names 107
same operator 107
SAOL 131
SASY 129
satisfied statically 197
SAUA 134
SAVE 131
SBAC 241
SBAS 239
SCCD 82
SCCH 84
SCNO 83
SCOM 86
scope 255
scope 256
scope 71
scope of a read-only void test 256
scope of an identifier 71

Signed by JV - 2026/10/8

scope of an Object Test 255
scope of an object test 255
scope of the Object Test 255
scope of the object test 255
SDAE 135
SDPR 150
SDRD 149
SDRS 152
SDUS 152
secondary part 225
secondary part of a conditional instruction 225
secondary part of a Conditional_instruction 225
Secret 96
Secret 96
secret 96
secret 96
Secret properties 96
secret properties 96
secret property 96
seed 154
seeds 154
SEEN 245
Select Subclause rule 160
Selectively available 96
selectively available 96
self-initializing 175
self-initializing 246
self-initializing 247
self-initializing formal generic parameter 175
self-initializing formal parameter 175
self-initializing type 246
self-initializing variable 247
semantics 285
semantics 68
Semantics of inline agents 213
semantics of manifest strings 270
Semicolon Optionality rule 73
semistrict 222
semistrict 255
semistrict operator 222
Semistrict operators 222
semistrict operators 222
separator 64
setting 249
SFAD 143
SFAL 105
SFAM 112
SFDB 109
SFDE 102
SFDF 113
SFNA 104
SFNL 102
SFOP 108
SFPA 101
SFQM 110
SGFG 174
SGGC 176
SGGP 173
shared 159
Sharing 159
sharing 159
SHIP 143
short form 98
SHRC 147
SIBR 281
SIBT 280
SICH 279
SICO 280
SIDR 282
SIFC 281
Sign Syntax rule 266
signature 111
signature conformance 190
signature of a feature 111
SIID 282
SIIN 286
SIMC 284
simple client 89
Simple clients 89
simple string 285
simple strings 285
Simple_string 285
single 144
single inheritance 144
single tuple 185
single-tuple 185
single-tuple class 187
single-tuple routine 185
SIPC 285
SIRE 287
SISP 285
SIST 285
sized basic type 271
sized basic types 271
sized character variants 271
sized floating-point variants 271
sized integer variants 271
sized real variants 271
sized string variant 271
sized string variants 271
sized variants 271
sized variants 271
sized variants 271
sized variants 271
SJMT 181
SJTU 181
SKEX 237
SKIN 234
SKPA 232
SLCL 92
SLEA 93
SMRI 159
SMSE 160
SOCE 219
SOCI 224
SODB 230
SOLO 228
SOMB 225
SOUL 229
source 239
source of a reattachment 239
source of the reattachment 239
SPAG 211
special character 285
special character code 285
special character codes 285
special symbol 282
Special symbols 282
special symbols 282
specified parent 150
Specimen 60
specimen 60
Specimens 60
specimens 60
SPET 214
SPFO 212
SQBO 117
SRBO 123
SRFE 121
SRIN 124
SRLV 123
SSCN 78
stand-alone 171
stand-alone type 171
standard binary operator 283
standard binary operators 283
standard operator 283
standard operators 283
standard unary operator 283
standard unary operators 283
static feature 203
static feature 234
static feature of 203
static feature of 234
static feature of a call 203
static feature of a creation call 234
static feature of the call 203
statically 197
statically satisfied 197
statically satisfied precondition 197
String 285
string 285
STSA 171
STTY 165
SUAA 202
subcluster 77
Subclusters 77
subclusters 77
subcontract 130
subcontract of a feature 130
subexpression 216
subobject 245
substitution 179
SUCA 200
SUFN 203
SUIF 205
supplier 88
supplier of 88
supplier of a class 88
supplier of a typd 88
SUQU 201
SUTA 201
SUTU 201
SVOT 254
SWBR 219
SWBR 219
SWEX 215
SWOP 217
SWPE 217
SWPR 217
SXRC 260
SYCV 194
symbol 281
Symbols 281
symbols 281
synonym 103
synonym of 103
synonym of a feature 103
Synonyms 103
synonyms 103
synonyms of 103
Synonyms of a feature 103
syntactical forms 212
syntactical forms of a call agent 212
syntax 59
syntax 59
System 79
system 79
System execution 81
System-Level Call rule 258
Systems 79
systems 79
system-valid 258
SZC+ 277
SZDL 278
SZEC 276
SZEF 275
SZER 273
SZES 274
SZRL 274
Tab 280
tab 280
target 201
target 206
target 234
target 239
target object 206
target objects 206
target of a call 201
target of a reattachment 239
target of the call 201
target of the calls 201
target of the reattachment 239
Target rule 204
target type 202
target type 212
target type of a call 202
target type of a call agent 212
target types 202
Target types 212
target types 212
target types of calls 202
Target validity 204
target validity 204
target-converted 218
target-converted form 218
targeted 201
targeted call 201
targeted calls 201
targets 201
targets 206
targets 234
Target-valid 204
target-valid 204
term 255
terminal 61
terminal cluster 77
Textual Conventions principle 66
the same 107
the same alias 107
the same feature name 107
the same feature name 107
the same feature names 107
the same name 107
the same names 107
the same operator 107
their base classes 166
their identifiers 107
THIS IS CONFORMANCE FOR EXPRESSIONS 189
THIS IS FOR ENTITIES 240
THIS IS SIGNATURE CONFORMANCE 190
THIS IS THE DEFINITION OF CONFORMANCE FOR TYPES 191
through a function 196
through a procedure 196
through a query 196
token 61
tokens 61
total correctness 136
totally correct 136
transpose 155
transposed 155
transposition 155
trigger 260
trigger of an expression 260
trigger of the exception 260
triggered 260
triggering 260
triggers 260
tuple constraint position 187
tuple position 185
Tuple type 182
tuple type 182
tuple type values 184
tuple types 182
tuple-argument-unfolded form 185
tuple-argument-unfolded form of a call 185
Tuple-Argument-Unfolding property 186
Tuple-Argument-Unfolding size 186
tuple-type-unfolded form 187
tuple-type-unfolded form of a type 187
Tuple-Type-Unfolding property 188
Tuple-Type-Unfolding Size property 188
type 214
type 221
type 243
type 262
type 267
type dependency 80
type of 221
type of a manifest constant 267
type of an agent expression 214
type of an exception 262
type of an expression 221
type of an object 243
type of the expression 221
Type Semantics rule 167
Type semantics rule 167
type sequence 182
type-compatible 189
types involved in a type 169
types it involves 169
unconstrained 175
unconstrained generic class 175
unconstrained parameter 175
unconstrained parameters 175
unconstrained formal generic parameter 175
undefine 152
Undefine Subclause rule 152
undefines 152
Undefining 152
undefining 152
Undefinition 152
undefinition 152
Undefinition semantics 153
Undefinitions 152
undefinitions 152
underlined text 59
unfolded 156
unfolded class invariant 134
unfolded Creators part 232
unfolded creators part 232
unfolded creators part 232
unfolded Creators part of a class 232
unfolded form 103
unfolded form 122
unfolded form 134
unfolded form 134
unfolded form 136
unfolded form 151
unfolded form 183
unfolded form 185
unfolded form 226
unfolded form 226
unfolded form 229
unfolded form 232
unfolded form 234
unfolded form 237
unfolded form of a class invariant 134
unfolded form of a creation expression 237
unfolded form of a creation instruction 234
unfolded form of a feature declaration 103
unfolded form of a Formal_argument_list 122
unfolded form of a loop 229
unfolded form of a Multi_branch 226
unfolded form of a multi-branch 226
unfolded form of a possibly multiple declaration 103
unfolded form of a Precursor 151
unfolded form of a tuple type 183
unfolded form of an assertion 134
unfolded form of an interval 226
unfolded form of its declaration 103
unfolded form of the class invariant 134
unfolded form of the loop 229
unfolded inheritance part 144
unfolded inheritance part of a class 144
unfolded part 232
unfolded part 232
unfolded postcondition 134
unfolded postconditions 134
unfolded precondition 134
unfolded preconditions 134
Unfolded redeclaration 156
Uniform Access principle 118
Universal Conformance 144
Universal Conformance principle 144
universe 78
Unqualified 201
unqualified 201
Unqualified call 201
unqualified call 201
Unqualified calls 201
Unqualified calls 201
unqualified calls 201
untargeted 201
untargeted call 201
untargeted calls 201
Upper case 279
upper case 279
upper name 70
upper names 70
Upper-case 279
upper-case 279
use of Result in an inline function agent 213
V3CA 266
V3MQ 267
V3VS 270
V5DA 258
V5SC 258
VAAI 133
valid 67

Signed by JV - 2026/10/8

validity constraint 67
Validity constraints 67
validity constraints 67
value 184
value 210
value 249
value of a call expression 210
value of a setting 249
value of a type 165
value of the call 210
value of the expression 217
value of the setting 249
value sequence 182
Value sequences 182
value sequences 182
Values 184
values 184
values of a tuple type 184
Values of a type 165
values of a type 165
values of a type 165
VAOX 131
VAPE 133
variable 115
variable 115
variable 245
variable attribute 115
variable attribute 115
Variable attributes 115
Variable attributes 115
Variable rule 246
Variable semantics 249
variable setting 249
Variables 115
Variables 115
Variables 245
variables 245
variance 104
variant 130
Variant Expression rule 131
VAVE 131
VBAC 242
VBAS 239
VBGV 68
VCCH 85
VDJR 157
VDPC 151
VDPS 150
VDRD 154
VDRS 152
VDUS 152
VEEN 245
Verbatim String rule 270
Version 160
version of 160
Version of a feature 160
version of a feature 160
Versions 160
Versions 160
versions 160
versions 160
VEVA 246
VFAC 113
VFAV 105
VFCA 118
VFDB 109
VFDE 119
VFDF 114
VGFG 174
VGGC 176
VGGD 177
VGMC 180
VHCA 144
VHPR 146
VHRC 147
VHUC 144
VIID 283
VIIN 286
violated 139
violation 139
VJTT 183
VJUT 183
VKCC 232
VKCP 231
VKEP 237
VKEX 237
VKIN 235
VKIP 235
VLEL 94
VMCS 159
VMFN 162
VMNC 162
VMRC 160
VMSS 160
VNCC 191
VNCE 192
VNCF 192
VNCN 191
VNCS 190
VNCT 192
void 243
void reference 243
void references 243
void target 207
Void-Unsafe 207
VOIN 227
VOIT 229
VOLO 228
VOMB 227
VPCA 212
VPIA 213
VPIR 213
VRED 123
VRFA 122
VRLV 123
VSCI 78
VSCN 79
VSRP 81
VSRT 80
VTAT 170
VTCT 167
VUAR 204
VUCC 206
VUCN 200
VUES 203
VUEX 206
VUNO 205
VUTA 204
VVOT 255
VWBE 221
VWBP 219
VWID 221
VWOP 218
VXRC 260
VYCP 194
VYCQ 195
VYEC 197
VYPF 197
VZAE 274
VZC+ 277
VZDL 278
VZEC 277
VZEF 275
VZES 274
whose class infariant 130
whose feature 100
whose feature 203
whose features 100
whose invariant 130
whose postcondition 130
whose precondition 130
whose static feature 203
word 281
Words 281
words 281

Signed by JV - 2026/10/8