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*; such text only plays an explanatory role for human readers.

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.

The specification of Partition 8 consists of precise specification elements, originating with the book *Standard Eiffel* where these elements are accompanied by extensive explanations and examples. The elements retained are:

- Definitions of technical terms and Eiffel concepts.
- Syntax specifications.
- Validity constraints (with their codes, such as *VVBG*).
- Semantic specifications.

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, 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.

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 the following classes to be present in ELKS: *ANY*, *ARRAY* [*G*], *BOOLEAN*, *CHARACTER*, *CHARACTER_8*, *CHARACTER_32*, *INTEGER*, *INTEGER_8*, *INTEGER_16*, *INTEGER_32*, *INTEGER_64*, *PART_COMPARABLE* [*G*], *REAL*, *REAL_32*, *REAL_64*, *STRING*, *STRING_8*, *STRING_16*, *TYPE* [*G*]. All are non-generic except those marked [*G*], indicating a single, non-constrained generic parameter. The clauses referring to these classes specify the needed features.

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).

4 Definitions

All the Eiffel-specific terms used in the Standard are defined in paragraphs labeled "**Definition**".

5 Notational conventions

5.1 Standard elements

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

- **Syntax**: rule affecting the structure of Eiffel texts, including lexical properties as well as syntax proper. The conventions for describing syntax appear below.
- **Validity**: restrictions on syntactically legal texts.
- **Semantics**: properties of the execution of valid texts.
- **Definition**: introduction of a term defining a certain concept, which may relate to syntax, validity of semantic properties; the rest of the Standard may use the term as an abbreviation for the concept.
- **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 clauses of the Standard are labeled with the corresponding category. A clause with no label shall be considered as "Definition".

5.2 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*.

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.

5.3 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.

5.4 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, one of the conditions in the Name Clash rule (8.16.16) reads:

1 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" is precisely defined (8.5.19) as meaning that their identifiers, excluding any Operator aliases, 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*	

5.5 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.

5.6 Language description

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

5.7 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 Identifier_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.

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.

7 General description

The following is an informal introduction to the Eiffel language. It is informative only.

7.1 Design principles

The aim of Eiffel is to help specify, design, implement 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: reusability, extendibility and reliability. Also important were other factors such as efficiency, openness and portability.

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 or integrity.

The third fundamental factor is **reliability,** 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.

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.
- 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.

("Object-Oriented" is a misnomer; "Class-Oriented Analysis, Design and Programming" would be a more accurate description of the method.)

To see what a class looks like, let us look at a simple example, *ACCOUNT*, which describes bank accounts. But 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.

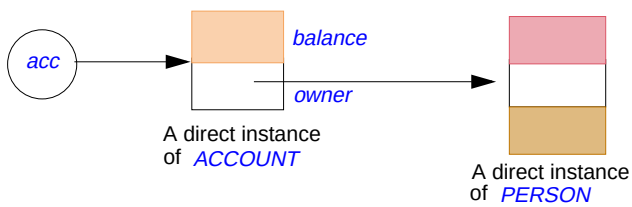

Figure 1: References and objects

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*, attaches *acc* to that instance, and initializes all its fields to default values. 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. For example it is possible to *redefine* a function into an attribute and conversely (in descendant classes), 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 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
        owner: PERSON
        minimum_balance: INTEGER = 1000
        open (who: PERSON)
                        -- Assign the account to owner who.
                do
                        owner := who
                end
        deposit (sum: INTEGER)
                        -- Deposit sum into the account.
                do
                        add (sum)
                end
```

```
        withdraw (sum: INTEGER)
                -- Withdraw 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 initial balance of an account object, just after creation, will be zero. Each type has a default initialization value: zero for *INTEGER* and *REAL*, false for *BOOLEAN*, null character for *CHARACTER*, and a void reference for reference types. 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.

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 will 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 any 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** *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:

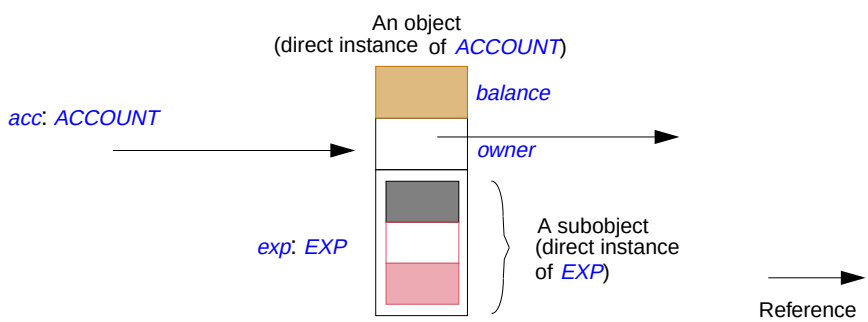

Figure 2: Object and subobject

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.

With appropriate assertions, the *ACCOUNT* class becomes:

```
class ACCOUNT create
        make
feature
        ... Attributes as before:
                balance, minimum_balance, owner, open ...

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

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

feature {NONE}
        add ... -- As before
        make (initial: INTEGER)
                        -- Initialize account with balance initial.
                require
                        initial >= minimum_balance
                do
                        balance := initial
                end
```

```
invariant
        balance >= minimum_balance
end
```

The notation **old** *attribute_name* may only be used in a routine postcondition. It denotes the value the attribute 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

```
create acc1.make (5_500)
```

is to allocate the object (as with default creation) and to call procedure *make* on it, with the argument given. This call is correct since it satisfies the precondition; it will ensure the invariant. (The underscore _ in the integer constant *5_500* has no semantic effect; you can improve the readability of numbers by separating digits into groups, usually of three.)

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 (8_000)
```

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 combining them with an “and”, 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**

clauses of routines, but keeping interface information and in particular assertions. Here is the interface of the above class (without the extra attribute):

```
class interface ACCOUNT create
        make
feature
        balance: INTEGER
                ...
        deposit (sum: INTEGER)
                        -- Deposit sum into the account.
                require
                        sum >= 0
                ensure
                        balance = old balance + sum
        withdraw (sum: INTEGER)
                        -- Withdraw sum from the account.
                require
                        sum >= 0
                        sum <= balance – minimum_balance
                ensure
                        balance = old balance – sum
        may_withdraw ...
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.

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 execute a **retry** instruction, which causes the routine to restart its execution from the beginning, attempting again to fulfil its contract, usually through another strategy. This assumes that the instructions of the **rescue** clause, before the **retry**, have attempted to correct the cause of the exception.
- If the **rescue** clause does not end with **retry**, 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
        end
```

Initialization rules ensure that *failures*, a local variable, is set to zero on entry.

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.

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.

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:

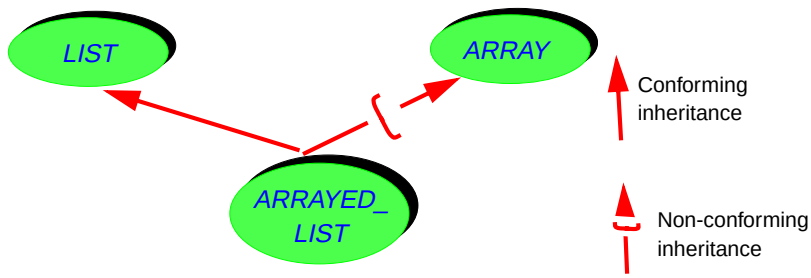

Figure 3: Inheritance

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.

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 ...
```

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 after the **rename** if present).

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 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 in simple cases, *V* conforms to *T* if and only if *V* 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* **infix** "+" 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 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 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 (year)
                deferred
                ensure
                        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*, **effect** 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.

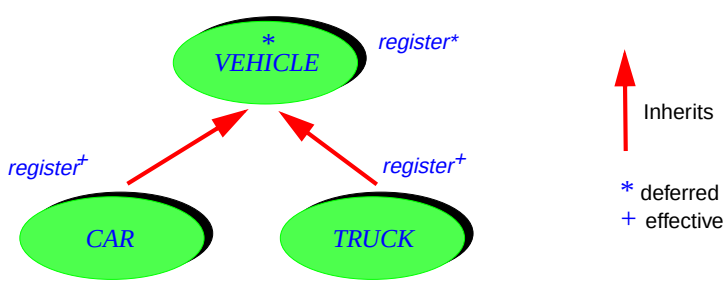

Figure 4: Abstracting variants into a deferred parent

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 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 **?** *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.

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 one of these classes as the system's **root class**.
- The designation of one of the creation procedures of the root class as the **root creation procedure**.

To execute such a system is to create one direct instance of the root class (the **root object** for the current execution), and to apply to it the root creation 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 name of the root class (which must be one of the classes of the universe) and of the root creation 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.

8 Language specification

8.1 General organization

Informative text

The remainder of the text provides the precise specification of Eiffel.

The overall order of the description is top-down, global structure to specific details:

- Conventions for the language description and basic conventions of the language itself.
- Architecture of Eiffel software, including the fundamental structuring mechanisms: cluster, class, feature, inheritance, client.
- Key elements of a class: routines and assertions.
- Type and type adaptation mechanisms, including redeclaration, genericity, tuples, conformance, convertibility and repeated inheritance.
- Control structures.
- Dynamic model: objects, attributes, entities, creation, copying.
- The calling mechanism and its consequences: expressions, type checking, barring void calls.
- Advanced mechanisms: exceptions, agents.
- Elementary mechanisms: constants, basic types, lexical elements.

End

8.2 Syntax, validity and semantics

8.2.1 Definition: Syntax, BNF-E

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

"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 Algol 60. BNF-E adds a few conventions — one production per construct, a simple notation for repetitions with separators — to make descriptions clearer. The range of formalisms that can be described by BNF-E is the same as for traditional BNF.

End

8.2.2 Definition: Component, construct, specimen

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

Informative text

For example, any particular class text, built according to the rules given in this language description, is a *component*. The *construct* Class describes the structure of class texts; any class text is a *specimen* of that construct. At the other end of the complexity spectrum, an identifier such as *your_variable* is a specimen of the construct Identifier.

Although we could use the term "instance" in lieu of "specimen", it could cause confusion with the instances of an Eiffel class — the run-time objects built according to the class specification.

End

8.2.3 Construct Specimen convention

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

Informative text

For example, ''a Class'' means ''a specimen of construct Class'': a text built according to the syntactical specification of the construct Class.

End

8.2.4 Construct Name convention

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

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

End

8.2.5 Definition: Terminal, non-terminal, token

Specimens of a **terminal** construct have no further syntactical structure. Examples include:

- Reserved words such as **if** and **Result**.
- Manifest constants such as the integer *234*; 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**.

In contrast, the specimens of a **non-terminal** construct are defined in terms of other constructs.

Informative text

Tokens (also called **lexical components**) form the basic vocabulary of Eiffel texts. By starting with tokens and applying the rules of syntax you may build more complex components — specimens of non-terminals.

End

8.2.6 Definition: Production

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

Construct $\triangleq$ right-side

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

Informative text

The symbol $\triangleq$ may be read aloud as “is defined as”.

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

End

8.2.7 Kinds of production

A production is of one of the following three kinds, distinguished by the form of the right-side:

- **Aggregate**, describing a construct whose specimens are made of a fixed sequence of parts, some of which may be optional.
- **Choice**, describing a construct having a set of given variants.
- **Repetition**, describing a construct whose specimens are made of a variable number of parts, all specimens of a given construct.

8.2.8 Definition: Aggregate production

An **aggregate** right side is of the form $C_1\ C_2 \ldots C_n$ ($n > 0$), where every one of the C_i is a construct and any contiguous subsequence may appear in square brackets as $[C_i \ldots C_j]$ for $1 \le i \le j \le n$.

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.

8.2.9 Definition: Choice production

A **choice** right side is of the form $C_1 \mid C_2 \mid \ldots \mid C_n$ ($n > 0$), where every one of the C_i is a construct.

Every specimen of the corresponding construct consists of exactly one specimen of one of the C_i.

8.2.10 Definition: Repetition production, separator

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

$\{C\ §\ \ldots\}^*$

$\{C\ §\ \ldots\}^+$

where C and § (the **separator**) are constructs.

Every 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 §.

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

C^*

C^+

Informative text

The language definition makes only moderate use of recursion thanks to the availability of Repetition productions: when the purpose is simply to describe a construct whose specimens may contain successive specimens of another construct, a Repetition generally gives a clearer picture. Recursion remains necessary to describe constructs with unbounded nesting possibilities, such as Conditional and Loop.

End

8.2.11 Basic syntax description rule

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

Informative text

Unlike in most BNF variants, every BNF-E production always uses exactly one of Aggregate, Choice and Repetition, *never* mixing them in the right sides. This convention yields a considerably clearer grammar, even if it has a few more productions (which in the end is good since they give a more accurate image of the language's complexity).

End

8.2.12 Definition: Non-production syntax rule

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

Informative text

Unlike validity rules, 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.

For example the 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 a few cases, for example in an Alias declaration such as **alias** "+", it is convenient to use BNF-E — with a right-side listing the keyword **alias**, a double quote, an Operator and again a double quote — but we need to *prohibit* breaks between either double quote and the operator. In other cases we *require* at least one break character. We still use BNF-E to specify such constructs, but add a non-production syntax rule stating the supplementary constraints.

End

8.2.13 Textual conventions

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

1. Symbols of BNF-E itself, 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 Class.
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 (blue like other Eiffel text) enclosed in two single quote characters (black since they belong to BNF-E, not Eiffel).
5. All other special symbols appear 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 **class** and **Result**, appear in **bold** (blue like other Eiffel text), except *TUPLE*. They do not require quotes since the conventions avoid ambiguity with construct names: Class is the name of a construct, **class** a keyword.

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

8 Other elements of Eiffel text, such as entities and feature names (including in comments) appear in non-bold *italic* (blue). This also applies to *TUPLE*.

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

Because of the difference between cases 1 and 3, **{** 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.

In case 2 the use of an upper-case first letter is a consequence of the "Construct Name convention".

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.

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

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*.

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

End

8.2.14 Definition: Validity constraint

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.

8.2.15 Definition: Valid

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 tools of any Eiffel environment, if and only if it satisfies the validity constraints, if any, applying to the construct.

8.2.16 Validity: General Validity rule **Validity 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.

8.2.17 Definition: Semantics

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.

8.2.18 Definition: Execution terminology

- **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, manipulate, explore and execute the text of an Eiffel system on a machine.

Informative text

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*".

End

8.2.19 Semantics: Case Insensitivity principle

In writing the letters of an Identifier serving as name for a class, feature or entity, or a reserved word, using the upper-case or lower-case versions has no effect on the semantics.

Informative text

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

End

8.2.20 Definition: Upper name, lower name

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

Informative text

In the example the lower name is *document* and the upper name *DOCUMENT*.

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

The reason for not letting letter case stand in the way of semantic interpretation 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; this can only cause confusion and errors. Different things should, in reliable and maintainable software, have clearly different names.

Letter case is of course significant in "manifest strings", denoting texts to be taken verbatim, such as error messages or file names.

This letter case policy goes with strong rules on **style**:

- Classes and types should always use the upper name, as with a class *DOCUMENT*.
- Non-constant features and entities should always use the lower name, as with an attribute *document*.
- Constants and "once" functions should use the lower name with the first letter changed to upper, as with a constant attribute *Document*.

End

8.2.21 Syntax (non-production): Semicolon Optionality rule

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 structure, validity and semantics of the software, to omit any of the semicolons, or to add a semicolon after the last element.

Informative text

This rule applies to instructions, declarations, successive groups of formal arguments, and many other Repetition constructs. It does not rely on the *layout* of the software: 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.

The rule also guarantees that an extra semicolon at the end, as in *a* ; *b* **;** instead of just *a* ; *b* is harmless.

The style guidelines suggest 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.

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 expected meaning.

End

8.3 The architecture of Eiffel software

Informative text

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**.

These three concepts provide the basis for structuring Eiffel software:

- A *class* is a modular unit.
- A *cluster* is a logical grouping of classes.
- A *system* results from the assembly of one or more classes to produce an executable unit.

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 tools of the supporting environment.

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

- At one extreme, a cluster may be a simple group of a few classes.
- 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*).
- In-between, a cluster may be a library consisting of several subclusters, or an existing system that you wish to integrate as a subcluster into a larger system.

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

After the basic definitions, the language description 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

8.3.1 Definition: Cluster, subcluster, contains directly, contains

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

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

End

8.3.2 Definition: Terminal cluster, internal cluster

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

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

End

8.3.3 Definition: Universe

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

Informative text

The universe provides a reference from which to draw classes of interest for a particular system. Any Eiffel environment will provide a way to specify a universe.

End

8.3.4 Validity: Class Name rule **Validity code:** *VSCN*

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

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:

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.

End

8.3.5 Semantics: Class name semantics

A Class_name *C* appearing in the text of a class *D* denotes the class called *C* in the enclosing universe.

8.3.6 Definition: System, root type name, root procedure name

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**.

8.3.7 Definition: Type dependency

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

This states that *C* depends on *A* if it is connected to *A* directly or indirectly through some combination of the basic relations between types and classes — inheritance and client — studied later. 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

8.3.8 Validity: Root Type rule

Validity code: *VSRT*

It is valid to designate a type *TN* as root type of a system of universe *U* if and only if it satisfies the following conditions:

1 *TN* is the name of a stand-alone type *T*.

2 *T* only involves classes in *U*.

3 *T*'s base class is not deferred.

4 The base class of any type on which *T* depends is in *U*.

Informative text

These conditions make it possible to create the root object:

- A type is "*stand-alone*" if it only involves class names; this 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.
- A deferred class is not fully implemented, and so cannot have any direct instances. It wouldn't work as base class here, since the very purpose of a root type is to be instantiated, as the first event of system execution.
- 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.

In condition 2, a type *TN* "involves" a class *C* if it is defined in terms of *C*, meaning that *C* is the base class of *TN* or of any of its generic parameters: *U* [*V*, *X* [*Y*, *Z*]] involves *U*, *V*, *X*, *Y* and *Z*. A non-generic class *T* used as a type "involves" only itself.

End

8.3.9 Validity: Root Procedure rule

Validity 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 creation procedure *p* of *S*'s root type.

2 *p* has no formal argument.

3 *p* is precondition-free.

Informative text

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 triggers 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.

Regarding condition 1, note that a non-deferred class that doesn't 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.

End

8.3.10 Definition: Root type, root procedure, root class

In a system *S* of root type name *TN* and root procedure name *pn*, the **root type** is the type of name *TN*, the **root class** is the base class of that root type, and the **root procedure** is the procedure of name *pn* in that class.

8.3.11 Semantics: System execution

To **execute** a system on a machine means to cause the machine to apply a creation instruction to the system's root type.

Informative text

If a routine is a creation procedure of a type used as root of a system, its execution will usually create other objects and call other features on these objects. In other words, the execution of any system is a chain of explosions — creations and calls — each one firing off the next, and the root procedure is the spark that detonates the first step.

End

8.4 Classes

Informative text

Classes are the components used to build Eiffel software.

Classes serve two complementary purposes: they are the modular units of software decomposition; they also provide the basis for the type system of Eiffel.

End

8.4.1 Definition: Current class

The **current class** of a construct specimen is the class in which it appears.

Informative text

Every Eiffel software element — feature, expression, instruction, … — indeed appears in a class, justifying this definition. Most language properties refer directly or indirectly, through this notion, to the class in which an element belongs.

End

8.4.2 Syntax: Class declarations

Class_declaration $\triangleq$ [Notes]
Class_header
[Formal_generics]
[Obsolete]
[Inheritance]
[Creators]
[Converters]
[Features]
[Invariant]
[Notes]
end

8.4.3 Syntax: Notes

Notes $\triangleq$ **note** Note_list

Note_list $\triangleq$ {Note_entry ";" …}*

Note_entry $\triangleq$ Note_name Note_values

Note_name $\triangleq$ Identifier ":"

Note_values $\triangleq$ {Note_item ","…}$^{+}$

Note_item $\triangleq$ Identifier | Manifest_constant

Informative text

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.

End

8.4.4 Semantics: Notes semantics

A Notes part has no effect on system execution.

8.4.5 Syntax: Class headers

Class_header $\triangleq$ [Header_mark] **class** Class_name

Header_mark $\triangleq$ **deferred** | **expanded** | **frozen**

Informative text

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 the upper name.

End

8.4.6 Validity: Class Header rule **Validity code:** *VCCH*

A Class_header appearing in the text of a class *C* is valid if and only if has either no deferred feature or a Header_mark of the **deferred** form.

Informative text

If a class has at least one deferred feature, either introduced as deferred in the class itself, or inherited as deferred and not “effected” (redeclared in non-deferred form), then its declaration must start not just with **class** but with **deferred class**.

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.

End

8.4.7 Definition: Deferred class, effective class

A class is **deferred** if its Class_header is of the **deferred** form. It is **effective** otherwise.

Informative text

Any class that has at least one deferred feature is deferred; any class that only has effective features is effective *except* if the class is explicitly declared as **deferred class**.

End

8.4.8 Syntax: Obsolete marks

Obsolete $\triangleq$ **obsolete** Message

Message $\triangleq$ Manifest_string

8.4.9 Semantics: Obsolete semantics

Specifying an 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 obsolescence Message and advising software authors to replace the class or feature by a newer version.

8.5 Features

Informative text

A class is characterized by its features. Every feature describes an operation for accessing or modifying instances of the class.

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.

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 or bracket syntax.

The following discussion introduces the various categories of feature, explains how to write feature declarations, and describes the form of feature names.

End

8.5.1 Definition: Inherited, immediate; origin; redeclaration; introduce

Any 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*. In this case any declaration of *f* in *C* (adapting the original properties of *f* for *C*) is a **redeclaration**.
2. If a declaration appearing in *C* applies to a feature that is not inherited, the feature is said to be **immediate** in *C*. Then *C* is the **origin** (short for "class of origin") of *f*, and is said to **introduce** *f*.

Informative text

A feature redeclaration is a declaration that locally changes an inherited feature. The details of redeclaration appear in the study of inheritance; what is important here is that a declaration in the Features part only introduces a new feature (called "immediate" in *C*, or "introduced" by *C*) if it is not a redeclaration of some feature obtained from a parent.

Every feature of a class is immediate either in the class or in one of its proper ancestors (parents, grandparents and so on).

End

8.5.2 Syntax: Feature parts

Features $\triangleq$ Feature_clause$^+$

Feature_clause $\triangleq$ **feature** [Clients] [Header_comment] Feature_declaration_list

Feature_declaration_list $\triangleq$ {Feature_declaration **";"** …}*

Header_comment $\triangleq$ Comment

Informative text

As part of a general syntactical convention, semicolons are **optional** between a Feature_declaration and the next. The recommended style rule suggests omitting them except in the infrequent case of two successive declarations on a single line.

End

8.5.3 Feature categories: overview

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*.

Informative text

A set of definitions in the discussion that follows introduces each of these notions precisely, making it possible to recognize, from a valid feature declaration, which kind of feature it introduces.

End

8.5.4 Syntax: Feature declarations

Feature_declaration ≜ New_feature_list Declaration_body

Declaration_body ≜ [Formal_arguments] [Query_mark] [Feature_value]

Query_mark ≜ Type_mark [Assigner_mark]

Type_mark ≜ ":" Type

Assigner_mark ≜ **assign** Feature_name

Feature_value ≜ [Explicit_value]
[Obsolete]
[Header_comment]
[Attribute_or_routine]

Explicit_value ≜ "=" Manifest_constant

Informative text

Not all combinations of Formal_arguments, Query_mark and Feature_value are possible; the Feature Body rule and Feature Declaration rule will give the exact constraints. For example it appears from the above syntax that both a Declaration_body and a Feature_value can be empty, since their right-side components are all optional, but the validity constraints rule this out.

End

8.5.5 Syntax: New feature lists

New_feature_list ≜ {New_feature "," …}$^{+}$

New_feature ≜ [**frozen**] Extended_feature_name

Informative text

Having a list of features, rather than just one, makes it possible for example to declare together several attributes of the same type or, in the case of routines, to introduce several “synonym” routines, with the same body.

End

8.5.6 Syntax: Feature bodies

Attribute_or_routine ≜ [Precondition]
[Local_declarations]
Feature_body
[Postcondition]
[Rescue]
end

Feature_body ≜ Deferred | Effective_routine | Attribute

8.5.7 Validity: Feature Body rule

Validity code: *VFFB*

A Feature_value is valid if and only if it satisfies one of the following conditions:

1. It has an Explicit_value and no Attribute_or_routine.
2. It has an Attribute_or_routine with a Feature_body of the Attribute kind.
3. It has no Explicit_value and has an Attribute_or_routine with a Feature_body of the Effective_routine kind, itself of the Internal kind (beginning with **do** or **once**).

4 It has no Explicit_value and has an Attribute_or_routine with neither a Local_declarations nor a Rescue part, and a Feature_body that is either Deferred or an Effective_routine of the External kind.

Informative text

The Explicit_value only makes sense for an attribute — either declared explicitly with Attribute or simply given a type and a value — so cases 3 and 4 exclude this possibility.

The Local_declarations and Rescue parts only make sense (case 4) 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.

In both cases 1 and 2 the feature will be an attribute. Case 1 is the implicit form where we don't take the trouble to write the keyword **attribute**, writing for example just *your_attribute*: *SOME_TYPE*. Case 2 is the long form, explicitly using the keyword **attribute** and making it possible, as with routines, to have a Precondition, a Postcondition, and even an implementation (including a Rescue clause if desired) which will be used, for "self-initializing" types, on first use of an uninitialized field.

The Feature Body rule is the basic validity condition on feature declarations. But for a full view of the constraints we must take into account a set of definitions appearing next, which say what it takes for a feature declaration — already satisfying the Feature Body rule — to belong to one of the relevant categories: *variable attribute*, *constant attribute*, *function*, *procedure*. Another fundamental constraint, the Feature Declaration rule (VFFD), will then require that the feature described by any declaration match one of these categories. So the definitions below are a little more than definitions: they collectively yield a validity requirement complementing the Feature Body rule.

End

8.5.8 Definition: Variable attribute

A Feature_declaration is a **variable attribute** declaration if and only if it satisfies the following conditions:

1 There is no Formal_arguments part.

2 There is a Query_mark part.

3 There is no Explicit_value part.

4 If there is a Feature_value part, it has an Attribute_or_routine with a Feature_body of the Attribute kind.

8.5.9 Definition: Constant attribute

A Feature_declaration is a **constant attribute** declaration if and only if it satisfies the following conditions:

1 It has no Formal_arguments part.

2 It has a Query_mark part.

3 There is a Feature_value part including an Explicit_value.

8.5.10 Definition: Routine, function, procedure

A Feature_declaration is a **routine** declaration if and only if it satisfies the following condition:

- There is a Feature_value including an Attribute_or_routine, whose Feature_body is 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

For a routine the Formal_arguments (like the Query_mark) may or may not be present.

By convention this definition 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

8.5.11 Definition: Command, query

A **command** is a procedure. A **query** is an attribute or function.

Informative text

These notions underlie two important principles of the Eiffel method:

- The Command-Query separation principle, which suggests that queries should not change objects.
- The Uniform Access principle, which enjoins, whenever possible, to make no distinction between attributes and argumentless functions.

End

8.5.12 Definition: Signature, argument signature of a feature

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**.

Informative text

The argument signature is an empty sequence for attributes and for routines without arguments.

End

8.5.13 Feature principle

Every feature has an associated identifier.

Any valid call (qualified or unqualified) to the feature can be expressed through this identifier.

Informative text

The syntactic variants, available through **alias** clauses, offer other ways to express calls, reconciling object-oriented structure with earlier notations:

- You may qualify the name with **alias** "§" where § is some operator. For example if a feature is named *plus*, clients must call it as *a*.*plus* (*b*); by naming it *plus* **alias** "+" you still allow this form of calls — per the Feature principle — but you also permit *a* + *b* in accordance with traditional syntax for arithmetic expressions. The details of alias operators, as well as the associated conversion mechanism, appear next.
- You may also use a "bracket alias", written simply **alias** "[]" (with an opening bracket immediately followed by a closing bracket). This allows access through bracket syntax *x* [*index*], For example if a class describing some table structure has a feature *item* **alias** "[]" (*index: H*): *G* where *H* is some index type, items can be accessed through *your_table*.*item* (*i*) but also through the more concise *your_table* [*i*]. Again this is just a syntactic facility: the second form is a synonym for the first, which remains available.

End

8.5.14 Syntax: Feature names

Extended_feature_name $\triangleq$ Feature_name [Alias]

Feature_name $\triangleq$ Identifier

Alias $\triangleq$ **alias** '"' Alias_name '"' [**convert**]

Alias_name $\triangleq$ Operator | Bracket

Bracket $\triangleq$ "**[]**"

Informative text

The optional **convert** mark, for an operator feature, 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.

End

8.5.15 Syntax (non-production): Alias Syntax rule

The Alias_name of 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 one or more spaces (but no other characters) between the two words.

Informative text

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** " + ".

End

8.5.16 Definition: Operator feature, bracket feature, identifier-only

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. It is **identifier-only** if neither of these cases applies.

Informative text

The most common case is identifier-only. The other two 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.

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: when a class text needs to refer to one of its own features, the Feature_name is sufficient since (from the Feature Identifier principle below) it uniquely identifies the feature. So the Extended_feature_name is used in only two cases: when you first introduce a feature, in a Feature_declaration as discussed above, and when you change its name for a descendant, in a Rename clause (for both inheritance and constrained genericity).

This 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 its Alias.

End

8.5.17 Definition: Identifier of a feature name

The Identifier that starts a Extended_feature_name is called the **identifier of** that Extended_feature_name and, by extension, of the associated feature.

8.5.18 Validity: Feature Identifier principle

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

Informative text

This principle reflects a critical property of object-oriented programming in general and Eiffel in particular: no "*overloading*" of feature names within a class. It is marked as "validity" but has no code of its own since it is just a consequence of other validity rules.

End

8.5.19 Definition: Same feature name, same operator, same alias

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 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 either the same Operator or both Bracket.
- If either has a **convert** mark, so does the other.

Informative text

So *my_name*, *MY_NAME* and *mY_nAMe* are considered to be the same feature name. The recommended style uses a name with an initial capital and the rest in lower case (as in *My_name*) for constant attributes, and the lower name, all in lower case (as in *my_name*) for all other features. 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.

This notion is 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

8.5.20 Syntax: Operators

Operator $\triangleq$ Unary | Binary

Unary $\triangleq$ **not** | "**+**" | "–" | Free_unary

Binary $\triangleq$ "+" | "–" | "*" | "/" | "//" | "\\" | "^" | "**.**" |
"<" | ">" | "<=" | ">=" |
and | **or** | **xor** | **and then** | **or else** | **implies** |
Free_binary

Informative text

Free operators 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 unary or infix operators. You may for example define operators such as ******, **|–|** (maybe as an infix alias for a *distance* function), or various forms of arrow such as **<–>**, **–|–>**, **=>**.

End

Informative text

In an assignment *x* := *v* the target *x* must be a variable. If *item* is an attribute of the type *T* of *a*, programmers used to other languages may be tempted to write an assignment such as *a***.***item* := *v* to assign directly to the corresponding object field, but this 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*).

The class author may, for convenience, permit *a*.*item* := *v* as a shorthand for this call *a*.*put* (*v*), by specifying *put* as an **assigner command** associated with *item*. An instruction such as *a*.*item* := *v* is not an assignment, but simply a different notation for a procedure call; it is known as an **assigner call**. This scheme, a notational simplification only, is also convenient for features that have a Bracket alias, allowing for example, with *a* an array, an assigner call *a* [*i*] := *v* as shorthand for a call *a*.*put* (*v*, *i*).

The mechanism is applicable not just to attributes but (in line with the Uniform Access principle) to all queries, including functions with arguments.

The following rules 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

8.5.21 Validity: Assigner Command rule **Validity 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 is the result type of *q*.

4 For every *i* in 1..*n*, the type of the *i*+1-st argument of *c* is the type of the *i*-th argument of *q*.

Informative text

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.

In cases 3 and 4, we require the types 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*) with, as a typical implementation:

item: *T* **assign** *put* **do** … **end**

put (*b*: *U*) **do** … *item* := *b* … **end**

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 should certainly permitted. They become invalid, however, 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's generally impossible for two different types to be each compatible with the other.

This explains clause 3: the first argument of the assigner procedure must have *exactly* the same type as the result of the query. Similar reasoning applied to other arguments (if any) leads to clause 4.

End

8.5.22 Definition: Synonym

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*.

8.5.23 Definition: Unfolded form of a possibly multiple declaration

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, 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

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. This convention is used throughout the language description; 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.

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.

This means in particular that a proper descendant of the class may rename or redeclare one without affecting the other.

Each f_i, being a New_feature, may include a **frozen** mark. In the unfolded form this mark only applies to the *i*-th declaration.

End

8.5.24 Validity: Feature Declaration rule

Validity code: *VFFD*

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 rules given earlier, is one of: variable attribute, constant attribute, procedure, function.
2. *f* does not have the same feature name as any other feature introduced in *C* (in particular, any other feature of the unfolded form).
3. If *f* has the same feature name as the final name of any inherited feature, the Declaration_body satisfies the Redeclaration rule.
4. If the Declaration_body describes a deferred feature, then the Extended_feature_name of *f* is not preceded by **frozen**.
5. If the Declaration_body describes a once function, the result type is stand-alone.
6. Any anchored type for an argument is detachable.
7. The Alias clause, if present, is alias-valid for *f*.

Informative text

As stated at the beginning of the rule, the conditions apply to the *unfolded form* of the declaration; this means that the rule treats a multiple declaration *f_1, f_2, ... , f_n declaration_body* as a succession of *n* separate declarations with different feature names but the same *declaration_body*.

Conditions 1 and 2 are straightforward: the Declaration_body must make sense, and the name or names of the feature being introduced must not conflict with those of any other feature introduced in the class.

In applying conditions 2 and 3, remember that two feature names are “the same” 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.

The Feature Name rule will state a consequence of conditions 2 and 3 that may be more appropriate for error messages in some cases of violation.

Condition 4 prohibits a frozen feature from being declared as deferred. The two properties are conceptually incompatible since frozen features, by definition, may not be redeclared, whereas the purpose of deferred features is precisely to force redeclaration in proper descendants.

Condition 5 applies to once functions. A once routine 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. This 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 instances obtained from different generic derivations; and *T* must not be anchored, as in the context of dynamic binding it could yield incompatible types depending on the type of the target of each particular call. The notion of stand-alone type captures these constraints on *T*.

Condition 6 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.

The last condition, 7, is the consistency requirement on features with an operator or bracket alias. It relies on the following definition (which has a validity code enabling compilers to give more precise error messages).

End

8.5.25 Validity: Alias Validity rule

Validity 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 query of *C* has an Operator alias using the same operator and the same number of arguments; 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 query with at least one argument, and no other feature of *C* has a Bracket alias.
3. If it includes a **convert** mark: it lists an Operator and *f* has one argument.
4. If it lists one of the “semistrict” operators **and then**, **or else** and **implies**: *C* is the Kernel Library class *BOOLEAN*.

Informative text

The first two conditions express the uniqueness and signature requirements on operator and bracket aliases:

- 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.
- A bracket feature, of which there may be at most one in a class, will be called under the form $x\ [a_1, \ldots a_n]$ with $n \geq 1$, and so must be a query with at least one argument (and hence a function). Condition 2 tells us that there may be at most one bracket feature per class.

Condition 3 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.

Condition 4 addresses the special case of the three semistrict boolean operators, which follow unique semantic rules enabling them not to evaluate their second operand in some cases: *a* **and then** *b* is guaranteed not to evaluate *b* if *a* evaluates to false, the result then being necessarily false. Although the language definition almost always provides generality — based on the principle that if a technique is useful to the language designer it must also be useful to the language user — it makes an exception here, because there is no simple way for programmers

to write a semistrict function definition. So **and then**, **or else** and **implies** are for the private use of class *BOOLEAN*. As condition 4 indicates, even its own proper descendants cannot redefine these features.

End

8.6 The inheritance relation

Informative text

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:

- As a **module extension** mechanism, inheritance makes it possible to define new classes from existing ones by adding or adapting features.
- 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

8.6.1 Syntax: Inheritance parts

Inheritance $\triangleq$ Inherit_clause$^+$

Inherit_clause $\triangleq$ **inherit** [Non_conformance] Parent_list

Non_conformance $\triangleq$ "**{**" [*NONE*] "**}**"

Parent_list $\triangleq$ {Parent ";" …}$^+$

Parent $\triangleq$ Class_type [Feature_adaptation]

Feature_adaptation $\triangleq$ [Undefine]
[Redefine]
[Rename]
[New_exports]
[Select]
end

Informative text

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

8.6.2 Definition: Parent part for a type, for a class

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*.

Informative text

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

8.6.3 Definition: Multiple, single inheritance

A class has **multiple inheritance** if it has an Unfolded Inheritance part with two or more Parent parts. It has **single inheritance** otherwise.

Informative text

What counts for this definition is the number not of parent classes but of Parent parts. If two clauses 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*.

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.

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

8.6.4 Definition: Inherit, heir, parent

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).

8.6.5 Definition: Conforming, non-conforming parent

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.

8.6.6 Definition: Ancestor types of a type, of a class

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

The basic notion is for ancestor types of a type. Case 1 indicates that a type is its own ancestor. 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

8.6.7 Definition: Ancestor, descendant

Class *A* is an **ancestor** of class *B* if and only if *A* is the base class of an ancestor type of *B*.

Class *B* is a **descendant** of class *A* if and only if *A* is an ancestor of *B*.

Informative text

Any class, then, is both one of its own descendants and one of its own ancestors. *Proper* descendants and ancestors exclude these cases.

End

8.6.8 Definition: Proper ancestor, proper descendant

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.

Informative text

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. 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

8.6.9 Validity: Class *ANY* rule **Validity code:** *VHCA*

Every system must include a non-generic class called *ANY*.

Informative text

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

8.6.10 Validity: Universal Conformance principle **Validity code:** *VHUC*

Every type conforms to *ANY*.

Informative text

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 notion of Unfolded Inheritance Part. The definition of "parent" below, and through it the definition of "ancestor", refer to the Unfolded Inheritance Part of a class rather than its actual Inheritance part.

End

8.6.11 Definition: Unfolded Inheritance Part of a class

Any class *C* has an **Unfolded Inheritance Part** defined as follows:

1 If *C* has an Inheritance part: that part.

2 Otherwise: an Inheritance part of the form **inherit** *ANY*.

8.6.12 Validity: Parent rule **Validity 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 If one or more Parent parts are present, at least one of them is conforming.

5 No two ancestor types of *D* are different generic derivations of the same class.

6 Every Parent is generic-creation-ready.

Informative text

Condition 1 ensures that there are no cycles in the inheritance relation.

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.

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*.

Condition 4 governs non-conforming inheritance; it ensures the Universal Conformance principle. If there are no Inheritance part we accept this — since the rule applies to the Unfolded Inheritance Part of the class — as shorthand for one of the form **inherit** *ANY*; but with an Inheritance part that would only have branches marked {*NONE*}, this rule would not apply, and so the current type would not conform to *ANY*. If at least one branch is conforming, then the corresponding parent type will (through recursive application of the same property) conform to *ANY*, and so will the current type.

Condition 5 avoids ambiguity when one of the ancestor classes is a generic class *A* [*G*] with, for example, a feature *f* (*x*: *G*); if we allowed a class *C* to inherit from both *A* [*T*] and *A* [*U*] for different types *T* and *U*, there could be no proper signature for *f* in *C*.

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*

End

8.6.13 Syntax: Rename clauses

Rename $\triangleq$ **rename** Rename_list

Rename_list $\triangleq$ {Rename_pair "," …}*

Rename_pair $\triangleq$ Feature_name **as** Extended_feature_name

Informative text

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:

- To identify the feature you are renaming, its Feature_name suffices.
- At the same time you are renaming the feature, you may give it a new operator or bracket alias, or remove the alias if it had one.

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

8.6.14 Validity: Rename Clause rule

Validity 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

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

8.6.15 Semantics: Renaming principle

Renaming does not affect the semantics of an inherited feature.

Informative text

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.

End

8.6.16 Definition: Final name, extended final name, final name set

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, *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:

- If the Parent part for *B* in *C* contains a Rename_pair of the form **rename** *parent_name* **as** *new_name*: *new_name*.
- Otherwise: *extended_parent_name*.

The final names of all the features of a class constitute the **final name set** of a class.

Informative text

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.

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

8.6.17 Definition: Inherited name

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

In the rest of the language description, references to the “name” of a feature, if not further qualified, always denote the final name.

End

8.6.18 Definition: Declaration for a feature

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

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

8.7 Clients and exports

Informative text

Along with inheritance, the client relation is one of the basic mechanisms for structuring software.

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.

The simplest and most common way is for *C* to contain the declaration of an entity of type *S*.

Variants of the relation introduce similar dependencies through other mechanisms, in particular generic parameters.

Although the original definitions introduce "client" in its various forms as a relation between a class and a type, we'll immediately extend it, by considering *S*'s base class, to a relation between classes.

End

Informative text

It is useful to distinguish between several variants of the client relation: simple client, expanded client and generic client relations. Each is studied below. The more general notion of client is the union of these cases, according to the following definition.

End

8.7.1 Definition: Client relation between classes and types

A class *C* is a **client** of a type *S* if some ancestor of *C* is a simple client, an expanded client or a generic client of *S*.

Informative text

Recall that the ancestors of *C* include *C* itself. The inclusion of *C*'s ancestors is necessary because the dependencies caused by inherited features are just as significant as those caused by the immediate features of *C*. Assume that an inherited routine *r* of *C* uses a local variable *x* of type *S*; this means that *C* depends on *S*, even if the text of *C* does not mention *S*.

End

8.7.2 Definition: Client relation between classes

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.

8.7.3 Definition: Indirect client

A class *A* is an **indirect client** of a type *S* of base class *B* if there is a sequence of classes $C_1 = A, C_2, \ldots, C_n = B$ 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.

8.7.4 Definition: Supplier

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.

8.7.5 Definition: Simple client

A class *C* is a **simple client** of a type *S* if, in *C*, *S* is the type of some entity or expression or the Explicit_creation_type of a Creation_instruction, or is one of the Constraining_types of a formal generic parameter of *C*, or is involved in the Type of a Non_object_call or of a Manifest_type.

Informative text

The constructs listed reflect the various ways in which a class may, by listing a type *S* in its text, enable itself to use features of *S* on targets of type *S*.

No constraint restricts how the classes of a system may be simple clients of one another. In particular, cycles are permitted: a class may be its own simple client, both directly according to this definition and indirectly.

End

8.7.6 Definition: Expanded client

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*.

8.7.7 Definition: Generic client, generic supplier

A class *C* is a **generic client** of a type *S* if for some generically derived type *T* of the form *B* [..., *S*, ...] one of the following holds:

1 *C* is a client of *T*.

2 *T* is a parent type of an ancestor of *C*.

Informative text

Case 1 captures for example the use in *C* of an entity of type *B* [*S*] (with *B* having just one generic parameter). Case 2 covers *C* inheriting directly or indirectly (remember that *C* is one of its own ancestors) from *B* [*S*].

End

8.7.8 Definition: Client set of a Clients part

The **client set** of a Clients part is the set of descendants of every class of the universe whose name it lists.

By convention, the client set of an absent Clients part includes all classes of the system.

Informative text

The descendants of a class include the class itself. 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.

No validity rule prevents listing in a Clients part a name *n* that does not denote a class of the universe. In this case — explicitly permitted by the phrasing of the definition — *n* does not denote any class and hence has no descendants; it does not contribute to the client set.

This important convention is in line with the reuse focus of Eiffel and its application to component-based development. You may develop a class *C* in a certain system, where it lists some class *S* in a Clients part, to give *S* access to some of its features; then you reuse *C* in another system that does not include *S*. You should not have to change *C* since no bad consequence can result from listing a class not present in the system, as long as *C* does not itself use *S* as its supplier or ancestor.

Even in a single system, this policy means that you can remove *S* — if you find it is no longer needed — without causing compilation errors in the classes that list it in their Clients parts. With a stricter rule, you would have to remove *S* from every such Clients part. But then if you later change your mind — as part of the normal hesitations of an incremental design process — you

would have to put it back in each of these places. This process is tedious, and it wouldn't take many iterations until programmers start making many features public just in case — hardly an improvement for information hiding, the purpose of all this.

End

8.7.9 Syntax: Clients

Clients $\triangleq$ "**{**" Class_list "**}**"

Class_list $\triangleq$ {Class_name "**;**" …}$^+$

Informative text

There is **no validity constraint** on Clients part. In particular, it is valid for a Clients part both:

- To list a class that does not belong to the universe.
- To list a class twice.

End

End

8.7.10 Syntax: Export adaptation

New_exports $\triangleq$ **export** New_export_list

New_export_list $\triangleq$ {New_export_item "**;**" …}$^+$

New_export_item $\triangleq$ Clients [Header_comment] Feature_set

Feature_set $\triangleq$ Feature_list | **all**

Feature_list $\triangleq$ {Feature_name "**,**" …}$^+$

8.7.11 Validity: Export List rule

Validity 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 elements 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

To obtain the export status of a feature, we need to look at the Feature_clause that introduces it if it is immediate, at the applicable New_exports clause, if any, if it is inherited, and at the Feature_clause containing its redeclaration if it is inherited and redeclared. In a New_exports, the keyword **all** means that the chosen status will apply to all the features inherited from the given parent.

The following definitions and rules express these properties. They start by extending the notion of "client set" from entire Clients parts to individual features.

End

8.7.12 Definition: Client set of a feature

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 introduced 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 the client sets (recursively) 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.

Informative text

This definition is the principal rule for determining the export status of a feature. It has two important properties:

- 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).
- As a result of case 2, **the client set can never diminish under conforming inheritance**: features can win new clients, but never lose one. This is necessary under polymorphism and dynamic binding to avoid certain type of "catcalls" leading to run-time crashes.

End

8.7.13 Definition: Available for call, available

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

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.

The key validity constraint on calls, export validity, will express that a call *a*.*f*(…) can only be valid if *f* is available to the type of *a*.

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".

There are three degrees of availability, as given by the following definition.

End

8.7.14 Definition: Exported, selectively available, secret

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

This is 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:

- A feature introduced by **feature** {*NONE*} (case 3) is available to no useful classes.
- A feature introduced by **feature** {*ANY*}, or just **feature**, is available to all classes and so will be considered to fall under case 1.
- A feature introduced by **feature** {*A, B, C*}, where none of {*A, B, C*} is *ANY*, falls under case 2.

A feature available to a class is also available to all the proper descendants of that class.

End

8.7.15 Definition: Secret, public

A property of a class text is **secret** if and only if it involves any of the following, describing information on which client classes cannot rely to establish their correctness:

1. Any feature that is not available to the given client, unless this is overridden by the next case.
2. Any feature that is not available for creation to the given client, unless this is overridden by the previous case.
3. The 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.
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

Software developers must be able to use a class as supplier on the basis of public information only.

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.

Whether a feature is external (case 3) or constant (case 4) determines whether it is possible to use it in a Non_object_call and hence is public information.

End

8.7.16 Definition: Incremental contract view, short form

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

Eiffel environments usually provide tools that automatically produce the incremental contract view of a class from the class text. This provides the principal form of software documentation: abstract yet precise, and extracted from the program text rather than written and maintained separately.

The definition specifies the information that the incremental contract view must retain, but not its exact display format, which typically will be close to Eiffel syntax.

End

8.7.17 Definition: Contract view, flat-short form

The **contract view** of a class, also called its **flat-short form**, is a text following the same conventions as the incremental contract view form but extended to include information about inherited as well as immediate features, the resulting combined preconditions and postconditions and the unfolded form of the class invariant including inherited clauses.

Informative text

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. 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. The "unfolded form" of the class invariant includes clauses from parents. In all these, of course, we still eliminate any clause that includes secret information, as with the incremental contract view.

The contract view 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 to remain accurate when the software changes.

End

8.8 Routines

Informative text

Routines describe computations.

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. Since every Eiffel operation applies to a specific object, a routine of a class describes a computation applicable to instances of that class. When applied to an instance, a routine may query or update some or all fields of the instance, corresponding to attributes of the class.

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**.

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.

The present discussion explores the structure of routine declarations, ending with the list of possible various forms of instructions.

End

8.8.1 Definition: Formal argument, actual argument

Entities declared in a routine to represent information passed by callers are the routine's **formal arguments**.

The corresponding expressions in a particular call to the routine are the call's **actual arguments**.

Informative text

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.

A note on 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

8.8.2 Syntax: Formal argument and entity declarations

Formal_arguments $\triangleq$ "(" Entity_declaration_list ")"

Entity_declaration_list $\triangleq$ {Entity_declaration_group ";" …}$^+$

Entity_declaration_group $\triangleq$ Identifier_list Type_mark

Identifier_list $\triangleq$ {Identifier "," …}$^+$

Informative text

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).

End

8.8.3 Validity: Formal Argument rule

Validity code: *VRFA*

Let *fa* be the Formal_arguments part of a routine *r* in a class *C*. Let *formals* be the concatenation of every Identifier_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*.

Informative text

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 we had a scoping rule removing any ambiguity by specifying that the local name overrides the feature name) and serves no useful purpose.

End

8.8.4 Validity: Entity Declaration rule

Validity code: *VRED*

Let *el* be an Entity_declaration_list. Let *identifiers* be the concatenation of every Identifier_list of every Entity_declaration_group in *el*. Then *el* is valid if and only if no Identifier appears more than once in the list *identifiers*.

8.8.5 Syntax: Routine bodies

Deferred $\triangleq$ **deferred**

Effective_routine $\triangleq$ Internal | External

Internal $\triangleq$ Routine_mark Compound

Routine_mark $\triangleq$ **do** | Once

Once $\triangleq$ **once** ["**(**"Key_list "**)**"]

Key_list $\triangleq$ {Manifest_string "**,**" …}$^+$

8.8.6 Definition: Once routine, once procedure, once function

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**.

8.8.7 Syntax: Local variable declarations

Local_declarations $\triangleq$ **local** [Entity_declaration_list]

8.8.8 Validity: Local Variable rule

Validity code: *VRLV*

Let *ld* be the Local_declarations part of a routine *r* in a class *C*. Let *locals* be the concatenation of every Identifier_list of every Entity_declaration_group in *ld*. Then *ld* is valid if and only if every Identifier *e* in *locals* satisfies the following conditions:

1 No feature of *C* has *e* as its final name.

2 No formal argument of *r* has *e* as its Identifier.

Informative text

Most of the rules governing the validity and semantics of declared local variables also apply to a special predefined entity: **Result**, which may only appear in a function or attribute, and denotes the value to be returned by the function. The following definition of "local variable" reflects this similarity.

End

8.8.9 Definition: Local variable

The local variables of a routine include all entities declared in its Local_declarations part, if any, and, if it is a query, the predefined entity **Result**.

Informative text

Result can appear not only in the Compound 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 allows stating abstract properties of that value, for example after a redefinition. In this case execution cannot change that value, but for simplicity we continue to call **Result** a local "variable" anyway.

End

8.8.10 Syntax: Instructions

Compound $\triangleq$ {Instruction ";" …}*

Instruction $\triangleq$ Creation_instruction | Call | Assignment | Assigner_call | Conditional | Multi_branch | Loop | Debug | Precursor | Check | Retry

Informative text

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 or the body of a Loop, the syntax never specifies Instruction but always Compound, so that you can include zero, one or more instructions.

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.

Call applies a routine to the object attached to a non-void expression. For the Call to yield an instruction, the routine must be a procedure.

Assignment changes the value attached to a variable.

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*.

Conditional, Multi_branch, Loop and Compound describe complex instructions, or control structures, made out of other 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.

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.

Precursor enables you, in redefining a routine, to rely on its original implementation.

Check is used to express that certain assertions must hold at certain moments during run time.

Retry is used in conjunction with the exception handling mechanism.

End

8.9 Correctness and contracts

Informative text

Eiffel software texts — classes and their routines — may be equipped with elements of formal specification, called **assertions**, expressing correctness conditions.

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.

Assertions are at the basis of the **Design by Contract** method of Eiffel software construction.

End

8.9.1 Syntax: Assertions

Precondition $\triangleq$ **require** [**else**] Assertion

Postcondition $\triangleq$ **ensure** [**then**] Assertion [Only]

Invariant $\triangleq$ **invariant** Assertion

Assertion $\triangleq$ {Assertion_clause ";" …}*

Assertion_clause $\triangleq$ [Tag_mark] Unlabeled_assertion_clause

Unlabeled_assertion_clause $\triangleq$ Boolean_expression | Comment

Tag_mark $\triangleq$ Tag ":"

Tag $\triangleq$ Identifier

8.9.2 Syntax (non-production): Assertion Syntax rule

An Assertion 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

This rule participates in the achievement of the general Semicolon Optionality rule. Without it, after an Assertion_clause starting 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.

The rule does 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

8.9.3 Definition: Precondition, postcondition, invariant

The **precondition** and **postcondition** of a feature, or the **invariant** of a class, is the Assertion of, respectively, the corresponding Precondition, Postcondition or Invariant clause if any, and otherwise the assertion **True**.

Informative text

So in these three contexts we consider any absent 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.

End

8.9.4 Definition: Contract, subcontract

Let *pre* and *post* be the precondition and postcondition of a feature *rout*. The **contract** of *rout* is the pair of assertions [*pre*, *post*].

A contract [*pre', post'*] is said to be a **subcontract** of [*pre, post]* if and only if *pre* implies *pre'* and *post'* implies *post*.

8.9.5 Validity: Precondition Export rule

Validity code: *VAPE*

A Precondition of a feature *r* of a class *S* is valid if and only if every feature *f* appearing in every Assertion_clause of its unfolded form *u* satisfies the following two conditions for every class *C* to which *r* is available:

1 If *f* appears as target of a call or Creation_expression or 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

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.

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.

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 the current class.

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".

End

8.9.6 Definition: Availability of an assertion clause

An Assertion_clause *a* of a routine Precondition or Postcondition is **available** to a class *B* if and only if all the features involved in the Equivalent Dot Form of *a* are available to *B*.

Informative text

This notion is necessary to define interface forms of a class adapted to individual clients, such as the incremental contract view ("short form").

End

8.9.7 Syntax: "Old" postcondition expressions

Old $\triangleq$ **old** Expression

8.9.8 Validity: Old Expression rule — Validity 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 *r*.

2 It does not involve **Result**.

3 Replacing *oe* by *e* in *post* yields a valid Postcondition.

Informative text

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

8.9.9 Semantics: Old Expression Semantics, associated variable, associated exception mark

The effect of including an Old expression *oe* in a Postcondition of an effective feature *f* is equivalent to adding to the Feature_body of *f* a fresh local variable *av*, called the **associated variable** of *oe* and add at the beginning of the Compound of the Feature_body a fictitious instruction that:

1 Evaluates *oe*.

2 If this evaluation triggers an exception, records this event in an **associated exception marker** for *oe*.

3 Otherwise, assigns the value of *oe* to *av*.

Informative text

The recourse to a fictitious variable, fictitious instructions and a fictitious marker 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.

End

The matter of exceptions is particularly delicate and justifies the use of "associated exception markers". If an Old expression'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*) + (1 / (**old** *x*)))) = *y*)

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 various Old expressions, such as 1 / **old** *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.

The "associated variable" is defined only for effective features, since a deferred feature has no Feature_body. If an Old expression 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.

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

8.9.10 Semantics: Associated Variable Semantics

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.

8.9.11 Syntax: "Only" postcondition clauses

Only $\triangleq$ **only** Feature_list

Informative text

The syntax of assertions indicates that an Only clause may only appear in a Postcondition of a feature, as its last clause.

Those other postcondition clauses let you specify how a feature *may* change specific properties of the target object, as expressed by queries. You may also want — this is called the **frame problem** — to restrict the scope of features by specifying which properties it *may not* change. You can always do this through postcondition clauses *q* = **old** *q*, one for each applicable query *q*. This is inconvenient, not only because there may be many such *q* to list but also, worse, because it forces you to list them all even though evolution of the software may bring in some new queries, which will not be listed. Inheritance makes matters even more delicate since such "frame" requirements of parents should be passed on to heirs.

An Only clause addresses the issue by enabling you to list which queries a feature may affect, with the implication that:

- Any query *not* listed is left unchanged by the routine.
- The constraints apply not only to the given version of the routine but also, as enforced by the following rules, to redeclarations in descendants.

End

8.9.12 Definition: Unfolded feature list of an Only clause

The **unfolded feature list** of an Only clause appearing in a Postcondition of a feature *f* in a class *C* is the Feature_list containing:

1 All the feature names appearing in its Feature_list.

2 If *f* is the redeclaration of one or more features, the final names in *C* of all the features whose names appear (recursively) in the unfolded forms of their Only clauses if any.

Informative text

For an immediate feature (a feature introduced in *C*, not a redeclaration), the purpose of an Only clause of the form

only *q, r, s*

is to state that *f* may only change the values of queries *q*, *r*, *s*.

In the case of a redeclaration, previous versions may have had their own Only clauses. Then:

- If there was already an Only clause in an ancestor *A*, the features listed, here *q*, *r* and *s*, must be new features, not present in *A*. Otherwise specifying **only** *q*, *r*, *s* would either contradict the Only clause of *A* if it did not include these features (thus ruling out any modification to them in any descendant), or be redundant with it if it listed any one of them.
- The meaning of the Only clause is that *f* may only change *q*, *r* and *s* *in addition* to inherited queries that earlier Only clauses allowed it to change.

End

8.9.13 Validity: Only Clause rule **Validity code:** *VAON*

An Only clause of unfolded feature list *fl*, appearing in a Postcondition of a feature of a class *C*, is valid if and only if it satisfies the following conditions:

1 No Feature_name appears more than once in *fl*.

2 Every Feature_name in *fl* is the final name of a query of *C*, with no arguments.

Informative text

Another condition, following from the syntax, is that an Only clause appears at the last element of a Postcondition; in particular, you cannot have more than one Only clause in a postcondition.

End

8.9.14 Definition: Unfolded form of an Only clause

The unfolded form of an Only clause *oc* appearing in a Postcondition of a feature of a class *C* is a sequence of Assertion_clause components of the following form, one for every argument-less query *q* of *C* that does not appear in the unfolded feature list of *oc*:

q = (**old** *q*)

Informative text

This will make it possible to express the semantics of an Only clause through a sequence of assertion clauses stating that the feature may change the value of no queries except those explicitly listed.

End

Note the use of the equal sign: for a query *q* returning a reference, the Only clause states (by *not* including *q*) that after the feature's execution the reference will be attached to the same object as before. That object might, internally, have changed. You can still rule out such changes by listing in the Only clause other queries reflecting properties of the object's *contents*.

End

8.9.15 Definition: Hoare triple notation (total correctness)

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*.

8.9.16 Semantics: Class consistency

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** $post_p\}$

2 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

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

8.9.17 Syntax: Check instructions

Check $\triangleq$ **check** Assertion [Notes] **end**

8.9.18 Definition: Check-correct

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.

8.9.19 Syntax: Variants

Variant $\triangleq$ **variant** [Tag_mark] Expression

8.9.20 Validity: Variant Expression rule **Validity code:** *VAVE*

A Variant is valid if and only if its variant expression is of type *INTEGER* or one of its sized variants.

8.9.21 Definition: Loop invariant and variant

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**.

8.9.22 Definition: 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* ≥ 0}

3 {*INV* **and then not** *EXIT*} *BODY* {*INV*}

4 {*INV* **and then not** *EXIT* **and then** (*VAR* = *v*)} *BODY* {0 ≤ *VAR* < *v*}

Informative text

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

8.9.23 Definition: Correctness (class)

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.

8.9.24 Definition: Local unfolded form of an assertion

The **local unfolded form** of an assertion *a* — a Boolean_expression — is the Equivalent Dot Form of the expression that would be obtained by applying the following transformations to *a* in order:

1 Replace any Only clause by its unfolded form.

2 Replace any Old expression by its associated variable.

3 Replace any clause of the Comment form by True.

Informative text

The unfolded form enables you to understand an assertion, possibly with many clauses, as a single boolean expression. The use of **and then** to separate the clauses indicates that you may, in a later clause, use an expression that is defined only if an earlier clause holds (has value true).

This unfolded form is "local" because it does not take into account any inherited assertion clauses. This is the business of the full (non-local) unfolded form, introduced in the discussion of inheritance.

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.

If an Only clause is present, we replace it by its own unfolded form, a sequence of Assertion_clause components of the form *q* = **old** *q*, so that we can treat it like other clauses for the assertion's local unfolded form.

The syntax permits a Comment as Unlabeled_assertion_clause. Such clauses are useful for clarity and documentation but, as reflected by condition 3, cannot have any effect on run-time monitoring.

End

8.9.25 Semantics: Assertion monitoring

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 both start and termination of a *qualified* call to a routine of *C*.
4. Invariant of 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 of a loop: as with the loop invariant.

8.9.26 Semantics: Evaluation of an assertion

To **evaluate** an assertion consists of computing the value of its unfolded form.

Informative text

This defines the value of an assertion in terms of the value of a boolean expression, as given by the discussion of expressions.

End

8.9.27 Semantics: Assertion violation

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

To simplify the discussion these cases are all called "*assertion* violations" even though a variant is not technically an assertion.

End

8.9.28 Semantics: Assertion semantics

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.

8.9.29 Semantics: Assertion monitoring levels

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.
- 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.

8.10 Feature adaptation

Informative text

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 discussion covers the principal mechanisms, leaving to a later one some important complements related to repeated inheritance.

End

8.10.1 Definition: Redeclare, redeclaration

A class **redeclares** an inherited feature if it redefines or effects 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

This definition relies on two others, appearing below, for the two cases: *redefinition* and *effecting*.

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

8.10.2 Definition: Unfolded form of an assertion

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*.

Informative text

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.

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

8.10.3 Definition: Assertion extensions

The Assertion of a Precondition starting with **require else** is a **precondition extension**.

The Assertion of a Postcondition starting with **ensure then** is a **postcondition extension**.

Informative text

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

8.10.4 Definition: Covariance-aware form of an assertion extension

The **covariance-aware form** of an assertion extension *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** $\{x_n: U_n\}\ y_n$) **implies** *a'*
   where y_1, … y_n are fresh names and *a'* is the result of substituting y_i for each corresponding x_i in *a*.
2. Otherwise: *a*.

Informative text

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

8.10.5 Definition: Combined precondition, postcondition

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 preconditions of these versions, and $post_1$, … $post_n$ their postconditions.

Let *pre'* be the covariance-aware form of the precondition extension of *f* if any, otherwise **False**, and *post'* the covariance-aware form of the postcondition extension of *f* if any, 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

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

8.10.6 Definition: Inherited as effective, inherited as deferred

An inherited feature is **inherited as effective** if it has at least one effective precursor and the corresponding Parent part does not undefine it.

Otherwise the feature is **inherited as deferred**.

8.10.7 Definition: Effect, effecting

A class **effects** an inherited feature *f* if an only if it inherits *f* as deferred and contains a declaration for *f*.

Such a declaration is then known as an **effecting** of *f*

Informative text

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

8.10.8 Definition: Redefine, redefinition

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

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*.)

Redefinition must keep the inherited status, deferred or effective, of *f*:

- It cannot turn a deferred feature into an effective one, as this would fall be an effecting.
- 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.

As defined earlier, the two cases, effecting and redefinition, are together called *redeclaration*.

End

8.10.9 Definition: Name clash

A class has a **name clash** if it inherits two or more features from different parents under the same final name.

Informative text

Since final names include the identifier part only, aliases if any play no role in this definition.

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

8.10.10 Syntax: Precursor

Precursor $\triangleq$ **Precursor** [Parent_qualification] [Actuals]

Parent_qualification $\triangleq$ "**{**" Class_name "**}**"

8.10.11 Definition: Relative unfolded form of a Precursor

In a class *C*, consider a Precursor specimen *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 *r'* if *p* has no Actuals, or *r'* (*args*) if *p* has actual arguments *args*, where *r'* is a fictitious feature name added, with a **frozen** mark, as synonym for *r* in *B*.

8.10.12 Validity: Precursor rule **Validity code:** *VDPR*

A Precursor is valid if and only if it satisfies the following conditions:

1 It appears in the Feature_body of a Feature_declaration of a routine *r*.

2 If the Parent_qualification part is present, its Class_name is the name of a parent class *P* of *C*.

3 Among the routines of *C*'s parents, limited to routines of *P* if condition 2 applies, exactly one is an effective routine redefined by *C* into *r*. (The class to which this routine belongs is called the **applicable parent** of the Precursor.)

4 The unfolded form relative to the applicable parent is, as an Unqualified_call, argument-valid.

In addition:

5 It is valid as an Instruction if and only if *r* is a procedure, and as an Expression if and only if *r* is a function.

Informative text

This constraint also serves, in condition 3, as a definition of the "applicable parent": the parent from which we reuse the implementation. Condition 4 relies on this notion.

Condition 1 states that the Precursor construct is only valid in a routine redefinition. In general the language definition treats functions and attributes equally (*Uniform Access* principle), but here an attribute would not be permissible, even with an Attribute body.

Because of our interpretation of a multiple declaration as a set of separate declarations, this means that if Precursor appears in the body of a multiple declaration it applies separately to every feature being redeclared. This is an unlikely case, and this rule makes it unlikely to be valid.

Condition 2 states that if you include a class name, as in *Precursor* {*B*}, then *B* must be the name of one of the parents of the current class. The following condition 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.

Condition 3 specifies when this explicit parent qualification is required. This is 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.

Condition 4 simply expresses that we understand the Precursor 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.

A Precursor 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 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. Condition 5 takes care of this.

End

8.10.13 Definition: Unfolded form of a Precursor

The **unfolded form** (absolute) of a valid Precursor is its unfolded form relative to its applicable parent.

8.10.14 Semantics: Precursor semantics

The effect of a Precursor is the effect of its unfolded form.

8.10.15 Syntax: Redefinition

Redefine ≙ **redefine** Feature_list

8.10.16 Validity: Redefine Subclause rule **Validity 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.

8.10.17 Semantics: Redefinition semantics

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

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

8.10.18 Syntax: Undefine clauses

Undefine ≙ **undefine** Feature_list

8.10.19 Validity: Undefine Subclause rule **Validity 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*.

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.

8.10.20 Semantics: Undefinition semantics

The effect in a class *C* of undefining a feature *f* in an Inheritance 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.

8.10.21 Definition: Effective, deferred feature

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*, specifying it as either as an attribute or as a routine whose Routine_body is of the Effective form (not the keyword **deferred** but beginning with **do**, **once** or **external**).

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

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".

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

8.10.22 Definition: Effecting

A redeclaration into an effective feature of a feature inherited as deferred is said to **effect** that feature.

8.10.23 Deferred class property

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**.

8.10.24 Effective class property

A class whose features, if any, are all effective, is effective unless its Class_header starts with the keyword **deferred**.

8.10.25 Definition: Origin, seed

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

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

8.10.26 Validity: Redeclaration rule **Validity 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

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**.

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.

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.

Signature conformance permits *covariant* redefinition of both query results and routine arguments, but for arguments you must make the new type detachable — **?***U* rather than just *U* — to prevent "catcalls".

Condition 3 requires adapting the assertions of a redeclared feature, as governed by rules given earlier.

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.

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.

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.

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.

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

8.10.27 Definition: Precursor (joined features)

A **precursor** of an inherited feature is a version of the feature in the parent from which it is inherited.

8.10.28 Validity: Join rule

Validity 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. After possible redeclaration in *C*, their signatures are identical.
2. They either have both no aliases or have the same alias.
3. If they both have assigner commands, the associated procedures have the same final name in *C*.
4. If both are inherited as effective, *C* redefines both into a common version.

Informative text

The Join rule indicates that joined features must have exactly the same signature — argument and result types.

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.

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.

In either case, nothing requires the precursors' signatures to conform to each other, as long as the signature of the redeclared version 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: **?** *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*).

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.

Conditions 2 and 3 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

8.10.29 Semantics: Join Semantics rule

Joining two or more inherited features with the same final name, under the terms of the Join rule, yields the feature resulting from their redeclaration if any, and otherwise defined as follows:

1. Its name is the final name of all its precursors.
2. Its signature is the precursors' signature, which the Join rule requires to be the same for all precursors after possible redeclaration.

3 Its precondition is the **or** of all the precursors' combined preconditions.

4 Its postcondition is the **and** of all the precursors' combined postconditions.

5 Its Header_comment is the concatenation of those of all precursors.

6 Its body is deferred if all the precursors are inherited as deferred, otherwise is the body of the single effective precursor.

7 It is not obsolete (even if some of the precursors are obsolete).

Informative text

Point 5 leaves the concatenation order unspecified.

In point 7 (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

8.11 Types

Informative text

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

8.11.1 Syntax: Types

Type $\triangleq$ Class_or_tuple_type | Formal_generic_name | Anchored

Class_or_tuple_type $\triangleq$ Class_type | Tuple_type

Class_type $\triangleq$ [Attachment_mark] Class_name [Actual_generics]

Attachment_mark $\triangleq$ "**?**" | "**!**"

Anchored $\triangleq$ [Attachment_mark] **like** Anchor

Anchor $\triangleq$ Feature_name | **Current**

Informative text

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.

An Attachment_mark **?** indicates that the type is **detachable**: its values may be void — not attached to an object. The **!** mark indicates the reverse: the type is **attached**, meaning that its values will always denote an object; language rules, in particular constraints on attachment, guarantee this. No Attachment_mark means the same as **!**, to ensure that a type, by default, will be attached.

End

8.11.2 Semantics: Direct instances and values of a type

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*.

8.11.3 Semantics: Instance of a type

The **instances** of a type *TX* are the direct instances of any type conforming to *TX*.

Informative text

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

8.11.4 Semantics: Instance principle

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*.

8.11.5 Definition: Instance, direct instance of a class

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

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

8.11.6 Base principle

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

A Class_type is its own base type; an anchored type **like** *anchor* with *anchor* having base type *U* also has *U* as its base type. For a formal generic parameter *G* in **class** *C* [*G* –> *T*] … the base type is (in simple cases) the constraining type *T*, or *ANY* if the constraint is implicit.

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.

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

8.11.7 Base rule

The **base type** of any type is a Class_or_tuple_type, with no Attachment_mark.

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

Why are these notions 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

8.11.8 Validity: Class Type rule **Validity 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 an Attachment_mark, that class is not expanded.

Informative text

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

8.11.9 Semantics: Type Semantics rule

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.

8.11.10 Definition: Base class and base type of an expression

Any expression *e* has a **base type** and a **base class**, defined as the base type and base class of the type of *e*.

8.11.11 Semantics: Non-generic class type semantics

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.

8.11.12 Definition: Expanded type, reference type

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

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.

Tuple types are, as a consequence of the definition, reference types.

End

8.11.13 Definition: Basic type

The basic types are *BOOLEAN*, *CHARACTER* and its sized variants, *INTEGER* and its sized variants, *REAL* and its sized variants and *POINTER*.

Informative text

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.

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

8.11.14 Definition: Anchor, anchored type, anchored entity

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

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.

End

Informative text

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

8.11.15 Definition: Anchor set; cyclic anchor

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

The anchor set of *LIST* [**like** *a, HASH_TABLE* [**like** *b, STRING*]] is, according to this definition, the set {*a, b*}.

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

8.11.16 Definition: Types and classes involved in a type

The types **involved** in a type *T* are the following:

- *T* itself.
- If *T* is of the form *a T'* where *a* is an Attachment_mark: (recursively) the types involved in *T'*.
- If *T* is a generically derived Class_type or a Tuple_type: all the types (recursively) involved in any of its actual parameters.

The *classes* involved in *T* are the base classes of the types involved in *T*.

Informative text

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

8.11.17 Definition: Constant type

A type *T* is **constant** if every type involved in *T* is a Class_or_tuple_type.

Informative text

The restriction to Class_or_tuple_type excludes formal generic parameters and anchored types. Constant types are the only ones permitted for constant attributes denoting manifest types.

End

8.11.18 Definition: Deanchored form of a type

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: *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*.

Informative text

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.

Case 4 enables us to treat **? like** *anchor* as a detachable type whether the type of *anchor* is attached or detachable.

End

8.11.19 Validity: Anchored Type rule **Validity 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

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.

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

8.11.20 Definition: Attached, detachable

A type is **detachable** if its deanchored form is a Class_type declared with the **?** Attachment_mark.

A type is **attached** if it is not detachable.

Informative text

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.

As a consequence of this definition, an expanded type is attached.

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

8.11.21 Semantics: Attached type semantics

Every run-time value of an attached type is non-void (attached to an object).

Informative text

In contrast, values of a detachable type may be void.

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.

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.

From the above semantics, the **!** 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 **?**. You can then use **!** 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

8.11.22 Definition: Stand-alone type

A Type is **stand-alone** if and only if it involves neither any Anchored type nor any Formal_generic_name.

Informative text

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.

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

8.12 Genericity

Informative text

The types discussed so far were directly defined by classes. The *genericity* mechanism, 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*.

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.

Two forms of genericity are available: with *unconstrained* genericity, *G* represents an arbitrary type; 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.

End

8.12.1 Syntax: Actual generic parameters

Actual_generics $\triangleq$ "**[**" Type_list "**]**"

Type_list $\triangleq$ {Type "**,**" …}$^+$

8.12.2 Syntax: Formal generic parameters

Formal_generics $\triangleq$ "[" Formal_generic_list "]"

Formal_generic_list $\triangleq$ {Formal_generic ","…}$^+$

Formal_generic $\triangleq$ [**frozen**] Formal_generic_name [Constraint]

Formal_generic_name $\triangleq$ [**?**] Identifier

8.12.3 Validity: Formal Generic rule — Validity code: *VCFG*

A Formal_generics part of a Class_declaration is valid if and only if every Formal_generic_name *G* in its Formal_generic_list satisfies the following conditions:

1 *G* is different from the name of any class in the universe.

2 *G* is different from any other Formal_generic_name appearing in the same Formal_generics_part.

Informative text

Adding the **frozen** qualification to a formal generic, as in *D* [**frozen** *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*.

Adding the **?** mark to a Formal_generic_name, as in **?** *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

8.12.4 Definition: Generic class; constrained, unconstrained

Any class declared with a Formal_generics part (constrained or not) is a **generic class**.

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

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

8.12.5 Definition: Generic derivation, non-generic type

The process of producing a type from a generic class by providing 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

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

8.12.6 Definition: Self-initializing formal

A Formal_generic_parameter is **self-initializing** if and only if its declaration includes the optional **?** mark.

Informative text

This is related to the notion of self-initializing *type*: 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 must itself, if attached, be a self-initializing type.

End

8.12.7 Definition: Constraint, constraining types of a Formal_generic

The **constraint** of a formal generic parameter is its Constraint part if present, and otherwise *ANY*.

Its **constraining types** are all the types listed in its Constraining_types if present, and otherwise just *ANY*.

8.12.8 Syntax: Generic constraints

Constraint $\triangleq$ "**–>**" Constraining_types [Constraint_creators]

Constraining_types $\triangleq$ Single_constraint | Multiple_constraint

Single_constraint $\triangleq$ Type [Renaming]

Renaming $\triangleq$ Rename **end**

Multiple_constraint $\triangleq$ "**{**" Constraint_list "**}**"

Constraint_list $\triangleq$ {Single_constraint "**,**" …}$^+$

Constraint_creators $\triangleq$ **create** Feature_list **end**

8.12.9 Validity: Generic Constraint rule **Validity code:** *VTGC*

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** *T* **rename** *rename_list* **end** (preceded by **deferred** if the base class of *T* is deferred) would be valid.

Informative text

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.

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 violated 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.

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

8.12.10 Definition: Constraining creation features

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 Constraining_creators if present.

Informative text

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

8.12.11 Validity: Generic Derivation rule

Validity code: *VTGD*

Let *C* be a generic class. A Class_type *CT* having *C* as base class is valid if and only if it 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 and *T* is attached, then *T* is a self-initializing type.

Informative text

In the case of unconstrained generic parameters, only condition 1 applies, since the constraint in that case is *ANY*, which trivially satisfies the other two conditions.

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.

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.

End

8.12.12 Definition: Generic-creation-ready type

A type is **generic-creation-ready** if and only if every actual generic parameter *T* of its deanchored form satisfies the following conditions:

1 If the specification of the corresponding formal generic parameter includes a Constraint_creators, the versions in *T* of the constraining creators for the corresponding formal parameter are creation procedures, and *T* is (recursively) generic-creation-ready.

2 If *T* is expanded, it is (recursively) generic-creation-ready.

Informative text

Although phrased so that it is applicable to any type, the condition is only interesting for generically derived types of the form *C* [..., *T*, ...]. Non-generically-derived types satisfy it trivially since there is no applicable *T*.

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.

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 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:

- For the creation type of a creation operation: conditions 4 of the Creation Instruction rule and 3 of the Creation Expression rule.
- For a Parent in an Inheritance part: condition 6 of the Parent rule.
- For an expanded type: condition 3 of the just seen Generic Derivation rule.

End

8.12.13 Semantics: Generically derived class type semantics

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

So *LINKED_LIST* [*POLYGON*] is its own base type, and its base class is *LINKED_LIST*.

End

8.12.14 Definition: Base type of a single-constrained formal generic

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_or_tuple_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

The definition is never cyclic since the only recursive part is the use of case 1 from case 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.

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

8.12.15 Definition: Base type of an unconstrained formal generic

The base type of an unconstrained Formal_generic_name type is *ANY*.

8.12.16 Definition: Reference or expanded status of a formal generic

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.

8.12.17 Definition: Current type

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

Clearly, the base class of the current type is always the current class.

End

8.12.18 Definition: Features of a type

The features of a type are the features of its base class.

Informative text

These are the features applicable to the type's instances (which are also instances of its base class).

End

8.12.19 Definition: Generic substitution

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.

2 Otherwise, σ is the identity substitution.

8.12.20 Generic Type Adaptation rule

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*.

Informative text

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

8.12.21 Definition: Generically constrained feature name

Consider a generic class *C*, a constrained Formal_generic_name *G* of *C*, a type *T* appearing as one of the Constraining_types for *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_constraint clauses for *T* include a Rename 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*.

8.12.22 Validity: Multiple Constraints rule **Validity code:** *VTMC*

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 it satisfies the following conditions:

1 At least one of the $CONST_i$ has a feature available to *C* whose generically constrained name for *G* in *C* is *fname*.

2 If this is the case for two or more of the $CONST_i$, all the corresponding features names, after possible renaming through Renaming clauses in the constraints, are the same.

8.12.23 Definition: Base type of a multi-constraint formal generic type

The base type of a multiply constrained Formal_generic_name type 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 and resolves any conflicts between potentially ambiguous features by renaming them to new names not available to developers.

8.13 Tuples

Informative text

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:

- Writing functions with multiple results, ensuring complete symmetry with multiple arguments.
- 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.
- Achieving the effect of routines with a variable number of arguments.
- Achieving the effect of generic classes with a variable number of generic parameters.
- 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.
- Making possible the agent mechanism through which you can handle routines as objects and define higher-order routines.

End

8.13.1 Syntax: Tuple types

Tuple_type ≜ *TUPLE* [Tuple_parameter_list]

Tuple_parameter_list ≜ [**frozen**] "**[**" Tuple_parameters "**]**"

Tuple_parameters ≜ Type_list | Entity_declaration_list

Informative text

A **frozen** mark indicates, as for generic classes, that the actual parameters, and hence the types of the tuple values, must match the exact types given, not just conform to it. Unlike with classes, the mark applies to all parameters at once.

End

8.13.2 Syntax: Manifest tuples

Manifest_tuple ≜ "**[**" Expression_list "**]**"

Expression_list ≜ {Expression "**,**" …}*

8.13.3 Definition: Type sequence of 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

The type sequence for *TUPLE* is empty; the type sequence for *TUPLE* [*INTEGER*; *REAL*; *POLYGON*] is *INTEGER*, *REAL*, *POLYGON*; the type sequence for *TUPLE* [*i*, *j*: *INTEGER*; *r*: *REAL*; *p*: *POLYGON*] is *INTEGER*, *INTEGER*, *REAL*, *POLYGON*, where *INTEGER* appears twice because of the two labels *i, j*.

End

8.13.4 Definition: Value sequences associated with a tuple type

The **value sequences** associated with a tuple type *T* are sequences of values, each of the type appearing at the corresponding position in *T*'s type sequence.

Informative text

Parameter labels play no role in the semantics of tuples and their conformance properties. They never intervene in tuple expressions (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.

End

8.14 Conformance

Informative text

Conformance is the most important characteristic of the Eiffel type system: it determines when a type may be used in lieu of another.

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.

Conformance, as the rest of the type system, relies on inheritance. The basic condition for *V* to conform to *T* is straightforward:

- The base class of *V* must be a descendant of the base class of *T*.
- 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*.
- If *T* is expanded, inheritance is not involved: *V* can only be *T* itself.

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.

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

8.14.1 Definition: Compatibility between types

A type is **compatible** with another if it either conforms or converts to it.

8.14.2 Definition: Compatibility between expressions

An expression *b* **is compatible with** an expression *a* if and only if *b* either conforms or converts to *a*.

8.14.3 Definition: Expression conformance

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, so is *SOURCE*.

3 If *SOURCE* is expanded, its version of the function *cloned* from *ANY* is available to the base class of *TARGET*.

Informative text

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.

A later definition will state what it means for an expression *b* to *convert* to another *a*. As a special case these properties also apply to entities.

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

8.14.4 Validity: Signature conformance

Validity 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

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.

Condition 3 adds a particular rule for "covariant redefinition" of arguments as defined next.

End

8.14.5 Definition: Covariant argument

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

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*: **?***U*). This forces the body of the redefined version, when applying to *x* any feature of *f*, to ensure that the value is indeed attached to an instance of *U* by applying an Object_test, for example in the form

if {*x*: *U*} *y* **then** *y*.*feature_of_U* **else** … **end**

This 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.

The rule only applies to *arguments*, not results, which do not pose a risk of catcall.

This rule is the reason why the Feature Declaration rule 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

8.14.6 Validity: General conformance

Validity 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 parameter does not specify **frozen**, conforms (recursively) to X_i.
5. *T* is a reference type and, 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

Cases 1 and 2 are immediate: a type conforms to itself, and direct conformance is a case of conformance.

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.

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.

Case 5 is indirect conformance through an intermediate type *U*. Note the restriction that *T* be a reference type; this excludes indirect conformance through an expanded type, as explained in later discussions.

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.

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

8.14.7 Definition: 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 current types.

8.14.8 Validity: Direct conformance: reference types

Validity 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*.

Informative text

The restriction to a reference type in this rule applies only to the target of the conformance, *BT*. The source, *CT*, may be expanded.

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*].

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.

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

8.14.9 Validity: Direct conformance: formal generic **Validity 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.

8.14.10 Validity: Direct conformance: expanded types **Validity code:** *VNCE*

No type **conforms directly** to an expanded type.

Informative text

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*.

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

8.14.11 Validity: Direct conformance: tuple types **Validity 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* is identical to *X* or, if *X* is not specified **frozen**, conforms to *X*.

No type conforms directly to a tuple type except as implied by these conditions.

Informative text

Labels, if present, play no part in the conformance.

End

8.15 Convertibility

Informative text

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

8.15.1 Definition: Conversion procedure, conversion type

A procedure whose name appears in a Converters clause is a **conversion procedure**.

A type listed in a Converters clause is a **conversion type**.

8.15.2 Definition: Conversion query, conversion feature

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**.

8.15.3 Validity: Conversion principle

No type may both conform and convert to another.

8.15.4 Validity: Conversion Asymmetry principle

No type *T* may convert to another through both a conversion procedure and a conversion query.

8.15.5 Validity: Conversion Non-Transitivity principle

That *V* converts to *U* and *U* to *T* does not imply that *V* converts to *T*.

8.15.6 Syntax: Converter clauses

Converters $\triangleq$ **convert** Converter_list

Converter_list $\triangleq$ {Converter ","…}$^+$

Converter $\triangleq$ Conversion_procedure | Conversion_query

Conversion_procedure $\triangleq$ Feature_name "**(**" "**{**" Type_list "**}**" "**)**"

Conversion_query $\triangleq$ Feature_name "**:**" "**{**" Type_list "**}**"

8.15.7 Validity: Conversion Procedure rule **Validity 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 constraint.
4 *SOURCE*'s base class is different from the base class of any other conversion type listed for a Conversion_procedure in 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*.

Informative text

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.

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

8.15.8 Validity: Conversion Query rule **Validity 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 constraint 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*.

Informative text

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

8.15.9 Definition: Converting to a class

A type *T* of base class *CT* **converts to** a class *C* if either:

- *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*.

8.15.10 Definition: Converting to and from a type

A type *U* of base class *D* **converts to** a Class_type *T* of base class *C* if and only if either:

1. *U* is the result of applying the generic substitution of *T* to a conversion type for a procedure *cp* appearing in the Converters clause of *C*.
2. *T* is the result of applying the generic substitution of *U* to a conversion type for a query *cq* appearing in the Converters clause of *D*.

A Class_type *T* **converts from** a type *U* if and only if *U* converts to *T*.

8.15.11 Definition: Converting "through"

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

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

8.15.12 Semantics: Conversion semantics

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 the same as the effect of either:

1 If *U* converts to *T* through a procedure *cp*: the creation instruction **create** *x*.*cp (e)*.

2 If *U* converts to *T* through a query *cq*: the assignment *x* := *e*.*cq*.

Informative text

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

8.15.13 Definition: Explicit conversion

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

Explicit conversion involves no new language mechanism, simply a feature of a Kernel Library class and the notion of bracket alias.

For example, assuming a tuple type that converts to *DATE*, you may use

{*DATE*} [[20, "*April*", 2005]]

The Basic_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

8.15.14 Validity: Expression convertibility **Validity 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.

8.15.15 Definition: Statically satisfied precondition

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

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.

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

8.15.16 Validity: Precondition-free routine

Validity 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

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.

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

Informative text

No specific validity rule limits our ability to include a **convert** mark in an Alias as long as it applies to a feature with one argument and an Operator alias. Of course in an example such as *your_integer* + *your_real* we expect the argument type *AT*, here *REAL*, to include a feature with the given operator, here +, and the target type *CT*, here *INTEGER*, to convert to *AT*. But we don’t require this of *all* types to which *CT* converts, because:

- This would have to be checked for every new type (since *CT* may convert to *AT* not only through its own “from” specification but also through a “to” specification in *AT*).
- In any case, it would be too restrictive: *INTEGER* may well convert to a certain type *AT* for which we don’t use target conversion.

Instead, the validity constraints will simply rule out individual *calls* that would require target conversion if the proper conditions are not met. For example if *REAL* did not have a function specifying **alias** "+" and accepting an integer argument, or if *INTEGER* did not convert to *REAL*, the expression would be invalid.

Remarkably, there is no need for any special validity rule to enforce these properties. All we'll need is the definition of **target-converted form of a binary expression** in the discussion of expressions. The target-converted form of *x* + *y* (or a similar expression for any other binary operator) is *x* + *y* itself unless *both* of the following properties hold:

- The declaration of "+" for the type of *x* specifies **convert**.
- The type of *y* does **not** conform or convert to the type of the argument of the associated function, here *plus*, so that the usual interpretation of the expression as shorthand for *x*.*plus* (*y*) cannot possibly be valid. This is critical since we don't want any ambiguity: either the usual interpretation or the targeted conversion should be valid, but not both.

Under these conditions the targeted-converted form is ({*TY*} [*x*]) + *y*, using as first operand the result of converting *x* to the type *TY* of *y*. Then:

- The target-converted form is only valid if *TY* has a feature with the "+" alias, and *y* is acceptable as an argument of this call. The beauty of this is that we don't need any new validity rule: if any of this condition is not met, the normal validity rules on expressions (involving, through the notion of Equivalent Dot Form, the rules on calls) will make it illegal.
- We don't need any specific semantic rule either: the normal semantic rules, applied to the target-converted form, yield exactly what we need.

End

8.16 Repeated inheritance

Informative text

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.

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

8.16.1 Definition: Repeated inheritance, ancestor, descendant

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*.

8.16.2 Semantics: Repeated Inheritance rule

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

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

8.16.3 Definition: Sharing, replication

A repeatedly inherited feature is **shared** if case 1 of the Repeated Inheritance rule applies, and **replicated** if case 2 applies.

8.16.4 Validity: Call Replication rule — Validity code: *VMCR*

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 include an unqualified call to a feature *g* of *A* or (if *f* is an attribute) to be the target of an assignment whose source involves *g* if and only if *g* is, along the corresponding inheritance paths, also shared.

Informative text

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

8.16.5 Semantics: Replication Semantics rule

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*.

Informative text

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

8.16.6 Syntax: Select clauses

Select $\triangleq$ **select** Feature_list

Informative text

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

8.16.7 Validity: Select Subclause rule — Validity 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

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

8.16.8 Definition: Version

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.

8.16.9 Definition: Multiple versions

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*.

8.16.10 Validity: Repeated Inheritance Consistency constraint **Validity 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.

8.16.11 Definition: Dynamic binding version

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 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.

8.16.12 Definition: Inherited features

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.)
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 of 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*.

8.16.13 Semantics: Join-Sharing Reconciliation rule

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.

8.16.14 Definition: Precursor

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*.

8.16.15 Validity: Feature Name rule **Validity 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

Condition 1 follows from other rules: the Feature Declaration rule, 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.

Two feature names are "the same" if the lower-case version of their identifiers is the same.

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.

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.

Condition 2 complements these requirements by ensuring that sharing doesn't inadvertently give a feature more than one alias.

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

8.16.16 Validity: Name Clash rule

Validity 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

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

8.17 Control structures

Informative text

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.

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.

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

8.17.1 Semantics: Compound (non-exception) semantics

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

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.

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: 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

8.17.2 Syntax: Conditionals

Conditional $\triangleq$ **if** Then_part_list [Else_part] **end**

Then_part_list $\triangleq$ {Then_part **elseif** …}$^{+}$

Then_part $\triangleq$ Boolean_expression **then** Compound

Else_part $\triangleq$ **else** Compound

8.17.3 Definition: Secondary part

The **secondary part** of a Conditional possessing at least one **elseif** is the Conditional obtained by removing the initial “**if** Then_part_list” and replacing the first **elseif** of the remainder by **if**.

8.17.4 Definition: Prevailing immediately

The execution of a Conditional starting with **if** $condition_1$ is said to **prevail immediately** if $condition_1$ has value true.

8.17.5 Semantics: Conditional semantics

The effect of a Conditional 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

Like the instruction studied next, the Conditional 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

8.17.6 Definition: Inspect expression

The **inspect expression** of a Multi_branch is the expression appearing after the keyword **inspect**.

8.17.7 Syntax: Multi-branch instructions

Multi_branch $\triangleq$ **inspect** Expression [When_part_list] [Else_part] **end**

When_part_list $\triangleq$ When_part$^+$

When_part $\triangleq$ **when** Choices **then** Compound

Choices $\triangleq$ {Choice "**,**" …}$^+$

Choice $\triangleq$ Constant | Manifest_type | Constant_interval | Type_interval

Constant_interval $\triangleq$ Constant "**..**" Constant

Type_interval $\triangleq$ Manifest_type "**..**" Manifest_type

8.17.8 Definition: Interval

An **interval** is a Constant_interval or Type_interval.

8.17.9 Definition: Unfolded form of a multi-branch

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

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

8.17.10 Definition: Unfolded form of an interval

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

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.

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.

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:

- 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*.
- 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*.
- 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

8.17.11 Validity: Interval rule **Validity code:** *VOIN*

An Interval is valid if and only if its unfolded form is not empty.

8.17.12 Definition: Inspect values of a multi-branch

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

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

8.17.13 Validity: Multi-branch rule **Validity 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*.

8.17.14 Semantics: Matching branch

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

The Multi-branch rule is designed to ensure that in any execution there will be at most one matching branch.

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.

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

8.17.15 Semantics: Multi-Branch semantics

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*.

8.17.16 Syntax: Loops

Loop $\triangleq$ Initialization
[Invariant]
Exit_condition
Loop_body
[Variant]
end

Initialization $\triangleq$ **from** Compound

Exit_condition $\triangleq$ **until** Boolean_expression

Loop_body $\triangleq$ **loop** Compound

8.17.17 Semantics: Loop semantics

The effect of a Loop 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.

8.17.18 Syntax: Debug instructions

Debug $\triangleq$ **debug** ["**(**"Key_list "**)**"] Compound **end**

8.17.19 Semantics: Debug semantics

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.

8.18 Attributes

Informative text

Attributes are one of the two kinds of feature.

When, in the declaration of a class, you introduce an attribute of a certain type, you specify that, for every instance of the class that may exist at execution time, there will be an associated value of that type.

Attributes are of two kinds: **variable** and **constant**. The difference affects what may happen at run time to the attribute's values in instances of the class: for a variable attribute, the class may include routines that, applied to a particular instance, will change the value; for a constant attribute, the value is the same for every instance, and cannot be changed at run time.

End

8.18.1 Syntax: Attribute bodies

Attribute $\stackrel{\Delta}{=}$ **attribute** Compound

Informative text

The Compound is empty in most usual cases, but it 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.

End

8.18.2 Validity: Manifest Constant rule

Validity code: *VQMC*

A declaration of a feature *f* introducing a manifest_constant is 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.
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 constant type *X* to which *Y* conforms.

Informative text

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*.

In case 6, we require 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. This would not be suitable for a constant attribute, which must have a single, well-defined value.

End

8.19 Objects, values and entities

Informative text

The execution of an Eiffel system consists of creating, accessing and modifying **objects**.

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.

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.

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

8.19.1 Semantics: Type, generating type of an object; generator

Every run-time object is a direct instance of exactly one Class_or_tuple_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.

8.19.2 Definition: Reference, void, attached, attached to

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.

8.19.3 Semantics: Object principle

Every non-void value is either an object or a reference attached to an object.

8.19.4 Definition: Object semantics

Every run-time object has either **copy semantics** or **reference semantics**.

An object has copy semantics if and only if it is the result of executing a creation operation whose creation target is of an expanded type, or of cloning such an object.

Informative text

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

8.19.5 Definition: Non-basic class, non-basic type, field

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**.

A direct instance of a non-basic type is a sequence of zero or more values, called **fields**. There is one field for every attribute of the type's base class.

8.19.6 Definition: Subobject, composite object

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**.

8.19.7 Definition: Entity, variable, read-only

An **entity** is an Identifier, or one of two reserved words (**Current** and **Result**), used in one of the following roles:

1 Final name of an attribute of a class.

2 Local variable of a routine or Inline_agent, including **Result** for a query.

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

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.

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.

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

8.19.8 Syntax: Entities and variables

Entity $\triangleq$ Variable | Read_only

Variable $\triangleq$ Variable_attribute | Local

Variable_attribute $\triangleq$ Feature_name

Local $\triangleq$ Identifier | **Result**

Read_only $\triangleq$ Formal | Constant_attribute | **Current**

Formal $\triangleq$ Identifier

Constant_attribute $\triangleq$ Feature_name

8.19.9 Validity: Entity rule

Validity code: *VEEN*

An occurrence of an entity *e* in the text of a class *C* (other than as the 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*.

3 *e* is the local variable **Result**, and the occurrence is in a Feature_body, Postcondition or Rescue part of an Attribute_or_routine 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 listed in the Identifier_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*.

6 *e* is listed in the Identifier_list of an Entity_declaration_group in a Formal_arguments part for a routine *r*, and the occurrence is in a declaration for *r*.

7 *e* is listed in the Identifier_list of an Entity_declaration_group in the Agent_arguments part of an Agent *a*, and the occurrence is in the Agent_body of *a*.

8 *e* is the Object-Test Local of an Object_test, and the occurrence is in its scope.

Informative text

"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*. The constraint on *b* is the General Call rule, requiring *b* to be the name of a feature in *D*'s base class.

End

8.19.10 Validity: Variable rule **Validity code:** *VEVA*

A Variable entity *v* is valid 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.

8.19.11 Definition: Self-initializing type

A type is **self-initializing** if it is one of:

1 A detachable type.
2 A self-initializing formal.
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*.

Informative text

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.

A "self-initializing formal" (case 2) 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*.

In the definition, the "creation procedures" of a *type* are 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.

The more directly useful notion is that of a self-initializing *variable*, appearing below.

The term "self-initializing" is justified by the following semantic rule, specifying the actual initialization values for every self-initializing type.

End

8.19.12 Semantics: Default Initialization rule

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: 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

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

8.19.13 Definition: Self-initializing variable

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 Attribute part such that the entity **Result** is properly set at the end of its Compound.

Informative text

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 (partly recursively from the above definition) .

End

8.19.14 Definition: Evaluation position, precedes

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, a special **end position**.

A position *p* **precedes** a position *q* if they are both in the same Compound or Assertion, and either:

- *p* and *q* are both Instruction or Assertion_clause components, and *p* appears before *q* in the corresponding list.
- *q* is the end position and *p* is not.

Informative text

This notion is needed to ensure that entities are properly set before use.

In a compound *i1; i2; i3* we have four positions; *i1* precedes *i2*, *i3* and the end position, and so on.

The relation as defined only applies to **first-level** components of the compound: if *i2* itself contains a compound, for example if it is of the form **if** *c* **then** *i4; i5* **end**, then *i4* is not an evaluation position of the outermost compound, and so has no "precedes" relation with any of *i1*, *i2* and *i3*.

End

8.19.15 Definition: Setter instruction

A **setter instruction** is an assignment or creation instruction.

If *x* is a variable, a setter instruction is a **setter for** *x* if its assignment target or creation target is *x*.

8.19.16 Definition: Properly set variable

At an evaluation position *ep* in a class *C*, a variable *x* is **properly set** if one of the following conditions holds:

1 *x* is self-initializing.

2 *ep* is an evaluation position of the Compound of a routine or Inline_agent of the Internal form, one of whose instructions precedes *ep* and is a setter for *x*.

3 *x* is a variable attribute, and is (recursively) properly set at the end position of every creation procedure of *C*.

4 *ep* is an evaluation position in a Compound that is part of an instruction *ep'*, itself belonging to a Compound, and *x* is (recursively) properly set at position *ep'*.

5 *ep* is in a Postcondition of a routine or Inline_agent of the Internal form, and *x* is (recursively) properly set at the end position of its Compound.

6 *ep* is **Result** in the Postcondition of a constant attribute

Informative text

The key cases are 2, particularly useful for local variables but also applicable to attributes, and 3, applicable to attributes when we cannot deduce proper initialization from the enclosing routine but find that every creation procedure will take care of it. Case 4 accounts for nested compounds. For assertions other than postconditions, which cannot use variables other than attributes, 3 is the only applicable condition. The somewhat special case 6 is a consequence of our classification of **Result** among local variables even in the Postcondition of a constant attribute.

As an artefact of the definition's phrasing, every variable attribute is "properly set" in any effective routine of a deferred class, since such a class has no creation procedures. This causes no problem since a failure to set the attribute properly will be caught, in the validity rule below, for versions of the routine in effective descendants.

End

8.19.17 Validity: Variable Initialization rule **Validity code:** *VEVI*

It is valid for an Expression, other than the target of an Assigner_call, to be also a Variable if it is properly set at the evaluation position defined by the closest enclosing Instruction or Assertion_clause.

Informative text

This is the fundamental requirement guaranteeing that the value will be defined if needed.

Because of the definition of "properly set", this requirement is pessimistic: some examples might be rejected even though a "smart" compiler might be able to prove, by more advanced control and data flow analysis, that the value will always be defined. But then the same software might be rejected by another compiler, less "smart" or simply using different criteria. On purpose, the definition limits itself to basic schemes that all compilers can implement.

If one of your software elements is rejected because of this rule, it's a sign that your algorithms fail to initialize a certain variable before use, or at least that the proper initialization is not clear enough. To correct the problem, you may:

- Add a version of *default_create* to the class, as creation procedure.
- Give the attribute a specific initialization through an explicit Attribute part that sets **Result** to the appropriate value.

End

8.19.18 Definition: Variable setting and its value

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 setter for *x*. (*Value*: the object attached to *x* by the setter, or a void reference if none.)

2 If *x* is a variable attribute with an 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

As a consequence of case 2, an attribute *a* that is self-initializing through an 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.

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

8.19.19 Definition: Execution context

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.

8.19.20 Semantics: Variable Semantics

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.

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 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

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 Attribute part won't be used.

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

8.19.21 Semantics: Entity Semantics rule

Evaluating an entity yields a **value** as follows:

1 For **Current**: a value 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

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.

The Object-Test Local Semantics rule appears in the discussion of the Object_test construct.

End

8.20 Creating objects

Informative text

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.

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.

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.

In using all these facilities, you should never forget the methodological rule governing creation, as expressed by the following principle.

End

8.20.1 Semantics: Creation principle

Any execution of a creation operation must produce an object that satisfies the invariant of its generating class.

Informative text

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

8.20.2 Definition: Creation operation

A **creation operation** is a creation instruction or expression.

8.20.3 Validity: Creation Precondition rule

Validity code: *VGCP*

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 the *uf* is available to every class to which *r* is available for creation.

Informative text

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.

Requiring preconditions to be creation-valid will ensure that a creation procedure doesn't try to access, in the object being created, fields whose properties are not guaranteed before initialization.

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

8.20.4 Syntax: Creators parts

Creators $\triangleq$ Creation_clause$^+$

Creation_clause $\triangleq$ **create** [Clients] [Header_comment] Creation_procedure_list

Creation_procedure_list $\triangleq$ {Creation_procedure ";"…}$^+$

Creation_procedure $\triangleq$ Feature_name

8.20.5 Definition: Unfolded Creators part of a class

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

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.

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

8.20.6 Validity: Creation Clause rule

Validity code: *VGCC*

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

As a result of conditions 1 and 4, a creation procedure may only be of the **do** form (the most common case) or External.

The prohibition of **once** creation procedures in condition 4 is a 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.

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

8.20.7 Definition: Creation procedures of a class

The **creation procedures** of a class are all the features appearing in any Creation_clause of its unfolded creators part.

Informative text

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*.

The following property is a consequence of the definitions of "unfolded creators part" and "creation procedures of a class".

End

8.20.8 Creation procedure property

An effective class has at least one creation procedure.

Informative text

Those explicitly listed if any, otherwise *default_create*.

End

8.20.9 Definition: Creation procedures of a type

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

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

8.20.10 Definition: Available for creation; general creation procedure

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**.

8.20.11 Syntax: Creation instructions

Creation_instruction $\triangleq$ **create** [Explicit_creation_type] Creation_call

Explicit_creation_type $\triangleq$ "**{**" Type "**}**"

Creation_call ≜ Variable [Explicit_creation_call]

Explicit_creation_call ≜ "." Unqualified_call

8.20.12 Definition: Creation target, creation type

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 instruction's target.

8.20.13 Semantics: Creation Type theorem

The creation type of a creation instruction is always effective.

8.20.14 Definition: Unfolded form of a creation instruction

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*.

8.20.15 Validity: Creation Instruction rule **Validity code:** *VGCI*

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 target's type.

2 The 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

In spite of its compactness, the Creation Instruction rule suffices in fact 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:

- 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_clause with either no Clients specification or one that lists an ancestor of *C*).
- 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.
- 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.

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

8.20.16 Validity: Creation Instruction properties

Validity code: *VGCP*

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 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 to *C* for creation.

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 to *C* for creation.

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:

7 *CT* denotes a constrained generic parameter.

8 The Constraint for *CT* specifies one or more procedures as constraining creators.

9 If *ci* has no Creation_call, one of the constraining creators is the Constraint's version of *default_create* from *ANY*.

10 If *ci* has a Creation_call, one of the constraining creators is the feature_of the Creation_call.

Informative text

Compiler writers may refer, in error messages, to either these "Creation Instruction Properties" or the earlier "Creation Instruction rule" 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

8.20.17 Semantics: Creation Instruction Semantics

The effect of a creation instruction of 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 feature of the Unqualified_call of the instruction's unfolded form.

4 Attach *x* to the object.

Informative text

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 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.

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.

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*.

Also note the order of steps: attachment to the target *x* is the last operation. Until then, *x* retains its earlier value, void if *x* is a previously unattached reference.

End

8.20.18 Syntax: Creation expressions

Creation_expression $\triangleq$ **create** Explicit_creation_type [Explicit_creation_call]

8.20.19 Definition: Properties of a creation expression

The **creation type** and **unfolded form** of a creation expression are defined as for a creation instruction.

8.20.20 Validity: Creation Expression rule **Validity code:** *VGCE*

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 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.

8.20.21 Validity: Creation Expression Properties **Validity code:** *VGCX*

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 to *C* for creation.
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 to *C* for creation.
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:

6 *CT* denotes a constrained generic parameter.
7 The Constraint for *CT* specifies one or more procedures as constraining creators.
8 If *ce* has no Creation_call, one of the constraining creators is the Constraint's version of *default_create* from *ANY*.

9 If *ce* has a Creation_call, one of the constraining creators is the feature of the Creation_call.

Informative text

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

8.20.22 Semantics: Creation Expression Semantics

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 feature of the Unqualified_call of the expression’s unfolded form.

Informative text

The notes appearing after the Creation Instruction Semantics rule also apply here.

End

8.21 Comparing and duplicating objects

Informative text

The just studied Creation instruction is the basic language mechanism for obtaining new objects at run time; it produces fresh direct instances of a given class, initialized from scratch.

Sometimes you will need instead to copy the contents of an existing object onto those of another. This is the **copying** operation.

A variant of copying is **cloning**, which produces a fresh object by duplicating an existing one.

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.

A closely related problem is that of *comparing* two objects for shallow or deep equality.

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

8.21.1 Object comparison features from *ANY*

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* /= *Void*) **and then** *other*.*default_is_equal* (**Current**))
consistent: **Result implies** *is_equal* (*other*)
is_equal (*other*: **? like** *Current*)
-- Is *other* attached to object considered equal
-- to current object?
ensure
same_type: **Result implies** *same_type* (*other*)
symmetric: **Result** = ((*other* /= *Void*) **and then** *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

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.

Both functions accept a void argument and will in that case, as the header comment implies, return **False**.

End

8.21.2 Syntax: Equality expressions

Equality $\triangleq$ Expression Comparison Expression

Comparison $\triangleq$ "**=**" | "**/=**" | "**~**" | "**/~**"

8.21.3 Semantics: Equality Expression Semantics

The Boolean_expression *e* **~** *f* has value true if and only if the values of *e* and *f* are both attached and such that *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

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.

End

8.21.4 Semantics: Inequality Expression Semantics

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.

8.21.5 Copying and cloning features from *ANY*

The features whose contract views appear below are provided by class *ANY* as secret features.

copy (*other*: **? 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: ? 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)
    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

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.

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*.

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

8.21.6 Deep equality, copying and cloning

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. Detailed descriptions are part of the ELKS specification.

Informative text

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* version recursively compare or copy entire object structures.

End

8.22 Attaching values to entities

Informative text

At any instant of a system’s execution, every entity of the system has a certain attachment status: it is either attached to a certain object, or void (attached to no object). Initially, all entities of reference types are void; one of the effects of a Creation instruction is to attach its target to an object.

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.

The validity and semantic properties of these two mechanisms are essentially the same; we study them jointly here.

End

8.22.1 Definition: Reattachment, source, target

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).

Informative text

We group assignment and argument passing into the same category, reattachment, because their validity and semantics are essentially the same:

- 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.
- The semantics in both cases is to attach the target to the value of the source or a copy of that value.

End

8.22.2 Syntax: Assignments

Assignment $\triangleq$ Variable ":=" Expression

8.22.3 Validity: Assignment rule

Validity code: *VBAR*

An Assignment is valid if and only if its source expression is compatible with its target entity.

Informative text

To be “compatible” means to conform or convert.

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**, part of the general discussion of call validity.

End

8.22.4 Semantics: Reattachment principle

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*.

8.22.5 Semantics: Attaching an entity, attached entity

Attaching an entity *e* to an object O is the operation ensuring that the value of *e* becomes **attached to O**.

Informative text

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:

- 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.
- 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.

End

8.22.6 Semantics: Reattachment Semantics

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

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.

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** to *target*.

Case 2 is, from the validity rules, possible only if both *target* and *source* are declared of *detachable* types.

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.

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.

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 an reference type.

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

8.22.7 Semantics: Assignment Semantics

The effect of a reassignment *x* := *y* is determined by the Reattachment Semantics rule, with source *y* and target *x*.

Informative text

The other cases where Reattachment Semantics applies is actual-formal association, per step 5 of the General Call rule.

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

8.22.8 Definition: Dynamic type

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.

8.22.9 Definition: Polymorphic expression; dynamic type and class sets

An expression that has two or more possible dynamic types is said to be **polymorphic**.

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*.

8.22.10 Syntax: Assigner calls

Assigner_call $\triangleq$ Expression ":=" Expression

Informative text

The left-hand side is surprisingly general: any expression. The validity rule 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*.

You could even use operator syntax, as in

a + *b* := *c*

assuming that, in the type of *a*, the function *plus* **alias** "+" has been defined with an assigner command, maybe a procedure *subtract*. Then the left side *a* + *b* is just an abbreviation for the query call

a.*plus* (*b*)

and the Assigner_call is just an abbreviation for the procedure call

a.*subtract* (*c*, *b*)

End

8.22.11 Validity: Assigner Call rule **Validity 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 The Equivalent Dot Form of *target* is a qualified Object_call whose feature has an assigner command.

8.22.12 Semantics: Assigner Call semantics

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

This confirms that the construct is just an abbreviation for a procedure call.

End

8.23 Feature call

Informative text

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.

This is feature call, one of the most important constructs in Eiffel's object-oriented approach, and the topic of the following discussions.

End

8.23.1 Validity: Call Use rule **Validity code:** *VUCN*

A Call of feature *f* denotes:

1 If *f* is a query (attribute or a function): an expression.

2 If *f* is a procedure: an instruction.

8.23.2 Syntax: Feature calls

Call ≜ Object_call | Non_object_call

Object_call ≜ [Target "**.**"] Unqualified_call

Unqualified_call ≜ Feature_name [Actuals]

Target ≜ Local | Read_only | Call | Parenthesized_target

Parenthesized_target ≜ "**(|**" Expression "**|)**"

Non_object_call ≜ "**{**" Type "**}**" "**.**" Unqualified_call

Informative text

A call is most commonly of the form *a*.*b*.… where *a*, *b* … are features, possibly with arguments. Target allows a Call to apply to an explicit target object (rather then the current object); it can itself be a Call, allowing multidot calls. Other possible targets are a local variable, a Read_only (including formal arguments and **Current**) a "non-object call" (studied below), or a complex expression written as a Parenthesized_target (|…|).

End

8.23.3 Syntax: Actual arguments

Actuals ≜ "**(**" Actual_list "**)**"

Actual_list ≜ {Expression "**,**" …}$^{+}$

8.23.4 Definition: Unqualified, qualified call

An Object_call is **qualified** if it has a Target, **unqualified** otherwise.

Informative text

In equivalent terms, a call is “unqualified” if and only if it consists of just an Unqualified_call component.

The call *f* (*a*) is unqualified, *x*.*f* (a) is qualified.

Another equivalent definition, which does not explicitly refer to the syntax, is that a call is qualified if it contains one or more dots, unqualified if it has no dots — counting only dots at the dot level, not those that might appear in arguments; for example *f* (*a*.*b*) is unqualified.

End

8.23.5 Definition: Target of a call

Any Object_call has a **target**, defined as follows:

1 If it is qualified: its Target component.

2 If it is unqualified: **Current**.

Informative text

The target is an expression; in *a* (*b, c*).*d* the target is *a* (*b, c*) and in (| *a* (*b, c*) + *x* |).*d* the target (case 1) is *a* (*b, c*) + *x*. In a multidot case the target includes the Call deprived of its last part, for example *x*.*f (args)*.*g* in *x*.*f (args)*.*g*.*h (args1)*.

End

8.23.6 Definition: Target type of a call

Any Call has a **target type**, defined as follows:

1 For an Object_call: the type of its target. (In the case of an Unqualified_call this is the current type.)

2 For a Non_object_call having a type *T* as its Type part: *T*.

8.23.7 Definition: Feature of a call

For any Call the “**feature 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 of its Unqualified_call part.

Informative text

Case 1 tells us that the feature of *f* (*args*) is *f* and the feature of *g*, an Unqualified_call to a feature without arguments, is *g*.

The term is a slight abuse of language, since *f* and *g* are feature names rather than features. The actual feature, deduced from the semantic rules given below and involving dynamic binding. is the dynamic feature of the call.

It follows from case 2 that the feature of a qualified call *x*.*f (args)* is *f*. The recursive phrasing addresses the multidot case: the feature of *x*.*f (args)*.*g*.*h (args1)* is *h*.

End

8.23.8 Definition: Imported form of a Non_object_call

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 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 Declaration_body obtained from the Declaration_body of *f* by replacing every type by its deanchored form, then applying the generic substitution of *T*.

Informative text

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*.

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

8.23.9 Validity: Non-Object Call rule

Validity 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

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

8.23.10 Semantics: Non-Object Call Semantics

The effect of a Non_object_call is that of its imported form.

8.23.11 Validity: Export rule

Validity code: *VUEX*

An Object_call appearing in a class *C*, with *fname* as the feature of the call, is **export-valid** for *C* if and only if it satisfies the following conditions.

1 *fname* is the final name of a feature of the target type of the call.

2 If the call is qualified, that feature is available to *C*.

Informative text

For an unqualified call *f* or *f (args)*, only condition 1 is applicable, requiring simply (since the target type of an unqualified class is the current type) that *f* be a feature, immediate or inherited, of the current class.

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*.)

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.

End

8.23.12 Validity: Export Status principle

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

This is a validity property, but it has no code since it is not a separate rule, just a restatement for emphasis of condition 2 of the Export rule.

End

8.23.13 Validity: Argument rule **Validity code:** *VUAR*

An export-valid call of target type *ST* and feature *fname* appearing in a class *C* where it denotes a feature *sf* is **argument-valid** if and only if it 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*.

Informative text

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* with an entity if its type either conforms or converts to the entity's type.

End

8.23.14 Validity: Target rule **Validity code:** *VUTA*

An Object_call is **target-valid** if and only if either:

1 It is unqualified.

2 Its target is an attached expression.

Informative text

Unqualified calls (case 1) are always target-valid since they are applied to the current object, which by construction is not void.

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 compilers — 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

8.23.15 Validity: Class-Level Call rule **Validity code:** *VUCC*

A call of target type *ST* is **class-valid** if and only if it is export-valid, argument-valid and target-valid.

8.23.16 Definition: Void-Unsafe

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**. Void-unsafe systems are not valid Eiffel systems.

8.23.17 Definition: Target Object

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.

8.23.18 Semantics: Failed target evaluation of a void-unsafe system

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*.

8.23.19 Definition: Dynamic feature of a call

Consider an execution of a call of feature *fname* and target object O. Let *ST* be its target type and *DT* the type of O. The **dynamic feature** of the call is the dynamic binding version in *DT* of the feature of name *fname* in *ST*.

Informative text

Behind the soundness of this definition stands a significant part of the validity machinery of the language:

- The rules on reattachment imply that *DT* conforms to *ST*.
- The Export rule imply that *fname* is the name of a feature of *ST* (meaning a feature of the base class of *ST*).
- As a consequence, this feature has a version in *DT*; it might have several, but the definition of “dynamic binding version” removes any ambiguity.

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've taken care of the two key parts of Object_call semantics, although we still have to integrate a few details and special cases.

End

8.23.20 Definition: Freshness of a once routine call

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* has the once key "*THREAD*".

3 Its target was not the current object, and *r* has the once key "*OBJECT*".

4 After it was started, a call was executed to one of the refreshing features of *onces* from *ANY*, including among the keys to be refreshed at least one of the once keys of *r*.

8.23.21 Definition: Latest applicable target of a non-fresh call

The **latest applicable target** of a non-fresh call to a once function *df* to a target object O is last value to which it was attached in the call to *df* most recently started on:

1 If *df* has the once key "*OBJECT*": O.

2 Otherwise, if *df* has the once key "*OBJECT*": any target in the current thread.

3 Otherwise: any target in any thread.

8.23.22 Semantics: Once Routine Execution Semantics

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 the Non-once Routine Execution rule.

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** for *df* to the reused target of the call.

Informative text

Case 2 is known as "*once an 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.

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 entity **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

8.23.23 Semantics: Current object, current routine

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.

4 If *cr* starts executing any construct whose semantics does not involve a call: the current object and current routine remain the same.

8.23.24 Semantics: Current Semantics

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.

8.23.25 Semantics: Non-Once Routine Execution rule

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, 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.

8.23.26 Semantics: General Call Semantics

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 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*.

5 Attach every formal argument of *df* to the corresponding element of *arg_values* by applying the Reattachment Semantics rule.

6 If the call is qualified and class invariant monitoring is on, evaluate the class invariant of O's base type on O.

7 If precondition monitoring is on, evaluate the precondition of *df*.

8 If *df* is not an attribute, not a once routine and not external, apply the Non-Once Routine Execution rule to O and *df*.

9 If *df* is a once routine, apply the Once Routine Execution rule to O and *df*.

10 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.

11 If *df* is a self-initializing attribute and has not yet been initialized, initialize it through the Default Initialization rule.

12 If the call is qualified and class invariant monitoring is on, evaluate the class invariant of O's base type on O.

13 If postcondition monitoring is on, evaluate the postcondition 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.

8.23.27 Definition: Type of a Call used as expression

Consider a call denoting an expression. Its **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.

8.23.28 Semantics: Call Result

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 the Non-Once Execution rule.

2 If *df* is a once function: the value attached to **Result** as a result of the application of the Once Execution rule.

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.

8.23.29 Semantics: Value of a call expression

The **value** of a Call *c* used as an expression is, at any run-time moment, the result of executing *c*.

8.24 Eradicating void calls

Informative text

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.

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?

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.

The basic idea is simple. Its the combination of three rules:

- 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).
- 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 **?***T*.
- The validity rules ensure that attached types — those without a **?** — 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

8.24.1 Syntax: Object test

Object_test ≜ "**{**" Identifier "**:**" Type "**}**" Expression

Informative text

An Object_test of the form {*x*: *T*} *exp*, 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* over the execution of a neighboring part of the text known as the **scope** of the Object_test. For example, in **if** {*x*: *T*} *exp* **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*.

The following rules define these notions precisely.

End

8.24.2 Definition: Object-Test Local

The **Object-Test Local** of an Object_test is its Identifier component.

8.24.3 Validity: Object Test rule

Validity code: *VUOT*

An Object_test {*x*: *T*} *exp* is valid if and only if it satisfies the following conditions:

1 Its Object-Test Local *x* does not have the same lower name as any feature of the enclosing class, or any formal argument or local variable of the enclosing routine and Inline_agent if any.

2 Its type *T* is an attached type.

Informative text

Condition 2 reflects the intent of an Object_test: to test whether an expression is *attached* to an instance of a given type. It would make no sense then to use a detachable type.

End

8.24.4 Definition: Conjunctive, disjunctive, implicative; Term, semistrict term

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 (respectively **and then**, **or else**, **implies**).

8.24.5 Definition: Scope of an Object-Test Local

The scope of the Object-Test Local of an Object_test *ot* includes any applicable program element from the following:

1. If *ot* is a semistrict term of a conjunctive expression: any subsequent terms.
2. If *ot* is a term of an implicative expression: the next term.
3. If **not** *ot* is a semistrict term of a disjunctive expression *e*: any subsequent terms.
4. If *ot* is a term of a conjunctive expression serving as the Boolean_expression in the Then_part in a Conditional: the corresponding Compound.
5. If **not** *ot* is a term of a disjunctive expression serving as the Boolean_expression in the Then_part in a Conditional: any subsequent Then_part and Else_clause.
6. If **not** *ot* is a term of a disjunctive expression serving as the Exit_condition in a Loop: the Loop_body.
7. If **not** *ot* is a term of a conjunctive expression used as Unlabeled_assertion_clause in a Precondition: the subsequent components of the Attribute_or_routine.
8. If **not** *ot* is a term of a conjunctive expression used as Unlabeled_assertion_clause in a Check: its Compound.

Informative text

The definition ensures that, for an Object_test {*x*: *T*} *exp*, 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

8.24.6 Semantics: Object Test semantics

The value of an Object_test {*x*: *T*} *exp* is true if the value of *exp* is attached to an instance of *T*, false otherwise.

Informative text

In particular, if *x* is void (which is possible only if *T* is a detachable type), the result will be false.

End

8.24.7 Semantics: Object-Test Local semantics

For an Object_test {*x*: *T*} *exp*, the value of *x*, defined only over its scope, is the value of *exp* at the time of the Object_test's evaluation.

8.24.8 Definition: Read-only void test

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.

8.24.9 Definition: Scope of a read-only void test

The **scope** of a read-only void test appearing in a class text, for *e* of type *T*, is the scope that the Object-Test Local *ot* would have if the void test were replaced by:

1 For *e* = *Void*: {*ot*: *T*} *e*.

2 For *e* /= *Void*: **not** ({*ot*: *T*} *e*)

Informative text

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.

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

8.24.10 Definition: Certified Attachment Pattern

A **Certified Attachment Pattern** (or **CAP**) for an expression *exp* whose type is detachable 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

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.

- 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**.
- 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

8.24.11 Definition: Attached expression

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

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

8.25 Typing-related properties

Informative text

This Part does not define any new rules, only a few definitions that facilitate discussion of type issues.

End

8.25.1 Definition: Catcall

A **catcall** is a run-time attempt to execute a Call, such that the feature of the call is not applicable to the target of the call.

Informative text

The role of the type system is to ensure that a valid system can never, during its execution, produce a catcall.

"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

8.25.2 Validity: Descendant Argument rule

Validity code: *VUDA*

Consider a call of target type *ST* and feature *fname* appearing in a class *C*. Let *sf* the feature of final name *fname* in *ST*. Let *DT* be a type conforming to *ST*, and *df* the version of *sf* in *D*. The call is **descendant-argument-valid** for *DT* if and only if it satisfies the following conditions:

1 The call is argument-valid.

2 Every actual argument conforms, after conversion to the corresponding formal argument of *sf* if applicable, to the corresponding formal argument of *df*.

8.25.3 Validity: Single-level Call rule

Validity code: *VUSC*

A call with target *x* is **system-valid** if for any element *D* of the dynamic class set of *x* it is export-valid for *D* and descendant-argument-valid for *D*.

8.25.4 Definition: System-valid, valid

A call of target *target* is **system-valid** if and only if it is class-valid for *target* assumed to be of any type of its dynamic type set.

Informative text

The goal of the various type mechanisms and rules of the language is to ensure that every call is both class-valid and system-valid.

End

8.25.5 Definition: Dynamic type set

The dynamic type set of an expression *e* is the set of types of all objects that can become attached to *e* during execution.

8.26 Exception handling

Informative text

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.

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

8.26.1 Definition: Failure, exception, trigger

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 routine, the execution of one of the components of the routine's Body fails, this prevents the routine's execution from continuing the Body's execution normally; such an event is said to **trigger** an **exception**.

Informative text

Examples of exception causes include:

- Assertion violation (in an assertion monitoring mode).
- Failure of a called routine.
- Impossible operation, such as a Creation instruction attempted when not enough memory is available, or an arithmetic operation which would cause an overflow or underflow in the platform's number system.
- 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.
- An exception explicitly raised by the software itself.

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

8.26.2 Syntax: Rescue clauses

Rescue $\triangleq$ **rescue** Compound

Retry $\triangleq$ **retry**

8.26.3 Validity: Rescue clause rule

Validity code: *VXRC*

It is valid for a Routine to include a Rescue clause if and only if its Feature_body is an Effective_routine of the Internal form.

Informative text

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

8.26.4 Validity: Retry rule

Validity code: *VXRT*

A Retry instruction is valid if and only if it appears in a Rescue clause.

Informative text

Because this constraint requires the Retry physically to appear within the Rescue clause, it is not possible for a Rescue to call a procedure containing a Retry. In particular, a redefined version of *default_rescue* (see next) may not contain a Retry.

8.26.5 Definition: Exception-correct

A routine is **exception-correct** if any branch of the Rescue clause not terminating with a Retry ensures the invariant.

8.26.6 Semantics: Default Rescue Original Semantics

Class *ANY* introduces a non-frozen procedure *default_rescue* with no argument and a null effect.

Informative text

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

8.26.7 Definition: Rescue block

Any Internal or 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

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.

Condition 3 avoids endless recursion in the case of *default_rescue* itself.

End

8.26.8 Semantics: Exception Semantics

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 rescue block executes a Retry, 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 rescue block executes to the end without executing a Retry), 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

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.

In step 1, the Retry will only re-execute the Feature_body of *r*, with all entities set to their current value; it does **not** repeat argument passing and local variable initialization. This may be used to ensure that the execution takes a different path on a new attempt.

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".

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 a Retry may execute the body two or more times. 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

8.26.9 Definition: Type of an exception

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.

8.26.10 Semantics: Exception Cases

The triggering of an exception in a routine *r* called by a routine *caller* results in the setting of the following properties, accessible through features of 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 rule	
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

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": per case 3 of the Exception Semantics rule, a Retry is applicable only if the recipient is itself. Otherwise a *ROUTINE_FAILURE* will be triggered in the *caller*.

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

8.26.11 Semantics: Exception Properties

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

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 a Retry, 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

8.26.12 Definition: Ignoring, continuing an exception

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

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

8.27 Agents, iteration and introspection

Informative text

Objects represent information equipped with operations. These are clearly defined concepts; no one would mistake an operation for an object.

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.

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

8.27.1 Definition: Operands of a call

The **operands** of a call include its target (explicit in a qualified call, implicit in an unqualified call), and its arguments if any.

8.27.2 Definition: Operand position

The target of a call has **position** 0. The *i*-th actual argument, for any applicable *i*, has **position** *i*.

8.27.3 Definition: Construction time, call time

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.

8.27.4 Syntactical forms for a call agent

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:

- You may replace any argument by a question mark **?**, making the argument open.
- You may replace the target, by {*TYPE*} where *TYPE* is the name of a type, making the target open.
- You may remove the argument list (…) altogether, making all arguments open.

Informative text

This is not a formal syntax definition, but a summary of the available forms permitted by the syntax and validity rules that follow.

End

8.27.5 Syntax: Agents

Agent $\triangleq$ Call_agent | Inline_agent

Call_agent $\triangleq$ **agent** Call_agent_body

Inline_agent $\triangleq$ **agent** [Formal_arguments] [Type_mark] [Attribute_or_routine] [Agent_actuals]

8.27.6 Syntax: Call agent bodies

Call_agent_body $\triangleq$ Agent_qualified | Agent_unqualified

Agent_qualified $\triangleq$ Agent_target "**.** " Agent_unqualified

Agent_unqualified $\triangleq$ Feature_name [Agent_actuals]

Agent_target $\triangleq$ Entity | Parenthesized | Manifest_type

Agent_actuals $\triangleq$ "**(**" Agent_actual_list "**)**"

Agent_actual_list $\triangleq$ {Agent_actual "**,**" …}*

Agent_actual $\triangleq$ Expression | Placeholder

Placeholder $\triangleq$ "**?**"

8.27.7 Definition: Target type of an agent call

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).

8.27.8 Validity: Call Agent rule **Validity 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_actuals 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 Actual kind is of a type conforming to the type of the corresponding formal in *f*.

8.27.9 Definition: Associated feature of an inline agent

Every inline agent *ia* of a class *C* has an **associated feature**, defined as a fictitious routine 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 The Routine of *f* is defined by the Routine part of *ia*.

5 *f* is a function if *ia* has a Type_mark (its return type being given by the Type in that Type_mark), a procedure otherwise.

8.27.10 Validity: Inline Agent rule **Validity 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.

8.27.11 Validity: Inline Agent Requirements **Validity code:** *VPIR*

An Inline_agent *a* must satisfy the following conditions:

1 No formal argument or local variable of *a* has the same name as a feature of the enclosing class.

2 Every entity appearing in the Routine 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 Routine is not of the Deferred form.

Informative text

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.

End

8.27.12 Definition: Call-agent equivalent of an inline agent

An inline agent *ia* with *n* formal arguments ($n \geq 0$) has a **call-agent equivalent** defined as the Call_agent

agent *f* (**?**, **?**, …, **?**, a_1, a_2, …, a_m)

using *n* question marks, where a_1, a_2, …, a_m ($m \geq 0$) are the formal arguments and local variables of the enclosing routine (if any) and any enclosing agents, and *f* is the associated feature of *ia*. (If both *n* and *m* are 0, the Call_agent is just **agent** *f*.)

8.27.13 Semantics: Semantics of inline agents

The semantic properties of an inline agent are those of its call-agent equivalent.

8.27.14 Semantics: Use of Result in an inline function agent

In an agent of the Inline_routine form denoting a function, the local variable **Result** denotes the result of the agent itself.

8.27.15 Definition: Open and closed operands

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.

8.27.16 Definition: Open and closed operand positions

The **open operand positions** of a Feature_agent are the operand positions of its open operands, and the **closed operand positions** those of its closed operands.

8.27.17 Definition: Type of an agent expression

Consider a Call_agent *a*, whose associated feature *f* has a generating type *T0*. Let *i1*, …, *im* (*m* ≥ 0) be its open operand positions, if any, and let T_{i1}, .., T_{im} be the types of *f*'s formals at positions *i1*, …, *im* (taking T_{i1} to be *T0* if *i1* = 0).

The type of *d* 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*.

8.27.18 Semantics: Agent Expression semantics

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.

8.27.19 Semantics: Effect of executing *call* on an agent

Let D0 be an agent object with associated feature *f* and open positions *i1*, …, *im* (*m* ≥ 0). The information in D0 enables a call to the procedure *call*, executed at any **call time** posterior to D0's construction time, with target D0 and (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.

8.28 Expressions

Informative text

Through the various forms of Expression, software texts can include denotations of run-time values — objects and references.

Previous discussions have already introduced some of the available variants of the construct: Formal, Local, Call, Old, Manifest_tuple, Agent. 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.

End

8.28.1 Syntax: Expressions

Expression $\triangleq$ Basic_expression | Special_expression

Basic_expression $\triangleq$ Read_only | Local | Call | Precursor | Equality | Parenthesized | Old | Operator_expression | Bracket_expression | Creation_expression

Special_expression ≜ Manifest_constant | Manifest_tuple | Agent | Object_test | Once_string | Address

Parenthesized ≜ "**(**" Expression "**)**"

Address ≜ "**$**" Variable

Once_string ≜ **once** Manifest_string

Boolean_expression ≜ Basic_expression | Boolean_constant | Object_test

8.28.2 Definition: Subexpression, operand

The **subexpressions** of an expression *e* are *e* itself and (recursively) all the following expressions:

1. For a Parenthesized (*a*) or a Parenthesized_target (|*a*|): the subexpressions of *a*.
2. For an Equality or Binary_expression *a* § *b*, where § is an operator: the subexpressions of *a* and of *b*.
3. For a Unary_expression ◊ *a*, where ◊ is an operator: the subexpressions of *a*.
4. For a Call or Precursor expression: the subexpressions of the Actuals part, if any, of its Unqualified_part
5. For an Agent: the subexpression of its Agent_actuals if any.
6. For a qualified call: the subexpressions of its target.
7. For a Bracket_expression *f* [*a*1, … a_n]: the subexpressions of *f* and those of all of *a*1, … a_n.
8. For an Old expression **old** *a*: *a*.
9. For a Manifest_tuple [*a*1, … a_n]: the subexpressions of all of *a*1, … a_n.

In cases 2 and 3, the **operands** of *e* are *a* and (in case 2) *b*.

8.28.3 Semantics: Parenthesized Expression Semantics

If *e* is an expression, the value of the Parenthesized (*e*) is the value of *e*.

8.28.4 Syntax: Operator expressions

Operator_expression ≜ Unary_expression | Binary_expression

Unary_expression ≜ Unary Expression

Binary_expression ≜ Expression Binary Expression

8.28.5 Operator precedence levels

13	**.**	(Dot notation, in qualified and non-object calls)
12	**old**	(In postconditions)
	not + –	**Used as unary**
	All free unary operators	
11	All free binary operators.	
10	^	(Used as binary: power)
9	* **/ // **	(As binary: multiplicative arithmetic operators)
8	**+ –**	**Used as binary**
7	**..**	(To define an interval)
6	**= /= ~ /~ < > <= >=**	(As binary: 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

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.

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

8.28.6 Definition: Parenthesized Form of an expression

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 ♣ different 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.

8.28.7 Definition: Target-converted form of a binary expression

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 declaration of *f* includes a **convert** mark and the type *TY* of *y* is not compatible with the type of the formal argument of *f*: ({*TY*} [*x*]) § *y*.

2 Otherwise: the original expression, *x* § *y*.

Informative text

({*TY*} [*x*]) denotes *x* converted to type *TY*. This definition allows us, if the feature from *x's* type *TX* cannot accept a *TY* argument but has explicitly been specified, through the **convert** mark, to allow for target conversion, and *TY* does include the appropriate feature accepting a *TX* argument, to use that feature instead.

The archetypal example is *your_integer + your_real* which, with the appropriate **convert** mark in the "+" feature in *INTEGER*, we can interpret as ({*REAL*} [*your_integer*]) + *your_real*, where "+" represents the *plus* feature from *REAL*.

End

8.28.8 Validity: Operator Expression rule **Validity code:** *VWOE*

A Unary_expression § *x* or Binary_expression *x* § *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.

8.28.9 Semantics: Expression Semantics (strict case)

The **value** of an Expression, other than a Binary_expression whose Binary is semistrict, is the value of its Equivalent Dot Form.

Informative text

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.

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.

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

8.28.10 Definition: Semistrict operators

A **semistrict operator** is any one of the three operators **and then, or else** and **implies**, applied to operands of type *BOOLEAN*.

8.28.11 Semantics: Operator Expression Semantics (semistrict cases)

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

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.

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

8.28.12 Syntax: Bracket expressions

Bracket_expression $\triangleq$ Bracket_target "**[**" Actuals "**]**"

Bracket_target $\triangleq$ Target | Once_string | Manifest_constant | Manifest_tuple

Informative text

Target 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

8.28.13 Validity: Bracket Expression rule **Validity code:** *VWBR*

A Bracket_expression *x* [*i*] 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.

8.28.14 Definition: Equivalent Dot Form of an expression

Any Expression *e* has an **Equivalent Dot Form**, not involving (in any of its subexpressions) any Bracket_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 (obtained 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_expression of 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* has no subexpression other than itself: *pe*.
5. In all other cases: (recursively) the result of replacing every subexpression of *e* by its Equivalent Dot Form.

8.28.15 Validity: Boolean Expression rule **Validity code:** *VWBE*

A Basic_expression is valid as a Boolean_expression if and only if it is of type *BOOLEAN*.

8.28.16 Validity: Identifier rule **Validity code:** *VWID*

An Identifier appearing in an expression in a class *C*, other than as the feature of an Object_call or qualified Call, must be the name of a feature of *C*, or a local variable of the enclosing routine or inline agent if any, or a formal argument of the enclosing routine or inline agent if any, or the Object-Test Local of an Object_test.

Informative text

The restriction "other than as the feature of an Object_call or 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*'s type.

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

8.28.17 Definition: Type of an expression

The type of an Expression *e* is:

1. For the predefined Read_only **Current**: the current type.
2. For a routine's Formal argument : the type declared for *e*.
3. For an Object-Test local: its declared type.

4 For **Result**, appearing in the text of a query *f*: the result type of *f*.

5 For a Local variable other than **Result**: the type declared for *e*.

6 For a Call: the type of *e* as determined by the Expression Call Type definition with respect to the current type.

7 For a Precursor: (recursively) the type of its unfolded form.

8 For an Equality: *BOOLEAN*.

9 For a Parenthesized (*f*): (recursively) the type of *f*.

10 For **old** *f*: (recursively) the type of *f*.

11 For an Operator_expression or Bracket_expression: (recursively) the type of the Equivalent Dot Form of *e*.

12 For a Manifest_constant: as given by the definition of the type of a manifest constant.

13 For a Manifest_tuple $[a_1, \ldots a_n]$ ($n \geq 0$): *TUPLE* $[T_1, \ldots T_n]$ where each T_i is (recursively) the type of a_i.

14 For an Agent: as given by the definition of the type of an agent expression.

15 For an Object_test: *BOOLEAN*.

16 For a Once_string: *STRING*.

17 For an Address **$***v*: *TYPED_POINTER* [*T*] where *T* is (recursively) the type of *v*.

Informative text

Case 6, which refers to a definition given in the discussion of calls, also determines case 11, operator and bracket expressions.

End

8.29 Constants

Informative text

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.

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 $[item_1, \ldots item_n]$.

End

8.29.1 Syntax: Constants

Constant $\triangleq$ Manifest_constant | Constant_attribute

Constant_attribute $\triangleq$ Feature_name

8.29.2 Validity: Constant Attribute rule

Validity code: *VWCA*

A Constant_attribute appearing in a class *C* is valid if and only if its Feature_name is the final name of a constant attribute of *C*.

8.29.3 Syntax: Manifest constants

Manifest_constant $\triangleq$ [Manifest_type] Manifest_value

Manifest_type $\triangleq$ "**{**" Type "**}**"

Manifest_value $\triangleq$ 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

8.29.4 Syntax (non-production): Sign Syntax rule

If present, the Sign of an Integer_constant or Real_constant must immediately precede the associated Integer or Real, with no intervening tokens or components (such as breaks or comments).

8.29.5 Syntax (non-production): Character Syntax rule

The quotes of a Character_constant must immediately precede and follow the Character, with no intervening tokens or components (such as breaks or comments).

Informative text

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* '.

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.

End

8.29.6 Definition: Type of a manifest constant

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

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.

Case 7 involves the Kernel Library class *TYPE*.

End

8.29.7 Validity: Manifest-Type Qualifier rule

Validity code: *VWMQ*

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

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

8.29.8 Semantics: Manifest Constant Semantics

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*.

8.29.9 Definition: Manifest value of a constant

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

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 (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

8.29.10 Syntax: Manifest strings

Manifest_string $\triangleq$ Basic_manifest_string | Verbatim_string

Basic_manifest_string $\triangleq$ ' **"** ' String_content ' **"** '

String_content $\triangleq$ {Simple_string Line_wrapping_part …}$^+$

Verbatim_string $\triangleq$ Verbatim_string_opener Line_sequence Verbatim_string_closer

Verbatim_string_opener $\triangleq$ ' **"** ' [Simple_string] Open_bracket

Verbatim_string_closer $\triangleq$ Close_bracket [Simple_string] ' **"** '

Open_bracket $\triangleq$ "**[**" | "**{**"

Close_bracket $\triangleq$ "**]**" | "**}**"

Informative text

In the “basic” case, most examples of String_content involve just one Simple_string (a sequence of printable characters, with no new lines). 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.

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_sequence is kept exactly as is, whereas with **"[** … **]"** the lines are left-aligned (stripped of any common initial blanks and tabs). Details below.

End

8.29.11 Syntax (non-production): Line sequence

A specimen of Line_sequence is a sequence of one or more Simple_string components, each separated from the next by a single New_line.

8.29.12 Syntax (non-production): Manifest String rule

In addition to the properties specified by the grammar, every Manifest_string must satisfy the following properties:

1 The Simple_string components of its Line_sequence may 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

Like other “non-production” syntax rules, the last two rules capture simple syntax requirements not expressible through BNF-E productions.

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

8.29.13 Definition: Line_wrapping_part

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

This construct requires such a definition since it can’t be specified through a context-free syntax formalism such as BNF-E.

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_string the same indentation as to the neighboring elements. The definition also permits blanks and tabs after the 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

8.29.14 Semantics: Manifest string semantics

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.

8.29.15 Validity: Verbatim String rule **Validity code:** *VWVS*

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.

8.29.16 Semantics: 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_bracket is the value of its Line_sequence.

The value of a Verbatim_string using braces **[ ]** as Open_bracket and Close_bracket is the value of the left-aligned form of its Line_sequence.

Informative text

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

8.29.17 Definition: Prefix, longest break prefix, left-aligned form

A **prefix** of a string *s* is a string *p* of some length *n* (*n* ≥ 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* ≥ 0).

8.30 Basic types

Informative text

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.

The following presentation explains the general concepts behind the design and use of these classes.

End

8.30.1 Definition: Basic types and their sized variants

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*.

8.30.2 Definition: Sized variants of *STRING*

The sized variants of *STRING* are *STRING*, *STRING_8* and *STRING_32*.

8.30.3 Semantics: Boolean value semantics

Class *BOOLEAN* covers the two truth values.

The reserved words *True* and *False* denote the corresponding constants.

8.30.4 Semantics: Character types

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.

The expanded class *CHARACTER* describes characters with a length and encoding settable through a compilation option. The recommended default is Unicode.

8.30.5 Semantics: Integer types

The reference class *INTEGER_GENERAL* describes integers, signed or not, of arbitrary length. The expanded classes *INTEGER_xx*, for *xx* = 8, 16, 32 or 64, describe 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.

8.30.6 Semantics: Floating-point types

The reference class *REAL_GENERAL* describes floating-point numbers with arbitrary precision. The expanded classes *REAL_xx*, for *xx* = 32 or 64, describe IEEE floating-point numbers with *xx* bits of precision.

The expanded class *REAL* describes floating-point numbers with a precision settable through a compilation option. The recommended default is 64 bits.

8.30.7 Semantics: Address semantics

The expanded class *POINTER* describes addresses of Eiffel features or expressions, intended for transmission to non-Eiffel routines.

8.31 Interfacing with C, C++ and other environments

Informative text

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.

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.

The following presentation describes how Eiffel systems can integrate components from other languages and environments.

End

8.31.1 Syntax: External routines

External $\triangleq$ **external** External_language [External_name]

External_language $\triangleq$ Unregistered_language | Registered_language

Unregistered_language $\triangleq$ Manifest_string

External_name $\triangleq$ **alias** Manifest_string

Informative text

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.

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.

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.

The language name (External_language) 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.

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.

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

8.31.2 Validity: Address rule

Validity code: *VZAR*

An Address is valid if and only if its Address_mark is of a reference type.

Informative text

An expanded type would not make sense here as its values have copy rather than reference semantics.

End

8.31.3 Address Type rule

An argument of the Address form is of type *POINTER*.

8.31.4 Semantics: Address semantics

The value of an Address expression is a *POINTER* enabling foreign software to access the associated Variable.

Informative text

The manipulations that the foreign software can perform on the corresponding pointer depend on the foreign programming language. It is the implementation's responsibility to ensure that such manipulations do not violate Eiffel semantic properties.

End

8.31.5 Syntax: Registered languages

Registered_language $\triangleq$ C_external | C++_external | DLL_external

8.31.6 Syntax: External signatures

External_signature $\triangleq$ **signature** [External_argument_types] [**:** External_type]

External_argument_types $\triangleq$ "**(**" External_type_list "**)**"

External_type_list $\triangleq$ {External_type "**,**" …}*

External_type $\triangleq$ Identifier

8.31.7 Validity: External Signature rule

Validity 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

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.

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.

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

8.31.8 Semantics: External signature semantics

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.

8.31.9 Syntax: External file use

External_file_use $\triangleq$ **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

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:

- "System" files, used only in a C context, appear between angle brackets <> and refer to specific locations in the C library installation.
- 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

8.31.10 Validity: External File rule

Validity code: *VZEF*

An External_file is valid if and only if its Simple_string satisfies 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

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.

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

8.31.11 Semantics: External file semantics

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.

8.31.12 Syntax: C externals

C_external ≜ " " ' **" "** ' **C**
'[**inline**]
[External_signature] [External_file_use]
' **" "** '

Informative text

The C_external mechanism makes it possible, from Eiffel, to use the mechanisms of C. The syntax covers two basic schemes:

- 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.
- You may also write C code *within* the Eiffel routine, putting that code in the **alias** clause and specifying **inline**.

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.

The **alias** part is a Manifest_string of one of the two available forms:

- 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**.
- 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.

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

8.31.13 Validity: C external rule

Validity code: *VZCC*

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*.

8.31.14 Semantics: C Inline semantics

In an external routine *er* of the **inline** 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*.

8.31.15 Syntax: C++ externals

C++_external ≜ ' **"** ' **C++**
inline
[External_signature]
[External_file_use]
' **"** '

Informative text

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.

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

8.31.16 Validity: C++ external rule **Validity 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*.

8.31.17 Semantics: C++ Inline semantics

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*.

8.31.18 Syntax: DLL externals

DLL_external ≜ ' **"** ' **dll**
[**windows**]
DLL_identifier
[Blanks_or_tabs DLL_index]
[External_signature]
[External_file_use]
' **"** '

DLL_identifier ≜ Simple_string

DLL_index ≜ Integer

Informative text

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.

The mechanism assumes a dynamic loading facility, such as exist on modern platforms; it is specified to work with any such platform.

End

8.31.19 Validity: External DLL rule **Validity 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.

8.31.20 Semantics: External DLL semantics

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.

8.32 Lexical components

Informative text

The previous discussions 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. 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

8.32.1 Syntax (non-production): Character, character set

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

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 new-line) or %/59/. The details appear below.

End

8.32.2 Definition: Letter, alpha_betic, numeric, alpha_numeric, printable

A **letter** is any character belonging to one of the following categories:

1. Any of the following fifty-two, each a lower-case or upper-case element of the Roman alphabet:

   abcdefghijklmnopqrst u v w x y z

   ABCDEFGHIJKLMNOPQRSTUVWXYZ

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

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.

“*Printable*” characters exclude such special characters as new line and backspace.

Case 2 of the definition of “letter” refers to the 8-bit extended ASCII character set. Only the 7-bit ASCII character set 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.

End

8.32.3 Definition: Break character, break

A **break character** is one of the following characters:

- Blank (also known as space).
- Tab.
- New Line (also known as Line Feed).

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.

8.32.4 Semantics: Break semantics

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

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

8.32.5 Definition: Expected, free comment

A comment is **expected** if it appears in a construct as part of the style guidelines for that construct. Otherwise it is **free**.

8.32.6 Syntax (non-production): "Blanks or tabs", new line

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.

8.32.7 Syntax: Comments

Comment $\triangleq$ "– –" {Simple_string Comment_break …}*

Comment_break $\triangleq$ New_line [Blanks_or_tabs] "– –"

Informative text

This syntax implies that two or more successive comment lines, with nothing other than new lines to separate them, form a single comment.

End

8.32.8 Syntax (non-production): Free Comment rule

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

An example of construct whose specimens may not include comments is Line_sequence, defined 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.

Similarly, the Alias Syntax rule excludes any characters — and hence comments — between an Alias_name and its enclosing quotes.

End

8.32.9 Header comment rule

A feature Header_comment is an abbreviation for a Note clause of the form

```
note
    what: Explanation
```

where *Explanation* is a Verbatim_string with **[** and **]** as Open_bracket and Close_bracket and a Line_sequence made up of the successive lines (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

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 Blank_or_tabs. To make up the Verbatim_string we remove the Blank_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

8.32.10 Definition: Symbol, word

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**.

8.32.11 Syntax (non-production): Break rule

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.

Informative text

Without this rule, adjacent words not separated by a break — as in *ifxthen* — or adjacent symbols would be ambiguous.

End

8.32.12 Semantics: Letter Case rule

Letter case is significant for the following constructs: Character_constant, Manifest_string, 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.

8.32.13 Definition: Reserved word, keyword

The following names are **reserved words** of the language.

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**	**only**	**or**	**Precursor**
redefine	**rename**	**require**	**rescue**	**Result**	**retry**	**select**
separate	**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

The non-keyword reserved words, such as **True**, have a semantics of their own (**True** denotes one of the two boolean values).

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

8.32.14 Syntax (non-production): Double Reserved Word rule

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.

8.32.15 Definition: Special symbol

A **special symbol** is any of the following character sequences:

`--  :  ;  ,  ?  !  '  "  $  .  ->  :=`
`=  /=  ~  /~  (  )  (|  |)  [  ]  {  }`

8.32.16 Syntax (non-production): Identifier

An Identifier is a sequence of one or more alpha_numeric characters of which the first is a letter.

8.32.17 Validity: Identifier rule

Validity code: *VIID*

An Identifier is valid if and only if it is not one of the language's reserved words.

8.32.18 Definition: Predefined operator

A **predefined operator** is one of:

`=  /=  ~  /~`

Informative text

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

8.32.19 Definition: Standard operator

A **standard unary operator** is one of:

`+  -`

A **standard binary operator** is any one of the following one- or two-character symbols:

`+  -  *  /  ^  <  >`
`<=  >=  //  \\  ..`

Informative text

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

8.32.20 Definition: Operator symbol

An **operator symbol** is any non-alpha_numeric 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 **%**, question mark **?**, or exclamation mark **!**.

Informative text

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

8.32.21 Definition: Free operator

A **free operator** is 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.

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

Condition 3 gives us maximum flexibility without ambiguity; for example:

- You may **not** use **---** as an operator because, its subsequence **--** clashes with the special symbol introducing comments.
- 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.
- You **may**, however, use a single **-**, or define a free operator such as **-*** which does not cause any such confusion.
- You may **not** use **?**, **!, =** or **~**, but you **may** use operators containing these characters, for example **!=**.
- 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 the 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*).)

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

8.32.22 Syntax (non-production): Manifest character

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

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.

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.

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.

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

8.32.23 Special characters and their codes

Character	*Code*	*Mnemonic name*
@	**%A**	**A**t-sign
BS	**%B**	**B**ackspace
^	**%C**	**C**ircumflex
$	**%D**	**D**ollar
FF	**%F**	**F**orm feed
****	**%H**	Backslas**H**
~	**%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

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

8.32.24 Syntax (non-production): Percent variants

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

The characters "of" such a constant do not include the single **'** or double **"** quotes, which you must enter as themselves.

End

8.32.25 Semantics: Manifest character semantics

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.

8.32.26 Syntax (non-production): String, simple string

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).

8.32.27 Semantics: String semantics

The value of a String or Simple_string is the sequence of the values of its characters.

8.32.28 Syntax: Integers

Integer $\triangleq$ [Integer_base] Digit_sequence

Integer_base $\triangleq$ "**0**" Integer_base_letter

Integer_base_letter $\triangleq$ "**b**" | "**c**" | "**x**" | "**B**" | "**C**" | "**X**"

Digit_sequence $\triangleq$ Digit$^+$

Digit $\triangleq$ "**0**" | "**1**" | "**2**" | "**3**" | "**4**" | "**5**" | "**6**" | "**7**" | "**8**" | "**9**" |
"**a**" | "**b**" | "**c**" | "**d**" | "**e**" | "**f**" |
"**A**" | "**B**" | "**C**" | "**D**" | "**E**" | "**F**" | "_"

Informative text

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. Per the Letter Case rule the upper-case versions of these letters are permitted, although lower-case is the recommended style.

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

8.32.29 Validity: Integer rule

Validity 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

The rule has no requirement for the hexadecimal case, which accepts all the digits permitted by the syntax.

Integer is a purely lexical construct and does not include provision for a sign; the construct Integer_constant denotes possibly signed integers.

End

8.32.30 Semantics: Integer semantics

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.

Informative text

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*.

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

8.32.31 Syntax (non-production): Real number

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

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

8.32.32 Semantics: Real semantics

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).