Syntax, validity and semantics

2.1 OVERVIEW

To study the details of Eiffel, you will need a few conventions and basic rules. In particular, you will need to understand the role of the three levels of language description:

- **Syntax**, defining the textual structure of Eiffel software.
- **Validity**, defining when a syntactically well-formed software element has a meaning.
- **Semantics**, specifying what that meaning is, in terms of its effect on the software's execution.

Each of these levels conditions the next: validity is only defined for a syntactically legal element, and semantics only for a valid element. This chapter defines the three levels more precisely and introduces the notations used, in the rest of the book, to describe the syntax, validity and semantics of Eiffel constructs. It also offers an overview of *correctness*, a part of semantics.

Before proceeding, you should have read the note about the language description style after the Preface.

← *"About the language description", page xv.*

2.2 SYNTAX: COMPONENTS, SPECIMENS, CONSTRUCTS

Eiffel's syntax defines the structure of class texts.

The structure only: to express further limitations on legal texts, we need validity constraints; and to describe the effect of these texts, we need semantic rules.

→ *See below 2.7 and 2.8, starting on page 96, about validity constraints, and 2.9 about semantics.*

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**.

"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.

Here are the key syntax notions:

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.

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.

An important convention will simplify the discussions:

Construct Specimen convention

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

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

This example illustrates another convention

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).

Besides Class, examples of construct names include Parenthesized and Unlabeled_assertion_clause. Every non-terminal appears in the index with the page of its syntactical definition. An appendix gives the full list.

→ *Appendix J, Syntax in alphabetical order*

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*.

→ *"Textual conventions", page 94*.

2.3 TERMINALS, NON-TERMINALS AND TOKENS

Every construct is either a "terminal" or a "non-terminal":

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.

Chapter 32, about the lexical structure, explains the various kinds of tokens.

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.

An example of non-terminal is Conditional, whose specimens are "conditional instructions" such as

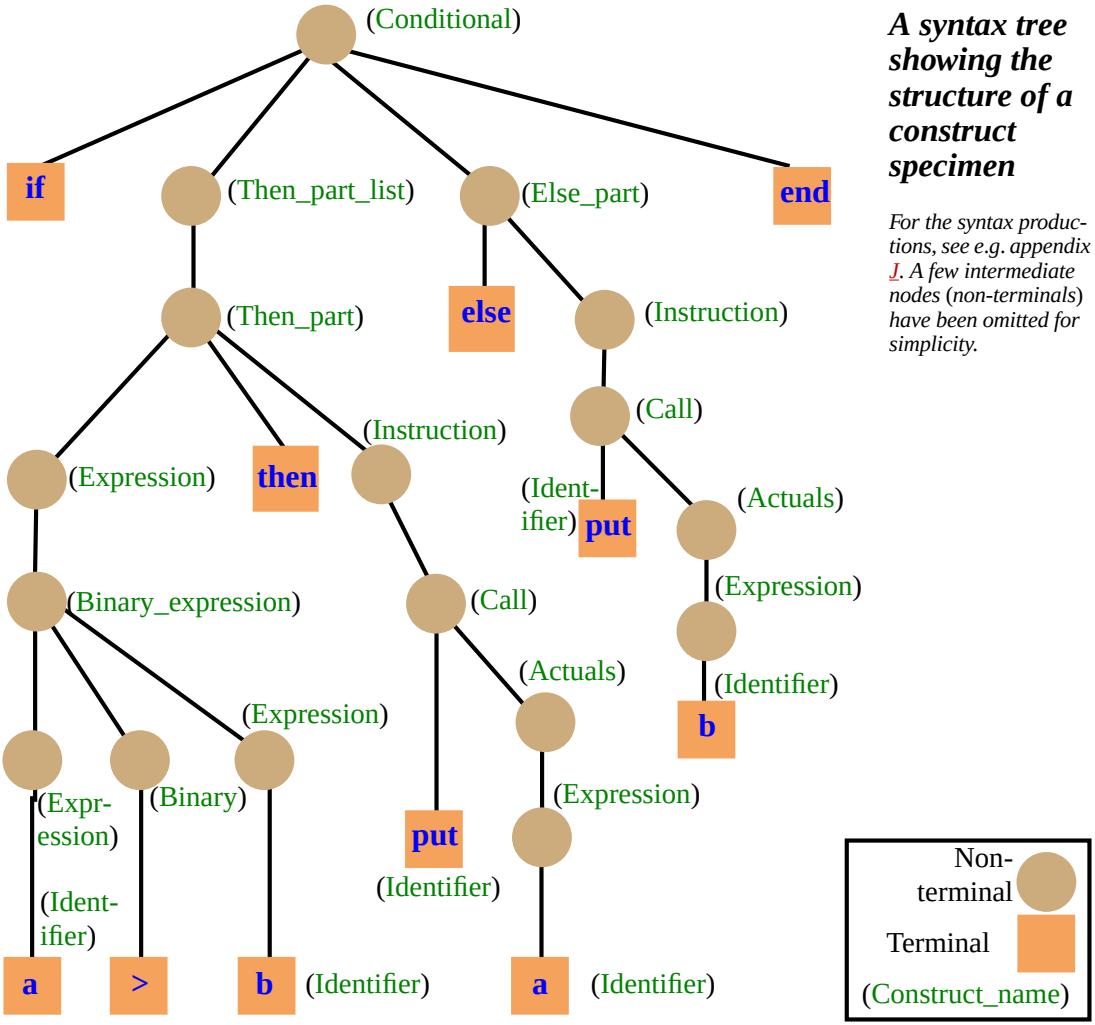
```
if a > b then put (a) else put (b) end
```

Such a non-terminal construct specimen includes further syntactical components, here the expression *a* > *b* and the instructions *put* (*a*) and *put* (*b*) — themselves specimens of non-terminals, with further sub-components. We may represent the full structure as a "syntax tree" as illustrated on the next page.

2.4 THE LEXICAL LEVEL

As the figure shows, constructs defining keywords and symbols, such as **if** and **;**, do not have construct names, since they each have a single specimen (**if** etc.) which you can use directly to refer to the construct.

Other terminal constructs such as Identifier represent many possible specimens (an infinity of them in the case of Identifier). So, like non-terminal constructs, they need a name and a description of how to obtain their specimens. They are called **lexical** constructs. Other examples of lexical constructs include Integer, denoting unsigned integer constants, such as *598*, and String, denoting sequences of arbitrary characters. As noted earlier, the specimens of a lexical construct, such as individual integers or strings, are called *tokens*.



A syntax tree showing the structure of a construct specimen

For the syntax productions, see e.g. appendix J. A few intermediate nodes (non-terminals) have been omitted for simplicity.

Lexical constructs are covered in a later chapter. Because tokens have such a simple structure, we usually don't need, for lexical constructs, the formal production rules that we will study next for non-terminal constructs. We satisfy ourselves with definitions such as "an Identifier is a sequence of one or more characters, of which the first is a letter and any subsequent one a letter or a number or an underscore".

→ *Chapter 32.*

2.5 PRODUCTIONS

To specify a non-terminal construct we need to define what its possible specimens are. That was true for terminals too, but here we require a little more machinery, in the form of a *production* for every non-terminal:

Production

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

Construct ≜ right-side

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

The symbol ≜ 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.

We need three kinds of right side:

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.

The rest of this section explores them in turn.

Aggregate productions

The right-side of an Aggregate production lists one or more constructs, some of which may be in square brackets to indicate optional parts. This specifies that, to obtain a specimen of the left-hand side construct, you simply provide a **succession** ("*aggregation*") of specimens of the listed constructs, in the order given. So the production

→ *Page 473*.

Conditional ≜ **if** Then_part_list [Else_part] **end**

indicates that a Conditional is made of the keyword **if**, followed by a Then_part_list (that is to say, a specimen of the Then_part_list construct), possibly followed by an Else_part — "possibly" because brackets indicate an optional component —, followed by the keyword **end**. More generally:

Aggregate production

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

Choice productions

A Choice production also lists a number of constructs, but with a different purpose: we want to state that a specimen of the left-hand side construct is a specimen of **one** — any one — of the listed constructs (rather than specimens of *all* the non-optional specimens, as in the aggregate case).

We separate the alternatives by vertical bars | to suggest exclusive choice. For example:

→ *Page 320.*

```
Type ≜ Class_or_tuple_type |
       Formal_generic_name |
       Anchored
```

This specifies that a Type is one of: a Class_or_tuple_type, a Formal_generic_name, an Anchored. More generally:

Choice production

A **choice** right side is of the form $C_1 \mid C_2 \mid ... \mid C_n$ ($n > 0$), 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.

Repetition productions

Use a Repetition production for a certain construct to express that its specimens are made of **any number** of specimens of some given construct, separated, if more than one, by a specified separator. The production will use braces, and three dots … to suggest repetition; it will also list either an asterisk* if "any number" means "zero or more", or a plus sign $^+$ to specify "one or more".

So a right side of the form {C § …}+ means "one or more specimens of C, to be separated by §", where § is a separator symbol or, more generally a construct. Replace + by * for "zero or more", allowing constructs with empty specimens.

For example:

→ *Page 473.*

Then_part_list ≜ {Then_part **elseif** …}+

means that a specimen of Then_part_list — representing the "then part" of a conditional instruction, as specified by the production for Conditional shown earlier as an example of aggregate — consists of one or more Then_part clauses, separated, if more than one, by the keyword **elseif**. So typical specimens of Then_part_list are of the forms

t1
t1 **elseif** *t2*
t1 **elseif** *t2* **elseif** *t3*

and so on, where *t1*, *t2*, *t3* … are specimens of Then_part.

If the production had used an asterisk * instead of a plus +, the empty text would also have been an acceptable specimen of Then_part_list.

More generally:

Repetition production, separator

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

{C § ...}*
{C § ...}+

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+

The last two cases are not common, since most repetitions involve a separator, but for those that don't the simpler notation suffices.

Using recursive productions

You will note that many productions appear to define constructs recursively (that is to say, in terms of themselves). For example the grammar includes the following three productions:

Instruction	≜	... Other choices ... | Conditional
Conditional	≜	**if** Then_part_list [Else_part] **end**
Else_part	≜	**else** Compound
Compound	≜	{Instruction ";" …} $^{+}$

which, taken together, show that Instruction is defined in terms of Conditional, defined in terms of Else_part, defined in terms of Compound, defined in terms of Instruction. This may seem strange (if you haven't seen syntax descriptions before), but in fact may make perfect sense.

Such recursive chains, to be useful, must always include a Choice production, with at least one branch leading to a construct entirely defined from terminals. Although the mathematical theory falls beyond the scope of this book, the general idea is that if the mutually recursive productions involving a construct A are

A	≜	T1 B T2
B	≜	A | T3

where T1, T2 and T3 are terminal constructs, then the possible specimens of A are of the form

t1 t3 t2
t1 t1 t3 t2 t2
t1 t1 t1 t3 t2 t2 t2
… and so on ...

where *t1* is a specimen of T1 etc.

Informally, you may view a set of mutually recursive productions as an equation N = T + A ∗ N, where N is the vector of non-terminals (A and B in the last example), T is a vector of terminals, A is a matrix of terminals, + is choice (the same as |), and ∗ is concatenation. Then the solution of the equation is N = T + A ∗ T + A^2 ∗ T + A^3 ∗T + ...

For a detailed discussion of the theory of fixpoints, which underlies these comments, see the book "Introduction to the Theory of Programming Languages".

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.

One production per non-terminal

The conventions of ensure a property mentioned earlier:

Basic syntax description rule

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

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).

We do *not*, for example, define

Type ≜ … Other Choices … | **like** Anchor

with the last Choice branch involving an Aggregate. Instead, we use two productions, one Choice and one Aggregate:

Type ≜ ... Other Choices ... | Anchored
Anchored ≜ **like** Anchor

Non-production syntax rules

BNF-E and other BNF variants only cover a certain category of grammatical structures, known as "context-free". Not all properties of interest are context-free; in addition some could in principle be described by context-free productions, but not easily.

To capture such properties we must use any applicable description technique, often just plain English (but as usual with much care and precision). We call such excursions from BNF "non-production" rules:

Non-production syntax rule

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

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.

2.6 REPRESENTING TERMINALS

As shown by the preceding examples, the right sides of productions often list some terminals. This raises a problem for reserved words, which might be mistaken for construct names — consider for example the keyword **class** and the construct Class — and special symbols, some of which, such as {, [and +, are also used as symbols of the syntax notation.

The following conventions remove any ambiguity.

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.

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". ← *Page 86.*

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. → *Pages 880, 1140.*

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.

As an example, here is the syntactic definition of the construct Compound, given by a repetition production. A specimen of Compound is formed of zero or more specimens of Instruction, separated by semicolons:

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

2.7 VALIDITY

The productions and other elements labeled SYNTAX, as described so far, specify the structure of constructs. In many cases, however, adherence to the structural requirements does not suffice to guarantee that a specimen of a construct will be meaningful.

For example, the following Assignment is built according to the syntactical specification of the corresponding construct:

x := *f*. *func* (*a* + *b*, *x*)

→ *The specification of* Assignment *is on page* 581. *The right side in this example is a specimen of qualified* Call, *whose syntax appears on page* 618.

But this does not mean that the Assignment will be acceptable in every possible context. It must also satisfy certain rules regarding the types of the components involved, the number of arguments passed to a routine such as *func* etc. Such supplementary requirements are called validity constraints

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.

Validity constraints come in addition to syntactic rules, and are in fact defined only for what the definition calls “syntactically well-formed specimens”. In the Assignment example, the validity constraint is:

Assignment rule *VBAR*

An Assignment is valid if and only if its source expression conforms to its target entity.

→ *See a full discussion of this constraint on page* 582.

(The “target entity” is the left side, *x* in the example; the “source expression” is the right side.)

Such validity constraints are introduced by the VALIDITY road sign as shown. Every constraint has a four-character code, here *VBAR*, uniquely identifying it. You do not need to pay any attention to these codes as you are first reading this book; but implementors of language processing tools, especially compilers, should include the appropriate code in any error message that reports a constraint violation. Then, if you get one of these error messages during system development, you will be able to look up the code in the index of this book, where they all appear under the heading “validity codes”, directing you to the detailed explanation of the language rule that you may have violated.

Some compilers, such as Eiffel Software’s EiffelStudio environment, give you the exact validity constraint, out of this book, as part of the error message.

The first letter of a validity code is always *V* (for "Validity"), the second one identifies the chapter, such as *B* for this chapter; the last two are a mnemonics for the constraint, for example *AR* for Assignment Rule.

A number of the validity rules have been reorganized from the previous editions. The appendix on changes gives the list of differences.

→ *"CHANGES IN VALIDITY CONSTRAINTS AND CONFORMANCE RULES"*,

Many constraints, such as the Feature Declaration rule, *VFFD*, list several conditions, each identified with a number. Error messages in this case should include not just the constraint code but also the number of the particular condition which was violated, for example *VFFD (2)*.

→ *Page 160*.

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.

2.8 INTERPRETING THE CONSTRAINTS

To avoid confusion, use the language properly, and benefit from the diagnostics of compilers and other tools, you must understand the precise nature of constraints and the conventions governing their interpretation.

Almost all the constraints listed in this book are *necessary and sufficient conditions*. This is not the usual style for other programming language descriptions, which commonly tell you that specimens of a certain construct *must* satisfy a certain property, or *may not* be of a certain form. Constraints in this book tell you instead that specimens of a certain construct will be valid **if and only if** they meet a specified set of requirements.

← *See "FORMALITY", page xvi*.

As discussed in the Preface, such a form is preferable, since it allows you not just to detect that certain specific components are *not* valid, but also to ascertain without doubt whether an arbitrary component *is* valid.

This style requires a general convention. When reading the Assignment rule, *VBAR*, used in the previous section to illustrate the notion of constraint, it may have struck you that the rule cannot possibly suffice to ensure the validity of the example assignment: what about the validity of the right side, *f.func (a + b, x)*, which must satisfy all the validity constraints on function calls (*func* a properly defined and exported function applicable to objects of *f*'s type, with exactly two formal arguments of types matching the actual arguments given)?

Spelling out all such conditions on the components of a construct would lead to needlessly complex and repetitive validity constraints. Instead, all validity discussions rely on a universal interpretation rule:

General Validity rule *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.

In the Assignment case, this means that constraint *VBAR* is considered to be automatically extended with the condition

"… and x satisfies all validity constraints on specimens of Variable*, and y satisfies all validity constraints on specimens of* Expression*"*

→ *The constraint on* Variable *is the Entity rule, page 505; for the constraints on* Expression *see chapter 28.*.

so that, for the example Assignment above, the Assignment rule implicitly requires that *f.func (a + b, x)* be a valid function call.

2.9 SEMANTICS

Lexical, syntactic and validity rules are only there to help us ensure that our software makes sense. The next question — even more important — is: what *is* that sense? The task of semantics is to answer that question.

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.

The "effect" may include executing actions, producing a value, or both. It is defined by a rule marked SEMANTICS. For specimens having subcomponents, the rule will recursively refer to the semantics of the subcomponents.

The definition of "semantics" above explicitly bassumes that the construct is syntactically legal and valid. When reading the SEMANTICS paragraphs, remember that they only apply to valid specimens. In many cases, the semantic rules would not even make sense otherwise; attempting to describe the effect of an invalid component is useless.

Most construct presentations will cover first syntax, then validity, then semantics. This is the expected order: first how to build language components of a certain kind; then what restrictions may exist on their parts; finally, what the result means. In a few cases, the semantics comes before the validity; such departures from the normal sequence occur when the best way to understand the reason for a constraint is to look first at the construct's effect in valid cases, and then find out what is required for that semantics to make sense. The change of order in such cases is, of course, only a pedagogical device; as everywhere else, the semantic specification is meaningless for invalid components.

2.10 CORRECTNESS

Validity is only a structural property; execution of valid Eiffel software may produce undesired results, or not terminate, or produce an exception that lead to failure.

→ *"Failure" is a technical term defined in the discussion of exceptions in chapter 26.*

The loop **from until** *False* **loop end** is a valid instruction, but, if executed, will never terminate.

*This loop has an empty initialization (**from**), an empty loop body (**loop**), and an exit condition that can never hold. See "LOOP", 17.7, page 486.*

Even for a valid component, then, we need a more advanced criterion: the component's ability to operate properly at run-time. This is called **correctness** and is a more elusive aim than validity, since it involves semantic properties.

Ascertaining the correctness of an executable software component requires two pieces of information: what the component does (its implementation), but also what it is expected to do (its specification, or **contract**). Eiffel supports both aspects: along with the executable elements of a class (the bodies of its routines, made of executable instructions), you may provide **assertions**, which state the contracts.

→ *Assertions, specifications and correctness are studied in chapter 9.*

A class will be said to be correct if its features are guaranteed to perform according to their contracts.

2.11 TWO-TIER DEFINITION AND UNFOLDED FORMS

A number of Eiffel mechanisms provide high-level idioms for program schemes that could also be addressed — less concisely, or less elegantly — through other constructs. As a simple example, for a Multi_branch instruction dealing with character constants, you may use character intervals rather than listing individual characters; instead of writing

→ *"MULTI-BRANCH CHOICE", 17.4, page 474.*

```
inspect
        char
when 'a', 'b', 'c', 'd', 'e' then                    [1]
        case1
else
        case2
end
```

you may replace the consecutive character choices by a single interval:

```
when 'a'..'e' then          -- The rest as above     [2]
```

The effect is the same but the text is simpler. We may call such language mechanisms *second-tier*, where the first tier would contain the constructs that cannot easily be expressed in terms of any others. The presence of second-tier constructs doesn't contradict the Eiffel language design principle that "the language should provide *one* good way to do anything useful", since the intention of the second-tier mechanisms is to provide a significantly better means of expression in applicable cases.

For such mechanisms the language definition often relies on the technique of **unfolded forms**. The idea is simply to define the properties of second-tier variants in terms of the more basic constructs; then it suffices to define the validity constraints, semantic specification, or both, for these basic forms. In the discussion of multi-branch instructions, the definition of "Unfolded form of a Multi_branch" reduces any variant of the instruction to one without intervals, so that the unfolded form of variant [2] above is [1]. After that, the validity and semantic definition for Multi_branch only address (through a number of intermediate definitions) the case of unfolded forms.

→ *Page 478.*

We will find unfolded forms useful for specifying the following constructs:

- Multiple declarations. → *Unfolded form definition: : page 158.*
- Inheritance parts, to ensure conformance of all types to *ANY*. → *Page 173.*
- Only clauses in postconditions. → *Page 239.*
- Assertions. → *Page 250.*
- Precursor. → *Page 299.*
- Anchored declarations. → *Page 337.*
- Formal generic parameters → *Page 345.*
- Tuples, through the notion of "anonymous class".
- Conversion → *Page 407.*
- Multi_branch choice instruction and associated Interval definitions. → *Pages 478 and 478.*
- Creators part of a class. → *Page 540.*
- Creation instruction. → *Page 544.*
- Assigner call. → *Page 592.*
- Non-object call. → *Pages 622.*
- Once routine in the case of a "fresh" call. → *Pages 637.*
- Operator expressions, which we unfold into steps: first through the "Parenthesized Form" to remove potential ambiguities thanks to operator precedence; then through the "Equivalent Dot Form" reduce every expression to a Call. → *Pages 760 and 771.*

2.12 THE CONTEXT OF EXECUTING SYSTEMS

As explained in the next chapter, the executable units of Eiffel software are called "systems" (although many people also use the more traditional term "program"). The following terminology will serve to discuss the context of system execution:

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.

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*".

2.13 TEXTUAL CONVENTIONS

Eiffel texts are written in "free format": the only purpose of separating them into lines and including extra "white space" (space or tab characters) in these lines is to improve the readability of class texts, according to the style rules of a later chapter.

→ *Style guidelines are the topic of appendix 34.*

A language processing tool treats any sequence of line separations, spaces and tabs between lexical elements of the language as a single "break", as explained in the chapter on the lexical structure. For such a tool the only relevant information is the presence of a break, not its precise makeup — for example its use of a line return rather than a space — which is interesting only for human readers.

→ *"BREAKS", 32.5, page 871*

Eiffel is **case-insensitive**:

→ *"Letter Case rule", page 876*

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.

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

Hence the definitions:

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.

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*.

These rules are detailed in the corresponding chapter. They are for the benefit of your fellow human readers; language processing tools such as compilers will ignore them, except if they include an option for enforcing style standards. Do apply these standards: one of the attractions of Eiffel is its readability; consistency of Eiffel style, from Saõ Paulo to Sakhalin, makes it even better.

→ *Chapter 34*.

Another convention that greatly facilitates the writing and maintenance of Eiffel systems is the optional nature of semicolons:

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.

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.