32 Lexical components

32.1 OVERVIEW

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.

This chapter defines the various kinds of lexical element.

The lexical structure of Eiffel is simple and predictable. For a first approach to Eiffel, the examples found in the rest of this book should provide enough models to enable you to write your own class texts without studying this chapter.

32.2 CHARACTER SETS

Every lexical component is a sequence of characters.

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*.

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.

32.3 CHARACTER CATEGORIES

The discussion will rely on a classification of characters into letters, digits and other categories:

DEFINITIONS

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:

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

2 • If the underlying character set is 8-bit extended ASCII, the characters of codes 192 to 255 in that set.

3 • If the underlying character set is Unicode, all characters defined as letters in that set.

An **alpha_betic character** is a letter or an underscore _.

A **numeric character** is one of the ten characters *0 1 2 3 4 5 6 7 8 9*.

An **alpha_numeric character** is alpha_betic or numeric.

A **printable character** is any of the characters listed as printable in the definition of the character set (Unicode or extended ASCII).

PURPOSE

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.

To avoid such headaches, switch to Unicode (and discover a few new headaches).

32.4 GENERAL FORMAT

At the lexical level, a class text is made of **tokens**, **breaks** and **comments**. Tokens are the meaningful components; breaks play a purely lexical role (separating tokens); comments add informal explanations for the benefit of human readers.

The next sections examine breaks, comments, layout conventions, the influence of letter case, and the various categories of token.

32.5 BREAKS

DEFINITIONS

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.

Some platforms do not support the concept of a New Line character, but represent texts as sequences of lines. On such a platform, you may apply the rules of this chapter by considering a text as made of the concatenation of all its lines, with a New Line character between consecutive lines.

Breaks separate successive tokens. Any break is as good as any other:

→ *As detailed below: "TEXT LAYOUT", 32.7, page 875.*

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.

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.

32.6 COMMENTS

PURPOSE

A class text may contain comments, which have no effect on the semantics of the classes in whose texts they appear, but provide explanations for the benefit of readers of these texts.

Any part of a line beginning with two consecutive hyphens -- and extending to the end of a line is a comment, as in

```
Some other text -- A comment
```

or, without any preceding text

```
-- A comment
```

Some comments are "expected", others "free":

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**.

Free comments appear in any position where you feel you should include some explanations for the reader of your software:

```
your_array.sort                -- Sorting algorithm must be stable.
```

Expected comments are more closely connected to the syntax structure. The three main examples are informal assertions, feature clause qualifiers and feature headers. As an example of the first, you may use a Comment (possibly with a Tag_mark) as an Assertion_clause expressing a property which you have not been able to write formally as a Boolean_expression; here is an example from *FIXED_QUEUE* in EiffelBase:

```
invariant
        0 < = first_index
        first_index < = last_index
       -- If queue is not empty, items are in positions
       -- first_index, first_index + 1, . . . , last_index –1 (mod capacity)
.       . . . More invariant clauses . . .
```

Feature comments introduce successive feature categories, each in a separate Feature_clause, as in

← From the sketch of LINKED_LIST on page 134.

```
class LINKED_LIST [T] inherit
        . . .
feature -- Access
            . . . Feature declarations . . .
feature -- Measurement
            . . . Feature declarations . . .
```

```
feature {LINKED_LIST} -- Implementation
            … Feature declarations …
      …
end
```

Finally the optional Header_comment of a feature appears after its signature and expresses concisely the purpose of the routine, as in:

```
convert_to_resolution (res_val: REAL)
            -- Convert to world coordinates,
            -- using res_val as resolution.
      ... Rest of Routine ommitted ...
```

If a comment is important, it is often advantageous to replace it by a **note** clause, which has an official place in the syntactic structure. The convention in this case is to use what as note tag:

```
convert_to_resolution (res_val: REAL)
            note
                  what: "[
                              Convert to world coordinates,
                              using res_val as resolution.
                        ]"
      ... Rest of Routine ommitted ...
```

The of Free and Expected comments is the same: a comment is made of one or more line segments, each beginning with two consecutive dash characters -- and extending to the end of the line

With an auxiliary definition

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.

the following syntax captures the form of comments:

Comments

$$\text{Comment} \triangleq \text{"}--\text{"} \; \{\text{Simple\_string Comment\_break} \ldots\}^*$$

$$\text{Comment\_break} \triangleq \text{New\_line [Blanks\_or\_tabs] "}--\text{"}$$

where Simple_string denotes sequences of characters without a new line.

This syntax implies that two or more successive comment lines, with nothing other than new lines to separate them, form a single comment.

For example, the text extract

```
c :=                     -- This is comment text
                         -- This is the first comment's continuation
      a + b              -- This is a second comment.

                         -- This is a third comment.
```

contains three-comments as indicated (with a blank line between the second and the third)..

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.

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.

The Header_comment of a routine is formally equivalent to the more explicit **note** form:

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.

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.

32.7 TEXT LAYOUT

An Eiffel text is a sequence; each of the elements of the sequence is a break, a comment or a token.

You may always insert a break between two elements without affecting the semantics of the text.

A break is not required between two adjacent elements if one is a comment and the other a token or another comment. Between two successive tokens, a break may be required or not depending on the nature of the tokens.

We may divide tokens into two categories:

DEFINITIONS

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**.

→ *For the list of symbols see below "SPECIAL SYMBOLS", 32.11, page 879.*

Then:

SYNTAX

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.

Without this rule, adjacent words not separated by a break — as in *ifxthen* — or adjacent symbols would be ambiguous.

Between adjacent words or adjacents symbols a break is required. For example, a break is needed between a keyword and an identifier (both of which are words); in

DEFINITION

```
if x then ...
```

the breaks both before and after *x* are required. But the assignment

```
c:=a+b
```

may be written without any break, although the standard style guidelines suggest using a one-blank break both around the assignment symbol **:=** and around every operator.

→ *Chapter 34.*

The syntax actually permits few cases of adjacent symbols; the most common is a prefix operator appearing after an infix operator, as in 3 + –5.

More generally, the physical layout of components should be so designed as to foster the readability of software texts. For example, indentation (using tab characters) highlights the structure of nested components. Since readability will benefit from consistency, this book introduces some recommended style conventions.

Appendix A.

32.8 LETTER CASE

The conventions on letter case were introduced at the beginning of this book. Here is the precise rule.

← *"TEXTUAL CONVENTIONS", 2.13, page 101.*

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.

In particular, letter case is not significant for identifiers and for reserved words; remember the notion of "same feature name", which ignores letter case.

← *"Same feature name, same operator, same alias", page 153*

→ *Appearing throughout the book and collected in chapter 34.*

This policy goes with a precise set of style guidelines enjoining you to use specific conventions for specific constructs, in particular identifiers (class names in all upper case, variable identifiers in all lower case etc.).

32.9 TOKEN CATEGORIES

Tokens are the basic meaningful elements of software texts.

As noted in the description of general conventions at the beginning of this book, tokens are specimens of **terminal constructs**. For example the token *8940* is a specimen of the terminal construct Integer. (In contrast, higher-level syntactical structures, such as class texts or routines, are specimens of non-terminal constructs such as Class or Routine.) Terminal constructs do not appear in left sides of the productions of the grammar; instead, their structure is defined in this chapter.

← *“THE LEXICAL LEVEL”, 2.4, page 87*.

There are two categories of tokens, fixed and variable:

- **Fixed tokens** have a single, frozen form. They include reserved words such as **class** or *Current*, and special symbols such as **:=**. For fixed tokens this book not distinguish between the form of a token and the underlying terminal construct. For example, **class** is the single specimen of a construct which could be called Class_keyword but remains implicit; an occurrence of the token in the grammar denotes the construct.
- **Variable tokens** are specimens of terminal constructs such as Integer, Identifier, Free_binary, for which this chapter defines a general structure, within which you can define tokens that fit the needs of your software. For example, the rules for Integer, given below, permit specimens made of one or more decimal digits; a token such as *327* satisfies this specification.

The following sections examine reserved words, special symbols, and the various terminal constructs defining variable tokens: Identifier, Integer, String, Simple_string, Real, Operator, Character.

32.10 RESERVED WORDS

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.

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.

---- MOVE AND REWRITE The first and simplest tokens are reserved words, listed in an appendix. Each is a sequence of letters, with in two cases — **and then**, **or else** — an intervening blank (normally just one, but we tolerate more). Formally:

→ *Appendix K.*

Reserved words are called that way because you may not choose them for your own identifiers. They include **keywords** and **predefined names**:

See below on identifiers.

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.

- Keywords, such as **class** and **feature**, introduce and delimit the various components of constructs.
- Predefined names come at positions where variable tokens would also be permissible: *Result*, denoting the result of a function, may appear in lieu of a local variable, for example as target of an assignment; *INTEGER* may appear at a position where a type is expected.

The definition of Result as a special kind of local variable appeared in 8.6, page 114..

In accordance with the Letter Case rule, letter case is not significant for reserved words, so that *CLASS*, *result* or even *rEsULt* are permissible forms. According to the style rules, however:

- Keywords appear in lower-case, as with **class**. In a typeset text they should always appear in **bold**, as in this book.
- Predefined names start with a capital letter; the rest is in lower case (as with *Result*), except for types since the general convention for all types is to use all upper-case (as with *INTEGER*). When typeset, they appear in italics.

The following general guidelines presided over the choice of reserved words and should help you to learn reserved words quickly and remember them without hesitation:

- Reserved words are simple and common English words. They are never abbreviations and, with one exception, they are never composite words.

  The exception is **elseif**, denoting a simple idea for which there is no one-word name in English.

- For simplicity and consistency, the grammatical form is always the shortest. For a noun it's the singular, even if the plural might seem more natural: the clause introducing the features of a class begins with the keyword **feature**. For a verb it's in the infinitive form, as in **require**, again without an "s".

32.11 SPECIAL SYMBOLS

A small number of one- and two-character strings, called special symbols, have a special role in the syntax of various constructs.

Earlier chapters have introduced these symbols in connection with the syntactic form of various constructs. Here is the complete list:

Special symbol

A **special symbol** is any of the following character sequences:

-- : ; , ? ! ' " $. –> :=

= /= ~ /~ ( ) (| |) [ ] { }

The following table gives a reminder of their role and the page where the corresponding syntax productions appear.

Symbol	**Name**	**Role**	**Pages**
--	Double dash	Introduces comments.	873
;	Semicolon	Separates instructions, declarations, assertion clauses…; always optional.	
,	Comma	Separates elements in lists of of entities or expressions.	
:	Colon	Separates the Type_mark in a declaration, a Tag_mark in an Assertion_clause, and a Note_name term in a Notes clause.	
'	Single quote	Encloses manifest constants.	
"	Double quote	Encloses manifest strings.	
%	Percent	Introduces special character codes.	
/	Slash	In a special character code, introduces a character through its code.	
+ –	Plus and minus	Signs of integer and real constants. (Also permitted as prefix and infix operators, appearing in a separate table.)	778
$	Dollar	Address operator for passing the address of an Eiffel feature or expression to a routine (usually external).	823
%	Percent	Introduces a special character code.	
/	Slash	In a special character, introduces a character by its numerical code.	
.	Dot	Separates target from feature in a feature call or creation call. Separates integer from fractional part in a real number.	
–>	Arrow	Introduces the constraint of a constrained formal generic parameter.	
:=	Receives	Assignment operator.	
= /=	Equal, not-equal signs	Equality and non-equality operators.	
~ /~	Tilde, slash-tilde	Object equality and non-equality operators.	
( )	Parentheses	Group subexpressions in operator expressions; enclose formal and actual arguments of routines.	
(| |)	Target parentheses	Enclose a constant or non-atomic expression used as target of a call in dot or bracked notation.	
[ ]	Brackets	Enclose formal and actual generic parameters to classes; enclose items of a manifest tuple; specify that a feature has a Bracket alias.	
{ }	Braces	Enclose types in various contexts: Clients part, Feature_clause or New_export_list, Creation_type.	

The special symbols must be written as given in the above table, with no intervening blanks or other characters. They should be typeset in roman.

32.12 IDENTIFIERS

So much for fixed tokens; on to variable tokens.

An important category of variable tokens is identifiers, describing symbolic names which class texts use to denote various components such as classes, features or entities.

Here are some example identifiers:

A
LINKED_LIST
a
an_identifier
feature_1

The construct is defined as follows:

Syntax (non-production): Identifier

An Identifier is a sequence of one or more alpha_numeric characters of which the first is a letter.

According to the earlier definitions of character categories this implies that an identifier must start with a letter (not a digit or an underscore) and continue with zero or more letters, digits and underscores.

← *"Letter, alpha_betic, numeric, alpha_numeric, printable", page 870*.

Identifiers are subject to a number of restrictions to avoid ambiguity; see for example the Entity rule, which prevents you from using the same identifier to declare two entities within a given scope. But these are rules on higher-level constructs, such as Entity, relying on identifiers. For themselves, identifiers are only subject to a basic validity constraint on the choice of name:

← *"VEEN", page 505*.

Identifier rule *VIID*

An Identifier is valid if and only if it is not one of the language's reserved words.

Two of the reserved words, **and then** and **or else**, include a blank and would not be lexically acceptable as identifiers anyway. Their components — **and**, **or**, **then**, **else** — are themselves all reserved.

There is no limit to the length of identifiers, and all characters are significant: to determine whether two identifiers are the same or not, you must take all their characters — but not letter case — into account.

← *"Same feature name, same operator, same alias", page 153*; *"Letter Case rule", page*

32.13 OPERATORS

Operators appear as **alias** for identifier features when you wish to let your clients call the feature in infix or prefix form, as with

```
no_better_than alias "<=" (other INVESTMENT): BOOLEAN
            -- Is other a least as good as current invesment?
      ... Rest of function declaration omitted ...
```

which, can then be called under the form *inv1* <= *inv2* as well as the standard dot-notation call *inv1*.*no_better_than* (*inv2*).

← “OPERATOR FEATURES”, 5.15, page 154.

There are three kinds of operators: **predefined**, **standard** and **free**.

There are only four *predefined* operators:

Predefined operator

A **predefined operator** is one of:

= /= ~ /~

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.

The *standard* operators, used as aliases for features of the basic types (*INTEGER* etc.). The list given in the discussion of features includes boolean operators (such as **not**, **implies**, **and**, **or**) which lexically are keywords, leaving the following as standard operators in the lexical sense:

← *Page* 154.

Standard operator

A **standard unary operator** is one of:

+ –

A **standard binary operator** is any one of the following one- or two-character symbols:

+ – * / ^ < >

<= >= // \\ ..

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.

The above *no_better* example illustrated such a use as Alias.

Free operators allow you to make up your own operators, for example to support specific notations in mathematics or physics. You can use almost any combination of "operator symbols", a notion defined very broadly:

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 **!**.

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.

Thanks to this notion, the definition of free operators lets you make up your own notations as long as they cause no ambiguity:

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.

← From the table in "SPECIALSYMBOLS", 32.11, page 879.

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.

Remember that — style guidelines aside — we do **not** want to require breaks between an operator such as **+** and the following identifier such as *b*, so that we can interpret *a*+*b* as an expression.

← *"Syntax (non-production): Break rule", page 875.*

The last two parts of the definition separate "free" unary and binary operators from the standard ones. They allow, for example, defining ∗ as unary operator in one of your classes — whether or not it also uses it as binary operator — even though, in the basic types *INTEGER*, *REAL* and their sized variants, it figures only as binary.

This rule still leaves you considerable room in choosing free operators to match the needs of just about any application domain, as illustrated by other examples: ∗∗, |–|, <–>, –|–>, **=>** and many others.

General-purpose libraries such as EiffelBase and EiffelVision make very limited use of free operators; this facility is mostly for specialized application domains that are accustomed to their own notations.

32.14 CHARACTERS

PURPOSE

Characters — specimens of construct Character — are used in various constructs: Character_constant (of which a specimen is a Character in single quotes, as '*A*'); Manifest_string (zero or more characters in double quotes, as in "*ABC DE*!#$"); Identifier.

A character is an element of the character set as defined at the beginning of this chapter: either Unicode or extended ASCII. To define the notion properly let us assume that the device used to enter software texts is a keyboard, offering its users a number of keys, each defined by a code. We must distinguish between *keys*, which simply serve to enter certain codes, and *characters*, the atoms of Eiffel lexical elements.

In the simplest and most common case, of course, you enter a character just by pressing an associated key. For certain characters, however, you may have to press a succession of two or more keys; and some keys do not yield a character at all. For example the Manifest_string in

```
String_with_backspace: INTEGER is "AAA%BZZZ"
```

includes a non-printable character, Backspace, appearing as **%B**. For this character there is in fact a key on usual keyboards, but pressing it does not yield a character: it simply erases the previous character you have entered. To make Backspace part of your string, you may represent it by the two-key sequence **%B**, as here. Another possibility, using the numerical code for this character, is to enter it as **%/8/**.

The definition of "character" must be general enough to encompass all such cases and address portability problems raised by the different in keyboards found in various countries:

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.

Appearing on the next page.

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.

→Introduced later in this chapter: "INTEGERS", 32.16, page 889.

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).

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

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.

The major application of forms 2 and 3 is to express a Manifest_string containing characters that you cannot type directly into the class text. This includes non-printable characters, such as Backspace (see *String_with_backspace* above), and others not supported on all keyboards. If you have an American ASCII keyboard but want to define a Manifest_string that output devices supporting the appropriate codes will display as *ambiguïté*, with two letters bearing diacritical marks, you may enter it as the string "*ambigu*%/139/*t*%/130/".

The codes are also useful in defining character intervals for Inspect instructions, as in

```
inspect
        entry                      -- Of type CHARACTE
when %/128/  . .  %/165/ then
        >…
… Other clauses …
end
```

Since this convention is of no particular benefit for entering such tokens as identifiers, it is reserved for character and string constants:

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.

The characters "of" such a constant do not include the single ' or double " quotes, which you must enter as themselves.
The use of the percent character

The semantic specification follows the cases of the syntax definition:

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.

32.15 STRINGS

Do not confuse String or Simple_string with Manifest_string, seen in the discussion of expressions. A specimen of Manifest_string, a non-terminal construct, is a Simple_string in double quotes, as in "*SOME STRING*".

← *29.8, page 784.*

In the definition of String, a "character" is any legal Eiffel character as defined in the preceding sections. This includes in particular:

See 32.14, page 884, for the various forms of characters and the use of the percent sign.

• A keyboard key other than %.

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).

- A special character code such as **%B**
- A character given by its numerical code, such as **%**/59/ or **%**/**0b**3B/.

The semantics is straightforward:

String semantics

The value of a String or Simple_string is the sequence of the values of its characters.

32.16 INTEGERS

Integer, a lexical construct, describes unsigned integer values. You may express an Integer in either the usual decimal notation or in one of three bases other than 10: binary (base 2), octal (base 8), hexadecimal (base 16).

←Do not confuse Integer with Integer_constant, seen s in “INTEGER CONSTANTS”, 29.5, page 782. A specimen of Integer_constant, a non-terminal construct, is an Integer optionally preceded by a sign.

Examples of the most common case, decimal notation are:

```
0
327
3197865
3_197_865
```

Except for underscores, no intervening characters (such as blanks) are permitted between digits.

You can use underscores to improve readability by dividing a long integer into pieces. The recommended convention, as in the last example is to use groups of three digits from the right (so that the leftmost one may be shorter). Underscores have no effect on the value: the last two examples denote the same integer value.

Examples in non-decimal notation, all representing the number twenty-nine, are:

```
0b11101                  -- Binary
0c35                     -- Octal
0x1D                     -- Hexadecimal
```

The convention is clear: the constant starts with the digit **0**, followed by a code for the base (**b** for Binary, **c** for oCtal, the conventional **x** for heXadecimal), followed by digits meaningful for the appropriate base.

Here too underscores may be used to group digits, as in **0b**1_1101, with no particular style guideline.

As with other lexical elements, letter case is not significant for the Integer_base (so that **0X**, for example, is acceptable in lieu of **0x**) and for the hexadecimal digits **A** to **F**. The forms given, such as **0x**1D, indicate the recommended style: lower case for the base and upper case for the digits.

← *“Syntax (non-production): Double Reserved Word rule”, page 879*.

To describe this structure it is best to resort to a syntax production and a validity rule, with the understanding that unlike with ordinary syntax productions (and as indicated in the first clause of the validity rule) the successive elements are not separated by breaks. The syntax is

Integers

Integer	≜	[Integer_base] Digit_sequence
Integer_base	≜	"**0**" Integer_base_letter
Integer_base_letter	≜	"**b**" | "**c**" | "**x**" | "**B**" | "**C**" | "**X**"
Digit_sequence	≜	Digit$^+$
Digit	≜	"**0**" | "**1**" | "**2**" | "**3**" | "**4**" | "**5**" | "**6**" | "**7**" | "**8**" | "**9**" | "**a**" | "**b**" | "**c**" | "**d**" | "**e**" | "**f**" | "**A**" | "**B**" | "**C**" | "**D**" | "**E**" | "**F**" | "**_**"

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**.

The associated constraint is:

Integer rule *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.

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.

← *“INTEGER CONSTANTS”, 29.5, page 782*.

Finally, the 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.

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.

32.17 REAL NUMBERS

Real numbers – specimens of construct Real – define the manifest constants for the basic types *REAL* and its sized variants.

The following are real numbers:

1.0	1.
0.1	.1
2345.632E-7	2345.632e-7

Here is the definition:

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.

As with Integer*, there is no sign; only a* Real_constant *may introduce the sign.*

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.

The integral part, fractional part and exponent, all specimens of Integer, must be expressed in decimal (no binary, octal or hexa).

The denoted value is the expected one:

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).

As with integers, this semantics yields the *exact* mathematical value. Approximation to the supported floating-point type — *REAL*, *REAL_32* or *REAL_64* — occurs only when you use the Real as a Real_constant.

←*“REALCONSTANTS”, 29.6, page 782*.