16

Control structures

16.1 OVERVIEW

The previous chapters have described the "bones" of Eiffel software: the module and type structure of systems. With this chapter 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 five of them: sequencing, null instruction, conditional, multi-branch choice and loop. A complementary construct is the Debug instruction.

The semantic specifications given in this chapter assume that none of the instructions executed as part of a control structure triggers an exception. If an exception occurs, the normal flow of control is interrupted, as described by the chapter that specifically deals with exception handling.

→ *Chapter 17.*

16.2 COMPOUND

The first control structure, Compound, enables you to specify a list of instructions to be executed in a specified order.

From its inconspicuous syntax, you wouldn't guess that this is a fundamental program composition mechanism: the instructions of a Compound are just written one after another, in the order of their intended execution. You may emphasize the sequencing of the instructions by using a separator, the semicolon, which is not only discreet but optional to boot.

A typical specimen of the Compound construct is:

```
window1.display
mouse.wait_for_click (middle)
if not last_event.is_null then
        last_event.handle; screen.refresh
end
```

This Compound is made of three instructions; it specifies the execution of these instructions in the order given. The last of the three (a Conditional instruction, as studied below) itself includes a two-instruction Compound.

The use and non-use of semicolons in this example illustrate the recommended style convention: no semicolon has been included between the three instructions of the outermost Compound since they appear on separate lines (the most common case), enough to remove any confusion. The two instructions of the innermost Compound — inside the Conditional — appear on the same line; here the semicolon should be included for the benefit of the human reader, even though compilers don't need it.

The syntax for Compound is:

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

This is complemented by a special syntactical convention reinforcing the semicolon's natural shyness:

Other constructs where the semicolon separator is optional: Parent *clauses*, 6.3, page 103; *feature declarations*, 5.7, page 77; *local entity declarations*, 8.7, page 149; *assertion clauses*, 9.5, page 156.

Semicolon rule for instructions

The semicolon used as a separator between two instructions is **optional** except if the second begins with an opening parenthesis.

The exception corresponds to a case which could cause ambiguity; an example is

```
target := source ; (parenthesized).routine
```

Omitting the semicolon here could confuse a compiler with limited look-ahead capabilities, or a hurried human reader, into parsing the right-hand side of the initial assignment as *source* (*parenthesized*). To avoid such confusion the semicolon is required.

In the common, non-confusing case, the style rule is to **omit the semicolons** between instructions appearing on separate lines. The semicolon in that case is just visual noise and actually hampers readability. For successive instructions on the same line make sure to **keep** the semicolon. The above example illustrated this style rule, observed throughout this book.

→ *"OPTIONAL SEMICOLONS", 31.10, page 721.*

All this does not diminish the role of sequencing as a control structure, even if the only syntactical trace left in the software text is the textual order of instructions, indicating the temporal order in which they should be executed at run time.

Aside from its role as a control structure, the Compound construct serves an frequent syntactical need : allowing any construct that involves an instruction — so that it may execute it as part of its own execution — to involve *any number* of instructions, including zero. The syntax of Eiffel consistently adheres to this rule: Instruction never appears in the definition of a construct other than Compound; other construct definitions use Compound instead. They includes:

- The body of a non-deferred routine (construct Internal). *← Syntax on page 148.*
- The Then_part and Else_part of a Conditional instruction. *→ Page 357.*
- The When_part and Else_part of a Multi_branch instruction. *→ Page 357.*
- The Initialization and Loop_body of a Loop instruction. *→ Page 370.*
- The Debug instruction. *→ Page 372.*
- The Rescue clause of a non-deferred routine. *→ Page 385.*

The effect of executing a Compound is defined as follows:

- If the Compound has zero instructions, the effect is to leave the state of the computation unchanged.
- If the Compound has one or more instructions, its effect is that of executing the first instruction of the Compound and then (recursively) to execute the Compound obtained by removing the first instruction.

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.

16.3 NULL INSTRUCTION

Even though Eiffel texts tend to make little use of the semicolon as instruction separator, a programmer coming from other languages might distractedly write

```
if c then; il ;; i2; else i3 ;;; end
```

with extra semicolons whose bad style is not really a reason for rejection, as they seem harmless in the end. To avoid bothering programmers we assume a null instruction, whose form may be illustrated as follows

The effect of a null instruction is to leave the state of the computation unchanged.

This specification indicates that we can understand a null instruction as a Compound, with zero instructions. The instruction doesn't explicitly appear in the grammar (because it would cause syntactical ambiguity with the case of an empty compound), but we accept its silent presence as a token of our support to recovering semicolon addicts.

16.4 CONDITIONAL

A Conditional instruction prescribes execution of one among a number of possible compounds, the choice being made through boolean conditions associated with each compound.

You should remain alert to an important aspect of the Eiffel method, which de-emphasizes explicit programmed choices between a fixed set of alternatives, in favor of automatic selection at run-time based on the type of the objects to which an operation may be applied. Such an automatic selection is achived by the object-oriented techniques of inheritance and dynamic binding. This methodological guideline, discussed in more detail below, does not diminish the usefulness of Conditional instructions — a widely used mechanism —but should make you wary of complicated decision structures with too many **elseif** branches. This applies even more to the Multi_branch instruction studied next.

→ *"A NOTE ON SELECTION INSTRUCTIONS", 16.6, page 366*.

An example Conditional is

```
if x > 0 then
        il; i2
elseif x = 0 then
        i3
else
        i4; i5; i6
end
```

whose execution is one among the following: execution of the compound *i1; i2* if *x > 0* evaluates to true; execution of *i3* if the first condition does not hold and *x = 0* evaluates to true; execution of *i4; i5; i6* if none of the previous two conditions holds.

There may be zero or more "**elseif** Compound" clauses. The "**else** Compound" clause is optional; if it is absent, no instruction will be executed when all boolean conditions are false.

The general form of the construct is

$$
\begin{aligned}
\text{Conditional} &\triangleq \textbf{if}\ \text{Then\_part\_list}\ [\text{Else\_part}]\ \textbf{end} \\
\text{Then\_part\_list} &\triangleq \{\text{Then\_part}\ \textbf{elseif}\ \ldots\}^{+} \\
\text{Then\_part} &\triangleq \text{Boolean\_expression}\ \textbf{then}\ \text{Compound} \\
\text{Else\_part} &\triangleq \textbf{else}\ \text{Compound}
\end{aligned}
$$

To define precisely the semantics of this construct, a few auxiliary notions are useful. As the syntax specification shows, a Conditional begins with

if $condition_1$ **then** $compound_1$

where $condition_1$ is a boolean expression and $compound_1$ is a Compound. The remaining part may optionally begin with **elseif**. If so, we may consider that it forms a new, simpler Conditional, called its *secondary part*:

Secondary part

The **secondary part** of a Conditional possessing an Else_part is the Conditional obtained by removing the initial "**if** Then_part_list" and replacing the first **elseif** of the remainder, if any, by **if**.

The secondary part of the above example Conditional is

```
if x=0 then
        i3
else
        i4; i5; i6
end
```

The following notion helps define the semantics of conditionals:

Prevailing immediately

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

The final part of a Conditional, also optional, is of the form **else** $compound_n$. For consistency we may consider that it is always present by considering an empty Else_part to stand for one with an empty Compound.

With these conventions, the effect of a Conditional may be defined as follows. If the Conditional prevails immediately, then its effect is that of its *compound*$_1$ part. Otherwise:

← *As defined above in "COMPOUND", 16.2, page 353*

- If it has a secondary part, the effect of the entire Conditional is (recursively) the effect of the secondary part.
- If it has no secondary part, its effect is that of the (possibly empty) Compound in its Else part.

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.

16.5 MULTI-BRANCH CHOICE

Like the conditional, the Multi_branch supports a selection between a number of possible instructions. In contrast with the Conditional, however, the order in which the branches are written does not influence the effect of the instruction. Indeed, the validity constraints seen below guarantee that at most one of the selecting conditions may evaluate to true.

Like the Conditional, the Multi_branch instruction is less commonly used in proper Eiffel style than its counterparts in traditional design and programming languages. This is explained in more detail below.

→ *"A NOTE ON SELECTION INSTRUCTIONS", 16.6, page 366.*

You may use a Multi_branch if the conditions are all of the form

"Is *exp* equal to v_i ?"

or all of the form

"Is *exp* of type T_i ?"

where *exp* is an expression, the same for every branch, the v_i are constant values, different for each branch and (in the second variant) the T_i are all distinct types, not conforming to one another. In such cases, the Multi_branch provides a more compact notation than the Conditional, and makes a more efficient implementation possible.

Here is an example of the first kind, assuming an entity *last_input* of type *CHARACTER*:

```
inspect
      last_input
when 'a' .. 'z', 'A' .. 'Z', '_' then
      command_table.item (upper(last_input)).execute
      screen.refresh
when '0' .. '9' then
      history.item (last_input).display
when Control_L then
      screen.refresh
when Control_C, Control_Q then
      confirmation.ask
      if confirmation.ok then
            cleanup; exit
      end
else
      display_proper_usage
end
```

Depending on the value of *last_input*, this instruction selects and executes one Compound among five possible ones. It selects the first (*command_table*…) if *last_input* is a lower-case or upper-case letter, that is to say, belongs to one of the two intervals '*a*' .. '*z*' and '*A*' .. '*Z*', or is an underscore '_'. It selects the second if *last_input* is a digit. It selects the third (refresh the screen) for the character *Control_L*, and the fourth (exit after confirmation) for either one of two other control characters; here *Control_L*, *Control_C* and *Control_Q* must be constant attributes. In all other cases, the instruction executes the fifth compound given (*display_proper_usage*).

This example discriminates on the value of an expression of type *CHARACTER*. Other permitted types include *INTEGER* and *STRING*. You may also discriminate on the basis of the *type* of the run-time object attached to the value of an arbitrary expression, as illustrated by the following example of dealing with various kinds of exception objects:

```
inspect
        last_exception
when {DEVELOPER_EXCEPTION} then
        process_developer_exception
when {OS_SIGNAL}, {NO_MORE_MEMORY} then
        cancel_operation
else
        reset
end
```

In this form the “inspect values” — the values listed in the **when** parts — are type descriptors, each listing a type in braces, as *{OS_SIGNAL}*. The instruction examines the type of the object associated with *last_exception*, if any, and if it conforms to one of the types listed executes the corresponding **then** branch; otherwise (including if *last_exception* is void) the instruction executes its **else** branch. The validity rule requires that none of the types listed conform to another, so there can be no ambiguity as to which branch will be executed.

The expression that determines the choice — *last_input* and *last_exception* in these two examples — has a name:

Inspect expression

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

The inspect expressions of the last two examples are last_choice and *last_exception*. Except when the choices are type descriptors (such as *{DEVELOPER_EXCEPTION}* in the last example), the inspect expression may only be of type *CHARACTER*, *INTEGER*, *STRING*.

The instruction includes one or more When_part, each giving a list of one or more Choice, separated by commas, and a Compound to be executed when the value of the inspect expression is one of the given Choice values.

Every Choice specifies zero or more inspect values. More precisely, a Choice is either a single constant (Manifest_constant or constant attribute) or an interval of consecutive constants yielding all the interval’s elements as inspect values. If present, the instruction’s optional Else_part is executed when the inspect expression is not equal to any of the inspect values.

As the validity constraint will state precisely, all the inspect values must either all be type descriptors, of the form *{SOME_TYPE}*, or all be of the same type as the inspect expression: all characters, all integers, or all strings. They must all be different to avoid ambiguity and ensure that the order of the When_part branches has no influence on the semantics of the construct.

Every constant in the preceding examples is either a Type_descriptor, a Manifest_constant such as '*a*' whose value is an immediate consequence of the way it is written, or a constant attribute such as *Control_L* whose value is given in a constant attribute declaration such as

```
Control_L: CHARACTER is '%/217/'
```

→ *On character codes such as '%/217/' see 29.15, page 701.*

In other cases, you may just want to use a set of integer constants to distinguish between several possibilities, for example possible marital statuses (single, married etc.), without any need to know their actual values as long as they are guaranteed to be different. **Unique attributes** serve this purpose, and may be used in Multi_branch instructions. A typical declaration of Unique attributes is

→ *"UNIQUE ATTRIBUTES", 18.6, page 395.*

```
Single, Married, Divorced, Widowed: INTEGER is unique
```

which yields four integer constants. This guarantees that these constants' values are all different, all positive, and, for Unique constants introduced in the same declaration as here, consecutive. That's all you can rely on: the language definition provides no other guarantee on the values.

A Multi_branch may use such Unique constants as inspect values:

```
inspect
        n
when Single, Divorced then
        …
when Married then
        …
when Widowed then
        …
end
```

If any Unique constants are involved, the validity constraint will guarantee that all values are different in this case too. In particular if you mix Unique and non-unique integer constants, the non-unique ones must all be zero or negative to ensure that there is no conflict.

Now the formal rules. First, the syntax of Multi_branch:

```
     Multi_branch ≜ inspect Expression
                    [When_part_list] [Else_part] end
   When_part_list ≜ when {When_part when ...}+
        When_part ≜ Choices then Compound
          Choices ≜ {Choice "," ...}*
           Choice ≜ Constant | Interval | Type_descriptor
  Type_descriptor ≜ "{" Type "}"
         Interval ≜ Integer_interval | Character_interval |
                    String_interval
 Integer_interval ≜ Integer_constant ".." Integer_constant
Character_interval≜ Character_constant ".." Character_constant
  String_interval ≜ Manifest_string ".." Manifest_string
```

Construct Constant *describes manifest or symbolic constants, including* Unique *constants, and is studied in "GENERAL FORM OF CONSTANTS", 27.2, page 613*.

To discuss the constraint and the semantics, it is convenient to consider the *unfolded form* of the instruction.

Unfolded form of a multi-branch

To obtain the **unfolded form** of a Multi_branch instruction, replace any Interval, in the Choices of a When_part, by a Choices list made up of all constants between the interval's bounds, or empty if the second bound is smaller than the first.

Use integer order for an Integer_interval, character code order for a Character_interval, and lexicographical order based on character order for a String_interval.

So of the intervals

```
3 .. 5
'i' .. 'n'
"ab" .. "ad"
5 .. 3
```

the first two unfold into

```
3, 4, 5
'i', 'j', 'k', 'l', 'm' 'n'
```

the third into the (infinite) set of strings lexicographically between "*ab*" and "*ad*", and the last into an empty Choices list. Thanks to unfolding, the constraint and semantics may limit themselves to the case of Multi_branch instructions where every Choice is a Constant or Type_descriptor.

This definition also enables us to say exactly what “inspect values” means:

Inspect values of a multi-branch

The **inspect values** of the instruction are all the values listed in the Choices parts of the instruction’s unfolded form.

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.

A Multi_branch must satisfy a validity constraint:

Multi-branch rule *COMB*

A Multi_branch instruction is valid if and only if its unfolded form satisfies the following conditions.

1 • inspect values are either all of the Type_descriptor kind, or all constants of the same type, which must be one of: *INTEGER*, *CHARACTER*, *STRING*.

2 • If the inspect values are not of the Type_descriptor kind, the inspect expression is of the same type as the inspect values.

3 • Any two non-unique inspect values are different.

4 • Any two Unique inspect values have different names.

5 • If any inspect value is Unique, then every other one in the instruction is either Unique or has a negative or zero value.

6 • Any two inspect values of the Type_descriptor kind do not conform, either way, to each other.

Clauses 3 to 5 guarantee that values in different branches are different, with 4 and 5 specifically devoted to ensuring this property for a Multi_branch instruction that constains one or more Unique constants, possibly in conjunction with non-unique ones (which must then be of type *INTEGER*).

For inspect values of the Type_descriptor kind, such as *{SOME_TYPE}*, clause 6 — which implies 3 in this case — requires that none of the types listed conform to another. It rules out examples such as

```
inspect
      last_exception
when {YOUR_DEVELOPER_EXCEPTION} then
      "Something"
when {DEVELOPER_EXCEPTION} then
      "Something else"
end
```

WARNING: invalid with the assumed inheritance link.

where the class *YOUR_DEVELOPER_EXCEPTION* inherits from *DEVELOPER_EXCEPTION*. This may appear too strong a constraint until you realize that giving non-ambiguous semantics to such examples would require that we take into account the order of the When_part clauses: the rule, presumably, would be to select the first one that matches. This conflicts with the principle stating that the semantics of a Multi_branch should never depend on the order of the **when** clauses.

If you do want type-based discrimination with more than one possibly matching type, nest Multi_branch instructions, or use a Conditional.

To define the semantics of a Multi_branch instruction, we will use the concept of matching branch:

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 conditions for the value *val* of its inspect expression:

1 • *equal* (*val*, *i*) holds where *i* is one of the (non-Type_descriptor) inspect values listed in *wp*.

2 • *val* is attached to an object whose type appears in a Type_descriptor listed among the choices of *wp*.

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

Case 1 applies to a Multi_branch that lists actual inspect values: integers (including Unique), characters or strings. The matching criterion is equality in the sense of *equal*. Strings, in particular, will be compared according to the function *is_equal* of class *STRING*.

→ “OBJECT EQUALITY”, 21.5, page 457

Case 2 covers a Multi_branch that discriminates on the type of an object attached to the value of an expression. Note that a void value will never have a matching branch.

The specification of a Multi_branch’s effect follows directly from this definition. If there is a matching branch, the effect of the Multi_branch is that of the Compound following the **then** in that branch. Otherwise:

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 an exception is triggered and the current routine execution fails.

In case 2, the exception object is of type *BAD_INSPECT_VALUE*.

→ See 17.11, page 388, about exception objects.

Note the difference between Conditional and Multi_branch regarding the meaning of an absent Else_part when none of the selection conditions holds:

- A Conditional just amounts to a null instruction in this case
- Multi_branch will **fail**, triggering an exception.

The reason is a difference in the nature of the instructions. A Conditional tries a number of possibilities in sequence until it finds one that holds. A Multi_branch selects a Compound by comparing the value of an expression with a fixed set of constants; the Else_branch, if present, catches any other values.

If you expect such values to occur and want them to produce a null effect, you should use an Else_part with an empty Compound. By writing a Multi_branch without an Else_part, you state that you do *not* expect the expression ever to take on a value not covered by the inspect values. If your expectations prove wrong, the effect is to trigger an exception — not to smile, do nothing, and pretend that everything is proceeding according to plan.

16.6 A NOTE ON SELECTION INSTRUCTIONS

If you have accumulated some experience with some of the traditional design or programming languages, many of which include a "case" or "switch" instruction, you will recognize the Multi_branch as similar in syntax and semantics. But when it comes to writing Eiffel applications, you should be careful to not misuse this instruction. This warning extends to Conditional instructions with many branches.

Staying away from explicit discrimination is an important part of the Eiffel approach to software construction. When a system needs to execute one of several possible actions, the appropriate technique is usually not an explicit test for all cases, as with Multi_branch or Conditional, but a more flexible inheritance-based mechanism: **dynamic binding**. With explicit tests, every discriminating software element must list all the available choices — a dangerous practice since the evolution of a software project inevitably causes choices to be added or removed. Dynamic binding avoids this pitfall.

→ *“DYNAMIC BINDING”, 23.9, page 506*.

You should reserve Multi_branch instructions, then, to simple situations where a single operation depends on a fixed set of well-understood choices.

When the purpose is to apply a different operation to an object depending on its type (for example categories of employees, for which a certain operation, such as paying the salary, has a different effect), then Multi_branch is not appropriate: instead, you should define different classes that inherit from a common ancestor — for example *MANAGER*, *ENGINEER* etc. all inheriting from *EMPLOYEE* — and redefine one or more features (such as *pay_salary*) to take care of the local context. Then dynamic binding guarantees application of the proper variant: the call

```
Caroline.pay_salary
```

will automatically use the variant of *pay_salary* adapted to the exact type of the object attached to *Caroline* at run time (which may be an instance of *MANAGER*, or *ENGINEER* etc.).

This is more flexible than a Conditional or Multi_branch that lists the choices explicitly, especially if other operations besides *pay_salary* have variants for the given categories. To add a variant, it suffices to write a new class, say *INTERN*, as a descendant *EMPLOYEE*, equipped with new versions of the operations that differ from the default *EMPLOYEE* version. Unlike a system that makes explicit choices through Conditional or Multi_branch instructions, a system built with this method will only have to undergo minimal change for such an extension.

Explicit choices do have a role, as illustrated by the earlier examples of Multi_branch. The first read

```
inspect
        last_input
when 'a' .. 'z', 'A' .. 'Z', '_' then
        command_table.item (upper(last_input)).execute
        screen.refresh
when '0' .. '9' then
        history.item (last_input).display
when Control_L then
        screen.refresh
when Control_C, Control_Q then
        confirmation.ask
        if confirmation.ok then
                cleanup; exit
        end
else
        display_proper_usage
end
```

This decodes a user input consisting of a single character and executes an action depending on that character, What is interesting is that the Multi_branch does only the “easy” part: separating the major categories of characters (letters, digits, control characters).

In the branches for letters and characters, however, the finer choice is made not through explicit instructions but through dynamic binding. For example, letters are used to index a table *command_table* of objects representing command objects with operations such as *execute*. (These objects might be agents as studied in a later chapter.) After retrieving the command object associated with the upper-case version of a given letter, the above Multi_branch applies *execute* to it, relying on dynamic binding to ensure that the proper action will be selected.

→ Agents are the topic of chapter 25.

Using a Multi_branch to discriminate between the actions associated with individual letters *'A'*, *'B'* etc. would have resulted in a more complicated and inflexible architecture. At the outermost level, however, the above extract does use a Multi_branch, which appears justified because of the small number of cases involved and the diversity of actions in each case, which do not fall into a single category such as "execute the command attached to the selected object".

See also the Single Choice principle in "Object-Oriented Software Construction", and, in the present book, "Single choice and factory objects", page 423.

The second example used Type_descriptor inspect values:

```
inspect
        last_exception
when {DEVELOPER_EXCEPTION} then
        process_developer_exception
when {OS_SIGNAL}, {NO_MORE_MEMORY} then
        cancel_operation
else
        reset
end
```

Even though we are using a Multi_branch to select different actions depending on the type of an object, we are not doing anything else with the object in question. The choices, in addition, are from a fixed set of possibilities — exception types — provided by the Kernel Library, not under developer control. So this case is similar to the previous ones; the discrimination could just as well, with a different design, have been based on a set of Unique values.

If you do anything else with the inspected object, however, Multi_branch will cease to be the better choice and you should look into dynamic binding and associated mechanisms.

16.7 LOOP

The next control structure is the only construct (apart from recursive routine calls) allowing iteration. This is the Loop instruction, describing computations that obtain their result through successive approximations.

The following example of a search routine illustrates the Loop construct with all possible clauses:

```
search_same_child (sought: like first_child) is
            -- Move cursor to first child position where sought
appears
            -- at or after current position.
            -- If no such position, move cursor after last item.
      require
            sought_child_exists: sought /= Void
      do
            from
                  child_start
            invariant
                  0 <= position
                  position <= arity + 1
            variant
                  arity — child_position + 1
            until
                  child_off or else (sought = child)
            loop
                  child_forth
            end
      ensure
            (not child_off) implies (sought = child)
      end
```

This example is close to actual tree searching routines in EiffelBase. Actual versions, however, can check for equal as well as '='.

The Loop construct extends from the keyword **from** to the first **end**.

The Initialization clause (**from**…) introduces actions, here a call to procedure *child_start*, to be executed before the actual iteration starts. The Loop_body (**loop**…) introduces the instruction to be iterated, here a call to *child_forth*; this will be executed zero or more times, after the Initialization, until the Exit condition, introduced in the **until**… clause, is satisfied.

← *"LOOP INVARIANTS AND VARIANTS", 9.14, page 169.*

The optional Invariant and Variant clauses help reason about a loop, ascertain its correctness, and debug it:

- The keyword **invariant** introduces an assertion, describing a property that must be satisfied by the initialization and maintained by every execution of the loop body if the exit condition is not satisfied.
- The keyword **variant** introduces an integer expression which must be non-negative after the initialization and will decrease whenever the body is executed, but will remain non-negative; these properties ensure that the loop's execution terminates.

Here is the general form of the Loop construct.

```
          Loop ≜ Initialization
                 [Invariant]
                 [Variant]
                 Loop_body
                 end

Initialization ≜ from Compound

     Loop_body ≜ Exit loop Compound

          Exit ≜ until Boolean_expression
```

← Invariant *and* Variant *were studied in* 9.14.

The Initialization (**from** clause) is required. If you do not need any specific initialization, use a **from** clause with an empty Compound, as in

```
from
until
        printer.queue_empty
loop
        printer.process_next_job
end
```

In general, however, the Initialization does introduce a Compound of one or more instructions, as in this example from a list duplication routine in EiffelBase:

```
from
        mark
        Result.start
until
        off
loop
        Result.put (item)
        forth
        Result.forth
end
```

The effect of a Loop is the effect of executing its Initialization followed by the effect of executing its Loop_body. The effect of executing an Initialization clause is the effect of executing its Compound. As to the effect of the Loop_body, it is:

- No effect (leave the state of the computation unchanged) if the Boolean_expression of the Exit clause evaluates to true.
- Otherwise, the effect of executing the Compound clause, followed (recursively) by the effect of executing the Loop_body again in the resulting state.

The optional Invariant and Variant parts have no effect on the execution of a correct loop; they describe correctness conditions. Their precise use was explained in the discussion of assertions and correctness. As a reminder:

← *"LOOP INVARIANTS AND VARIANTS", 9.14, page 169.*

- The Invariant must be ensured by the Initialization; any execution of the Loop_body started in a state where the Invariant is satisfied, but not the Exit condition, must produce a state that satisfies the Invariant again.
- The Initialization must produce a state where the Variant expression is non-negative; and any execution of the Loop_body started in a state where the Variant has a non-negative value *v* and the Exit condition is not satisfied must produce a state in which the Variant is still non-negative, but its new value is less than *v*. Since the Variant is an integer expression, this guarantees termination.

16.8 THE DEBUG INSTRUCTION

The Debug instruction serves to request the conditional execution of a certain sequence of operations, depending on a compilation option.

The existence of this instruction implies an obligation for Eiffel development environments to include a user option for turning "Debug mode" on and off and, more generally, to set a "Debug key". The Lace control language includes the necessary mechanisms, enabling you to set the option at all relevant levels:

→ *Appendix B discusses Lace; see "SPECIFYING OPTIONS", B.9, page 854.*

- Default for an entire system.
- Default for a cluster, overriding the system default.
- Value for a particular class, overriding the cluster default.

The basic form of a Debug instruction is

```
debug
    instruction_1
    …
    instruction_n
end
```

The instruction will be ignored at execution time if the Debug option is off. If the option is on, the execution of the Debug instruction is the execution of all the *instruction*$_i$ in the order given, as with a Compound.

A variant of the instruction enables you to exert finer control over the debugging level by specifying one or more "debug key" in the form of a Manifest_string in parentheses. For example:

```
debug ("GRAPHICS_DEBUG")
    instruction_1
    …
    instruction_n
end
```

This will be executed if and only if the Debug option has been turned on either generally as before or specifically for the given Debug_key. This way you can exercise various parts of the software separately by playing with the option, typically in the Ace file, without touching the Eiffel text itself.

→ The Ace file is the Lace control file used to set options. See appendix B.

Here is the syntax of the instruction:

Debug ≜ **debug** [Key_list] Compound **end**

Key_list was introduced in connection with the Once routine specification:

← Page 148.

Key_list ≜ {Manifest_string "," …}*

The effect of a Debug instruction depends on the mode that has been set for the enclosing 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.

Letter case is not significant for a ebug key: "*GRAPHICS_DEBUG*" is the same as "*graphics_debug*".