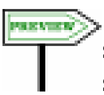
17

Control structures

17.1 OVERVIEW

The previous discussions have described the "bones" of Eiffel software: the module and type structure of systems. Here we begin studying the "meat": the elements that govern the execution of applications.

Control structures are the constructs used to schedule the run-time execution of instructions. There are four of them: sequencing (compound), conditional, multi-branch choice and loop. A complementary construct is the Debug instruction.



As made clear by the definition of "non-exception semantics" in the semantic rule for Compound, which indirectly governs all control structures (since al instructions are directly or indirectly part of a Compound), the default semantics assumes that none of the instructions *→ Chapter 26.* executed as part of a control structure triggers an *exception*. If an exception does occur, the normal flow of control is interrupted, as described by the rules of exception handling in the discussion of this topic.

17.2 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 specified:

Compound ≜ {Instruction ";" …}*

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", 34.10, page 909.

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.

There is no validity rule for Compound. The semantic specification follows from the above explanations:

Compound (non-exception) semantics

The effect of executing a Compound is:

- If it has zero instructions: to leave the state of the computation unchanged.
- If it has one or more instructions: to execute the first instruction of the Compound, then (recursively) to execute the Compound obtained by removing the first instruction.

This specification, the **non-exception semantics** of Compound, assumes that no exception is triggered. If the execution of any of the instructions triggers an exception, the Exception Semantics rule takes effect for the rest of the Compound's instructions.

Less formally, this means executing the constituent instructions in the order in which they appear in the Compound, each being started only when the previous one has been completed.

Note that a Compound can be empty, in which case its execution has no effect. This is useful for examples when refactoring the branches of a Conditional: you might temporarily remove all the instructions of the Else_part, but not the Else_part itself yet as you think it may be needed later.

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 include:

- The body of a non-deferred routine (construct Internal). ← *Syntax on page 218.*
- The Then_part and Else_part of a Conditional instruction. → *Page 473.*
- The When_part and Else_part of a Multi_branch instruction. → *Page 473.*
- The Initialization and Loop_body of a Loop instruction. → *Page 487.*
- The Debug instruction. → *Page 490.*
- The Rescue clause of a non-deferred routine. → *Page 693.*

17.3 CONDITIONAL

A basic algorithmic mechanism is the ability to discriminate between a se of values, executing a different set of instructions in each case. Eiffel provides three variants of this notion: Conditional, where discriminating criteria are boolean conditions; Multi_branch, comparing an expression to a set of specified values; and Object_test, matching a reference against a specified object type. They're studied in this section and the next two.

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.

→ *"USING SELECTION INSTRUCTIONS PROPERLY", 17.6, page 483.*

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

Conditionals

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

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

Then_part ≜ Boolean_expression **then** Compound

Else_part ≜ **else** Compound

Two auxilary notions help define precisely the semantics of this construct. 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 at least one **elseif** is the Conditional obtained by removing the initial "**if** Then_part_list" and replacing the first **elseif** of the remainder by **if**.

The secondary part of the above example Conditional is

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

The other useful notion is "prevailing immediately":

Prevailing immediately

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

These conventions enable a simple definition of the semantics:

> **Conditional semantics**
>
> The effect of a Conditional is:
>
> - If it prevails immediately: the effect of the first Compound in its Then_part_list.
> - Otherwise, if it has at least one **elseif**: the effect (recursively) of its secondary part.
> - Otherwise, if it has an Else part: the effect of the Compound in that Else part.
> - Otherwise: no effect.

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.

17.4 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.

→ "USING SELECTION INSTRUCTIONS PROPERLY", 17.6, page 483.

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*; *STRING*; and *TYPE* [*G*] for some *G*, which describe object types (conforming to *G*). This last possibility allows you to discriminate on the basis of the *type* of the object attached at run time to the value of an arbitrary expression, as illustrated by the following example of dealing with various kinds of exception object:

```
inspect
    last_exception.type
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*, as given by *last_exception.type*, and if it conforms to one of the types listed executes the corresponding **then** branch; otherwise 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.type* 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*. The inspect expression may only be of one of the types *CHARACTER*, *INTEGER*, *STRING*, *TYPE*.

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 all be of the same type as the inspect expression: all characters, all integers, all strings or all types. They must all be different, and non-conforming in the case of types; this avoids ambiguity, ensuring 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 Manifest_type, 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 *32.14, page 884*.

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

Multi-branch instructions

Multi_branch	≜	**inspect** Expression [When_part_list] [Else_part] **end**
When_part_list	≜	When_part$^+$
When_part	≜	**when** Choices **then** Compound
Choices	≜	{Choice "**,**" …}$^+$
Choice	≜	Constant | Manifest_type | Constant_interval | Type_interval
Constant_interval	≜	Constant "**..**" Constant
Type_interval	≜	Manifest_type "**..**" Manifest_type

Construct Constant *describes manifest or symbolic constants and is studied in* *"GENERAL FORM OF CONSTANTS", 29.2, page 777*.

Interval

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

To discuss the constraint and the semantics, it is convenient to consider the *unfolded form* of the instruction. First, constant and type intervals have similar properties, justifying a general term:

which enables us to define the unfolded form

DEFINITION

Unfolded form of a multi-branch

To obtain the **unfolded form** of a Multi_branch instruction, apply the following transformations in the order given:

1 • Replace every constant inspect value by its manifest value.

2 • If the type *T* of the inspect expression is any sized variant of *CHARACTER*, *STRING* or *INTEGER*, replace every inspect value *v* by {*T*} *v*.

3 • Replace every interval by its unfolded form.

Step 2 enables us, with an inspect expression of a type such as *INTEGER_8*, to use constants in ordinary notation, such as 1, rather than the heavier {*INTEGER_8*} 1. Unfolded form constructs this proper form for us. The rules on constants make this convention safe: a value that doesn't match the type, such as 1000 here, will cause a validity error. → ---- *[Add reference]*

The last unfolded form is based on another, for intervals:

DEFINITION

Unfolded form of an interval

The **unfolded form** of an interval *a*..*b* is the following (possibly empty) list:

1 • If *a* and *b* are constants, both of either a character type, a string type or an integer type, and of manifest values *va* and *vb*: the list made up of all values *i*, if any, such that $va \leq i \leq vb$, using character, integer or lexicographical order respectively.

2 • If *a* and *b* are both of type *TYPE* [*T*] for some *T*, and have manifest values *va* and *vb*: the list containing every Manifest_type of the system conforming to *vb* and to which *va* conforms.

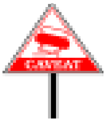
3 • If neither of the previous two cases apply: an empty list.

The "manifest value" of a constant is the value that has been declared for it, ignoring any Manifest_type: for example both 1 and {*INTEGER_8*} 1 have the manifest value 1.

→ ---- *[Add reference]*

The symbol **..** is not a special symbol of the language but an alias for a feature of the Kernel Library class *PART_COMPARABLE*, which for any partially or totally ordered set and yielding the set of values between a lower and an upper bound. Here, the bounds must be constant.

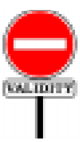
→ *In the Kernel Library specifications see classes "PART_COMPARABLE", page 967, and "INTERVAL", page 971.*



A note for implementers: type intervals such as {*U*}**..**{*T*}, denoting all types conforming to *T* and to which *U* conforms, may seem to raise difficult implementation issues: the set of types, which the unfolded form seems to require that we compute, is potentially large; the validity (Multi-Branch rule) requires that all types in the unfolded form be distinct, which seems to call for tricky computations of intersections between multiple sets; and all this may seem hard to reconcile with incremental compilation, since a type interval may include types from both our own software and externally acquired libraries, raising the question of what happens on delivery of a new version of such a library, possibly without source code. Closer examination removes these worries:

- There is no need actually to compute entire type intervals as defined by the unfolded form. Listing {*U*}**..**{*T*} simply means, when examining a candidate type *Z*, finding out whether *Z* conforms to *T* and *U* to *Z*.
- To ascertain that such a type interval does not intersect with another {*Y*}**..**{*X*}, the basic check is that *Y* does not conform to *T* and *U* does not conform to *X*.
- If we add a new set of classes and hence types to a previously validated system, a new case of intersection can only occur if either: a new type inherits from one of ours, a case that won't happen for a completely external set of reusable classes and, if it happens, should require re-validating since existing Multi_branch instructions may be affected; or one of ours inherits from a new type, which will happen only when we modify our software *after* receiving the delivery, and again should require normal rechecking.

An interval may not be empty:



Interval rule *VOIN*

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

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 Manifest_type.

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

Inspect values of a multi-branch

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

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 --- DEFINE CONSTANT MANIFEST TYPE ---:

Multi-branch rule *VOMB*

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

1 • Inspect values are all valid.

2 • Inspect values are all constants.

3 • The manifest values of any two inspect values are different.

4 • If the inspect expression is of type *TYPE [T]* for some type *T*, all inspect values are types.

5 • If case 4 does not apply, the inspect expression is one of the sized variants of *INTEGER*, *CHARACTER* or *STRING*.

--- IN CLAUSE 2: CHECK THAT DEFINITION OF CONSTANT" FOR TYPES ONLY COVERS CONSTANT TYPES ----

The clauses guarantee that there won't be any ambiguity for choosing the branch to be executed, if any.

--- NOT TRUE ANY MORE, FIX THIS --- For inspect values of the Manifest_type kind, such as *{SOME_TYPE}*, clause 4 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 or Object_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 for the value *val* of its inspect expression:

1 • *val* ~ *i*, where *i* is one of the non-Manifest_type inspect values listed in *wp*.

2 • *val* denotes a Manifest_type listed among the choices of *wp*.

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

In case 1, we look for object equality, as expressed by ~. Strings, in particular, will be compared according to the function *is_equal* of *STRING*. A void value, even if type-wise permitted by the inspect expression, will never have a matching branch.

In case 2, we look for an exact type match, not just conformance. For conformance, we have type intervals: to match types conforming to some *T*, use {*NONE*}..{*T*}; for types to which *T* conforms, use {*T*}..{*ANY*}.

Case 1 applies to a Multi_branch that lists actual inspect values: integers, characters or strings. The matching criterion is equality in the sense of *equal*.

→ *"OBJECT EQUALITY", 21.6, page 572*

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.

SEMANTICS

Multi-Branch semantics

Executing a Multi_branch with a matching branch consists of executing the Compound following the **then** in that branch. In the absence of matching branch:

1 • If the Else_part is present, the effect of the Multi_branch is that of the Compound appearing in its Else_part.

2 • Otherwise the execution triggers an exception of type *BAD_INSPECT_VALUE*.

→ *See 26.12, page 701, about exception objects.*

CAVEAT

Note the difference between the semantics of Conditional and Multi_branch when there's no Else_part and 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.

17.5 OBJECT TEST

--- SECTION REMOVED, BUT MATERIAL WILL BE REUSED FOR NEW MECHANISM REPLACING ASSIGNMENT ATTEMPT S----

17.6 USING SELECTION INSTRUCTIONS PROPERLY

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. Similarly, the Object_test may remind you of techniques for discriminating between cases based on the type of an object, sometimes known as “Run-Time Type Idenfification” or RTTI. But when it comes to writing Eiffel applications, you should be careful to not misuse these instructions. 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”, 23.12, page 630.*

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 27.

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 529.

The second example used Manifest_type inspect values:

```
inspect
    last_exception.type
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.

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.

17.7 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.

Loop structure and properties

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

```
search_same_child (sought: like first_child)
            -- 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
            until
                  child_off or else (sought = child)
            loop
                  child_forth
            variant
                  arity — child_position + 1
            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.

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

← *"LOOP INVARIANTS AND VARIANTS", 9.11, page 245.*

- 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.

Loops

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

Initialization ≜ **from** Compound

Exit_condition ≜ **until** Boolean_expression

Loop_body ≜ **loop** Compound

← Invariant *and* Variant *were studied in* 9.11.

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
```

Loop semantics

Loop semantics

The effect of a Loop is the effect of executing the Compound of its Initialization, then its Loop_body.

The effect of executing a Loop_body is:

- If the Boolean_expression of the Exit_condition evaluates to true: no effect (leave the state of the computation unchanged).
- Otherwise: the effect of executing the Compound clause, followed (recursively) by the effect of executing the Loop_body again in the resulting state.

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", 9.11, page 245.

- 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.

Ensuring non-void references in a loop

--- [SECTION REMOVED, SOME MATERIAL WILL BE REUSED] ---

17.8 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 1018.*

- 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
```

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

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.

Here is the syntax of the instruction:

SYNTAX

Debug instructions

Debug ≜ **debug** ["**(**"Key_list "**)**"]
Compound **end**

Key_list was introduced in connection with the Once routine specification: ← *Page 218*.

Key_list ≜ {Manifest_string "**,**" …}$^{+}$

SEMANTICS

Debug semantics

A language processing tool must provide an option that makes its possible to enable or disable Debug instructions, both globally and for individual keys of a Key_list. Such an option may be settable for an entire system, or for individual classes, or both.

Letter case is not significant for a debug key.

The effect of a Debug instruction depends on the mode that has been set for the current class:

- If the Debug option is on generally, or if the instruction includes a Key_list and the option is on for at least one of the keys in the list, the effect of the Debug instruction is that of its Compound.
- Otherwise the effect is that of a null instruction.