A short course in Eiffel

Colm Ó Dúnlaing

September 23, 1997

Contents.

1 Compiling a simple program.

(1.1) Eiffel/S. These notes give a brief and incomplete sketch of the Eiffel language. We shall be using the Eiffel/S programming environment, version 1.3 (which is available free). This includes a language compiler. It also includes essential documentation and other tools essential for building a runtime system, plus an extensive library of program modules.

The Eiffel system is typically stored in the subdirectory /usr/local/eiffel. In any case, you must name the correct subdirectory in an environment variable EIFFEL_S. (See the listing below 1.3.)

For a fuller introduction to the language, look at the subdiretory 'introduction' of the Eiffel directory. You could also refer to the Eiffel Home Page

```
http://www.cs.cf.ac.uk/CLE/
```

The language is fully defined by its designer, Bertrand Meyer, in *Eiffel: the Language*, which will be our 'Bible.'

(1.2) Eiffel an object-oriented language. The most important phrase describing the Eiffel programming language is 'object-oriented.' It can be considered as a radical improvement of C++.

Eiffel itself owes much to C++, and something to Ada. Its main advantages compared with C++ are: it is much less complicated, it has much better error detection, and it has garbage collection, freeing the programmer of all memory management responsibilities. Also, with their Ada-like syntax, Eiffel programs are much easier to read than C, C++, or Java. Eiffel does not encourage the terse do-everything-at-once style of those languages.

The idea of object-oriented programming is that the data should be stored in 'objects,' where each ohject stores data and is allowed to process that data in restricted ways. In contrast, a Pascal program would be just a single object.

Eiffel is small. Eiffel is a sophisticated language but it is em , and its author Bertrand Meyer expects every Eiffel programmer to use every feature of the language.

Java. Java seems to have borrowed some ideas from Eiffel, though I have not heard this stated explicitly. At present the only technical advantage that Java possesses is concurrency: a concurrent version of Eiffel has been announced, but we'll not study it.

(1.3) PDL files. A runtime system is built from a collection of class definitions. (Classes will be discussed presently.) In EIFFEL/S, the version we shall be using, the 'plan' for building a system is in a 'pdl file' (PDL= program definition language). So we begin with a pdl file. This file would typically be named 'test.pdl'.

```
program test
root
    test : go
cluster
    "./"
    find
        "test" in "test.e"
    end
    include  "$EIFFEL_S/library/lib.pdl"
end -- program test
```

(1.4) This PDL file tells the Eiffel compiler how to build the runtime system (which will eventually emerge as an executable file). In order to build the system, it needs to know where the various classed are defined. Also, the executable file needs to begin somewhere: here, the compiler is told that the system begins by creating an object of class 'test' and having that object execute the procedure 'go'.

In other words, the *root object* for the program is of class 'test,' and this class is defined in the file 'test.e' in the current directory. In a pdl file, 'cluster' means 'directory.' The 'include' line reads the file lib.pdl, which describes where the various Eiffel/S library modules can be found.

(1.5) So now let us write a simple Eiffel program. According to the pdl file, the class 'test,' to which the root object belongs, is in 'test.e' in the current directory. The following will work.

```
class TEST
creation
        go
feature
  go is
  local
        io: BASIC_IO
        n : INTEGER
  do
        !! io
        io.put_string ("Running test.%N")
        from
           n := 1
        until
           n > 3
        loop
           io.put_int (n)
           io.put_newline
           n := n+1
        end
        io.put_string("Test finished.%N")
  end
end
```

(1.6) The basic unit in Eiffel is the 'class,' which, as in C++ and Java, describes a data-structure together with operations on the structure. A class is presented as a list of *features*, which we would recognise as being either constants, variables, functions, or procedures.[1]

(1.7) Our example describes the class 'test'[2]. The class test has just one feature, the routine 'go.'

To compile the program, type in

```
ecc test
```

and wait. The executable file is named 'test.' Type `test` and, not surprisingly, the program prints

[1]Constants and variables are called *attributes*, procedures and functions are called *routines.*

[2]Case is not significant: test, TEST, and tEsT are synonymous

```
Running test
1
2
3
Test finished.
```

(1.8) The next section will summarise a usable subset of the language, and should explain all of the above simple example. However, all of the example should be self-explanatory except for `%N`, which means 'new line,' and `!!io`. This is a *creation command.* It means 'create an object instance of class BASIC_IO and store it in the variable io.' Until this is done, io contains a special value, `void`, and is useless. Afterwards, the object stored in io can be used, and is used, for simple i/o functions.

(1.9) CONSERVE DISK SPACE — USE ECLEAN. The Eiffel compiler translates into C code and then invodes a C compiler. It generates *gigantic* amounts of C code, typically 3 megabytes for any program no matter how trivial. This is slow, but you will find that if you make small program changes then recompilation will be fast: the C code is held in a file called `test.p` (if your PDL file was called 'test.')

When you have finihsed for the day, please invoke `eclean` to remove this subdirectory.

2 Basic language elements.

(2.1) Disclaimer. It would be impractical to give full details here. Most Eiffel constructs have optional components, and only those of immediate use will be described at this point. More detailed information can be obtaines as menioned in 1.1. Your Bible, of course, is 'Eiffel the Language' by Bertrand Meyer.

(2.2) Types. Eiffel runtime systems manipulate objects. Every object is of a certain *type*. There are five *basic types*: INTEGER, REAL, DOUBLE, BOOLEAN, and CHARACTER. All other types are 'reference types.'

A variable whose type is a reference type will be implemented as a pointer, but officially there are NO pointers in Eiffel. (Java likewise.)

(2.3) Constants. Constants in Eiffel look the same as in other languages: 0, 0.0, true, false, 'c', "string". There is a constant `void` of reference type. A string constant is recognised in Eiffel although string is not one of the basic types.

There is provision for special character constants, such as '%N' for newline. '%%' means '%'.

(2.4) Comments. Comments are preceded by two hyphens, `--`, and continue to the end of the line.

(2.5) Eiffel is not case-sensitive. Identifiers are the same as in most languages. Underscores can be used. Case is ignored: Integer, integer, INTEGER, and iNtEgEr are regarded as the same identifier. Meyer lays down some guidelines for capitalisation.

(2.6) Semicolons are unnecessary. Semicolons between statements are optional! This may be surprising and may come as a relief to those who never quite cottoned on to the interaction of semicolons with braces in C. The language does not need semicolons because it never allows two identifiers to appear beside one another in the same statement.

(2.7) Assignments and reference semantics. Assignment statments look the same as in Pascal. For reference types, however, they behave utterly differently. Supposing that a and b are variables of the same reference type. Let O be the object stored in b. Then b will actually contain the address of O, which the statment `a := b` will cause to be copied to a. If O is changed through b, then the change will be visible from a.

In practice, this is much less troublesome than it looks. (It may come as a surprise that Java behaves the same way.)

Of course, an assignment statement a:=b is subject to type restrictions. For the time being we'll expect that a and b be of the same type.

(2.8) No need for memory management. Eiffel uses garbage collection, which should be very welcome to victims of C++. Allocation of memory, and its re-use, are entirely handled by Eiffel. (Likewise Java.)

(2.9) Classes. Now to business. The basic programming unit in Eiffel is the *class*, which is declared as follows.

```
class THISCLASS
creation
    ... creation clause
feature
    ... features
end -- class THISCLASS
```

(2.10) Creation clause. The creation clause, which is optional, names a routine which is invoked (usually for initialisation) when an object is created. It is not absolutely necessary, since Eiffel has a uniform initialisation rule: objects of numeric type are initialised to zero, booleans to `false`, characters to the null character ('%U' in Eiffel's strange notation), and reference types to `void`.

(2.11) Features. A feature can be a constant, variable, procedure, or function.[3] A variable is declared like:

```
<name> : <TYPE>
```

a constant like

[3]Eiffel calls constants and variables 'attributes,' and procedures and functions 'routines.'

```
<name> : <TYPE> is <constant>
```

('is' is a keyword); a procedure like

```
<name> is
    <local variables>
do
    <procedure body>
end
```

and a function like

```
<name> : <TYPE> is
    <local variables>
do
    <function body>
end
```

(2.12) 'Result.' Functions return their values through a predefined local variable 'Result' ('result' is a keyword). It can be treated the same as a local variable whose type is the same as the function's.

(2.13) Type like type. 'a: INTEGER' is a simple example of variable declaration. Other type names could be taken from the names of classes, such as BASIC_IO; and 'parametrised types,' such as LIST[INTEGER], are possible.

A very convenient way of defining type is to declare it to be the same as some other variable's. This is easier to write and easier to edit:

```
a: LIST[INTEGER]
b: like a
```

Although it looks silly, it has the advantage that the type of `a` can be altered, without having to edit the declaration of `b`.

(2.14) Current. At runtime, there could be many objects of the same class. An object can reference itself through the keyword 'current,' meaning 'current object.' So a declaration

```
b: like Current
```

says that the type of `b` is the same as the enclosing class.

(2.15) More about assignment statements. The general form is

```
<target> := <expression>
```

where <target> is just a variable and <expression> looks much the same as in Pascal — combinations of variables, constants, operators, and function calls such as

```
x := f(y)
```

In this case, the function `f` is a feature of the surrounding class; we could also have a function call like

```
x := c.f(y)
```

where `x`, `y`, and `c` are local variables (local to a routine or features of the surrounding class), and `c` belongs to a class of which `f` is a feature.[4]

(2.16) Operator features. Eiffel does not contain much 'syntactic sugar' — palatable but unnecessary ways to vary the notation — but it sometimes allows a different style of function or procedure call. This is when the routine is introduced as

```
infix "<op. name>" (<second arg>) is .......
```

or `prefix "<op. name>" is ....` In this case the feature is called an 'operator feature' and is called with prefix or infix notation. The form of <op. name> is different from ordinary identifiers, and must be either one of[5]

```
not, +, -  (all prefix operators),
        OR
+, -, *, /, <, >, <=, >=, //, \\, ^,
and, or, xor, and then, or else,
        OR
implies (all infix operators).
```

[4] `c.f(y)` is called a qualified call and `f(y)` an unqualified call.

[5] `//` is for integer division and `\\` is for integer remainder or modulus.

or must be a ‘free operator.’ This can be any string of nonblank characters beginning with `@`, `#`, `|`, or `&`. The operator name, whether free or standard, must be given as a string constant, enclosed in double quotation marks.

A typical use is with arrays in the Eiffel/S library: `A@i` means the ith element of array A.

(2.17) If-statements. These have the general form

```
if <condition> then
    <statements>
elseif <condition> then
    <statements>
...
else
    <statements>
end
```

The ‘elseifs’ (there can be any number) and the ’else’ are optional. There is only one ‘end.’

Boolean expressions. These can include the boolean operators ’not,’ ’and,’ ’or,’ and ’xor.’ The last is ‘exclusive or’: `a xor b` is true when either `a` or `b`, but not both, is true.

There are three useful ‘semi-strict’ operators: ‘and then,’ ‘or else,’ and ‘implies.’

- `a and then b` is false if `a` is false and is true if both `a` and `b` are true. The point is that if `a` is false then `b` is not evaluated, which can be extremely useful (when evaluation of `b` might crash the program).
- `a or else b` is true if `a` is true and false if both `a` and `b` are false.
- `a implies b` is equivalent to `(not a) or else b`: true if `a` is false and false if both `a` is true and `b` false.

(2.18) Inspect-statements. These correspond to the case statement of Pascal and the switch statement of C:

```
inspect
   <variable>
when <list 1> then
   <statements>
when <list 2> then
   <statements>
......
else
   <statements>
end
```

The variable must be CHARACTER or INTEGER, and the lists must be disjoint lists of constants of the variable's type.

(2.19) Iteration. There is only one form of repetition:

```
from
    <preparation>
until
    <termination>
loop
    <statements>
end
```

It is an unusual format, but quite simple. Notice that the preparation is an action — a group of statements — whereas termination is a condition — a boolean-valued expression.

3 The Eiffel/S Library.

(3.1) The Eiffel/S system comes with an excellent library of predefined classes. Here is a partial listing of `$EIFFEL_S/library`:

```
lib.lib                  -- some kind of configuration file
lib.pdl                  -- pdl file describing library clusters
                         -- (as in Section 1)

basic:                   -- basic classes
array.e string.e         -- classes ARRAY and STRING
compar.e hashable.e      -- COMPARABLE and HASHABLE
basic_io.e file.e file_sys.e format.e
fsys_dat.e textfile.e    -- input/output

contain:                 -- 'container' classes (all encrypted)
dictnary.ce h_table.ce   -- DICTIONARY and HASH_TABLE
iterator.ce travers.ce   -- classes for 'iterators.'
list.ce                  -- (linked) lists
prqueue.ce queue.ce stack.ce  -- priority queue, queue, stack.

math:
math.e                   -- mathematical functions: sin, cos, etc.

sorter:
sorter.ce                -- sorting tool (encrypted)
```

(3.2) Most of these classes are 'generic.' For example, the class ARRAY describes arrays containing objects of any given type: ARRAY[INTEGER] describes arrays of integers, ARRAY[STRING] describes arrays of strings.

(3.3) If A is an array, its ith element is returned from the function call A@i (@ is an infix operator feature), or equivalently from A.item(i). It cannot be the target of an assignment statement: to change this entry use A.put(x,i).

A.count returns the size (number of elements) of the array A. Surprisingly, this can be altered: arrays in Eiffel can grow and shrink.

(3.4) Strings are similar to ARRAY[CHARACTER] with some extra operations, such as concatenation. (As mentioned in 5.2, this means an array of characters, not an array indexed by characters).

(3.5) The class COMPARABLE describes objects which can be compared (hence linearly ordered), and the class HASHABLE describes objects for which a hashing function is supplied, enabling them to be accessed through hash tables.

(3.6) The class BASIC_IO provides routines for reading from standard input and writing to standard output. FORMAT provides routines for converting types BOOLEAN, INTEGER, and so on, to strings, and vice-versa. It does not handle i/o.

(3.7) The class FILE is, of course, for file handling. Files in Eiffel/S are similar to arrays, allowing random access. As in Pascal, and unlike C, for any type G there can be a file of elements of that type. Class TEXTFILE is intended to describe ASCII files, but they are treated as arrays of strings (separated by newline).

(3.8) Class FILE_SYSTEM describes clusters (the Eiffel word for directories). Class FSYS_DAT describes the format of file descriptions used by FILE_SYSTEM.

(3.9) The ITERATOR class is very intriguing. Through it, Eiffel/S can provide a uniform way of iterating through traversable objects (most container classes are traversable). If iter is a variable of iterator type, it would be used in the following way to traverse a 'container' object Oholding elements of type G, say; suppose that the variable x is 'attached' to O:

```
from
  iter := x.iterator
       -- creation initialises iter
       -- to hold first container element.
until
  iter.finished
       -- condition signalled when
       -- container has been fully traversed
loop
  ....
       -- iter.item is the item
       -- currently being scahhed
  iter.forth
       -- (idiosyncratic word).  Iterator reset
       -- to hold next element in traversal.
end
```

Notice that `iter` is not of type `G`, but `iter.item` is.

4 Assignment statements, arguments, and copying.

(4.1) Restricted target in assignment statement. An assignment statement has the form

```
<target> := <value>
```

and Eiffel is very severe about what the 'target' may be:

> *The target of an assignement statement must be either a variable local to the class (i.e., an 'attribute') in which the statement occurs, or local to the routine in which the statement occurs.*

(4.2) Routine arguments. Subroutine arguments are 'call-by-value,' but, what is more, they can not be used as local variables as in other languages: they cannot be the targets of assignments within the subroutine body.

(4.3) Attributes of other objects. Qualified attributes cannot be the targets of assignemts, i.e., statements of the form

```
a.b := c
```

are forbidden. (Indeed, it may be impossible to discover whether `a.b` is an attribute or a function, if the definition of `a`'s class is encrypted).

So, in Eiffel, you motify an array by invoking 'put.' Direct assignment to array elements is impossible.

Reason for restriction. There is a reason for this. Eiffel emphasises the use of *assertions*, whereby the designer of a class guarantees some conditions and assumes others. The class 'takes responsibliity' for internal data consistency. This obviously is impossible if the data can be altered via assignment statements.

(4.4) Type restrictions. 'Conformance.' It is required that the type of the value should *conform* to the type of the target. Identical types conform automatically. The numerical types are `DOUBLE`, `REAL`, and `INTEGER`, in descending order of 'weight.' A numeric value is considered to conform to a 'heavier' target.

Otherwise there is no relation among the basic numeric types and the basic types `CHARACTER` and `BOOLEAN`.

For reference types, conformance is related to class inheritance, and will be discussed later.

(4.5) Assignment attempt.[6] Recall that for reference types an assignment *shares* an object: `a := b` means:

> the object currently attached to `b` becomes also attached to `a`.

The object may be altered through both `a` and `b`: any alteration is visible through both.

Eiffel allows for a curious generalised assignment statement, called an 'assignment attempt,' for reference types. The format is

```
<target> ?= <value>
```

and the meaning is: if the value conforms to the target, attach it to the target; otherwise, the target becomes `void`.

(4.6) Equality. In Eiffel, `p = q` is true iff the expressions are identical: if two expressions `p` and `q` are of basic type `CHARACTER`, etc., then, `p = q` is true if and only if they are identical. Likewise for reference types they must both be void or both attached to the same object.

Also, the assignment statement `a := b` makes `a` and `b` identical.

(4.7) For most classes, more general versions of equality are defined: `p.is_equal(q)` and `p.is_deep_equal(q)`. The latter is true iff the entire 'network of objects' accessible through `p` is isomorphic to the 'network of objects' accessible through `q`.

(4.8) Analogously there are variations on simple assignment: `a.copy(b)` and `a.deep_copy(b)`. There are also `clone(b)` and `deep_clone(b)`, which create new copies of `b`.[7]

---

[6] This is possibly an advanced feature, not for immediate use.

[7] In cloning, *b* can be of almost any type, unrelated to the class in which the clone function is invoked. See the next section, on inheritance (5.8).

(4.9) The details are as follows.

- `p.is_equal(q)` is actually an asymmetric relation: the type of `q` must conform to that of `p` but not necessarily vice-versa. Then for every attribute `p.x` of `p` the corresponding attribute `q.x` must be identical.
- `p.is_deep_equal(q)` is true if for every attribute `p.x` of `p`, the corresponding attribute of `q` is either identical or `p.x` is a nonvoid reference and `p.x.is_deep_equal (q.x)` (recursively).
- `a.copy(b)` copies attributes from `b` to `a`, so afterwards `a` and `b` are `equal`.
- `a.deep_copy(b)` is a recursive version of the same, so the objects attached to `a` and `b` access disjoint isomorphic structures.
- `a := clone(b)` causes a new object `O` to be created, then `O.copy(b)` is executed, and then `a` is attached to `b`.
- `a := deepclone (b)` is the 'deep copy' version of cloning.

5 Generic classes, inheritance, and deferred classes.

(5.1) Generic classes. In most programming languages it is possible to have arrays of characters, integers, etcetera, indeed, of any type. The keyword 'array' is commonly used to introduce array types, In Eiffel, 'array' is just another class (part of the standard library), but it is *generic* (or *parametrised*). It is declared in the form

```
class array [G] ...
```

so `G` is the type parameter, and the type must be instantiated when using the class, as in

```
ascii: ARRAY [CHARACTER]
```

(5.2) Brackets exclusive to type paramtrisation. This notation is similar to Pascal's with the opposite meaning. In Pascal, `array [char] of boolean` would mean a boolean array indexed by character values; here we have a character array indexed by integer values. In Eiffel, array indeces *must* be integers.

In Eiffel the bracket notation `[...]` is used exclusively for parametrised types. For this reason the array indexing operator is '@', as in `A@i`.

(5.3) A class can, of course, have several type parameters, as where

```
class hash_table [ G, H ] ...
```

provides for tables with different kinds of data and different kinds of keys. `G` could be the 'key type' and `H` the 'data record type.'

(5.4) Inheritance, Eiffel is designed for 'off the shelf' programming. Your programs should combine your own code with existing code. This is enabled by *inheritance.* A class can inherit features from one or more other classes. So we now reveal some more of the class definition:

```
class <name> <type parameters>
inherit
    <inherited class>
        rename <feature name> as <feature name>
               <feature name> as <feature name>
        ....
        undefine  <feature> <feature> ....
        redefine  <feature> <feature> ....
        select  <feature> <feature> ....
    end
    <inherited class> ....
    end
creation ....
feature ....
end -- class <name>
```

(5.5) Example of inheritance. As an imaginary example, a group is a monoid with an added inverse operation. This could be reflected in class inheritance.

```
class monoid
creation ...
feature
  unit is ....
  product ( x, y : ... ) is ...
end

class group
inherit MONOID
creation ...
feature
  inverse ( x: ... ) is ....
end
```

(5.6) Multiple inheritance. A class can inherit from several class. Recall that a ring is a set which is a commutative group under one operator and a semigroup under the other.

```
class ring
inherit
  GROUP
    rename
      unit as zero
      product as sum
      inverse as minus
  end
  MONOID
    undefine
      unit
  end
creation ....
feature ....
end
```

(5.7) Ancestors and descendants. In defining a class `X`, the classes named in `inherit` clauses are called the 'parents' of `X`. Continuing, we get all the ancestors of `X`. `X` inherits (perhaps indirectly) from all these classes.

Needless to say, 'descendant' is opposite to 'ancestor.'

(5.8) Cloning, equality, etcetera. There is a standard Eiffel cllass `ANY` from which most classes are descended. In this class, or an ancestor of this class, are defined the features 'is_equal,' 'deep_copy,' etcetera, as mentioned in 4.8.

(5.9) Deferred features. Recall the form of feature declaration for a constant or routine

```
class A
...
feature ....
   x: .... is ....
end -- class A
```

where the 'is' is followed by a constant or a procedure body. There is another possibility: the keyword 'deferred.' This means that the class `X` omits the details required for the feature `x`, leaving it up to descendant classes to fill in these details.

(5.10) If the class has a deferred feature, then the above code is incorrect syntactically. The keyword `class` should be preceded by the keyword `deferred`: so, `deferred class A ....`

(5.11) Type conformance. The rule for conformance of reference types, as mentioned in 4.4, is basically that

The target's class should be an ancestor of the value's.

This is basically because every feature of the ancestor class is a feature of the descendant class, so the target may 'forget about' some of the features of the assigned value...

(5.12) Variables can be of deferred type, but there are restrictions. They can be targets of assignments, but cannot use creation statements like `!!x`.

(5.13) Synonym features. It is possible to define several routines together: for example,

```
feature ...

      item, infix "@" (x: ... ): like Current is ....
```

This is 'syntactic sugar,' allowing the user to invoke the same feature by different names (or as shown here, in standard and infix forms). But also, the features can be redefined in descendant classes, thereby becoming distinct.

6 Information hiding.

(6.1) 'Information hiding' means hiding information from *programs,* not from *programmers.* Eiffel program libraries are held as source code, and unless encrypted (usually indicated by the suffix `.ce` instead of `.e`) these can be read by anyone.

(6.2) Clients and suppliers. Baically, if a class `C` contains a declaration

```
.... x : S ....
```

where `S` is a class, then `C` is called a *client* of `S`, and `S` a *supplier* of `C`. 'Information hiding' is where the supplier class `S` restricts the range of its features accessible to objects of the client class `C`.

(6.3) Eiffel has a very simple means of restricting information, by means of 'export lists, as follows.' A class definition can contain several different `feature` parts.

```
class MYCLASS
inherit
....
feature { export list }
... some features ...

feature { another export list }
.... more features ....

end -- class MYCLASS
```

The 'export lists' are simply lists of classes to whom the features are accessible. `{ANY}` is the same as omitting an export list (remember most classes are descendants of `ANY`). `MYCLASS` alllows other objects of the class `MYCLASS` to access the features. The export list `{NONE}` makes the feature completely private to the current object.

(6.4) Inherited features and export lists. When a class inherits features from another class, it can modify its export lists, simply by including export clauses

```
inherit
   A
     rename ...
     export
        { export list } feature, feature, ....
        { export list } feature, feature, ....
        ....
     undefine ...
     redefine ...
     select ...
   end
   B ....
```

7 Assertions.

Eiffel encourages the use of assertions, which are ways of stating that certain conditions should be met at various points in a program.

(7.1) Require and ensure: preconditions and postconditions. Firstly, a routine can require certain preconditions to hold when it is invoked, and can promise certain postonditions to hold when it terminates, such as:

```
feature ...
   gcd (x, y: INTEGER)  : INTEGER is
   require
      nonneg: x >= 0 and y >= 0
   do
   .....
   ensure
      x = 0 and y = 0 implies Result = 0
      x /= 0 or y /= 0 implies
          (x - y) \\ Result = 0
           -- note: \\ is the mod operator
   end -- gcd
```

meaning, of course, that the `gcd` function expects that both its arguments are nonnegative. Eiffel programs are usually run with 'require' monitoring enabled, so if the condition is violated, the program will abort with a message to the effect that the requirement labelled 'nonneg' was violated.

Labels such as 'nonneg' are optional. Several assertions can be bundled together. They are checked in the order given, and checking stops once an assertion fails to hold.

(7.2) Assertion checking priority. There are other kinds of assertion: class invariants, loop invariants, loop variants, and check statments. Their monitoring can be controlled through a so-called RCL (runtime control language) file, using the 'runtime compiler' ercc.

Assertion checking has several priority levels. To remember the order of priority we use the mnemonic

REILAC

or if you prefer

sCALIER

meaning: Require has highest priority, then come Ensure, class Invariants, Loop variants and invariants, and then All (including Check statements). So, for example, if loop invariant monitoring is enabled, then so are class invariants, ensures, and requires. The default monitoring level is Require.

(7.3) Class invariants. These are introduced at the end of a class definition:

```
class MYCLASS
.... features etcetera ....

invariant
    .... assertions ....
end -- class MYCLASS
```

The class invariant is evaluated together with any 'require' assertions, when a feature is accessed from outside (as in `a.gcd(x,y)`), and again after it is completed (together with any 'ensure' assertions).

(7.4) Loop invariants. The complete form of a loop statement is

```
from
   ....
until
   ....
invariant
    ... assertions ...
variant
    expression
loop
    ....
end
```

The loop invariant is checked before every iteration of the loop, and the loop variant, if included, is an expression which is checked bewteen iterations: it should always be a nonnegative integer and it shoud always decrease with each iteration.

(7.5) Check. A check instruction has the form

```
check
   ... assertions ...
end
```

Check instructions can be written like ordinary statements. If check monitoring is enabled, the assertions are checked; otherwise the instruction has no effect.

(7.6) Debug. A debug instruction is similar to a check instruction, except that it involves statements and usually does not trigger abnormal program termination. The format is

```
debug ("SIGN","gcd")
   ... statements ....
end
```

where the list of strings `( ... )`, called 'debug keys,' is optional. Execution is controlled through the RCL file. If debugging is disabled, then the statements are skipped. If enabled, or if enabled for one of the debug keys, then statements are executed.

Debug keys are case insensitive: `"Sign"` and `"SIGN"` are the same.