Introduction to Eiffel

By: Helmut Brandl, copyright 2009

The Eiffel language is an open standard. The *tecomp* compiler is likewise one of the open source compilers available for the language.

In CSE3311, we will be using the *eiffel.com* compiler (part of *EiffelStudio* IDE). The only difference is that *tecomp* uses ACE files and *EiffelStudio* uses ECF for configuration. Otherwise, both compilers are in the process of trying to comply with the ECMA standard.

So, in the tutorial that follows, just ignore the details specific to *tecomp*.

The latest version of this tutorial is available from http://tecomp.sourceforge.net

SOURCEFORGE.NET The Eiffel Compiler / Interpreter (tecomp)

TUTORIAL

This document gives a quick introduction into Eiffel. The introduction covers the most basic elements necessary to write programs in Eiffel.

Like stated by Brian Kernighan and Denis Ritchie in their famous book "The C programming Language" the best way to learn a programming language is to write programs in it. Therefore the focus in this introduction is to write simple but useful programs. Most of the programs written in this introductory section are just Eiffel versions of first programs of the mentioned book from Kernighan and Ritchie.

The following is not an introduction into programming but into writing programs in the Eiffel language. A basic working knowledge for writing programs in languages like C, C++ or java is assumed.

Table of contents

Hello world

Our first program will just print the words

    Hello, world

An Eiffel program doing this consists in

```
class
        HELLO
create
        make
feature
        make
             do
                    io.put_string ("Hello, world")
                    io.put_new_line
                end
end
```

All Eiffel code resides in classes. Each class has its program text in a file. The source code of the above class has to be written in a file named "hello.e" (all in lowercase letters). Eiffel is not case sensitive. However class names are usually written in uppercase letters and feature names in lower case letters.

The way to compile and execute an Eiffel program depends on the system and the used compiler. The information given here is valid for the eiffel compiler `tecomp` running in a UNIX environment.

In order to compile the program, the compiler needs some information. The information is given in a ace-file. E.g. the above program can have the ace-file "hello.ace" with the content

```
root
  HELLO.make
```

```
cluster
  "./"
  "`path_to_tecomp_installation'/library/kernel"
end
```

In the current version of `tecomp` the path to your tecomp installation has to be given as an absolute or relative path. Future version will surely have some way to give this information more symbolically.

You can compile and execute the program by typing the command

```
tecomp hello.ace
```

and it will print

```
Hello, world
```

Now some explanations to the program. An Eiffel program consists of an arbitrary number of classes. One of the classes has to be the root class and one procedure has to be the root procedure. For the above program the root class is named HELLO and the root procedure `make`.

The compiler needs to know where to find classes. All classes reside in clusters which are usually implemented by directories of the used operating system. The above ace-file names the two clusters "./" (i.e. the current directory) and "`path_to_tecomp_installation'/library/kernel" (i.e. the directory where the Eiffel kernel classes are stored). The compiler searches for Eiffel classes in these clusters (i.e. directories) and complains, if the used classes in your program are not found in these clusters. The set of all Eiffel classes found in the clusters is called the *universe*.

The execution of an Eiffel program starts by creating an object of its root type and calling its root procedure (for the time being the words *type* and *class* are used synonymously, they only differ in case of generic classes/types).

The root procedure can create any number of other objects and call any routine of any created object.

In Eiffel like in many modern languages the input is in free format, i.e. any blanks, tabs and newlines in the software text are not important. The indentation is for readability for the human reader and not for the compiler.

Names like **class**, **create**, **feature**, **do** and **end** are *keywords* of the language. They are reserved. No class, feature or variable can have a name identical to a keyword.

Now lets look at the structure of the above Eiffel program

```
class
       HELLO            -- the class name
create
       make             -- the creation procedure(s)
feature
       ...              -- the features of the class
end
```

This skeleton says that we are defining a class with the name HELLO. Objects of type HELLO can only be created by using the creation procedure `make`. All features of the class are declared in the feature block `feature...end`.

A feature is either a routine or an attribute. Our simple program has only one feature named `make`. The feature `make` is a routine. A routine can take arguments and return a result. A routine which does not return a result is called a command or a procedure, routines with result are called queries.

the routine `make` is the creation procedure of the class HELLO because `make` is listed in the set of creation procedures (in HELLO it is the only one).

The code of `make`

```
make
     do
            io.put_string ("Hello, world")
            io.put_new_line
     end
```

has only 2 statements. The statement `io.put_string("Hello, world")` calls the feature `io`. Every class in Eiffel can call `io`, because the feature `io` of type STD_FILES is defined in the universal class ANY which is implicitely inherited by every class of an eiffel system. Since `io` returns an object of type STD_FILES, features of STD_FILES can be called.

STD_FILES has the feature `put_string` with a string argument. The feature `put_string` is a command because it does not return anything. It outputs its string argument to standard output. The feature `put_new_line` does just what it says.

The notion of a feature is fundamental in the Eiffel language. Therefore some basics are explained here.

A feature has two views. The user or client view and the implementation view.

In the client view we distinguish between *queries* and *commands*. A query can take zero or more arguments and has a return value. It is good practice -- allthough not enforced by the language -- that a query only gives the result without any side effect. A command can also take zero or more arguments and does not return anything. It is supposed to change to state of the object.

In the implementation view we distinguish between *attributes* and *routines*. The value of an attribute is stored within the object. Calling an attribute does not do any computation. It just returns the value of the attribute. Routines do have a body and are therefore computation elements. Routines are queries or commands. In the implementation view commands are also called *procedures* and queries which are implemented by routines are also called *functions*.

Therefore a query can be implemented by an attribute or a function, a command has to be implemented by a procedure.

A sequence of characters in double quotes, like "Hello, world", is called a character string or a string constant. Special characters like *newline* and *tab* can be included into string constants by escape sequences. E.g. `%N` and `%T` are

the escape sequences for *newline* and *tab*. Therefore we can also write

```
io.put_string ("Hello, world%N")
```

giving the same output as

```
io.put_string ("Hello, world")
io.put_new_line
```

If you want to output large strings spanning over several lines in exactly the same format as given in the code, you can use verbatim strings. E.g.

```
io.put_string ("[
     usage: tecomp options ace_file

            options
                  -t{p,v,e}{0,1,2,3}  trace parsing, validation,
                                      execution with level 0,1,2,3
                  -ws{0,1,2,3}        write statistics
             ]")
```

outputs your string exactly a formatted. Putting the verbatim line sequence between "[and "] removes the longest common whitespace prefix of all lines (i.e. output the string left justified). Putting the verbatim line sequence between "{ and }" make an absolute verbatim copy without any whitespace prefix removing.

Verbatim string constants are similar to *here docs* encoutered in many UNIX shells.

If you want to write a string constant over several lines without having the newlines embedded you can use *line wrapped* strings. The statement

```
io.put_string ("Hello, %
            %world")
```

gives exactly the same output as

```
io.put_string ("Hello, world")
```

The whitespace between the two percent signs is simply ingored in constructing the string constant.

The class ANY also has feature called `print` which can print any object to default output. So the shortest *Hello world* program has the form:

```
class  HELLO create make feature
       make
            do
                  print ("Hello, world%N")
               end
end
```

Local variables, arithmetic expressions and loops

The next program uses the formula

```
degrees Celsius = ( 5 / 9 ) ( degrees Fahrenheit - 32 )
```

to print the following table of Fahrenheit temperatures and their centigrade or Celsius equivalents:

```
0       -17
20      -6
40      4
60      15
80      26
100     37
120     48
140     60
160     71
180     82
200     93
220     104
240     115
260     126
280     137
300     148
```

This table can be printed by the Eiffel program

```
class
        FAHR_CELSIUS
                -- print a fahrenheit-celsius table
create
        make
feature
        make
                local
                        fahr: INTEGER -- degrees Fahrenheit
                do
                        from
                                fahr := 0
                        until
                                fahr > 300
                        loop
                                io.put_character ('%T')
                                io.put_integer   ( fahr )
                                io.put_character ('%T')
                                io.put_integer   ( (fahr-32) * 5 // 9 )
                                io.put_new_line
                                fahr := fahr + 20
                        end
                end
end
```

Any characters between `--` and the end of the line are ignored by the compiler, they are *comments*.

In Eiffel local variables can be declared in each routine. They come before the `do end` block of a routine. In the above program the local variable named *fahr* is declared to be of type INTEGER. Eiffel is strongly typed. Therefore any variable, expression, etc. has to have a type. In Eiffel an INTEGER is a number between -2^31 and 2^31 - 1, i.e. it has at least 32 bits.

INTEGER is a class of the kernel library.

The procedure *make* of FAHR_CELSIUS has a loop with an initialisation section, an exit condition and a loop body. In Eiffel a loop works as follows.

- The inititialisation section is executed (`from` ...).
- The exit condition is tested (`until` ...).
- As long as the condition is false, the loop body (`loop` ...) is executed and the exit condition is retested
- As soon as the exit condition evaluates true, the loop terminates and executions resumes with the first statement below the loop (... `end`).

The expression

```
(fahr-32) * 5 // 9
```

is an integer expression. The usual arithmetic precedence rules are valid. Therefore `fahr-32` has to be in parenthesis. The operator `//` is an integer division.

Characters are in single quotes. 'a' is the character *a*. '%T' is the special character tab.

Character input and output

The kernel library allows you to read and write from and to files. A file is seen as a sequence of lines separated by newline characters. Each line is a sequence of characters.

This view is independant from the used platform or operating system. On some systems (e.g. Windows) the lines in a file are separated by the two characters *carriage return linefeed*. Does kernel library does the corresponding mapping that from the perspective of the Eiffel program the file looks like a sequence of lines separated by newline characters.

Each Eiffel program has 3 files or *text streams* open: `standard_input`, `standard_output` and `standard_error`. By default `standard_input` is the keyboard and `standard_output` and `standard_error` are the screen. By using pipes or io redirection the standard files can also be connected to physical files or temporary file buffers. A program which just reads from `standard_input` and writes to `standard_output` does not care, to which resources the files are connected.

The query `io` from the class ANY returns an object of type STD_FILES which gives us access to the standard files connected to our program. STD_FILES has many features. The most important features used in the following programs are:

```
end_of_file: BOOLEAN
```

```
                -- Has end of file been reached on standard_input
                -- by the last read operation?

    read_character
                -- Read the next character from standard_input
                -- and make it available in last_character. Set end_of_file
                -- to True, if there are no more characters.
        require
                not end_of_file
         ...

    last_character: CHARACTER
                -- Character, read by the last call to read_character

    put_character (c: CHARACTER)
                -- Write `c' at end of default output.
         ...
```

The above is just a copy some text in the file `std_files.e` of the kernel library. Usually each feature in Eiffel is documented with a short *header comment* which describes what the feature does or returns.

The four described features show the usual command query separation. `read_character` is a command. It tries to read a character from standard_input. It makes the encountered character available in the query `last_character` or flags the end of file in the query `end_of_file` in case, that there are no more characters available on standard_input. The command `put_character` writes a character to `default_output` which by default is `standard_output`.

Calls to `put_character`, `put_string`, etc. can be interleaved; the output will appear in the order in which the calls are made.

The command `read_character` has the precondition

```
require not end_of_file
```

i.e. you are not allowed to call `read_character`, if the last read operation has already encountered the end of the input stream.

You can configure your eiffel system to monitor assertions. If your write in the ace-file of your program

```
root
     ...
default
     assertions(all)
cluster
     ...
end
```

all assertions like preconditions are monitored at runtime. This is a great aid in debugging programs. Once your program is mature and well tested, you can switch the monitoring of the assertions off by `assertions(no)` without any change in your program text.

Preconditions are part of Design by Contract which is extensively used in Eiffel programs. A precondition establishes a part of a contract between the client and the supplier of a feature which puts an obligation on the client (i.e. the caller).

- Obligation of the client: Only call a feature if you are sure that the precondition is met.

The other part of the contract can be specified in a postcondition, which puts an obligation on the supplier. Possible Design by Contract *assertions* are *preconditions*, *postconditions*, *class invariants*, *loop invariants*, *loop variants* and *checks*. More on Design by Contract later.

File copying

Given just character input output a lot of useful programs can be written without knowing anything more about input and output.

The first program just copies all characters from input to output.

```
class
	COPY
create
	make
feature
	make
		do
			from
				io.read_character
			until
				io.end_of_file
			loop
				io.put_character (io.last_character)
				io.read_character
			end
		end

end
```

The program is self documentary. In the loop we try to read the first character from the input stream. The exit condition `io.end_of_file` checks, whether the end of the stream has been reached.

As long as the end has not yet been reached, the last read character is written by `io.put_character(io.last_character)` to the output stream.

The exit condition guaratees, that we satisfy the precondition of `read_character` i.e. that we never try to read beyond the end of the stream.

Character counting

A slight modification of the copy programs gives us a program, which counts the number of characters in the input stream.

```
class
	CHAR_COUNT
```

```
create
	make
feature
	make
		local
			nc: INTEGER -- number of characters
		do
			from
				io.read_character
			until
				io.end_of_file
			loop
				nc := nc + 1
				io.read_character
			end

			io.put_string  ("number of characters: ")
			io.put_integer ( nc )
			io.put_new_line
		end
end
```

Instead of copying the read character to output, we increment the counter `nc`. At the end we output the number of characters encountered.

Most types in Eiffel have reasonable default values. All variables of type INTEGER are initialized with 0. Therefore it is not necessary to initialize `nc`.

Unlike C Eiffel does not have an increment operator. You have to write `nc:=nc+1` to increment `nc`.

Arrays and objects

In order to demonstrate the use of arrays and objects we write a program to count number of occurences of each digit, the number of encountered white spaces and other characters in the input.

There are twelve categories of input. In order to store the occurences of each digit we use an array of integers instead of an indiviual variable for each digit.

```
class COUNT_DIGITS create make feature
	make
		local
			ndigit:          ARRAY[INTEGER]
			nwhite, nother:  INTEGER
			c:               CHARACTER
			i:               INTEGER
		do
			create ndigit.make ( ('0').code, ('9').code )
					-- create array object

			from io.read_character until io.end_of_file loop
				c := io.last_character
				if '0' <= c and c <= '9' then
					i := c.code
					ndigit[i] := ndigit[i] + 1
```

```
                        elseif c = ' ' or c = '%N' or c = '%T' then
                                nwhite := nwhite + 1
                        else
                                nother  := nother + 1
                        end
                        io.read_character
                end

                io.put_string ("digits = ")
                from i := ('0').code until i > ('9').code loop
                        io.put_character (' ')
                        io.put_integer   ( ndigit[i] )
                        i := i + 1
                end
                io.put_string (", white space = "); io.put_integer( nwhite )
                io.put_string (", other = ");       io.put_integer( nother )
                io.put_new_line
        end
end
```

The output of the program on itself is something like

```
digits =  5 5 0 0 0 0 0 0 0 2, white space = 298, other = 58
```

The declaration

```
ndigit: ARRAY[INTEGER]
```

declares the variable `ndigit` to be an array of integers. The size of arrays in Eiffel is given at runtime and not at compile time. The statement

```
create ndigit.make(('0').code,('9').code)
```

creates an array object with the character code of '0' as the lower index and the character code of '9' as the upper index, i.e. an array of size 10. But instead of explaining the features of ARRAY lets have a look at the corresponding declaration in the source file array.e.

```
class ARRAY[G] ... create make ... feature ...

    make (l,u: INTEGER)
            -- Create an array with the lower bound `l' and        upper bound `u'.
        -- In case of u < l the array is empty

    lower: INTEGER
            -- The lower bound of the array index.

    upper: INTEGER
            -- The upper bound of the array index.

    count: INTEGER
        -- Number of elements in the array.

     item alias "[]" (i: INTEGER): G
        -- The i-th element of the array.
        require
            lower <= i and i <= upper
```

```
            ...
            end

      put (v: G; i: INTEGER)
            -- Put `v' at position `i' of the array.
            require
               lower <= i and i <= upper
            ...
            end
end
```

Looking at this, it should be quite clear what the array statements in the program mean.

The class ARRAY is a generic class with the generic parameter `G`. You can use any type for `G` to declare an array. The following are valid array declarations:

```
a1: ARRAY[CHARACTER]
a2: ARRAY[INTEGER]
a3: ARRAY[ARRAY[INTEGER]]
```

The last one declares an array of arrays. However

```
a1: ARRAY[ARRAY]     -- invalid declaration
```

is invalid. Now we can understand the differences between classes and types. ARRAY is a class and ARRAY[INTEGER] is a type. For non generic classes the class name denotes a class and a type at the same time.

The feature `item` is declared with the alias "[]". That means that instead of writing `ndigit.item(i)` you can use the shorthand `ndigit[i]`.

The alias mechanism is also used for the basic types like INTEGER. E.g. in the source of the class INTEGER your will find the declaration (well, not exactly, but in principle)

```
plus alias "+" (other: INTEGER): INTEGER
```

i.e. the expression `a + b` is just a shorthand (an *alias*) for `a.plus(b)` which calls the feature `plus` on the object `a` (or target `a` in Eiffel speak) with the argument `b`.

Now back to the digit counting program. The loop of the program reads one character at a time. It has to be decided whether the character is a digit, whitespace or anything else. In order to do this we use a conditional statement. It has the general form:

```
if condition_1 then
   compound_1
elseif condition_2 then    -- zero or more elseif parts
   compound_2
...
else                       -- optional else part
   compound
```

```
end
```

> **Note**: The keyword `elseif` does not have any embedded blank! Due to the use of the keywords `if`, `then`, `elseif` and `end` no parentheses are necessary to delimit the conditions.

A compound is any sequence of valid Eiffel statements. The conditional statement behaves in the same manner as conditional statements in other languages like C, java, etc.

Characters can be compared with the usual relational operators. Each character has a code (usually the ascii code). The class CHARACTER has the query `code` which returns the corresponding character code. The condition `'0' <= c and c <= '9'` tests if `c` is a digit.

To denote special characters like newline etc. Eiffel character constants can be written with escape sequences '%N' for newline and '%T' for tab.

A special provision is necessary for using features on constants (and also on operator expressions). The character constant '0' is an expresssion of type CHARACTER. Therefore all features of the class CHARACTER can be called with objects which are given as character constants.

However, it is not possible to write `'0'.code` because this would lead to some ambiguities in parsing the language. In order to use a constant as an object (or a target) for a feature call, it has to be parenthesized with (| and |), i.e. `(|'0'|).code` denotes the character code of the character '0'. The construct `(|expression|)` is called a *parenthesized target*.

Although the program is a little bit artificial (who wants to count the digits in a file?), we will write a different version of the program to demonstrate more Eiffel techniques.

In the above program the decision whether a character is a digit, a whitespace or any other is done in the conditional statement. Since there are only 256 different characters (at least as long we do not consider unicode characters), we could use an array to make this decision.

The key idea is to use an array of size 256. Each array element references a counter. The three whitespace entries for blank, tab and newline shall reference the same counter. Each entry for a digit shall reference its corresponding digit counter and all other entries shall reference a counter for the other characters.

It is quite easy to design a class for the counter object:

```
class COUNTER_OBJECT feature
       value: INTEGER
       increment do value := value + 1 end
invariant
       value >= 0
end
```

In Eiffel a class (and also the corresponding type) has either copy or reference semantics. The COUNTER_OBJECT class has reference semantics. The class INTEGER has copy semantics. The class INTEGER is declared like

```
expanded class INTEGER ... end
```

with the keyword `expanded` to declare a class with copy semantics. The difference between copy and reference semantics is important for assignment, argument passing and comparison with the operator =.

Objects with copy semantics are copied in assignment and argument passing (call by value). The comparison operator = compares the content (i.e. the value) of the objects.

Objects with reference semantics are not copied during assigment and argument passing (call by reference), just a reference to the object is copied from the source to the target. The comparison operator = only evaluates true, if the left hand side and the right hand side of the comparison reference the same object.

If you want to compare the equality (i.e. same content) of two objects with reference semantics, you have to use the equality operator ~. On expanded type objects the comparison operators = and ~ give identical results.

In COUNTER_OBJECT we declared the class invariant

```
  invariant
       value >= 0
```

A class invariant can be declared at the end of the class (beyond the last feature block). It is a consistency condition. It states that before and after each feature call the consistency condition has to be satisfied.

For this small class COUNTER_OBJECT, the invariant does not give us a lot. But it states clearly our design intention that a counter has a non negative value.

In classes with many attributes writing a class invariant is very helpful. By extending the classes (adding more features or making the features more powerful) you might forget to satisfy the invariant. Switching assertion monitoring on allows the runtime to remind you of the invariant condition by giving you a strong message about the violation.

With the COUNTER_OBJECT class the modified program digit count program can be written easily. We give first the layout:

```
class COUNT_DIGITS2 create make feature {NONE}

       white_counter, other_counter:   COUNTER_OBJECT
       char_counter:                   ARRAY[COUNTER_OBJECT]

       make
              do
                     initialize
                     read_input
                     write_statistics
              end

       initialize
              ...
```

```
        read_input
                ...
        write_statistics
                ...
end
```

Since the program is a little bit longer, we split for better readability and maintainability the code over the three different routines `initialize`, `read_input` and `write_statistics`.

The procedure `initialize` initializes the counter objects and the arrays, `read_input` scans the input and fills the counters apropriately and `write_statistics` gives as the expected output at the end of the program.

The three routines must have access to the counters. Therefore we put the counters into attributes. This avoids argument passing.

The code for `initialize` looks like:

```
        initialize
                local
                        co:   COUNTER_OBJECT
                        i:   INTEGER
                do
                        create white_counter; create other_counter
                        create char_counter.make  (0,255)

                        from i:=0 until i=256 loop
                                char_counter[i] := other_counter
                                i := i + 1
                        end

                        from i:= ('0').code until i = ('9').code + 1 loop
                                create co
                                char_counter[i] := co
                                i := i + 1
                        end

                        char_counter[('%N').code] := white_counter
                        char_counter[('%T').code] := white_counter
                        char_counter[(' ').code] := white_counter
                ensure
                        char_counter.count  = 256
                end
```

There is nothing really surprising here. It makes a straight forward initialization of the counter objects. The postcondition states, that the array `char_counter` is properly initialized. The following routines can rely on this property.

With that done, the routine `read_input` really gets very simple

```
        read_input
                require
                        char_counter.count  = 256
                local
                        c:   CHARACTER
                do
```

```
                from
                        io.read_character
                until
                        io.end_of_file
                loop
                        c := io.last_character
                        char_counter[c.code].increment
                        io.read_character
                end

        end
```

On each read character `c` it just retrieves a reference to its counter object by `char_counter[c.code]` and calls the feature `increment` on the counter object.

The precondition states that the routine expects the array `char_counter` properly initialized. If the precondition were not satisfied, `read_character` could access the array `char_counter` out of bounds.

Now writing the routine `write_statistics` is nothing more than a piece of cake.

```
    write_statistics
            require
                    digit_counter.count = 10
            local
                    i: INTEGER
            do
                    io.put_string ("digits = ")
                    from i := 0 until i = 10 loop
                            io.put_character (' ')
                            io.put_integer   ( digit_counter[i].value )
                            i := i + 1
                    end
                    io.put_string ( ", white space = ")
                    io.put_integer( white_counter.value )
                    io.put_string (", other = ")
                    io.put_integer( other_counter.value )
                    io.put_new_line
            end
```

Functions

Up to now we only have written procedures. Remember that the general term is *routine*. From a user perspective routines are commands. The other category from the user perspective are *queries* (features that return a result, i.e. give an answer to a question). Queries can either be implemented as attributes or functions.

We are going to write a *function* which calculates the factorial. Remember the mathematical definition

```
n! =  1,          if n = 0
n! =  n * (n-1)!, if n > 0
```

In Eiffel you can write recursive functions. Since the definition is recursive, it is easy to implement it by a recursive function.

```
fac (n: INTEGER): INTEGER
        require
                n >= 0
        do
                if n = 0 then
                        Result := 1
                else
                        Result := n * fac ( n - 1 )
                end
        end
```

Any function in Eiffel has an implicitely declared local variable with name `Result`. There is no necessity to declare it. The compiler does it for you. The type of `Result` is the return type of the routine. In the routine you have to assign to `Result` (or create `Result`), nothing more. Whatever has been assigned to `Result` will be returned to the caller of the function.

Note: Functions can have zero or more arguments. Nothing prevents you from defining a functions like

```
five: INTEGER do Result := 5 end

array_of_10_ints: ARRAY[INTEGER] do create Result.make (0,9) end
```

The user does not know, that `five` and `array_of_10_ints` are functions. For the user, they are argumentless queries, indistinguishable from attributes. This is the principle of *uniform access*. The implementer can decide to implement an argumentless query as a function or an attribute without affecting any client code.

For those who don't like recursive functions, we give also an iterative version of `factorial`

```
factorial_iterative (n: INTEGER): INTEGER
        require
                n >= 0
        local
                i: INTEGER
        do
                from Result:=1; i:=0 until i = n loop
                     i      := i + 1
                     Result := i * Result
                end
        end
```

Stylistic note: Eiffel does not require semicolons as statement terminators or separators. But they are allowed. The syntax defines all semicolons as optional. In the above function, we could have written `Result:=1 n:=i` leaving out the semicolon. However it is good practice for readability to use the semicolons if you write more than one statement on line.

The recursive version of the function is easy to verify because it is just the

mathematical definition transcribed to Eiffel syntax. To verify the iterative version requires some thinking. Are the loop bounds correct? Is "one too few iterations" or "one too many iterations" possible? Alltough the loop is not very complicated, let us try to verify the loop a little bit more formally and learn more Eiffel techniques.

The key idea is, that Result always contains i!. We start the loop with `Result=1` which is by definition 0!. I.e. at the start of the loop `Result=i!` is satisfied.

In each iteration we increment `i` by one and assign to `Result` the value `i*Result`, i.e. `i*(i-1)!`. Therefore if `Result=i!` is valid at the start of the loop body, it is also valid at the end of the loop body.

We call a condition, which is true at the start and at the end of the loop body a *loop invariant*.

Up to now, we have convinced ourselves, that `Result=i!` is a loop invariant.

At the end of the loop we know that the exit condition `i=n` is true. Therefore at the end of the loop we have

```
i = n  and Result = i!
```

which is identical to

```
Result = n!
```

Eiffel allows us to specify loop invariants. Since we have already the very reliable recursive function `fac`, we can write the invariant completely in Eiffel.

```
factorial_iterative ( n: INTEGER): INTEGER
        require
                n >= 0
        local
                i: INTEGER
        do
                from i:=0; Result:=1 invariant
                        0 <= i and i <= n
                        Result = fac ( i )
                until
                        i = n
                loop
                        i       := i + 1
                        Result := i * Result
                variant
                        n - i
                end
        end
```

We have added the trivial invariant condition, that `i` loops between `0` and `n`. You can use the assertion monitoring facilities of Eiffel to check the loop invariants. If you write in the corresponding ace-file "`default assertions (all)`", the loop invariants are monitored. A violated loop invariant will be flagged by the Eiffel runtime.

In the above program we have also added a *loop variant*. This is a facility to detect infinite loops. A variant is a non negative integer expression. It has to decrement at least by one on each iteration of the loop. The variant is an upper bound of the number of the remaining iterations. Since `i` loops from `0` to `n`, the remaining iterations are `n-i`.

In assertion monitoring mode, the Eiffel runtime checks on each loop iteration that the variant is non negative and that it decrements at least by one on each iteration. If this is not the case, the runtime flags a violated loop variant.

More focus on classes

Up to now we have only created root classes for programs and used some classes of the kernel library. We were focussing on the algorithmic aspects by mainly writing procedures with control structures like loops and alternative commands. But the real power of Eiffel is its possibility to make very different kind of classes and combine them.

The examples in this section will show some different uses of classes. The first one demonstrates some possibilities to use inheritance, the second shows how you can use genericity and the third one allows you to make classes to represent e.g. complex numbers.

A rectangle is a sort of a shape

In graphics we want to deal with graphical objects like rectangles, circles, etc. We are going to call these graphical objects shapes.

There are some common things to do with shapes. Shapes can be moved, displayed, put on top of other shapes. The code of a graphical program gets very cluttered if it has to distinguish in many places whether a shape is a rectangle or a circle etc.

Eifel allows us define an abstract class SHAPE with some common features without providing the implementation. The more specific classes like RECTANGLE inherit from SHAPE and have to define their specific implemention of the features.

In order to keep the example simple we define four abstract and one concrete feature in SHAPE. Let us look at the class text.

```
deferred class SHAPE feature
       x_left:  INTEGER  deferred end
       x_right: INTEGER  deferred end
       y_lower: INTEGER  deferred end
       y_upper: INTEGER  deferred end
       write_dimensions
             do  ...  end
invariant
       x_left  <= x_right
       y_lower <= y_upper
end
```

The four abstract features give the maximum extension of the shape in the x

and y dimension. There is no implementation for the features. Instead of the usual `do end` block we encounter a `deferred end` block. The implementation of the features is *deferred* to the descendants which are going to inherit from the class SHAPE.

`x_left`, `x_right`, `y_lower` and `y_upper` are called *deferred* features.

No objects of type SHAPE can be created, because such an object would have undefined features. A class with deferred features is itself deferred. This has to be written into the class header. Therefore we have written `deferred class SHAPE` instead of just `class SHAPE`. It is a language rule that every class which has deferred features has to be tagged with the keyword `deferred` in the class header.

You could argue, that the keyword `deferred` in the class header is redundant, because the compiler already knows from the features, that the class is deferred. But the keyword `deferred` in the class header is required by the language in order to state clearly that the class is an abstract one.

Note also, that the class SHAPE does not have any creation procedure. A creation procedure would be meaningless, because it is no possible to create direct instances of an abstract class.

Allthough the class just declares four abstract features it can already state some properties of these features in the class invariant. Remember that the class invariant is a consistency relation of the features of the class.

The class SHAPE puts the requirement on all its descendants that they satisfy this consistency realation i.e. that they satisfy the class invariant. Beside all other features descendants inherit the class invariant as well.

The class invariant is an assertion, which can be monitored at runtime. The class invariant has to be satisfied after creation of an object and before and after the execution of any publicly available feature. This gives a strong guarantee that any modificating routine will not violate its invariant.

With four for abstract features `x_left`, `x_right`, `y_lower` and `y_upper` it is possible to write the procedure `write_dimensions` which e.g. writes the x and y dimensions of the shape to standard output. The procedure `write_dimensions` is an effective procedure. The implemention is not spelled out completely above because it is straightforward. It could be written e.g. like

```
write_dimensions
        do
                io.put_string  ("shape with dimensions x = ")
                io.put_integer (x_left)
                io.put_string  ("..")
                io.put_integer (x_right)
                io.put_string  (" and y = ")
                io.put_integer (y_lower)
                io.put_string  ("..")
                io.put_integer (y_upper)
                io.put_new_line
        end
```

As you see the routine `write_dimensions` can use the features `x_left`,

`x_right`, `y_lower` and `y_upper` even if they are only deferred features. The class SHAPE is sometimes called a *partial implementation*. It implements the feature `write_dimensions` but leaves the implementation of the deferred features to its descendants.

Now lets define a rectangle as a kind of a shape. The definition is straightforward, because a rectangle is defined by its left/right and lower/upper dimensions.

```
class
        RECTANGLE
inherit
        SHAPE
create
        make
feature
        x_left:  INTEGER
        x_right: INTEGER
        y_lower: INTEGER
        y_upper: INTEGER
feature {NONE}
        make ( x1, y1, x2, y2: INTEGER )
                        -- Make a rectangle with lower left corner (`x1',`y1')
                        -- and upper right corner (`x2',`y2').
                require
                        x1 <= x2
                        y1 <= y2
                do
                        x_left  := x1;    x_right := x2
                        y_lower := y1;    y_upper := y2
                end
end
```

A RECTANGLE inherits from the deferred class SHAPE the deferred features and has to effect them (i.e. provide an implementation for them). Redeclaring a deferred feature into an effective one is called *effecting* a feature in Eiffel terminology.

The class RECTANGLE has chosen to declare the deferred features as attributes.

Since RECTANGLE has redeclared all deferred features into effective ones, it is no longer a deferred class. In order to create objects of type RECTANGLE, the class RECTANGLE provides the creation procedure `make` which, given the coordinates of the lower left and the upper right corner, initializes its attributes properly.

In order to satisfy the class invariant of its parent SHAPE, the creation procedure `make` puts a precondition on the coordinates it receives. If called with arguments satisfying its precondition, it can guarantee that the rectangle fullfils the class invariant imposed by its parent SHAPE.

We made the creation procedure `make` secret because we put the procedure in a feature block with the specification `feature {NONE}`. This means, that the procedure cannot be called as a normal procedure. It is only available for creation because it is listed in the class header as a creation procedure. This is a practive often used with creation procedures in order to allow them to be used only for creating objects and for nothing else.

But nothing prevents you from making the creation procedure public. If you do this, you must be sure, that calling the procedure at any time in the lifecycle of an object does not do any harm.

Another kind of shape is a circle. A circle e.g. can be defined by the coordinates of its center point and by its radius. Therefore it makes sense for a circle to have the attributes `x_center`, `y_center` and `radius`.

With these attributes it is easy to calculate the outer dimensions `x_left`, `x_right`, `y_lower` and `y_upper`. In the class CIRCLE these features will not be attributes like in rectangle, they will be redeclared into routines or more specifically functions (remember that a function is a routine which returns a value and a procedure is a routine which does not return a value).

The class CIRCLE could be defined like

```
class
	CIRCLE
inherit
	SHAPE
create
	make
feature
	x_center: INTEGER
	y_center: INTEGER
	radius:   INTEGER

	x_left:  INTEGER  do  Result := x_center - radius  end
	x_right: INTEGER  do  Result := x_center + radius  end
	y_lower: INTEGER  do  Result := y_center - radius  end
	y_upper: INTEGER  do  Result := y_center + radius  end

feature {NONE}
	make ( x, y: INTEGER; r:INTEGER )
			-- Make a circle with center coordinates (`x',`y')
			-- and radius `r'.
		require
			r >= 0
		do
			x_center  := x
			y_center  := y
			radius    := r
		end

end
```

The two classes RECTANGLE and CIRCLE have chosen two different techniques to redeclare the deferred features of its parent SHAPE into effective ones. RECTANGLE has used attributes and CIRCLE has used functions. Both possibilities are valid in Eiffel. The complete implementation is deferred to the descendant. This includes the decision between a memory based (attribute) and computation based (function) implementation.

With the classes RECTANGLE and CIRCLE it is possible to create rectangle and circle objects. Because RECTANGLE and CIRCLE inherit from SHAPE, it is possible e.g. to assign a variable of type RECTANGLE to a variable of type SHAPE. We say that RECTANGLE conforms to SHAPE.

```
        local
             s: SHAPE
            r: RECTANGLE
            c: CIRCLE
        do
             create r.make (0,0, 10, 20)
            create c.make (5,5, 30)

            s := r    -- possible because RECTANGLE conforms to SHAPE
            s.write_dimensions  -- write the dimensions of `r'

            s := c
            s.write_dimensions  -- write the dimensions of `c'
        end
```

This example is rather naive, because the variables `r' and `c' could have been used directly. The technique gets more interesting if you define e.g. an array of shapes which contains all the different graphical objects in your system. For this purpose you can define a variable `objects` of type `ARRAY[SHAPE]` and insert all your graphical objects into this array.

Let us assume that you declared the some more deferred features define in class SHAPE

```
deferred class SHAPE feature
        ...
        wipe_out
             -- Wipe the object out.
           deferred end

        move (x, y: INTEGER)
             -- Move the object `x' to the left and `y' up.
           deferred end

        draw
             -- Draw the object.
           deferred end
        ...
end
```

and provided specific implementations in all the effective descendants.

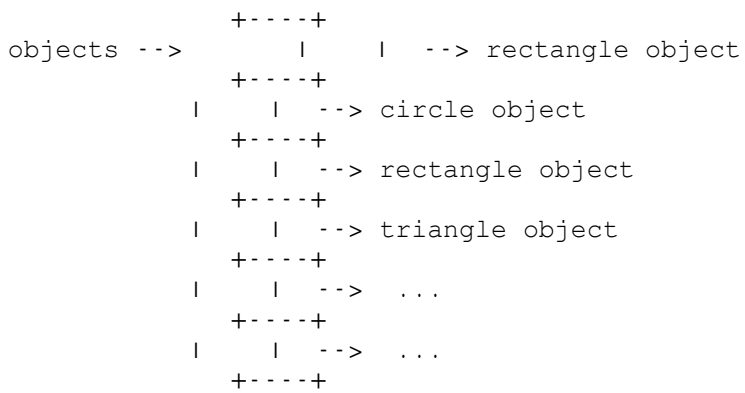
With these definitions it is easy to write a procedure which moves all graphical objects by a certain displacement.

```
class GRAPHICAL_SYSTEM feature
        ...
         objects: ARRAY[SHAPE]

         move_all (x, y: INTEGER)
             -- Move all objects in `objects' x to the right and `y' up.
            local
                  i: INTEGER
            do
              from i:=objects.lower until i > objects.upper loop
                   objects[i].wipe_out
                   objects[i].move (x,y)
```

```
                    objects[i].draw
                    i := i + 1
                end
            end
        ...
end
```

Since SHAPE is a reference class, each entry in the array `objects` is a reference to the corresponding objects.



Matrices as objects

A matrix is a rectangular scheme of numbers

```
          col 1       col 2      col 3

   row 1:    1           10          -5

   row 2:   -1            2          -7
```

The above matrix has 2 rows and 3 columns. The entries are integer numbers. With the bracket notation a[1,3] we address the entry in the first row and the third column.

We want to write `a.rows` to get the number of rows and `a.columns` to get the number of columns.

Two matrices can be added, substracted and multiplied.

For addition and substraction the two added matrices must have exactly the same dimensions. The sum `c=a+b` is calculated according to the formula

```
  c[i,j] = a[i,j] + b[i,j]
```

For the multiplication `a*b`, the number of columns of a has to be the same as the number of rows of b. The product `c=a*b` is calculated according to the formula

```
  c[i,k] = a[i,1]*b[1,k] + a[i,2]*b[2,k] +  ...  + a[i,n]*b[n,k]

        where n = a.columns = b.rows
```

Up to now this elementary mathematics.

For our Eiffel class MATRIX we want to be able to create matrix objects with a certain number of rows and columns. We don't want to write a matrix class for INTEGERs and one for REALs. We want to write the code of MATRIX only once. Eiffel has generic classes to achieve this. Omitting the details the outline the class looks like

```
class
        MATRIX[G->NUMERIC]
create
        make
feature {NONE}
        make (r, c: INTEGER)
                        -- Make a matrix with `r' rows and `c' columns
                   ...
                ensure
                        rows    = r
                        columns = c
                end
feature
        rows:    INTEGER   ...
        columns: INTEGER   ...

        is_valid_row ( i: INTEGER ): BOOLEAN
                do  Result := 1 <= i and i <= rows     end
        is_valid_column ( j: INTEGER ): BOOLEAN
                do  Result := 1 <= j and j <= columns  end

        item alias "[]" ( i, j: INTEGER ): G
                        -- The element at row `i' and column `j'.
                require
                        is_valid_row    ( i )
                        is_valid_column ( j )
                   ...
                end
        put ( el: G; i,j: INTEGER )
                        -- Put element `el' at row `i' and column `j'.
                require
                        is_valid_row    ( i )
                        is_valid_column ( j )
                ...
                ensure
                        item (i,j) = el
                end
        plus    alias "+" ( other: like Current ): like Current
                        -- The  sum `Current' + `other'.
                require
                        rows    = other.rows
                        columns = other.columns
                   ...
                end
        minus   alias "-" ( other: like Current ): like Current
                        -- The  difference `Current' - `other'.
                require
                        rows    = other.rows
                        columns = other.columns
                   ...
                end
        product alias "*" ( other: like Current ): like Current
                        -- The product `Current' * `other'.
                require
```

```
                    columns = other.rows
                ...
            end
feature {NONE} -- implementation
                ...
end
```

Several aspects of the Eiffel language are encountered in this outline:

1. Constrained genericity

class MATRIX[G->NUMERIC] states that MATRIX is a generic class. You can define a variable of type MATRIX[INTEGER] or MATRIX[REAL]. We don't want variables of type MATRIX[BOOLEAN], because you cannot do the elementwise calculation with booleans. Therefore we *constrain* the generic parameter G to be anything which *conforms to* NUMERIC.

NUMERIC is a class in the kernel library declaring the standard numeric operations like plus, minus, product etc. as deferred features. The classes INTEGER and REAL inherit from NUMERIC (i.e. they conform to NUMERIC) and implement the numeric operations.

2. Operator aliases

We don't want to write `a.plus(b)` to add the matrices a and b. We prefer the mathematical notation `a+b`. Each feature in an Eiffel class can be given an alias name. The alias has to be an operator or the bracket alias. This allows us to write `a+b`, `a-b` and `a*b`.

3. Bracket alias

To address an element of the matrix the notation `a[i,j]` is very concise and easy to read. It is preverable over `a.item(i,j)`. Each class can have at most one bracket alias. We have used the bracket alias as an alias for the feature `item`.

4. Anchored types

We could have defined the feature `plus` as

```
plus    alias "+" ( other: MATRIX[G] ):    MATRIX[G]
```

but instead of this we have written

```
plus    alias "+" ( other: like Current ): like Current
```

In the class MATRIX the types `MATRIX[G]` and `like Current` are equivalent, because the current type is `MATRIX[G]`.

Things become different if we want to define a `class SPECIAL_MATRIX inherit MATRIX ... end` which inherits all features of MATRIX and defines some additional features. In the class SPECIAL_MATRIX we want to add, substract and multiply objects of type SPECIAL_MATRIX[G] with objects of type SPECIAL_MATRIX[G] returning an object of type SPECIAL_MATRIX[G].

Anchoring the argument and the result of the operations `plus`, `minus` and `product` does exactly this. The type `like Current` is anchored to the current type. In SPECIAL_MATRIX the current type is SPECIAL_MATRIX[G] and not MATRIX[G].

We can anchor types to `Current` and to other features (which must be queries) of the class.

5. Design by Contract

As we have already seen in other examples preconditions and postconditions can be used document the allowed use and the expected properties of the outcome of routines.

E.g. the features `item` and `put` has to be protected againts its use outside the bounds of the matrix. The conditions `is_valid_row(i)` and `is_valid_column (j)` express this condition. The postcondition `item(i,j)=el` states that the feature `put` really does what it says, put the element `el' and position (i,j) of the matrix.

The preconditions of `plus`, `minus` and `product` protect us from combining matrices with incompatible row and column numbers.

The assertions play a very important role in Eiffel programming.

First of all they are a means to document your intentions. Every user of your routines can look at the header comment, the precondition and the postcondition to get a very precise picture of what your routine does without looking at the implementation.

Secondly they help you to debug your program. During the development of a program usually all types of assertions are monitored at runtime. If you put many assertions into your program a bug is usually caught very close to its origin. This speeds up debugging significantly.

Thirdly you can try to verify your program piece by piece. If you convince yourself, that a certain routine fulfills its postcondition given that the precondition has been met, you only have to check that each client calls a routine with a valid precondition. The assertions help you to reason about your program (like we have done in the above factorial example).

Now we have to find a proper implemention to store the elements of the matrix. The easiest way is to represent a row by an array with indices ranging from 1 to `columns`. The whole matrix is represented by an array of rows ranging from 1 to `rows`. If we define

```
feature {NONE} -- implementation
                matrix: ARRAY[ARRAY[G]]
end
```

we have with `matrix[i]` or `matrix.item(i)` the array representing the i-th row and with `matrix[i][j]` or in its long form `matrix.item(i).item(j)` the element at position (i,j).

The feature make initializes the matrix properly

```
feature {NONE}
	make (r, c: INTEGER)
			-- Make a matrix with `r' rows and `c' columns
		local
			i: INTEGER
			one_row: ARRAY[G]
		do
			create matrix.make (1, r) -- array of rows
			from i:=1 until i > r loop
				create one_row.make (1,c)
				matrix[i]  := one_row
				i := i + 1
			end
		ensure
			rows    = r
			columns = c
		end
```

With the dimensions of the attribute `matrix` we know the rows and the columns of the matrix implicitely. We implement the features `rows` and `columns` as functions which calculate the value by looking at `matrix`.

```
	rows: INTEGER
		 do Result := matrix.upper  end
	columns: INTEGER
		do
			if rows > 0 then
				Result := matrix[1].upper
			end
		end
```

The element access routines `item` and `put` are given by

```
	item alias "[]" ( i, j: INTEGER ): G
			-- The element at row `i' and column `j'.
		require
			is_valid_row    ( i )
			is_valid_column ( j )
		do
			Result := matrix[i][j]
		end
	put ( el: G; i,j: INTEGER )
			-- Put element `el' at row `i' and `j'.
		require
			is_valid_row    ( i )
			is_valid_column ( j )
		do
			matrix[i][j] := el
		ensure
			item (i,j) = el
		end
```

From the arithmetic routines `plus`, `minus` and `product` we present here only the last one. The others are left as an exercise to the reader.

```
	product alias "*" ( other: like Current ): like Current
			-- The product `Current' * `other'.
```

```
require
        columns = other.rows
local
        i,j,k: INTEGER
do
        create Result.make ( rows, other.columns )
        from i:=1 until i > rows loop
                from k:=1 until k > other.columns loop
                        from j:=1 until j > columns loop
                                Result[i,k] :=
                                    Result[i,k] + Current[i,j] * other[j,k]
                                j := j + 1
                        end
                        k := k + 1
                end
                i := i + 1
        end
end
```

Excercises:

- Write an invariant for the class MATRIX which asserts that all rows have the same length.
- Augment the class MATRIX with a feature `transposed` which returns the matrix with rows and columns exchanged (i.e. `a.transposed[i,j] = a[j,i]`). Write a postcondition which asserts that the correct properties for `rows` and `columns`.

Complex numbers

A complex number consists of the two real numbers, one representing the real part, the other representing the imaginary part. So the general outline for class COMPLEX could be

```
class COMPLEX feature
        ...
       real: REAL
       imag: REAL
       ...
end
```

But we want an object of type COMPLEX to behave like an number. If we defined it as above written, an object of type complex would be of reference type. So the statement

```
        c1 := c2
```

with `c1` and `c2` of type COMPLEX would make both variables pointing to the same object. This is not what we usually expect dealing with numbers. We expect that the value of `c2` is copied into `c1`, i.e. we want objects of type COMPLEX to have copy semantics instead of reference semantics. So it is better to define `expanded class COMPLEX` instead of `class COMPLEX`.

Furthermore the kernel library already provides the class NUMERIC which is an ancestor of the numeric classes INTEGER and REAL. The class NUMERIC defines

the some features to be expected from a number.

Looking at the definition of the class NUMERIC in the kernel library

```
deferred class NUMERIC feature
	zero: like Current
			-- Neutral element for addition.
		deferred  end
	one: like Current
			-- Neutral element for multiplication.
		deferred  end
	plus    alias "+" (other: like Current): like Current
			-- The sum `Current' + `other'.
		deferred  end
	minus   alias "-" (other: like Current): like Current
			-- The difference `Current' - `other'.
		deferred  end
	product    alias "*" (other: like Current): like Current
			-- The product `Current' * `other'.
		deferred  end
	divided alias "/" (other: like Current): like Current
			-- `Current' divided  by `other'.
		require
			good_divisor: divisible ( other )
		deferred
		end
	identity alias "+": like Current
		deferred  end
	negated alias "-": like Current
			-- The negated value of `Current'.
		deferred  end
	divisible ( other: like Current ): BOOLEAN
			-- May current object be divided by `other'?
		deferred  end
end
```

we convince ourselves, that all features can be implemented by our class COMPLEX. Classes like NUMERIC where most of the features are deferred are also called behaviour classes. The define a certain behaviour which all its descendants have to satisfy. So the user can expect that any class inheriting from NUMERIC implements the deferred features apropiately.

The task of implementing COMPLEX is to effect the deferred features of NUMERIC. With some standard mathematics we define

```
expanded class COMPLEX inherit NUMERIC create
	make, default_create
feature {NONE}
	make (r, i: REAL)  do  real := r; imag := i  end
feature
	real: REAL
	imag: REAL

	one: like Current          do create Result.make (1.,0.) end
	zero: like Current         do  end

	plus alias "+" ( other: like Current ): like Current
		do
```

```
                create Result.make ( real + other.real,
                                     imag + other.imag )
        end
    minus alias "-" ( other: like Current ): like Current
        do
                create Result.make ( real - other.real,
                                     imag - other.imag )
        end
    product alias "*" ( other: like Current ): like Current
        do
                create Result.make ( real * other.real - imag * other.imag,
                                     imag * other.real + real * other.imag)
        end
    divided alias "/" ( other: like Current ): like Current
        local
                a,b,c,d: REAL
                r, i:    REAL
                n:       REAL   -- denominator
        do
                a := real;        b := imag
                c := other.real;  d := other.imag

                r := a*c + b*d
                i := b*c - a*d
                n := c*c + d*d
                create Result.make ( r/n, i/n )
        end
    identity alias "+": like Current
        do
                Result := Current
        end
    negated alias "-" : like Current
        do
                create Result.make ( -real, -imag )
        end
    divisible ( other: like Current ): BOOLEAN
        do
                Result := other /~ zero
        end
end
```

Note that there are some subtleties in getting floating point arithmetic really correct. Since real number are represented as IEEE floating point numbers there is a positive and a negative zero. Both values represent the same number but they are not identical in the computer. The boolean expression `0. ~ -0.` will evaluate `False`. The feature `divisible` therefore gives an incorrect result, if the argument `other` contains a negative zero in the real or the imaginary part.

A correct implementation of divisible would compare the absolute value of `other` against some very small REAL value. But since we do not want to do numerical mathematics here we gloss over this issue.

But there is another interesting point here to mention. We have defined the two creation procedures `make` and `default_create`. The procedure `make` is used to initialize a complex number with its given real and imaginary part. The feature `default_create` is not defined in COMPLEX. It is an inherited feature of the kernel class ANY which does nothing.

Doing nothing is a valid creation procedure for COMPLEX. This at first glance surprising fact can be understood, if we realize, that COMPLEX has only two attributes of type REAL. The type REAL is self initializing, i.e. no explicit initialization means initialization with zero. So if we do nothing we just initialize the real and imaginary part with zero.

Due to this fact we can write the feature `zero` in a very short form.

```
zero: like Current          do  end
```

and not with the rather verbose definition

```
zero: like Current          do  create Result.make(0.,0.) end
```

Any not explicitely initialized variable `var` will be initialized by the system as if the instruction

```
create var.default_create
```

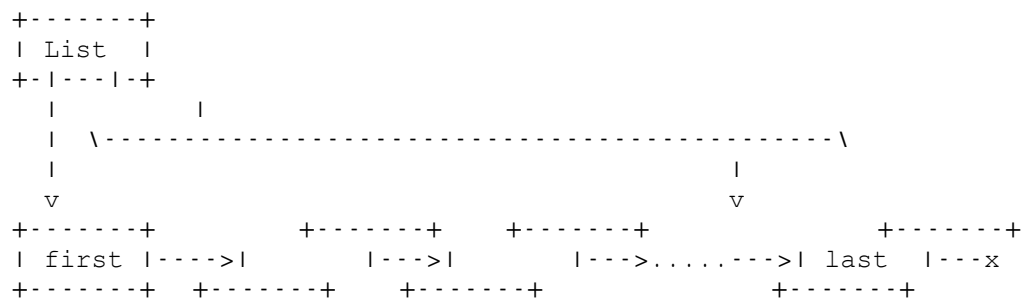
were written before its first use. But this is only possible if the variable is of a self initializing type, i.e. a type, which makes the feature `default_create` from any available for creation. This has been done with `create make, default_create` in the class COMPLEX. Therefore all objects of type COMPLEX are self initializing.

Note that our implementation gives us some additional benefit. We have implemented the class COMPLEX inheriting from the parent NUMERIC, i.e. objects of type COMPLEX conform to NUMERIC. Therefore we can use our class MATRIX from the previous chapter and define variables of type MATRIX [COMPLEX].

Linked lists

A linked list is a very basic data structure in information processing. It has the advantage, that the insertion of elements at both ends of the list are fast operations. However, linked lists have the disadvantage, that access to random elements can be expensive if the list is long.

We want to write a simple linked list. We have the following structure in mind.



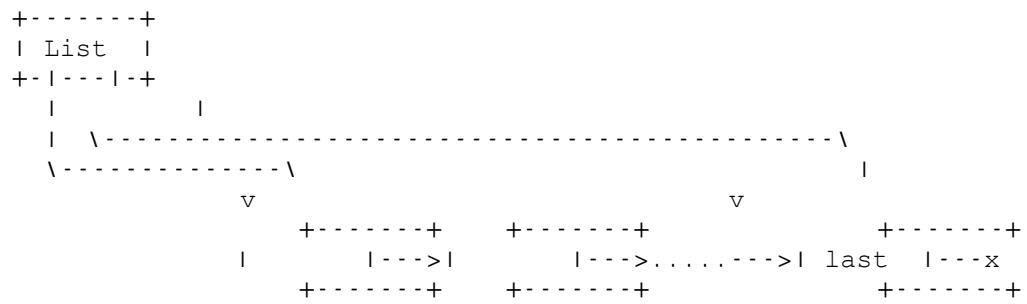
The list has a reference to the first cell and a reference to the last cell. The first cell has a reference to the second, the second to the third and soon. The last

cell has a void reference.

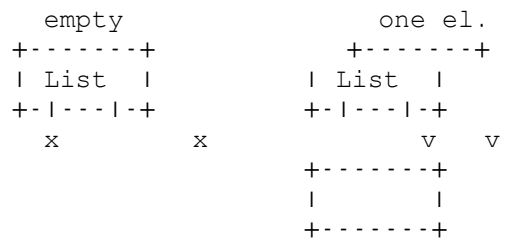
If the list is empty, the references to the first and the last element are void references.

The cells are containers of the elements, i.e. they consist of an element and a reference to the next cell.

It is very easy to remove the first element from the list. We simply let the reference to the first cell now point to the second cell.



Now let us look at the *boundary conditions* i.e. an empty list and a list with one element.



The design for the information cells is very simple.

```
class LINKABLE[G] create put feature
    item: G                -- Information element of the cell
    next: ?like Current    -- Link to the next cell
    put (el:G)
       do
         item := el
       end
    put_next (n: ?like Current)
       do
         next := n
       end
end
```

The only new thing we introduced here is the question mark in front the type `like Current`. The type `?T` is called a detachable type. By default all types in Eiffel are attached, i.e. the corresponding variables, expressions, etc. always have to be attached to objects. But for the linked list implementation we need types which can refer to nothing. Detachable types in Eiffel are good for that purpose.

If we have a variable `v` of type `?T`, we can write `v:=Void` or we can ask,

whether the variable is attached to an object by the boolean expression `v/=Void`.

For attached type variables the compiler doesn't allow that. On the contrary. For attached type variables the Eiffel compiler verifies that they are always attached to real objects. If your code does not make sure that attached type variables are really attached to objects, the compiler will flag you an error.

The skeleton of the class for the linked list is straightforward

```
class
   LINKED_LIST[G]
feature {NONE}
   first_linkable: ?LINKABLE[G]   -- Reference to the first cell
   last_linkable:  ?LINKABLE[G]   -- Reference to the last cell
feature
   is_empty: BOOLEAN
      do
       Result := first_linkable = Void
      end
   first: G
      ...
   last:  G
      ...
   extend_front ( el: G )
      ...
   extend_rear  ( el: G )
      ...
   remove_first
      ...
invariant
   (first_linkable = Void) =  (last_linkable = Void)
end
```

Our simple linked list class has two attributes which refer to the first and the last cell. They either both refer to nothing or they both refer to linkable cells. This property is expressed in the invariant of the class. The attachment status of both attributes has to be the same.

A naive attempt to write the function `first` results in the following invalid code:

```
   first: G              -- Invalid Eiffel code !!!
     require
       not is_empty
     do
       Result := first_linkable.item   -- Invalid !!!
     end
```

A valid Eiffel compiler does not accept the expression `first_linkable.item`. Since `first_linkable` is detachable, it can be a void reference and the call `first_linkable.item` can be a call with a void target.

However we are sure, that `first_linkable` refers to a real object, because of the precondition `not is_empty` for the function `first`. I.e. we allow a call to `first` only if the list is not empty. From our design above it is clear, that for a

non-empty list both `first_linkable` and `last_linkable` always refer to information cells.

But the compiler is not intelligent enough to infer that from the precondition. You have to tell it about the attachment status. We can assert that property by

```
first: G
  require
    not is_empty
  do
    check
      {l:LINKABLE[G]} first_linkable
    end
    Result := l.item  -- Now valid!
  end
```

The expression

```
{x:T} expr
```

is an *object test*. It is a boolean expression which checks if `expr` is attached to an object with a type conforming to `T`. If that is the case, the object test evaluates to true and the object will be attached to the *object test local* `x`. The boolean expression `{x:T} expr` has a side effect.

The above check assertion opens up a scope for the object test local `l`. The object test local `l` can be used until the end of the surrounding compound i.e. in this case until the end of the routine. An object test local is a read only variable. You cannot assign any new value to it.

Outside its scope it is not possible to use an object test local. E.g. the following would be invalid:

```
...
if condition then
   ...
   check {x:T} expr end
   ...
   x.some_feature       -- valid, because within scope
   ...
end
x.some_other_feature    -- invalid, because use of x outside scope
...
```

The function `last` is nearly a copy of the function `first`.

The procedure to insert an element before the first element is not difficult. We create a new information cell to contain the new element and let `first_linkable` refer to the cell. The new information cell has to refer to the old first information cell.

In order to get the code correct, we have take the possibility of an empty list into account.

```
extend_front (el:G)
```

```
        -- insert element `el' before the first element
    local
       new_cell: LINKABLE[G]    -- Attached type, because it is never void
    do
       create new_cell.put (el)

      if is_empty then
          -- case of the empty list
          first_linkable := new_cell
          last_linkable  := new_cell
       else
          -- case of the non-empty list
        new_cell.put_next (first_linkable)
        first_linkable := new_cell
      end
  ensure
      first = el
  end
```

In order to insert after the last element we again need an object test.

```
extend_rear (el:G)
        -- insert element `el' after the last element
    local
       new_cell: LINKABLE[G]    -- Attached type, because it never is void
    do
       create new_cell.put (el)

      if is_empty then
          -- case of the empty list
          first_linkable := new_cell
          last_linkable  := new_cell
       else
          -- case of the non-empty list
        check
            {l:LINKABLE[G]} last_linkable
        end
        l.put_next (new_cell)      -- let the previous last cell point
                                   -- to the new last cell
        last_linkable := new_cell -- let new_cell be the last cell
      end
  ensure
      last = el
  end
```

We need the object test `{l:LINKABLE[G]} last_linkable` to let the next pointer of the previous last cell point to the newly created last information cell. The check instruction asserts that `last_linkable` is attached. Note that the else part will only be entered if the list is not empty.

Exercises:

- Write the procedure `remove_first`.
- Write the procedure `remove_last`. Why is it more complex?
- Add an attribute `count:INTEGER` to the linked list which stores the number of elements in the list. Update the invariant and the features appropiately.
- Write a function `item alias "[]" (i:INTEGER):G` which returns the i-th

element. Assume that counting starts with zero.

- Can you modify the class LINKED_LIST without changing the class LINKABLE such that an iteration over all elements with a loop like the following guarantees a fast access to the elements of the list.

```
from i:=0 until i=list.count loop
  list[i].do_something
  i := i + 1
end
```



```
Local Variables:
mode: outline
coding: iso-latin-1
outline-regexp: "=\\(=\\)*"
End:
```