

THE AUSTRALIAN NATIONAL UNIVERSITY

First Semester 2001

COMP2100 (Software Construction)

Writing Period: 3 hours duration

Study Period: 15 minutes duration

*Permitted Materials: An unmarked paper
foreign-language dictionary*

Answer all questions

Your answers must be written in the boxes provided in this booklet. You will be provided with scrap paper for working, but only those answers written in this booklet will be marked. Do not remove this booklet from the examination room. There is additional space at the end of the booklet in case the boxes provided are insufficient. Label any answers you write at the end of the booklet to indicate which question they refer to. Do not write in red ink anywhere in the booklet.

Name (family name first):

Student Number:

Official use only:

Q1 (20)	Q2 (20)	Q3 (20)	Total (60)

QUESTION 1 [20 marks]

- (a) Explain what an *assignment attempt* is in Eiffel.

QUESTION 1(a)	[1 mark]

- (b) Explain the difference between a *once function* and an ordinary function in Eiffel.

QUESTION 1(b)	[1 mark]

- (c) Give an example of a situation in which there would be an advantage to using a once function rather than an ordinary function.

QUESTION 1(c)	[1 mark]

- (d) Give two reasons why Eiffel is preferable to Bash for writing large programs.

QUESTION 1(d)	[1 mark]

- (e) Give two characteristics of tasks that are more suitable to solution using a scripting language like Bash rather than an object-oriented language like Eiffel.

QUESTION 1(e)	[1 mark]

- (f) Give two possible reasons why a procedural language like C might be preferable to an object-oriented language like Eiffel in some situations.

QUESTION 1(f)	[1 mark]

- (g) Give one possible use for inserting a string like "\$Revision: 1.1 \$" into a program maintained using RCS.

QUESTION 1(g)	[1 mark]

- (h) Give two examples of common problems with poorly designed graphical user interfaces.

QUESTION 1(h)	[1 mark]

(i) What is a *callback*?

QUESTION 1(i)	[1 mark]

(j) Outline the roles of and the interactions between the three components of the Model-View-Controller architecture for graphical applications.

(i) What is the role of the *Model*? How does it interact with the View and the Controller?

QUESTION 1(j)[i]	[1 mark]

(ii) What is the role of the *View*? How does it interact with the Model and the Controller?

QUESTION 1(j)[ii]	[1 mark]

(iii) What is the role of the *Controller*? How does it interact with the Model and the View?

QUESTION 1(j)[iii]	[1 mark]

(l) Explain the difference between these expressions in the C programming language:

1. `a = b`
2. `a == b`

QUESTION 1(l)	[1 mark]

(m) Explain the difference between a function *declaration* and a function *definition* in C.

QUESTION 1(m)	[1 mark]

(n) The following C function is supposed to swap the values of two integer variables.

```
void swap(int x, int y){  
    int tmp = x;  
    x = y;  
    y = tmp;  
}
```

Explain why this will not work, and how to fix it.

QUESTION 1(n)	[1 mark]

(o) Explain why early defect detection and removal are important.

QUESTION 1(o)	[1 mark]

QUESTION 2 [20 marks]

- (a) This question is about expression trees that can be used for evaluating and printing simple arithmetic expressions. This is similar to Expression Version 3 from lectures, and to the structure used for the second part of Lab 3.

A designer has written the following deferred class. Each different type of expression will implement this interface, so that client code can build and traverse a polymorphic tree structure representing any arithmetic expression.

```
deferred class
    EXPRESSION

    feature {ANY} -- Queries

        value: INTEGER is
            -- The value.
            deferred
            end

        infix: STRING is
            -- String representation in infix order.
            deferred
            end

        postfix: STRING is
            -- String representation in postfix order.
            deferred
            end

    end -- class EXPRESSION
```

So far, only one effective descendant of class *EXPRESSION* has been completed, for handling integer constant expressions.

```
class
    CONSTANT

    inherit
        EXPRESSION

    creation
```

```

make

feature {NONE} -- Private features

make (v: INTEGER) is
    -- Initialise 'value' to 'v'.
    do
        value := v
    end

feature {ANY} -- Queries

value: INTEGER
    -- The value.

infix: STRING is
    -- String representation in infix order.
    do
        Result := value.to_string
    end

postfix: STRING is
    -- String representation in postfix order.
    do
        Result := value.to_string
    end

end -- class CONSTANT

```

(i) Write a new effective class *MULTIPLICATION* which inherits from class *EXPRESSION* and which is used to represent products of other expressions.

- Each object of class *MULTIPLICATION* must store its left and right sides in private attributes of type *EXPRESSION*.
- The creation routine *make* must take two arguments of class *EXPRESSION*, representing the left and right hand sides, and initialise the object appropriately.
- The *value* feature must be implemented as a recursive function that returns the product of the values of the two subexpressions.
- The *infix* function must return a string representing the current expression in infix or algebraic notation, in the form ‘(<left> * <right>)', where ‘<left>’ and ‘<right>’ are the infix representations of the left and right subexpressions respectively.

(The parentheses are important, but do not include the apostrophes, the angle brackets, or the words ‘left’ or ‘right’ in your output.)

Assume that another developer has written classes *ADDITION* and *SUBTRACTION*, also inheriting from class *EXPRESSION*, and which can be used to represent sums and difference of expressions respectively.

- (ii) Draw a diagram representing the tree structure corresponding to the expression $(a + ((b - c) * d))$.

QUESTION 2(a)[ii]

[1 mark]

(b) Consider the following excerpt from a version of class *URL* of the webget program from Assignment 4. It contains an attempted implementation of the feature *make_relative*. It contains several defects. These range from violations of the coding standard to syntax errors, simple logic errors and more profound or subtle logic errors.

The effect of a call on *make_relative* is supposed to be that it makes the current object identical to *base*, but with the *path* modified according to the value of the string *s*.

```
1     make_relative (s: STRING; b: URL) is
2         -- Builds this URL object using 'string' as an
3         -- offset relative to 'base'.
4     require
5         not s.empty or else s /= Void
6         base.is_valid
7     do
8         copy (b)
9         if s.item (1) = '/' then
10            path := clone (s)
11        else
12            from
13                remove_last_word_from (path)
14            until
15                not (s.has_prefix ("../") or path.is_empty)
16            loop
17                s.remove_prefix ("../")
18                remove_last_word_from_path
19            end
20            path.append (s)
21        end
22        is_valid := is_http implies not path.has_string ("..")
23    end
24 end

25
26 feature {NONE} -- Implementation
27
28     remove_last_word_from_path is
29         -- Remove the last '/' and everything
30         -- after it from 'path'.
31     do
32         from path.remove_last until path.last = '/' loop
33             path.remove_last
34         end
35     end
```


Excerpts from the short form of class *STRING*.

```
is_empty: BOOLEAN
  -- Has string length 0?
empty: BOOLEAN
  -- Has string length 0?
item (index: INTEGER): CHARACTER
  -- Character at position 'index'.
  require
    valid_index(index)
has_prefix (p: STRING): BOOLEAN
  -- True if prefix of Current is 'p'.
  require
    p /= Void
append (other: STRING)
  -- Append 'other' to Current.
  require
    other /= Void
remove_last (nb: INTEGER)
  -- Remove 'nb' last characters.
  require
    0 <= nb;
    nb <= count
  ensure
    count = old count - nb
remove_prefix (s: STRING)
  -- Remove the prefix 's' of current string.
  require
    has_prefix(s)
  ensure
    old count = count + s.count
first: CHARACTER
  require
    not empty
last: CHARACTER
  require
    not empty
```

QUESTION 3 [20 marks]

- (a) This question is about class *FIXED_ARRAY_OF_INTEGER* from Lab 8. This class uses the following C code, from file *array.c* in its implementation.

```
#include <stdlib.h>

/* Return a pointer to the first element of enough memory to
   store n integers. */
int *new_array(int n){
    return (int *)calloc(n, sizeof(int));
}

/* Return the element at index i of the array a. */
int c_item(int a[], int i){
    return a[i];
}

/* Insert the value x at index i of array a. */
void c_put(int a[], int x, int i){
    a[i] = x;
}
```

Here is an incomplete implementation of class *FIXED_ARRAY_OF_INTEGER*. Fill in the implementations of the missing features.

Hint: Feel free to do these parts in any order. It may be easier to write the wrapper routines first in 3(a)[iv], then come back and fill in the implementations of *make*, *item* and *put* in parts 3(a)[i] – 3(a)[iii].

```
class
    FIXED_ARRAY_OF_INTEGER
        -- Non-resizable arrays of integers, implemented in C.

    creation
        make

    feature {NONE} -- Creation
```

```

make (lo, hi: INTEGER) is
  -- Create with indices running from 'lo' to 'hi'.
  require
    lo <= hi
  do

```

(i) Fill in your implementation of *make* here.

QUESTION 3(a)[i]	[2 marks]

```

  ensure
    storage /= Void
    lower = lo
    upper = hi
  end

```

```

feature -- Queries

```

```

  lower: INTEGER
    -- The lowest valid index.

```

```

  upper: INTEGER
    -- The highest valid index.

```

```

  valid_index (index: INTEGER): BOOLEAN is
    -- Is 'index' allowed?
    do
      Result := (lower <= index) and (index <= upper)
    ensure
      Result = (lower <= index) and (index <= upper)
    end

```

```

  count: INTEGER is
    -- Number of elements stored.
    do
      Result := upper - lower + 1
    end

```

```

item (index: INTEGER): INTEGER is
  -- The value stored at 'index'.
  require
    valid_index (index)
  do

```

(ii) Fill in your implementation for *item* here.

QUESTION 3(a)[ii]	[2 marks]

```

end

```

```

feature -- Commands

```

```

  put (element: INTEGER; index: INTEGER) is
    -- Put 'element' at position 'index'.
    require
      valid_index (index)
    do

```

(iii) Fill in your implementation of *put* here.

QUESTION 3(a)[iii]	[2 marks]

```

  ensure
    item (index) = element
  end

```

```

feature {NONE} -- Private attribute

```

```

  storage: POINTER
  -- Location of C array.

```

```

feature {NONE} -- External calls.

```


(c) Your final task is to write a Makefile that automates the process of compilation and testing. The source files are

- The Eiffel class files `fixed_array_of_integer.e` and `test_array.e`
- The C file `array.c`
- The test data files `test*.in` and `test*.out`
- The Bash script `run_tests`

There is no header file.

Targets are

- the object file `array.o`
- the executable program `test_array`
- the test report `report.txt` which should be the output of running the Bash script from the previous part.

(i) Draw a diagram showing the dependencies between all the files in the system just described.

QUESTION 3(c)[i]	[2 marks]
------------------	-----------

Additional answers to QUESTION —(—)[—]