

THE AUSTRALIAN NATIONAL UNIVERSITY

First Semester 2004

**COMP2100
Software Construction**

Writing Period: 3 hours

Study Period: 15 minutes

Permitted Materials: None

Answer all questions

*Write your answers in the boxes provided in this booklet. Only those answers written in this booklet will be marked. 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 remove this booklet from the examination room. **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)	Q4 (20)	Q5 (20)	Total (100)

QUESTION 1 [20 marks]

- (a) Suppose that class *CHILD* inherits from class *PARENT*, and that we have made the declarations *child: CHILD* and *parent: PARENT*. What will be the outcome of the assignment attempt *child ?= parent* if:

- (i) *parent* is attached to an object of class *PARENT*?

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

- (ii) *parent* is attached to an object of class *CHILD*?

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

- (iii) *parent* is Void?

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

- (b) Give two reasons for keeping a file under version control using a system such as RCS.

QUESTION 1(b)	[2 marks]

- (c) Explain the role of the routines *signal_connect* and *execute* and of the class *GTK_COMMAND* in setting up callbacks with the EXG graphical user interface library.

QUESTION 1(c)	[3 marks]

- (d) What is the difference between *static* and *dynamic* code analysis? Name one strength and one weakness of each.

QUESTION 1(d)	[3 marks]

- (e) What is a pointer?

QUESTION 1(e)	[1 mark]

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

QUESTION 1(f)	[1 mark]

- (g) What are the essential features of an object-oriented language (that distinguish it from a non-object-oriented language)?

QUESTION 1(g)	[3 marks]

- (h) Write a short Bash script that counts the number of lines of code in a single Eiffel class file. The script shall take the name of the file as its only command-line argument. It shall count all lines except those that only contain white space (blank lines). For the purposes of this question, white space characters are defined to be spaces and tabs. After scanning the named file, the script shall write a single line to the standard output, containing the number of lines of code in the file, a space, and the file's name.

You will probably want to use pipes, and some of the commands `tr`, `grep`, and `wc`. You can write a tab as `\t` and a newline as `\n`.

QUESTION 1(h)	[4 marks]

QUESTION 2 [20 marks]

- (a) This part of the question is about recursive tree data structures for representing algebraic expressions, similar to Expression Version 3 studied in lectures and in the calculator example in Lab 4. Your task is to implement the notion of *variables* in the calculator by defining a new subclass of *EXPRESSION* called *VARIABLE*.

A variable has (a) a name, i.e. a textual representation, which is a single letter, represented in Eiffel code using the type *CHARACTER*; and (b) a value, which is an integer, represented in Eiffel code using the type *INTEGER*. The set of variables used in evaluating an expression is stored in a *DICTIONARY* which maps variable names to variable values.

You may assume that the *CALCULATOR* class has a new attribute:

```
variables: DICTIONARY[INTEGER,CHARACTER]
```

which is initialized thus:

```
create variables.make
from
  i := 'a'
until
  i > 'z'
loop
  variables.put(i.code - ('a').code + 1, i)
  i := i.next
end
```

(So, for instance, `variables.at('c')` will return the value 3.)

The *EXPRESSION* class now looks like this. Notice that the *value* method takes a *DICTIONARY* as a parameter. Notice also the new routine *variables_substituted_with*. Its return value is a new expression with the same structure as *Current*, *except* that each *VARIABLE* node has been replaced with a *CONSTANT* node whose value is the corresponding value of that variable in the *DICTIONARY* passed as the parameter. Each implementation of *variables_substituted_with* must create a new node.

```
deferred class
  EXPRESSION

  -- Arithmetic integer expressions.

feature {ANY} -- Queries

  value(d: DICTIONARY[INTEGER,CHARACTER]): INTEGER is
```

```

        -- The value.
deferred
end

to_string: STRING is
    -- String representation in in-order notation.
deferred
end

variables_substituted_with (d: DICTIONARY[INTEGER,CHARACTER]):
    EXPRESSION is
        -- Return a new expression with all variables replaced with
        -- constants according to the values stored in d.
deferred
end

end -- class EXPRESSION

```

Class *EXPRESSION* will have four effective subclasses, representing constants, variables, sums of two expressions, and products of two expressions. Only one of these, class *CONSTANT*, has been implemented.

```

class
    CONSTANT

    -- Constant expressions

inherit
    EXPRESSION

creation
    make

feature {NONE} -- Creation

    make (v: INTEGER) is
        -- Set 'value' equal to 'v'.
    do
        c := v
    end

    c: INTEGER

feature {ANY} -- Queries

```


(b) Consider binary trees as represented using the *NODE* class as used in lectures:

```
class interface
  NODE[T]
creation
  make
feature
  make
  item: T
  left: NODE[T]
  right: NODE[T]
end of NODE
```

Recall that we also considered this particular version of queues:

```
class interface
  QUEUE[T]
creation
  make
feature
  make
  empty: BOOLEAN
  enqueue(x: T)
  head: T require not empty
  dequeue require not empty
end of QUEUE
```

Now implement a breadth-first search of a binary tree using a queue, that prints out the values stored in the tree in the order they are visited.

In other words, complete the body of this routine:

```
breadth_first_search (p: NODE[T]) is
  local
    n: NODE[T]
    q: QUEUE[NODE[T]]
  do
    -- write the code to go here

  end
```

Hint: you will need the statement `io.put_string(n.item.to_string)`.

QUESTION 3 [20 marks]

In this question you have to do a code review. The code you are to review is an attempted solution to Homework 12. The requirements for the program were:

Write an Eiffel program called 'commonest' that reads text from the standard input and then prints out a table of the 5 most common words in that text, together with their number of occurrences, in descending order of frequency.

For example, suppose that the file `stuff.txt` contains the text:

```
Eiffel! eiffel? eiffel 'EIFFEL' eiFfeL
java Java "JAVA" java
pascal Pascal%$#@!&*PASCAL
C, C; C.
bash-bash
BASIC
COBOL
Fortran66
Fortran77
Fortran90
SQL
```

Then we could have the following interaction with the finished program:

```
comp2100@partch$ commonest < stuff.txt
  5 eiffel
  4 java
  3 c
  3 pascal
  2 bash
```

The program must ignore all punctuation and must reduce all words to lower-case for comparison. A word is defined as an unbroken sequence of *letters* (both upper-case and lower-case) and *digits*. Words are separated by sequences of other stuff (anything else: punctuation, special symbols, white space, ends of lines etc).

Words with the same frequency must appear in alphabetical order. Only the first five words should be printed. The frequency must be printed in a 3-digit field, followed by a single space and then the word.

The attempted solution consists of two classes. The root class is class *COMMONEST*.

```
1 class
2   COMMONEST
3
4 creation
5   make
6
7 feature {ANY}
8
9   separators: STRING is " %T%N!@#$$%^&*()-:;%'"<, > . ? / "
10
11  words: ARRAY [WORD]
12
13  make is
14    -- Read words from input, add to array with
15    -- frequencies, sort, print first five.
16    local
17      input: TEXT_FILE_READ
18      sorter: COLLECTION_SORTER [STRING]
19    do
20      from
21        create input.connect_to (argument (1))
22        create words.make (1, 0)
23        input.read_word_using (separators)
24      until
25        input.end_of_input
26      loop
27        if not input.last_string.is_empty then
28          word := clone (input.last_string)
29          add_word (word)
30          input.read_word_using (separators)
31        end
32      end
33      sorter.sort (words)
34      print_list (5)
35    end
36
37  add (s: STRING) is
38    -- Add an occurrence of 's' to 'words': if it's
39    -- already in there, increment its count, otherwise
40    -- add a new entry.
41    local
42      i: INTEGER
```

```

43     word: WORD
44   do
45     from
46       i := words.lower
47     until
48       i > words.upper or words.item (i).item.same_as (s)
49     loop
50       i := i + 1
51     end
52     if i > words.upper then
53       create word.make (s.to_lower)
54       words.add_last (word)
55     else
56       words.item (i).increment
57     end
58   end
59
60   print_list (n: INTEGER) is
61     -- Print out the first 'n' words with frequencies.
62     local
63       i: INTEGER
64     do
65       from i:= words.lower until i > words.lower + n loop
66         std_output.put_string (words.item (i).to_string + "%N")
67       end
68     end
69
70 end -- class COMMONEST

```

The other class is class *WORD*.

```

71 class
72   WORD
73
74 inherit
75   COMPARABLE
76
77 creation
78   make
79
80 feature {NONE} -- Creation
81
82   make (w: STRING) is

```

```

83     do
84         item := w
85         frequency := 1
86     end
87
88 feature {ANY} -- Modification
89
90     increment is
91         -- Add one to 'frequency'.
92         do
93             frequency := frequency + 1
94         end
95
96 feature {ANY} -- Queries
97
98     word: STRING
99
100    frequency: INTEGER
101
102    to_string: STRING is
103        do
104            Result := frequency + " " + item
105        end
106
107 feature {ANY} -- Comparison
108
109    infix "<" (other: like Current): BOOLEAN is
110        -- Should 'Current' be listed before 'other'?
111        do
112            if frequency = other.frequency then
113                Result := item < other.item
114            else
115                Result := frequency < other.frequency
116            end
117        end
118
119 end -- class WORD

```

There are at least fifteen defects in this program. All are genuine defects in the code, that will either generate a compilation error or cause the program to crash or give wrong results. None are just violations of the coding standard.

Perform a careful review of this code and locate all defects. For each defect, write the number of the line on which it occurs, together with a clear, concise statement of what is wrong there.

If the same defect occurs more than once, only count it as one defect, but indicate all lines that it occurs on.

For your information, here is an excerpt from the interface documentation for the class *INPUT_STREAM* (from which class *TEXT_FILE_READ* inherits).

```
read_word
    -- Read a word using 'is_separator' of class CHARACTER.
    -- Result is available in the 'last_string' common
    -- buffer. Heading separators are automatically
    -- skipped. Trailing separators are not skipped
    -- ('last_character' is left on the first one). If
    -- 'end_of_input' is encountered, Result can be the
    -- empty string.
require
    is_connected;
    not end_of_input
...
read_word_using (separators: STRING)
    -- Same jobs as 'read_word' using 'separators'.
require
    is_connected;
    not end_of_input;
    separators /= Void
```

Here is an excerpt from the interface documentation for class *COMPARABLE*. All classes that inherit from *COMPARABLE* must implement this feature:

```
infix "<" (other: like Current): BOOLEAN
    -- Is 'Current' strictly less than 'other'?
require
    other_exists: other /= Void
ensure
    asymmetric: Result implies not (other < Current)
```

Here is an excerpt from the interface documentation for class *STRING*.

```
to_lower
    -- Convert all characters to lower case.
to_upper
    -- Convert all characters to upper case.
as_lower: like Current
    -- New object with all letters in lower case.
as_upper: like Current
    -- New object with all letters in upper case.
```


QUESTION 4 [20 marks]

- (a) This question is about a class *FIXED_ARRAY_OF_INTEGER*, very similar to the one in Lab 9. This class uses the following C code, from the file *array.c*.

```
#include <stdlib.h>

int *get_new_array(int n){
    return (int *)calloc(n, sizeof(int));
}

int get_array_item(int a[], int i){
    return a[i];
}

void put_array_item(int a[], int i, int x){
    a[i] = x;
}
```

Here is an incomplete implementation of class *FIXED_ARRAY_OF_INTEGER*. Your task is to fill in the missing implementations of some of the features.

You *must* implement this class using external calls to the C code in *array.c*. You may not add any extra attributes to this class.

Hint: Feel free to do these parts in any order. It may be easier to write the wrapper routines first, and then come back and fill in the implementations of *make*, *item* and *put*.

```
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
      good_range: lo <= hi
    do
```

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

QUESTION 4(a)[i]	[3 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 4(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 for *put* here.

QUESTION 4(a)[iii]	[2 marks]

```

        ensure
            -- item (index) = element
        end

```

```

feature {NONE} -- Private attribute

```

```

    storage: POINTER
        -- Location of C array.

```


- (i) Draw a diagram (like the one shown in lectures) showing the *dependencies* between the eight files listed above.

Indicate which files are *source files* by underlining their names.

QUESTION 4(b)[i]	[3 marks]

QUESTION 5 [20 marks]

- (a) Explain why the PSP includes a personal code review *before* the Compile phase, rather than after.

QUESTION 5(a)	[2 marks]

- (b) What is the purpose of calculating the values in the “To Date %” column of the PSP Project Plan Summary?

QUESTION 5(b)	[2 marks]

- (c) What are some of the reasons why software developers do *not* follow a disciplined process like the PSP?

QUESTION 5(c)	[2 marks]

- (d) Explain why there is more to software quality than defect management.

QUESTION 5(d)	[2 marks]

(e) The Process Yield is defined as

$$\text{Yield} = 100 \times \frac{\text{defects removed before Compile}}{\text{defects injected before Compile}}$$

Explain the significance of this number, and why it is defined the way it is. What do high and low values mean? What is a reasonable yield to aim at?

QUESTION 5(e)	[3 marks]

(f) The appraisal to failure ratio is defined as

$$\text{A/FR} = \frac{\text{total time spent looking for defects}}{\text{total time spent fixing defects}}$$

Explain the significance of this number, and why it is defined the way it is. What do high and low values mean? What is a reasonable A/FR to aim at?

QUESTION 5(f)	[3 marks]

