University of Glasgow

Monday, 26th April 2010
9.30 am – 11.00 am
(Duration: 1 hour 30 minutes)

DEGREES OF MSci, MEng, BEng, BSc, MA and MA (Social Sciences)

COMPUTING SCIENCE 3Z: PROGRAMMING LANGUAGES 3

Answer all 4 questions.

This examination paper is worth a total of 80 marks.

You must not leave the examination room within the first half-hour or the last fifteen minutes of the examination.

1. (*Functional programming*)

(a) Explain the principle of *primitive recursion* as applied to:

(i) non-negative integers;

(ii) lists.

[2+2]

(b) In a fictional university, each student receives a *grade* (A, B, C, D, or F) for his/her work in each course. The student's *grade-point average* is the weighted average of the student's grades in all courses (where A counts as 4.0, B as 3.0, C as 2.0, D as 1.0, and F as 0.0).

Write a Haskell function, `gpa gs ws`, that yields the grade-point average of the grades in list `gs`, using the weights in `ws`. You may assume that `ws` has the same length as `gs`, and that the weights in `ws` add up to 1.0 exactly. For example:

```
gpa [‘B’,‘A’,‘D’] [0.25,0.25,0.5]
```

should yield 2.25 (= 3.0×0.25 + 4.0×0.25 + 1.0×0.5).

Your answer must explicitly declare the type of the `gpa` function, and the types of any auxiliary functions.

[7]

(c) Consider a binary search tree (BST) whose nodes contain integers.

(i) Define a Haskell type suitable for such a BST.

(ii) Write a Haskell function, `search i t`, that yields true if and only if the BST `t` contains the integer `i`.

(iii) Write a Haskell function, `insert i t`, that yields the BST obtained by inserting a new node containing the integer `i` into the BST `t`.

Your answer must explicitly declare the type of each function.

[1+4+4]
[*total 20*]

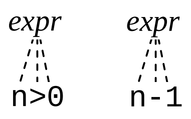
2. (*Syntax*)

Box 1 shows part of the EBNF grammar of the programming language Tiny, which featured in the course. It shows the syntax of commands. It does not show the syntax of expressions (not needed in this question).

(a) Show the syntax tree of the following Tiny command:

```
while n>0: n = n-1;
```

You may assume that `n>0` and `n-1` are expressions. Your syntax tree should show these expressions in outline:



[5]

(b) Suppose that Tiny is to be extended with a loop-command with multiple conditional exits. For example, the following loop-command contains two conditional exits:

```
loop {
     m = m+1;
     exit when m == n;
     f = f*m;
     exit when f > 1000;
}
```

In general, the loop-command may contain zero or more commands and conditional exits, in any order, all enclosed in a pair of curly brackets. Conditional exits are permitted immediately inside a loop-command, but nowhere else.

Modify the grammar to allow for loop-commands.

[5]
[*total 10*]

seq-command	::=	*command*$^+$
command	::=	*ident* = *expr* ;
	|	`read` *ident* ;
	|	`write` *expr* ;
	|	`if` *expr* : *command* (`else` *command*)$^?$
	|	`while` *expr* : *command*
	|	{ *seq-command* }

Box 1 Part of the grammar of Tiny (Question 2).
(Here *expr* is an expression, and *ident* is an identifier.)

3. (*Implementation*)

(a) Explain the difference between a *compiler* and *interpreter*.

[2]

(b) How does an interpreter work? What assumptions must be made about the interpreted language? What differences would you expect to see between an interpreter implemented in C and an interpreter implemented in Haskell?

[8]

(c) Box 2 (next page) shows a description (in Haskell) of a simple stack machine, which is suitable for evaluating simple integer expressions. For example, the following integer expression:

$2\times(-(3+7))-4$

would be evaluated by executing the following list of stack-machine instructions:

```
[PUSH 2, PUSH 3, PUSH 7, ADD, NEG, MUL, PUSH 4, SUB]
```

Implement the function `execInstr`.

[10]

[*total 20*]

```
data Instruction =
              PUSH Int | ADD | SUB | MUL | NEG | HALT
     -- This represents a single instruction. Its effect is:
     --    PUSH i   push i on to the stack
     --    ADD      pop i2; pop i1; push (i1+i2)
     --    SUB      pop i2; pop i1; push (i1-i2)
     --    MUL      pop i2; pop i1; push (i1*i2)
     --    NEG      pop i; push (-i)
     --    HALT     status <- HALTED

     type Code = [Instruction]
     -- This represents code by a list of instructions.

     data Status = RUNNING | FAILED | HALTED

     type Stack = [Int]
     -- This represents a stack by a list of integers.
     -- Elements may be added and removed only at the head
     -- of the list (the stack top).

     type State = (Status, Int, Stack)
     -- This represents the state of the computation. A state
     -- consists of a status, a program counter, and a stack.

     state0 :: State
     -- This is the initial state.
     state0 = (RUNNING, 0, [])

     execCode :: Code -> State -> State
     -- execCode c s yields the state that results from
     -- executing code c in state s.
     …

     execInstr :: Instruction -> State -> State
     -- execInstr i s yields the state that results from
     -- executing instruction i in state s.
     …
```

Box 2 Description of a stack machine, expressed in Haskell (Question 3).

4. (*Concepts*)

(a) What is meant by the *lifetime* of a variable?

What is the lifetime of:

(i) a global variable?

(ii) a local variable?

(iii) a heap variable?

[5]

(b) Consider the Java program outlined in Box 3 (next page). Draw a diagram showing the lifetimes of all global and heap variables created by this program.

[6]

(c) What is meant by the *scope* of a declaration?

Illustrate your answer by stating the scopes of the declarations of the variable `word`, the method `add()`, and the variable `d` in Box 3.

[5]

(d) Briefly explain the general concept of *encapsulation* in programming languages. Why is this an important concept?

[4]

(e) Encapsulation is supported in different ways by particular programming languages.

(i) How is encapsulation supported by Java? Illustrate your answer by referring to the Java code of Box 3.

(ii) How is encapsulation supported by Haskell? Illustrate your answer by outlining Haskell code to implement dictionaries as an abstract data type.

[4+6]

[*total 30*]

```
public class Dict {
     // A Dict object represents a dictionary by a sorted
     //linked list of words.

     private String word;
     private Dict rest;

     public Dict () { word = null; rest = null; }

     public void add (String w) {…}
     // Adds word w to this dictionary.

     public void rem (String x) {…}
     // Removes word x from this dictionary.

     public static void main (String[] args) {
          Dict d = new Dict();
          d.add("rat");
          d.add("cat");
          d.rem("rat");
          d.add("hat");
     }

}
```

Box 3 A Java class (Question 4).