VICTORIA UNIVERSITY OF WELLINGTON

Te Whare Wananga o te Upoko o te Ika a Maui

EXAMINATIONS — 2004

MID-YEAR

COMP 304

Programming Languages

Time Allowed: 3 Hours

Instructions: There are 5 (five) questions, each worth 40 (forty) marks. You should attempt all the questions.

Question 1.

(a) Explain what the following terms mean:

(i). formal parameter;

(ii). actual parameter.

[6 marks]

(b) You are working for a company whose business involves the design of programming languages. One of your colleagues is designing an imperative programming language. State what advice would you give her about providing the following parameter passing mechanisms:

(i). pass-by-value;

(ii). pass-by-name;

(iii). pass-by-reference.

[12 marks]

(c) The same colleague, who has been impressed with your advice on parameter passing mechanisms, comes to you for advice on control structures. State what advice you would give her about which control structures to provide. [12 marks]

(d) The same colleague (who, it would appear, has faked her CV in order to obtain employment) wants to know how to assess a programming language. Explain the criteria that you would use to assess a programming language. [10 marks]

Question 2.

(a) Explain what the following terms mean to a Prolog programmer:

(i). fact;

(ii). rule;

(iii). query.

[4 marks]

(b) Explain what the declarative and procedural semantics of a Prolog program are. [8 marks]

(c) Give an example of two Prolog predicates whose declarative semantics are the same, but whose procedural semantics differ. [4 marks]

(d) Define a predicate `member/2`, such that `member(A, B)` succeeds if the element `A` occurs in the list `B`. [4 marks]

(e) State the results of the following queries:

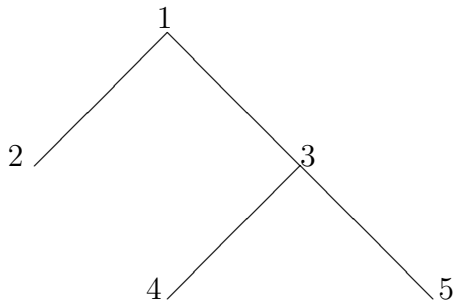
(i). `member(a, B).`

(ii). `member(A, b).`

(iii). `member(A, B).`

[4 marks]

(f) Consider a binary tree like:



In Prolog we can represent it as a term like this:

```
node(1, node(2, leaf, leaf),
        node(3, node(4, leaf, leaf),
                node(5, leaf, leaf)))
```

We can build up a list of the values in a tree. The order of the values in the list will depend on the order we visit the nodes in the tree. A preorder traversal of the tree given above will give the list `[1,2,3,4,5]`. A postorder traversal will give the list `[2,4,5,3,1]`.

Write predicates:

(i). `preorder(Tree, List)`.

(ii). `postorder(Tree, List)`.

which build up the list of nodes using preorder and postorder traversals, respectively. [10 marks]

(g) Explain what some of the deficiencies of Prolog are. [6 marks]

Question 3.

(a) Explain what benefits there are of providing a formal semantics for a programming language. [8 marks]

(b) Explain how we provide an operational semantics for a programming language. [8 marks]

(c) For any imperative programming language with which you are familiar, give a suitable operational semantics for:

(i). the assignment statement; and

(ii). a looping construct.

[12 marks]

(d) Give a suitable semantic algebra and valuation function to give a denotational semantics for dynamic arrays. [12 marks]

Question 4.

(a) Describe a model for imperative programming. [10 marks]

(b) Describe the significant ways an imperative programming language with which you are familiar differs from the model you described above. [10 marks]

(c) Explain why the λ-calculus is of interest to computer scientists. [10 marks]

(d) Explain what facilities Haskell provides to allow the programmer to define new data types. [10 marks]

Question 5.

(a) The type `Snoc a` can be defined in Haskell as follows:

```
infixl 5 :*:

data Snoc a = Empty
            | (Snoc a) :*: a
```

The following are examples of `Snoc`s:

```
Empty :*: 1 :*: 2 :*: 3 :*: 4 :: Snoc Int

Empty :*: 'a' :*: 'b' :*: 'c' :*: 'd' :*: 'e' :: Snoc Char

Empty :*: rem :*: div :*: mod :: Snoc (Int -> Int -> Int)
```

Write the following functions on `Snoc`s. In each case clearly state a suitable type for the function you have defined.

(i) `snocsum`, which adds all the values in a `Snoc`, e.g.

```
snocsum (Empty :*:  1 :*:  2 :*:  3 :*:  4)
```

will evaluate to

```
10
```

[3 marks]

(ii) `snocreverse`, which reverses a `Snoc`, e.g.

```
snocreverse (Empty :*:  1 :*:  2 :*:  3 :*:  4)
```

will evaluate to

```
Empty :*:  4 :*:  3 :*:  2 :*:  1
```

[4 marks]

(iii) `snocmin`, which returns the minimum value in a `Snoc`, e.g.

```
snocmin (Empty :*:  's' :*:  'w' :*:  'o' :*:  'r' :*:  'd')
```

will evaluate to

```
'd'
```

[4 marks]

(iv) `insrt`, which takes a value, `v` and a sorted `Snoc`, `s1`, and returns a sorted `Snoc` containing `v` and the values in `s1`, e.g.

```
insrt 3 (Empty :*:  1 :*:  1 :*:  4)
```

will evaluate to

```
Empty :*:  1 :*:  1 :*:  3 :*:  4
```

[4 marks]

(v) `srt`, which takes a `Snoc` and sorts it, e.g.

```
srt (Empty :*:  4 :*:  1 :*:  3 :*:  2)
```

will evaluate to

```
Empty :*:  1 :*:  2 :*:  3 :*:  4
```

[5 marks]

(b) State what it means to describe a function as a higher-order function. [5 marks]

(c) State what it means to describe a language as using lazy evaluation. [5 marks]

(d) State what advantages, if any, higher-order functions and lazy evaluation provide for the programmer. [10 marks]

****