SCHOOL OF MATHEMATICAL AND COMPUTER SCIENCES

Computer Science

F28PL

Programming Languages MOCK EXAM

Semester 1 2014/15

Sometime before 16 December 2014

Duration: **As long as you like**

ANSWER THREE QUESTIONS
(but there are only two in this mock, for my half of the course)

Answer each question in a separate script book
Do not pick your nose in public
Be nice to waiters
To keep an avocado half fresh in the fridge, leave the stone in it,
and wrap it well in cling film

Q3

(a) Consider these three functions:

```
explode : string -> char list
rev     : 'a list -> 'a list
implode : char list -> string
```

Recall that `o` indicates *composition* (i.e. *chaining* or *pipelining*) of functions.

For each of the following programs state its type, or if it does not have a type, explain why:

(i) `implode o rev o explode` (2)

(ii) `explode o rev o implode` (2)

(iii) `fn f => implode o f o explode` (2)

(iv) `fn f => explode o f o implode` (2)

(v) `fn f => (f explode) rev` (4)

(b) Consider the following function:

```
fun LE [] [] = true
|   LE (hd1::tl1) (hd2::tl2) = hd1=hd2 andalso LE tl1 tl2;
```

(i) State the type of `LE`, and explain what `LE` calculates. (4)

Your explanation should be sufficient to describe to a friend or colleague "What does this program mean?".

(ii) Compiling `LE` will raise a warning. Explain what that warning is, and what it means. (2)

(c) An integer n is a *positive prime* when it is strictly greater than 1 and is divisible only by 1 and by n. (2)

Write a program `isprime : int -> bool` which returns true precisely when input is a positive prime number.

You may assume a function
`divides : int -> int -> bool`
which calculates divisibility (so `divides 2 6` and `divides ~2 6` return `true`, whereas `divides 2 7` returns `false`). You may define any other helper functions you find convenient.

Q4 (a)

(i) Explain in detail what *parametric polymorphism* and *ad hoc polymorphism* are, with at least one example of each. Your examples should make clear how they are similar and how they are different. (6)

(ii) Describe, with explanation, one advantage and one disadvantage of using a typed programming language. (2)

(iii) Discuss to what extent C is typed. (2)

(b) Consider the following Prolog database: (6)

```
mother(X,Y):- parent(X,Y), female(X).
father(X,Y):- parent(X,Y), male(X).
grandmother(X,Y):- mother(X,Z), parent(Z,Y).
parent(diana,william).
parent(william,george).
female(diana).
male(william).
male(george).
```

State and explain the output of the query
`grandmother(X,george)`
You may assume the clauses are numbered 1 to 8.

(c) Suggest clauses to add to the database to calculate `brother(X,Y)` when `X` and `Y` are brothers or half-brothers (share at least one parent). (2)

(d) Socrates said “To be is to do”. (2)
Jean-Paul Sartre said “To do is to be”.

Discuss whether we should formalise this in Prolog as

```
do(X) :- be(X).
be(X) :- do(X).
```

E con le soluzioni …
(And now with the solutions ...)

Q3

(a) Consider these three functions:

```
explode : string -> char list
rev     : 'a list -> 'a list
implode : char list -> string
```

Recall that `o` indicates *composition* (i.e. *chaining* or *pipelining*) of functions.

For each of the following programs state its type, or if it does not have a type, explain why:

(i) `implode o rev o explode` (2)

`string -> string`
explode takes a string to a char list, rev takes that char list to another char list (reversed, in fact), then implode takes the char list to a string.

(ii) `explode o rev o implode` (2)

Type error: the output type string of `implode` cannot match the input type `'a list` of `rev`.

(iii) `fn f => implode o f o explode` (2)

`(char list -> char list) -> string -> string`
We just match the output type of explode to the input type of implode, to deduce the type of f.

(iv) `fn f => explode o f o implode` (2)

`(string -> string) -> char list -> char list`

(v) `fn f => (f explode) rev` (4)

I have no idea why you'd write this. But can you type it?
`((string -> char list) -> ('a list -> 'a list) -> 'b) -> 'b`

(b) Consider the following function:

```
fun LE [] [] = true
|   LE (hd1::tl1) (hd2::tl2) = hd1=hd2 andalso LE tl1 tl2;
```

(i) State the type of `LE`, and explain what `LE` calculates. (4)

Your explanation should be sufficient to describe to a friend or colleague “What does this program mean?”.

`''a list -> ''a list -> bool`
List equality of lists known to be of equal length.

(ii) Compiling `LE` will raise a warning. Explain what that warning is, and what it means. (2)

Nonexhaustive match error. The cases `[] (hd2::tl2)` and `(hd1::tl1) []` are missing.

(c) An integer n is a *positive prime* when it is strictly greater than 1 and is divisible only by 1 and by n. (2)

Write a program `isprime : int -> bool` which returns true precisely when input is a positive prime number.

You may assume a function
```
 divides : int -> int -> bool
```
which calculates divisibility (so `divides 2 6` and `divides ~2 6` return `true`, whereas `divides 2 7` returns `false`). You may define any other helper functions you find convenient.

For completeness I implement `divides` too:

```
fun divides n x = (x mod n = 0);
fun isprimehelper x n =
   n<x  andalso
   ((divides n x) orelse (isprimehelper x (n+1)));
fun isprime x = x=2 orelse (x>2 andalso not (isprimehelper x
2));
```

My questions will tend to include a hard “sting in the tail” at the end.

Q4 (a)

(i) Explain in detail what *parametric polymorphism* and *ad hoc polymorphism* are, with at least one example of each. Your examples should make clear how they are similar and how they are different. (6)

Ad hoc polymorphism is when the same symbol is used for several distinct functions at distinct types.
Example: addition + is defined on integers and floating point numbers, but the implementation is very different.
Parametric polymorphism is when a function is defined parametrically (uniformly) across a family of types.
Example: list reverse is defined on all list types and the implementation of it is uniform regardless of the type.
In the case of addition, even though the implementations are very different, the underlying mathematical function being implemented is the same, justifying the use of the same symbol +. In the case of list reverse, we have the same algorithm operating parametrically over a type variable. This is fairly typical.

(ii) Describe, with explanation, one advantage and one disadvantage of using a typed programming language. (2)

Advantage: types give some guarantees that if a program terminates, it will at least return a value of the correct type. E.g. if a program has type char, we can be sure that if it terminates, then our result will be a char (rather than, say, a long string –- this is how buffer overflow exploits work).
Disadvantage: there are plenty of cool untyped programs out there. With great power comes great responsibility.

(iii) Discuss to what extent C is typed. (2)

C is indeed typed, however, using pointers it is possible to point two variables of different types at the same memory location and thus subvert the type system (e.g. write a floating point number and read its bitwise representation as a string).

(b) Consider the following Prolog database: (6)

```
mother(X,Y):- parent(X,Y), female(X).
father(X,Y):- parent(X,Y), male(X).
grandmother(X,Y):- mother(X,Z), parent(Z,Y).
parent(diana,william).
parent(william,george).
female(diana).
male(william).
male(george).
```

State and explain the output of the query
`grandmother(X,george)`
You may assume the clauses are numbered 1 to 8.

`grandmother(X,george)` unifies with clause 3 generating the binding `Y=george` and two subgoals `mother(X,Z)` and `parent(Z,george)`.
`mother(X,Z)` unifies with clause 1 generating the subgoals `parent(X,Y)` and `female(X)`.
`parent(X,Y)` unifies with clause 4 generating the bindings `X=diana` and `Y=william`. Subgoal solved!
`female(diana)` matches clause 6. Subgoal solved!
`parent(william,george)` matches clause 5. Subgoal solved!

Thus the query succeeds with the binding `X=diana` and `Z=george`.

(c) Suggest clauses to add to the database to calculate `brother(X,Y)` when `X` and `Y` are brothers or half-brothers (share at least one parent). (2)

```
brother(X,Y) :- parent(Z,X), parent(Z,Y), male(X), male(Y).
```

(d) Socrates said "To be is to do". (2)
Jean-Paul Sartre said "To do is to be".

Discuss whether we should formalise this in Prolog as

```
do(X) :- be(X).
be(X) :- do(X).
```

The query `do(X)` will loop indefinitely as "do be do be do be do", as discussed in the song by Frank Sinatra.
(Another possible database would add the clause `scoo(X) :- be(X)`.)