SCHOOL OF MATHEMATICAL AND COMPUTER SCIENCES

Computer Science

F28PL

Programming Languages (mock exam)

Semester 1 201516

Duration: Two Hours

This mock exam is aimed slightly harder than the real exam, because doing real exams is harder in real exam conditions because of the stress.

If you ace this mock then you should pass the exam without difficulty. If you don't understand something in this mock, then make sure you understand it . . . soon!

This mock is available in two versions: one with answers and one without. For best results, try to do the one without answers first (you'll learn more from reading the answers if you've made a serious effort to figure it out yourself).

ANSWER:
If you can read this, then you're looking at the version with the answers. By the way, I wrote a Python script to automatically compile the version without the answers.

1. (a) Clearly write the ML types of the following expressions, or if the expression has no ML type, explain why: (5)

1. `[0,1,2,3]`
   ANSWER:
   `int list`
2. `[0.0,1.0,1]`
   ANSWER:
   Type error (elements should all have same type but `0.0:real` and `1:int`)
3. `0 div 0`
   ANSWER:
   `int` (It has a type: it just raises a runtime error.)
4. `fn f => f f`
   ANSWER:
   Type error. If `f` has type `'a` then since it is applied to itself, it must also have `'a -> 'a`. Thus the type system is being asked to solve `'a='a->'a`, which is impossible (in ML!).
5. `fn (f,g,x) => g(f(x))`
   ANSWER:
   `('a -> 'b)*('b -> 'c)*'a -> 'c`

(b) State the type of the following ML program, and explain what function is calculated by it, making specific reference to the ML execution model (in other words: convince the examiner you understand not only what the program computes, but how):

```
exception Break;
fn f => fn a => fn b =>
   (f(raise Break)
      handle Break => if (f a) then a else raise Break)
         handle Break => if not(f a) then a else b;
```

(3)

ANSWER:
The type is `(bool -> bool) -> bool -> bool -> bool`.

The program inputs `f`, `a`, and `b` and returns `a`. Thus calculates it calculates the same function as

`fn f => fn a => fn b => a.`

This is because when `f(raise Break)` is calculated, flow of control passes to the first handler; then if `(f a)` is true then it returns `a`, otherwise flow of control passes to the second handler which now must return `a`.

(c) Write ML functions of the following types:

1. `('a -> 'b) -> ('a -> 'c) -> 'a -> ('b*'c)` (2)
2. `('a * 'b) -> ('a -> 'c) -> ('b -> 'd) -> 'c*'d` (2)
3. `'a -> 'b list` (2)

ANSWER:

```
fn f => fn g => fn a => (f a,g a);
fn (a,b) => fn f => fn g => (f a,g b);
fn a => [];
```

(d) The **logistic map** is specified by

$$x_0 = 0.5 \quad \text{and} \quad x_{n+1} = r x_n (1 - x_n)$$

where $x_0, x_1, x_2, \ldots$ is a sequence of reals and r is a real number. The logistic map is (part of) the basis of *chaos theory*.

1. Write an ML program

   `logistic : real -> int -> real`

   that if given arguments `r:real` and `n:int` will compute x_n (for the given value of `r`). Answers that do not respect ML's strict type system may lose marks. (3)

   ANSWER:

```
fun logistic r 0 = 0.5
  | logistic r n = let val xnminusone = logistic r (n-1) in
       r * xnminusone * (1.0-xnminusone) end;
```

   Lose one mark for writing `1-xnminusone` instead of `1.0-xnminusone` above.

2. Write an ML program

   `list_f : (int -> 'a) -> (int -> 'a list)`

   that if given `f:int->'a` and `n:int` computes $[f(0), f(1), \ldots, f(n)]$. (2)

   ANSWER:

```
fun listf f 0 = [f 0]
  | listf f n = (listf f (n-1))@[f n];
```

3. Using your answers above, write a program

   `logistic_map : real -> int -> real list`

   that if given `r:real` and `n:int` computes $[x_0, x_1, \ldots, x_n]$. (1)

   ANSWER:

```
fun logistic_map r = listf (logistic r);
(* or *)
fun logistic_map r n = listf (logistic r) n;
(* or *)
val logistic_map = fn r => listf (logistic r);
```

2. (a) State the output of the following programs and explain why, or, if the program terminates with an error state what that error is and why it arises:

1. `"Hello dolly"[::-1]` (2)

   ANSWER:
   `"yllod olleH"` We're stepping back through the string.

2.
```
"".join([x[0] for x in
   "Young Men's Christian Association".split(" ")])
```
(2)

   ANSWER:
   YMCA. What else.

3. `["Hello"].append(["world!"])` (2)

   ANSWER:
   No output. An instance of `["Hello"]` is created, and the result of calling the append method is returned, which is *not* the list itself. Try `type(["Hello"].append(["world!"]))`.

4.
```
x=[]
for i in range(4):
   x=[x]*(len(x)+1)

x[3][2][1][0]
```
(2)

   ANSWER:
   `[]` We create 4 copies of 3 copies of 2 copies of 1 copy of `[]`, then take the first element of the second element of the third element of the fourth element of the result. Back where we started.

(b) Consider the following Python3 code:

```
1 fun mystery d:
2    L = []
3    for k in d:
4       if d[k] not in L:
5          L.append(d[k])
6 return(sorted(L))
```

1. The program is defective and contains four errors. State what they are and how to correct them. (4)

   ANSWER:
   Line 1 `fun` should be `def`.
   Line 1 `d` should be in brackets.
   Line 6 `sorted(L)` should *not* be in brackets.
   Line 6 `return` should be indented by three spaces.
   Final program reads thus:

```
1 def mystery(d):
2    L = []
3    for k in d:
4       if d[k] not in L:
5          L.append(d[k])
6    return sorted(L)
```

2. Describe what function the program calculates. Clearly state any typing assumptions that you make of the input `d`.
   Note that we do not want a blow-by-blow account of execution: we want to know mathematically what it calculates, or to put another way, how you might document the program for a user. (2)

   ANSWER:
   `d` must be a dictionary and the program returns a sorted list of the entries in `d` without repetitions (not the keys). Thus for instance `mystery({ 0:2, 1:2, 2:0, 3:0 })` returns `[0,2]`.

3. The function `mystery` can be expressed in one line of code, using lambda. Propose how. (2)

   ANSWER:

```
1 mystery = lambda d: sorted(list(set(d.values())))
```

(c) Consider the following Python code:

```
x=["Live, Die, "]
x.extend([x])
while True:
    print(x[0],end="")
    x=x[1]
```

Describe the data structure stored in `x` when execution is at line 3. (2)

ANSWER:
`x` satisfies the equation $x = [\text{“Live, Die, ”}, x]$; that is, the first/left-hand element of x is the string “Live, Die, ” and the second/right-hand element of x is x itself. This can be seen as a tree structure that is infinite on its right-hand branch, or as a finite graph structure where the right-hand branch loops back to the root of the graph (diagram omitted in this model answer).

(d) Describe and explain the output of the program. (2)

ANSWER:
Live, Die, . . . repeat. (See the great film with Tom Cruise and Emily Blunt.)

3. (a) Compare and constrast the following terms in detail. Where appropriate illustrate your explanation with concrete code samples, being clear about which language you intend to be writing in. Prove to the examiner that you not only understand these terms, but understand their concrete relevance to specific code of the languages in this course. Note the number of marks for the questions: these give some indication of a minimum of how many individual points you should make in each answer.

1. Functional, logic, and imperative programming. (3)

   ANSWER:

   I will not necessarily write out code samples in these model answers: in exam conditions *you* should, to make sure.

   In functional programming everything is a function. Programs are built up using function application.

   In logic programming everything is a predicate. Programs are built up by accumulating clauses in a database and then querying that database.

   In imperative programming everything is an instruction to an abstract machine. Programs are built up by listing instructions in sequence, and hoping for the best.

2. Global and local state. (2)

   ANSWER:

   Global state is a resource (memory, or a DLL, I/O device, etc) that is accessible from every point in the program. Local state is a resource that is accessible from some strictly-defined part of the program code.

3. Mutable vs. immutable variable. (2)

   ANSWER:

   A mutable variable is one that can be reassigned to a new value, as in `x += 1`. An immuatable variable is assigned when it is created, but cannot them be reassigned, as in `(fn x => x+1)` or `DEF ASCII_a = 97` (*constants* can be viewed as immutable variables!).

4. Mutable vs. immutable type (in Python). Be specific giving at least one example of each. (4)

   ANSWER:

   Python distinguishes between types that carry methods to update values of that type, and types that don't. For instance, *lists* have a `listname.append(x)` method; *tuples* don't. Only immutable types can be keys in dictionary structures, so that we can be sure to find them (i.e. a method can't be called on a key to turn it into another key, losing the data stored at that key). this is why `{ [0]:[0] }` is not Python syntax whereas `{ (0,):[0] }` and `{ 0:[0] }` are.

5. Ad hoc polymorphism, and parametric polymorphism. Be specific and give at least one example of each. (4)

   ANSWER:

   Ad hoc polymorphism is when a single symbol is used to denote different (but probably related) operations on different types. The classic example is +, which denotes an exact addition on integers and an approximate addition on floating point numbers (with very different implementations). See also the equality method == in Python which examines two arguments for whatever the programmer considered to be 'equality'.

   Parametric polymorphism is when a single operation is represented uniformly over a range of types. Examples include `fn x => x : 'a->'a` in ML, or the identity `is` in Python which examines two arguments for literal identity in the abstract machine.

6. Dynamic type error and static type error. (2)

   ANSWER:

   A dynamic type error occurs at runtime. A static type error occurs at compile time. A program that fails due to a dynamic type error has a run-time bug; a 'program' that fails due to a static type error is *not really a program* and does not compile to code.

(b) Imperative programming is doomed: in ten years we'll all be using pure functional programming. Discuss, giving at least two points for and two points against. (4)

ANSWER:
Imperative programming relies on a specific abstract machine. Usually this is a von Neumann machine. Modern computers are no longer von Neumann machines, for (at least) three reasons:

1. CPUs routinely have multiple cores; single digit multiples are common, and double or even triple digit multiples are increasingly so.
2. We are trying to program systems that are often extravagantly distributed and asynchronous, such as satellite networks or distributed architectures.
3. Virtualisation (either of whole machines, or using bytecode) very commonly abstracts away from the hardware.

So imperative programming is a poor paradigm for current hardware, and getting worse. In that sense, imperative programming as it was traditionally understood when C, C++, and Java were designed, is indeed doomed, or at least, is and will increasingly become not the best way to do things.

On the other hand, it is far from clear that functional programming is *the* answer, though it is certainly *an* answer to part of the problem. Pure functional programming is far too restrictive to be a solution to all ills, however, it seems likely that a pure functional programming core will be as taken for granted by my students in the not-too-distant future, as a for-next loop is today.

I have noted this in my teaching: five years ago functional programming was for scientists and highly-paid specialist programmers. Now it's in a scripting language that runs on a computer that costs less than a pair of jeans and is aimed at (amongst other people) small children.

(I know this question has more than 20 points. With so many interesting questions to set, I couldn't decide what to cut.)

4. (a) Explain the differences in Prolog between the *static* and the *dynamic* databases. Your answer should make clear the usage and meaning of the `assert` and `retract` keywords, and should be specific about where and when they can be used. (3)

ANSWER:
The static database is loaded using `[filename]` and is immutable as a query is calculated. The dynamic database is created and destroyed by `assert` and `retract` while a query is calculated. Predicate symbols mentioned in the static database may not be added to the dynamic database, so the dynamic database behaves like a mutable local state.

(b) Convert the following list of English sentences into a Prolog database:

1. If I have chocolate, then I want chocolate.
2. If I want chocolate, then I buy chocolate.
3. If I want chocolate and I have chocolate, then I eat chocolate.

(3)

ANSWER:

```
want(chocolate) :- have(chocolate).
buy(chocolate) :- want(chocolate).
eat(chocolate) :- want(chocolate), have(chocolate).
```

(c) The French word *sera* means 'will be', and the French word *que* means 'whatever'. Thus the French saying *que sera, sera* can be translated into English as *what will be, will be*, and into Prolog as

```
sera(X) :- sera(X).
```

Explain the behaviour of Prolog when asked to predict whether there will be world peace by asking the query `sera(world_peace)` in this database. Your answer should demonstrate specific understanding of the Prolog execution model. (2)

ANSWER:
Infinite loop. Prolog just resolves the head of the clause and generates a subgoal identical to the original goal, and so forth.

(d) Now consider the following database:

```
sera(Y) :- sera(X).
```

Explain the behaviour of Prolog when asked to predict whether there will be world peace by asking the query `sera(world_peace)` in this database. Your answer should demonstrate specific understanding of the Prolog execution model. (3)

ANSWER:
Infinite loop as for the previous question, however this time the subgoal uses a fresh local variable. Eventually we run out of local stack (similar to what Python does after 999 recursive calls) and execution terminates with an overflow error.

(e) Write a Prolog program `sumsq` to calculate the sum of squares of a list of integers (that is, the sum of the list divided by its length). Thus `sumsq([3,4],25)` should return `true` and `sumsq([0,-1,1],X)` should return `X=2`. (4)

ANSWER:

```
sumsq([],0).
sumsq([H|T],X) :- sumsq(T,X1), X is X1+H*H.
```

(f) Consider the following two databases:

- Database 1:

```
food(chicken).
food(fish).
eat(X) :- food(X),!.
```

- Database 2:

```
food(chicken).
food(fish).
eat(X) :- !,food(X).
```

Describe and explain the behaviour of the query `eat(X)` in each database, with specific reference to the Prolog execution model. (2)

ANSWER:
In Database 1 the query returns just one instantiation `X=chicken`. In Database 2 it also returns `X=fish`. In Database 1, the cut `!` prevents the first instantiation found from being undone.

(g) Note that `writef(string)` prints `string` to standard output and then succeeds. With an empty database we type the following at the interactive prompt:

```
assert((eat(chocolate)  :- want(chocolate), have(chocolate),
                           writef("Chocolate face!"),
                           retract(have(chocolate)))).
assert((want(chocolate) :- buy(chocolate))).
assert((buy(chocolate)  :- assert(have(chocolate)))).
eat(chocolate).
```

What will Prolog do, and why? (2)

ANSWER:
It will print `Chocolate face!` and succeed. This is best understood by reading the clauses as imperative programs from left to right. `eat(chocolate)` calls

- `want(chocolate)` which calls `buy(chocolate)` and thus inserts `have(chocolate)` into the database. Then
- `have(chocolate)` succeeds, because we just added this clause to the dynamic database.
- We print `Chocolate face!`.
- `have(chocolate)` is deleted from the dynamic database.
- All subgoals have succeeded, so the query `eat(chocolate)` terminates with success.

(h) Suggest a simple modification to the code above that will cause it to print `Chocolate face!` forever. (1)

ANSWER:
Just add a recursive looping call to `eat(chocolate)` at the end of the subclauses of `eat(chocolate)`.

```
assert((eat(chocolate)  :- want(chocolate), have(chocolate),
                           writef("Chocolate face!"),
                           retract(have(chocolate)),
                           eat(chocolate))).
assert((want(chocolate) :- buy(chocolate))).
assert((buy(chocolate)  :- assert(have(chocolate)))).
eat(chocolate).
```