Programming Languages 3: Questions and Answers: April/May 2012

Duration: 90 minutes.

Rubric: Answer all three questions. Total 60 marks.

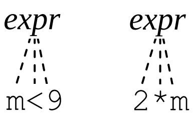
1. (*Syntax*)

Box 1 shows part of the BNF grammar of a fictional programming language called FPL. It shows the syntax of statements and sequential statements. It does not show the syntax of expressions (not needed in this question).

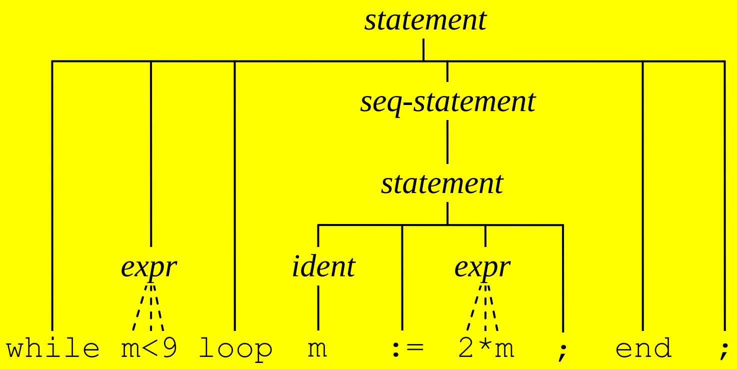
(a) Show the syntax tree of the following FPL statement:

```
while m<9 loop m := 2*m; end;
```

You may assume that `m<9` and `2*m` are expressions. Your syntax tree should show these expressions in outline:



[Unseen problem]



[5]

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

```
loop
    m := m+1;
    exit when m=n;
    f := f*m;
    exit when f>1000;
end;
```

In general, the loop-statement may contain any number of statements and conditional exits, in any order, all enclosed between “`loop`” and “`end`”. Conditional exits are permitted immediately inside a loop-statement, but nowhere else.

Modify the grammar to allow for loop-statements. You may use either BNF or EBNF.

[Unseen problem]

statement = …
| “`loop`” *loop-body* “`end`” “`;`”

cond-exit = “`exit`” “`when`” *expr* “`;`”

loop-body =
| *statement loop-body*
| *cond-exit loop-body*

or (in EBNF):

loop-body = (*statement* | *cond-exit*)*

[5]
[*total 10*]

statement = *ident* “`:=`” *expr* “`;`”
| “`while`” *expr* “`loop`” *seq-statement* “`end`” “`;`”
| …

seq-statement = *statement*
| *seq-statement statement*

Box 1 Part of the grammar of programming language FPL.
(Here *expr* is an expression, and *ident* is an identifier.)

2. (*Implementation*)

(a) Consider the assignment statement:

```
a = b*(c-4*d)
```

where `a`, `b`, `c`, and `d` are all 32-bit integer numbers. Give two assembly-code translations of the statement:

(i) using stack code; and

(ii) using register code.

[Unseen problem]

Assume the following data section for both code examples:

```
vara: dd 0
varb: dd 0
varc: dd 0
vard: dd 0
const1: dd 4
```

(i) Stack code:

```
fild dword [varb]
fild dword [varc]
fild dword [const1]
fild dword[vard]
fmulp  ST(1),ST(0)
fsubp st(1),st(0)
fmulp ST(1),ST(0)
fistp dword [vara]
```

(ii) Register code:

```
mov eax, dword[varc]
mul ebx, [vard],4
sub eax,ebx
mul eax,dword[varb]
movd [vara],eax
```

[1 mark per discrete instruction or concept revealed in the answer]

[8+8]

Now give a quantitative analysis of the relative efficiencies of the two translations, in terms of the number of clock cycles.

The register code involves 4 memory fetches and 5 instructions. Assume that one cycle is taken to execute an instruction and that each variable is in cache and assume 3 cycles to access cache, this gives a total cost of 5+3*4 = 17 cycles.

The stack code requires 5 memory transfers and 8 instructions, so the total cost will be 8+3*5 or 23 cycles

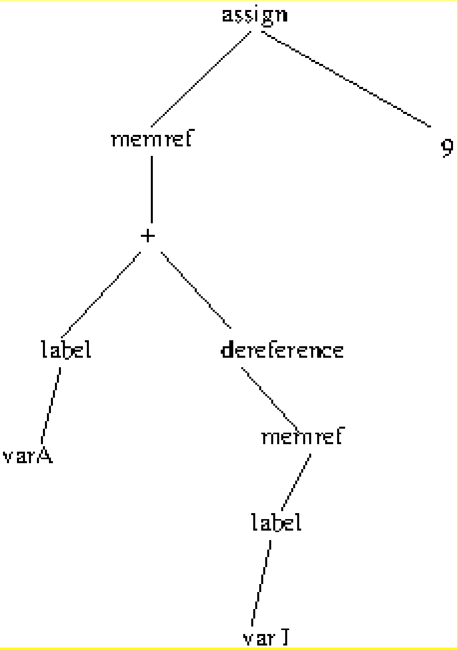
[5]

(b) In a Basic compiler similar to the one you constructed in your course, what would be the intermediate code tree and the assembly code generated for the following statement?

```
LET A(I):= 9
```

[Notes + seen problem]

Intermediate code:



Assembly-code:

```
mov eax,dword[varI]
mov dword[eax*4+varA],9
```

[9]

[*total 30*]

3. (*Concepts*)

(a) Briefly explain how the concepts of *Cartesian products*, *disjoint unions*, and *mappings* are relevant to the understanding of programming languages.

[Notes]

Cartesian products: $S \times T$ is a set of ordered pairs whose components are selected from S and T, respectively. This concept underlies records, structs, and tuples.

Disjoint unions: $S + T$ is a set of tagged values, each selected from either S or T. This concept underlies objects. [The answer could also mention algebraic data types and/or variant records.]

Mappings: $S \rightarrow T$ is the set of all possible mappings from S to T. This concept underlies arrays and functions.

[3]

(b) Using the notation of Cartesian products, disjoint unions, and mappings, write equations defining the set of values of each of the following C types:

```
enum Colour {RED, GREEN, BLUE};
struct CharCount {char c; int i;};
typedef CharCount[] CharProfile;
```

[Unseen problem]

Colour = {0, 1, 2}
CharCount = Character × Integer
CharProfile = Integer → CharCount

[3]

(c) Again using the notation of Cartesian products, disjoint unions, and mappings, write an equation defining the set of objects in a Java program that includes the following classes:

```
abstract class Animal {
     private float weight;
     private boolean can_fly, can_swim;
     …   // methods
}

class Bird extends Animal {
     private int eggs;
     …   // methods
}

class Mammal extends Animal {
     private float gestation;
     …   // methods
}
```

Note that `Animal` is an abstract class.

[Unseen problem]

Object = …
+ *Bird* (Float × Boolean × Boolean × Integer)
+ *Mammal* (Float × Boolean × Boolean × Float)

where *Bird* and *Mammal* are tags.

[3]

(d) Explain the difference between the *copy-in* and *reference* parameter mechanisms.

[Notes]

Copy-in: The formal parameter denotes a local variable of the procedure. The argument value is copied into that local variable on call to the procedure.

Reference: The formal parameter denotes a reference to the argument itself.

[2]

(e) Which of the parameter mechanisms of part (d) are supported by Java, and for which types of parameters?

Illustrate your answer using the following method:

```
static void p (Animal b, float[] fs, float f) { … }
```

What happens to this method's parameters (i) on call and (ii) on return?

[Notes + unseen problem]

In Java the copy-in mechanism is used for parameters of primitive type (such as `f`). The reference mechanism is used (in effect) for parameters of object type (such as `b` and `fs`).

[Equally acceptable answer: The copy-in mechanism is used for all types of parameters. References to objects are themselves values, and are copied into local variables in the same way as primitive values.]

(i) On call, a local variable named `f` is created and initialized with the corresponding argument value. At the same time, `b` and `fs` are made to refer to the corresponding argument objects.

(ii) On return, the local variable `f` is destroyed.

[4]

(f) Suppose that the class `Animal` contains the following abstract method:

```
abstract public void m (float x);
```

and that the classes `Bird` and `Mammal` define different versions of this method.

Consider the method call in the following code:

```
Animal b = …;
b.m(1.5);
```

What determines the target object of the method call? What determines which version of the method is called? How does the called method access the target object?

[Unseen problem]

The target object is the object to which `b` refers; this will be an object of class `Bird` or `Mammal`.

Each object contains a class tag. The class tag of the target object determines which version of the method `m` is called.

A reference to the target object is passed to the method `m` as an additional argument, and **`this`** is bound to that object.

[5]

[*total 20*]