NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
EXAMINATION FOR

SEMESTER 2 2010/2011

CS 4215 - PROGRAMMING LANGUAGE IMPLEMENTATION

April 2011 **Time Allowed: 2 Hours**

INSTRUCTIONS TO CANDIDATES

This question booklet has 18 pages, including this cover page, and contains 9 questions. Answer all questions. This is an open book examination. Any written and printed material may be used during the examination, but no electronic devices are allowed to remain switched on.

Please enter your matriculation number here:

Do not write below this line

Q#	1	2	3	4	5	6	7	8	9	Σ
Max	8	4	12	12	7	12	6	8	6	75
Sc										

Question 1: (8 marks) In the labs of the module, we implemented the language cPL

- by writing in Java a compiler from cPL to cVML, and
- by writing in Java a virtual machine that can execute cVML programs.

Assume that you have an x86 computer, a Java Virtual Machine written in x86 code, and a Java compiler written in x86 code, which produces Java Virtual Machine code. Draw the T-diagrams for all four language processing steps required to execute some cPL program, called `bank-account.cpl`.

Question 2: (4 marks) Recall that the language imPL added assignments to the language simPL (among other constructs).

$$\frac{E}{x \texttt{ := } E}$$

Here, x denotes a variable that needs to be declared elsewhere and E denotes an expression in imPL. Recall that we decided that the result of evaluating such an assignment is the result of evaluating E.
Let us say that we want to design a type system for imPL. Complete the following typing rule that defines the well-typedness of assignments with respect to a given type environment Γ, by adding conditions above the bar.

$$\frac{\Gamma \vdash E : t \qquad \Gamma(x) = t}{\Gamma \vdash x \texttt{ := } E \texttt{ : } t}\textbf{[AssmtT]}$$

Question 3: (12 marks) Let us assume we want to add arrays of integers to the language imPL and its virtual machine iVM. In order to do so, we add the following constructs:

$$\frac{E}{\texttt{newarray}\ E} \qquad \frac{E_1 \quad E_2}{E_1\ \texttt{[}\ E_2\ \texttt{]}} \qquad \frac{E_1 \quad E_2 \quad E_3}{E_1\ \texttt{[}\ E_2\ \texttt{]}\ :=\ E_3}$$

Here, `newarray` is a new keyword in the language. The array indices in an array created by `newarray` E range from 0 to $v-1$, where v is the integer to which E evaluates. Initially, all array values are 0. Here is an example program:

```
let high = 7
in
    let a = newarray (high + 2)
    in
        a[3+2] := 6;
        a[1+5] := 8;
        a[5] + a[2*3] + a[7]
    end
end
```

This example program should evaluate to 14.
In order to implement these new constructs in iVM, we add instructions `NEWARRAY`, `ARRAYACCESS`, and `ARRAYASSIGN` using the following compiler rules:

$$\frac{E \hookrightarrow s}{\texttt{newarray}\ E \hookrightarrow\ s\texttt{.NEWARRAY}} \qquad \frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1\ \texttt{[}\ E_2\ \texttt{]}\ \hookrightarrow\ s_1.s_2\texttt{.ARRAYACCESS}}$$

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2 \qquad E_3 \hookrightarrow s_3}{E_1\ \texttt{[}\ E_2\ \texttt{]}\ \texttt{:=}\ E_3\ \hookrightarrow\ s_1.s_2.s_3\texttt{.ARRAYASSIGN}}$$

Complete the following virtual machine code interpretation for these instructions, using Java. In case you need new classes, please use page 6.
You can make use of the current operand stack using the variable `os`, the current environment using the variable `pc`, the current environment using the variable `e`, and the current runtime stack using the variable `rts`, is the program counter of the virtual machine.
You may assume that there are no type errors; the expressions E_1 above always evaluate to arrays, and the expressions E, E_2, E_3 above always evaluate to integers. If an array access or assignment encounters an index that lies outside the array range, iVM should terminate with an error message.

```
case OPCODES.NEWARRAY:
   {

      IntValue size = (IntValue) os.pop();  // pop size int fr op stack
      ArrayValue a = new Array(size.value); // unbox integer
      os.push(a);                           // push array on op stack
      pc++;                                 // increment program counter
      break;                                // get out of switch stmt




    }
case OPCODES.ARRAYACCESS:
    {

     IntValue index = (IntValue) os.pop(); // pop index fr op stack
     ArrayValue a = (ArrayValue) os.pop(); // pop array fr op stack
     IntValue v = new IntValue(a.get(index.value)); // access array
     os.push(v);                           // push value on op stack
     pc++;                                 // increment program counter
     break;                                // get out of switch stmt



    }
case OPCODES.ARRAYASSIGN:
    {

     IntValue v = (IntValue) os.pop();     // pop new val fr op stack
     IntValue index = (IntValue) os.pop(); // pop index fr op stack
     ArrayValue a = (ArrayValue) os.pop(); // pop array fr op stack
     a.put(index.value,v);                 // array assignment
     os.push(v);                           // push value on op stack
     pc++;                                 // increment program counter
     break;                                // get out of switch stmt


    }
```

(for helper classes)

```
class ArrayValue {
   int[] a;
   ArrayValue(int size) {
      a = new int[size];
   }
   int get(int index) {
      return a[index];
   }
   int put(int index, int val) {
      a[index] = val;
   }
}
```

Question 4: (12 marks) In the following, you are asked to write programs in the virtual machine code SVM. You are free to use any sequence of sVML instructions in any order. To indicate jump addresses, you may label instructions as in the following example.

```
0: LDCI 1
1: LDCI 2
2: GOTO 4
3: PLUS
4: TIMES
```

A stack *underflow* is an attempt to remove or access the top element of an empty stack. In practice, all stacks have finite maximum size, say s. A stack *overflow* is an attempt to push an element on a stack that already holds s elements.

(A) (3 marks) Write an sVML program that will lead to an *operand stack overflow*, regardless of the size of the operand stack.

```
0: LDCI 0
1: GOTO 0
```

(B) (3 marks) Write an sVML program that will lead to an *operand stack underflow*.

```
0: PLUS
```

(C) (3 marks) Write an sVML program that will lead to a *runtime stack overflow*, regardless of the size of the runtime stack.

```
0: LDF 0
1: LDCI 0
2: CALL 1
```

(D) (3 marks) Write an sVML program that will lead to a *runtime stack underflow*.

```
0: RTN
```

Question 5: (7 marks) Let us add to the instruction set sVML an instruction `GOTOD`, whose execution is defined as follows:

$$\frac{s(pc) = \texttt{GOTOD}}{(i.os, pc, e, rs) \rightrightarrows_s (os, pc + i, e, rs)}$$

(A) (2 marks) What will be the result of executing the following SVM program with this new instruction?

```
0: LDCI 5
1: LDCI 1
2: LDCI 3
3: GOTOD
4: TIMES
5: DONE
6: PLUS
7: DONE
8: MINUS
9: DONE
```

Your answer: 6

(B) (5 marks) Translate the following Java program to sVML using `GOTOD`, and without using `JOF`, assuming that the variable `x` is located in the environment at position 1, the function `f` is located in at position 2, the function `g` is located at position 3 and the function `h` is located at position 4. Assume that the function `f` *always* returns 0, 1 or 2.

```
switch (f(x)) {
  case 0 : g(10); return;
  case 1 : h(20); return;
  case 2 : g(30); return;
}
```

```
0: LD 2
1: LD 1
2: CALL
3: GOTOD
4: GOTO 7
5: GOTO 10
6: GOTO 13
7: LD 3
8: LDCI 10
9: CALL
10: LD 4
11: LDCI 20
12: CALL
13: LD 3
14: LDCI 30
15: CALL
```

Question 6: (12 marks) A major disadvantage of memory management by reference counting is that unreachable cyclic data structures are not re-used.

(A) (4 marks) In simPL, cyclic data structures are only created by recursive function definition. Write a simPL expression that will create 1000 nodes on the heap, none of which will ever be re-used during execution of the compiled version of the expression, using reference counting.

```
(recfun f x -> let g = recfun g y -> 0 end in (f x - 1) end end
 1000)
```

(B) (6 marks) Recall that in simPL, cyclic data structures are only created by recursive function definition. Sketch a modified version of reference counting for simPL that uses this fact to achieve re-use of space occupied by recursive function definitions. Use pseudo-code in your solution. Feel free to suggest modifications to the virtual machine and/or compiler to make your memory management work; in that case clearly describe the suggested modifications.

(idea) The closure nodes created by `LDRF` should be marked as "recursive" using a tag that is different than the tags for regular function nodes. Then execution of LDRF simply subtracts 1 from the reference count of the closure and its environment.

(C) (2 marks) In the virtual machine implementation of imPL, which we called `imPLvm`, it is possible to construct cyclic data structures even without recursive function definitions. Write an imPL program without recursive function definitions that creates a cyclic data structure in `imPLvm`.

```
let x = [A:[]]
in x.A := x
end
```

Question 7: (6 marks) Consider the virtual machine for imPL presented in the lectures. The `CALL` instruction is the most complex instruction, since it allocates three new nodes in the heap; a stack frame, an environment and an operand stack.
You have seen that imPL function calls can sometimes be compiled to an optimized form, where the allocation of some of these nodes can be avoided. For example, the optimized instruction `TAILCALL` avoids the creation of any of these nodes.
Consider the call of `g` in the following imPL program.

```
( recfun g x y ->
    if x > 0
    then x := x - 1; y := y + 1; (g x y) + 2
    else 0
    end
  4 5 )
```

(A) (2 marks)

1. Why can the call of `g` not be implemented using `TAILCALL`? (explain in at most three sentences)

   The recursive call for `g` is not that last operation in the body of `g`. The call needs to return to the body of `g` in order to load the integer 2 and perform the addition.

(B) (4 marks) Describe an optimization of `CALL` for the call of `g` that avoids the creation of at least one of the mentioned three nodes.

In this particular case, we can reuse the environment; the callee is going to need `x` and `y`, just like the caller. Here is the idea:

```
case OPCODES.OPTCALL:  {
                         int n = i.NUMBEROFARGUMENTS;
                         Closure closure
                             = os.pop(); // function value
                         rs.push(new StackFrame(pc+1,e,os));
                         pc = closure.ADDRESS;
                         os = new Stack();
                         break;
                       }
```

assuming that the recursive call of `g` is compiled into `OPTCALL`, avoiding compiling of the arguments `x` and `y` into `LD` instructions. Students who do not do that would need to pop two argument values from the operand stack before popping the closure.

Question 8: (8 marks) Object-oriented languages often have a keyword `super` that allow the programmer to call methods of inherited classes. In this question, you are asked to design `super` for the language oPL. Example:

```
let c1 =
    class
        method F(x) -> x + 1 end
    end
in
let c2 =
    class extends C1
        method F(x) -> super.F(x) * 2 end
    end
in
    (new c2).F(3)
end
```

A super call, such as `super.F(x)` in this example, looks for a method with the given property, here `F`, in the class *that surrounding class extends*. It then applies the corresponding function to `this` and other arguments of the super call. Thus, the program above will result in the integer 8.

(turn the page)

(A) (3 marks) The `super` class may be different from the parent class of `this`. Complete the following program by inserting one method in one of the provided spaces such that the program evaluates to `false`.

```
let c1 =
    class
        method Val() -> 1 end




    end
in
let c2 =
    class extends c1
        method Test() ->
           (this.Class.Parent.Val this) = super.Val()
        end

        method Val() -> 2 end


    end
in
let c3 =
    class extends c2




    end
in
    (new c3) . Test()
end
end
end
```

(B) (5 marks) Sketch how such a `super` construct could be implemented, based on the oPL interpreter covered in the module. If your implementation requires support from the parser, then clearly describe this support by saying what data structures it should generate for the new language construct.

(sketch) The compiler needs to get a hold of the class from which the surrounding class inherits. This can be done by compiling a `class extends E ... end` into `let parent = E in [Parent:parent,...] end` where ... stands for the methods of the class, as usual.
Then a call `super.F(E1...En)` can be compiled to `((lookup parent F) this E1 ... En)`.

Question 9: (6 marks) Consider the language cPL presented in the lectures. Assume that there is a built-in function `print` that displays the value of its parameter. Consider the following coPL program:

```
let x = 3
in
    thread x := x + x end ;
    thread x := x * x end ;
    thread print x end
end
```

Assume that the assignment `x := x + x` is compiled to

```
LDCI 1
LD 1
LD 1
PLUS
ASSIGN
```

and the assignment `x := x * x` is compiled to

```
LDCI 1
LD 1
LD 1
TIMES
ASSIGN
```

List all possible values that can be displayed by this program, using interleaving semantics at the level of virtual machine instructions.

```
3
6
9
18
36
12
(18 again: 3 * 6)
```

END OF QUESTIONS