Degree Examinations 2012-13

DEPARTMENT OF COMPUTER SCIENCE

Principles of Programming Languages

Time allowed: **Two hours**

Candidates should answer question 1 in Section A, one question from Section B and one question from Section C.

Calculators may be used in this examination.

Do not use red ink.

Section A: You must answer this question

1 (20 marks)

(i) [2 marks] What is the difference between an *assignment* and a *binding*?

(ii) [2 marks] What form of recursive definition is equivalent to iteration, and how do recursive and iterative processes differ?

(iii) [2 marks] How do *expressible values* and *denotable values* differ?

(iv) [2 marks] Distinguish between the terms *static scoping* and *dynamic scoping*.

(v) [2 marks] Briefly explain what is meant by *type inference*, indicating why it is a useful facility.

(vi) [2 marks] Describe *two* ways in which scheme and Haskell differ *in principle*.

(vii) [2 marks] What is the difference between parallel and concurrent execution?

(viii) [2 marks] What is the difference between a nested and a flat process structure?

(ix) [2 marks] Define the Liveness Property, as it applies to concurrent tasks.

(x) [2 marks] What happens to a task that calls the Wait method on a semaphore that has the value 2?

Section B: Answer one question from this section

2 (40 marks)

(i) [10 marks]

(a) [2 marks] Briefly distinguish between the terms *call-by-need* and *call-by value*.

(b) [4 marks] The scheme *special form* `(if predicate expr1 expr2)` has an evaluation rule which differs from that of normal 'compound' procedures such as those defined using lambda expressions. What is this different rule, and why is it required?

(c) [4 marks] In a language with a *lazy evaluation* computational model, there is no need for an 'if' expression to have a special evaluation rule. Why not?

(ii) [10 marks] Given this Haskell definition:

```
f a b = (a+b) : f b (a+b)
```

carefully explain what is being calculated by the Haskell expression:

```
f 0 1
```

(iii) [20 marks] A programmer wrote a version of the above Haskell program in scheme as follows:

```
(define (f a b)
 (cons (+ a b) (f b (+ a b)))
)
```

To test this, the programmer wrote a procedure `(nth n lst)` which gives the n^{th} element of a non-empty list, counting from 0:

```
(define (nth n lst)
  (if (= n 0) (car lst)
              (nth (- n 1) (cdr lst)))
)
```

Unfortunately, when trying to evaluate the expression: `(nth 4 (f 0 1))`, no output was produced, and the scheme interpreter eventually crashed.

(a) [3 marks] Briefly explain why the scheme version failed to work.

(b) [3 marks] The programmer remembered that there is a standard way of delaying the evaluation of an arbitrary expression. What is that method?

(c) [14 marks] Rewrite the two procedures `(f a b)` and `(nth n lst)`, using this delayed evaluation technique or otherwise, so that an evaluation of `(nth 5 (f 0 1))` does not crash (giving the correct result of 13). The two revised procedures must have the same control-space complexity as the originals.

3 (40 marks)

(i) [10 marks]

(a) [5 marks] Why is it correct to say that scheme is a *functional programming language*, whereas Java is not?

(b) [5 marks] In what sense is scheme not a *pure* functional programming language?

(ii) [10 marks] The following is a recursive scheme procedure. The call `(times x y 0)` computes $x \times y$ when `x` and `y` are two positive integer values.

```
(define (times a b r)
  (if (= b 0)
      r
      (times a (- b 1) (+ r a))))
```

Write an equivalent *iterative* (non-recursive) Java method which does not use the `*` operator.

(iii) [20 marks] The following shows a Java method.

```
int f(int x) {
  int result=0, temp=1;
  for(int i=0; i<x; i++) {
    int r = result;
    result = temp;
    temp = r + temp;
  }
  return result;
}
```

(a) [5 marks] What function does `f(n)` compute?

(b) [15 marks] Translate the Java method into a scheme procedure which has the same function, and which exhibits *iterative* control behaviour.

Section C: Answer one question from this section

4 (40 marks)

(i) [6 marks] What is the difference between synchronous and asynchronous message passing; and what is the difference between a rendezvous and an extended rendezvous? What do the languages Ada and Erlang support?

(ii) [18 marks] The following Ada code defines a Server task that manages a single resource by accepting calls to Use and Replace the resource in a strict sequence.

If the requirements for this task are changed so that there are now 10 resources managed by the task, how could the code be changed to accommodate more concurrent calls to Use and Replace? Your code (in Ada) should allow calls to Use to be accepted if there is a resource available, and calls to Replace should always be accepted.

You should explain the behaviour of any language features you use.

```
task Server is
  entry Use(...);
  entry Replace(...);
end Server;

task body Server is
  Number_Of_Resources : constant := 1;
  ...
begin
  loop
    accept Use(...) do
        -- code for this service
    end Use;
    accept Replace(...) do
        -- code for this service
    end Replace;
  end loop;
end Server;
```

(iii) [4 marks] How would you change your code so that the Server task terminates if all client tasks have terminated?

(iv) [12 marks] Outline, using code fragments, how the Server task given in part(ii) of this question could be coded in Erlang. You need not consider how the Erlang processes are created (spawned), rather concentrate on the communication between the processes.

5 (40 marks)

(i) [5 marks] Describe the Readers and Writers Problem. Why can a standard monitor not solve this problem?

(ii) [15 marks] The following code implements a protocol to solve the Readers and Writers Problem in Java. Give a full explanation of how this code provides the necessary communication. Are Writers or Readers given preference?

```
startRead(); // code for Reader
  // call object to read data structure
stopRead();

startWrite(); // code for Writer
  // call object to write data structure
stopWrite();

public class ReadersWriters {

  private int readers = 0;
  private int waitingWriters = 0;
  private boolean writing = false;

  public synchronized void StartWrite()
        throws InterruptedException {
    while(readers > 0 || writing) {
      waitingWriters++;
      wait();
      waitingWriters--;
    }
    writing = true;
  }

  public synchronized void StopWrite() {
    writing = false;
    notifyAll();
  }

  public synchronized void StartRead()
        throws InterruptedException {
    while(writing || waitingWriters > 0) wait();
    readers++;
  }

  public synchronized void StopRead() {
    readers--;
    if(readers == 0) notifyAll();
  }

}
```

(iii) [5 marks] What criticisms, due to the facilities of Java, can be made of this protocol?

(iv) [5 marks] What facilities in Pascal-FC (for the support of standard Hoare monitors) can be used to improve the protocol?

(v) [10 marks] Sketch the Pascal-FC code that will implement the improved protocol.

Department of Computer Science

Degree Examinations 2012-13

Principles of Programming Languages

Marking Notes

Marking Notes

Answers for Section A: You must answer this question

Question 1 (20 marks)

Part (i) [2 marks]

A *binding* associates a name with a value. An *assignment* changes the contents of the variable (which may, or may not, be bound to a name).

Part (ii) [2 marks]

Tail-recursion. Iterative => O(1) control space complexity, recursive => worse. control space complexity

Part (iii) [2 marks]

Expressible values are those that can be the results of (compound) expressions. Denotable are those that can be defined, for instance using primitive syntactical means, but cannot be the results of expressions.

Part (iv) [2 marks]

Static scope is where the appropriate binding is determined *lexically*, and thus can be determined at compile-time . . . it is the 'closest' binding textually.

Dynamic scope is where the binding is determined by the call-sequence of program blocks . . . it is the 'latest' binding made a run-time.

Part (v) [2 marks]

Types of primitives are known, so the type of an expression can be inferred from the types of its components. This means that, in such a system, it is not necessary for a programmer to explicitly state the types of every name.

Part (vi) [2 marks]

- Haskell uses lazy evaluation, scheme doesn't.
- Haskell is statically-typed, scheme is (at most) dynamically typed.
- Haskell uses call-by-need, scheme call-by value.
- Haskell has no 'assignment' (mutation), scheme does (`(set! ...)`)
- . . . or any other valid differences I've not mentioned.

Part (vii) [2 marks]

Concurrent is potential parallel.

Part (viii) [2 marks]

Nested allows tasks to be defined within tasks; flat implies tasks only allowed at outermost program level.

Part (ix) [2 marks]

Something good will eventually happen. Eventually all desirable future states will be reached.

Part (x) [2 marks]

Task continues executing, semaphore goes to 1.

Answers for Section B: Answer one question from this section

Question 2 (40 marks)

Part (i) [10 marks]

Sub-part (a) [2 marks]

Both are forms of *lazy evaluation* . . . 'need' implies sharing, whereas 'name' does not.

Sub-part (b) [4 marks]

Normal rule: (applicative order): evaluate all the arguments then apply the procedure.

Special rule: evaluate predicate, then evaluate one or other of expr1 / expr2.

Need the special rule to a) avoid unwanted computation, b) to 'emulate' the effects of normal-order evaluation.

Sub-part (c) [4 marks]

Since the arguments are only evaluated when 'proven' to be needed — in this case when the predicate determines which expression is required — then the applicative-order problems don't occur.

Part (ii) [10 marks]

This evaluates to the Fibonacci series. Important points:

3 marks represented as an *unbounded* ('infinite') list,

3 marks lazy evaluation, so nothing is actually computed until demanded.

General description [4 marks].

Part (iii) [20 marks]

Sub-part (a) [3 marks]

Scheme is not lazy, therefore the recursive definition of `f`, having no base case, runs until out of memory.

Sub-part (b) [3 marks]

Thunking . . . using a nullary procedure to delay the evaluation of the body expression.

Sub-part (c) [14 marks]

The two procedure re-definitions are:

```
(define (f a b)
  (cons (+ a b) (lambda () (f b (+ a b))))  ; NB thunk
)
```

```
(define (nth n lst)
  (if (= n 0)
      (car lst)
      (nth (- n 1) ( (cdr lst) ) )    ; NB invocation of
                                      ; thunked tail:
                                      ;     ( (cdr lst) )
  )
)
```

It is possible to answer this using the `force` and `delay` special forms (not taught in the module), and this would be acceptable e.g.:

```
(define (f a b)
  (cons (+ a b) (delay (f b (+ a b))))
)

(define (nth n lst)
  (if (= n 0) (car (force lst))
              (nth (- n 1) (force (cdr lst))))
)
```

Marks:

- Correct use of cons in `f` [4 marks]
- Correct use of thunk in `f`[6 marks]
- Correct invocation of thunk in `nth`[4 marks]

Marking Notes

Question 3 (40 marks)

Part (i) [10 marks]

Sub-part (a) [5 marks]

In java, functions/procedures are not *first-class* values (functions are not *expressible* values). but in scheme they are.

Sub-part (b) [5 marks]

A 'pure' functional language has no *side-effecting* operations such as assignment. The are various forms of 'mutation' (assignment) operations in scheme.

Part (ii) [10 marks]

The scheme code is tail-recursive, consequently the equivalent Java can be simply written using a `while` (or `for`) loop:

```
int times( int a, int b, int r){
      while( b != 0 ) {
            r = r + a;
            b = b - 1;
      }
      return r;
}
```

Other versions are acceptable.

Marks:

5 marks Identifying tail-recursion with a `while` loop

5 marks Identifying assignments with the updates of arguments.

Part (iii) [20 marks]

Sub-part (a) [5 marks]

The function returns the n^{th} Fibonacci number.

Sub-part (b) [15 marks]

The following is *one* way of defining the procedure. However, any equivalent will be acceptable.

```
(define (f n)
  (define (fi a b count)  ;; local definition
    (if (= count 0)
      b
```

```
            (fi (+ a b) a (- count 1)))
        )
        (fi 1 0 n)
      )
```

Marks:

5 marks Correct use of local definition (can use a `(let ...)` expression etc.

5 marks Correctly tail-recursive.

5 marks Correct initial arguments.

Marking Notes

Answers for Section C: Answer one question from this section

Question 4 (40 marks)

Part (i) [6 marks]

With synchronous, the caller waits until the receiver has the message, this is also called a rendezvous. An asynchronous send does not wait. With an extended rendezvous the caller also waits until the received has constructed a reply.

Ada supports the extended rendezvous, Erlang asynchronous message passing.

Part (ii) [18 marks]

Code must be changed to include a select statement and a guard. [10 marks for code]

```
task Server is
  entry Use(...);
  entry Replace(...);
end Server;

task body Server is
  Number_Of_Resources : constant := 10;
  In_Use : integer := 0;
  ...
begin
  loop
    select
      when In_Use < Number_Of_Resources =>
      accept Use(...) do
        -- code for this service
      end Use;
      In_Use := In_Use + 1;
    or
      accept Replace(...) do
        -- code for this service
      end Replace;
      In_Use := In_Use - 1;
    end select;
  end loop;
end Server;
```

A select statement allows a task to choose, non-deterministically, between different accept statements. The task blocks if there are no calls to either Use or Replace; accepts one, if there is one, and accepts either one if there are call on both. [4 marks]

However branches of a select statement can be guarded; so for that execution of the select a false guard implies that the associated accept statement will not be chosen. A guard is used to prevent a call to Use being accepted if there are no resources available.

Part (iii) [4 marks]

Add a 'or terminate' alternative to the select statement.

```
    select
      when In_Use < Number_Of_Resources =>
    accept Use(...) do
      -- code for this service
    end Use;
    In_Use := In_Use + 1;
  or
    accept Replace(...) do
      -- code for this service
    end Replace;
    In_Use := In_Use - 1;
  or
    terminate;
  end select;
```

Part (iv) [12 marks]

As communication is asynchronous the Use call must be implemented as a double call - one to request, the other to receive notification that the resource is assigned. A call to replace can be asynchronous with no notification needed. So a call of Use would be:

```
Server ! Use, self(),
receive
  confirm
end
```

A call to Replace is simply

```
Server ! Replace
```

The server itself would repeat

```
receive
  Use, Pid -> Pid ! confirm,
  receive
    Replace
  end
end
```

or

```
receive
  Use, Pid -> Pid ! confirm
end,
receive
  Replace
end
```

Question 5 (40 marks)

Part (i) [5 marks]

Many reader threads and many writer threads are attempting to access an object encapsulating, say, a large data structure. Readers can read concurrently, as they do not alter the data. Writers require mutual exclusion over the data both from other writers and from readers.

A standard monitor will provide the mutual exclusion, but will not allow Readers to read concurrently.

Part (ii) [15 marks]

Requires the students to read and understand code in Java. The following points should be addressed.

1. Reader and Writers need a pre and post protocol, rather than just mutual exclusion. (2 marks)
2. A class is defined to control calls to the four operations. (2 marks)
3. Use of synchronous methods to ensure mutual exclusion. (2 marks)
4. Use of 'wait' to stop thread making progress. So reader stopped if a write in progress, and write stopped if any other thread active (indicated by values of the two local variable). (3 marks)
5. use of notifyAll to awake all waiting threads (one at a time). they need to retest the condition to make progress. (3 marks)
6. Writers are given preference via the waitingWriters variable. (3 marks)

Part (iii) [5 marks]

On awaking after the wait request, the thread must re-evaluate the conditions under which it can proceed. Although this approach will allow multiple readers or a single writer, arguably it is inefficient, as all threads are woken up every time the data becomes available. Many of these threads, when they finally gain access to the monitor, will find that they still cannot continue and, therefore, will have to wait again.

It should also be noted that this solution is not tolerant to the InterruptedException being thrown. But this is not required in the answer.

Part (iv) [5 marks]

Pascal-FC's monitors have condition variables (as did Hoare's original monitors). So two

Marking Notes

CV can be used: OKTOREAD and OKTOWRITE.

Part (v) [10 marks]

The following is quite complete; not all details are needed for high mark. The key aspect is that no process is woken up unless it is ready to proceed.

```
monitor ReaderWriters is
export
  StartRead, StopRead, StartWrite, StopWrite;
var
  OKTOREAD, OKTOWRITE : condition;
  writers : boolean;
  WaitingWriters, readers : integer;

  procedure StartRead;
  begin
    if writers then
      delay(OKTOREAD)
  end;

  procedure StopRead;
  begin
    if WaitingWriters and readers = 0 then
      resume(OKTOWRITE)
  end;

  procedure StartWrite;
  begin
    if readers > 0 or writers then
    begin
      WaitingWriters := WaitingWriters + 1;
      delay(OKTOWRITE)
      WaitingWriters := WaitingWriters - 1;
    end;
    writers := true
  end;

  procedure StopWrite;
  begin
    writers := false;
    if WaitingWriters then
      resume(OKTOWRITE)
    else
      resume(OKTOREAD)
  end;

begin
  WaitingWriters := 0;
  readers := 0;
  writers := false;
end;
```