CHALMERS TEKNISKA HÖGSKOLA
Dept. of Computer Science and Engineering
John Hughes

Tuesday 16th December, 2008.
Programming Paradigms
DAT120(CTH) / DIT330(GU)

Exam in Programming Paradigms

Tuesday 16th December, 2007, EM.
Lecturer: John Hughes, tel 070 756 3760.

Permitted aids:
English-Swedish or English-other language dictionary.

There are five questions, one on each paradigm, worth 12 points each for a total of 60 points. 24 points is required to pass (grade 3), 36 points is required for grade 4, and 48 points is required for grade 5.

1 Imperative Programming [12 points]

1. (Parameter passing.) **[4 points]**

   (a) Given the following program:

```
int i;
int a[2];
void p(int x, int y) {
    x++;
    i++;
    y++;
}
int main () {
    a[0] := 1;
    a[1] := 1;
    i := 0;
    p(a[i],a[i]);
    print ("a[0] = ", a[0]);
    print ("a[1] = ", a[1]);
    return 0;
}
```

   For each of the following methods of parameter passing, draw the activation record of the subroutine `p` at the time when the parameters are passed. Also give the output of above program in each of the cases

   i. Call-by-reference
   ii. Call-by-value
   iii. Call-by-result
   iv. Call-by-value-result; if the return address is computed...
      A. early (beginning of the call)
      B. for x and y separately (before and after i++)
      C. late (when the call returns)

   (**2p.**)

   (b) Consider the following definition of the subroutine `sum`:

```
int sum (by-name int a, by-name int index, int size)
{
    int tmp = 0;
    for (index = 0; index < size; index++)
    {
        tmp +=a;
    }
    return tmp;
}
int x[10];
...
int result =  sum(?,?,10);              //  (*)
```

Assume the integer array `x[10]`, of size 10, is appropriately initialized. Demonstrate the use of *call-by-name* in the invocation of `sum` (see the line marked (*)) by filling in the proper actual arguments so that the call to `sum` computes the sum of all elements from `x[0]` to `x[9]`. Introduce auxiliary variables where necessary. (**2p.**)

2. (Immutability.) **[4 points]**
The `String` concatenation operator `+` in the Java language is specified as follows:

> (JLS, 15.18.1) String Concatenation Operator +
> "If only one operand expression is of type String, then string conversion is performed on the other operand to produce a string at run time. The result is a reference to a String object (newly created) that is the concatenation of the two operand strings. The characters of the left-hand operand precede the characters of the right-hand operand in the newly created string. If an operand of type String is null, then the string "null" is used instead of that operand."

Given the following class in the Java language:

```
class GoodMorning
{
   public static void appendMorning (String t)
   {
       t = t + "Morning";
   }
   public static void main(String [] args)
   {
       String s = "Good ";
       appendMorning(s);
       System.out.println(s);
   }
}
```

(a) The specification of the string concatenation operator allows the Java String class to be *immutable*. Read the specification above carefully and determine *where* in the specification immutability comes into play. Give the phrase(s) literally, as used in the wording of the specification. (**2p.**)

(b) What does the program print? Draw a snapshot of the stack of activation records ("run-time stack") when the string concatenation operator `+` is invoked and the parameter is passed; draw another snapshot directly after `+` has returned. Use the snapshots to explain. (**2p.**)

3. (Scoping.) **[4 points]**
Assume a for-statement in a Java-like language, that is, of the form

for (ForInit; Expression; ForUpdate) Statement.

Further assume the scope of a local variable declared in the *ForInit* part of a for-statement includes all of the following:

- Its own initializer.
- Any further declarators to the right in the *ForInit* part of the for-statement.
- The *Expression* and *ForUpdate* parts of the for-statement.
- The contained *Statement*.

Your task:

(a) There are other scopes besides for-loops. Give the names of two additional examples of scopes. (**1p.**)

(b) Consider the following three snippets of for-loops. For each snippet, determine whether it compiles or fails with a scoping-related error (make no additional assumptions about the variable `i`). In case of an error, mark its position in the snippet and give a useful error message that a compiler could provide. (**3p.**)

```
//  Snippet 1
for (int i = 0; i < 10; i++) {
      ...
}
i = 5;

//  Snippet 2
int i = 5;
for (int i = 0; i < 10; i++) {
    ...
}

// Snippet 3
for (int i = 0; i < 10; i++) {
    ...
}

for (int i = 0; i < 10; i++) {
    ...
}
```

2 Object-Oriented Programming [12 points]

1. (Smalltalk.) **[4 points]**
Consider the message `lineCount` in the Smalltalk class `String`.

```
lineCount
"Answer the number of lines represented by the receiver, where
every cr adds one line."
| cr count |                                                    "(1)"
cr := Character cr.                                          "(2)"
count := 1 min: self size.                                   "(3)"
self do: [:c | c == cr ifTrue: [count := count + 1]].        "(4)"
^ count                                                      "(5)"
```

   (a) What do the bars in line (1) and the period (in lines 2-4) mean? Give the technical term for each. (**1p**.)

   (b) In lines (3) and (4): determine all messages, and for each message determine the receiver object and all argument object(s). For readability, provide your answers in tabular form

Message	Receiver	Argument(s)
...	...	...

   (**3p**.)

2. (Algebraic specification.) **[4 points]**
Assume an abstract data type Natural with 4 operations: zero, succ, add, and is-zero. Consider zero and succ as constructors.

   (a) Provide an appropriate signature. (**2p**.)

   (b) Provide the appropriate axioms that capture the intuitive understanding of zero, succ, addition, and is-zero for natural numbers. Your axioms should allow one to show the identity

$$\text{add}(\text{succ}(\text{succ}(\text{zero})), \text{succ}(\text{zero})) = \text{succ}(\text{succ}(\text{succ}(\text{zero}))).$$

   (**2p.**)

3. (Polymorphic methods, static and dynamic binding.) **[4 points]**
Many object-oriented languages provide keywords with which users can control whether a method gets bound dynamically or statically; in the pseudo language used below, `virtual` indicates dynamic binding.

```
class A {
     public non-virtual void f() { print( "A.f "); }
     public virtual void g()     { print( "A.g "); }
     public non-virtual void h() { f(); g(); }

}
class B inherits A {
     public non-virtual void f() { print( "B.f "); }
     public virtual void g()     { print( "B.g "); }
}
int main() {
      A a;
      B b;
      // initialization of a,b
      a.h();
      b.h();
      a = b;
      a.h();
}
```

(a) What gets printed if a,b have value types? (**2p**.)

(b) What gets printed if a,b have reference types? (**2p**.)

3 Functional Programming [12 points]

1. Given

$$\begin{aligned} inc &= \lambda x.\ x + 1 \\ twice &= \lambda f.\ \lambda x.\ f\ (f\ x) \end{aligned}$$

use β-conversion to simplify the following λ-expression as far as possible:

$$twice\ twice\ inc\ 0$$

Show each step in your answer. 1 point

2. *Church numerals* are a representation of natural numbers as λ-expressions, where the λ-expression $\lambda f.\ \lambda x.\ f^n(x)$ represents the natural number n. For brevity, we shall write this λ-expression as $\hat{n}$. Define λ-expressions *suc* and *add* that implement the successor and addition functions, such that

$$\begin{aligned} suc\ \hat{n} &= \widehat{n+1} \\ add\ \hat{m}\ \hat{n} &= \widehat{m+n} \end{aligned}$$

(Note that defining *suc* as $\lambda\hat{n}.\ \widehat{n+1}$ would make no sense—$\hat{n}$ stands for a λ-expression, not a variable name, and indeed, no "hats" (ˆ) should appear in your answer). 2 points

3. Suppose the Haskell type `Set a` represents a *set* of values of type `a`.

   (a) Suggest suitable types for the functions `insert` and `delete`, where `insert a s` inserts an element `a` into the set `s`, and `delete a s` removes an element `a` from a set `s`. 1 point

   (b) What will the value of the following expression be, given that `s` is the set $\{1, 2\}$? You may write set values informally using the usual mathematical notation $\{x_1, x_2, x_3 \ldots\}$.

   ```
   (delete 2 s, insert 3 s)
   ```

   1 point

4. Sets can be implemented by *ordered binary trees*, in which each node is either a *leaf*, or a *branch* containing a left sub-tree, an element, and a right sub-tree, with the invariant that all the elements in the left sub-tree are less than the element in the branch node, and all the elements in the right sub-tree are greater than it. Ordered binary trees can be represented by the following Haskell type:

   ```
   data Set a = Leaf | Branch (Set a) a (Set a)
   ```

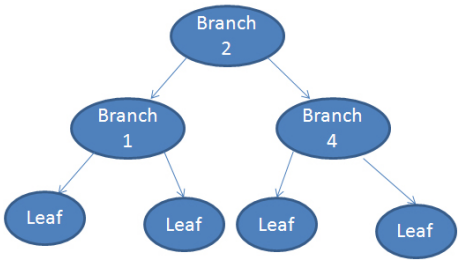
   The following function inserts a new element into such a tree, in the correct position.

```
insert a Leaf = Branch Leaf a Leaf
insert a (Branch l b r) =
  if a==b then Branch l b r
  else if a < b then Branch (insert a l) b r
  else Branch l b (insert a r)
```

(a) Given the definition

```
t = insert 1 (insert 4 (insert 2 Leaf))
```

then the following diagram illustrates the value of `t`.



Copy this diagram, and *in the same diagram* draw the data-structure returned by `insert 3 t`. Make sure you accurately represent any sharing between old and new trees. 1 point

(b) In your diagram, mark the nodes that will be reclaimed by the garbage collector assuming that there are no references to `t`, but there is a reference to the result of `insert 3 t`. 1 point

(c) Define a function `delete`, which deletes an element from a tree. 3 points

Hint: you will find you need a function to join together the left and right subtrees of a branch-node into a single tree. You can use the following function to do this:

```
join Leaf t = t
join (Branch l b r) t = Branch l b (join r t)
```

There is no need to repeat this definition in your answer.

(d) Suppose the function

```
setElements :: Ord a => Set a -> [a]
```

returns a sorted list of the elements of a set. Write a *QuickCheck property* that uses an ordered list of elements as a *model* to test the `insert` function. You may assume that QuickCheck can already generate random `Set` values (so you do not need to define a generator for `Set`s), and you may use the function `List.insert x xs` that inserts an element `x` into an ordered list `xs` in the correct position. 2 points

4 Concurrency Oriented Programming [12 points]

Study the following Erlang code, which implements a simple *generic server*.

```
start_server(M) ->
  register(M,spawn(fun() -> server(M,M:init()) end)).

server(M,S) ->
  receive
    {Pid,Msg} ->
      {Reply,NewS} = M:handle(S,Msg),
      reply(M,Pid,Reply),
      server(M,NewS)
  end.

reply(M,Pid,Msg) ->
  Pid ! {M,Msg}.

rpc(M,Msg) ->
  M ! {self(),Msg},
  receive {M,Reply} -> Reply end.
```

1. What do the following notations mean?

   (a) `Pid ! Msg`

   (b) `receive Msg -> ... end`

   2 points

2. Erlang does not provide locks to protect shared data from simultaneous modification by two or more concurrent processes. What prevents Erlang processes from corrupting shared data? 1 point

3. Suppose we want to use the generic server above to create a server that implements a *variable*, with initial value zero, handling requests `read` to read the variable's value, and `{write,X}` to set the variable's value to `X`. For example, we might increment the value held in the variable using the following code in a client:

   ```
   N = rpc(variable,read),
   rpc(variable,{write,N+1})
   ```

   Write definitions of the callback functions `init` and `handle` to implement this behaviour. 2 points

4. When a client tries to increment the variable's value using the code above, there is a risk that a different client might change the variable's value between the read and the write. We might therefore wish to add two new requests to the server's repertoire: `lock` and `unlock`, so that the code above can be written as

```
rpc(variable,lock),
N = rpc(variable,read),
rpc(variable,{write,N+1}),
rpc(variable,unlock)
```

with the effect that the server only accepts requests from this client between the lock and the unlock. Show how to *modify the generic server* so that *all* servers implemented using it support the `lock` and `unlock` requests. 2 points

Hint: you can write code such as

```
receive
  {Pid,{write,X}} when X>0 ->
    ...
end
```

to accept a message only when a certain condition holds.

5. What happens to requests from *other* clients, which are sent while the lock is held? 1 point

6. What is the effect of *linking* two Erlang processes? 1 point

7. Show how to modify your generic-server-with-locking, so that if a client crashes while holding the lock, then

   - the lock is released, so that other clients can use the server,
   - the *state* of the server is restored to the value it had when the lock was last claimed—so that a client which crashes while holding the lock has no visible effect, as far as other clients are concerned.

   3 points

5 Logic Programming [12 points]

1. What is the result of unification of the following pairs of terms? In each case, state whether or not unification succeeds, and if it succeeds, give the values bound to the variables. 2 points

   (a) `[X|Xs]` and `[1,2,3]`

   (b) `[X,2]` and `[1,Y]`

   (c) `[A,A]` and `[1,B]`

   (d) `[A,B]` and `[1,2,3]`

   (a) `true` and `X=1` and `Xs=[2,3]`

   (b) `true` and `X=1` and `Y=2`

   (c) `true` and `A=B=1`

   (d) `false`

2. Given the clauses

```
mem(X,[X|Xs]).
mem(X,[Y|Xs]) :- mem(X,Xs).
```

   what will Prolog display in response to these queries? Make sure to include all the solutions Prolog will find. 3 points

   (a) `mem(X,[]).`

   (b) `mem(X,[1,2,3]).`

   (c) `mem(1,X).`

   (a) `false`

   (b) `true` and (`X=1` or `X=2` or `X=3`)

   (c) `true` and `X=[...,1,...]`

3. Using no more than two clauses, define a predicate `del(X,Xs,Ys)` which holds when `X` is an element of the list `Xs`, and removing one `X` from `Xs` leaves the list `Ys`. For example, given the query `del(2,[1,2,3],Ys)`, Prolog should reply with `Ys = [1,3]`. 2 points

```
del(X,[X|Xs],Xs).
del(X,[Y|Xs],[Y|Ys]) :- del(X,Xs,Ys).
```

4. What solutions will Prolog find for the queries 2 points

   (a) `del(1,[1,2,1],Ys).` (a) `Ys=[2,1]` or `Ys=[1,2]`

   (b) `del(3,Xs,[1,2]).` (b) `Xs=[3,1,2]` or `Xs=[1,3,2]` or `Xs=[1,2,3]`

5. Study the following clauses, which define the predicate `reverse(Xs,Ys)` that holds if the lists `Xs` and `Ys` are each other's reversal.

```
reverse([],[]).
reverse([X|Xs],Ys) :- reverse(Xs,Zs), append(Zs,[X],Ys).
```

   This uses the `append` predicate from the lectures:

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

(a) Both of the queries `reverse([1,2,3],Xs)` and `reverse(Xs,[1,2,3])` find the solution `Xs = [3,2,1]`, but one of them falls into an infinite loop if we ask for more solutions. Which query can fall into an infinite loop? 1 point

The second (`reverse(Xs,[1,2,3])`) will loop endlessly, because the code only specifies that `Xs1` should be unified with `[X|Xs2]`, and that `Ys` should be unified with `[1,2,3]`. The recursive use of `reverse` then performs a similar unification, without making progress.

(b) Using no more than three clauses, give a definition of the `reverse` predicate which does not fall into a loop in either of these cases. 2 points

```
reverse([],[]).
reverse(Xs,[Y|Ys]) :- nonvar(Ys), !, reverse(Ys,Zs), append(Zs,[Y],Xs).
reverse([X|Xs],Ys) :- reverse(Xs,Zs), append(Zs,[X],Ys).
```