CHALMERS TEKNISKA HÖGSKOLA
Dept. of Computer Science and Engineering
Jean-Philippe Bernardy

14:00, March 14th, 2013.
Programming Paradigms
DAT121 / DIT331(GU)

Exam in Programming Paradigms

14:00, March 14th, 2013.
Examiner: Jean-Philippe Bernardy

Permitted aids: Pen and paper.

There are 6 questions: each worth 10 points. The total sum is 60 points.

Some questions come with *remarks*: you must take those into account. Some questions come with *hints*: you may ignore those.

You will be asked to write programs in various paradigms. Choose the language appropriately in each case, and indicate which you choose at the beginning of your answer.

Paradigm	Acceptable language
Imperative	C or (an OO language where you refrain to use Objects)
Object oriented	C++ or Java
Functional	Haskell, ML
Concurrent	Erlang, Concurrent-Haskell
Logic	Prolog, Curry

You may also use pseudo-code resembling an actual language in the relevant list. In that case, make sure your code can only be interpreted in the way you intent (the responsibility lies with you). In particular, in the case of functional/logic languages, omitting parentheses is NOT acceptable: `a b c` is not acceptable pseudo-code for `a (b c)`.

Chalmers: 24 points is required to pass (grade 3), 36 points is required for grade 4, and 48 points is required for grade 5.

GU: 24 points is required to pass (grade G) and 42 points is required for grade VG.

1 Objects as records

Consider the following class hierarchy written in a Java-like language:

```
class A {
  float x;
  float y;
};

class B {
  int n;
  void f(A p) {
    <BODY 1>
  };

};

class C extends B {
  int m;
  // the following method overrides f from class B
  void f(A p) {
    <BODY 2>
  }
};
```

Translate the above classes to records with explicit method pointers.

Demonstrate your solution by drawing a diagram of a memory area corresponding to an object of type...

- `A`. **1 points**
- `B`. **3 points**
- `C`. **6 points**

Remarks: make sure all the fields are initialized to some value. The values can be arbitrary but must be consistent with the idea of the translation to records with explicit method pointers. If your solution contains pointers, you must show what they point to.

Answer:

```
struct A {
  float x = 0
  float y = 0
}
struct B {
```

```
  int n = 0
  void* f(B* this, A* p) = <BODY1>;
}
struct C {
  int n = 0
  void* f(C* this, A* p) = <BODY2>;
  int m;
}
```

2 Calls and recursion

Consider the following function `hanoi`, which solves the Hanoi tower problem.

```
void hanoi(int n,int s,int i,int d) {
  if (n>0) {
    hanoi(n-1,s,d,i);
    move(s,d);
    hanoi(n-1,i,s,d);
  }
}
```

Solve the following two exercises independently.

1. Replace the last recursive call by a while loop (tail-call elimination). Remark: do not use a stack. **4 points**

   Answer:

   ```
   void move_many(int n, int source, int intermediate, int target) {
     while (n > 0) {
       move_many(n-1,source,target,intermediate);
       move_disk(source,target);
       n = n-1;
       swap(source,intermediate);
     }
   }
   ```

2. Assume a stack; with the usual push, pop and top functions. Transform the original `hanoi` function by making the calls explicit. Remarks: use gotos, and stack to transform *each* recursive call. Transform the original `hanoi` function, not the one obtained in the previous exercise. **6 points**

   Answer:

   ```
   void move_many(int n, int s, int i, int d) {
     push(n,s,i,d,0);
   call:
     if (stk->n!=0) {
       push(stk->n-1,stk->s,stk->i,stk->d,1);
       goto call;
   loc1:
       move_disk(stk->s,stk->d);
       push(stk->n-1,stk->i,stk->s,stk->d,2);;
       goto call;
   loc2:
   ```

```
  }
  caller = stk->caller;
  pop();
  if (caller == 1) goto loc1;
  if (caller == 2) goto loc2;
}
```

3 Pattern matching Higher-Order Abstractions

Assume the following Haskell data type:

```
data List = Nil | Cons Int List
```

and consider the following code:

```
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)

append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)

reverse Nil = Nil
reverse (Cons x xs) = append (reverse xs) (Cons x Nil)
```

Express each of the functions above (`map`, `append`, `reverse`) in terms of the higher-order function `fold` provided below. Remarks: you may *not* use recursion directly in your solution. The type of the functions should remain the same. **3,3,4 points**

```
fold k f Nil = k
fold k f (Cons x xs) = f x (fold k f xs)
```

Answer:

```
map f = fold Nil (\x -> Cons (f x))
append xs ys = fold ys Cons xs
reverse = fold Nil (\x xs -> append xs (Cons x Nil))
```

4 Closures

Consider the following partial implementation of the "Observer" pattern in a Java-like language:

```
interface Observer {
  bool notify(Event e);
}

class MyPrintAction implements Observer {
   String text;

   bool notify(Event e) {
     print("Got the event:" + e);
     print(text);
     return true;
   }
}

class Button {
  List<Observer> os;

  void onPress (Observer l) {
    os.add(l);
  }
}
```

And the example user code:

```
button.onPress(new MyPrintAction("Button pressed."));
```

Assuming that the interface `Observer` represents a closure, how could this pattern be implemented naturally in a language with native support for higher-order functions?
Demonstrate your answer by translating the pattern to Haskell as follows:

1. Give the Haskell type corresponding to the Observer interface. **3 points**
2. Give the type of the translated onPress function. (A Haskell type) **2 points**
3. Translate the MyPrintAction class. **2 points**
4. Translate the example call to onPress. **3 points**

Hints: You may assume a Haskell function `print` which prints objects of any type. You may (if you prefer) answer questions 3. and 4. in a single piece of code.

Answer:

```
type Observer = Event -> IO Bool

onPress :: Button -> Observer -> IO ()

myPrintAction text e = do
  print ("Got the event" ++ show e)
  print text

example = onPress button (myPrintAction "button pressed");
```

5 Variable-managing process

Assume a language without pointers nor variables, but support for concurrency. In particular, assume primitives for creating processes and channels, and primitives for reading and writing to to channels. For example C++-style:

```
Chan<A> newChan();
A readChan(Chan<A> c);
void writeChan(Chan<A> c,A x);
void forkProcess(*void());
```

or Haskell-style:

```
newChan :: IO (Chan a)
readChan :: Chan a -> IO a
writeChan :: Chan a -> a -> IO ()
forkProcess :: IO () -> IO ()
```

Your task is to simulate references to mutable variables using the above primitives. This can be done be using a process that manages the variable state.

1. Define the representation for a variable of type "reference to Integer". Name this type `Reference`. **3 points**
2. Write the code for the process that manages the variable state. **4 points**
3. Write the code for primitives to create, read and write references. Their type should be: **3 points**

   ```
   Reference newRef();
   int readRef(Reference);
   void writeRef(Reference, int);
   ```

   or Haskell-style:

   ```
   newRef :: IO Reference
   readRef :: Reference -> IO Int
   writeRef :: Reference -> Int -> IO ()
   ```

Hint: The channels can transmit any type of information, including references to channels.

Remark: You can use either a functional or imperative language to write your answer, however you may not use any global variable nor primitive reference types in it.

Answer:

```haskell
data Command a = Get (Chan a) | Set a
type Variable a = Chan (Command a)

handler :: Variable a -> a -> IO ()
handler v a = do
  command <- readChan v
  case command of
    Set a' -> handler v a'
    Get c -> do
      writeChan c a
      handler v a

newVariable :: a -> IO (Variable a)
newVariable a = do
  c <- newChan
  forkIO (handler c a)
  return c

get :: Variable a -> IO a
get v = do
  c <- newChan
  writeChan v (Get c)
  readChan c

set :: Variable a -> a -> IO ()
set v a = do
  writeChan v (Set a)
```

6 Relations

Consider the following Haskell code:

```
data List = Nil | Cons Int List
data Tree = Tip | Bin Tree Int Tree

append :: List -> List -> List
append Nil xs = xs
append (Cons x xs) ys = Cons x (append xs ys)

flatten :: Tree -> List
flatten Tip = Nil
flatten (Bin a x b) = append (flatten a) (Cons x (flatten b))
```

Translate the above functions to relations `flatten'` and `append'` such that:

```
append' x y z       is equivalent to      append x y == z
flatten' x y        is equivalent to      flatten x == y
```

1. Write the type of the relations `flatten'` and `append'`. **4 points**
2. Write their code. **6 points**

Remark: you cannot use any other helper function in your answer, only constructors and relations. The types should be written in Curry or Haskell-like syntax.
You may write the code in Curry syntax or Prolog syntax.

Answer:

```
append :: List -> List -> List -> Success
append []     ys zs = ys =:= zs
append (x:xs) ys zs = append xs ys zs' &
                      zs =:= x:zs'
   where zs' free

flatten :: Tree -> List -> Success
flatten Tip Nil = success
flatten (Bin a x b) zs = flatten a xs & flatten b ys & append xs (Cons x ys) zs
   where xs, ys free
```