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

14:00, August 30th, 2013.
Programming Paradigms
DAT121 / DIT331(GU)

Exam in Programming Paradigms

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. **4 points**
Complete your solution by drawing a diagram of a memory area corresponding to an object of type each type `A`,`B`,`C`.

- `A`. **1 points**
- `B`. **2 points**
- `C`. **3 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 function

```
procedure fib(n : Int)
  if n == 0
    return 0
  else if n == 1
    return 1
  else
    tmp1 := fib (n-1)
    tmp2 := fib (n-2)
    return tmp1 + tmp2
```

Remove recursion, using an explicit stack. **10 points**

Remarks

- You should assume a "global" stack accessed via `push`, `pop` and `top` primitives.
- You must implement *the same algorithm*. Do not change the algorithm in the process of doing the translation. Do not apply tail-call optimisation.

Hints: you should first define the type of values that you can push on the stack, and you can assume that labels can be used as values.

Answer:

```
Stack frame: (n:Int,tmp:Int,caller:Pointer)

fib:
  if top.n == 0
     result = 0;
     goto top.caller
  else if top.n == 1
     result = 1
     goto top.caller
  else
     push (n-1,X,ret1)
     goto fib
ret1:
     pop;
     top.tmp = result;
     push (n-2,X,ret2);
     goto fib;
ret2:
```

```
pop;
result := result + top.tmp;
goto top.caller;
```

3 Pattern matching and Higher-Order Abstractions

Assume the following Haskell data type:

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

and consider the following code:

```
foobar :: List -> Int
foobar []     = 0
foobar (x:xs)
  | x > 3     = (7*x + 2) + foobar xs
  | otherwise = foobar xs

sum [] = 0
sum (x:xs) = x + sum xs

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

filter f Nil = Nil
filter f (Cons x xs) = if f x then Cons x (filter f xs) else filter f xs

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

Express the function `foobar` in terms of `sum`, `map` and `filter`. **5 points**
Express the function `foobar` in terms of `fold` ONLY. **5 points**
Remark: In particular you may NOT use recursion (directly) in your definitions.

Answer:

```
foobar = sum . map (\x -> 7*x+2) . filter (>3)
foobar = fold 0 f where
   f x acc = if x > 3 then 7*x + 2 + acc else acc
```

4 Closures

You will demonstrate how to transform higher order functions to first order ones, by using explicit closures. That is, use a Haskell *data structure* to represent λ-expressions.
Consider the function filter:

```
filter f Nil = Nil
filter f (Cons x xs) = if f x then Cons x (filter f xs) else filter f xs
```

- Translate the code of `filter`. **2 points**
- Give the type of the closure application function used in the translation. **2 points**

Consider the following use of `filter`:

```
above x xs = filter (greaterThan x) xs
greaterThan x y = x < y
```

- Give the representation `greaterThan` in the translation. **2 points**
- Translate the function `above` to use explicit closures. **2 points**
- Give the code of the closure application function. **2 points**

Answer:

1.

```
filter f Nil = Nil
filter f (Cons x xs) = if f `ap` x then Cons x (filter f xs) else filter f xs
```

2. `ap :: Fun -> Int -> Bool`

3.

```
data Fun = GreaterThan Int
```

4. `above x y = filter (GreaterThan x) xs`

5. `ap (GreaterThan x) y = x < y`

5 Continuations

Consider the following function, which solves the Hanoi tower problem:

```
hanoi 0 s i d = return ()
hanoi n s i d = do
    hanoi (n-1) s d i
    move s d
    hanoi (n-1) i s d
```

Transform `hanoi` to use explicit continuations. **10 points**

Remarks:

- In the translation, you will use a different version of `move`, which takes an explicit continuation as argument.
- The translation of `hanoi` should also take an explicit continuation as argument.
- Do not change the algorithm. Do not "optimize" it. The call structure should remain the same.

Answer:

```
hanoi 0 s i d k = k ()
hanoi n s i d k =
    hanoi (n-1) s d i (
    move s d          (
    hanoi (n-1) i s d
    k))
```

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' `and`
                      zs =:= x:zs'
   where zs' free

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