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

8:30, March 15th, 2014 (Location:Maskin).
Programming Paradigms
DAT121 / DIT331(GU)

Exam in Programming Paradigms

8:30, March 15th, 2014 (Location:Maskin).
Examiner: Jean-Philippe Bernardy

Permitted aids: Pen and blank paper.

There are 5 questions, each worth 12 points. The total sum is 60 points.

Some questions come with *remarks*: you must take those into account. Some questions come with *hints*: even though they are meant to help you, 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 necessary parentheses is NOT acceptable: `a b c` means `(a b) c` and 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 Explicit gotos

Consider the following optimised code for array copy:

```
short *to, *from;
int count;
...
{
   /* pre: count > 0 */
   int n = (count + 7) / 8;
   switch(count % 8){
     case 0: do{     *to++ = *from++;
     case 7:         *to++ = *from++;
     case 6:         *to++ = *from++;
     case 5:         *to++ = *from++;
     case 4:         *to++ = *from++;
     case 3:         *to++ = *from++;
     case 2:         *to++ = *from++;
     case 1:         *to++ = *from++;
              } while (--n > 0);
   }
}
```

Translate the above snippet to equivalent code that does not use `switch` nor `while`: use explicit gotos instead. Write your answer in a C-like language.

Remark: you must retain the optimisation. Do not test the loop condition after each copy of a byte.

Note: You can get partial credit for removing either the switch or the while.

2 Objects as records

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

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

class B {
  int n;
  // this method is overridable (the default in Java)
  // in C++ it would be marked 'virtual'
  void f(A p) {
    n = n+n;
  };

};

class C extends B {
  int m;
  // the following method overrides f from class B
  void f(A p) {
    n = n+n;
    m = m+n;
  }
};
```

Translate the above classes to records with explicit method pointers. Remark: remember to write the code which initialises the explicit method pointers.

3 Pattern matching and Higher-Order Abstractions

Consider the following code:

```
foobar :: [Int] -> Int
foobar []     = 0
foobar (x:xs)
  | x > 20     = (5*x-20) + foobar xs
  | otherwise = foobar xs

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

map f [] = []
map f (x:xs) = (f x) : (map f xs)

filter f [] = []
filter f (x:xs) = if f x then x:(filter f xs) else filter f xs

fold k f [] = k
fold k f (x:xs) = f x (fold k f xs)
```

- Express the function `foobar` in terms of `sum`, `map` and `filter`. **6 points**
- Express the function `foobar` in terms of `fold` ONLY. **6 points**

Remark: In particular you may NOT use recursion (directly) in your definitions.

4 Continuations and Closures

Replace lambda abstractions by explicit closures in the following code.

```
f # x = f x

data Tree a = Leaf | Bin (Tree a) a (Tree a)

traverseC :: Tree Int -> (Int -> Int) -> Int
traverseC Leaf k = k 0
traverseC (Bin l x r) k =
  traverseC l # \tl ->
  traverseC r # \tr ->
  k (tl + x + tr)

traverse t = traverseC t (\x -> x)
```

5 Unification

The following implementation of the unification algorithm contains two mistakes. Hint: they are within the code of 'unify'.

- In which lines(s) of the unify function do the mistake occur? **4 points**
- Explain each of the mistakes. **4 points**
- Write a correct version of the algorithm. You need only write the lines that need to be changed. Be explicit about where the correct code should go or what it should replace. **4 points**

```
both f (x,y) = (f x, f y)
data Term = Con String [Term] -- the terms are the arguments to the constant
          | Var Int -- metavariable
  deriving Eq
(1) unify :: [(Term,Term)] -> Substitution -> Maybe Substitution
(2) unify [] s = Just s -- Base case
(3) unify ((t,t'):ts) s | t == t' = unify ts s
(4) unify ((Con f as,Con g bs):ts) s
(5)   | length as == length bs = unify (zip as bs ++ ts) s
(6)   | otherwise = Nothing -- Clash
(7) unify ((Var x,t):ts) s = unify (map (both (applySubst (x ==> t))) ts) (s +> (x,t))
(8) unify ((t, Var x):ts) s = unify ((Var x, t):ts) s -- Re-orient
```

For completeness here is the code for management of substitutions.

```
type Substitution = Map Int Term

-- | Identity (nothing is substituted)
idSubst = M.empty

-- | Add an "assignment" to a substitution
(+>) :: Substitution -> (Int, Term) -> Substitution
s +> (x,t) = M.insert x t (M.map (applySubst (x ==> t)) s)

-- | Single substitution
(==>) :: Int -> Term -> Substitution
x ==> t = M.singleton x t

-- | Apply a substitution to a term
applySubst :: Substitution -> Term -> Term
applySubst f (Var i) = case M.lookup i f of
                         Nothing -> Var i
                         Just t -> t
applySubst f (Con c xs) = Con c (map (applySubst f) xs)
```