CHALMERS TEKNISKA HÖGSKOLA	14:00, March 9th, 2012.
Dept. of Computer Science and Engineering	Programming Paradigms
Jean-Philippe Bernardy	DAT121 / DIT331(GU)

Exam in Programming Paradigms

14:00, March 9th, 2012.
Examiner: John Hughes
Lecturer: 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 in the following. 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 ressembling 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 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 from records

Consider the following C++ code, which is an encoding of a class hierarchy.

```
struct C {
   float x;
   float y;
};

struct B {
   void *n(B*, C*);
   int f;
   void *m(B*, B*);
   int g;
};

struct A {
   void *m(A*, B*);
   int f;
};

void B_n(B* p, C* q) {
  q->x += p->f;
  q->y += p->g;
}

void B_m (B* p, B* q) {
  q->f += p->f;
  q->g += p->g;
}

void A_m(A* p, B* q) {
  q->f += p->f;
}
```

Quick refresher:

- If `A` is a type, then `A*` is the type of pointers to `A`.
- `void *f(A,B,...)` stands for a pointer to a function `f` taking arguments `A,B,...`
- `p->f` is used to access a field `f` in a structure pointed by `p`.

Write the original class hierarchy in a C++-like or Java-like language.

3 Higher-order abstractions

Consider the following Haskell code:

```
concat [] = []
concat (x:xs) = x ++ concat xs

data Tree = B Tree Tree | Leaf Int

reverse (Leaf x) = (Leaf (0-x))
reverse (B x y) = B (reverse y) (reverse x)
```

Express the same functions in terms of the higher-order functions provided below. Remark: you may *not* use recursion directly in your solution.

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

foldT k f (Leaf x) = k x
foldT k f (Bin x y) = f (foldT k f x) (foldT k f y)
```

4 Closures

Consider the Haskell program:

```
f m grades bonuses = zipWith (\x y -> x + y) bonuses (filter (\ x -> x < m) grades)

zipWith f [] [] = []
zipWith f (x:xs) (y:xs) = f x y : zipWith f xs ys

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

Translate the code to use explicit closures.

Remarks:

- All higher-order functions must be removed.
- You cannnot *specialize* any higher-order function (eg. don't make a special version of `zipWith` which does only addition). Use closures.

5 Laziness

In Haskell you can write a matrix comprehension like so:

```
matrix = array bounds [((x,y),valueAtIndex x y) | (x,y) <- range bounds]
  where bounds = ((low_x,low_y),(high_x,high_y))
```

You can then access a cell x, y in the array using `matrix!(x,y)`.

For example, in this matrix of 100 elements the value at index (x, y) contains $x \times y$: `example!(x,y) == x * y`.

```
example = array bounds [((x,y),x * y) | (x,y) <- range bounds]
  where bounds = ((1,1),(10,10))
```

Consider the following algorithm to compute the Levenshtein distance between two strings:

```
int LevenshteinDistance(char s[1..m], char t[1..n])
{
  // for all i and j, d[i,j] will hold the Levenshtein distance between
  // the first i characters of s and the first j characters of t;
  // note that d has (m+1)x(n+1) values
  declare int d[0..m, 0..n]

  for i from 0 to m
    d[i, 0] := i // the distance of any first string to an empty second string
  for j from 0 to n
    d[0, j] := j // the distance of any second string to an empty first string

  for j from 1 to n {
    for i from 1 to m {
      if s[i] = t[j] then
        d[i, j] := d[i-1, j-1]       // no operation required
      else
        d[i, j] := minimum
                   (
                     d[i-1, j] + 1,  // a deletion
                     d[i, j-1] + 1,  // an insertion
                     d[i-1, j-1] + 1 // a substitution
                   )
    }
  }
  return d[m,n]
}
```

Using *a single comprehension*, construct a matrix `d`, such that `d!(i,j)` is the Levenshtein distance between the first `i` characters of `s` and the first `j` characters of `t`, assuming

```
s :: String
t :: String
```

Remark: The distance between any two substrings must not be computed twice. Use laziness!

Hints:

- You may also assume a predefined function `minimum`:

  ```
  minimum :: [Int] -> Int
  ```

- You can access the `i`th character of string `s` with the expression `s!!(i-1)`.

6 Functions to relations

Consider the function

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Translate the function `fib` to a relation `fibo`, such that

```
fibo x y      is equivalent to      fib x == y
```

Remarks:

- You may not use the *functions* `fib` nor `(+)` in the translated version.
- Use a relation `plus` which coincides with addition:

  ```
  plus x y z     is equivalent to      x + y == z
  ```

Hint: you may want to start by writing the types of the *relations* `fibo` and `plus`.

You may write your answer in Curry syntax or Prolog syntax.