CHALMERS TEKNISKA HÖGSKOLA	August 2012.
Dept. of Computer Science and Engineering	Programming Paradigms
Jean-Philippe Bernardy	DAT121 / DIT331(GU)

Exam in Programming Paradigms

August 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. 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 Generalities

Suppose that you are facing the task to translate an application written in Erlang into an imperative language like C. This can be done by a sequence of transformations seen in the course. List a sequence of transformations seen in the course that, if applied *in the order given*, will transform the Erlang program to an imperative program easy to write in C syntax.

Reminder: Erlang is a functional language equipped with concurrency features such as channels and processes.

2 Explicit Stack

Consider the Ackermann function, defined as follows:

```
function a(m,n)
  if m = 0
    return n+1
  else if n = 0
    return a(m-1,1)
  else
    return a(m-1,a(m,n-1))
```

- Write a version of the Ackermann function that does not use recursion, but may use an explicit stack.
- Is it possible to tail-call optimise some recursive calls? List which calls which can be optimised:
  1. `a(m-1,1)`
  2. `a(m-1,a(m,n-1))`
  3. `a(m,n-1)`

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.

Note: Optimising all possible tail-calls is worth only 2 points.

3 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 *m(B*, B*);
   int f;
   void *n(B*, C*);
   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;
}
```

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

4 Closures

Consider the Haskell program:

```
g [] = []
g (p:ps) = p : g (filter (\x -> x ‘mod‘ p == 0) ps)

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

Transform the code to use explicit closures. That is, use a Haskell *data structure* to represent λ-expressions.
Remarks:

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

5 Continuations

Consider the following function, which computes the fibbonaci number of its argument.

```
fib 0 = return 1
fib n = do x1 <- fib (n-1)
           yield
           x2 <- fib (n-2)
           return (x1 + x2)
```

The above algorithm will be used on very big numbers, hence potentially running for a long time, concurrently with other processes. Therefore, the implementer of `fib` has decided to insert a call to `yield`, which performs no useful computational task, but gives the opportunity for the runtime environment to schedule another processes. (The computation of the fibbonaci number will be continued later.)
Transform `fib` to use explicit continuations. Remarks:

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

6 Relations to lists of successes

Consider the Curry program

```
ancestor a c = (parent a x & ancestor x c) | a =:= c
```

Convert the program from relational to functional style. That is, write a function `ancestors` that returns all the ancestors of a given person. More precisely, given a person `c`, construct the list of `x`s such that `ancestor x c` is true.

To do so, you may assume that you have at your disposal a function `parents` which returns the list of `x`s such that `parent x c` is true.

Hints:

- Remember that `|` stands for logical disjunction and `&` stands for logical conjunction.
- Begin by writing the types of the functions `ancestors` and `parents`.