Funtional and Logic Programming
Exercise Set 1

Tom Schrijvers Steven Keuchel
{tom.schrijvers,steven.keuchel}@ugent.be

Deadline: Tuesday, October 2nd, 2012, 11:59pm

General remarks

These are exercises you may use to practice the contents of the course. During the lab session you have the opportunity to ask questions to the assistant regarding the exercises and the course contents itself. Solutions are not graded and are not obligatory. However, you should try to solve and submit these anyway. That way you will have a better understanding of the topics and moreover get valuable feedback specific to your solutions from the assistant with pointers on how to even improve your solutions and/or coding style. Use the Indianio system for submissions

`https://indianio.ugent.be/`.

You are encouraged to share your ideas and solutions with others as you will also learn by explaining your thoughts. However, you should try to think and solve as much as possible yourself first before seeking advice from your fellow students or the assistant.

Rock - Paper - Scissors

In the Rock, Paper and Scissors game, two players choose one of the following gestures after counting to three:

- **A clenched fist** which represents a rock.
- **A flat hand** representing a piece of paper.
- **Index and middle finger extended** which represents a pair of scissors.

The result of a round is decided this way

- **Rock defeats scissors**, because a rock will blunt a pair of scissors.
- **Paper defeats rock**, because a paper can wrap up a rock.
- **Scissors defeat paper**, because scissors cut paper.
- **Otherwise**, the players chose the same gesture which is **a draw**.

In the following you are asked to declare datatypes as you have seen in the lecture. However, we have to add a **deriving** *Show* part immediately after the datatype declaration to let GHCi automatically print values of that datatype. The *Light* datatype declaration from the slides thus becomes

data *Light* = *Red* | *Orange* | *Green*
  deriving *Show*

The precise meaning of this will be made clear in the following lectures. For this exercise set you can simply add **deriving** *Show* to all your declarations.

1. Define a datatype *Move* with three choices $Rock, Paper$ and *Scissors* that represent the valid moves.

2. Write a function $beat :: Move \to Move$ that tells us the move that beats the given move and also a function $lose :: Move \to Move$ that tells us the move that will lose against the given move.

3. Define a datatype *Result* that represents the outcome of a round of Rock - Paper - Scissors.

4. Write a function $outcome :: Move \to Move \to Result$ that calculates the result of a round for the first player.

Lists, functions over lists and list comprehensions.

In the following you are asked to write functions over integers and list of intergers yourself. Try to reuse previously defined functions as much as possible and aim for a concise definition using list ranges and list comprehensions. *Provide type signatures.*

5. Write a function

$$allTrue :: [Bool] \to Bool$$

that evaluates to *True* iff all elements are *True* and a function *or*

$$oneTrue :: [Bool] \to Bool$$

that evaluates to *True* iff some element is *True*.

6. Write functions that satisfy the given identity. *Provide type signatures.*

(a) $append\ [1,2,3]\ [4,5] \equiv [1,2,3,4,5]$

(b) $produkt\ [1,2,3,4,5] \equiv 120$

(c) $factorial\ 6 \equiv 720$

(d) $insert\ 5\ [1,2,4,6] \equiv [1,2,4,5,6]$

(e) $sumIntegers\ a\ b \equiv a + ... + b$

Make sure that *sumIntegers* works for a $>$ b!

7.

(a) Write copies of the *sum* and *product* functions that work over floating point numbers.

$$sumf :: [Float] \to Float$$
$$productf :: [Float] \to Float$$

(b) Write a function *piSum* that approximates π using the formula

$$\frac{\pi}{8} \approx \frac{1}{1*3} + \frac{1}{5*7} + \frac{1}{9*11} + \ldots + \frac{1}{(4n+1)*(4n+3)}.$$

Note: Haskell does not perform implicit coercions of integers to floating point numbers, i.e. mixing integer with floating point arithmetic will result in a type error. Either write a function of type $(Float \to Float)$ to avoid this or write a functoion of type $(Float \to Int)$ and coerce explicitly using $(fromIntegral :: Int \to Float)$.

(c) Write a function *piProd* that approximates π using the formula

$$\frac{\pi}{4} = \frac{2*4}{3*3} * \frac{4*6}{5*5} * \frac{6*8}{7*7} * \frac{8*10}{9*9} * \ldots$$

Arithmetic expressions

The following *Exp* datatype encodes the abstract syntax for arithmetic expressions. Note that it is recursively defined: it is used on the right hand side of its own datatype declaration.

```
data Exp = Const Int
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
  deriving Show
```

8. Write an interpreter *eval* that evaluates arithmetic expressions.

$eval :: Exp \to Int$

Instead of evaluating the arithmetic expressions directly, we can also compile it to a program of a simple stack machine and subsequently execute the program. We represent programs as a list of instructions. The instructions *IAdd*, *ISub*, *IMul* take the two topmost elements from the stack, perform the corresponding operation and push the result onto the stack. The *IPush* instructions pushes the given immediate value onto the stack. A stack is modelled by a list of integers.

```
data Inst = IPush Int | IAdd | ISub | IMul
  deriving Show
type Prog  = [Inst]
type Stack = [Int]
```

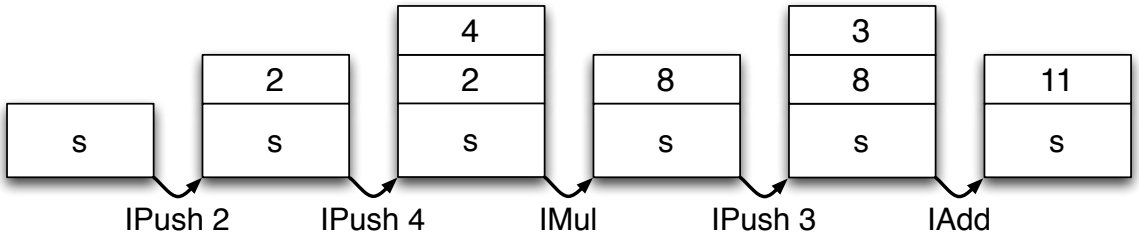
In this setting the arithmetic expression

$Add\ (Mul\ (Const\ 2)\ (Const\ 4))\ (Const\ 3)$

is equivalent to the stack program

$[IPush\ 2, IPush\ 4, IMul, IPush\ 3, IAdd]$

The stack program leaves the result on top of the stack. The stack machine performs the following steps when executing the program on an initial stack s



9. For this exercise you may use the magic stack value *runtimeError* to indicate runtime errors such as a stack underflow.

$$runtimeError :: Stack$$
$$runtimeError = error \texttt{ "Runtime error."}$$

(a) Write an execution function

$$execute :: Inst \to Stack \to Stack$$

that executes a single instruction.

(b) Write a function

$$run :: Prog \to Stack \to Stack$$

that runs a whole program on a given initial stack.

10.
Write a compiler

$$compile :: Exp \to Prog$$

that compiles arithmetic expressions to stack machine programs. The compiled progam should leave the result of the computation as the top element on the stack. Make sure that your compiler produces results equivalent to the interpreter, i.e. the following identity holds

$$\forall s :: Stack.\ \ run\ (compile\ e)\ s\ \equiv\ eval\ e : s.$$