Homework 2 (Abstract Syntax)

CS 381, Spring 2019

June 16, 2019

Prepared by

Adam Stewart *(stewaada)*
Anish Asrani *(asrania)*
Lucas Frey *(freyl)*
Mazen Alotaibi *(alotaima)*
Sergei Poliakov *(poliakos)*

CONTENTS

1 Exercise 1. Mini Logo

1.1 Define the abstract syntax for Mini Logo as a Haskell data type

```
data Cmd = Pen Mode
        | Moveto (Pos,Pos)
        | Def String Pars Cmd
        | Call Int Vals
        | Multiple Cmd Cmd
        deriving Show
data Mode = Up | Down deriving Show
data Pos = PN Int | PS String deriving Show
data Pars = ManyS String Pars | S String deriving Show
data Vals = ManyN Int Vals | SN Int deriving Show
```

1.2 Write a Mini Logo macro vector that draws a line from a given position (x1,y1) to a given position (x2,y2) and represent the macro in abstract syntax, that is, as a Haskell data type value

```
vector = Def "vector" (ManyS "x1" (ManyS "y1" (ManyS "x2" (S "y2"))))
                      (Multiple (Moveto (PS "x1", PS "y1"))
                                (Multiple (Pen Down)
                                          (Multiple (Moveto (PS "x2", PS "y2"))
                                                    (Pen Up))))
```

1.3 Define a Haskell function steps :: Int -> Cmd that constructs a Mini Logo program which draws a stair of n step

```
step :: Int -> Cmd
step 0 = Pen Up
step n = Multiple (Moveto ((PN (n-1)), PN n))
        (Multiple (Moveto (PN (n-1), PN (n-1)))
        (step (n-1)))

steps :: Int -> Cmd
steps n = Multiple (Moveto (PN n, PN n)) (Multiple (Pen Down) (step n))
```

2 Exercise 2. Digital Circuit Design Language

2.1 Define the abstract syntax for the above language as a Haskell data type

```
data Circuit = C Gates Links deriving Show
data Gates = MultipleG Int GateFn Gates
           | EmptyG
           deriving Show
data GateFn = And
            | Or
            | Xor
            | Not
            deriving Show
data Links = MultipleL Int Int Int Int Links
           | EmptyL
           deriving Show
```

2.2 Represent the half adder circuit in abstract syntax, that is, as a Haskell data type value

```
hac = C (MultipleG 1 Xor (MultipleG 2 And EmptyG))
        (MultipleL 1 1 2 1 (MultipleL 1 2 2 2 EmptyL))
```

2.3 Define a Haskell function that implements a pretty printer for the abstract syntax

```
printGates :: Gates -> String
printGates (MultipleG num gatefn gates) = show num ++ ":" ++ show gatefn ++ ";\n" ++ printGates gates
printGates EmptyG = ""

printLinks :: Links -> String
printLinks (MultipleL x1 y1 x2 y2 links) = "from " ++ show x1 ++ "." ++ show y1 ++ " to " ++ show x2 ++
    "." ++ show y2 ++ ";\n" ++ printLinks links
printLinks EmptyL = ""

printHac :: Circuit -> IO()
printHac (C gates links) = putStr((printGates gates) ++ (printLinks links))
```

3 Exercise 3. Designing Abstract Syntax

3.1 Represent the expression -(3+4)*7 in the alternative abstract syntax

```
as = Apply Multiply [Apply Negate [Apply Add [Num 3,Num 4]],Num 7]
```

3.2 What are the advantages or disadvantages of either representation?

The advantage of the alternative representation is that syntax is more expressive with fewer non-terminals. For instance, adding up a list of integers can be done with the following construction:

```
Apply Add list_of_integers
```

In the original syntax, this would require a recursive definition in order to add all the elements. e.g.

```
rec_add :: [Int] -> Expr
rec_add [x] = Num x
rec_add (x:xs) = Plus x (rec_add xs)
```

The syntax tree for the alternative syntax will be significantly smaller and more straight forward.

3.3 Define a function translate :: Expr -> Exp that translates expressions given in the first abstract syntax into equivalent expressions in the second abstract syntax

```
translate :: Expr -> Exp
translate (N x) = (Num x)
translate (Plus x y) = (Apply Add [(translate x), (translate y)])
translate (Times x y) = (Apply Multiply [(translate x), (translate y)])
translate (Neg x) = (Apply Negate [(translate x)])
```