<html class="gr__web_engr_oregonstate_edu"><head><style type="text/css"></style></head><body data-gr-c-s-loaded="true"><pre style="word-wrap: break-word; white-space: pre-wrap;">Note: This file can be directly loaded into ghci.

EXERCISE 1 [10+10+20 = 40 Points]

(a) [10 Points]

We need three data types to represent the nonterminals cmd, mode, and pos. Since the nonterminals pars and vals define lists of names and nums, respectively, we can use Haskell lists to represent those.

```
&gt; data Cmd = Pen Mode
&gt;          | MoveTo Pos Pos
&gt;          | Def  String [String] Cmd
&gt;          | Call String [Int]
&gt;          | Seq Cmd Cmd
&gt;
&gt; data Mode = Up | Down
&gt;
&gt; data Pos = I Int | Var String
```

One can also represent the lists using additional data types for the nonterminals pars and vals.

```
data Pars = Par String Pars | NoPars

data Vals = Val Int Vals | NoVals
```

In that case the definition of Cmd has to be changed as follows.

```
data Cmd = Pen Mode
         | MoveTo Pos Pos
         | Def  String Pars Cmd
         | Call String Vals
         | Seq Cmd Cmd
```

However, the first definition is definitely better for two reasons: First, it saves the definition of additional data types, and second, it allows the reuse of list operations when manipulating parameter and value lists in abstract syntax trees.

To make the following examples more readable when used in ghci, here is a simple Mini Logo pretty printer. The pretty printer adds curly brackets around bodies of macros.

```
&gt; ppCmd :: String -&gt; Cmd -&gt; String
&gt; ppCmd s (Pen Up)     = s++"pen up"
&gt; ppCmd s (Pen Down)   = s++"pen down"
&gt; ppCmd s (MoveTo x y) = s++"moveto ("++show x++","++show y++")"
&gt; ppCmd s (Def n xs c) = "def "++n++"("++ppPars xs++") {\n"++
&gt;                        ppCmd "   " c++"\n}"
&gt; ppCmd s (Call n xs)  = "call "++n++"("++ppArgs xs++")"
&gt; ppCmd s (Seq c c')   = ppCmd s c++";\n"++
&gt;                        ppCmd s c'
&gt;
&gt; printList :: (a -&gt; String) -&gt; String -&gt; [a] -&gt; String
&gt; printList f s [x]    = f x
&gt; printList f s (x:xs) = f x++s++printList f s xs
&gt;
&gt; ppPars :: [String] -&gt; String
&gt; ppPars xs = printList id "," xs
&gt;
&gt; ppArgs :: [Int] -&gt; String
&gt; ppArgs xs = printList show "," xs
&gt;
&gt; instance Show Cmd where
&gt;   show = ppCmd ""
&gt;
```

```
&gt; instance Show Pos where
&gt;   show (I i)   = show i
&gt;   show (Var s) = s
```

(b) [10 Points]

The macro definition in concrete Mini Logo syntax looks as follows.

```
  def vector (x1,y1,x2,y2) pen up; moveto (x1,y1); pen down; moveto (x2,y2)
```

The abstract syntax is represented as a value of the data type Cmd.

```
&gt; vector :: Cmd
&gt; vector = Def "vector" ["x1","y1","x2","y2"]
&gt;           (Seq (Pen Up)
&gt;                (Seq (MoveTo (Var "x1") (Var "y1"))
&gt;                     (Seq (Pen Down)
&gt;                          (MoveTo (Var "x2") (Var "y2")) )))
```

A representation that uses the explicit data type representation for the nonterminals pars and vals looks as follows.

```
vector :: Cmd
vector = Def "vector"
          (Par "x1" (Par "y1" (Par "x2" (Par "y2" NoPar))))
          (Seq (Pen Up)
               (Seq (MoveTo (Var "x1") (Var "y1"))
                    (Seq (Pen Down)
                         (MoveTo (Var "x2") (Var "y2")) )))
```

Writing nested applications of Seq (and Par, etc.) can get quite tedious. Here are two alternatives that show how to simplify writing syntax trees.

First, any binary function or constructor can be written using infix notation by enclosing it in backquotes. E.g. instead of div 7 3 we can write 7 `div` 3. Since Seq is a binary constructor, we can use this notation to write a sequence of Cmds in a way that is closer to the concrete syntax.

```
&gt; vector1 :: Cmd
&gt; vector1 = Def "vector" ["x1","y1","x2","y2"]
&gt;            (Pen Up                          `Seq`
&gt;             MoveTo (Var "x1") (Var "y1") `Seq`
&gt;             Pen Down                        `Seq`
&gt;             MoveTo (Var "x2") (Var "y2"))
```

A representation that uses the explicit data type representation for the nonterminals pars and vals looks as follows.

```
vector1 :: Cmd
vector1 = Def "vector" ("x1" `Par` "y1" `Par` "x2" `Par` "y2" `Par` NoPar)
           (Pen Up                          `Seq`
            MoveTo (Var "x1") (Var "y1") `Seq`
            Pen Down                        `Seq`
            MoveTo (Var "x2") (Var "y2"))
```

Another alternative is to put all commands into a Haskell list and use a function to convert this list into a tree built from Seq constructors.

```
&gt; vector2 :: Cmd
&gt; vector2 = Def "vector" ["x1","y1","x2","y2"]
&gt;            (foldr1 Seq [Pen Up,  MoveTo (Var "x1") (Var "y1"),
&gt;                         Pen Down,MoveTo (Var "x2") (Var "y2")])
```

The Haskell function foldr1 takes a binary operation (the constructor Seq in this case) and a list and combines all the elements of the list with this operation.

(c) [20 Points]

Here is a solution that creates the command sequences for the topmost step and connects it (using Seq) to the command sequence for the remaining stair. We use the representation shown in vector2 in part (b).

```
&gt; steps :: Int -&gt; Cmd
&gt; steps 1 = foldr1 Seq [Pen Up,  MoveTo (I 0) (I 0),
&gt;                       Pen Down,MoveTo (I 0) (I 1),
&gt;                                MoveTo (I 1) (I 1)]
&gt; steps n = Seq (steps (n-1))
&gt;               (foldr1 Seq [MoveTo (I (n-1)) (I n),
&gt;                            MoveTo (I n)     (I n)])
```

EXERCISE 2 [10+5+20 = 35 Points]

(a) [10 Points]

```
&gt; data Circuit = Circ Gates Links
&gt;
&gt; data Gates   = Gate Int GateFn Gates | EmptyGate
&gt;
&gt; data GateFn  = And | Or | XOr | Not
&gt;
&gt; data Links   = Link Int Int Int Int Links | EmptyLinks
```

The above solution uses the direct encoding of grammar productions by data types. We can simplify the definition by observing that both constructors EmptyGate and EmptyLinks are only used to represent lists of gates and links. Therefore an alternative definition could be obtained by defining data types representing individual gates and links and using type definitions to represent the list structures.

```
data Gate1 = Gate Int GateFn

data Link  = Link Int Int Int Int

type Gates = [Gate1]

type Links = [Link]
```

(b) [5 Points]

```
&gt; halfAdder = Circ gates links
&gt;             where  gates = Gate 1 XOr (Gate 2 And EmptyGate)
&gt;                    links = Link 1 1 2 1 (Link 1 2 2 2 EmptyLinks)
```

Using the alternative representation for lists shown in (a) we would define halfAdder as follows.

```
halfAdder = Circ [Gate 1 XOr,Gate 2 And] [Link 1 1 2 1, Link 1 2 2 2]
```

(c) [20 Points]

We define a pretty printing function for each nonterminal. ppGt and ppLink are defined as auxiliary functions. (Note: This pretty printer does not print a final semicolon, which the grammar in the homework produces. But this is a mistake in the original grammar.)

```
&gt; ppCircuit :: Circuit -&gt; String
&gt; ppCircuit (Circ gs ls) = ppGates gs++";\n"++ppLinks ls
&gt;
&gt; ppGt :: Int -&gt; GateFn -&gt; String
```

```
&gt; ppGt :: Int -&gt; GateFn -&gt; String
&gt; ppGt i g = show i++":"++ppGateFn g
&gt;
&gt; ppGates :: Gates -&gt; String
&gt; ppGates EmptyGate            = ""
&gt; ppGates (Gate i g EmptyGate) = ppGt i g
&gt; ppGates (Gate i g gs)        = ppGt i g++";\n"++ppGates gs
&gt;
&gt; ppLink :: Int -&gt; Int -&gt; Int -&gt; Int -&gt; String
&gt; ppLink fg fp tg tp = "from "++show fg++"."++show fp++" to "++
&gt;                         show tg++"."++show tp
&gt;
&gt; ppLinks :: Links -&gt; String
&gt; ppLinks EmptyLinks = ""
&gt; ppLinks (Link s1 s2 t1 t2 EmptyLinks) = ppLink s1 s2 t1 t2
&gt; ppLinks (Link s1 s2 t1 t2 ls) = ppLink s1 s2 t1 t2++";\n"++ppLinks ls
&gt;
&gt; ppGateFn :: GateFn -&gt; String
&gt; ppGateFn And = "and"
&gt; ppGateFn Or  = "or "
&gt; ppGateFn XOr = "xor"
&gt; ppGateFn Not = "not"
```

To install the pretty printer so that values of those data types will be automatically printed using them, we can define the data types as instances of the Show class as follows.

```
&gt; instance Show Circuit where
&gt;   show = ppCircuit
&gt;
&gt; instance Show Gates where
&gt;   show = ppGates
&gt;
&gt; instance Show Links where
&gt;   show = ppLinks
&gt;
&gt; instance Show GateFn where
&gt;   show = ppGateFn
```

EXERCISE 3 [5+6+14 = 25 Points]

Here are the data type definitions to allow this file to be loaded into Haskell.

```
&gt; data Expr = N Int
&gt;           | Plus  Expr Expr
&gt;           | Times Expr Expr
&gt;           | Neg Expr
&gt;             deriving Show
&gt;
&gt; data Op = Add | Multiply | Negate deriving Show
&gt; data Exp = Num Int
&gt;          | Apply Op [Exp]
&gt;            deriving Show
```

(a) [5 Points]

```
&gt; ex =  Apply Multiply [Apply Negate [Apply Add [Num 3, Num 4]],
&gt;                        Num 7]
```

(b) [6 Points]

Advantage: A potential advantage of the new syntax is that it is a bit easier to extend by new operations: Just add a constructor to Op, whereas adding a constructor to Expr requires also to be explicit about the exact number of arguments. In particular, for overloaded operations, the new representation is simpler since in the old representation multiple constructors would be needed.

Disadvantage: The new syntax does not enforce the correct number of arguments for operations, for example, it doesn't restrict Negate to a single operand or Plus to two operands. For example, what is the meaning of (Apply Negate [1,2]) or Apply Plus []? So the additional work required for the first representation (the disadvantage) pays off in the form of better consistency checking.

(c) [14 Points]

```
&gt; translate :: Expr -&gt; Exp
&gt; translate (N i)        = Num i
&gt; translate (Plus e e')  = Apply Add (map translate [e,e'])
&gt; translate (Times e e') = Apply Multiply (map translate [e,e'])
&gt; translate (Neg e)      = Apply Negate [translate e]
```

</pre></body></html>