Assignment 3

Robin Suda, Roy Thompson, and Hilario Mendez-Vallejo

BNF Grammar

Below is the proposed BNF grammar for our language. Use `StmtList` as the start symbol. A program in our language is considered to be a list of Statements.

```
StmtList    ::= Stmt StmtList | ε
Stmt        ::= Skip ";" | Assign ";" | Dec ";" | Block | Iter | Cond | Print ";" |
Incr ";"

Dec         ::= "int" identifier | "bool" identifier
Assign      ::= identifier "=" Expr
Incr        ::= identifier "++" | identifier "--" | "++" identifier | "--" identifier

Block       ::= "{" StmtList "}"

Cond        ::= "if" "(" Expr ")" Block "else" Block
          | "if" "(" Expr ")" Block

Iter        ::= ForIter | WhileIter
ForIter     ::= "for" "(" Expr ")" Block
WhileIter   ::= "while" "(" Expr ")" Block

Expr        ::= LogOr
LogOr       ::= LogOr "||" LogAnd | LogAnd
LogAnd      ::= LogAnd "&&" LogEq | LogEq
LogEq       ::= LogEq "==" RelOp | LogEq "!=" RelOp | RelOp
RelOp       ::= RelOp "<" AddOp | RelOp "<=" AddOp | RelOp ">" AddOp |
                RelOp ">=" AddOp | AddOp
AddOp       ::= AddOp "+" MulOp | AddOp "-" MulOp | MulOp
MulOp       ::= MulOp "*" UnaryOp | MulOp "/" UnaryOp | MulOp "%" UnaryOp | UnaryOp
UnaryOp     ::= "-" UnaryOp | "!" UnaryOp | ExpOp
ExpOp       ::= Base "^" ExpOp | Base
Base        ::= "|" Expr "|" | "(" Expr ")" | Value | identifier | Incr

Value       ::= integer | boolean

Print       ::= "print" "(" Expr ")" ";"
```

Type checking equations

Expr ::= LogOr

```
typeOf ([[ LogOr ]], m) = typeOf (LogOr, m)
```

LogOr ::= LogOr "||" LogAnd | LogAnd

```
typeOf ([[ LogOr || LogAnd ]], m) =
    let
        val t1 = typeOf (LogOr, m)
        val t2 = typeOf (LogAnd, m)
    in
        if t1 = t2 andalso t1 = BOOL then BOOL
        else ERROR
    end

typeOf ([[ LogAnd ]], m) = typeOf (LogAnd, m)
```

LogAnd ::= LogAnd "&&" LogEq | LogEq

```
typeOf ([[ LogAnd && LogEq ]], m) =
    let
        val t1 = typeOf (LogAnd, m)
        val t2 = typeOf (LogEq, m)
    in
        if t1 = t2 andalso t1 = BOOL then BOOL
        else ERROR
    end

typeOf ([[ LogEq ]], m) = typeOf (LogEq, m)
```

LogEq ::= LogEq "==" RelOp | LogEq "!=" RelOp | RelOp

```
typeOf ([[ LogEq == RelOp ]], m) =
    let
        val t1 = typeOf (LogEq, m)
        val t2 = typeOf (RelOp, m)
    in
        if t1 = t2 andalso t1 <> ERROR then BOOL
        else ERROR
    end

typeOf ([[ LogEq != RelOp ]], m) =
    let
        val t1 = typeOf (LogEq, m)
        val t2 = typeOf (RelOp, m)
    in
        if t1 = t2 andalso t1 <> ERROR then BOOL
        else ERROR
    end

typeOf ([[ RelOp ]], m) = typeOf (RelOp, m)
```

RelOp ::= RelOp "<" AddOp | RelOp "<=" AddOp | RelOp ">" AddOp | RelOp ">=" AddOp | AddOp

```
typeOf ([[ RelOp < AddOp ]], m) =
    let
        val t1 = typeOf ( RelOp, m )
        val t2 = typeOf ( AddOp, m )
    in
        if t1 = t2 andalso t1 = INT then BOOL
        else ERROR
    end

typeOf ([[ RelOp <= AddOp ]], m) =
    let
        val t1 = typeOf (RelOp, m)
        val t2 = typeOf (AddOp, m)
    in
        if t1 = t2 andalso t1 = INT then BOOL
        else ERROR
    end

typeOf ([[ RelOp > AddOp ]], m) =
    let
        val t1 = typeOf (RelOp, m)
        val t2 = typeOf (AddOp, m)
    in
        if t1 = t2 andalso t1 = INT then BOOL
        else ERROR
    end

typeOf ([[ RelOp >= AddOp ]], m) =
    let
        val t1 = typeOf (RelOp, m)
        val t2 = typeOf (AddOp, m)
    in
        if t1 = t2 andalso t1 = INT then BOOL
        else ERROR
    end

typeOf ([[ AddOp ]], m ) = typeOf (AddOp, m)
```

AddOp ::= AddOp "+" MulOp | AddOp "-" MulOp | MulOp

```
typeOf ([[ AddOp + MulOp ]], m) =
    let
        val t1 = typeOf (AddOp, m)
        val t2 = typeOf (MulOp, m)
    in
        if t1 = t2 andalso t1 = INT then INT
        else ERROR
    end

typeOf ([[ AddOp - MulOp ]], m) =
    let
        val t1 = typeOf (AddOp, m)
        val t2 = typeOf (MulOp, m)
    in
        if t1 = t2 andalso t1 = INT then INT
        else ERROR
    end

typeOf ([[ MulOp ]], m) = typeOf (MulOp, m)
```

MulOp ::= MulOp "*" UnaryOp | MulOp "/" UnaryOp | MulOp "%" UnaryOp | UnaryOp

```
typeOf ([[ MulOp * UnaryOp ]], m) =
    let
        val t1 = typeOf (MulOp, m)
        val t2 = typeOf (UnaryOp, m)
    in
        if t1 = t2 andalso t1 = INT then INT
        else ERROR
    end

typeOf ([[ MulOp / UnaryOp ]], m) =
    let
        val t1 = typeOf (MulOp, m)
        val t2 = typeOf (UnaryOp, m)
    in
        if t1 = t2 andalso t1 = INT then INT
        else ERROR
    end

typeOf ([[ MulOp % UnaryOp ]], m) =
    let
        val t1 = typeOf (MulOp, m)
        val t2 = typeOf (UnaryOp, m)
    in
        if t1 = t2 andalso t1 = INT then INT
        else ERROR
    end

typeOf ([[ UnaryOp ]], m) = typeOf (UnaryOp, m)
```

UnaryOp ::= "-" UnaryOp | "!" UnaryOp | ExpOp

```
typeOf ([[ - UnaryOp ]], m) =
    let
        val t1 = typeOf (UnaryOp, m)
    in
        if t1 = INT then INT
        else ERROR
    end

typeOf ([[ ! UnaryOp ]], m) =
    let
        val t1 = typeOf ( UnaryOp, m )
    in
        if t1 = BOOL then BOOL
        else ERROR
    end

typeOf ([[ ExpOp ]], m) = typeOf (ExpOp, m)
```

ExpOp ::= Base "^" ExpOp | Base

```
typeOf ([[ Base ^ ExpOp ]], m) =
    let
        val t1 = typeOf (Base, m)
        val t2 = typeOf (ExpOp, m)
    in
        if t1 = t2 andalso t1 = INT then INT
        else ERROR
    end

typeOf ([[ Base ]], m) = typeOf (Base, m)
```

Base ::= "|" Expr "|" | "(" Expr ")" | Value | identifier | Incr

```
typeOf ([[ | Expr | ]], m) =
    let
        val t1 = typeOf (Expr, m)
    in
        if t1 = INT then INT
        else ERROR
    end

typeOf ([[ ( Expr ) ]], m) =
    let
        val t1 = typeOf (Expr, m)
    in
        if t1 = INT then INT
        else if t1 = BOOL then BOOL
        else ERROR
    end

typeOf ([[ Value ]], m) = typeOf (Value, m)

typeOf ([[ identifier ]], m) = typeOf (identifier, m)

typeOf ([[ Incr ]], m) = typeOf (Incr, m)
```

Incr ::= identifier "" | identifier "–" | "" identifier | "–" identifier

```
typeOf ([[ identifier ++ ]], m) =
    let
        val t1 = typeOf (identifier, m)
    in
        if t1 = INT then INT
        else ERROR
    end

typeOf ([[ identifier -- ]], m) =
    let
        val t1 = typeOf (identifier, m)
    in
        if t1 = INT then INT
        else ERROR
    end

typeOf ([[ ++identifier ]], m) =
    let
        val t1 = typeOf (identifier, m)
    in
        if t1 = INT then INT
        else ERROR
    end

typeOf ([[ --identifier ]], m) =
    let
        val t1 = typeOf (identifier, m)
    in
        if t1 = INT then INT
        else ERROR
    end
```

Value ::= integer | boolean

```
typeOf ([[ integer ]], m) = INT

typeOf ([[ boolean ]], m) = BOOL
```

StmtList ::= Stmt StmtList | ε

```
typeCheck ([[ Stmt StmtList ]], m0) =
    let
        val m1 = typeCheck (Stmt, m0)
        val m2 = typeCheck (StmtList, m1)
    in
        m2
    end

typeCheck ([[  ]], m) = m
```

Stmt ::= Skip ";" | Assign ";" | Dec ";" | Block | Iter | Cond | Print ";"

```
typeCheck ([[ Skip ; ]], m) = m

typeCheck ([[ Assign ; ]], m) = m

typeCheck ([[ Dec ; ]], m) = m

typeCheck ([[ Block ]], m) = m

typeCheck ([[ Iter ]], m) = m

typeCheck ([[ Cond ]], m) = m

typeCheck ([[ Print ; ]], m) = m
```

Dec ::= "int" identifier | "bool" identifier

```
typeCheck ([[ int identifier ]], m) = updateEnv(id, INT, new( ), m)

typeCheck ([[ bool identifier ]], m) = updateEnv(id, BOOL, new(), m)
```

Assign ::= identifier "=" Expr

```
typeCheck ([[ identifier = Expr ]], m) =
    let
        val t1 = typeOf (Expr, m)
        val t2 = getType ( accessEnv (identifier, m))
    in
        if t1 = t2 then m
        else raise model_error
    end
```

Block ::= "{" StmtList "}"

```
typeCheck ([[ { StmtList } ]], m) = m
```

Cond ::= "if" "(" Expr ")" Block "else" Block | "if" "(" Expr ")" Block

```
typeCheck ([[ if Expr Block else Block ]], m) =
    let
        val t = typeOf (Expr, m0)
        val m1 = typeCheck (Block1, m0)
        val m2 = typeCheck (Block2, m0)
    in
        if t = BOOL then m0
        else raise model_error
    end
```

Iter ::= ForIter | WhileIter

```
typeCheck ([[ ForIter ]], m) = m

typeCheck ([[ WhileIter ]], m) = m
```

ForIter ::= "for" "(" Expr ")" Block

```
typeCheck ([[ for ( Expr ) Block ]], m) =
    let
        val t = typeOf (Expr, m0)
        val m = typeCheck (Block1, m0)
    in
        if t = BOOL then m0
        else raise model_error
    end
```

WhileIter ::= "while" "(" Expr ")" Block

```
 typeCheck ([[ while ( Expr ) Block ]], m) =
    let
        val t = typeOf (Expr, m0)
        val m1 = typeCheck (Block1, m0)
    in
        if t = BOOL then m0
        else raise model_error
    end
```

Print ::= "print" "(" Expr ")"

```
typeCheck ([[ print ( Expr )  ]], m) =
    let
        val t = typeOf (Expr, m0)
    in
        if t = ERROR then raise model_error
        else m0
    end
```