Assignment 3

Programming Languages - Group 8

Goal: Create a set of denotational semantic equations capable of executing programs in your language abstractly at the level of type.

Grammar

program ::= stmtList

typeCheck([[stmtList$_1$]], m) = typeCheck(stmtList$_1$, m)

stmtList ::= stmt ; stmtList | ε

typeCheck([[stmt$_1$; stmtList$_1$]], m0) =

```
    let
        val m1 = typeCheck(stmt1, m0)
        val m2 = typeCheck(stmtList1, m1)
    in
        m2
    end
```

typeCheck ([[ε]], m) = m

stmt ::= declaration | assign | conditional | decoratedId | iterative | block | output

typeCheck([[declaration$_1$]], m) = typeCheck(declaration$_1$, m)

typeCheck([[assign$_1$]], m) = typeCheck(assign$_1$, m)

typeCheck([[conditional$_1$]], m) = typeCheck(conditional$_1$, m)

typeCheck([[decoratedId$_1$]], m) = typeCheck(decoratedId$_1$, m)

typeCheck([[iterative$_1$]], m) = typeCheck(iterative$_1$, m)

typeCheck([[$block_1$]], m) = typeCheck($block_1$, m)

typeCheck([[$output_1$]], m) = typeCheck($output_1$, m)

declaration ::= “int” id | “bool” id

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

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

assign ::= id “=” expression

typeCheck([[id = $expression_1$]], m)=

```
     let
          val t1 = typeOf(expr1,m)
          val t2 = getType(accessEnv(id,m))
     in
          if t1 = t2 then m else raise model_error
     end
```

conditional ::= if expression then block | if expression then block else block

typeCheck([[if $expression_1$ then $block_1$]], m0) =

```
     let
          val t = typeOf(expression1, m0)
          val m1 = typeCheck(block1, m0)
     in
          if t = BOOL then m0 else raise model_error
     end
```

typeCheck([[if $expression_1$ then $block_1$ else $block_2$]], m0) =

```
     let
          val t = typeOf(expression₁, m0)
          val m1 = typeCheck(block₁, m0)
          val m2 = typeCheck(block₂, m0)
     in
          if t = BOOL then m0 else raise model_error
     end
```

decoratedId ::= ++ id | -- id | id ++ | id -

```
typeCheck( [[ ++ id ]], m0) =
     let
          val t = typeOf(id, m0)
     In
          if t = INT then m0 else raise model_error
     end
typeCheck( [[ -- id ]], m0) =
     let
          val t = typeOf(id, m0)
     In
          if t = INT then m0 else raise model_error
     end
typeCheck( [[ id ++ ]], m0) =
     let
          val t = typeOf(id, m0)
     In
          if t = INT then m0 else raise model_error
```

```
     end
typeCheck( [[ id -- ]], m0) =
     let
          val t = typeOf(id, m0)
     In
          if t = INT then m0 else raise model_error
     end
```

iterative ::= whileLoop ::= while (expression) block |
forLoop ::= for (assign ; expression ; decoratedId) block

```
typeCheck( [[ while ( expression₁ ) block₁]], m0) =
    let
        val t = typeOf(expression₁,m0)
        val m1 = typeCheck(block₁,m0)
    in
        if t = BOOL then m0 else raise model_error
    end

typeCheck( [[ for (assign₁; expression₁; decoratedId₁ ) block₁]] ,
m0) =
    let
          val m1 = typeCheck(assign₁, m0)
          val t = typeOf(expression₁, m0)
          val m2 = typeCheck(decoratedId₁, m0)
          val m3 = typeCheck(block₁, m0)
     in
```

```
          if t = BOOL then m0 else raise model_error
    end
```

block ::= {stmtList}

typeCheck([[{stmtList$_1$}]], m0) = typeCheck(stmtList$_1$, m0)

output ::= print (expression)

typeCheck([[print (expression$_1$)]], m0) =

```
    let
        val t = typeOf(expression1,m0)
    in
        if t = BOOL then m0 else raise model_error
    end
```

expression ::= (expression)

typeOf ([[(expression$_1$)]], m) = typeOf (expression$_1$, m)

expression ::= expression "||" logicalOR | logicalOR

typeOf ([[expression$_1$ || logicalOR$_1$]], m) =

```
    let
        val t1 = typeOf(expression1,m)
        val t2 = typeOf(logicalOR1,m)
    in
        if t1 = t2 andalso t1 = BOOL then BOOL
        else ERROR
    end
```

```
typeOf([[logicalOR₁]], m) = typeOf(logicalOR₁, m)
```

logicalOR ::= logicalOR && logicalAND | logicalAND

```
typeOf ( [[ logicalOR₁ && logicalAND₁ ]], m) =
    let
        val t1 = typeOf(logicalOR₁,m)
        val t2 = typeOf(logicalAND₁,m)
    in
        if t1 = t2 andalso t1 = BOOL then BOOL
        else ERROR
    end
typeOf([[logicalAND₁]], m) = typeOf(logicalAND₁, m)
```

logicalAND ::= logicalAND == equality | equality

```
typeOf ( [[ logicalAND₁ == equality₁ ]], m) =
    let
        val t1 = typeOf(logicalAND₁,m)
        val t2 = typeOf(equality₁,m)
    in
        if t1 = t2 then BOOL
        else ERROR
    end
typeOf([[equality₁]], m) = typeOf(equality₁, m)
```

equality ::= equality < additive | equality > additive | additive

```
typeOf ( [[ equality₁ < additive₁ ]], m) =
    let
        val t1 = typeOf(equality₁,m)
        val t2 = typeOf(additive₁,m)
    in
        if t1 = t2 andalso t1 = INT then BOOL
        else ERROR
    end
typeOf ( [[ equality₁ > additive₁ ]], m) =
    let
        val t1 = typeOf(equality₁,m)
        val t2 = typeOf(additive₁,m)
    in
        if t1 = t2 andalso t1 = INT then BOOL
        else ERROR
    end
typeOf([[additive]], m) = typeOf(additive, m)
```

additive :: additive + multiplicative | additive - multiplicative | multiplicative

```
typeOf( [[ additive₁ + multiplicative₁ ]], m) =
    let
        val t1 = typeOf(additive₁,m)
        val t2 = typeOf(multiplicative₁,m)
    in
        if t1 = t2 andalso t1 = INT then INT
```

```
        else ERROR
    end
typeOf( [[ additive₁ - multiplicative₁ ]], m) =
    let
        val t1 = typeOf(additive₁,m)
        val t2 = typeOf(multiplicative₁,m)
    in
        if t1 = t2 andalso t1 = INT then INT
        else ERROR
    end
typeOf([[multiplicative₁]], m) = typeOf(multiplicative₁, m)
```

multiplicative ::= multiplicative * unary | multiplicative div unary | multiplicative mod unary | unary

```
typeOf( [[ multiplicative₁ * unary₁ ]], m) =
    let
        val t1 = typeOf(multiplicative₁,m)
        val t2 = typeOf(unary₁,m)
    in
        if t1 = t2 andalso t1 = INT then INT
        else ERROR
    end
typeOf( [[ multiplicative₁ div unary₁ ]], m) =
    let
        val t1 = typeOf(multiplicative₁,m)
        val t2 = typeOf(unary₁,m)
```

```
    in
        if t1 = t2 andalso t1 = INT then INT
        else ERROR
    end
typeOf( [[ multiplicative₁ mod unary₁ ]], m) =
    let
        val t1 = typeOf(multiplicative₁,m)
        val t2 = typeOf(unary₁,m)
    in
        if t1 = t2 andalso t1 = INT then INT
        else ERROR
    end
typeOf( [[unary₁]], m) = typeOf(unary₁, m)
```

unary ::= - unary | ! unary | exponent

```
typeOf ( [[ -unary₁ ]], m) =
    let
        val t1 = typeOf(unary₁,m)
    in
        if t1 = INT then INT
        else ERROR
    end
typeOf ( [[ !unary₁ ]], m) =
    let
        val t1 = typeOf(unary₁,m)
    in
```

```
        if t1 = INT then INT
        else ERROR
    end
typeOf( [[exponent₁]], m) = typeOf(exponent₁, m)
```

exponent :: factor ** exponent | factor

```
typeOf ( [[ factor₁ ** exponent₁ ]], m) =
    let
        val t1 = typeOf(factor₁,m)
        val t2 = typeOf(exponent₁,m)
    in
        if t1 = t2 andalso t1 = INT then INT
        else ERROR
    end
typeOf( [[factor₁]], m) = typeOf(factor₁, m)
```

factor ::= integer_value | boolean_value | id | (expr)

```
typeOf( [[integer_value]], m) = INT
typeOf( [[boolean_value]], m) = BOOL
typeOf( [[ id ]], m) = getType(accessEnv(id,m))
typeOf( [[ ( expr₁ ) ]], m) = typeOf(expr₁ ,m)
```