Milestone 2

Roy Thompson, Robin Suda, and Hilario Mendez-Vallejo

Denotational Semantics

Expressions

- We define a valuation function M such that `M: parse_expression * model -> model`
- We define a valuation function E' such that `E': parse_expression * model -> value * model`
- M is defined as a set of equations.
- E' is defined as a set of equations.
- We assume a Turing Complete context in which computation occurs.

Denotational Semantics

StmtList (List of Statements)

```
M([[stmt stmtList]], m0) =
    let
        val m1 = M(stmt1, m0)
        val m2 = M(stmtList1, m1)
    in
        m2
    end

| M([[stmt1]], m0) =
    let
        val m1 = M(stmt1, m0)
    in
        m1
    end

| M([[ε]], m) = m
```

Stmt (Statement)

```
M([[skip ";"]], m0) =
    let
        val m1 = M(skip, m0)
    in
        m1
    end
```

```
| M([[assign ";"]], m0) =
    let
        val m1 = M(assign, m0)
    in
        m1
    end

| M([[dec ";"]], m0) =
    let
        val m1 = M(dec, m0)
    in
        m1
    end

| M([[block]], m0) =
    let
        val m1 = M(block, m0)
    in
        m1
    end

| M([[iter]], m0) =
    let
        val m1 = M(iter, m0)
    in
        m1
    end

| M([[cond]], m0) =
    let
        val m1 = M(cond, m0)
    in
        m1
    end

| M([[print]], m0) =
    let
        val m1 = M(print, m0)
    in
        m1
    end

| M([[expr ";"]], m0) =
    let
        val m1 = M(expr, m0)
    in
        m1
    end
```

Dec (Data Type)

```
M([["int" id]], m0) =
    let
        val m1 = updateEnv(id, int, new(), m0)
    in
        m1
    end

| M([["bool" id]], m0) =
    let
        val m1 = updateEnv(id, bool, new(), m0)
    in
        m1
    end
```

Assign (Assignment)

```
M([[id "=" expr1]], m0) =
    let
        val (v, m1) = E'(expr1, m0)
        val loc     = getLoc(accessEnv(id, m1))
        val m2      = updateStore(loc, v, m1)
    in
        m2
    end
```

Id (Value/Variable)

```
M([[id]], m0) = m0
| E([[id "++"]], m) =
    let
        val v1 = E(id, m)
    in
        v1 + 1
    end
| E([[id "--"]], m) =
    let
        val v1 = E(id, m)
    in
        v1 - 1
    end
| E([["++" id]], m) =
    let
        val v1 = E(id, m)
    in
        1 + v1
    end
| E([["--" id]], m) =
    let
        val v1 = E(id, m)
    in
        1 - v1
    end
```

Block (Block)

```
M([["{" stmtList1 "}"]], m0) =
    let
        val m1 = (stmt1, m0)
    in
        m1
    end
```

Cond (Conditional)

```
M([[if]], m0) =
    let
        val m1 = M(if, m0)
    in
        m1
    end

| M([[if else]], m0) =
    let
        val m1 = M(if else, m0)
    in
        m1
    end
```

Expr (Expression)

```
E'( [[ Expr1 ]], m0 ) = E'( expr1, m0 )
| E'( [[ int ]], m0 ) = ( int, m0 )
| E'( [[ bool ]], m0 ) = ( bool, m0 )
```

LogOr (Logical Or)

```
E'( [[ LogOr1 || LogAnd1 ]], m0 ) =
    let
        val (v1, m1) =  E'( LogOr1, m0 )
        val (v2, m2) = E'( LogAnd1, m1 )
    in
        (v1 orelse v2, m2)
    end

| E'( [[ LogAnd1 ]] ) = E'( LogAnd1, m )
```

LogAnd (Logical And)

```
E'( [[ LogAnd1 "&&" LogEq1 ]], m0 ) =
    let
        val (v1, m1) = E'( LogAnd1, m0 )
        val (v2, m2) = E'( LogEq1, m1 )
    in
        ( v1 andalso v2, m2 )
    end

| E'( [[ LogEq1 ]], m ) = E'( LogEq1, m )
```

LogEq (Logical Equality)

```
E'( [[ LogEq "==" RelOp ]], m0 ) =
    let
        val (v1, m1) = E'( LogEq, m0 )
        val (v2, m2) = E' ( LogEq, m1 )
    in
        ( v1 = v2, m2 )
    end
| E'( [[ LogEq "!=" RelOp ]], m0) =
    let
        val (v1, m1) = E'( LogEq, m0 )
        val (v2, m2) = E' ( LogEq, m1 )
    in
        ( v1 <> v2, m2 )
    end
| E' ( [[ RelOp ]], m ) = E' ( RelOp, m )
```

RelOp (Relational Operators)

```
E' ( [[ RelOp < AddOp ]], m0 ) =
    let
        val (v1, m1) = E'(RelOp, m0)
        val (v2, m2) = E'(AddOp, m1)
    in
        ( v1 < v2 , m2 )
    end
| E' ( [[ RelOp <= AddOp ]], m0 ) =
    let
        val (v1, m1) = E'(RelOp, m0)
        val (v2, m2) = E'(AddOp, m1)
    in
        ( v1 <= v2 , m2 )
    end
| E' ( [[ RelOp > AddOp ]], m0 ) =
    let
        val (v1, m1) = E'(RelOp, m0)
        val (v2, m2) = E'(AddOp, m1)
    in
        ( v1 > v2 , m2 )
    end
| E' ( [[ RelOp >= AddOp ]], m0 ) =
    let
        val (v1, m1) = E'(RelOp, m0)
        val (v2, m2) = E'(AddOp, m1)
    in
        ( v1 >= v2 , m2 )
    end
| E' ( [[ AddOp ]], m ) = E'( AddOp, m )
```

AddOp (Additive Operators)

```
E'( [[ AddOp "+" MulOp ]], m0 ) =
    let
        val (v1, m1) = E'(AddOp, m0)
        val (v2, m2) = E'(MulOp, m1)
    in
        (v1 + v2, m2)
    end
| E'( [[ AddOp "-" MulOp ]], m0 ) =
    let
        val (v1, m1) = E'(AddOp, m0)
        val (v2, m2) = E'(MulOp, m1)
    in
        (v1 - v2, m2)
    end
| E'( [[ MulOp ]], m ) = E'( MulOp, m )
```

MulOp (Multiplicitive Operators)

```
E'( [[ MulOp "*" ExpOp ]], m0 ) =
    let
        val (v1, m1) = E'(MulOp, m0)
        val (v2, m2) = E'(ExpOp, m1)
    in
        (v1 * v2, m2)
    end
| E'( [[ MulOp "/" ExpOp ]], m0 ) =
    let
        val (v1, m1) = E'(MulOp, m0)
        val (v2, m2) = E'(ExpOp, m1)
    in
        (v1 / v2, m2)
    end
| E'( [[ MulOp "%" ExpOp ]], m0 ) =
    let
        val (v1, m1) = E'(MulOp, m0)
        val (v2, m2) = E'(ExpOp, m1)
    in
        (v1 mod v2, m2)
    end
| E'( [[ ExpOp ]], m ) = E'( ExpOp, m )
```

ExpOp (Exponentiation)

```
fun power(x, 0) = 1 | power(x, n) = x * power(x,n-1);

E'( [[ AbsOp "^" ExpOp ]], m0 ) =
    let
        val (v1, m1) = E'(AbsOp, m0)
        val (v2, m2) = E'(ExpOp, m1)
    in
        (power(v1, v2), m2)
    end
| E'( [[ AbsOp ]], m ) = E'( AbsOp, m )
```

AbsOp (Absolute Value)

```
E'( [[ "|" AbsOp "|" ]], m0 ) =
    let
        val (v1, m1) = E'(AbsOp, m0)
    in
       ( v1 * ((v>0) - (v<0)) , m1)
    end
| E'( [[ Expr ]], m ) = E'( Expr, m )
```

> Referenced https://stackoverflow.com/questions/9772348/get-absolute-value-without-using-abs-function-nor-if-statement

Print (Print Values)

```
M([["print" "(" expr1 ")" ";"]], m0) =
    let
        val m1 = print(expr1)
    in
        m1
    end
```