Group 7: Milestone 2

Persons Involved: Chloe Parker, Matthew Heftie, Andrew Carlson

Note: This milestone contains the revised version of our grammar from M. The semantic equation(s) corresponding to each grammar rule can be found under the bolded grammar rule.

Denotational Semantics of BNF Grammer

Program ::= stmtList

M([[stmtList]], m0) =

```
    let
        val m1 = M( stmtList, m0)
    in
        m2
    end
```

stmtList :: = ε | Statement stmtList

M([[stmt1 stmtList1]], m0) =

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

Statement :: = Declaration | Assignment | Conditional | Iterative | Print |Block

M([[Declaration1]], m) = M(Declaration1, m)

M([[Assignment1]], m) = M(Assignment1, m)

M([[Conditional1]], m) = M(Conditional1, m)

M([[Iterative1]], m) = M(Iterative1, m)

M([[Print1]], m) = M(Print1, m)

M([[Block1]], m) = M(Block1, m)

Declaration ::= int id | bool id

M([[int id]], m) = updateEnv(id, int, new(), m)

M([[bool id]], m) = updateEnv(id, bool, new(), m)

Print ::= print(expression)

M([[Print expression1]], m0)=

Let

val(v,m1) = E'(Expression1, m0)

in

print(v);

m1

end

Assignment ::= id = Expression

M([[id = Expression1]], m0) =

let

val (v,m1) = E'(Expression1, m0)

val loc = getLoc(accessEnv(id, m1))

val m2 = updateStore(loc, v, m1)

in

m2

end

Conditional ::= if Expression Block else Statement | if Expression else Block

M([[if Expression1Block1 else Block2]], m0) =

let

val (v,m1) = E'(expression1, m0)

```
        in
                if v then
                        M([[Block1]]), m1)
                else
                        M([[Block1]]), m1)
        end
M([[if Expression1Block1, m0) =
        let
                val(v,m1) = E'(expression1, m0)
        in
                if v then
                        M([[Block1]]), m1)
                else
                        m1
        end
```

Iterative ::= "for (" declaration "; "comparison"; " Increment "):" Block | "while " Expression ":" Block

```
M( [[ for(dec1; cond1; inc1):  stmt1 ]], m ) =
        let
                val m1 = M(dec1, m0)
        in
                N (cond1, inc1, stmt1, m1)
        end
N(cond1, inc1, stmt1, m0 ) =
        let
                val (v,m1) = E'( expr1, m0 )
```

```
        in
                if v then
                        let
                                val m2 = M(stmt1, m1)
                                val m3 = M(inc1, m2)
                        in
                                N ( expr1, stmt1, M(stmt1, m3) )
                        end
                else m1
        end
M( [[ while expr1: stmt1 ]], m0 ) =
        Let
                val (v,m1) = E'( expr1, m0 )
        in
                if v then
                        let
                                val m2 = M( stmt1, m1 )
                                val m3 = M( [[ while expr1 do stmt1 ]], m2 )
                        in
                                m3
                        end
                else m1
        end
```

Increment ::= PostInc | PreInc

M([[PostInc]], m) = M(PostInc, m)

M([[PreInc]], m) = M(PreInc, m)

Expression ::= Expression Expression | “(”Expression”)” | Id | Literal | Add | Sub | Mult | Div| Exponent | Equality | Inequality | LessThan | GreaterThan | LessThanequal | GreaterThanEqual

E’([[Id1]], m0) =

```
    let
        val location = getLoc(accessEnv(Id1, m0))
        val v = accessStore(loc, m0)
    in
        (v, m0)
    end
```

M ([[Literal]]), m) = (Literal, m)

M ([[Add]]), m) = (Add, m)

M ([[Sub]]), m) = (Sub, m)

M ([[Mult]]), m) = (Mult, m)

M ([[Div]]), m) = (Div, m)

M ([[Exponent]]), m) = (Exponent, m)

M ([[Equality]]), m) = (Equality, m)

M ([[Inequality]]), m) = (Inequality, m)

Block ::= stmtList

M([[stmtList1]]), m) = M(stmtlist1, m)

Literal ::= int | bool

E([[int]], m) = int

E([[bool]], m) = bool

Id ::= varName

M([[varName1]]), m) = M(varName1, m)

Comparison :== And | Or | Not

M([[And]]), m) = M(And, m)

M([[Or]]), m) = M(Or, m)

M([[Not]]), m) = M(Not, m)

Denotational Semantics for Operational Expressions

PostInc ::= Id++

E’([[Id1++]], m0) =

```
    let
        val location = getLoc(accessEnv(Id1, m0))
        val current = accessStore(location, m0)
        val new = current + 1
        val m1 = updateStore(location, new, m0)
    in
        (current, m1)
    end
```

PostDec ::= Id--

E’([[Id1--]], m0) =

```
    let
        val location = getLoc(accessEnv(Id1, m0))
        val current = accessStore(location, m0)
        val new = current - 1
        val m1 = updateStore(location, new, m0)
    in
        (current, m1)
    end
```

PreInc ::= ++Id

E’([[++Id1]], m0) =

let

val location = getLoc(accessEnv(Id1, m0))

val current = accessStore(location, m0)

val new = current + 1

val m1 = updateStore(location, new, m0)

in

(new, m1)

end

PreDec ::= --Id

E’([[--Id1]], m0) =

let

val location = getLoc(accessEnv(Id1, m0))

val current = accessStore(loc, m0)

val new = current - 1

val m1 = updateStore(loc, new, m0)

in

(new, m1)

end

Add ::= Expression+ Expression

E’([[expression1 + expression2]], m0) =

let

val (v1, m1) = E’(expression1, m0)

val (v2, m2) = E’(expression2, m1)

in

(v1 + v2, m2)

End

Sub ::= Expression - Expression

E’([[expression1 - expression2]], m0) =

let

val (v1, m1) = E’(expression1, m0)

val (v2, m2) = E’(expression2, m1)

in

(v1 - v2, m2)

End

Mult ::= Expression * Expression

E’([[expression1 * expression2]], m0) =

let

val (v1, m1) = E’(expression1, m0)

val (v2, m2) = E’(expression2, m1)

in

(v1 * v2, m2)

End

Div ::= expression % expression

E’([[expression1 % expression2]], m0) =

let

val (v1, m1) = E’(expression1, m0)

val (v2, m2) = E’(expression2, m1)

in

(v1 % v2, m2)

End

Exponent ::= expression ** expression

E’([[expression1 ** expression2]], m0) =

```
    let
        val (v1, m1) = E’(expression1, m0)
        val (v2, m2) = E’(expression2, m1)
        val result = exp(v1,v2)
    in
        (result, m2)
    End
```

And ::= expression AND expression

E’([[expression1 AND expression2]]) =

```
    let
        val (v1, m1) = E’(expression1, m0)
    in
        if v1 = false then
            (false, m1)
        else
            let
                val (v2, m2) = E’(expression2, m1)
            in
                (v2, m2)
            end
    end
```

Or ::= expression OR expression

E’([[expression1 OR expression2]]) =

```
    let
```

```
        val (v1, m1) = E'(expression1, m0)
    in
        if v1 = true then
            (true, m1)
        else
            let
                val (v2, m2) = E'(expression2, m1)
            in
                (v2, m2)
            end
    end
```

Not ::= not expression

```
E'([[not expression1]]) =
    let
        val (v1, m1) = E'(expression1, m0)
    in
        if v1 = true then
            (false, m1)
        else
            (true, m1)
    end
```

Equality ::= expression == expression

```
E'([[expression1 == expression2]], m0) =
    let
        val (v1, m1) = E'(expression1, m0)
        val (v2, m2) = E'(expression2, m1)
```

in

(v1 == v2, m2)

End

Inequality ::= expression != expression

E’([[expression1 != expression2]], m0) =

let

val (v1, m1) = E’(expression1, m0)

val (v2, m2) = E’(expression2, m1)

in

(v1 != v2, m2)

End

LessThan ::= expression < expression

E’([[expression1 < expression2]], m0) =

let

val (v1, m1) = E’(expression1, m0)

val (v2, m2) = E’(expression2, m1)

in

(v1 < v2, m2)

End

GreaterThan ::= expression > expression

E’([[expression1 > expression2]], m0) =

let

val (v1, m1) = E’(expression1, m0)

val (v2, m2) = E’(expression2, m1)

in

(v1 > v2, m2)

End

LessThanEqual ::= expression <= expression

E’([[expression1 <= expression2]], m0) =

let

val (v1, m1) = E’(expression1, m0)

val (v2, m2) = E’(expression2, m1)

in

(v1 <= v2, m2)

End

GreaterThanEqual ::= expression >= expression

E’([[expression1 >= expression2]], m0) =

let

val (v1, m1) = E’(expression1, m0)

val (v2, m2) = E’(expression2, m1)

in

(v1 >= v2, m2)

End