Rebecca Frederick
EECS 345
Written Exercise 1
February 5, 2016

1.
```
<C>  →  <V> = <D> | <V>
<V>  →  x | y | z
<D>  →  <E> ? <D> : <D> | <E>
<E>  →  <E> || <F> | <F>
<F>  →  <F> && <G> | <G>
<G>  →  !<G> | <H>
<H>  →  (<H>) | <I>
<I>  →  true | false
```

2. Static Semantic Attributes:

type	=	{integer, double}	(synthesized)
typetable(<var>)	=	{integer, double, error}	(inherited)
inittable(<var>)	=	{true, false, error}	(inherited)
typebinding	=	(<var>, {integer, double})	(synthesized)
initialized	=	(<var>, {true, false})	(synthesized)

Attribute Rules:

<start$_1$> $\rightarrow$ <stmt$_3$> ; <start$_3$>
<start$_1$>.type := N/A
<start$_1$>.typetable(<var>) := <stmt$_3$>.typetable
<start$_1$>.inittable(<var>) := <stmt$_3$>.initvar
<start$_1$>.typebinding := N/A
<start$_1$>.initialized := N/A

<stmt$_3$>.type := N/A
<stmt$_3$>.typetable := <start$_1$>.typetable
<stmt$_3$>.inittable := <start$_1$>.inittable
<stmt$_3$>.typebinding := N/A
<stmt$_3$>.initialized := N/A

<start$_3$>.type := N/A
<start$_3$>.typetable := <stmt$_3$>.typetable $\cup$ <start$_1$>.typetable
<start$_3$>.inittable := <stmt$_3$>.inittable $\cup$ <start$_1$>.inittable
<start$_3$>.typebinding := N/A
<start$_3$>.initialized := N/A

<start$_2$> $\rightarrow$ <stmt$_4$>
<start$_2$>.type := N/A
<start$_2$>.typetable(<var>) := $\emptyset$
<start$_2$>.inittable(<var>) := $\emptyset$
<start$_2$>.typebinding := N/A
<start$_2$>.initialized := N/A

<stmt$_4$>.type := N/A
<stmt$_4$>.typetable := <start$_2$>.typetable

<stmt$_4$>.inittable := <start$_2$>.inittable
<stmt$_4$>.typebinding := N/A
<stmt$_4$>.initialized := N/A

<stmt$_1$> $\rightarrow$ *<declare$_2$>*
<stmt$_1$>.type := N/A
<stmt$_1$>.typetable(<var>) := <start>.typetable
<stmt$_1$>.inittable(<var>) := <start>.inittable
<stmt$_1$>.typebinding := N/A
<stmt$_1$>.initialized := N/A

<declare$_2$>.type := N/A
<declare$_2$>.typetable(<var>) := <stmt$_1$>.typetable
<declare$_2$>.inittable(<var>) := <stmt$_1$>.inittable
<declare$_2$>.typebinding := N/A
<declare$_2$>.initialized := N/A

<stmt$_2$> $\rightarrow$ *<assign$_2$>*
<stmt$_2$>.type := N/A (I don't think it makes sense to keep propagating that attribute to this non-terminal (the synthesized attribute type should stop propagating upwards after the non-terminal which uses it to add an entry to the typetable and do error-checking))
<stmt$_2$>.typetable(<var>) := <start>.typetable
<stmt$_2$>.inittable(<var>) := <start>.inittable
<stmt$_2$>.typebinding := N/A
<stmt$_2$>.initialized := N/A

<assign$_2$>.type := (see <assign$_1$>)
<assign$_2$>.typetable(<var>) := <stmt$_2$>.typetable
<assign$_2$>.inittable(<var>) := (<stmt$_2$>.inittable - (M_{name}(<var>), false)) $\cup$ (M_{name}(<var>), true) (checking would need to be done here to ensure that the entry (M_{name}(<var>), false) actually exists)
<assign$_2$>.typebinding := N/A
<assign$_2$>.initialized := N/A

<declare$_1$> $\rightarrow$ *<type$_3$> <var>*
<declare$_1$>.type := <type$_3$>.type
<declare$_1$>.typetable(<var>) := <stmt>.typetable $\cup$ (M_{name}(<var>), <type$_3$>.type) (checking would need to be done here to ensure that <var> has not already been declared, i.e., the inittable entry for this variable must be (M_{name}(<var>), error))
<declare$_1$>.inittable(<var>) := <stmt>.inittable $\cup$ (M_{name}(<var>), false)
<declare$_1$>.typebinding := (M_{name}(<var>), <type$_3$>.type)
<declare$_1$>.initialized := <var>.initialized

<type$_3$>.type := (see <type$_1$> and <type$_2$>)
<type$_3$>.typetable(<var>) := <declare$_1$>.typetable
<type$_3$>.inittable(<var>) := <declare$_1$>.inittable
<type$_3$>.typebinding := N/A
<type$_3$>.initialized := N/A

<type$_1$> $\rightarrow$ *int*
<type$_1$>.type := int
<type$_1$>.typetable(<var>) := <declare>.typetable
<type$_1$>.inittable(<var>) := <declare>.inittable

<type$_1$>.typebinding := N/A
<type$_1$>.initialized := N/A

<type$_2$> $\rightarrow$ *double*
<type$_2$>.type := double
<type$_2$>.typetable(<var>) := <declare>.typetable
<type$_2$>.inittable(<var>) := <declare>.inittable
<type$_2$>.typebinding := N/A
<type$_2$>.initialized := N/A

<assign$_1$> $\rightarrow$ *<var> <expression$_3$>*
<assign$_1$>.type := <var>.type (checking would need to be done here to ensure that <var> actually has a type (i.e., it has already been declared), and also to ensure that <var>.type = ¡expression¿.type)
<assign$_1$>.typetable(<var>) := <stmt>.typetable
<assign$_1$>.inittable(<var>) := <stmt>.inittable
<assign$_1$>.typebinding := <var>.typebinding
<assign$_1$>.initialized := (M_{name}(<var>), true)

<expression$_3$>.type := (see <expression$_1$> and <expression$_2$>)
<expression$_3$>.typetable(<var>) := <assign$_1$>.typetable $\cup$ <var>.typetable
<expression$_3$>.inittable(<var>) := <assign$_1$>.inittable $\cup$ <var>.inittable
<expression$_3$>.typebinding := <value$_4$>.typebinding (¡expression¿ eventually has to produce ¡value¿; however this attribute only makes sense for ¡expression¿ if ¡value¿ produces <var> (rather than ¡integer¿ or ¡float¿))
<expression$_3$>.initialized := <value$_4$>.initialized

<expression$_1$> $\rightarrow$ *<expression$_4$> <op> <expression$_5$>*
<expression$_1$>.type :=

```
switch (<op>):
    case +:
    case -:
        if <expression4>.type = float || <expression5>.type = float
            <expression1>.type = float
        else
            <expression1>.type = int
        break;
    case *:
    case /:
        <expression1>.type = float
```

<expression$_1$>.typetable(<var>) := <assign>.typetable
<expression$_1$>.inittable(<var>) := <assign>.inittable
<expression$_1$>.typebinding := N/A
<expression$_1$>.initialized := N/A

<expression$_4$>.type := (see <expression$_1$> and <expression$_2$>)
<expression$_4$>.typetable(<var>) := <expression$_1$>.typetable
<expression$_4$>.inittable(<var>) := <expression$_1$>.inittable
<expression$_4$>.typebinding := (see <expression$_2$> and comment for <expression$_3$>)
<expression$_4$>.initialized := (see <expression$_2$> and comment for <expression$_3$>)

<expression$_5$>.type := (see <expression$_1$> and <expression$_2$>)
<expression$_5$>.typetable(<var>) := <expression$_1$>.typetable $\cup$ <op>.typetable

<expression$_5$>.inittable(<var>) := <expression$_1$>.typetable $\cup$ <op>.inittable
<expression$_5$>.typebinding := (see <expression$_2$> and comment for <expression$_3$>)
<expression$_5$>.initialized := (see <expression$_2$> and comment for <expression$_3$>)

<expression$_2$> $\rightarrow$ *<value$_4$>*
<expression$_2$>.type := <value$_4$>.type
<expression$_2$>.typetable :=

```
if produced by <assign1>:
    <expression2>.typetable := <assign1>.typetable \union \nterm{var}.typetable
elif produced by <expression1>:
    if <expression2> is <expression4>:
        <expression2>.typetable := <expression1>.typetable
    elif <expression2> is <expression5>:
        <expression2>.typtable := <expression1>.typetable \union <op>.typetable
```

<expression$_2$>.inittable := (same rules as typetable)
<expression$_2$>.typebinding := <value$_4$>.typebinding
<expression$_2$>.initialized := <value$_4$>.initalized

<value$_4$>.type := (see <value$_1$>, <value$_2$>, and <value$_3$>)
<value$_4$>.typetable(<var>) := <expression$_3$>.typetable
<value$_4$>.inittable(<var>) := <expression$_3$>.inittable
<value$_4$>.typebinding := (see <value$_1$>)
<value$_4$>.initialized := (see <value$_1$>)

<value$_1$> $\rightarrow$ *<var>*
<value$_1$>.type := <var>.type
<value$_1$>.typetable(<var>) := <expression>.typetable
<value$_1$>.inittable(<var>) := <expression>.inittable
<value$_1$>.typebinding := <var>.typebinding
<value$_1$>.initialized := <var>.initialized

<value$_2$> $\rightarrow$ *<integer>*
<value$_2$>.type := <integer>.type
<value$_2$>.typetable(<var>) := <expression>.typetable
<value$_2$>.inittable(<var>) := <expression>.inittable
<value$_2$>.typebinding := *N/A*
<value$_2$>.initialized := *N/A*

<value$_3$> $\rightarrow$ *<float>*
<value$_3$>.type := <float>.type
<value$_3$>.typetable(<var>) := <expression>.typetable
<value$_3$>.inittable(<var>) := <expression>.inittable
<value$_3$>.typebinding := *N/A*
<value$_3$>.initialized := *N/A*

3. (a) 'The type of the expression must match the type of the variable in all assignment statements'

   1. <assign$_1$>: <var>.type = <expression$_3$>.type

   (b) 'A variable must be declared before it is used'

    1. `<assign`$_1$`>`: `<var>`.typetable != ‘Error’

(c) ‘A variable must be assigned a value as its first use in the program’

    1. `<assign`$_1$`>`: if `<var>`.inittable = ‘Error’

4. Loop Invariants:

Outer (while) Loop Goal: The elements $A[0 \ldots n-1]$ are sorted in non-decreasing order

Outer (while) Loop Invariant: The elements $A[bound \ldots n-1]$ are in non-decreasing order $\wedge$ the elements $A[t \ldots bound-1]$ have yet to be sorted.

(The last condition may be redundant but I felt it necessary to include t in the outer loop invariant since it is initiallized outside of the inner loop and also interacts with a variable (*bound*) in the outer loop.)

Inner (for) Loop Goal: the elements $A[t \ldots n-1]$ are sorted in non-decreasing order

Inner (for) Loop Invariant: The elements $A[bound \ldots n-1]$ are in non-decreasing order $\wedge$
$A[0 \ldots i] \leq A[t]$ $\wedge$
$A[t] \leq A[bound]$.

Precondition: $n \geq 0$ and A contains n elements indexed from 0

```
bound = n;
while (bound > 0) {

  // Assume Outer Loop Invariant is true
  t = 0;

  for (i = 0; i < bound - 1; i++) {

    // Assume Inner Loop Invariant is true
    if (A[i] > A[i+1]) {

      // WP (Inner):
      // A[bound...n-1] are in non-decreasing order  ∧
      // A[0...i-1] ≤ A[i]   ∧
      // A[i] ≤ A[bound]
      swap = A[i];

      // WP (Inner):
      // A[bound...n-1] are in non-decreasing order  ∧
      // A[0...i-1] ≤ A[i+1]   ∧
      // A[i+1] ≤ A[bound]
      A[i] = A[i+1];

      // WP (Inner):
      // A[bound...n-1] are in non-decreasing order  ∧
      // A[0...i] ≤ swap   ∧
      // swap ≤ A[bound]
      A[i+1] = swap;

      // WP (Inner):
      // A[bound...n-1] are in non-decreasing order  ∧
      // A[0...i] ≤ A[i+1]   ∧
      // A[i+1] ≤ A[bound]
      t = i + 1;
    }
    // (loop termination: i=bound-1, t='the last i+1 for which A[i] > A[i+1]')
    // i=bound-1  ∧  A[t] ≤ A[bound]  ∧  A[0...i] ≤ A[t]  ∧
```

```
        //     A[bound...n-1] are in non-decreasing order →
        //          A[t...n-1] are sorted in non-decreasing order
        // i++
    }
    // WP (Outer):
    // A[bound...n-1] are in non-decreasing order ∧
    //     A[t...bound-1] have yet to be sorted
    bound = t;
}
// (loop termination: bound=0, t=0)
// bound=0 ∧
// A[0...n-1] are sorted in non-decreasing order ∧
//     A[0···-1] have yet to be sorted (trivially true) →
//          array A is sorted in non-decreasing order
```

Postcondition: $A[0] \leq A[1] \leq \cdots \leq A[n-1]$ (i.e., array A is sorted in non-decreasing order)

5. M_{state}(<var> = <expression>, S) =

```
{
    // test that <var> is a legal name in the language
    if M_name(<var>) = 'Error'
        return 'Error'

    // test that <var> has already been declared
    if Lookup(M_name(<var>), S) = 'Error'
        return 'Error'

    // calculate the value of <expression> using the old state
    V = M_value(<expression>, S)
    if V = 'Error'
        return 'Error'

    // calculate a new state including any side effects from evaluating <expression>
    S_1 = M_state(<expression>, S)

    // remove <var> from the new state
    Remove(M_name(<var>), S)

    // return the new state with the updated value of <var> added
    return Add(M_name(<var>), V, S_1)

}
```

M_{state}(if <condition> then <statement$_1$> else <statement$_2$>, S) =

```
{
    S_1 = M_state(<condition>, S)

    if M_boolean(<condition>, S_1) = true
        return M_state(<statement_1>, S_1)
    else if M_boolean(<condition>, S_1) = false
        return M_state(<statement_2>, S_1)
    else
        return 'Error'
}
```

M_{state}(while <condition> <loop body>, S) =

```
{
    S_1 = M_state(<condition>, S)
```

```
    if M_boolean(<condition>, S_1) = true
        // evaluate the loop body and call the while-loop again
        return M_state(while <condition> <loop body>, M_state(<loop body>, S_1))
    else if M_boolean(<condition>, S_1) = false
        return S_1
    else
        return 'Error'
}
```