Rebecca Frederick
EECS 345
Written Exercise 1
February 3, 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:

<start₁> → *<stmt₃>* ; *<start₃>*
`<start1>.type` := *N/A*
`<start1>.typetable(<var>)` := `<stmt3>.typetable`
`<start1>.inittable(<var>)` := `<stmt3>.initvar`
`<start1>.typebinding` := *N/A*
`<start1>.initialized` := *N/A*

`<stmt3>.type` := *N/A*
`<stmt3>.typetable` := `<start1>.typetable`
`<stmt3>.inittable` := `<start1>.inittable`
`<stmt3>.typebinding` := *N/A*
`<stmt3>.initialized` := *N/A*
`<start3>.type` := *N/A*
`<start3>.typetable` := `<stmt3>.typetable` ∪ `<start1>.typetable`
`<start3>.inittable` := `<stmt3>.inittable` ∪ `<start1>.inittable`
`<start3>.typebinding` := *N/A*
`<start3>.initialized` := *N/A*
<start₂> → *<stmt₄>*
`<start2>.type` := *N/A*
`<start2>.typetable(<var>)` := ∅
`<start2>.inittable(<var>)` := ∅
`<start2>.typebinding` := *N/A*
`<start2>.initialized` := *N/A*

`<stmt4>.type` := *N/A*
`<stmt4>.typetable` := `<start2>.typetable`
`<stmt4>.inittable` := `<start2>.inittable`

<stmt$_4$>.typebinding := N/A
<stmt$_4$>.initialized := N/A
<start$_3$>.type := N/A
<stmt$_1$> $\rightarrow$ *<declare$_2$>*
<stmt$_1$>.type := N/A
<stmt$_1$>.typetable(<var>) := $\emptyset$(inherited from <start>)
<stmt$_1$>.inittable(<var>) := $\emptyset$(inherited from <start>)
<stmt$_1$>.typebinding := N/A
<stmt$_1$>.initialized := N/A

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

<declare$_1$>.type :=
<declare$_1$>.typetable(<var>) :=
<declare$_1$>.inittable(<var>) :=
<declare$_1$>.typebinding :=
<declare$_1$>.initialized :=

<type$_1$>.type :=
<type$_1$>.typetable(<var>) :=
<type$_1$>.inittable(<var>) :=
<type$_1$>.typebinding :=
<type$_1$>.initialized :=

<type$_2$>.type :=
<type$_2$>.typetable(<var>) :=
<type$_2$>.inittable(<var>) :=
<type$_2$>.typebinding :=
<type$_2$>.initialized :=

<assign$_1$>.type :=
<assign$_1$>.typetable(<var>) :=
<assign$_1$>.inittable(<var>) :=
<assign$_1$>.typebinding :=
<assign$_1$>.initialized :=

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

<expression$_2$>.type :=
<expression$_2$>.typetable(<var>) :=
<expression$_2$>.inittable(<var>) :=
<expression$_2$>.typebinding :=
<expression$_2$>.initialized :=

<value$_1$>.type :=

```
<value1>.typetable(<var>) :=
<value1>.inittable(<var>) :=
<value1>.typebinding :=
<value1>.initialized :=

<value2>.type :=
<value2>.typetable(<var>) :=
<value2>.inittable(<var>) :=
<value2>.typebinding :=
<value2>.initialized :=

<value3>.type :=
<value3>.typetable(<var>) :=
<value3>.inittable(<var>) :=
<value3>.typebinding :=
<value3>.initialized :=
```

Table 1: Attribute Rules

3. asdf

4. Loop Invariants:

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

   Outer (while) Loop Invariant: The elements $A[bound \dots n-1]$ are in non-decreasing order $\wedge$ the elements $A[t \dots bound-1]$ 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 $A[t \dots n-1]$ are sorted in non-decreasing order

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

   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) =

```
{
    S1 = M_state(<condition>, S)

    if M_boolean(<condition>, S1) = true
        return M_state(<statement1>, S1)
    else if M_boolean(<condition>, S1) = false
        return M_state(<statement2>, S1)
    else
        return 'Error'
}
```

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

```
{
    S1 = M_state(<condition>, S)

    if M_boolean(<condition>, S1) = true
        return M_state(while <condition> <loop body>, M_state(<condition>, S1))
    else if M_boolean(<condition>, S1) = false
        return S1
    else
        return 'Error'
}
```