Static Semantics

Stat Sem Def

The only static semantics we impose that can be processed by the compiler (static) are proper use of variables.

- **Variables**
  - Variables have to be defined before used first time.
  - Variable name can only be defined once in a scope.

Two options for variable scope.

- **Global** option for variables
  - There is only one scope
- **Local** option for variables
  - Variables outside of a block are global
  - Variables in a block are scoped in this block
  - Rules as in C (smaller scope hides the outer/global scope variable)

Support and processign Global

Software support

- Use any container ST for names such as array, list, etc. with the following interface. It shows String as the parameter, which is the ID token instance, but it could include line number for more detailed error reporting.
  - `insert(String)` - insert the string if not already there or error if already there (you may return fail indication or issue detailed error here and exit)
  - `Bool verify(String)` - return true if the string is already in the container variable and false otherwise (I suggest you return false indicator rather than issue detailed error here with exit but either way could possibly work if you assume that no one checks verify() unless to process variable use)

Static semantics

- Instantiate STV for variables
- Traverse the tree and perform the following (looks like preorder traversal) based on the subtree you are visiting
  - If visiting <vars> or its subtree and you find ID token then call STV.insert(ID) // this is variable definition
  - Otherwise (you are under <stats> and not under another <vars> if you find token ID
    - call STV.verify(ID)

Support and Processing Local

You may process all variables using local scope rules, or process variables in the outside <vars> as global and all other variables as local. This describes the latter.

Software support

Implement a stack adapter according to the following

- Stack item type is String or whatever was your ID token instance. You may also store line number or the entire token for more detailed error messaging
- You can assume no more than 100 items in a program and generate stack overflow if more
- Interface
  - `void push(String);`
    - just push the argument on the stack
  - `void pop(void);`
    - pop, nothing returned
  - `int find(String);`
    - the exact interface may change, see below
    - find the first occurrence of the argument on the stack, starting from the top and going down to the bottom of the stack
    - return the distance from the TOS (top of stack) where the item was found (0 if at TOS) or -1 if not found

Static semantics

- Perform left to right traversal, and perform different actions depending on subtree and node visited
  - When working in the outer <vars> subtree
    - process as in the global option (or process as local if desired)
  - When working in a <block>
    - set varCount=0 for this block
    - under <vars>
      - upon each `v` variable definition
      - when `varCount>0` call `find(v)` and error/exit if it returns non-negative `number < varCount` (means that multiple definition in this block)
      - `push(v)` and `varCount++`
    - otherwise (variable use, suppose variable instance is `v`)
      - `find(v)`, if -1 try STV.`verify(v)` (if STV used for the global variables) and error if still not found
    - call `pop()` `varCount` times when leaving a block (note that `varCount` much be specific to each block)

Test Files - Global Error

```
Var x , y , x
Begin
  Output 1 ;
End

-----------------------------

Var x , y
Begin
  Var z, x
  Output 1 ;
End

-----------------------------

Var x , y , z
Begin
```

```
  Var a , b , b
  Output 1 ;
End

------------------------------

Var x , y
Begin
   Output z ;
End

------------------------------

Var x , y
Begin
   Output x + z ;
End

------------------------------

Var x , y
Begin
   Output 1 ;
   If [ x > 1 + z ]
   Begin
      Output 1 ;
   End
End

------------------------------

Var x , y
Begin
   Output 1 ;
   If [ x > 1 ]
   Begin
      Var z , y
      Output 1 ;
   End
End
```

Test Files - Global Good

```
Var x , y
Begin
  Output x + y ;
End

------------------------------

Var x , y
Begin
  Var z
  Output x + y + z ;
End

------------------------------

Var x , y
Begin
  Begin
    Var z
    Output z ;
  End
```

```
  Output z ;
End
```

Test Files - Local Bad

```
Var x , y , x
Begin
  Output 1 ;
End

-----------------------------

Var x , y
Begin
  Var z, x , x
  Output 1 ;
End

-----------------------------

Var x , y , z
Begin
  Var a , b
  Output w ;
End

-----------------------------

Var x , y
Begin
   Begin
      Var w
      Output x ;
   End
   Output w ;
End
```

Test Files - Local Good

```
Var x , y , z
Begin
  Var a , b
  Output x + a ;
  Begin
    Var x
    Output x ;
  End
  Begin
    Var x
    Output x ;
  End
  Output x ;
End
```