Invocation:

> statSem [*file*]

with *file* as before **in P2**

Wrong invocations will not be graded.

Grading

Graded 90% execution 10% structure/standards. We will use programs with all tokens separated by spaces. We will test with the file argument as previous projects.

Stat Semantics Definition

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

    1. A variable has to be defined before being used for the first time (after satisfying syntax)
        - **whole Identifier := Integer** defines the variable named **Identifier**. We can assume that it will be initially set to the value in **Integer.**
        - **Identifier** showing up in any statement is the variable's use
    2. Variable name can only be defined once in a scope but can possibly be reused in another scope.
- **Global** option for all variables (max 75)

    - There is only one scope, the global scope, regardless of where a variable is defined.
    - Note that this option in P3 will likely force you to use the easier option in P4 resulting in about 10-15% loss on P4.
- **Local** option (max 100)
    - Variables defined before the **program** keyword are considered in the global scope, those inside of any block are considered scoped in that block.
        - The very first block also has full scope.
    - We will use the same scoping rules as in C - a variable is for use in a scope if it is defined in this scope, any of its enclosing scopes, or the global scope (static scoping rules).

    - This option here will allow you to use the full option in P4 or the easier one

Do not display the tree any more.

P3 Global Option (75 point option)

Software support

- Use any container (called ST here for Symbol Table) 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/char or the entire token for more detailed error reporting. This container will process identifier tokens only.
    - **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 ST as STV for variables
- Traverse the tree and perform the following (looks like preorder traversal) based on the subtree you are visiting
    - If visiting <vars> and you find identifier token (variable definition) then call **STV.insert(String)** on the token - this is variable definition
    - Otherwise (you are not in <vars>) if you find identifier token call **STV.verify(String)** - this is variable use

P3 Local Option (100 point option)

You may process all variables using local scope rules, or process variables before **program** as global and all other variables as local.

Software support

Implement a stack adapter according to the following

- Stack item type is String or whatever was your identifier token instance - the stack will process identifiers. 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);
        - just remove
    - 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 <vars> before program, process the identifiers there as in the global option (or process as local if desired using similar mechanism as below)
    - when working in a <block>
        - set varCount=0 for this block
        - in <vars> of this <block> assume ID token v

        - 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++
    - when identifier token found in any non-var node (variable use)
        - call find(v), if -1 try STV.verify(v) (if STV used for the global variables) and error if still not found
    - when leaving the block, call pop() varCount times (note that varCount must be specific to each block)

P3 Testing

Here are some example files. **If you see potential problems please check with me. These programs should NOT generate scanner or parser errors.**

In this section you should test programs to see if they obey local and global rules. Make sure and check for double defined (redefined) variables and variables defined as the same name but in different scopes.

```
# p3g1, this should result in an error in all options #
program
begin
input x ;
end
# p3g2, this should be fine#
whole _x := 10 ;
program
begin
output _x ;
end
```

```
# p3g3, redefining a variable, should be an error #
whole yx := 5 ;
program
begin
  whole yx := 7 ;
  assign yx = yx * 5 - 3 + 8 / : 11 ;
  output yx ;
end
# p2g4, the same variable defined in different blocks #
program
begin
  whole ab := 5 ;
  while [ ab < 10 ]
  begin
     whole yx := 5 ;
     assign ab = yx + 1 ;
  end ;

  while [ ab < 10 ]
  begin
     whole yx := 7 ;
     assign yx = yx + 1 ;
  end ;

  output ab ;
end
```