ASSIGNMENT #4: Completion of AST

Given that assignment #3 is complete and functional, this assignment should not take very much time. It is a marriage of assignments 2 and 3, with the addition of statements, thus completing the front end part for the entire C language!

The goal of assignment #4 is to be able to accept any valid C program as input (subject to the limitations and exclusions mentioned) and produce an AST representation of all things in the program which will require code generation. Specifically, this means statements, including expression statements, for statements, if-else statements, etc. You'll be able to use most of your code from assignment #2 in recognizing expressions.

In assignment #3, you accepted and processed declarations. All of that functionality must be retained going forward. Declarations do not contribute to the statement AST (with the possible exception of initialized declarations), but the information which you have stored in the symbol table will be vital for processing the AST to generate quads (assignment #5).

You have also already developed a framework for printing out AST representations of types and expressions. This can be easily extended to handle statements as well. You should print the AST of *each function, after that function has been seen.* Here's an example:

```
************** input program **************
int a;
int f()
{
        int b, c;
          int z();
        for(a=1;a<b;a++)
                z(a);
}


*************** sample output **************
**** This first part is the same as assignment #3 ***

a is defined at <stdin>:1 [in global scope starting at <stdin>:1] as a
variable with stgclass extern  of type:
  int
f is defined at <stdin>:3 [in global scope starting at <stdin>:1] as a
extern   function returning
   int
  and taking unknown arguments
b is defined at <stdin>:4 [in function scope starting at <stdin>:3] as a
variable with stgclass auto  of type:
  int
c is defined at <stdin>:4 [in function scope starting at <stdin>:3] as a
```

```
variable with stgclass auto  of type:
  int
z is defined at <stdin>:5 [in function scope starting at <stdin>:3] as a
extern   function returning
   int
  and taking unknown arguments

***** and here is the stuff that is new for assignment 4 ****

AST Dump for function
 LIST {
  FOR
  INIT:
   ASSIGNMENT
    stab_var name=a def @<stdin>:1
    CONSTANT: (type=int)1
  COND:
   COMPARISON OP <
    stab_var name=a def @<stdin>:1
    stab_var name=b def @<stdin>:4
  BODY:
   FNCALL, 1 arguments
    stab_fn name=z def @<stdin>:5
   arg #1=
    stab_var name=a def @<stdin>:1
  INCR:
   UNARY OP POSTINC
    stab_var name=a def @<stdin>:1
 }
```

Your output doesn't have to match the output above; it is provided just to give an example. I have used the expedient of denoting all AST node types with CAPITAL LETTERS, following which any fields of that node appear set off by a space of indentation. Other representations are certainly possible. E.g. you could use curly braces to delimit the fields of an AST node.

Your recursive code to print an AST should be "defensive". If you come across an AST node with an incorrect type, report that but do not stop immediately with a fatal error. This will be helpful for debugging. Print as much information as will reasonably fit on the output lines.

On the course web site you'll find a few more test cases and sample output.

ISSUES WHICH MAY COME UP

- **Lists**: You'll need a way of representing lists of things in the AST, e.g. lists of statements (compound statement). Consider stringing a linked list through the AST nodes, or a dynamically allocated (and re-allocable) array of AST node pointers.
- **Identifiers**: When building ASTs for expressions, you should now resolve identifiers through the symbol table, whereas in assignment #2 you used just the bare lexeme. In this way, the AST will have access to the identifier type information (and other attributes, such as storage class.) In the example above, note the node for the identifier `a` is a symbol table entry (`stab_var` for symbol table -- variable). The node for `z` is `stab_fn` because that identifier in this scope represents the name of a function, not a variable. In both cases, the type of the identifier was entered into the symbol table at the time of declaration (this was assignment #3).
- **struct / union** : Eventually when you see a . operator, you'd like to have the symbol table entry corresponding to the member. But that's beyond the scope of this assignment, because you'd have to have the expression type engine working to determine the type of the lhs of the . or -> operator. So you'll have to just store the member name for now. Example:

```
struct s {
          int a;
} *p[10];

int f()
{
 int i;
          p[i]->b++;
}

AST Dump for function
 LIST {
  UNARY OP POSTINC
   INDIRECT SELECT, member b
    DEREF
     BINARY OP +
      stab_var name=p def @<stdin>:3
      stab_var name=i def @<stdin>:7

 }
```

Note that the AST node shows "member b" even though no such member exists! Later, in assignment 5, you'd be able to detect this as an error.

THINGS YOU DON'T HAVE TO DO

The same things which were optional in Assignment #3 continue to be optional. These include: abritrary constant expressions in array declarators, C99 variable-length arrays, qualifiers, inline functions, enums, bit fields, prototypes/formal param lists, K&R syntax, typedefs, initialized declarators, implicit function declarations. Structures and unions will not ultimately be something that will be required for the final "hello world" type project. If you are behind schedule, you could drop support for these.

A few more words about initialized declarators

If you choose to handle initializers: an initialized declaration of a variable of auto storage class behaves as if there were a declaration, followed by an assignment statement. This means that you'd need to insert the assignment AST into the body of the function, at the point at which the declaration was seen (remember, standard C only allows declarations at the beginning of a block, but C99 allows them anywhere). Example:

```
f()
{
          int i=3+4;
          i++;
}

          AST Dump for function
           LIST {
            LIST {
             ASSIGNMENT
              stab_var name=i def @<stdin>:3
              BINARY OP +
               CONSTANT: (type=int)3
               CONSTANT: (type=int)4
            }
            UNARY OP POSTINC
             stab_var name=i def @<stdin>:3
           }
```

A given declaration may have more than one initialized declarator, resulting in multiple initialization expressions.

When the variable is non-local, the initializer must evaluate to a constant, which ultimately gets translated to an assembly language pseudo-opcode. That's the topic of unit #6, so it's OK for now to choke on: `static int a=5;`

Array initializers are special:

```
f()
{
 char a[]="FOOBAR";

}
```

`a` is an array of `auto` storage class. In classic C before C99, the above declaration was not permitted for auto storage class. In modern C, this results in something similar to:

```
f()
{
 char *a=alloca(7);
 _builtin_memcpy(a,"FOOBAR",7);
}
```

Clearly, this is an example where the initialized declaration does not equate to a declaration followed by an assignment, as `a="FOOBAR"` is not a valid expression. You are certainly not required to handle this.