ASSIGNMENT #4: Completion of AST

The goal of assignment #4 is to complete the front end! You will then 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. If you made good design choices in assignments 2 and 3, then #4 should not take very much time at all. You should be able to use nearly 100% of the code from those two assignments. You will just need to add to the definition of a `statement` to include all the other forms of statement besides `expression ';'` and `compound_statement`. For reference, these are: goto, do-while, while, for, if-else, switch, return, continue, break, and labelled statements (including `case` and `default`).

In assignment #3, you accepted and processed declarations. Good news: all of that functionality must be retained going forward; your time was not spent in vain. While declarations do not contribute to the part of the AST that will result in IR generation (with the possible exception of initialized declarations), nonetheless the information about type and storage class (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, 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 f
 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 continued the convention first displayed in my sample for assignment #2: all AST node types are displayed with CAPITAL LETTERS, following which any fields of that node or children of that node appear set off by a space of indentation (e.g. INIT, COND, INCR, and BODY) Other representations are certainly possible. E.g. you could use curly braces to delimit the fields of an AST node. In my data structure, an AST node can point directly to a symbol table entry, hence the lines such as `stab_var name=a` which denotes a variable named a. You may have taken other approaches, such as an intermediate node to represent a VARIABLE, and that's fine.

As mentioned in the previous two assignments, your (recursive) code to print an AST should be "defensive." If you come across an AST node with incorrect info, report that but do not stop immediately with a fatal error. This will be helpful for debugging.

On the course web site you'll find a few more (non-exhaustive) test cases and sample output.

ISSUES WHICH MAY COME UP

- **C23**: I recommend that you totally ignore the C23 standard for this assignment and instead use H&S or the C99 grammar. C23 made some very confusing changes in order to (among other things) make it legal to attach a label

to a declaration as well as a statement.

- **Lists**: You'll need a way of representing lists of things in the AST, e.g. lists of statements (compound statement). You may have already dealt with this in assignment #2 when processing function call argument lists. Consider stringing a linked list through the AST nodes, or a dynamically allocated (and re-allocable) array of AST node pointers. In the highly unlikely case that you are being schedule and need to save time, and thus choose to use a static array of AST node pointers, make sure the static limit is high enough for reasonable test cases.
- **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).
- **case labels**: A case statement is the keyword `case`, followed by an expression, a ':', and a statement. At this time, you can simply attach the AST of whatever that expression is. At code generation time, the expression must be a compile-time constant. E.g. `case 3+4:` is valid but `case 3+f(3):` or `case a+b:` is not! You'll have no way of determining that right now, so just accept whatever is syntactically valid.
- **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'll need 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 f
 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 Assignments #2 and #3 continue to be optional.

C99 and later support placing a declaration in the first clause of the `for` statement:

```
for(int i=0;i<10;i++)
          printf("%d\n",i);
```

In the author's opinion, this was a stupid idea that solves nothing. If it is truly necessary to hide the iterator within the for statement, the whole thing could be enclosed in a block. Furthermore, if you do a for loop this way, the iterator variable goes out of scope after the loop. While that might seem "proper," it is a common idiom in C programming to examine what the final value of a loop control variable was at the conclusion of the loop, when `break` may have been used inside the loop to terminate it early.

To implement this misguided feature correctly requires the invention of another type of scope which is not actually described in the C standard, essentially a block scope which begins at the point of the declaration and ends at the end of the overall for statement (including the loop body statement). This stealthy scope encloses any explicit block:

```
for(int i=0;i<10;i++)
{
          int i;
          printf("%d\n",i);
}
```

The compound statement (block) is inside the hidden block created by the inline declaration, therefore the second `int i` is not an error, and the value printed is not that of the loop iterator. KR1nG3!

The stealth scope starts when a `type_specifier` is seen, therefore `for(int i=i;i<10;i++)` does not give a compile-time error, although the behavior is undefined at run time since `i` is not initialized to a predictable value!

A few more words about initialized declarators

Initialized declarators in general are hard, especially supporting initializers for compound data types (arrays, structs) and initializers where the value is an arbitrary expression. That's why we made them optional in Assignment #3.

An initialized declaration of a variable of auto storage class behaves as if there were a declaration, followed by an assignment statement (or something like a memcpy, if a non-scalar is being declared). 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, classic 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. That's why my sample output included the inner LIST node.

When the variable is in global memory, the initializer must evaluate to a constant, which ultimately gets translated to an assembly language pseudo-opcode to create the `.data` initializer.

Array initializers of auto variables create a special case:

```
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.

Some further thoughts about variable length arrays

The C23 standard (6.8.1.3) clarifies that when a declaration contains an initializer or a variably modified array type, the hidden executable code associated with that runs as if the declaration were in fact a statement, each time control passes through that point.

```
for(i=1;i<10;i++)
{
          char a[j++];
          printf("so=%d j=%d\n",sizeof(a),j);
}
```

The result is that the reported size of the array a, as well as the value of j, goes up with each iteration of the loop. But what about this:

```
for(i=1;i<10;i++)
{
          int (*p)[j++];
          printf("j=%d\n",j);
}
```

Yep, the value of j goes up each time too!

If we were handling variable length arrays / variably modified types, the presence of this declaration would require us to insert it into the AST, as a "hidden statement," just as we did for initialized declarations of autos. Whew! Aren’t you glad we’re skipping that?