Part 16: Declaring Global Variables Properly

I did promise to look at the issue of adding offsets to pointers, but I need to do some thinking about that first. So I've decided to move global variable declarations out of function declarations. Actually, I've also left the parsing of variable declarations inside functions, because later on we will change them to be local variable declarations.

I also want to extend our grammar so that we can declare multiple variables with the same type at the same time, e.g.

```
int x, y, z;
```

The New BNF Grammar

Here is the new BNF grammar for global declarations, both functions and variables:

```
global_declarations : global_declarations
     | global_declaration global_declarations
     ;

global_declaration: function_declaration | var_declaration ;

function_declaration: type identifier '(' ')' compound_statement   ;

var_declaration: type identifier_list ';'  ;
```

```
type: type_keyword opt_pointer  ;

type_keyword: 'void' | 'char' | 'int' | 'long'  ;

opt_pointer: <empty> | '*' opt_pointer  ;

identifier_list: identifier | identifier ',' identifier_list ;
```

Both `function_declaration` and `global_declaration` start with a `type`. This is now a `type_keyword` followed by `opt_pointer` which is zero or more '*' tokens. After this, both `function_declaration` and `global_declaration` must be followed by one identifier.

However, after the `type`, `var_declaration` is followed by an `identifier_list`, which is one or more `identifier` s separated by a ',' token. Also `var_declaration` must end with a ';' token but `function_declaration` ends with a `compound_statement` and no ';' token.

New Tokens

We now have the T_COMMA token for the ',' character in `scan.c`.

Changes to `decl.c`

We now convert the above BNF grammar into a set of recursive descent functions but, as we can do looping, we can turn some of the recursion into internal loops.

`global_declarations()`

As there are one or more global declarations, we can loop parsing each one. When we run out of tokens, we can leave the loop.

```
// Parse one or more global declarations, either
// variables or functions
void global_declarations(void) {
  struct ASTnode *tree;
  int type;

  while (1) {

    // We have to read past the type and identifier
    // to see either a '(' for a function declaration
    // or a ',' or ';' for a variable declaration.
    // Text is filled in by the ident() call.
    type = parse_type();
```

```
    ident();
    if (Token.token == T_LPAREN) {

      // Parse the function declaration and
      // generate the assembly code for it
      tree = function_declaration(type);
      genAST(tree, NOREG, 0);
    } else {

      // Parse the global variable declaration
      var_declaration(type);
    }

    // Stop when we have reached EOF
    if (Token.token == T_EOF)
      break;
  }
}
```

Knowing that, for now we only have global variables and functions, we can scan in the type here and the first identifier. Then, we look at the next token. If it's a '(', we call `function_declaration()`. If not, we can assume that it is a `var_declaration()`. We pass the `type` in to both functions.

Now that we are receiving the AST `tree` from `function_declaration()` here, we can generate the code from the AST tree immediately. This code was in `main()` but has now been moved here. `main()` now only has to call `global_declarations()`:

```
  scan(&Token);                 // Get the first token from the input
  genpreamble();                // Output the preamble
  global_declarations();        // Parse the global declarations
  genpostamble();               // Output the postamble
```

`var_declaration()`

The parsing of functions is much the same as before, except the code to scan the type and identifier are done elsewhere, and we receive the `type` as an argument.

The parsing of variables also loses the type and identifier scanning code. We can add the identifier to the global symbol and generate the assembly code for it. But now we need to add in a loop. If there's a following ',', loop back to get the next identifier with the same type. And if there's a following ';', that's the end of the variable declarations.

```
// Parse the declaration of a list of variables.
// The identifier has been scanned & we have the type
void var_declaration(int type) {
  int id;

  while (1) {
    // Text now has the identifier's name.
    // Add it as a known identifier
    // and generate its space in assembly
    id = addglob(Text, type, S_VARIABLE, 0);
    genglobsym(id);

    // If the next token is a semicolon,
    // skip it and return.
    if (Token.token == T_SEMI) {
      scan(&Token);
      return;
    }
    // If the next token is a comma, skip it,
    // get the identifier and loop back
    if (Token.token == T_COMMA) {
      scan(&Token);
      ident();
      continue;
    }
    fatal("Missing , or ; after identifier");
  }
}
```

Not Quite Local Variables

`var_declaration()` can now parse a list of variable declarations, but it requires the type and first identifier to be pre-scanned.

Thus, I've left the call to `var_declaration()` in `single_statement()` in `stmt.c`. Later on, we will modify this to declare local variables. But for now, all of the variables in this example program are globals:

```
int   d, f;
int  *e;

int main() {
  int a, b, c;
  b= 3; c= 5; a= b + c * 10;
  printint(a);
```

```
  d= 12; printint(d);
  e= &d; f= *e; printint(f);
  return(0);
}
```

Testing the Changes

The above code is our `tests/input16.c` . As always, we can test it:

```
$ make test16
cc -o comp1 -g -Wall cg.c decl.c expr.c gen.c main.c misc.c scan.c
      stmt.c sym.c tree.c types.c
./comp1 tests/input16.c
cc -o out out.s lib/printint.c
./out
53
12
12
```

Conclusion and What's Next

In the next part of our compiler writing journey, I promise to tackle the issue of adding offsets to pointers. [Next step]