Part 41: Local Variable Initialisation

Well, after the significant list of changes in the last part, doing local variable initialisation was easy.

We want to be able to do this sort of thing inside functions:

```
int x= 2, y= x+3, z= 5 * x - y;
char *foo= "Hello world";
```

As we are inside a function, we can build an AST tree for the expression, build an A_IDENT node for the variable and join them together with an A_ASSIGN parent node. And, because there can be several declarations with assignments, we may need to build an A_GLUE tree which holds all of the assignment trees.

The only wrinkle is that the code which parses local declarations is quite a call distance away from the code that deals with statement parsing. In fact:

- `single_statement()` in `stmt.c` sees a type identifier and calls
- `declaration_list()` in `decl.c` to parse several declarations, which calls
- `symbol_declaration()` to parse one declaration, which calls
- `scalar_declaration()` to parse a scalar variable declaration and assignment

The main problem is that all of these functions already return a value, so we can't build an AST tree in `scalar_declaration()` and return it back to `single_statement()`.

Also, `declaration_list()` parses multiple declarations, so it will have the job of building the A_GLUE tree to hold them all together.

The solution is to pass down a "pointer pointer" from `single_statement()` to `declaration_list()`, so that we can pass back the pointer to the A_GLUE tree. Similarly, we will pass a "pointer pointer" from `declaration_list()` down to `scalar_declaration()`, which will pass back the pointer to any assignment tree that it has built.

Changes to `scalar_declaration()`

If we are in local context and we hit an '=' in a scalar variable's declaration, here is what we do:

```
struct ASTnode *varnode, *exprnode;
struct ASTnode **tree;                  // is the ptr ptr argument that we ge

// The variable is being initialised
if (Token.token == T_ASSIGN) {
  ...
  if (class == C_LOCAL) {
    // Make an A_IDENT AST node with the variable
    varnode = mkastleaf(A_IDENT, sym->type, sym, 0);

    // Get the expression for the assignment, make into a rvalue
    exprnode = binexpr(0);
    exprnode->rvalue = 1;

    // Ensure the expression's type matches the variable
    exprnode = modify_type(exprnode, varnode->type, 0);
    if (exprnode == NULL)
      fatal("Incompatible expression in assignment");

    // Make an assignment AST tree
    *tree = mkastnode(A_ASSIGN, exprnode->type, exprnode,
                                      NULL, varnode, NULL, 0);
  }
}
```

That's it. We simulate the AST tree building that would normally occur in `expr.c` for an assignment expression. Once done, we pass back the assignment tree. This bubbles back up to `declaration_list()`. It now does:

```
  struct ASTnode **gluetree;            // is the ptr ptr argument that we get
  struct ASTnode *tree;
  *gluetree= NULL;
  ...
  // Now parse the list of symbols
  while (1) {
    ...
    // Parse this symbol
    sym = symbol_declaration(type, *ctype, class, &tree);
    ...
    // Glue any AST tree from a local declaration
    // to build a sequence of assignments to perform
    if (*gluetree== NULL)
      *gluetree= tree;
    else
      *gluetree = mkastnode(A_GLUE, P_NONE, *gluetree, NULL, tree, NULL, 0);
    ...
  }
```

So `gluetree` is set to the AST tree with a bunch of A_GLUE nodes, each of which has an A_ASSIGN child with an A_IDENT child and an expression child.

And, way up in `single_statement()` in `stmt.c`:

```
  ...
  case T_IDENT:
    // We have to see if the identifier matches a typedef.
    // If not, treat it as an expression.
    // Otherwise, fall down to the parse_type() call.
    if (findtypedef(Text) == NULL) {
      stmt= binexpr(0); semi(); return(stmt);
    }
  case T_CHAR:
  case T_INT:
  case T_LONG:
  case T_STRUCT:
  case T_UNION:
  case T_ENUM:
  case T_TYPEDEF:
    // The beginning of a variable declaration list.
    declaration_list(&ctype, C_LOCAL, T_SEMI, T_EOF, &stmt);
    semi();
    return (stmt);            // Any assignments from the declarations
  ...
```

Testing the New Code

The above changes were so short and simple that they compiled and worked first time. This is not a regular occurrence! Our test program, `tests/input100.c` is this:

```
#include <stdio.h>
int main() {
  int x= 3, y=14;
  int z= 2 * x + y;
  char *str= "Hello world";
  printf("%s %d %d\n", str, x+y, z);
  return(0);
}
```

and produces the following correct output: `Hello world 17 20` .

Conclusion and What's Next

It's nice to have a simple part on this journey now and then. I'm now starting to take wagers with myself as to:

- how many parts to the journey, in total, there will be, and
- will I get it all done by the end of the year

Right now I'm guessing about 60 parts and an 75% chance of completing by year's end. But we still have a bunch of small, but possibly difficult, features to add to the compiler.

In the next part of our compiler writing journey, I will add cast parsing to the compiler.
Next step