Part 11: Functions, part 1

I want to start work on implementing functions into our language, but I know this is going to involve a heck of a lot of steps. Some things that we will have to deal with along the way are:

- Types of data: `char`, `int`, `long` etc.
- The return type of each function
- The number of arguments to each function
- Variables local to a function versus global variables

That is way too much to get done in this part of our journey. So what I'm going to do here is to get to the point where we can *declare* different functions. Only the `main()` function in our resulting executable will run, but we will have the ability to generate code for multiple function.

Hopefully soon, the language that our compiler recognises will be enough of a subset of C that our input will be recognisable by a "real" C compiler. But just not yet.

The Simplistic Function Syntax

This is definitely going to be a placeholder, so that we can parse something that looks like a function. Once this is done, we can add those other important things: types, return types, arguments etc.

So, for now, I will add a function grammar that looks like this in BNF:

```
function_declaration: 'void' identifier '(' ')' compound_statement   ;
```

All functions will be declared `void` and have no arguments. We also won't introduce the ability to call a function, so only the `main()` function will execute.

We need a new keyword `void` and a new token T_VOID, which are both easy to add.

Parsing the Simplistic Function Syntax

The new function syntax is so simple that we can write a nice, small function to parse it (in `decl.c`):

```c
// Parse the declaration of a simplistic function
struct ASTnode *function_declaration(void) {
  struct ASTnode *tree;
  int nameslot;

  // Find the 'void', the identifier, and the '(' ')'.
  // For now, do nothing with them
  match(T_VOID, "void");
  ident();
  nameslot= addglob(Text);
  lparen();
  rparen();

  // Get the AST tree for the compound statement
  tree= compound_statement();

  // Return an A_FUNCTION node which has the function's nameslot
  // and the compound statement sub-tree
  return(mkastunary(A_FUNCTION, tree, nameslot));
}
```

This is going to do the syntax checking and AST building, but there is little to no semantic error checking here. What if a function gets redeclared? Well, we won't notice that yet.

Modifications to `main()`

With the above function, we can now rewrite some of the code in `main()` to parse multiple functions one after the other:

```
  scan(&Token);                 // Get the first token from the input
  genpreamble();                // Output the preamble
  while (1) {                   // Parse a function and
    tree = function_declaration();
    genAST(tree, NOREG, 0);     // generate the assembly code for it
    if (Token.token == T_EOF)   // Stop when we have reached EOF
      break;
  }
```

Notice that I've removed the `genpostamble()` function call. That's because its output was technically the postamble to the generated assembly for `main()`. We now need some code generation functions to generate the beginning of a function and the end of a function.

Generic Code Generation for Functions

Now that we have an A_FUNCTION AST node, we had better add some code in the generic code generator, `gen.c` to deal with it. Looking above, this is a *unary* AST node with a single child:

```
  // Return an A_FUNCTION node which has the function's nameslot
  // and the compound statement sub-tree
  return(mkastunary(A_FUNCTION, tree, nameslot));
```

The child has the sub-tree which holds the compound statement that is the body of the function. We need to generate the start of the function *before* we generate the code for the compound statement. So here's the code in `genAST()` to do this:

```
    case A_FUNCTION:
      // Generate the function's preamble before the code
      cgfuncpreamble(Gsym[n->v.id].name);
      genAST(n->left, NOREG, n->op);
      cgfuncpostamble();
      return (NOREG);
```

x86-64 Code Generation

Now we are at the point where we have to generate the code to set the stack and frame pointer for each function, and also to undo this at the end of the function and return to the function's caller.

We already have this code in `cgpreamble()` and `cgpostamble()`, but `cgpreamble()` also has the assembly code for the `printint()` function. Therefore, it's a matter of separating out these snippets of assembly code into new functions in `cg.c`:

```
// Print out the assembly preamble
void cgpreamble() {
  freeall_registers();
  // Only prints out the code for printint()
}

// Print out a function preamble
void cgfuncpreamble(char *name) {
  fprintf(Outfile,
          "\t.text\n"
          "\t.globl\t%s\n"
          "\t.type\t%s, @function\n"
```

```
// Print out a function postamble
void cgfuncpostamble() {
  fputs("\tmovl $0, %eax\n" "\tpopq     %rbp\n" "\tret\n", Outfile);
}
```

Testing The Function Generation Functionality

We have a new test program, `tests/input08` which is starting to look like a C program (apart from the `print` statement):

```
void main()
{
  int i;
  for (i= 1; i <= 10; i= i + 1) {
    print i;
  }
}
```

To test this, `make test8` which does:

```
cc -o comp1 -g cg.c decl.c expr.c gen.c main.c misc.c scan.c
    stmt.c sym.c tree.c
./comp1 tests/input08
cc -o out out.s
```

```
./out
1
2
3
4
5
6
7
8
9
10
```

I'm not going to look at the assembly output as it's identical to the code generated for the FOR loop test in the last part.

However, I've added `void main()` into all the previous test input files, as the language requires a function declaration before the compound statement code.

The test program `tests/input09` has two functions declared in it. The compiler happily generates working assembly code for each function, but at present we can't run the code for the second function.

Conclusion and What's Next

We've made a good start at adding functions to our language. For now, it's a pretty simplistic function declaration only.

In the next part of our compiler writing journey, we will begin the process to add types to our compiler. Next step