Part 26: Function Prototypes

In this part of our compiler writing journey, I've added the ability to write function prototypes. In the process, I've had to rewrite some of the code that I'd just written in the previous parts; sorry about that. I didn't see far enough ahead!

So what do we want with function prototypes:

- the ability to declare a function prototype with no body
- the ability to declare a full function later on
- to keep the prototype in the global symbol table section, and the parameters as local variables in the local symbol table section
- error checking on the number and types of parameters against a previous function prototype

And here is what I'm *not* going to do, at least not yet:

- `function(void)` : this will be the same as the `function()` declaration
- the declaration of a function with just the types, e.g. `function(int ,char, long);` as this will make the parsing harder. We can do this later.

What Functionality Needs to be Rewritten

In a recent part of our journey I added the declaration of a function with parameters and a full function body. As we were parsing each parameter, I immediately added it to both the global symbol table (to form the prototype) and also the local symbol table (to be the function's parameters).

Now that we want to implement function prototypes, it's not always true that a parameter list will become the actual function's parameters. Consider this function prototype:

```
int fred(char a, int foo, long bar);
```

We can only define `fred` as a function, and `a`, `foo` and `bar` as three parameters in the global symbol table. We have to wait until the full function declaration before we can add `a`, `foo` and `bar` to the local symbol table.

I'll need to separate the definition of C_PARAM entries on the global symbol table and on the local symbol table.

The Design of the New Parsing Mechanism

Here is my quick design for the new function parsing mechanism which also deals with prototypes.

```
Get the identifier and '('.
Search for the identifier in the symbol table.
If it exists, there is already a prototype: get the id position of
the function and its parammeter count.

While parsing parameters:
  - if a previous prototype, compare this param's type against the existing
    one. Update the symbol's name in case this is a full function
  - if no previous prototype, add the parameter to the symbol table

Ensure # of params matches any existing prototype.
Parse the ')'. If ';' is next, done.

If '{' is next, copy the parameter list from the global symtable to the
local sym table. Copy them in a loop so that they are put in reverse order
in the local sym table.
```

I got this done in the last few hours, so here are the code changes.

Changes to `sym.c`

I've changed the parameter list of a couple of functions in `sym.c`:

```
int addglob(char *name, int type, int stype, int class, int endlabel, int size
int addlocl(char *name, int type, int stype, int class, int size);
```

Previously, we had `addlocl()` also call `addglob()` to add a C_PARAM symbol to both symbol tables. Now that we are separating this function, it makes sense to pass the actual class of the symbol to both functions.

There are calls to these functions in `main.c` and `decl.c`. I'll cover the ones in `decl.c` later. The change in `main.c` is trivial.

Once we hit the declaration of a real function, we will need to copy its parameter list from the global to the local symbol table. As this is really something specific to the symbol table, I've added this function to `sym.c`:

```
// Given a function's slot number, copy the global parameters
// from its prototype to be local parameters
void copyfuncparams(int slot) {
  int i, id = slot + 1;

  for (i = 0; i < Symtable[slot].nelems; i++, id++) {
    addlocl(Symtable[id].name, Symtable[id].type, Symtable[id].stype,
            Symtable[id].class, Symtable[id].size);
  }
}
```

Changes to `decl.c`

Nearly all of the changes to the compiler are confined to `decl.c`. We'll start with the small ones and work up to the big ones.

`var_declaration()`

I've changed the parameter list to `var_declaration()` in the same way that I did for the `sym.c` functions:

```
void var_declaration(int type, int class) {
  ...
  addglob(Text, pointer_to(type), S_ARRAY, class, 0, Token.intvalue);
  ...
  if (addlocl(Text, type, S_VARIABLE, class, 1) == -1)
  ...
  addglob(Text, type, S_VARIABLE, class, 0, 1);
}
```

We will use the ability to pass in the class in the other `decl.c` functions.

`param_declaration()`

We have big changes here, as we might already have a parameter list in the global symbol table as an existing prototype. If we do, we need to check the number and types in the new list against the prototype.

```
// Parse the parameters in parentheses after the function name.
// Add them as symbols to the symbol table and return the number
// of parameters. If id is not -1, there is an existing function
// prototype, and the function has this symbol slot number.
static int param_declaration(int id) {
  int type, param_id;
  int orig_paramcnt;
  int paramcnt = 0;

  // Add 1 to id so that it's either zero (no prototype), or
  // it's the position of the zeroth existing parameter in
  // the symbol table
  param_id = id + 1;

  // Get any existing prototype parameter count
  if (param_id)
    orig_paramcnt = Symtable[id].nelems;

  // Loop until the final right parentheses
  while (Token.token != T_RPAREN) {
    // Get the type and identifier
    // and add it to the symbol table
    type = parse_type();
    ident();

    // We have an existing prototype.
    // Check that this type matches the prototype.
    if (param_id) {
      if (type != Symtable[id].type)
```

```
          fatald("Type doesn't match prototype for parameter", paramcnt + 1);
        param_id++;
      } else {
        // Add a new parameter to the new prototype
        var_declaration(type, C_PARAM);
      }
      paramcnt++;

      // Must have a ',' or ')' at this point
      switch (Token.token) {
      case T_COMMA:
        scan(&Token);
        break;
      case T_RPAREN:
        break;
      default:
        fatald("Unexpected token in parameter list", Token.token);
      }
    }

    // Check that the number of parameters in this list matches
    // any existing prototype
    if ((id != -1) && (paramcnt != orig_paramcnt))
      fatals("Parameter count mismatch for function", Symtable[id].name);

    // Return the count of parameters
    return (paramcnt);
  }
```

Remember that the first parameter's global symbol table slot position is immediately after the slot for the function name's symbol. We get passed the slot position of an existing prototype, or -1 if there is no prototype.

It's a happy coincidence that we can add one to this to get the first parameter's slot number, or have 0 to indicate that there is no existing prototype.

We still loop parsing each new parameter, but now there is new code to either compare against the existing prototype, or to add the parameter to the global symbol table.

Once we exit the loop, we can compare the number of parameters in this list against the number in any existing prototype.

Right now, the code feels a bit ugly and I'm sure that if I leave it a while, I'll be able to see a way to refactor it a bit.

`function_declaration()`

Previously, this was a fairly simple function: get the type and name, add a global symbol, read in the parameters, get the function's body and generate an AST tree for the function's code.

Now, we have to deal with the fact this this might only be a prototype, or it could be a full function. And we won't know until we parse either the ';' (for a prototype) or the '{' (for a full function). So let's take the exposition of the code in stages.

```
// Parse the declaration of function.
// The identifier has been scanned & we have the type.
struct ASTnode *function_declaration(int type) {
  struct ASTnode *tree, *finalstmt;
  int id;
  int nameslot, endlabel, paramcnt;

  // Text has the identifier's name. If this exists and is a
  // function, get the id. Otherwise, set id to -1
  if ((id = findsymbol(Text)) != -1)
    if (Symtable[id].stype != S_FUNCTION)
      id = -1;

  // If this is a new function declaration, get a
  // label-id for the end label, and add the function
  // to the symbol table,
  if (id == -1) {
    endlabel = genlabel();
    nameslot = addglob(Text, type, S_FUNCTION, C_GLOBAL, endlabel, 0);
  }
  // Scan in the '(', any parameters and the ')'.
  // Pass in any existing function prototype symbol slot number
  lparen();
  paramcnt = param_declaration(id);
  rparen();
```

This is nearly the same as the previous version of the code, except that `id` is now set to -1 when there is no previous prototype or a positive number when there is a previous prototype. We only add the function's name to the global symbol table if it's not already there.

```
  // If this is a new function declaration, update the
  // function symbol entry with the number of parameters
  if (id == -1)
    Symtable[nameslot].nelems = paramcnt;

  // Declaration ends in a semicolon, only a prototype.
```

```
  if (Token.token == T_SEMI) {
    scan(&Token);
    return (NULL);
  }
```

We've got the count of parameters. If no previous prototype, update this prototype with this count. Now we can peek at the token after the end of the parameter list. If it's a semicolon, this is just a prototype. We now have no AST tree to return, so skip the token and return NULL. I've had to slightly alter the code in `global_declarations()` to deal with this NULL value: no big change.

If we continue on, we are now dealing with a full function declaration with a body.

```
  // This is not just a prototype.
  // Copy the global parameters to be local parameters
  if (id == -1)
    id = nameslot;
  copyfuncparams(id);
```

We now need to copy the parameters from the prototype to the local symbol table. The `id = nameslot` code is there for when we have just added the global symbols ourselves and there was no previous prototype.

The rest of the code in `function_declaration()` is the same as before and I'll omit it. It checks that a non-void function does return a value, and generates the AST tree with an A_FUNCTION root node.

Testing the New Functionality

One of the drawbacks of the `tests/runtests` script is that it assumes the compiler will definitely produce an assembly output file `out.s` which can be assembled and run. This prevents us from testing that the compiler detects syntax and semantic errors.

A quick *grep* of `decl.c` shows these new errors are detected:

```
  fatald("Type doesn't match prototype for parameter", paramcnt + 1);
  fatals("Parameter count mismatch for function", Symtable[id].name);
```

Thus, I'd better rewrite `tests/runtests` to verify that the compiler does detect these errors on bad input.

We do have two new working test programs, `input29.c` and `input30.c`. The first one is the same as `input28.c` except that I've put the prototypes of all the functions at the top of the program:

```
int param8(int a, int b, int c, int d, int e, int f, int g, int h);
int fred(int a, int b, int c);
int main();
```

This, and all previous test programs, still work. `input30.c`, though, is probably the first non-trivial program that our compiler has been given. It opens its own source file and prints it to standard output:

```
int open(char *pathname, int flags);
int read(int fd, char *buf, int count);
int write(int fd, void *buf, int count);
int close(int fd);

char *buf;

int main() {
  int zin;
  int cnt;

  buf= "                                                                ";
  zin = open("input30.c", 0);
  if (zin == -1) {
    return (1);
  }
  while ((cnt = read(zin, buf, 60)) > 0) {
    write(1, buf, cnt);
  }
  close(zin);
  return (0);
}
```

We can't yet call the pre-processor, so we manually put in the prototypes for the `open()`, `read()`, `write()` and `close()` functions. We also have to use 0 instead of O_RDONLY in the `open()` call.

Right now, the compiler lets us declare a `char buf[60];` but we can't use `buf` itself as a char pointer. So I chose to assign a 60-character literal string to a char pointer and we use this as the buffer.

We still also have to wrap IF and WHILE bodies with '{' ... '}' to make them compound statements: I still haven't dealt with the dangling else problem. Finally, we can't accept `char *argv[]` as a parameter declaration for main yet, so I've had to hard-code the input file's name.

Still, we now have a very primitive *cat(1)* program which our compiler can compile! That's progress.

Conclusion and What's Next

In the next part of our compiler writing journey, I'll follow up on a comment above and improve the testing of our compiler's functionality. Next step