Part 13: Functions, part 2

In this part of our compiler writing journey, I want to add the ability to call functions and return a value. Specifically:

- define a function, which we already have,
- call a function with a single value which for now cannot be used,
- return a value from a function,
- use a function call as both a statement and also an expression, and
- ensure that void functions never return a value and non-void functions must return a value.

I've just got this working. I found that I spent most of my time dealing with types. So, on with the writeup.

New Keywords and Tokens

I've been using 8-byte (64-bit) `int` s in the compiler so far, but I've realised that Gcc treats `int` s as four bytes (32 bits) wide. Therefore, I've decided to introduce the `long` type. So now:

- `char` is one byte wide
- `int` is four bytes (32 bits) wide
- `long` is eight bytes (64 bits) wide

We also need the ability to 'return', so we have new keywords 'long' and 'return', and associated tokens T_LONG and T_RETURN.

Parsing Function Calls

For now, the BNF syntax that I'm using for a function call is:

```
function_call: identifier '(' expression ')'   ;
```

The function has a name followed by a pair of parentheses. Inside the parentheses we must have exactly one argument. I want this to be used as both an expression and also as a standalone statement.

So we'll start with the function call parser, `funccall()` in `expr.c`. When we get called, the identifier has already been scanned in and the function's name is in the `Text` global variable:

```
// Parse a function call with a single expression
// argument and return its AST
struct ASTnode *funccall(void) {
  struct ASTnode *tree;
  int id;

  // Check that the identifier has been defined,
  // then make a leaf node for it. XXX Add structural type test
  if ((id = findglob(Text)) == -1) {
    fatals("Undeclared function", Text);
  }
  // Get the '('
  lparen();

  // Parse the following expression
  tree = binexpr(0);

  // Build the function call AST node. Store the
  // function's return type as this node's type.
  // Also record the function's symbol-id
  tree = mkastunary(A_FUNCCALL, Gsym[id].type, tree, id);

  // Get the ')'
  rparen();
  return (tree);
}
```

I've left a reminder comment: *Add structural type test*. When a function or a variable is declared, the symbol table is marked with the structural type S_FUNCTION and S_VARIABLE, respectively. I should add code here to confirm the identifier is really an S_FUNCTION.

We build a new unary AST node, A_FUNCCALL. The child is the single expression to pass as the argument. We store the function's symbol-id in the node, and we also record the function's return type.

But I Don't Want That Token Any More!

There is a parsing problem. We have to distinguish between:

```
x= fred + jim;
x= fred(5) + jim;
```

We need to look ahead one token to see if there is a '('. If there is, we have a function call. But by doing so, we lose the existing token. To solve this problem, I've modified the scanner so that we can put back an unwanted token: this will be returned when we get the next token instead of a brand-new token. The new code in `scan.c` is:

```
// A pointer to a rejected token
static struct token *Rejtoken = NULL;

// Reject the token that we just scanned
void reject_token(struct token *t) {
  if (Rejtoken != NULL)
    fatal("Can't reject token twice");
  Rejtoken = t;
}

// Scan and return the next token found in the input.
// Return 1 if token valid, 0 if no tokens left.
int scan(struct token *t) {
  int c, tokentype;

  // If we have any rejected token, return it
  if (Rejtoken != NULL) {
    t = Rejtoken;
    Rejtoken = NULL;
    return (1);
  }

  // Continue on with the normal scanning
```

```
  ...
}
```

Calling a Function as an Expression

So now we can look at where, in `expr.c` we need to differentiate between a variable name and a function call: it's in `primary()` . The new code is:

```
// Parse a primary factor and return an
// AST node representing it.
static struct ASTnode *primary(void) {
  struct ASTnode *n;
  int id;

  switch (Token.token) {
    ...
    case T_IDENT:
      // This could be a variable or a function call.
      // Scan in the next token to find out
      scan(&Token);

      // It's a '(', so a function call
      if (Token.token == T_LPAREN)
        return (funccall());

      // Not a function call, so reject the new token
      reject_token(&Token);

      // Continue on with normal variable parsing
      ...
}
```

Calling a Function as a Statement

We have essentially the same problem when we try to call a function as a statement. Here, we have to distinguish between:

```
  fred = 2;
  fred(18);
```

Thus, the new statement code in `stmt.c` is similar to the above:

```
// Parse an assignment statement and return its AST
static struct ASTnode *assignment_statement(void) {
  struct ASTnode *left, *right, *tree;
  int lefttype, righttype;
  int id;

  // Ensure we have an identifier
  ident();

  // This could be a variable or a function call.
  // If next token is '(', it's a function call
  if (Token.token == T_LPAREN)
    return (funccall());

  // Not a function call, on with an assignment then!
  ...
}
```

We can get away with not rejecting the "unwanted" token here, because there *has* to be either an '=' or a '(' next: we can write the parser code knowing this is true.

Parsing a Return Statement

In BNF, our return statement looks like:

```
  return_statement: 'return' '(' expression ')'  ;
```

The parsing is easy: 'return', '(', call `binexpr()`, ')', done! What is more difficult is the checking of the type, and if we even should be allowed to return at all.

Somehow we need to know which function we are actually in, when we get to a return statement. I've added a global variable in `data.h`:

```
extern_ int Functionid;         // Symbol id of the current function
```

and this is set up in `function_declaration()` in `decl.c`:

```
struct ASTnode *function_declaration(void) {
  ...
  // Add the function to the symbol table
  // and set the Functionid global
  nameslot = addglob(Text, type, S_FUNCTION, endlabel);
```

```
  Functionid = nameslot;
  ...
}
```

With `Functionid` set up each time we enter a function declaration, we can get back to parsing and checking the semantics of a return statement. The new code is `return_statement()` in `stmt.c`:

```
// Parse a return statement and return its AST
static struct ASTnode *return_statement(void) {
  struct ASTnode *tree;
  int returntype, functype;

  // Can't return a value if function returns P_VOID
  if (Gsym[Functionid].type == P_VOID)
    fatal("Can't return from a void function");

  // Ensure we have 'return' '('
  match(T_RETURN, "return");
  lparen();

  // Parse the following expression
  tree = binexpr(0);

  // Ensure this is compatible with the function's type
  returntype = tree->type;
  functype = Gsym[Functionid].type;
  if (!type_compatible(&returntype, &functype, 1))
    fatal("Incompatible types");

  // Widen the left if required.
  if (returntype)
    tree = mkastunary(returntype, functype, tree, 0);

  // Add on the A_RETURN node
  tree = mkastunary(A_RETURN, P_NONE, tree, 0);

  // Get the ')'
  rparen();
  return (tree);
}
```

We have a new A_RETURN AST node that returns the expression in the child tree. We use `type_compatible()` to ensure the expression matches the return type, and widen it if required.

Finally, we see if the function was actually declared `void`. If it was, we cannot do a return statement in this function.

Types Revisited

I introduced `type_compatible()` in the last part of the journey and said that I wanted to refactor it. Now that I've added the `long` type, it's become necessary to do this. So here is the new version in `types.c`. You may want to revisit the commentary on it from the last part of the journey.

```
// Given two primitive types,
// return true if they are compatible,
// false otherwise. Also return either
// zero or an A_WIDEN operation if one
// has to be widened to match the other.
// If onlyright is true, only widen left to right.
int type_compatible(int *left, int *right, int onlyright) {
  int leftsize, rightsize;

  // Same types, they are compatible
  if (*left == *right) { *left = *right = 0; return (1); }
  // Get the sizes for each type
  leftsize = genprimsize(*left);
  rightsize = genprimsize(*right);

  // Types with zero size are not
  // not compatible with anything
  if ((leftsize == 0) || (rightsize == 0)) return (0);

  // Widen types as required
  if (leftsize < rightsize) { *left = A_WIDEN; *right = 0; return (1);
  }
  if (rightsize < leftsize) {
    if (onlyright) return (0);
    *left = 0; *right = A_WIDEN; return (1);
  }
  // Anything remaining is the same size
  // and thus compatible
  *left = *right = 0;
  return (1);
}
```

I now call `genprimsize()` in the generic code generator which calls `cgprimsize()` in `cg.c` to get the size of the various types:

```
// Array of type sizes in P_XXX order.
// 0 means no size. P_NONE, P_VOID, P_CHAR, P_INT, P_LONG
static int psize[] = { 0,       0,      1,     4,     8 };

// Given a P_XXX type value, return the
// size of a primitive type in bytes.
int cgprimsize(int type) {
  // Check the type is valid
  if (type < P_NONE || type > P_LONG)
    fatal("Bad type in cgprimsize()");
  return (psize[type]);
}
```

This makes the type sizes platform dependent; other platforms can choose different type sizes. It probably means my code to mark a P_INTLIT as a `char` not an `int` will need to be refactored:

```
  if ((Token.intvalue) >= 0 && (Token.intvalue < 256))
```

Ensuring Non-Void Functions Return a Value

We've just ensured that void functions can't return a value. Now how to we ensure that non-void functions will always return a value? To do this, we have to ensure that the last statement in the function is a return statement.

Down at the bottom of `function_declaration()` in `decl.c`, I now have:

```
  struct ASTnode *tree, *finalstmt;
  ...
  // If the function type isn't P_VOID, check that
  // the last AST operation in the compound statement
  // was a return statement
  if (type != P_VOID) {
    finalstmt = (tree->op == A_GLUE) ? tree->right : tree;
    if (finalstmt == NULL || finalstmt->op != A_RETURN)
      fatal("No return for function with non-void type");
  }
```

The wrinkle is that, if the function has exactly one statement, there is no A_GLUE AST node and there is only a left child in the tree which is the compound statement.

At this point, we can:

- declare a function, store its type, and record we are in that function
- make a function call (either as an expression or a statement) with a single argument
- return from a non-void function (only), and force that the last statement in a non-void function is a return statement
- check and widen the expression being returned so as to match the function's type definition

Our AST tree now has A_RETURN and A_FUNCCAL nodes to with the return statements and function calls. Let's now see how they generate the assembly output.

Why a Single Argument?

You might, at this point, be asking: why do you want to have a single function argument, especially as that argument isn't available to the function?

The answer is that I want to replace the `print x;` statement in our language with a real function call: `printint(x);` . To do this, we can compile a real C function `printint()` and link it with the output from our compiler.

The New AST Nodes

There is not much new code in `genAST()` in `gen.c` :

```
case A_RETURN:
  cgreturn(leftreg, Functionid);
  return (NOREG);
case A_FUNCCALL:
  return (cgcall(leftreg, n->v.id));
```

A_RETURN doesn't return a value as it's not an expression. A_FUNCCALL is an expression of course.

Changes in the x86-64 Output

All the new code generation work is in the platform-specific code generator, `cg.c` . Let's have a look at this.

New Types

Firstly, we now have `char`, `int` and `long`, and the x86-64 requires us to use the right register names for each type:

```
// List of available registers and their names.
static int freereg[4];
static char *reglist[4] = { "%r8", "%r9", "%r10", "%r11" };
static char *breglist[4] = { "%r8b", "%r9b", "%r10b", "%r11b" };
static char *dreglist[4] = { "%r8d", "%r9d", "%r10d", "%r11d" }
```

Defining, Loading and Storing Variables

Variables now have three possible type. The code we generate needs to reflect this. Here are the changed functions:

```
// Generate a global symbol
void cgglobsym(int id) {
  int typesize;
  // Get the size of the type
  typesize = cgprimsize(Gsym[id].type);

  fprintf(Outfile, "\t.comm\t%s,%d,%d\n", Gsym[id].name, typesize, typesize);
}

// Load a value from a variable into a register.
// Return the number of the register
int cgloadglob(int id) {
  // Get a new register
  int r = alloc_register();

  // Print out the code to initialise it
  switch (Gsym[id].type) {
    case P_CHAR:
      fprintf(Outfile, "\tmovzbq\t%s(\%%rip), %s\n", Gsym[id].name,
              reglist[r]);
      break;
    case P_INT:
      fprintf(Outfile, "\tmovzbl\t%s(\%%rip), %s\n", Gsym[id].name,
              reglist[r]);
      break;
    case P_LONG:
      fprintf(Outfile, "\tmovq\t%s(\%%rip), %s\n", Gsym[id].name, reglist[r]);
      break;
    default:
      fatald("Bad type in cgloadglob:", Gsym[id].type);
  }
  return (r);
```

```
}

// Store a register's value into a variable
int cgstorglob(int r, int id) {
  switch (Gsym[id].type) {
    case P_CHAR:
      fprintf(Outfile, "\tmovb\t%s, %s(\%%rip)\n", breglist[r],
              Gsym[id].name);
      break;
    case P_INT:
      fprintf(Outfile, "\tmovl\t%s, %s(\%%rip)\n", dreglist[r],
              Gsym[id].name);
      break;
    case P_LONG:
      fprintf(Outfile, "\tmovq\t%s, %s(\%%rip)\n", reglist[r], Gsym[id].name);
      break;
    default:
      fatald("Bad type in cgloadglob:", Gsym[id].type);
  }
  return (r);
}
```

Function Calls

To call a function with one argument, we need to copy the register with the argument value into `%rdi`. On return, we need to copy the returned value from `%rax` into the register that will have this new value:

```
// Call a function with one argument from the given register
// Return the register with the result
int cgcall(int r, int id) {
  // Get a new register
  int outr = alloc_register();
  fprintf(Outfile, "\tmovq\t%s, %%rdi\n", reglist[r]);
  fprintf(Outfile, "\tcall\t%s\n", Gsym[id].name);
  fprintf(Outfile, "\tmovq\t%%rax, %s\n", reglist[outr]);
  free_register(r);
  return (outr);
}
```

Function Returns

To return from a function from any point in the function's execution, we need to jump to a label right at the bottom of the function. I've added code in `function_declaration()` to make a label and store it in the symbol table. As the return value leaves in the `%rax` register, we need to copy into this register before we jump to the end label:

```
// Generate code to return a value from a function
void cgreturn(int reg, int id) {
  // Generate code depending on the function's type
  switch (Gsym[id].type) {
    case P_CHAR:
      fprintf(Outfile, "\tmovzbl\t%s, %%eax\n", breglist[reg]);
      break;
    case P_INT:
      fprintf(Outfile, "\tmovl\t%s, %%eax\n", dreglist[reg]);
      break;
    case P_LONG:
      fprintf(Outfile, "\tmovq\t%s, %%rax\n", reglist[reg]);
      break;
    default:
      fatald("Bad function type in cgreturn:", Gsym[id].type);
  }
  cgjump(Gsym[id].endlabel);
}
```

Changes to the Function Preamble and Postamble

There are no changes to the preamble, but previously we were setting `%rax` to zero on the return. We have to remove this bit of code:

```
// Print out a function postamble
void cgfuncpostamble(int id) {
  cglabel(Gsym[id].endlabel);
  fputs("\tpopq %rbp\n" "\tret\n", Outfile);
}
```

Changes to the Initial Preamble

Up to now, I've been manually inserting an assembly version of `printint()` at the beginning of our assembly output. We no longer need this, as we can compile a real C function `printint()` and link it with the output from our compiler.

Testing the Changes

There is a new test program, `tests/input14`:

```
int fred() {
  return(20);
}

void main() {
  int result;
  printint(10);
  result= fred(15);
  printint(result);
  printint(fred(15)+10);
  return(0);
}
```

We firstly print 10, then call `fred()` which returns 20 and print this out. Finally, we call `fred()` again, add its return value to 10 and print out 30. This demonstrates function calls with a single value, and function returns. Here is the test results:

```
cc -o comp1 -g cg.c decl.c expr.c gen.c main.c misc.c scan.c
    stmt.c sym.c tree.c types.c
./comp1 tests/input14
cc -o out out.s lib/printint.c
./out; true
10
20
30
```

Note that we link our assembly output with `lib/printint.c`:

```
#include <stdio.h>
void printint(long x) {
  printf("%ld\n", x);
}
```

So Nearly C Now

With this change, we can do this:

```
$ cat lib/printint.c tests/input14 > input14.c
$ cc -o out input14.c
$ ./out
10
20
30
```

In other words, our language is enough of a subset of C that we can compile it with other C functions to get an executable. Excellent!

Conclusion and What's Next

We've just added a simple version of function calls, function returns plus a new data type. As I expected, it wasn't trivial but I think the changes are mostly sensible.

In the next part of our compiler writing journey, we will port our compiler to a new hardware platform, the ARM CPU on a Raspberry Pi. Next step