Part 53: Mopping Up, part 2

In this part of our compiler writing journey, I fix a few annoying things that we use in the compiler's own source code.

Consecutive String Literals

C allows the declaration of string literals by splitting them across multiple lines or as multiple strings, e.g.

```
char *c= "hello " "there, "
         "how " "are " "you?";
```

Now, we could fix this problem up in the lexical scanner. However, I spent a lot of time trying to do this. The problem is that the code is now complicated with dealing with the C pre-processor, and I couldn't find a clean way of allowing consecutive string literals.

My solution is to do it in the parser, with a bit of help from the code generator. In `primary()` in `expr.c`, the code that deals with string literals now looks like this:

```
case T_STRLIT:
  // For a STRLIT token, generate the assembly for it.
  id = genglobstr(Text, 0);   // 0 means generate a label

  // For successive STRLIT tokens, append their contents
```

```
    // to this one
    while (1) {
      scan(&Peektoken);
      if (Peektoken.token != T_STRLIT) break;
      genglobstr(Text, 1);      // 1 means don't generate a label
      scan(&Token);             // Skip it
    }

    // Now make a leaf AST node for it. id is the string's label.
    genglobstrend();
    n = mkastleaf(A_STRLIT, pointer_to(P_CHAR), NULL, NULL, id);
    break;
```

`genglobstr()` now takes a second argument which tells it if this is the first part of the string or a successive part of the string. Also, `genglobstrend()` now has the job of NUL terminating the string literal.

Empty Statements

C allows both empty statements and empty compound statements, e.g.

```
  while ((c=getc()) != 'x') ;           // ';' is an empty statement

  int fred() { }                        // Function with empty body
```

and I use both of these in the compiler, so we need to support both of them. In `stmt.c`, the code now does this:

```
static struct ASTnode *single_statement(void) {
  struct ASTnode *stmt;
  struct symtable *ctype;

  switch (Token.token) {
    case T_SEMI:
      // An empty statement
      semi();
      break;
    ...
  }
  ...
}

struct ASTnode *compound_statement(int inswitch) {
  struct ASTnode *left = NULL;
  struct ASTnode *tree;
```

```
  while (1) {
    // Leave if we've hit the end token. We do this first to allow
    // an empty compound statement
    if (Token.token == T_RBRACE)
      return (left);
    ...
  }
  ...
}
```

and that fixes both shortcomings.

Redeclared Symbols

C allows a global variable to later be declared extern, and an extern variable to be declared later as a global variable, and vice versa. However, the types of both declarations have to match. We also want to ensure that only one version of the symbol ends up in the symbol table: we don't want both a C_GLOBAL and a C_EXTERN entry!

In `stmt.c` I've added a new function called `is_new_symbol()`. We call this after we have parsed the name of a variable and after we have tried to find it in the symbol table. Thus, `sym` may be NULL (no existing variable) or not NULL (is an existing variable).

If the symbol exists, it's actually quite complicated to ensure that it's a safe redeclaration.

```
// Given a pointer to a symbol that may already exist
// return true if this symbol doesn't exist. We use
// this function to convert externs into globals
int is_new_symbol(struct symtable *sym, int class,
                  int type, struct symtable *ctype) {

  // There is no existing symbol, thus is new
  if (sym==NULL) return(1);

  // global versus extern: if they match that it's not new
  // and we can convert the class to global
  if ((sym->class== C_GLOBAL && class== C_EXTERN)
      || (sym->class== C_EXTERN && class== C_GLOBAL)) {

      // If the types don't match, there's a problem
      if (type != sym->type)
        fatals("Type mismatch between global/extern", sym->name);

      // Struct/unions, also compare the ctype
```

```
        if (type >= P_STRUCT && ctype != sym->ctype)
          fatals("Type mismatch between global/extern", sym->name);

        // If we get to here, the types match, so mark the symbol
        // as global
        sym->class= C_GLOBAL;
        // Return that symbol is not new
        return(0);
    }

    // It must be a duplicate symbol if we get here
    fatals("Duplicate global variable declaration", sym->name);
    return(-1);   // Keep -Wall happy
  }
```

The code is straight-forward but not elegant. Also note that any redeclared extern symbol is turned into a global symbol. This means we don't have to remove the symbol from the symbol table and add in a new, global, symbol.

Operand Types to Logical Operations

The next bug I hit was something like this:

```
int *x;
int y;

if (x && y > 12) ...
```

The compiler evaluates the `&&` operation in `binexpr()`. To do this, it ensures that the types of each side of the binary operator are compatible. Well, if the operator above was a `+` then, definitely, the types are incompatible. But with a logical comparison, we can *AND* these together.

I've fixed this by adding some more code to the top of `modify_type()` in `types.c`. If we are doing an `&&` or an `||` operation, then we need either integer or pointer types on each side of the operation.

```
struct ASTnode *modify_type(struct ASTnode *tree, int rtype,
                            struct symtable *rctype, int op) {
  int ltype;
  int lsize, rsize;

  ltype = tree->type;
```

```
  // For A_LOGOR and A_LOGAND, both types have to be int or pointer types
  if (op==A_LOGOR || op==A_LOGAND) {
    if (!inttype(ltype) && !ptrtype(ltype))
      return(NULL);
    if (!inttype(ltype) && !ptrtype(rtype))
      return(NULL);
    return (tree);
  }
  ...
}
```

I've also realised that I've implemented `&&` and `||` incorrectly, so I'll have to fix that. Now now, but soon.

Return with No Value

One other C feature that is missing is the ability to return from a void function, i.e. just leave without returning any value. However, the current parser expects to see parentheses and an expression after the `return` keyword.

So, in `return_statement()` in `stmt.c`, we now have:

```
// Parse a return statement and return its AST
static struct ASTnode *return_statement(void) {
  struct ASTnode *tree= NULL;

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

  // See if we have a return value
  if (Token.token == T_LPAREN) {
    // Code to parse the parentheses and the expression
    ...
  } else {
    if (Functionid->type != P_VOID)
      fatal("Must return a value from a non-void function");
  }


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

  // Get the ';'
  semi();
```

```
  return (tree);
}
```

If the `return` token isn't followed by a left parenthesis, we leave the expression `tree` set to NULL. We also check that this is a void returning function, and print out a fatal error if not.

Now that we have parsed the `return` function, we may create an A_RETURN AST node with a NULL child. So now we have to deal with this in the code generator. The top of `cgreturn()` in `cg.c` now has:

```
// Generate code to return a value from a function
void cgreturn(int reg, struct symtable *sym) {

  // Only return a value if we have a value to return
  if (reg != NOREG) {
    ..
  }

  cgjump(sym->st_endlabel);
}
```

If there was no child AST tree, then there is no register with the expression's value. Thus, we only output the jump to the function's end label.

Conclusion and What's Next

We've fixed five minor issues in the compiler: things that we need to work to get the compiler to compile itself.

I did identify a problem with `&&` and `||`. However, before we get to that I need to solve an important, pressing, problem: we have a limited set of CPU registers and, for large source files, we are running out of them.

In the next part of our compiler writing journey, I will have to work on implementing register spills. I've been delaying this, but now most of the fatal errors from the compiler (when compiling itself) are register issues. So now it's time to sort this out. Next step