Part 38: Dangling Else and More

I started this part of our compiler writing journey hoping to fix up the dangling else problem. It turns out that what I actually had to do was restructure the way we parse a few things because I had the parsing wrong in the first place.

This probably happened because I was keen to add functionality, but in the process I didn't step back enough and consider what we had been building.

So, let's see what mistakes in the compiler need fixing.

Fixing Up the For Grammar

We'll start with our FOR loop structure. Yes it works, but it isn't as general as it should be.

Up to now, the BNF grammar for our FOR loop has been:

```
for_statement: 'for' '(' preop_statement ';'
                         true_false_expression ';'
                         postop_statement ')' compound_statement   ;
```

However, the BNF Grammar for C has this:

```
for_statement:
        | FOR '(' expression_statement expression_statement ')' statement
        | FOR '(' expression_statement expression_statement expression ')'
```

```
statement
        ;

expression_statement
        : ';'
        | expression ';'
        ;
```

and an `expression` is actually an expression list where expressions are separated by commas.

This means that all three clauses of a FOR loop can actually be expression lists. If we were writing a "full" C compiler, this would end up being tricky. However, we are only writing a compiler for a *subset* of C, and therefore I don't have to make our compiler deal with the full grammar for C.

So, I have changed the parser for the FOR loop to recognise this:

```
for_statement: 'for' '(' expression_list ';'
                         true_false_expression ';'
                         expression_list ')' compound_statement  ;
```

The middle clause is a single expression that must provide a true or false result. The first and last clauses can be expression lists. This allows a FOR loop like the one now in `tests/input80.c`:

```
    for (x=0, y=1; x < 6; x++, y=y+2)
```

Changes to `expression_list()`

To do the above, I need to modify the `for_statement()` parsing function to call `expression_list()` to parse the list of expressions in the first and third clause.

But, in the existing compiler, `expression_list()` only allows the ')' token to end an expression list. Therefore, I've modified `expression_list()` in `expr.c` to get the end token as an argument. And in `for_statement()` in `stmt.c`, we now have this code:

```
// Parse a FOR statement and return its AST
static struct ASTnode *for_statement(void) {
  ...
  // Get the pre_op expression and the ';'
  preopAST = expression_list(T_SEMI);
```

```
    semi();
    ...
    // Get the condition and the ';'.
    condAST = binexpr(0);
    semi();
    ...
    // Get the post_op expression and the ')'
    postopAST = expression_list(T_RPAREN);
    rparen();
  }
```

And the code in `expression_list()` now looks like this:

```
  struct ASTnode *expression_list(int endtoken) {
    ...
    // Loop until the end token
    while (Token.token != endtoken) {

      // Parse the next expression
      child = binexpr(0);

      // Build an A_GLUE AST node ...
      tree = mkastnode(A_GLUE, P_NONE, tree, NULL, child, NULL, exprcount);

      // Stop when we reach the end token
      if (Token.token == endtoken) break;

      // Must have a ',' at this point
      match(T_COMMA, ",");
    }

    // Return the tree of expressions
    return (tree);
  }
```

Single and Compound Statements

Up to now, I've forced the programmer using our compiler to always put code in '{' ... '}' for:

- the true body of an IF statement
- the false body of an IF statement
- the body of a WHILE statement
- the body of a FOR statement
- the body after a 'case' clause

- the body after a 'default' clause

For the first four statements in this list, we don't need curly brackets when there is only a single statement, e.g.

```
if (x>5)
  x= x - 16;
else
  x++;
```

But when there are multiple statements in the body, we *do* need a compound statement which is a set of single statements surrounded by curly brackets, e.g.

```
if (x>5)
  { x= x - 16; printf("not again!\n"); }
else
  x++;
```

But, for some unknown reason, the code after a 'case' or 'default' clause in a 'switch' statement can be a set of single statements and we don't need curly brackets!! Who was the crazy person who thought that was OK? An example:

```
switch (x) {
  case 1: printf("statement 1\n");
          printf("statement 2\n");
          break;
  default: ...
}
```

Even worse, this is also legal:

```
switch (x) {
  case 1: {
    printf("statement 1\n");
    printf("statement 2\n");
    break;
  }
  default: ...
}
```

Therefore, we need to be able to parse:

- single statements
- a set of statements which are surrounded by curly brackets
- a set of statements which don't start with a '{', but end with one of 'case', 'default', or '}' if they started with '{'

To this end, I've modified the `compound_statement()` in `stmt.c` to take an argument:

```
// Parse a compound statement
// and return its AST. If inswitch is true,
// we look for a '}', 'case' or 'default' token
// to end the parsing. Otherwise, look for
// just a '}' to end the parsing.
struct ASTnode *compound_statement(int inswitch) {
  struct ASTnode *left = NULL;
  struct ASTnode *tree;

  while (1) {
    // Parse a single statement
    tree = single_statement();
    ...
    // Leave if we've hit the end token
    if (Token.token == T_RBRACE) return(left);
    if (inswitch && (Token.token == T_CASE || Token.token == T_DEFAULT)) retur
  }
}
```

If this function get's called with `inswitch` set to 1, then we have been called during the parsing of a 'switch' statement, so look for 'case', 'default' or '}' to end the compound statement. Otherwise, we are in a more typical '{' ... '}' situation.

Now, we also need to allow:

- a single statement inside the body of an IF statement
- a single statement inside the body of an WHILE statement
- a single statement inside the body of a FOR statement

All of these are, at present, calling `compound_statement(0)`, but this enforces the parsing of a closing '}', and we won't have one of these for a single statement.

The answer is to get the IF, WHILE and FOR parsing code to call `single_statement()` to parse one statement. And, get `single_statement()` to call `compound_statement()` if it see an opening curly bracket.

Thus, I've also made these changes in `stmt.c`:

```
// Parse a single statement and return its AST.
static struct ASTnode *single_statement(void) {
  ...
  switch (Token.token) {
    case T_LBRACE:
      // We have a '{', so this is a compound statement
      lbrace();
      stmt = compound_statement(0);
      rbrace();
      return(stmt);
}
...
static struct ASTnode *if_statement(void) {
  ...
  // Get the AST for the statement
  trueAST = single_statement();
  ...
  // If we have an 'else', skip it
  // and get the AST for the statement
  if (Token.token == T_ELSE) {
    scan(&Token);
    falseAST = single_statement();
  }
  ...
}
...
static struct ASTnode *while_statement(void) {
  ...
    // Get the AST for the statement.
  // Update the loop depth in the process
  Looplevel++;
  bodyAST = single_statement();
  Looplevel--;
  ...
}
...
static struct ASTnode *for_statement(void) {
  ...
  // Get the statement which is the body
  // Update the loop depth in the process
  Looplevel++;
  bodyAST = single_statement();
  Looplevel--;
  ...
}
```

This now means the compiler will accept code which looks like this:

```
if (x>5)
  x= x - 16;
else
  x++;
```

Yes, But "Dangling Else?"

I still haven't solved the "dangling else" problem, which after all is why I started this part of the journey. Well, it turns out that this problem was solved due to the way that we already parse our input.

Consider this program:

```
// Dangling else test.
// We should not print anything for x<= 5
for (x=0; x < 12; x++)
  if (x > 5)
    if (x > 10)
      printf("10 < %2d\n", x);
    else
      printf(" 5 < %2d <= 10\n", x);
```

We want the 'else' code to pair up with the nearest 'if' statement. Therefore, the last `printf` statement above should only print when `x` is between 5 and 10. The 'else' code should *not* be invoked due to the opposite of `x > 5`.

Luckily, in our `if_statement()` parser, we greedily scan for any 'else' token after the body of the IF statement:

```
// Get the AST for the statement
trueAST = single_statement();

// If we have an 'else', skip it
// and get the AST for the statement
if (Token.token == T_ELSE) {
  scan(&Token);
  falseAST = single_statement();
}
```

This forces the 'else' to pair up with the nearest 'if' and solves the dangling else problem. So, all this time, I was forcing the use of '{' ... '}' when I'd already solved the problem I was worrying about! Sigh.

Some Better Debug Output

Finally, I've made a change to our scanner to improve debugging. Or, more exactly, to improve the debug messages that we print out. Up to now, we have been printing the token numeric value in our error messages, e.g.

- Unexpected token in parameter list: 23
- Expecting a primary expression, got token: 19
- Syntax error, token: 44

For the programmer who receives these error messages, they are essentially unusable. In `scan.c`, I've added this list of token strings:

```
// List of token strings, for debugging purposes
char *Tstring[] = {
  "EOF", "=", "||", "&&", "|", "^", "&",
  "==", "!=", ",", ">", "<=", ">=", "<<", ">>",
  "+", "-", "*", "/", "++", "--", "~", "!",
  "void", "char", "int", "long",
  "if", "else", "while", "for", "return",
  "struct", "union", "enum", "typedef",
  "extern", "break", "continue", "switch",
  "case", "default",
  "intlit", "strlit", ";", "identifier",
  "{", "}", "(", ")", "[", "]", ",", ".",
  "->", ":"
};
```

In `defs.h`, I've added another field to the Token structure:

```
// Token structure
struct token {
  int token;                    // Token type, from the enum list above
  char *tokstr;                 // String version of the token
  int intvalue;                 // For T_INTLIT, the integer value
};
```

In `scan()` in `scan.c`, just before we return a token, we set up its string equivalent:

```
  t->tokstr = Tstring[t->token];
```

And, finally, I've modified a bunch of `fatalXX()` calls to print out the `tokstr` field of the current token instead of the `intvalue` field. This means we now see:

- Unexpected token in parameter list: ==
- Expecting a primary expression, got token:]
- Syntax error, token: >>

which is much better.

Conclusion and What's Next

I set out to solve the "dangling else" misfeature in our compiler and ended up fixing a bunch of other misfeatures. In the process, I found out that there was no "dangling else" problem to solve.

We have reached a stage in the development of the compiler where all the essential elements we need to self-compile the compiler are implemented, but now we need to find and fix a bunch of small issues. This is the "mop up" phase.

What this means is, from now on, there will be less and less on how to write a compiler, and more and more on how to fix a broken compiler. I won't be disappointed if you choose to bail out on the future parts of our journey. If you do, I hope that you found all the parts of the journey so far useful.

In the next part of our compiler writing journey, I will pick something that currently doesn't work but we need to work to self-compile our compiler, and fix it. Next step