Part 10: FOR Loops

In this part of our compiler writing journey I'm going to add FOR loops. There is a wrinkle to work out in terms of implementation which I want to explain before I get to the discussion on how it got solved.

The FOR Loop Syntax

I assume that you are familiar with the syntax of FOR loops. One example is

```
for (i=0; i < MAX; i++)
  printf("%d\n", i);
```

I'm going to use this BNF syntax for our language:

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

preop_statement:  statement  ;        (for now)
postop_statement: statement  ;        (for now)
```

The `preop_statement` is run before the loop starts. Later on, we will have to limit exactly what sort of actions can be performed here (e.g. no IF statements). Then the `true_false_expression` is evaluated. If true the loop executes the `compound_statement`. Once this is done, the `postop_statement` is performed and the code loops back to redo the `true_false_expression`.

The Wrinkle

The wrinkle is that the `postop_statement` is parsed before the `compound_statement`, but we have to generate the code for the `postop_statement` *after* the code for the `compound_statement`.

There are several ways to solve this problem. When I wrote a previous compiler, I chose to put the `compound_statement` assembly code in a temporary buffer, and "play back" the buffer once I'd generated the code for the `postop_statement`. In the SubC compiler, Nils makes clever use of labels and jumps to labels to "thread" the code's execution to enforce the correct sequence.

But we build an AST tree here. Let's use it to get the generated assembly code in the correct sequence.

What Sort of AST Tree?

You might have noticed that a FOR loop has four structural components:

1. The `preop_statement`
2. The `true_false_expression`
3. The `postop_statement`
4. The `compound_statement`

I don't really want to change the AST node structure yet again to have four children. But we can visualise a FOR loop as an augmented WHILE loop:

```
preop_statement;
while ( true_false_expression ) {
  compound_statement;
  postop_statement;
}
```

Can we build an AST tree with our existing node types to reflect this structure? Yes:

```
         A_GLUE
        /     \
   preop     A_WHILE
             /    \
        decision  A_GLUE
                  /    \
            compound  postop
```

Manually traverse this tree top-down left-to-right and convince yourself that we will generate the assembly code in the right order. We had to glue the `compound_statement` and the `postop_statement` together so that, when the WHILE loop exits, it will skip over both the `compound_statement` and the `postop_statement`.

This also means that we need a new T_FOR token but we won't need a new AST node type. So the only compiler change will be scanning and parsing.

Tokens and Scanning

There is a new keyword 'for' and an associated token, T_FOR. No big changes here.

Parsing Statements

We do need to make a structural change to the parser. For the FOR grammar, I only want a single statement as the `preop_statement` and the `postop_statement`. Right now, we have a `compound_statement()` function that simply loops until it hits a right curly bracket '}'. We need to separate this out so `compound_statement()` calls `single_statement()` to get one statement.

But there's another wrinkle. Take the existing parsing of assignment statements in `assignment_statement()`. The parser must find a semicolon at the end of the statement.

That's good for compound statements but it won't work for FOR loops. I would have to write something like:

```
  for (i=1 ; i < 10 ; i= i + 1; )
```

because each assignment statement *must* end with a semicolon.

What we need is for the single statement parser *not* to scan in the semicolon, but to leave that up to the compound statement parser. And we scan in semicolons for some statements (e.g. between assignment statements) and not for other statements (e.g. not between successive IF statements).

With all of that explained, let's now look at the new single and compound statement parsing code:

```
// Parse a single statement
// and return its AST
static struct ASTnode *single_statement(void) {
  switch (Token.token) {
    case T_PRINT:
      return (print_statement());
    case T_INT:
      var_declaration();
      return (NULL);            // No AST generated here
    case T_IDENT:
      return (assignment_statement());
    case T_IF:
      return (if_statement());
    case T_WHILE:
      return (while_statement());
    case T_FOR:
      return (for_statement());
    default:
      fatald("Syntax error, token", Token.token);
  }
}

// Parse a compound statement
// and return its AST
struct ASTnode *compound_statement(void) {
  struct ASTnode *left = NULL;
  struct ASTnode *tree;

  // Require a left curly bracket
  lbrace();

  while (1) {
    // Parse a single statement
    tree = single_statement();

    // Some statements must be followed by a semicolon
    if (tree != NULL &&
        (tree->op == A_PRINT || tree->op == A_ASSIGN))
      semi();
```

```
      // For each new tree, either save it in left
      // if left is empty, or glue the left and the
      // new tree together
      if (tree != NULL) {
        if (left == NULL)
          left = tree;
        else
          left = mkastnode(A_GLUE, left, NULL, tree, 0);
      }
      // When we hit a right curly bracket,
      // skip past it and return the AST
      if (Token.token == T_RBRACE) {
        rbrace();
        return (left);
      }
    }
  }
```

I've also removed the calls to `semi()` in `print_statement()` and `assignment_statement()`.

Parsing FOR Loops

Given the BNF syntax for FOR loops above, this is straightforward. And given the shape of the AST tree we want, the code to build this tree is also straightforward. Here's the code:

```
// Parse a FOR statement
// and return its AST
static struct ASTnode *for_statement(void) {
  struct ASTnode *condAST, *bodyAST;
  struct ASTnode *preopAST, *postopAST;
  struct ASTnode *tree;

  // Ensure we have 'for' '('
  match(T_FOR, "for");
  lparen();

  // Get the pre_op statement and the ';'
  preopAST= single_statement();
  semi();

  // Get the condition and the ';'
  condAST = binexpr(0);
  if (condAST->op < A_EQ || condAST->op > A_GE)
    fatal("Bad comparison operator");
  semi();
```

```
    // Get the post_op statement and the ')'
    postopAST= single_statement();
    rparen();

    // Get the compound statement which is the body
    bodyAST = compound_statement();

    // For now, all four sub-trees have to be non-NULL.
    // Later on, we'll change the semantics for when some are missing

    // Glue the compound statement and the postop tree
    tree= mkastnode(A_GLUE, bodyAST, NULL, postopAST, 0);

    // Make a WHILE loop with the condition and this new body
    tree= mkastnode(A_WHILE, condAST, NULL, tree, 0);

    // And glue the preop tree to the A_WHILE tree
    return(mkastnode(A_GLUE, preopAST, NULL, tree, 0));
  }
```

Generating the Assembly Code

Well, all we have done is synthesized a tree which has a WHILE loop in it with some sub-trees glued together, so there are no changes to the generation side of the compiler.

Trying It Out

The `tests/input07` file has this program in it:

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

When we do `make test7`, we get this output:

```
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/input07
cc -o out out.s
./out
1
```

```
2
3
4
5
6
7
8
9
10
```

and here is the relevant assembly output:

```
        .comm   i,8,8
        movq    $1, %r8
        movq    %r8, i(%rip)            # i = 1
L1:
        movq    i(%rip), %r8
        movq    $10, %r9
        cmpq    %r9, %r8                # Is i < 10?
        jg      L2                      # i >= 10, jump to L2
        movq    i(%rip), %r8
        movq    %r8, %rdi
        call    printint                # print i
        movq    i(%rip), %r8
        movq    $1, %r9
        addq    %r8, %r9                # i = i + 1
        movq    %r9, i(%rip)
        jmp     L1                      # Jump to top of loop
L2:
```

Conclusion and What's Next

We now have a reasonable number of control structures in our language: IF statements, WHILE loops and FOR loops. The question is, what to tackle next? There are so many things we could look at:

- types
- local versus global things
- functions
- arrays and pointers
- structures and unions
- auto, static and friends

I've decided to look at functions. So, in the next part of our compiler writing journey, we will begin the first of several stages to add functions to our language. [Next step]