Part 9: While Loops

In this part of the journey we are going to add WHILE loops to our language. In some sense, a WHILE loop is very much like an IF statement without an 'else' clause, except that we always jump back to the top of the loop.

So, this:

```
while (condition is true) {
  statements;
}
```

should get translated to:

```
Lstart: evaluate condition
        jump to Lend if condition false
        statements
        jump to Lstart
Lend:
```

This means that we can borrow the scanning, parsing and code generation structures that we used with IF statements and make some small changes to also deal with WHILE statements.

Let's see how we make this happen.

New Tokens

We need a new token, T_WHILE, for the new 'while' keyword. The changes to `defs.h` and `scan.c` are obvious so I'll omit them here.

Parsing the While Syntax

The BNF grammar for the WHILE loop is:

```
// while_statement: 'while' '(' true_false_expression ')'
compound_statement  ;
```

and we need a function in `stmt.c` to parse this. Here it is; note the simplicity of this compared to the parsing of IF statements:

```
// Parse a WHILE statement
// and return its AST
struct ASTnode *while_statement(void) {
  struct ASTnode *condAST, *bodyAST;

  // Ensure we have 'while' '('
  match(T_WHILE, "while");
  lparen();

  // Parse the following expression
  // and the ')' following. Ensure
  // the tree's operation is a comparison.
  condAST = binexpr(0);
  if (condAST->op < A_EQ || condAST->op > A_GE)
    fatal("Bad comparison operator");
  rparen();

  // Get the AST for the compound statement
  bodyAST = compound_statement();

  // Build and return the AST for this statement
  return (mkastnode(A_WHILE, condAST, NULL, bodyAST, 0));
}
```

We need a new AST node type, A_WHILE, which has been added to `defs.h` . This node has a left child sub-tree to evaluate the condition, and a right child sub-tree for the compound statement which is the body of the WHILE loop.

Generic Code Generation

We need to create a start and end label, evaluate the condition and insert appropriate jumps to exit the loop and to return to the top of the loop. Again, this is much simpler than the code to generate IF statements. In `gen.c`:

```
// Generate the code for a WHILE statement
// and an optional ELSE clause
static int genWHILE(struct ASTnode *n) {
  int Lstart, Lend;

  // Generate the start and end labels
  // and output the start label
  Lstart = label();
  Lend = label();
  cglabel(Lstart);

  // Generate the condition code followed
  // by a jump to the end label.
  // We cheat by sending the Lfalse label as a register.
  genAST(n->left, Lend, n->op);
  genfreeregs();

  // Generate the compound statement for the body
  genAST(n->right, NOREG, n->op);
  genfreeregs();

  // Finally output the jump back to the condition,
  // and the end label
  cgjump(Lstart);
  cglabel(Lend);
  return (NOREG);
}
```

One thing I had to do was recognise that the parent AST node of the comparison operators could now be A_WHILE, so in `genAST()` the code for the comparison operators looks like:

```
    case A_EQ:
    case A_NE:
    case A_LT:
    case A_GT:
    case A_LE:
    case A_GE:
      // If the parent AST node is an A_IF or A_WHILE, generate
```

```
      // a compare followed by a jump. Otherwise, compare registers
      // and set one to 1 or 0 based on the comparison.
      if (parentASTop == A_IF || parentASTop == A_WHILE)
        return (cgcompare_and_jump(n->op, leftreg, rightreg, reg));
      else
        return (cgcompare_and_set(n->op, leftreg, rightreg));
```

And that, altogether, is all we need to implement WHILE loops!

Testing the New Language Additions

I've moved all of the input files into a `test/` directory. If you now do `make test`, it will go into this directory, compile each input and compare the output against known-good 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
(cd tests; chmod +x runtests; ./runtests)
input01: OK
input02: OK
input03: OK
input04: OK
input05: OK
input06: OK
```

You can also do a `make test6`. This compiles the `tests/input06` file:

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

This will print out the numbers from 1 to 10:

```
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/input06
cc -o out out.s
./out
1
```

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

And here is the assembly output from the compilation:

```
        .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                      # Greater than, jump to L2
        movq    i(%rip), %r8
        movq    %r8, %rdi               # Print out i
        call    printint
        movq    i(%rip), %r8
        movq    $1, %r9
        addq    %r8, %r9                # Add 1 to i
        movq    %r9, i(%rip)
        jmp     L1                      # and loop back
L2:
```

Conclusion and What's Next

The WHILE loop was easy to add, once we had already done the IF statement as they share a lot of similarities.

I think we also now have a Turing-complete language:

- an infinite amount of storage, i.e. an infinite number of variables
- the ability to make decisions based on stored values, i.e. IF statements
- the ability to change directions, i.e. WHILE loops

So we can stop now, our job is done! No, of course not. We are still working towards getting the compiler to compile itself.

In the next part of our compiler writing journey, we will add FOR loops to the language.
Next step