Part 5: Statements

It's time to add some "proper" statements to the grammar of our language. I want to be able to write lines of code like this:

```
print 2 + 3 * 5;
print 18 - 6/3 + 4*2;
```

Of course, as we are ignoring whitespace, there's no necessity that all the tokens for one statement are on the same line. Each statement starts with the keyword `print` and is terminated with a semicolon. So these are going to become new tokens in our language.

BNF Description of the Grammar

We've already seen the BNF notation for expressions. Now let's define the BNF syntax for the above types of statements:

```
statements: statement
      | statement statements
      ;

statement: 'print' expression ';'
      ;
```

An input file consists of several statements. They are either one statement, or a statement followed by more statements. Each statement starts with the keyword `print`, then one expression, then a semicolon.

Changes to the Lexical Scanner

Before we can get to the code that parses the above syntax, we need to add a few more bits and pieces to the existing code. Let's start with the lexical scanner.

Adding a token for semicolons will be easy. Now, the `print` keyword. Later on, we'll have many keywords in the language, plus identifiers for our variables, so we'll need to add some code which helps us to deal with them.

In `scan.c`, I've added this code which I've borrowed from the SubC compiler. It reads in alphanumeric characters into a buffer until it hits a non-alphanumeric character.

```
// Scan an identifier from the input file and
// store it in buf[]. Return the identifier's length
static int scanident(int c, char *buf, int lim) {
  int i = 0;

  // Allow digits, alpha and underscores
  while (isalpha(c) || isdigit(c) || '_' == c) {
    // Error if we hit the identifier length limit,
    // else append to buf[] and get next character
    if (lim - 1 == i) {
      printf("identifier too long on line %d\n", Line);
      exit(1);
    } else if (i < lim - 1) {
      buf[i++] = c;
    }
    c = next();
  }
  // We hit a non-valid character, put it back.
  // NUL-terminate the buf[] and return the length
  putback(c);
  buf[i] = '\0';
  return (i);
}
```

We also need a function to recognise keywords in the language. One way would be to have a list of keywords, and to walk the list and `strcmp()` each one against the buffer from `scanident()`. The code from SubC has an optimisation: match against the first letter before doing the `strcmp()`. This speeds up the comparison against dozens of keywords. Right now we don't need this optimisation but I've put it in for later:

```
// Given a word from the input, return the matching
// keyword token number or 0 if it's not a keyword.
// Switch on the first letter so that we don't have
// to waste time strcmp()ing against all the keywords.
static int keyword(char *s) {
  switch (*s) {
    case 'p':
      if (!strcmp(s, "print"))
        return (T_PRINT);
      break;
  }
  return (0);
}
```

Now, at the bottom of the switch statement in `scan()`, we add this code to recognise semicolons and keywords:

```
    case ';':
      t->token = T_SEMI;
      break;
    default:

      // If it's a digit, scan the
      // literal integer value in
      if (isdigit(c)) {
        t->intvalue = scanint(c);
        t->token = T_INTLIT;
        break;
      } else if (isalpha(c) || '_' == c) {
        // Read in a keyword or identifier
        scanident(c, Text, TEXTLEN);

        // If it's a recognised keyword, return that token
        if (tokentype = keyword(Text)) {
          t->token = tokentype;
          break;
        }
        // Not a recognised keyword, so an error for now
        printf("Unrecognised symbol %s on line %d\n", Text, Line);
        exit(1);
```

```
        }
        // The character isn't part of any recognised token, error
        printf("Unrecognised character %c on line %d\n", c, Line);
        exit(1);
```

I've also added a global `Text` buffer to store the keywords and identifiers:

```
#define TEXTLEN         512             // Length of symbols in input
extern_ char Text[TEXTLEN + 1];         // Last identifier scanned
```

Changes to the Expression Parser

Up to now our input files have contained just a single expression; therefore, in our Pratt parser code in `binexpr()` (in `expr.c`), we had this code to exit the parser:

```
  // If no tokens left, return just the left node
  tokentype = Token.token;
  if (tokentype == T_EOF)
    return (left);
```

With our new grammar, each expression is terminated by a semicolon. Thus, we need to change the code in the expression parser to spot the `T_SEMI` tokens and exit the expression parsing:

```
// Return an AST tree whose root is a binary operator.
// Parameter ptp is the previous token's precedence.
struct ASTnode *binexpr(int ptp) {
  struct ASTnode *left, *right;
  int tokentype;

  // Get the integer literal on the left.
  // Fetch the next token at the same time.
  left = primary();

  // If we hit a semicolon, return just the left node
  tokentype = Token.token;
  if (tokentype == T_SEMI)
    return (left);

    while (op_precedence(tokentype) > ptp) {
      ...

          // Update the details of the current token.
```

```
    // If we hit a semicolon, return just the left node
    tokentype = Token.token;
    if (tokentype == T_SEMI)
      return (left);
    }
  }
```

Changes to the Code Generator

I want to keep the generic code generator in `gen.c` separate from the CPU-specific code in `cg.c`. That also means that the rest of the compiler should only ever call the functions in `gen.c`, and only `gen.c` should call the code in `cg.c`.

To this end, I've defined some new "front-end" functions in `gen.c`:

```
void genpreamble()        { cgpreamble(); }
void genpostamble()       { cgpostamble(); }
void genfreeregs()        { freeall_registers(); }
void genprintint(int reg) { cgprintint(reg); }
```

Adding the Parser for Statements

We have a new file `stmt.c`. This will hold the parsing code for all the main statements in our language. Right now, we need to parse the BNF grammar for statements which I gave up above. This is done with this single function. I've converted the recursive definition into a loop:

```
// Parse one or more statements
void statements(void) {
  struct ASTnode *tree;
  int reg;

  while (1) {
    // Match a 'print' as the first token
    match(T_PRINT, "print");

    // Parse the following expression and
    // generate the assembly code
    tree = binexpr(0);
    reg = genAST(tree);
    genprintint(reg);
    genfreeregs();
```

```
      // Match the following semicolon
      // and stop if we are at EOF
      semi();
      if (Token.token == T_EOF)
        return;
    }
  }
```

In each loop, the code finds a T_PRINT token. It then calls `binexpr()` to parse the expression. Finally, it finds the T_SEMI token. If a T_EOF token follows, we break out of the loop.

After each expression tree, the code in `gen.c` is called to convert the tree into assembly code and to call the assembly `printint()` function to print out the final value.

Some Helper Functions

There are a couple of new helper functions in the above code, which I've put into a new file, `misc.c` :

```
// Ensure that the current token is t,
// and fetch the next token. Otherwise
// throw an error
void match(int t, char *what) {
  if (Token.token == t) {
    scan(&Token);
  } else {
    printf("%s expected on line %d\n", what, Line);
    exit(1);
  }
}

// Match a semicon and fetch the next token
void semi(void) {
  match(T_SEMI, ";");
}
```

These form part of the syntax checking in the parser. Later on, I'll add more short functions to call `match()` to make our syntax checking easier.

Changes to `main()`

`main()` used to call `binexpr()` directly to parse the single expression in the old input files. Now it does this:

```
  scan(&Token);                 // Get the first token from the input
  genpreamble();                // Output the preamble
  statements();                 // Parse the statements in the input
  genpostamble();               // Output the postamble
  fclose(Outfile);              // Close the output file and exit
  exit(0);
```

Trying It Out

That's about it for the new and changed code. Let's give the new code a whirl. Here is the new input file, `input01` :

```
print 12 * 3;
print
   18 - 2
      * 4; print
1 + 2 +
  9 - 5/2 + 3*5;
```

Yes I've decided to check that we have have tokens spread out across multiple lines. To compile and run the input file, do a `make test` :

```
$ make test
cc -o comp1 -g cg.c expr.c gen.c main.c misc.c scan.c stmt.c tree.c
./comp1 input01
cc -o out out.s
./out
36
10
25
```

And it works!

Conclusion and What's Next

We've added our first "real" statement grammar to our language. I've defined it in BNF notation, but it was easier to implement it with a loop and not recursively. Don't worry, we'll go back to doing recursive parsing soon.

Along the way we had to modify the scanner, add support for keywords and identifiers, and to more cleanly separate the generic code generator and the CPU-specific generator.

In the next part of our compiler writing journey, we will add variables to the language. This will require a significant amount of work. Next step