Part 43: Bugfixes and More Operators

I've started to pass some of the source code of our compiler as input to itself, as this is how we are going to get it to eventually compile itself. The first big hurdle is to get the compiler to parse and recognise its source code. The second big hurdle will be to get the compiler to generate correct, working, code from its source code.

This is also the first time that the compiler has been given some substantial input to chew on, and it's going to reveal a bunch of bugs, misfeatures and missing features.

Bugfixes

I started with `cwj -S defs.h` and found several header files missing. For now they exist but are empty. With these in place, the compiler crashes with a segfault. I had a few pointers which should be initialised to NULL and places where I wasn't checking for a NULL pointer.

Missing Features

Next up, I hit `enum { NOREG = -1 ...` in `defs.h` and realised that the scanner wasn't dealing with integer literals which start with a minus sign. So I've added this code to `scan()` in `scan.c`:

```
    case '-':
      if ((c = next()) == '-') {
        t->token = T_DEC;
      } else if (c == '>') {
        t->token = T_ARROW;
      } else if (isdigit(c)) {          // Negative int literal
        t->intvalue = -scanint(c);
        t->token = T_INTLIT;
      } else {
        putback(c);
        t->token = T_MINUS;
      }
```

If a '-' is followed by a digit, scan in the integer literal and negate its value. At first I was worried that the expression `1 - 1` would be treated as the two tokens '1', 'int literal -1', but I forgot that `next()` doesn't skip the space. So, by having a space between the '-' and the '1', the expression `1 - 1` is correctly parsed as '1', '-', '1'.

However, as Luke Gruber has pointed out, this also means that the input `1-1` **is** treated as `1 -1` instead of `1 - 1`. In other words, the scanner is too greedy and forces `-1` to always be treated as a T_INTLIT when sometimes it shouldn't be. I'm going to leave this for now, as we can work around it when writing our source code. Obviously, in a production compiler this would have to be fixed.

Misfeatures

In the AST node and symbol table node structures, I've been using unions to try and keep the size of each node down. I guess I'm a bit old school and I worry about wasting memory. An example is the AST node structure:

```
struct ASTnode {
  int op;                       // "Operation" to be performed on this tree
  ...
  union {                       // the symbol in the symbol table
    int intvalue;               // For A_INTLIT, the integer value
    int size;                   // For A_SCALE, the size to scale by
  };
};
```

But the compiler isn't able to parse and work with a union inside a struct, and especially an unnamed union inside a struct. I could add this functionality, but it will be easier to redo the two structs where I do this. So, I've made these changes:

```
// Symbol table structure
struct symtable {
  char *name;                   // Name of a symbol
  ...
#define st_endlabel st_posn     // For functions, the end label
  int st_posn;                  // For locals, the negative offset
                                // from the stack base pointer
  ...
};

// Abstract Syntax Tree structure
struct ASTnode {
  int op;                       // "Operation" to be performed on this tree
  ...
#define a_intvalue a_size       // For A_INTLIT, the integer value
  int a_size;                   // For A_SCALE, the size to scale by
};
```

This way, I still have two named fields sharing the same location in each struct, but the compiler will see only the one field name in each struct. I've given each `#define` a different prefix to prevent pollution of the global namespace.

A consequence of this is that I've had to rename the `endlabel`, `posn`, `intvalue` and `size` fields across half a dozen source files. C'est la vie.

So now the compiler, when doing `cwj -S misc.c` gets up to:

```
Expected:] on line 16 of data.h, where the line is
extern char Text[TEXTLEN + 1];
```

This fails as the compiler as it stands does not parse expressions in a global variable declaration. I'm going to have to rethink this.

My thoughts so far are to use `binexpr()` to parse the expression, and to add some optimisation code to perform constant folding on the resulting AST tree. This should result in a single A_INTLIT node from which I can extract the literal value. I could even let `binexpr()` parse any casts, e.g.

```
 char x= (char)('a' + 1024);
```

Anyway, that's something for the future. I was going to do constant folding at some point, but I thought it would be further down the track.

What I will do in this part of the journey is add some more operators: specifically, '+=', '-=', '*=' and '/='. We currently use the first two operators in the compiler's source code.

New Tokens, Scanning and Parsing

Adding new keywords to our compiler is easy: a new token and a change to the scanner. Adding new operators is much harder as we have to:

- align the token with the AST operation
- deal with precedence and associativity.

We are adding four operators: '+=', '-=', '*=' and '/='. They have matching tokens: T_ASPLUS, T_ASMINUS, T_ASSTAR and T_ASSLASH. These have corresponding AST operations: A_ASPLUS, A_ASMINUS, A_ASSTAR, A_ASSLASH. The AST operations **must** have the same enum value as the tokens because of this function in `expr.c`:

```
// Convert a binary operator token into a binary AST operation.
// We rely on a 1:1 mapping from token to AST operation
static int binastop(int tokentype) {
  if (tokentype > T_EOF && tokentype <= T_SLASH)
    return (tokentype);
  fatald("Syntax error, token", tokentype);
  return (0);                     // Keep -Wall happy
}
```

We also need to configure the precedence of the new operators. According to this list of C operators, these new operators have the same precedence as our existing assignment operator, so we can modify the `OpPrec[]` table in `expr.c` as follows:

```
// Operator precedence for each token. Must
// match up with the order of tokens in defs.h
static int OpPrec[] = {
  0, 10, 10,                      // T_EOF, T_ASSIGN, T_ASPLUS,
  10, 10, 10,                     // T_ASMINUS, T_ASSTAR, T_ASSLASH,
  20, 30,                         // T_LOGOR, T_LOGAND
  ...
};
```

But that list of C operators also notes that the assignment operators are *right_associative*. This means, for example, that:

```
    a += b + c;          // needs to be parsed as
    a += (b + c);        // not
    (a += b) + c;
```

So we also need to update this function in `expr.c` to do this:

```
// Return true if a token is right-associative,
// false otherwise.
static int rightassoc(int tokentype) {
  if (tokentype >= T_ASSIGN && tokentype <= T_ASSLASH)
    return (1);
  return (0);
}
```

Fortunately, these are the only changes we need to make to our scanner and expression parser: the Pratt parser for binary expressions is now primed to deal with the new operators.

Dealing with the AST Tree

Now that we can parse expressions with the four new operators, we need to deal with the AST that is created for each expression. One thing we need to do is dump the AST tree. So, in `dumpAST()` in `tree.c`, I added this code:

```
    case A_ASPLUS:
      fprintf(stdout, "A_ASPLUS\n"); return;
    case A_ASMINUS:
      fprintf(stdout, "A_ASMINUS\n"); return;
    case A_ASSTAR:
      fprintf(stdout, "A_ASSTAR\n"); return;
    case A_ASSLASH:
      fprintf(stdout, "A_ASSLASH\n"); return;
```

Now when I run `cwj -T input.c` with the expression `a += b + c`, I see:

```
  A_IDENT rval a
    A_IDENT rval b
    A_IDENT rval c
  A_ADD
A_ASPLUS
```

which we can redraw as:

```
          A_ASPLUS
        /          \
    A_IDENT      A_ADD
```

```
       rval a     /     \
                A_IDENT  A_IDENT
                 rval b  rval c
```

Generating the Assembly For the Operators

Well, in `gen.c` we already walk the AST tree and deal with A_ADD and A_ASSIGN. Is there a way to use the existing code to make implementing the new A_ASPLUS operator a bit easier? Yes!

We can rewrite the above AST tree to look like this:

```
                  A_ASSIGN
                 /       \
              A_ADD      lval a
           /        \
       A_IDENT     A_ADD
       rval a     /     \
              A_IDENT   A_IDENT
               rval b   rval c
```

Now, we don't *have* to rewrite the tree as long as we perform the tree walking *as if* the tree had been rewritten like this.

So in `genAST()`, we have:

```c
int genAST(...) {
  ...
  // Get the left and right sub-tree values. This code already here.
  if (n->left)
    leftreg = genAST(n->left, NOLABEL, NOLABEL, NOLABEL, n->op);
  if (n->right)
    rightreg = genAST(n->right, NOLABEL, NOLABEL, NOLABEL, n->op);
}
```

From the perspective of doing the work for the A_ASPLUS node, we have evaluated the left-hand child (e.g. `a`'s value) and the right-hand child (e.g. `b+c`) and we have the values in two registers. If this was an A_ADD operation, we would `cgadd(leftreg, rightreg)` at this point. Well, it is an A_ADD operation on these children, then followed by an assignment back into `a`.

So, the `genAST()` code now has this:

```
  switch (n->op) {
    ...
    case A_ASPLUS:
    case A_ASMINUS:
    case A_ASSTAR:
    case A_ASSLASH:
    case A_ASSIGN:

      // For the '+=' and friends operators, generate suitable code
      // and get the register with the result. Then take the left child,
      // make it the right child so that we can fall into the assignment code.
      switch (n->op) {
        case A_ASPLUS:
          leftreg= cgadd(leftreg, rightreg);
          n->right= n->left;
          break;
        case A_ASMINUS:
          leftreg= cgsub(leftreg, rightreg);
          n->right= n->left;
          break;
        case A_ASSTAR:
          leftreg= cgmul(leftreg, rightreg);
          n->right= n->left;
          break;
        case A_ASSLASH:
          leftreg= cgdiv(leftreg, rightreg);
          n->right= n->left;
          break;
      }

      // And the existing code to do A_ASSIGN is here
     ...
  }
```

In other words, for each new operator, we perform the correct maths operation on the children. But before we can drop into the A_ASSIGN we have to move the left-child pointer over to be the right child. Why? Because the A_ASSIGN code expects the destination to be the right child:

```
      return (cgstorlocal(leftreg, n->right->sym));
```

And that's it. We were lucky to have code which we could adapt to add in these four new operators. There are more assignment operators which I haven't implemented: '%=', '<=', '>>=', '&=', '^=' and '|='. They should also be as easy to add as the four we just added.

Example Code

The `tests/input110.c` program is our testing program:

```
#include <stdio.h>

int x;
int y;

int main() {
  x= 3; y= 15; y += x; printf("%d\n", y);
  x= 3; y= 15; y -= x; printf("%d\n", y);
  x= 3; y= 15; y *= x; printf("%d\n", y);
  x= 3; y= 15; y /= x; printf("%d\n", y);
  return(0);
}
```

and produces these results:

```
18
12
45
5
```

Conclusion and What's Next

We've added some more operators, and the hardest part really was aligning all the tokens, the AST operators and setting the precedence levels and right-associativity. After that, we could reuse some of the code generation code in `genAST()` to make our lives a bit easier.

In the next part of our compiler writing journey, it looks like I'll be adding constant folding to the compiler. Next step