Part 50: Mopping Up, part 1

We have definitely reached the "mopping up" phase, as in this part of our compiler writing journey I don't introduce any major feature. Instead, I fix a couple of problems and add a couple of minor functions.

Consecutive Cases

At present, the compiler can't parse

```
switch(x) {
  case 1:
  case 2: printf("Hello\n");
}
```

because the parser expects a compound statement after the ':' token. In `switch_statement()` in `stmt.c` :

```
        // Scan the ':' and get the compound expression
        match(T_COLON, ":");
        left= compound_statement(1); casecount++;
        ...
        // Build a sub-tree with the compound statement as the left child
        casetail->right= mkastunary(ASTop, 0, left, NULL, casevalue);
```

What we want is to allow an empty compound statement, so that any case with a missing compound statement falls down into the next existing compound statement.

The change in `switch_statement()` is:

```
        // Scan the ':' and increment the casecount
        match(T_COLON, ":");
        casecount++;

        // If the next token is a T_CASE, the existing case will fall
        // into the next case. Otherwise, parse the case body.
        if (Token.token == T_CASE)
          body= NULL;
        else
          body= compound_statement(1);
```

This is, however, only half the story. Now in the code generation section, we have to catch the NULL compound statement and do something about it. In `genSWITCH()` in `gen.c` :

```
    // Walk the right-child linked list to
    // generate the code for each case
    for (i = 0, c = n->right; c != NULL; i++, c = c->right) {
      ...
      // Generate the case code. Pass in the end label for the breaks.
      // If case has no body, we will fall into the following body.
      if (c->left) genAST(c->left, NOLABEL, NOLABEL, Lend, 0);
      genfreeregs(NOREG);
    }
```

So, this was a nice and simple fix. `tests/input123.c` is the test program to confirm this change works.

Dumping the Symbol Table

While I was trying to work out why the global `Text` variable wasn't visible to the compiler, I added code in `sym.c` to dump the symbol table at the end of every source code file. There is an `-M` command line argument to enable the functionality. I won't go through the code, but here is an example of its output:

```
Symbols for misc.c
Global
--------
void exit(): global, 1 params
```

```
    int status: param, size 4
void _Exit(): global, 1 params
    int status: param, size 4
void *malloc(): global, 1 params
    int size: param, size 4
...
int Line: extern, size 4
int Putback: extern, size 4
struct symtable *Functionid: extern, size 8
char **Infile: extern, size 8
char **Outfile: extern, size 8
char *Text[]: extern, 513 elems, size 513
struct symtable *Globhead: extern, size 8
struct symtable *Globtail: extern, size 8
...
struct mkastleaf *mkastleaf(): global, 4 params
    int op: param, size 4
    int type: param, size 4
    struct symtable *sym: param, size 8
    int intvalue: param, size 4
...
Enums
--------
int (null): enumtype, size 0
int TEXTLEN: enumval, value 512
int (null): enumtype, size 0
int T_EOF: enumval, value 0
int T_ASSIGN: enumval, value 1
int T_ASPLUS: enumval, value 2
int T_ASMINUS: enumval, value 3
int T_ASSTAR: enumval, value 4
int T_ASSLASH: enumval, value 5
...
Typedefs
--------
long size_t: typedef, size 0
char *FILE: typedef, size 0
```

Passing Arrays as Arguments

I made the following change, but in hindsight I realise that I probably need to rethink how I deal with arrays completely. Anyway ... when I compile `decl.c` with the compiler, I get the error:

```
Unknown variable:Text on line 87 of decl.c
```

which prompted me to write the symbol dumping code. `Text` is in the global symbol table, so why is the parser complaining that it's missing?

The answer is that `postfix()` in `expr.c`, after finding an identifier, consults the following token. If it is a '[', then the identifier must be an array. If there is no '[', then the identifier must be a variable:

```
// A variable. Check that the variable exists.
if ((varptr = findsymbol(Text)) == NULL || varptr->stype != S_VARIABLE)
  fatals("Unknown variable", Text);
```

This is preventing the passing of an array reference as an argument to a function. The "offending" line that prompts the error message is in `decl.c`:

```
      type = type_of_typedef(Text, ctype);
```

We are passing the address of the base of `Text` as an argument. But with no following '[', our compiler thinks that it's a scalar variable, and complains that there is no scalar variable `Text`.

I made the change to allow S_ARRAY as well as S_VARIABLE here, but this is just the tip of a bigger problem: arrays and pointers in our compiler are not as interchangeable as they should be. I'll tackle this in the next part.

Missing Operators

In our compiler, we've had these tokens and AST operators since part 21 of the journey:

- `||`, T_LOGOR, A_LOGOR
- `&&`, T_LOGAND, A_LOGAND

Somehow, I'd never implemented them! So, it's time to do them.

For A_LOGAND, we have two expressions. If both evaluate to true, we need to set a register to the rvalue of 1, otherwise 0. For A_LOGOR, if either evaluate to true, we need to set a register to the rvalue of 1, otherwise 0.

The `binexpr()` code in `expr.c` already parses the tokens and builds the A_LOGOR and A_LOGAND AST nodes. So we need to fix up the code generator.

In `genAST()` in `gen.c`, we now have:

```
  case A_LOGOR:
    return (cglogor(leftreg, rightreg));
  case A_LOGAND:
    return (cglogand(leftreg, rightreg));
```

with two corresponding functions in `cg.c`. Before we look at the `cg.c` functions, let's just see an example C expression and the assembly code that will be produced.

```
int x, y, z;
  ...
  z= x || y;
```

when compiled, results in:

```
        movslq  x(%rip), %r10           # Load x's rvalue
        movslq  y(%rip), %r11           # Load y's rvalue
        test    %r10, %r10              # Test x's boolean value
        jne     L13                     # True, jump to L13
        test    %r11, %r11              # Test y's boolean value
        jne     L13                     # True, jump to L13
        movq    $0, %r10                # Neither true, set %r10 to false
        jmp     L14                     # and jump to L14
L13:
        movq    $1, %r10                # Set %r10 to true
L14:
        movl    %r10d, z(%rip)          # Save boolean result to z
```

We test each expression, jump based on the boolean result and either store 0 or 1 into our output register. The assembly for A_LOGAND is similar, except that the conditional jumps are `je` (jump if equal to zero) and the `movq $0` and `movq $1` are swapped around.

So, without further comment, are the new `cg.c` functions:

```
// Logically OR two registers and return a
// register with the result, 1 or 0
int cglogor(int r1, int r2) {
  // Generate two labels
  int Ltrue = genlabel();
  int Lend = genlabel();

  // Test r1 and jump to true label if true
  fprintf(Outfile, "\ttest\t%s, %s\n", reglist[r1], reglist[r1]);
  fprintf(Outfile, "\tjne\tL%d\n", Ltrue);
```

```
  // Test r2 and jump to true label if true
  fprintf(Outfile, "\ttest\t%s, %s\n", reglist[r2], reglist[r2]);
  fprintf(Outfile, "\tjne\tL%d\n", Ltrue);

  // Didn't jump, so result is false
  fprintf(Outfile, "\tmovq\t$0, %s\n", reglist[r1]);
  fprintf(Outfile, "\tjmp\tL%d\n", Lend);

  // Someone jumped to the true label, so result is true
  cglabel(Ltrue);
  fprintf(Outfile, "\tmovq\t$1, %s\n", reglist[r1]);
  cglabel(Lend);
  free_register(r2);
  return(r1);
}
```

```
// Logically AND two registers and return a
// register with the result, 1 or 0
int cglogand(int r1, int r2) {
  // Generate two labels
  int Lfalse = genlabel();
  int Lend = genlabel();

  // Test r1 and jump to false label if not true
  fprintf(Outfile, "\ttest\t%s, %s\n", reglist[r1], reglist[r1]);
  fprintf(Outfile, "\tje\tL%d\n", Lfalse);

  // Test r2 and jump to false label if not true
  fprintf(Outfile, "\ttest\t%s, %s\n", reglist[r2], reglist[r2]);
  fprintf(Outfile, "\tje\tL%d\n", Lfalse);

  // Didn't jump, so result is true
  fprintf(Outfile, "\tmovq\t$1, %s\n", reglist[r1]);
  fprintf(Outfile, "\tjmp\tL%d\n", Lend);

  // Someone jumped to the false label, so result is false
  cglabel(Lfalse);
  fprintf(Outfile, "\tmovq\t$0, %s\n", reglist[r1]);
  cglabel(Lend);
  free_register(r2);
  return(r1);
}
```

The program `tests/input122.c` is the test to confirm that this new functionality works.

Conclusion and What's Next

So that's a few small things fixed up in this part of our journey. What I will do now is step back, rethink the array/pointer design and try to fix this up in the next part of our compiler writing journey. Next step