Part 55: Lazy Evaluation

I decided to move the coverage of fixing `&&` and `||` to here instead of in the previous part of our compiler writing journey as the previous part was already big enough.

So why was our original implementation of `&&` and `||` flawed? C programmers expect that these operators will perform lazy evaluation. In other words, the right-hand operand of `&&` and `||` is evaluated only if the left-hand operand's value is not enough to determine the result.

A common use of lazy evaluation is to see if a pointer is pointing at a specific value, but only if the pointer is actually pointing at something. The `test/input138.c` has an example of this:

```
int *aptr;
...
if (aptr && *aptr == 1)
  printf("aptr points at 1\n");
else
  printf("aptr is NULL or doesn't point at 1\n");
```

We don't want to evaluate both operands to the `&&` operator: if `aptr` is NULL, then the `*aptr == 1` expression will cause a NULL dereference and crash the program.

The Problem

The problem is that our current implementation of `&&` and `||` *does* evaluate both operands. In `genAST()` in `gen.c`:

```
  // Get the left and right sub-tree values
  leftreg = genAST(n->left, NOLABEL, NOLABEL, NOLABEL, n->op);
  rightreg = genAST(n->right, NOLABEL, NOLABEL, NOLABEL, n->op);

  switch (n->op) {
    ...
    case A_LOGOR:
      return (cglogor(leftreg, rightreg));
    case A_LOGAND:
      return (cglogand(leftreg, rightreg));
    ...
  }
```

We have to rewrite this to *not* evaluate both operands. Instead, we have to evaluate the left-hand one first. If it is enough to give the result, we can jump to the code to set the result. If not, now we evaluate the right-hand operand. Again, we jump to the code to set the result. And if we didn't jump, we must have the opposite result.

This is very much like the code generator for the IF statement, but it is different enough that I've written a new code generator in `gen.c`. It gets called *before* we run `genAST()` on the left- and right-hand operands. The code is (in stages):

```
// Generate the code for an
// A_LOGAND or A_LOGOR operation
static int gen_logandor(struct ASTnode *n) {
  // Generate two labels
  int Lfalse = genlabel();
  int Lend = genlabel();
  int reg;

  // Generate the code for the left expression
  // followed by the jump to the false label
  reg= genAST(n->left, NOLABEL, NOLABEL, NOLABEL, 0);
  cgboolean(reg, n->op, Lfalse);
  genfreeregs(NOREG);
```

The left operand is evaluated. Let's assume that we are doing the `&&` operation. If this result is zero, we can jump down to `Lfalse` and set the result to zero (false). Also, once the expression has been evaluated we can free all the registers. This also helps to ease the pressure on register allocation.

```
  // Generate the code for the right expression
  // followed by the jump to the false label
  reg= genAST(n->right, NOLABEL, NOLABEL, NOLABEL, 0);
  cgboolean(reg, n->op, Lfalse);
  genfreeregs(reg);
```

We do exactly the same for the right-hand operand. If it was false, we jump to the `Lfalse` label. If we don't jump, the `&&` result must be true. For `&&`, we now do:

```
  cgloadboolean(reg, 1);
  cgjump(Lend);
  cglabel(Lfalse);
  cgloadboolean(reg, 0);
  cglabel(Lend);
  return(reg);
}
```

The `cgloadboolean()` sets the register to true (if 1 is the argument) or false (if 0 is the argument). For the x86-64 this is 1 and 0, but I've coded it this way in case other architectures have different register values for true and false. The above produces this output for the expression `(aptr && *aptr == 1)`:

```
        movq    aptr(%rip), %r10
        test    %r10, %r10              # Test if aptr is not NULL
        je      L38                     # No, jump to L38
        movq    aptr(%rip), %r10
        movslq  (%r10), %r10            # Get *aptr in %r10
        movq    $1, %r11
        cmpq    %r11, %r10              # Is *aptr == 1?
        sete    %r11b
        movzbq  %r11b, %r11
        test    %r11, %r11
        je      L38                     # No, jump to L38
        movq    $1, %r11                # Both true, true is the result
        jmp     L39                     # Skip the false code
L38:
        movq    $0, %r11                # One or both false, false is the
```

```
result
L39:                                    # Continue on with the rest
```

I haven't given the C code to evaluate the `||` operation. Essentially, we jump if either the left or right is true and set true as the result. If we don't jump, we fall into the code which sets false as the result and which jumps over the true setting code.

Testing the Changes

`test/input138.c` also has code to print out an AND and an OR truth table:

```
  // See if generic AND works
  for (x=0; x <= 1; x++)
    for (y=0; y <= 1; y++) {
      z= x && y;
      printf("%d %d | %d\n", x, y, z);
    }

  // See if generic AND works
  for (x=0; x <= 1; x++)
    for (y=0; y <= 1; y++) {
      z= x || y;
      printf("%d %d | %d\n", x, y, z);
    }
```

and this produces the output (with a space added):

```
0 0 | 0
0 1 | 0
1 0 | 0
1 1 | 1

0 0 | 0
0 1 | 1
1 0 | 1
1 1 | 1
```

Conclusion and What's Next

Now we have lazy evaluation in the compiler for `&&` and `||`, which we definitely need for the compiler to compile itself. In fact, at this point, the only thing the compiler can't parse (in its own source code) is the declaration and use of local arrays. So, guess what...

In the next part of our compiler writing journey, I'll try to work out how to declare and use local arrays. Next step