Part 4: An Actual Compiler

It's about time that I met my promise of actually writing a compiler. So in this part of the journey we are going to replace the interpreter in our program with code that generates x86-64 assembly code.

Revising the Interpreter

Before we do, it will be worthwhile to revisit the interpreter code in `interp.c` :

```
int interpretAST(struct ASTnode *n) {
  int leftval, rightval;

  if (n->left) leftval = interpretAST(n->left);
  if (n->right) rightval = interpretAST(n->right);

  switch (n->op) {
    case A_ADD:      return (leftval + rightval);
    case A_SUBTRACT: return (leftval - rightval);
    case A_MULTIPLY: return (leftval * rightval);
    case A_DIVIDE:   return (leftval / rightval);
    case A_INTLIT:   return (n->intvalue);

    default:
      fprintf(stderr, "Unknown AST operator %d\n", n->op);
      exit(1);
```

```
  }
}
```

The interpretAST() function walks the given AST tree depth-first. It evaluates any left sub-

the `op` value indicates that the node is simply an integer literal, the literal value is return.

The function returns the final value for this tree. And, as it is recursive, it will calculate the final value for a whole tree one sub-sub-tree at a time.

Changing to Assembly Code Generation

We are going to write an assembly code generator which is generic. This is, in turn, going to call out to a set of CPU-specific code generation functions.

Here is the generic assembly code generator in `gen.c`:

```
// Given an AST, generate
// assembly code recursively
static int genAST(struct ASTnode *n) {
  int leftreg, rightreg;

  // Get the left and right sub-tree values
  if (n->left) leftreg = genAST(n->left);
  if (n->right) rightreg = genAST(n->right);

  switch (n->op) {
    case A_ADD:      return (cgadd(leftreg,rightreg));
    case A_SUBTRACT: return (cgsub(leftreg,rightreg));
    case A_MULTIPLY: return (cgmul(leftreg,rightreg));
    case A_DIVIDE:   return (cgdiv(leftreg,rightreg));
    case A_INTLIT:   return (cgload(n->intvalue));

    default:
      fprintf(stderr, "Unknown AST operator %d\n", n->op);
      exit(1);
  }
}
```

Looks familar, huh?! We are doing the same depth-first tree traversal. This time:

- A_INTLIT: load a register with the literal value

- Other operators: perform a maths function on the two registers that hold the left-child's and right-child's value

Instead of passing values, the code in `genAST()` passes around register identifiers. For example `cgload()` loads a value into a register and returns the identity of the register with the loaded value.

`genAST()` itself returns the identity of the register that holds the final value of the tree at this point. That's why the code at the top is getting register identities:

```
    if (n->left) leftreg = genAST(n->left);
    if (n->right) rightreg = genAST(n->right);
```

Calling genAST()

`genAST()` is only going to calculate the value of the expression given to it. We need to print out this final calculation. We're also going to need to wrap the assembly code we generate with some leading code (the *preamble*) and some trailing code (the *postamble*). This is done with the other function in `gen.c` :

```
void generatecode(struct ASTnode *n) {
  int reg;

  cgpreamble();
  reg= genAST(n);
  cgprintint(reg);      // Print the register with the result as an int
  cgpostamble();
}
```

The x86-64 Code Generator

That's the generic code generator out of the road. Now we need to look at the generation of some real assembly code. For now, I'm targetting the x86-64 CPU as this is still one of the most common Linux platforms. So, open up `cg.c` and let's get browsing.

Allocating Registers

Any CPU has a limited number of registers. We will have to allocate a register to hold the integer literal values, plus any calculation that we perform on them. However, once we've used a value, we can often discard the value and hence free up the register holding it. Then we can re-use that register for another value.

There are three functions that deal with register allocation:

- `freeall_registers()` : Set all registers as available
- `alloc_register()` : Allocate a free register
- `free_register()` : Free an allocated register

I'm not going to go through the code as it's straight forward but with some error checking. Right now, if I run out of registers then the program will crash. Later on, I'll deal with the situation when we have run out of free registers.

The code works on generic registers: r0, r1, r2 and r3. There is a table of strings with the actual register names:

```
static char *reglist[4]= { "%r8", "%r9", "%r10", "%r11" };
```

This makes these functions fairly independent of the CPU architecture.

Loading a Register

This is done in `cgload()` : a register is allocated, then a `movq` instruction loads a literal value into the allocated register.

```
// Load an integer literal value into a register.
// Return the number of the register
int cgload(int value) {

  // Get a new register
  int r= alloc_register();

  // Print out the code to initialise it
  fprintf(Outfile, "\tmovq\t$%d, %s\n", value, reglist[r]);
  return(r);
}
```

Adding Two Registers

`cgadd()` takes two register numbers and generates the code to add them together. The result is saved in one of the two registers, and the other one is then freed for future use:

```
// Add two registers together and return
// the number of the register with the result
int cgadd(int r1, int r2) {
  fprintf(Outfile, "\taddq\t%s, %s\n", reglist[r1], reglist[r2]);
  free_register(r1);
```

```
  return(r2);
}
```

Note that addition is *commutative*, so I could have added `r2` to `r1` instead of `r1` to `r2`. The identity of the register with the final value is returned.

Multiplying Two Registers

This is very similar to addition, and again the operation is *commutative*, so any register can be returned:

```
// Multiply two registers together and return
// the number of the register with the result
int cgmul(int r1, int r2) {
  fprintf(Outfile, "\timulq\t%s, %s\n", reglist[r1], reglist[r2]);
  free_register(r1);
  return(r2);
}
```

Subtracting Two Registers

Subtraction is *not* commutative: we have to get the order correct. The second register is subtracted from the first, so we return the first and free the second:

```
// Subtract the second register from the first and
// return the number of the register with the result
int cgsub(int r1, int r2) {
  fprintf(Outfile, "\tsubq\t%s, %s\n", reglist[r2], reglist[r1]);
  free_register(r2);
  return(r1);
}
```

Dividing Two Registers

Division is also not commutative, so the previous notes apply. On the x86-64, it's even more complicated. We need to load `%rax` with the *dividend* from `r1`. This needs to be extended to eight bytes with `cqo`. Then, `idivq` will divide `%rax` with the divisor in `r2`, leaving the *quotient* in `%rax`, so we need to copy it out to either `r1` or `r2`. Then we can free the other register.

```
// Divide the first register by the second and
// return the number of the register with the result
int cgdiv(int r1, int r2) {
```

```
  fprintf(Outfile, "\tmovq\t%s,%%rax\n", reglist[r1]);
  fprintf(Outfile, "\tcqo\n");
  fprintf(Outfile, "\tidivq\t%s\n", reglist[r2]);
  fprintf(Outfile, "\tmovq\t%%rax,%s\n", reglist[r1]);
  free_register(r2);
  return(r1);
}
```

Printing A Register

There isn't an x86-64 instruction to print a register out as a decimal number. To solve this problem, the assembly preamble contains a function called `printint()` that takes a register argument and calls `printf()` to print this out in decimal.

I'm not going to give the code in `cgpreamble()`, but it also contains the beginning code for `main()`, so that we can assemble our output file to get a complete program. The code for `cgpostamble()`, also not given here, simply calls `exit(0)` to end the program.

Here, however, is `cgprintint()`:

```
void cgprintint(int r) {
  fprintf(Outfile, "\tmovq\t%s, %%rdi\n", reglist[r]);
  fprintf(Outfile, "\tcall\tprintint\n");
  free_register(r);
}
```

Linux x86-64 expects the first argument to a function to be in the `%rdi` register, so we move our register into `%rdi` before we `call printint`.

Doing Our First Compile

That's about it for the x86-64 code generator. There is some extra code in `main()` to open out `out.s` as our output file. I've also left the interpreter in the program so we can confirm that our assembly calculates the same answer for the input expression as the interpreter.

Let's make the compiler and run it on `input01`:

```
$ make
cc -o comp1 -g cg.c expr.c gen.c interp.c main.c scan.c tree.c

$ make test
./comp1 input01
15
cc -o out out.s
```

```
./out
15
```

Yes! The first 15 is the interpreter's output. The second 15 is the assembly's output.

Examining the Assembly Output

So, exactly what was the assembly output? Well, here is the input file:

```
2 + 3 * 5 - 8 / 3
```

and here is `out.s` for this input with comments:

```
        .text                           # Preamble code
.LC0:
        .string "%d\n"                  # "%d\n" for printf()
printint:
        pushq   %rbp
        movq    %rsp, %rbp              # Set the frame pointer
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %eax          # Get the printint() argument
        movl    %eax, %esi
        leaq    .LC0(%rip), %rdi        # Get the pointer to "%d\n"
        movl    $0, %eax
        call    printf@PLT              # Call printf()
        nop
        leave                           # and return
        ret

        .globl  main
        .type   main, @function
main:
        pushq   %rbp
        movq    %rsp, %rbp              # Set the frame pointer
                                        # End of preamble code

        movq    $2, %r8                 # %r8 = 2
        movq    $3, %r9                 # %r9 = 3
        movq    $5, %r10                # %r10 = 5
        imulq   %r9, %r10               # %r10 = 3 * 5 = 15
        addq    %r8, %r10               # %r10 = 2 + 15 = 17
                                        # %r8 and %r9 are now free again
        movq    $8, %r8                 # %r8 = 8
        movq    $3, %r9                 # %r9 = 3
        movq    %r8,%rax
        cqo                             # Load dividend %rax with 8
```

```
idivq   %r9                     # Divide by 3
movq    %rax,%r8                # Store quotient in %r8, i.e. 2
subq    %r8, %r10               # %r10 = 17 - 2 = 15
movq    %r10, %rdi              # Copy 15 into %rdi in preparation
call    printint                # to call printint()

movl    $0, %eax                # Postamble: call exit(0)
popq    %rbp
ret
```

Excellent! We now have a legitimate compiler: a program that takes an input in one language and generates a translation of that input in another language.

We still have to then assemble the output down to machine code and link it with the support libraries, but this is something that we can perform manually for now. Later on, we will write some code to do this automatically.

Conclusion and What's Next

Changing from the interpreter to a generic code generator was trivial, but then we had to write some code to generate real assembly output. To do this, we had to think about how to allocate registers: for now, we have a naive solution. We also had to deal with some x86-64 oddities like the `idivq` instruction.

Something I haven't touched on yet is: why bother with generating the AST for an expression? Surely, we could have called `cgadd()` when we hit a '+' token in our Pratt parser, ditto for the other operators. I'm going to leave you to think about this, but I will come back to it in a step or two.

In the next part of our compiler writing journey, we will add some statements to our language, so that it starts to resemble a proper computer language. Next step