rzaharia Updated all readme files to contain links to the next step 2 years ago

296 lines (239 loc) · 9.11 KB

Part 7: Comparison Operators

I was going to add IF statements next, but then I realised that I'd better add some comparison operators first. This turned out to be quite easy, because they are binary operators like the existing ones.

So let's quickly see what the changes are to add the six comparison operators: `==`, `!=`, `<`, `>`, `<=` and `>=`.

Adding New Tokens

We will have six new tokens, so let's add them to `defs.h`:

```
// Token types
enum {
  T_EOF,
  T_PLUS, T_MINUS,
  T_STAR, T_SLASH,
  T_EQ, T_NE,
  T_LT, T_GT, T_LE, T_GE,
  T_INTLIT, T_SEMI, T_ASSIGN, T_IDENT,
  // Keywords
  T_PRINT, T_INT
};
```

I've rearranged the tokens so that the ones with precedence come, in low-to-high precedence order, before the tokens that don't have any precedence.

Scanning The Tokens

Now we have to scan them in. Note that we have to distinguish between `=` and `==`, `<` and `<=`, `>` and `>=`. So we will need to read in an extra character from the input and put it back if we don't need it. Here's the new code in `scan()` from `scan.c`:

```
  case '=':
    if ((c = next()) == '=') {
      t->token = T_EQ;
    } else {
      putback(c);
      t->token = T_ASSIGN;
    }
    break;
  case '!':
    if ((c = next()) == '=') {
      t->token = T_NE;
    } else {
      fatalc("Unrecognised character", c);
    }
    break;
  case '<':
    if ((c = next()) == '=') {
      t->token = T_LE;
    } else {
      putback(c);
      t->token = T_LT;
    }
    break;
  case '>':
    if ((c = next()) == '=') {
      t->token = T_GE;
    } else {
      putback(c);
      t->token = T_GT;
    }
    break;
```

I also changed the name of the `=` token to T_ASSIGN to ensure I didn't get confused between it and the new T_EQ token.

New Expression Code

We can now scan in the six new tokens. So now we have to parse them when they appear in expressions, and also enforce their operator precedence.

By now you would have worked out that:

- I'm building what will become a self-compiling compiler
- in the C language
- using the SubC compiler as a reference.

The implication is that I'm writing a compiler for enough of a subset of C (just as SubC) so that it will compile itself. Therefore, I should use the normal C operator precedence order. This means that the comparison operators have lower precedence than multiply and divide.

I also realised that the switch statement I was using to map tokens to AST node types was only going to get bigger. So I decided to rearrange the AST node types so that there is a 1:1 mapping between them for all the binary operators (in `defs.h`):

```
// AST node types. The first few line up
// with the related tokens
enum {
  A_ADD=1, A_SUBTRACT, A_MULTIPLY, A_DIVIDE,
  A_EQ, A_NE, A_LT, A_GT, A_LE, A_GE,
  A_INTLIT,
  A_IDENT, A_LVIDENT, A_ASSIGN
};
```

Now in `expr.c`, I can simplify the token to AST node conversion and also add in the new tokens' precedence:

```
// Convert a binary operator token into an AST operation.
// We rely on a 1:1 mapping from token to AST operation
static int arithop(int tokentype) {
  if (tokentype > T_EOF && tokentype < T_INTLIT)
    return(tokentype);
  fatald("Syntax error, token", tokentype);
}

// 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_PLUS, T_MINUS
```

```
  20, 20,                       // T_STAR, T_SLASH
  30, 30,                       // T_EQ, T_NE
  40, 40, 40, 40                // T_LT, T_GT, T_LE, T_GE
};
```

That's it for the parsing and operator precedence!

Code Generation

As the six new operators are binary operators, it's easy to modify the generic code generator in `gen.c` to deal with them:

```
case A_EQ:
  return (cgequal(leftreg, rightreg));
case A_NE:
  return (cgnotequal(leftreg, rightreg));
case A_LT:
  return (cglessthan(leftreg, rightreg));
case A_GT:
  return (cggreaterthan(leftreg, rightreg));
case A_LE:
  return (cglessequal(leftreg, rightreg));
case A_GE:
  return (cggreaterequal(leftreg, rightreg));
```

x86-64 Code Generation

Now it gets a bit tricky. In C, the comparison operators return a value. If they evaluate true, their result is 1. If they evaluate false, their result is 0. We need to write x86-64 assembly code to reflect this.

Luckily there are some x86-64 instructions to do this. Unfortunately, there are some issues to deal with along the way. Consider this x86-64 instruction:

```
cmpq %r8,%r9
```

The above `cmpq` instruction performs %r9 - %r8 and sets several status flags including the negative and zero flags. Thus, we can look at the flag combinations to see the result of the comparisons:

Comparison	Operation	Flags If True
%r8 == %r9	%r9 - %r8	Zero
%r8 != %r9	%r9 - %r8	Not Zero
%r8 > %r9	%r9 - %r8	Not Zero, Negative
%r8 < %r9	%r9 - %r8	Not Zero, Not Negative
%r8 >= %r9	%r9 - %r8	Zero or Negative
%r8 <= %r9	%r9 - %r8	Zero or Not Negative

There are six x86-64 instructions which set a register to 1 or 0 based on the two flag values: `sete`, `setne`, `setg`, `setl`, `setge` and `setle` in the order of the above table rows.

The problem is, these instructions only set the lowest byte of a register. If the register already has bits set outside of the lowest byte, they will stay set. So we might set a variable to 1, but if it already has the value 1000 (decimal), then it will now be 1001 which is not what we want.

The solution is to `andq` the register after the `setX` instruction to get rid of the unwanted bits. In `cg.c` there is a general comparison function to do this:

```
// Compare two registers.
static int cgcompare(int r1, int r2, char *how) {
  fprintf(Outfile, "\tcmpq\t%s, %s\n", reglist[r2], reglist[r1]);
  fprintf(Outfile, "\t%s\t%s\n", how, breglist[r2]);
  fprintf(Outfile, "\tandq\t$255,%s\n", reglist[r2]);
  free_register(r1);
  return (r2);
}
```

where `how` is one of the `setX` instructions. Note that we perform

```
    cmpq reglist[r2], reglist[r1]
```

because this is actually `reglist[r1] - reglist[r2]` which is what we really want.

x86-64 Registers

We need to take a short diversion here to discuss the registers in the x86-64 architecture. x86-64 has several 64-bit general purpose registers, but we can also use different register names to access and work on subsections of these registers.

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b

the low 32 bits of this register by using the "*r8d*" register. Similarly, the "*r8w*" register is the low 16 bits and the "*r8b*" register is the low 8 bits of the *r8* register.

In the `cgcompare()` function, the code uses the `reglist[]` array to compare the two 64-bit registers, but then sets a flag in the 8-bit version of the second register by using the names in the `breglist[]` array. The x86-64 architecture only allows the `setX` instructions to operate on the 8-bit register names, thus the need for the `breglist[]` array.

Creating Several Compare Instructions

Now that we have this general function, we can write the six actual comparison functions:

```
int cgequal(int r1, int r2) { return(cgcompare(r1, r2, "sete")); }
int cgnotequal(int r1, int r2) { return(cgcompare(r1, r2, "setne")); }
int cglessthan(int r1, int r2) { return(cgcompare(r1, r2, "setl")); }
int cggreaterthan(int r1, int r2) { return(cgcompare(r1, r2, "setg")); }
```

```
int cglessequal(int r1, int r2) { return(cgcompare(r1, r2, "setle")); }
int cggreaterequal(int r1, int r2) { return(cgcompare(r1, r2, "setge")); }
```

As with the other binary operator functions, one register is freed and the other register returns with the result.

Putting It Into Action

Have a look at the `input04` input file:

```
int x;
x= 7 < 9;  print x;
x= 7 <= 9; print x;
x= 7 != 9; print x;
x= 7 == 7; print x;
x= 7 >= 7; print x;
x= 7 <= 7; print x;
x= 9 > 7;  print x;
x= 9 >= 7; print x;
x= 9 != 7; print x;
```

All of these comparisons are true, so we should print nine 1s out. Do a `make test` to confirm this.

Let's look at the assembly code output by the first comparison:

```
        movq    $7, %r8
        movq    $9, %r9
        cmpq    %r9, %r8        # Perform %r8 - %r9, i.e. 7 - 9
        setl    %r9b            # Set %r9b to 1 if 7 is less than 9
        andq    $255,%r9        # Remove all other bits in %r9
        movq    %r9, x(%rip)    # Save the result in x
        movq    x(%rip), %r8
        movq    %r8, %rdi
        call    printint        # Print x out
```

Yes there is some inefficient assembly code above. We haven't even started to worry about optimised code yet. To quote Donald Knuth:

> **Premature optimization is the root of all evil (or at least most of it) in programming.**

Conclusion and What's Next

That was a nice and easy addition to the compiler. The next part of the journey will be much more complicated.

In the next part of our compiler writing journey, we will add IF statements to the compiler and make use of the comparison operators that we just added. Next step