Part 21: More Operators

In this part of our compiler writing journey, I decided to pick some low-hanging fruit and implement many of the expression operators which are still missing. These include:

- `++` and `--`, both pre-increment/decrement and post--increment/decrement
- unary `-`, `~`, and `!`
- binary `^`, `&`, `|`, `<<` and `>>`

I also implemented the implicit "not zero operator" which treats an expression rvalue as a boolean value for selection and loop statements, e.g.

```
for (str= "Hello"; *str; str++) ...
```

instead of writing

```
for (str= "Hello"; *str != 0; str++) ...
```

Tokens and Scanning

As always, we start off with any new tokens in the language. There are a few this time:

Scanned Input	Token
`||`	T_LOGOR
`&&`	T_LOGAND
`|`	T_OR
`^`	T_XOR
`<<`	T_LSHIFT
`>>`	T_RSHIFT
`++`	T_INC
`--`	T_DEC
`~`	T_INVERT
`!`	T_LOGNOT

Some of these are composed of new single characters, so the scanning of these is easy. For others, we need to distinguish between single characters and pairs of different characters. An example is `<`, `<<` and `<=`. We have already seen how to do the scanning for these in `scan.c`, so I won't give the new code here. Browse through `scan.c` to see the additions.

Adding the Binary Operators to the Parsing

Now we need to parse these operators. Some of these operators are binary operators: `||`, `&&`, `|`, `^`, `<<` and `>>`. We already have a precedence framework in place for binary operators. We can simply add the new operators to the framework.

When I did this, I realised that I had several of the existing operators in with the wrong precedence according to this table of C operator precedence. We also need to align the AST node operations with the set of binary operator tokens. Thus, here are the definitions of the tokens, the AST node types and the operator precedence table from `defs.h` and `expr.c`:

```
// Token types
enum {
  T_EOF,
  // Binary operators
  T_ASSIGN, T_LOGOR, T_LOGAND,
  T_OR, T_XOR, T_AMPER,
  T_EQ, T_NE,
```

```
  T_LT, T_GT, T_LE, T_GE,
  T_LSHIFT, T_RSHIFT,
  T_PLUS, T_MINUS, T_STAR, T_SLASH,

  // Other operators
  T_INC, T_DEC, T_INVERT, T_LOGNOT,
  ...
};

// AST node types. The first few line up
// with the related tokens
enum {
  A_ASSIGN= 1, A_LOGOR, A_LOGAND, A_OR, A_XOR, A_AND,
  A_EQ, A_NE, A_LT, A_GT, A_LE, A_GE, A_LSHIFT, A_RSHIFT,
  A_ADD, A_SUBTRACT, A_MULTIPLY, A_DIVIDE,
  ...
  A_PREINC, A_PREDEC, A_POSTINC, A_POSTDEC,
  A_NEGATE, A_INVERT, A_LOGNOT,
  ...
};

// Operator precedence for each binary token. Must
// match up with the order of tokens in defs.h
static int OpPrec[] = {
  0, 10, 20, 30,                // T_EOF, T_ASSIGN, T_LOGOR, T_LOGAND
  40, 50, 60,                   // T_OR, T_XOR, T_AMPER
  70, 70,                       // T_EQ, T_NE
  80, 80, 80, 80,               // T_LT, T_GT, T_LE, T_GE
  90, 90,                       // T_LSHIFT, T_RSHIFT
  100, 100,                     // T_PLUS, T_MINUS
  110, 110                      // T_STAR, T_SLASH
};
```

New Unary Operators.

Now we get to the parsing of the new unary operators, `++`, `--`, `~` and `!`. All of these are prefix operators (i.e. before an expression), but the `++` and `--` operators can also be postfix operators. Thus, we'll need to parse three prefix and two postfix operators, and perform five different semantic actions for them.

To prepare for this addition of these new operators, I went back and consulted the BNF Grammar for C. As these new operators can't be worked into the existing binary operator framework, we'll need to implement them with new functions in our recursive descent parser. Here are the *relevant* sections from the above grammar, rewritten to use our token names:

```
primary_expression
        : T_IDENT
        | T_INTLIT
        | T_STRLIT
        | '(' expression ')'
        ;

postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
        | postfix_expression '(' expression ')'
        | postfix_expression '++'
        | postfix_expression '--'
        ;

prefix_expression
        : postfix_expression
        | '++' prefix_expression
        | '--' prefix_expression
        | prefix_operator prefix_expression
        ;

prefix_operator
        : '&'
        | '*'
        | '-'
        | '~'
        | '!'
        ;

multiplicative_expression
        : prefix_expression
        | multiplicative_expression '*' prefix_expression
        | multiplicative_expression '/' prefix_expression
        | multiplicative_expression '%' prefix_expression
        ;

        etc.
```

We implement the binary operators in `binexpr()` in `expr.c`, but this calls `prefix()`, just as `multiplicative_expression` in the above BNF grammar refers to `prefix_expression`. We already have a function called `primary()`. Now we need a function, `postfix()` to deal with the postfix expressions.

Prefix Operators

We already parse a couple of tokens in `prefix()`: T_AMPER and T_STAR. We can add in the new tokens here (T_MINUS, T_INVERT, T_LOGNOT, T_INC and T_DEC) by adding more case statements to the `switch (Token.token)` statement.

I won't include the code here because all the cases have a similar structure:

- Skip past the token with `scan(&Token)`
- Parse the next expression with `prefix()`
- Do some semantic checking
- Extend the AST tree that was returned by `prefix()`

However, the differences between some of the cases are important to cover. For the parsing of the `&` (T_AMPER) token, the expression needs to be treated as an lvalue: if we do `&x`, we want the address of the variable `x`, not the address of `x`'s value. Other cases do need to have the AST tree returned by `prefix()` forced to be an rvalue:

- `-` (T_MINUS)
- `~` (T_INVERT)
- `!` (T_LOGNOT)

And, for the pre-increment and pre-decrement operators, we actually *require* the expression to be an lvalue: we can do `++x` but not `++3`. For now, I've written the code to require a simple identifier, but I know later on we will want to parse and deal with `++b[2]` and `++ *ptr`.

Also, from a design point of view, we have the option of altering the AST tree returned by `prefix()` (with no new AST nodes), or adding one or more new AST nodes to the tree:

- T_AMPER modifies the existing AST tree so the root is A_ADDR
- T_STAR adds an A_DEREF node to the root of the tree
- T_STAR adds an A_NEGATE node to the root of the tree after possibly widening the tree to be an `int` value. Why? Because the tree might be of type `char` which is unsigned, and you can't negate an unsigned value.
- T_INVERT adds an A_INVERT node to the root of the tree
- T_LOGNOT adds an A_LOGNOT node to the root of the tree
- T_INC adds an A_PREINC node to the root of the tree
- T_DEC adds an A_PREDEC node to the root of the tree

Parsing the Postfix Operators

If you look at the BNF grammar I hyperlinked to above, to parse a postfix expression we need to refer to the parsing of a primary expression. To implement this, we need to get the tokens of the primary expression first and then then determine if there are any trailing postfix tokens.

Even though the grammar shows "postfix" calling "primary", I've implemented it by scanning the tokens in `primary()` and then deciding to call `postfix()` to parse the postfix tokens.

> This turned out to be a mistake -- Warren, writing from the future.

The BNF grammar above seems to allow expressions like `x++ ++` because it has:

```
postfix_expression:
        postfix_expression '++'
        ;
```

but I'm not going to allow more than one postfix operator after the expression. So let's look at the new code:

`primary()` deals with recognising primary expressions: integer literals, string literals and identifiers. It also recognises parenthesised expressions. Only the identifiers can be followed by postfix operators.

```
static struct ASTnode *primary(void) {
  ...
  switch (Token.token) {
    case T_INTLIT: ...
    case T_STRLIT: ...
    case T_LPAREN: ...
    case T_IDENT:
      return (postfix());
    ...
}
```

I've moved the parsing of function calls and array references out to `postfix()`, and this is where we parse the postfix `++` and `--` operators:

```
// Parse a postfix expression and return
// an AST node representing it. The
// identifier is already in Text.
```

```
static struct ASTnode *postfix(void) {
  struct ASTnode *n;
  int id;

  // Scan in the next token to see if we have a postfix expression
  scan(&Token);

  // Function call
  if (Token.token == T_LPAREN)
    return (funccall());

  // An array reference
  if (Token.token == T_LBRACKET)
    return (array_access());


  // A variable. Check that the variable exists.
  id = findglob(Text);
  if (id == -1 || Gsym[id].stype != S_VARIABLE)
    fatals("Unknown variable", Text);

  switch (Token.token) {
      // Post-increment: skip over the token
    case T_INC:
      scan(&Token);
      n = mkastleaf(A_POSTINC, Gsym[id].type, id);
      break;

      // Post-decrement: skip over the token
    case T_DEC:
      scan(&Token);
      n = mkastleaf(A_POSTDEC, Gsym[id].type, id);
      break;

      // Just a variable reference
    default:
      n = mkastleaf(A_IDENT, Gsym[id].type, id);
  }
  return (n);
}
```

Another design decision. For `++`, we could have made an A_IDENT AST node with an A_POSTINC parent, but given that we have the identifier's name in `Text`, we can build a single AST node that contains both the node type and the reference to the identifier's slot number in the symbol table.

Converting an Integer Expression to a Boolean Value

Before we leave the parsing side of things and move to the code generation side of things, I should mention the change I made to allow integer expressions to be treated as boolean expressions, e.g.

```
x= a + b;
if (x) { printf("x is not zero\n"); }
```

The BNF grammar doesn't provide any explicit syntax rules to restrict expressions to be boolean, e.g:

```
selection_statement
        : IF '(' expression ')' statement
```

Therefore, we'll have to do this semantically. In `stmt.c` where I parse IF, WHILE and FOR loops, I've added this code:

```
// Parse the following expression
// Force a non-comparison expression to be boolean
condAST = binexpr(0);
if (condAST->op < A_EQ || condAST->op > A_GE)
  condAST = mkastunary(A_TOBOOL, condAST->type, condAST, 0);
```

I've introduced a new AST node type, A_TOBOOL. This will generate code to take any integer value. If this value is zero, the result is zero, otherwise the result will be one.

Generating the Code for the New Operators

Now we turn our attention to generating the code for the new operators. Actually, the new AST node types: A_LOGOR, A_LOGAND, A_OR, A_XOR, A_AND, A_LSHIFT, A_RSHIFT, A_PREINC, A_PREDEC, A_POSTINC, A_POSTDEC, A_NEGATE, A_INVERT, A_LOGNOT and A_TOBOOL.

All of these are simple calls out to matching functions in the platform-specific code generator in `cg.c` . So the new code in `genAST()` in `gen.c` is simply:

```
    case A_AND:
      return (cgand(leftreg, rightreg));
    case A_OR:
```

```
      return (cgor(leftreg, rightreg));
    case A_XOR:
      return (cgxor(leftreg, rightreg));
    case A_LSHIFT:
      return (cgshl(leftreg, rightreg));
    case A_RSHIFT:
      return (cgshr(leftreg, rightreg));
    case A_POSTINC:
      // Load the variable's value into a register,
      // then increment it
      return (cgloadglob(n->v.id, n->op));
    case A_POSTDEC:
      // Load the variable's value into a register,
      // then decrement it
      return (cgloadglob(n->v.id, n->op));
    case A_PREINC:
      // Load and increment the variable's value into a register
      return (cgloadglob(n->left->v.id, n->op));
    case A_PREDEC:
      // Load and decrement the variable's value into a register
      return (cgloadglob(n->left->v.id, n->op));
    case A_NEGATE:
      return (cgnegate(leftreg));
    case A_INVERT:
      return (cginvert(leftreg));
    case A_LOGNOT:
      return (cglognot(leftreg));
    case A_TOBOOL:
      // If the parent AST node is an A_IF or A_WHILE, generate
      // a compare followed by a jump. Otherwise, set the register
      // to 0 or 1 based on it's zeroeness or non-zeroeness
      return (cgboolean(leftreg, parentASTop, label));
```

x86-64 Specific Code Generation Functions

That means we can now look at the back-end functions to generate real x86-64 assembly code. For most of the bitwise operations, the x86-64 platform has assembly instructions to do them:

```
int cgand(int r1, int r2) {
  fprintf(Outfile, "\tandq\t%s, %s\n", reglist[r1], reglist[r2]);
  free_register(r1); return (r2);
}

int cgor(int r1, int r2) {
  fprintf(Outfile, "\torq\t%s, %s\n", reglist[r1], reglist[r2]);
```

```
  free_register(r1); return (r2);
}

int cgxor(int r1, int r2) {
  fprintf(Outfile, "\txorq\t%s, %s\n", reglist[r1], reglist[r2]);
  free_register(r1); return (r2);
}

// Negate a register's value
int cgnegate(int r) {
  fprintf(Outfile, "\tnegq\t%s\n", reglist[r]); return (r);
}

// Invert a register's value
int cginvert(int r) {
  fprintf(Outfile, "\tnotq\t%s\n", reglist[r]); return (r);
}
```

With the shift operations, as far as I can tell the shift amount has to be loaded into the `%cl` register first.

```
int cgshl(int r1, int r2) {
  fprintf(Outfile, "\tmovb\t%s, %%cl\n", breglist[r2]);
  fprintf(Outfile, "\tshlq\t%%cl, %s\n", reglist[r1]);
  free_register(r2); return (r1);
}

int cgshr(int r1, int r2) {
  fprintf(Outfile, "\tmovb\t%s, %%cl\n", breglist[r2]);
  fprintf(Outfile, "\tshrq\t%%cl, %s\n", reglist[r1]);
  free_register(r2); return (r1);
}
```

The operations that deal with boolean expressions (where the result must be either 0 or 1) are a bit more complicated.

```
// Logically negate a register's value
int cglognot(int r) {
  fprintf(Outfile, "\ttest\t%s, %s\n", reglist[r], reglist[r]);
  fprintf(Outfile, "\tsete\t%s\n", breglist[r]);
  fprintf(Outfile, "\tmovzbq\t%s, %s\n", breglist[r], reglist[r]);
  return (r);
}
```

The `test` instruction essentially AND's the register with itself to set the zero and negative flags. Then we set the register to 1 if it is equal to zero (`sete`). Then we move this 8-bit result into the 64-bit register proper.

And here is the code to convert an integer into a boolean value:

```
// Convert an integer value to a boolean value. Jump if
// it's an IF or WHILE operation
int cgboolean(int r, int op, int label) {
  fprintf(Outfile, "\ttest\t%s, %s\n", reglist[r], reglist[r]);
  if (op == A_IF || op == A_WHILE)
    fprintf(Outfile, "\tje\tL%d\n", label);
  else {
    fprintf(Outfile, "\tsetnz\t%s\n", breglist[r]);
    fprintf(Outfile, "\tmovzbq\t%s, %s\n", breglist[r], reglist[r]);
  }
  return (r);
}
```

Again, we do a `test` to get the zero-ness or non-zeroeness of the register. If we are doing this for an selection or loop statement, then `je` to jump if the result was false. Otherwise, use `setnz` to set the register to 1 if it was non-zero originally.

Increment and Decrement Operations

I've left the `++` and `--` operations to last. The subtlety here is that we have to both get the value out of the memory location into a register, and separately increment or decrement it. And we have to choose to do this before or after we load the register.

As we already have a `cgloadglob()` function to load a global variable's value, let's modify it to also alter the variable as required. The code is ugly but it does work.

```
// Load a value from a variable into a register.
// Return the number of the register. If the
// operation is pre- or post-increment/decrement,
// also perform this action.
int cgloadglob(int id, int op) {
  // Get a new register
  int r = alloc_register();

  // Print out the code to initialise it
  switch (Gsym[id].type) {
    case P_CHAR:
      if (op == A_PREINC)
```

```
        fprintf(Outfile, "\tincb\t%s(\%%rip)\n", Gsym[id].name);
      if (op == A_PREDEC)
        fprintf(Outfile, "\tdecb\t%s(\%%rip)\n", Gsym[id].name);
      fprintf(Outfile, "\tmovzbq\t%s(%%rip), %s\n", Gsym[id].name, reglist[r])
      if (op == A_POSTINC)
        fprintf(Outfile, "\tincb\t%s(\%%rip)\n", Gsym[id].name);
      if (op == A_POSTDEC)
        fprintf(Outfile, "\tdecb\t%s(\%%rip)\n", Gsym[id].name);
      break;
    case P_INT:
      if (op == A_PREINC)
        fprintf(Outfile, "\tincl\t%s(\%%rip)\n", Gsym[id].name);
      if (op == A_PREDEC)
        fprintf(Outfile, "\tdecl\t%s(\%%rip)\n", Gsym[id].name);
      fprintf(Outfile, "\tmovslq\t%s(\%%rip), %s\n", Gsym[id].name, reglist[r]
      if (op == A_POSTINC)
        fprintf(Outfile, "\tincl\t%s(\%%rip)\n", Gsym[id].name);
      if (op == A_POSTDEC)
        fprintf(Outfile, "\tdecl\t%s(\%%rip)\n", Gsym[id].name);
      break;
    case P_LONG:
    case P_CHARPTR:
    case P_INTPTR:
    case P_LONGPTR:
      if (op == A_PREINC)
        fprintf(Outfile, "\tincq\t%s(\%%rip)\n", Gsym[id].name);
      if (op == A_PREDEC)
        fprintf(Outfile, "\tdecq\t%s(\%%rip)\n", Gsym[id].name);
      fprintf(Outfile, "\tmovq\t%s(\%%rip), %s\n", Gsym[id].name, reglist[r]);
      if (op == A_POSTINC)
        fprintf(Outfile, "\tincq\t%s(\%%rip)\n", Gsym[id].name);
      if (op == A_POSTDEC)
        fprintf(Outfile, "\tdecq\t%s(\%%rip)\n", Gsym[id].name);
      break;
    default:
      fatald("Bad type in cgloadglob:", Gsym[id].type);
  }
  return (r);
}
```

I'm pretty sure that I'll have to rewrite this later on to perform `x= b[5]++`, but this will do for now. After all, baby steps is what I promised for each step of our journey.

Testing the New Functionality

I won't go through the new test input files in detail for this step. They are `input22.c`, `input23.c` and `input24.c` in the `tests` directory. You can browse them and confirm that the compiler can correctly compile them:

```
$ make test
...
input22.c: OK
input23.c: OK
input24.c: OK
```

Conclusion and What's Next

In terms of extending the functionality of our compiler, this part of the journey added a lot of functionality, but I hope the amount of additional conceptual complexity was minimal.

We added a bunch of binary operators and this was done by updating the scanner and changing the operator precedence table.

For the unary operators, we added them manually to the parser in the `prefix()` function.

For the new postfix operators, we separated the old function call and array index functionality out into a new `postfix()` function, and used this to add in the postfix operators. We did have to worry a bit about lvalues and rvalues here. We also had some design decisions about what AST nodes to add, or if we should just redecorate some existing AST nodes.

The code generation ended up being relatively simple because the x86-64 architecture has instructions to implement the operations we needed. However, we did have to set up some specific registers for some of the operations, or perform instruction combinations to do what we wanted.

The tricky operations were the increment and decrement operations. I've put code in to get these to work for ordinary variables but we will have to revisit this later.

In the next part of our compiler writing journey, I'd like to tackle local variables. Once we can get these to work, we can extend them to also include function parameters and arguments. This will take two or more steps. Next step