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

515 lines (402 loc) · 13.5 KB

Part 6: Variables

I've just finished adding global variables to the compiler and, as I suspected, it was a lot of work. Also, pretty much every file in the compiler got modified in the process. So this part of the journey is going to be long.

What Do We Want from Variables?

We want to be able to:

- Declare variables
- Use variables to get stored values
- Assign to variables

Here is `input02` which will be our test program:

```
int fred;
int jim;
fred= 5;
jim= 12;
print fred + jim;
```

The most obvious change is that the grammar now has variable declarations, assignment statements and variables names in expressions. However, before we get to that, let's look at how we implement variables.

The Symbol Table

Every compiler is going to need a symbol table. Later on, we will hold more than just global variables. But for now, here is the structure of an entry in the table (from `defs.h`):

```
// Symbol table structure
struct symtable {
  char *name;                   // Name of a symbol
};
```

We have an array of symbols in `data.h`:

```
#define NSYMBOLS        1024            // Number of symbol table entries
extern_ struct symtable Gsym[NSYMBOLS]; // Global symbol table
static int Globs = 0;                   // Position of next free global symbol
```

`Globs` is actually in `sym.c`, the file that manages the symbol table. In here we have these management functions:

- `int findglob(char *s)`: Determine if the symbol s is in the global symbol table. Return its slot position or -1 if not found.
- `static int newglob(void)`: Get the position of a new global symbol slot, or die if we've run out of positions.
- `int addglob(char *name)`: Add a global symbol to the symbol table. Return the slot number in the symbol table.

The code is fairly straight forward, so I won't bother to give the code here in the discussion. With these functions, we can find symbols and add new symbols to the symbol table.

Scanning and New Tokens

If you look at the example input file, we need a few new tokens:

- 'int', known as T_INT
- '=', known as T_EQUALS
- identifier names, known as T_IDENT

The scanning of '=' is easy to add to `scan()`:

```
    case '=':
      t->token = T_EQUALS; break;
```

We can add the 'int' keyword to `keyword()`:

```
    case 'i':
      if (!strcmp(s, "int"))
        return (T_INT);
      break;
```

For identifiers, we are already using `scanident()` to store words into the `Text` variable. Instead of dying if a word is not a keyword, we can return a T_IDENT token:

```
      if (isalpha(c) || '_' == c) {
        // Read in a keyword or identifier
        scanident(c, Text, TEXTLEN);

        // If it's a recognised keyword, return that token
        if (tokentype = keyword(Text)) {
          t->token = tokentype;
          break;
        }
        // Not a recognised keyword, so it must be an identifier
        t->token = T_IDENT;
        break;
      }
```

The New Grammar

We're about ready to look at the changes to the grammar of our input language. As before, I'll define it with BNF notation:

```
 statements: statement
      |      statement statements
      ;

 statement: 'print' expression ';'
      |     'int'   identifier ';'
      |     identifier '=' expression ';'
      ;
 identifier: T_IDENT
      ;
```

An identifier is returned as a T_IDENT token, and we already have the code to parse print statements. But, as we now have three types of statements, it makes sense to write a function to deal with each one. Our top-level `statements()` function in `stmt.c` now looks like:

```
// Parse one or more statements
void statements(void) {

  while (1) {
    switch (Token.token) {
    case T_PRINT:
      print_statement();
      break;
    case T_INT:
      var_declaration();
      break;
    case T_IDENT:
      assignment_statement();
      break;
    case T_EOF:
      return;
    default:
      fatald("Syntax error, token", Token.token);
    }
  }
}
```

I've moved the old print statement code into `print_statement()` and you can browse that yourself.

Variable Declarations

Let's look at variable declarations. This is in a new file, `decl.c`, as we are going to have lots of other types of declarations in the future.

```
// Parse the declaration of a variable
void var_declaration(void) {

  // Ensure we have an 'int' token followed by an identifier
  // and a semicolon. Text now has the identifier's name.
  // Add it as a known identifier
```

```
    match(T_INT, "int");
    ident();
    addglob(Text);
    genglobsym(Text);
    semi();
  }
```

The `ident()` and `semi()` functions are wrappers around `match()`:

```
void semi(void)  { match(T_SEMI, ";"); }
void ident(void) { match(T_IDENT, "identifier"); }
```

Back to `var_declaration()`, once we have scanned in the idenfiier into the `Text` buffer, we can add this to the global symbol table with `addglob(Text)`. The code in there allows a variable to be declared multiple times (for now).

Assignment Statements

Here's the code for `assignment_statement()` in `stmt.c`:

```
void assignment_statement(void) {
  struct ASTnode *left, *right, *tree;
  int id;

  // Ensure we have an identifier
  ident();

  // Check it's been defined then make a leaf node for it
  if ((id = findglob(Text)) == -1) {
    fatals("Undeclared variable", Text);
  }
  right = mkastleaf(A_LVIDENT, id);

  // Ensure we have an equals sign
  match(T_EQUALS, "=");

  // Parse the following expression
  left = binexpr(0);

  // Make an assignment AST tree
  tree = mkastnode(A_ASSIGN, left, right, 0);

  // Generate the assembly code for the assignment
  genAST(tree, -1);
  genfreeregs();
```

```
    // Match the following semicolon
    semi();
  }
```

We have a couple of new AST node types. A_ASSIGN takes the expression in the left-hand child and assigns it to the right-hand child. And the right-hand child will be an A_LVIDENT node.

Why did I call this node *A_LVIDENT*? Because it represents an *lvalue* identifier. So what's an lvalue?

An lvalue is a value that is tied to a specific location. Here, it's the address in memory which holds a variable's value. When we do:

```
     area = width * height;
```

we *assign* the result of the right-hand side (i.e. the *rvalue*) to the variable in the left-hand side (i.e. the *lvalue*). The *rvalue* isn't tied to a specific location. Here, the expression result is probably in some arbitrary register.

Also note that, although the assignment statement has the syntax

```
    identifier '=' expression ';'
```

we will make the expression the left sub-tree of the A_ASSIGN node and save the A_LVIDENT details in the right sub-tree. Why? Because we need to evaluate the expression *before* we save it into the variable.

Changes to the AST Structure

We now need to store either an integer literal value in A_INTLIT AST nodes, or the details of the symbol for A_IDENT AST nodes. I've added a *union* to the AST structure to do this (in `defs.h`):

```
  // Abstract Syntax Tree structure
  struct ASTnode {
    int op;                             // "Operation" to be performed on this tree
    struct ASTnode *left;               // Left and right child trees
    struct ASTnode *right;
    union {
```

```
    int intvalue;               // For A_INTLIT, the integer value
    int id;                     // For A_IDENT, the symbol slot number
  } v;
};
```

Generating the Assignment Code

Let's now look at the changes to `genAST()` in `gen.c`

```
int genAST(struct ASTnode *n, int reg) {
  int leftreg, rightreg;

  // Get the left and right sub-tree values
  if (n->left)
    leftreg = genAST(n->left, -1);
  if (n->right)
    rightreg = genAST(n->right, leftreg);

  switch (n->op) {
  ...
    case A_INTLIT:
    return (cgloadint(n->v.intvalue));
  case A_IDENT:
    return (cgloadglob(Gsym[n->v.id].name));
  case A_LVIDENT:
    return (cgstorglob(reg, Gsym[n->v.id].name));
  case A_ASSIGN:
    // The work has already been done, return the result
    return (rightreg);
  default:
    fatald("Unknown AST operator", n->op);
  }
```

Note that we evaluate the left-hand AST child first, and we get back a register number that holds the left-hand sub-tree's value. We now pass this register number to the right-hand sub-tree. We need to do this for A_LVIDENT nodes, so that the `cgstorglob()` function in `cg.c` knows which register holds the rvalue result of the assignment expression.

So, consider this AST tree:

```
           A_ASSIGN
          /        \
     A_INTLIT   A_LVIDENT
        (3)        (5)
```

We call `leftreg = genAST(n->left, -1);` to evaluate the A_INTLIT operation. This will `return (cgloadint(n->v.intvalue));` , i.e. load a register with the value 3 and return the register id.

Then, we call `rightreg = genAST(n->right, leftreg);` to evaluate the A_LVIDENT operation. This will `return (cgstorglob(reg, Gsym[n->v.id].name));` , i.e. store the register into the variable whose name is in `Gsym[5]` .

Then we switch to the A_ASSIGN case. Well, all our work has already been done. The rvalue is still in a register, so let's leave it there and return it. Later, we'll be able to do expressions like:

```
  a= b= c = 0;
```

where an assignment is not just a statement but also an expression.

Generating x86-64 Code

You would have noticed that I changed the name of the old `cgload()` function to `cgloadint()` . This is more specific. We now have a function to load the value out of a global variable (in `cg.c`):

```
int cgloadglob(char *identifier) {
  // Get a new register
  int r = alloc_register();

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

Similarly, we need a function to save a register into a variable:

```
// Store a register's value into a variable
int cgstorglob(int r, char *identifier) {
  fprintf(Outfile, "\tmovq\t%s, %s(\%%rip)\n", reglist[r], identifier);
  return (r);
}
```

We also need a function to create a new global integer variable:

```
// Generate a global symbol
void cgglobsym(char *sym) {
  fprintf(Outfile, "\t.comm\t%s,8,8\n", sym);
}
```

Of course, we can't let the parser access this code directly. Instead, there is a function in the generic code generator in `gen.c` that acts as the interface:

```
void genglobsym(char *s) { cgglobsym(s); }
```

Variables in Expressions

So now we can assign to variables. But how do we get a variable's value into an expression. Well, we already have a `primary()` function to get an integer literal. Let's modify it to also load a variable's value:

```
// Parse a primary factor and return an
// AST node representing it.
static struct ASTnode *primary(void) {
  struct ASTnode *n;
  int id;

  switch (Token.token) {
  case T_INTLIT:
    // For an INTLIT token, make a leaf AST node for it.
    n = mkastleaf(A_INTLIT, Token.intvalue);
    break;

  case T_IDENT:
    // Check that this identifier exists
    id = findglob(Text);
    if (id == -1)
      fatals("Unknown variable", Text);

    // Make a leaf AST node for it
    n = mkastleaf(A_IDENT, id);
    break;

  default:
    fatald("Syntax error, token", Token.token);
  }

  // Scan in the next token and return the leaf node
  scan(&Token);
```

```
    return (n);
  }
```

Note the syntax checking in the T_IDENT case to ensure the variable has been declared before we try to use it.

Trying It Out

I think that's about it for variable declarations, so let's try it out with the `input02` file:

```
int fred;
int jim;
fred= 5;
jim= 12;
print fred + jim;
```

We can `make test` to do this:

```
$ make test
cc -o comp1 -g cg.c decl.c expr.c gen.c main.c misc.c scan.c
               stmt.c sym.c tree.c
...
./comp1 input02
cc -o out out.s
./out
17
```

As you can see, we calculated `fred + jim` which is 5 + 12 or 17. Here are the new assembly lines in `out.s` :

```
        .comm   fred,8,8                # Declare fred
        .comm   jim,8,8                 # Declare jim
        ...
        movq    $5, %r8
        movq    %r8, fred(%rip)         # fred = 5
        movq    $12, %r8
        movq    %r8, jim(%rip)          # jim = 12
        movq    fred(%rip), %r8
        movq    jim(%rip), %r9
        addq    %r8, %r9                # fred + jim
```

Other Changes

I've probably made a few other changes. The only main one that I can remember is to create some helper functions in `misc.c` to make it easier to report fatal errors:

```
// Print out fatal messages
void fatal(char *s) {
  fprintf(stderr, "%s on line %d\n", s, Line); exit(1);
}

void fatals(char *s1, char *s2) {
  fprintf(stderr, "%s:%s on line %d\n", s1, s2, Line); exit(1);
}

void fatald(char *s, int d) {
  fprintf(stderr, "%s:%d on line %d\n", s, d, Line); exit(1);
}

void fatalc(char *s, int c) {
  fprintf(stderr, "%s:%c on line %d\n", s, c, Line); exit(1);
}
```

Conclusion and What's Next

So that was a lot of work. We had to write the beginnings of symbol table management. We had to deal with two new statement types. We had to add some new tokens and some new AST node types. Finally, we had to add some code to generate the correct x86-64 assembly output.

Try writing a few example input files and see if the compiler works as it should, especially if it detects syntax errors and semantic errors (variable use without a declaration).

In the next part of our compiler writing journey, we will add the six comparison operators to our language. That will allow us to start on the control structures in the part after that.
Next step