Part 18: Lvalues and Rvalues Revisited

As this is work in progress with no design document to guide me, occasionally I need to remove code that I've already written and rewrite it to make it more general, or to fix shortcomings. That's the case for this part of the journey.

We added our initial support for pointers in part 15 so that we could write code line this:

```
int  x;
int *y;
int  z;
x= 12; y= &x; z= *y;
```

That's all fine and good, but I knew that we would eventually have to support the use of pointers on the left-hand side of assignment statements, e.g.

```
*y = 14;
```

To do this, we have to revisit the topic of lvalues and rvalues. To revise, an *lvalue* is a value that is tied to a specific location, whereas an *rvalue* is a value that isn't. Lvalues are persistent in that we can retrieve their value in future instructions. Rvalues, on the other hand, are evanescent: we can discard them once their use is finished.

Examples of Rvalues and Lvalues

An example of an rvalue is an integer literal, e.g. 23. We can use it in an expression and then discard it afterwards. Examples of lvalues are locations in memory which we can *store into*, such as:

```
a            Scalar variable a
b[0]         Element zero of array b
*c           The location that pointer c points to
(*d)[0]      Element zero of the array that d points to
```

As I mentioned before, the names *lvalue* and *rvalue* come from the two sides of an assignment statement: lvalues are on the left, rvalues are on the right.

Extending Our Notion of Lvalues

Right now, the compiler treats nearly everything as an rvalue. For variables, it retrieves the value from the variable's location. Our only nod to the concept of the lvalue is to mark identifiers on the left of an assignment as an A_LVIDENT. We manually deal with this in `genAST()`:

```
case A_IDENT:
  return (cgloadglob(n->v.id));
case A_LVIDENT:
  return (cgstorglob(reg, n->v.id));
case A_ASSIGN:
  // The work has already been done, return the result
  return (rightreg);
```

which we use for statements like `a= b;`. But now we need to mark more than just identifiers on the left-hand side of an assignment as lvalues.

It's also important to make it easy to generate assembly code in the process. While I was writing this part, I tried the idea of prepending a "A_LVALUE" AST node as the parent to a tree, to tell the code generator to output the lvalue version of the code for it instead of the rvalue version. But this turned out to be too late: the sub-tree was already evaluated and rvalue code for it had already been generated.

Yet Another AST Node Change

I'm loath to keep adding more fields to the AST node, but this is what I ended up doing. We now have a field to indicate if the node should generate lvalue code or rvalue code:

```
// Abstract Syntax Tree structure
struct ASTnode {
  int op;                         // "Operation" to be performed on this tree
  int type;                       // Type of any expression this tree generates
  int rvalue;                     // True if the node is an rvalue
  ...
};
```

The `rvalue` field only holds one bit of information; later on, if I need to store other booleans, I will be able to use this as a bitfield.

Question: why did I make the field indicate the "rvalue"ness of the node and not the "lvalue"ness? After all, most of the nodes in our AST trees will hold rvalues and not lvalues. While I was reading Nils Holm's book on SubC, I read this line:

> Since an indirection cannot be reversed later, the parser assumes each partial expression to be an lvalue.

Consider the parser working on the statement `b = a + 2`. After parsing the `b` identifier, we cannot yet tell is this is an lvalue or an rvalue. It's not until we hit the `=` token that we can conclude that it's an lvalue.

Also, the C language allows assignments as expressions, so we can also write `b = c = a + 2`. Again, when we parse the `a` identifier, we can't tell if it's an lvalue or an rvalue until we parse the next token.

Therefore, I chose to assume each AST node to be an lvalue by default. Once we can definitely tell if a node is rvalue, we can then set the `rvalue` field to indicate this.

Assignment Expressions

I also mentioned that the C language allows assignments as expressions. Now that we have a clear lvalue/rvalue distinction, we can shift the parsing of assignments as statements and move the code into the expression parser. I'll cover this later.

It's now time to see what was done to the compiler code base to make this all happen. As always, we start with the tokens and the scanner first.

Token and Scanning Changes

We have no new tokens or new keywords this time. But there is a change which affects the token code. The `=` is now a binary operator with an expression on each side, so we need to integrate it with the other binary operators.

According to this list of C operators, the `=` operator has much lower precedence than `+` or `-`. We there need to rearrange our list of operators and their precedences. In `defs.h`:

```
// Token types
enum {
  T_EOF,
  // Operators
  T_ASSIGN,
  T_PLUS, T_MINUS, ...
```

In `expr.c`, we need to update the code that holds the precedences for our binary operators:

```
// Operator precedence for each token. Must
// match up with the order of tokens in defs.h
static int OpPrec[] = {
   0, 10,                       // T_EOF,  T_ASSIGN
  20, 20,                       // T_PLUS, T_MINUS
  30, 30,                       // T_STAR, T_SLASH
  40, 40,                       // T_EQ, T_NE
  50, 50, 50, 50                // T_LT, T_GT, T_LE, T_GE
};
```

Changes to the Parser

Now we have to remove the parsing of assignments as statements and make them into expressions. I also took the liberty of removing the "print" statement from the language, as we can now call `printint()`. So, in `stmt.c`, I've removed both `print_statement()` and `assignment_statement()`.

> I also removed the T_PRINT and 'print' keywords from the language. And now that our concept of lvalues and rvalues are different, I also removed the A_LVIDENT AST node type.

For now, the statement parser in `single_statement()` in `stmt.c` assumes that what's coming up next is an expression if it doesn't recognise the first token:

```
static struct ASTnode *single_statement(void) {
  int type;

  switch (Token.token) {
    ...
    default:
    // For now, see if this is an expression.
    // This catches assignment statements.
    return (binexpr(0));
  }
}
```

This does mean that `2+3;` will be treated as a legal statement for now. We will fix this later. And in `compound_statement()` we also ensure that the expression is followed by a semicolon:

```
    // Some statements must be followed by a semicolon
    if (tree != NULL && (tree->op == A_ASSIGN ||
                         tree->op == A_RETURN || tree->op == A_FUNCCALL))
      semi();
```

Expression Parsing

You might think that, now that `=` is marked as a binary expression operator and we have set its precedence, that we are all done. Not so! There are two things we have to worry about:

1. We need to generate the assembly code for the right-hand rvalue before the code for the left-hand lvalue. We used to do this in the statement parser, and we'll have to do this in the expression parser.
2. Assignment expressions are *right associative*: the operator binds more tightly to the expression on the right than to the left.

We haven't touched right associativity before. Let's look at an example. Consider the expression `2 + 3 + 4`. We can happily parse this from left to right and build the AST tree:

```
      +
     / \
    +   4
   / \
  2   3
```

For the expression `a= b= 3`, if we do the above, we end up with the tree:

```
      =
     / \
    =   3
   / \
  a   b
```

We don't want to do `a= b` before then trying to assign the 3 to this left sub-tree. Instead, what we want to generate is this tree:

```
        =
       / \
      =   a
     / \
    3   b
```

I've reversed the leaf nodes to be in assembly output order. We first store 3 in `b`. Then the result of this assignment, 3, is stored in `a`.

Modifying the Pratt Parser

We are using a Pratt parser to correctly parse the precedences of our binary operators. I did a search to find out how to add right-associativity to a Pratt parser, and found this information in Wikipedia:

```
   while lookahead is a binary operator whose precedence is greater than
op's,
   or a right-associative operator whose precedence is equal to op's
```

So, for right-associative operators, we test if the next operator has the same precedence as the operator we are up to. That's a simple modification to the parser's logic. I've introduced a new function in `expr.c` to determine if an operator is right-associative:

```
// Return true if a token is right-associative,
// false otherwise.
static int rightassoc(int tokentype) {
  if (tokentype == T_ASSIGN)
    return(1);
  return(0);
}
```

In `binexpr()` we alter the while loop as mentioned before, and we also put in A_ASSIGN-specific code to swap the child trees around:

```
struct ASTnode *binexpr(int ptp) {
  struct ASTnode *left, *right;
  struct ASTnode *ltemp, *rtemp;
  int ASTop;
  int tokentype;

  // Get the tree on the left.
  left = prefix();
  ...

  // While the precedence of this token is more than that of the
  // previous token precedence, or it's right associative and
  // equal to the previous token's precedence
  while ((op_precedence(tokentype) > ptp) ||
         (rightassoc(tokentype) && op_precedence(tokentype) == ptp)) {
    ...
    // Recursively call binexpr() with the
    // precedence of our token to build a sub-tree
    right = binexpr(OpPrec[tokentype]);

    ASTop = binastop(tokentype);
    if (ASTop == A_ASSIGN) {
      // Assignment
      // Make the right tree into an rvalue
      right->rvalue= 1;
      ...

      // Switch left and right around, so that the right expression's
      // code will be generated before the left expression
      ltemp= left; left= right; right= ltemp;
    } else {
      // We are not doing an assignment, so both trees should be rvalues
      left->rvalue= 1;
      right->rvalue= 1;
    }
    ...
  }
  ...
}
```

Notice also the code to explicitly mark the right-hand side of the assignment expression as an rvalue. And, for non assignments, both sides of the expression get marked as rvalues.

Scattered through `binexpr()` are a few more lines of code to explicitly set a tree to be an rvalue. These get performed when we hit a leaf node. For example the `a` identifier in `b= a;` needs to be marked as an rvalue, but we will never enter the body of the while loop to do this.

Printing Out the Tree

That is the parser changes out of the road. We now have several nodes marked as rvalues, and some not marked at all. At this point, I realised that I was having trouble visualising the AST trees that get generated. I've written a function called `dumpAST()` in `tree.c` to print out each AST tree to standard output. It's not sophisticated. The compiler now has a `-T` command line argument which sets an internal flag, `O_dumpAST`. And the `global_declarations()` code in `decl.c` now does:

```
      // Parse a function declaration and
      // generate the assembly code for it
      tree = function_declaration(type);
      if (O_dumpAST) {
        dumpAST(tree, NOLABEL, 0);
        fprintf(stdout, "\n\n");
      }
      genAST(tree, NOLABEL, 0);
```

The tree dumper code prints out each node in the order tree traversal order, so the output isn't tree shaped. However, the indentation of each node indicates its depth in the tree.

Let's take a look at some example AST trees for assignment expressions. We'll start with `a= b= 34;`:

```
      A_INTLIT 34
    A_WIDEN
    A_IDENT b
  A_ASSIGN
  A_IDENT a
A_ASSIGN
```

The 34 is small enough to be a char-sized literal, but it gets widened to match the type of `b`. `A_IDENT b` doesn't say "rvalue", so it's a lvalue. The value of 34 is stored in the `b` lvalue. This value is then stored in the `a` lvalue.

Now let's try `a= b + 34;`:

```
      A_IDENT rval b
        A_INTLIT 34
      A_WIDEN
    A_ADD
    A_IDENT a
  A_ASSIGN
```

You can see the "rval `b`" now, so `b`'s value is loaded into a register, whereas the result of the `b+34` expression is stored in the `a` lvalue.

Let's do one more, `*x= *y`:

```
      A_IDENT y
    A_DEREF rval
      A_IDENT x
    A_DEREF
  A_ASSIGN
```

The identifier `y` is dereferenced and this rvalue is loaded. This is then stored in the lvalue which is `x` dereferenced.

Converting The Above into Code

Now that the lvalue and rvalues nodes are clearly identified, we can turn our attention to how we translate each into assembly code. There are many nodes like integer literals, addition etc. which are clearly rvalues. It is only the AST node types which could possibly be lvalues that the code in `genAST()` in `gen.c` needs to worry about. Here is what I have for these node types:

```
    case A_IDENT:
      // Load our value if we are an rvalue
      // or we are being dereferenced
      if (n->rvalue || parentASTop== A_DEREF)
        return (cgloadglob(n->v.id));
      else
        return (NOREG);

    case A_ASSIGN:
      // Are we assigning to an identifier or through a pointer?
      switch (n->right->op) {
        case A_IDENT: return (cgstorglob(leftreg, n->right->v.id));
        case A_DEREF: return (cgstorderef(leftreg, rightreg, n->right->type));
        default: fatald("Can't A_ASSIGN in genAST(), op", n->op);
      }

    case A_DEREF:
      // If we are an rvalue, dereference to get the value we point at
      // otherwise leave it for A_ASSIGN to store through the pointer
      if (n->rvalue)
        return (cgderef(leftreg, n->left->type));
      else
        return (leftreg);
```

Changes to the x86-64 Code Generator

The only change to `cg.c` is a function which allows us to store a value through a pointer:

```
// Store through a dereferenced pointer
int cgstorderef(int r1, int r2, int type) {
  switch (type) {
    case P_CHAR:
      fprintf(Outfile, "\tmovb\t%s, (%s)\n", breglist[r1], reglist[r2]);
      break;
    case P_INT:
      fprintf(Outfile, "\tmovq\t%s, (%s)\n", reglist[r1], reglist[r2]);
      break;
    case P_LONG:
      fprintf(Outfile, "\tmovq\t%s, (%s)\n", reglist[r1], reglist[r2]);
      break;
    default:
      fatald("Can't cgstoderef on type:", type);
  }
  return (r1);
}
```

which is nearly exactly the opposite of `cgderef()` which appears immediately before this new function.

Conclusion and What's Next

For this part of the journey, I think I took two or three different design directions, tried them, hit a dead end and backed out before I reached the solution described here. I know that, in SubC, Nils passes a single "lvalue" structure which holds the "lvalue"-ness of the node of the AST tree being processed at any point in time. But his tree only holds one expression; the AST tree for this compiler holds one whole function's worth of nodes. And I'm sure that, if you looked in three other compilers, you would probably find three other solutions too.

There are many things that we could take on next. There are a bunch of C operators that would be relatively easy to add to the compiler. We have A_SCALE, so we could attempt structures. As yet, there are no local variables, which will need attending to at some point. And, we should generalise functions to have multiple arguments and the ability to access them.

In the next part of our compiler writing journey, I'd like to tackle arrays. This will be a combination of dereferencing, lvalues and rvalues, and scaling the array indices by the size of the array's elements. We have all the semantic components in place, but we'll need to add tokens, parsing and the actual index functionality. It should be an interesting topic like this one was. Next step