Part 44: Constant Folding

In the last part of our compiler writing journey, I realised that I'd have to add constant folding as an optimisation, so that I could parse expressions as part of doing global variable declarations.

So, in this part, I've added the constant folding optimisation for general expressions and in the next part I'll rewrite the code for global variable declarations.

What is Constant Folding?

Constant folding is a form of optimisation where an expression can be evaluated by the compiler at compile time, instead of generating code to evaluate the expression at run time.

For example, we can see that `x= 5 + 4 * 5;` is really the same as `x= 25;`, so we can let the compiler evaluate the expression and just output the assembly code for `x= 25;`.

So How Do We Do It?

The answer is: look for sub-trees in an AST tree where the leaves are integer literals. If there is a binary operation which has two integer literals leaves, the compiler can evaluate the expression and replace the sub-tree with a single integer literal node.

Similarly, if there is a unary operation with an integer literal leaf child, then the compiler can also evaluate the expression and replace the sub-tree with a single integer literal node.

Once we can do this for sub-trees, we can write a function to traverse the entire tree looking for sub-trees to fold. At any node, we can do this algorithm:

1. Try to fold and replace the left child, i.e. recursively.
2. Try to fold and replace the right child, i.e. recursively.
3. If it's a binary operation with two literals child leaves, fold that.
4. If it's a unary operation with one literal child leaf, fold that.

The fact that we replace the sub-trees means we recursively optimise the edges of the tree first, then work back up the tree to the root of the tree. An example:

```
   *         *         *     50
  / \       / \       / \
 +   -     10  -     10  5
/ \ / \       / \
6 4 8 3      8   3
```

A New File, `opt.c`

I've created a new source file for our compiler, `opt.c` and in it I've rewritten the same three functions, `fold2()`, `fold1()` and `fold()` that are in the SubC compiler written by Nils M Holm.

One thing that Nils spends a lot of time in his code is to get the calculations correct. This is important when the compiler is a cross-compiler. For example, if we do the constant folding on a 64-bit machine, then the range we have for integer literals is much bigger than for 32-bit machines. Any constant folding we do on the 64-bit machine may not be the same result (due to lack of truncation) than the calculation of the same expression on a 32-bit machine.

I know that this is an important concern, but I will stick with our "KISS principle" and write simple code for now. As required, I'll go back and fix it.

Folding Binary Operations

Here is the code to fold AST sub-trees which are binary operations on two children. I'm only folding a few operations; there are many more in `expr.c` that we could also fold.

```
// Fold an AST tree with a binary operator
// and two A_INTLIT children. Return either
// the original tree or a new leaf node.
static struct ASTnode *fold2(struct ASTnode *n) {
  int val, leftval, rightval;

  // Get the values from each child
  leftval = n->left->a_intvalue;
  rightval = n->right->a_intvalue;
```

Another function will call `fold2()` and this ensures that both `n->left` and `n->right` are non-NULL pointers to A_INTLIT leaf nodes. Now that we have the values from both children, we can get to work.

```
  // Perform some of the binary operations.
  // For any AST op we can't do, return
  // the original tree.
  switch (n->op) {
    case A_ADD:
      val = leftval + rightval;
      break;
    case A_SUBTRACT:
      val = leftval - rightval;
      break;
    case A_MULTIPLY:
      val = leftval * rightval;
      break;
    case A_DIVIDE:
      // Don't try to divide by zero.
      if (rightval == 0)
        return (n);
      val = leftval / rightval;
      break;
    default:
      return (n);
  }
```

We fold the normal four maths operations. Note the special code for division: if we try to divide by zero, the compiler will crash. Instead, we leave the sub-tree intact and let the code crash once it becomes an executable! Obviously, there is opportunity for a `fatal()` call here. We leave the switch statement with a single value `val` that represents the calculated value of the sub-tree. Time to replace the sub-tree.

```
  // Return a leaf node with the new value
  return (mkastleaf(A_INTLIT, n->type, NULL, val));
}
```

So a binary AST tree goes in, and a leaf AST node (hopefully) comes out.

Folding Unary Operations

Now that you've seen folding on binary operations, the code for unary operations should be straight forward. I am only folding two unary operations, but there is room to add more.

```
// Fold an AST tree with a unary operator
// and one INTLIT children. Return either
// the original tree or a new leaf node.
static struct ASTnode *fold1(struct ASTnode *n) {
  int val;

  // Get the child value. Do the
  // operation if recognised.
  // Return the new leaf node.
  val = n->left->a_intvalue;
  switch (n->op) {
    case A_WIDEN:
      break;
    case A_INVERT:
      val = ~val;
      break;
    case A_LOGNOT:
      val = !val;
      break;
    default:
      return (n);
  }

  // Return a leaf node with the new value
  return (mkastleaf(A_INTLIT, n->type, NULL, val));
}
```

There is one small wrinkle with implementing `fold1()` in our compiler, and that is we have AST nodes to widen values from one type to another. For example, in this expression `x= 3000 + 1;` , the '1' is parsed as a `char` literal. It needs to be widened to be of type `int` so that it can be added to the '3000'. The compiler without optimisation generates this AST tree:

```
      A_ADD
     /     \
A_INTLIT A_WIDEN
  3000      \
          A_INTLIT
              1
```

What we do here is treat the A_WIDEN as a unary AST operation and copy the literal value from the child and return a leaf node with the widened type and with the literal value.

Recursively Folding a Whole AST Tree

We have two functions to deal with the edges of the tree. Now we can code up the recursive function to optimise the edges and work from the edges back up to the root of the tree.

```
// Attempt to do constant folding on
// the AST tree with the root node n
static struct ASTnode *fold(struct ASTnode *n) {

  if (n == NULL)
    return (NULL);

  // Fold on the left child, then
  // do the same on the right child
  n->left = fold(n->left);
  n->right = fold(n->right);

  // If both children are A_INTLITs, do a fold2()
  if (n->left && n->left->op == A_INTLIT) {
    if (n->right && n->right->op == A_INTLIT)
      n = fold2(n);
    else
      // If only the left is A_INTLIT, do a fold1()
      n = fold1(n);
  }

  // Return the possibly modified tree
  return (n);
}
```

The first thing to do is return NULL on a NULL tree. This allows us to recursively call `fold()` on this node's children on the following two lines of code. We have just optimised the sub-trees below us.

Now, for an AST node with two integer literal leaf children, call `fold2()` to optimise them away (if possible). And if we have only one integer literal leaf child, call `fold1()` to do the same to it.

We either have trimmed the tree, or the tree is unchanged. Either way, we can now return it to the recursion level above us.

A Generic Optimisation Function

Constant folding is only one optimisation we can do on our AST tree; there will be others later. Thus, it makes sense to write a front-end function that applies all the optimisations to the tree. Here it is with just constant folding:

```
// Optimise an AST tree by
// constant folding in all sub-trees
struct ASTnode *optimise(struct ASTnode *n) {
  n = fold(n);
  return (n);
}
```

We can extend it later. This gets called in `function_declaration()` in `decl.c`. Once we have parsed a function and its body, we put the A_FUNCTION node on the top of the tree, and:

```
  // Build the A_FUNCTION node which has the function's symbol pointer
  // and the compound statement sub-tree
  tree = mkastunary(A_FUNCTION, type, tree, oldfuncsym, endlabel);

  // Do optimisations on the AST tree
  tree= optimise(tree);
```

An Example Function

The following program, `tests/input111.c`, should put the folding code through its paces.

```
#include <stdio.h>
int main() {
  int x= 2000 + 3 + 4 * 5 + 6;
  printf("%d\n", x);
  return(0);
}
```

The compiler should replace the initialisation with `x=2029;` . Let's do a `cwj -T -S tests/input111.c` and see:

```
$ ./cwj -T -S z.c
    A_INTLIT 2029
  A_WIDEN
  A_IDENT x
A_ASSIGN
...
$ ./cwj -o tests/input111 tests/input111.c
$ ./tests/input111
2029
```

It seems to work, and the compiler still passes all 110 previous tests, so for now it does its job.

Conclusion and What's Next

I was going to leave optimisation to the end of our journey, but I think it's good to see one type of optimisation now.

In the next part of our compiler writing journey, we will replace our current global declaration parser with code that evaluates expressions using `binexpr()` and this new constant folding code. Next step