Part 45: Global Variable Declarations, revisited

Two parts ago, I was trying to compile this line:

```
enum { TEXTLEN = 512 };         // Length of identifiers in input
extern char Text[TEXTLEN + 1];
```

and realised that our declaration parsing code could only deal with a single integer literal as the size of an array. But my compiler code, as shown above, uses an expression with two integer literals.

In the last part, I added constant folding to the compiler so that an expression of integer literals will be folded down to a single integer literal.

Now we need to discard all of that wonderful hand-written parsing of the literal value and associated casting, and call our expression parser to get an AST tree with the literal value in it.

Keep or Discard `parse_literal()`?

In our current compiler in `decl.c`, we have a function called `parse_literal()` which does the manual parsing of strings and integer literals. Should we keep it as a function, or just throw it away and call `binexpr()` manually elsewhere?

I've decided to keep the function, toss all the existing code and change the purpose of this function a little bit. It will now also parse any cast which precedes an expression with several literal values.

The function header in `decl.c` is now:

```
// Given a type, parse an expression of literals and ensure
// that the type of this expression matches the given type.
// Parse any type cast that precedes the expression.
// If an integer literal, return this value.
// If a string literal, return the label number of the string.
int parse_literal(int type);
```

So, it's a drop-in replacement for the old `parse_literal()` except that any cast parsing code we had before can be discarded. Let's now look at the new code in `parse_literal()`.

```
int parse_literal(int type) {
  struct ASTnode *tree;

  // Parse the expression and optimise the resulting AST tree
  tree= optimise(binexpr(0));
```

Ahah. We call `binexpr()` to parse whatever expression is at this point in the input file, and then `optimise()` to fold all the literal expressions.

Now, for this to be a tree we can use, the root node should be either an A_INTLIT, an A_STRLIT or a A_CAST (if there was a preceding cast).

```
  // If there's a cast, get the child and
  // mark it as having the type from the cast
  if (tree->op == A_CAST) {
    tree->left->type= tree->type;
    tree= tree->left;
  }
```

It was a cast, so we get rid of the A_CAST node but keep the type that the child was cast to.

```
  // The tree must now have an integer or string literal
  if (tree->op != A_INTLIT && tree->op != A_STRLIT)
    fatal("Cannot initialise globals with a general expression");
```

Oops, they gave us something we cannot use, so tell them and stop.

```
    // If the type is char * and
    if (type == pointer_to(P_CHAR)) {
      // We have a string literal, return the label number
      if (tree->op == A_STRLIT)
        return(tree->a_intvalue);
      // We have a zero int literal, so that's a NULL
      if (tree->op == A_INTLIT && tree->a_intvalue==0)
        return(0);
    }
```

We need to be able to accept both of these as input:

```
                char *c= "Hello";
                char *c= (char *)0;
```

and the two inner IF statements above match the two input lines shown. If not a string literal, ...

```
    // We only get here with an integer literal. The input type
    // is an integer type and is wide enough to hold the literal value
    if (inttype(type) && typesize(type, NULL) >= typesize(tree->type, NULL))
      return(tree->a_intvalue);

    fatal("Type mismatch: literal vs. variable");
    return(0);    // Keep -Wall happy
  }
```

This took me a while to figure out. We have to parse these:

```
    long  x= 3;    // allow this, where 3 is type char
    char  y= 4000; // prevent this, where 4000 is too wide
    char *z= 4000; // prevent this, as z is not integer type
```

so the IF statement checks the input type and ensures that it is wide enough to accept the integer literal.

The Other Parse Changes in `decl.c`

Now that we have a function that can parse a literal expression possibly preceded by a cast, we can use it. This is where we toss out our old cast parsing code and replace it. The changes are:

```
// Parse a scalar declaration
static struct symtable *scalar_declaration(...) {
    ...
    // Globals must be assigned a literal value
    if (class == C_GLOBAL) {
      // Create one initial value for the variable and
      // parse this value
      sym->initlist= (int *)malloc(sizeof(int));
      sym->initlist[0]= parse_literal(type);
    }
    ...
}

// Parse an array declaration
static struct symtable *array_declaration(...) {

  ...
  // See we have an array size
  if (Token.token != T_RBRACKET) {
    nelems= parse_literal(P_INT);
    if (nelems <= 0)
      fatald("Array size is illegal", nelems);
  }

  ...
  // Get the list of initial values
  while (1) {
    ...
    initlist[i++]= parse_literal(type);
    ...
  }
  ...
}
```

By doing this, we lose about 20 to 30 lines of code to parse any possible cast that used to come before the old `parse_literal()`. Mind you, we had to add 100 lines of constant folding to get that 30 line reduction! Luckily, the constant folding is used in general expressions as well as here, so it is still a win.

One `expr.c` Change

There is one further change to our compiler code to support the new `parse_literal()`. In our general function to parse expressions, `binexpr()`, we now must inform it that an expression can be ended by a '}' token, such as appears here:

```
  int fred[]= { 1, 2, 6 };
```

The small change to `binexpr()` is:

```
    // If we hit a terminating token, return just the left node
    tokentype = Token.token;
    if (tokentype == T_SEMI || tokentype == T_RPAREN ||
        tokentype == T_RBRACKET || tokentype == T_COMMA ||
        tokentype == T_COLON || tokentype == T_RBRACE) {    // T_RBRACE is new
      left->rvalue = 1;
      return (left);
    }
```

Code to Test The Changes

Our existing tests will test the situation where there is a single literal value to initialise a global variable. This code in `tests/input112.c` tests both a literal expression to initialise a scalar variable, and a literal expression as the size of an array:

```
#include <stdio.h>
char* y = NULL;
int x= 10 + 6;
int fred [ 2 + 3 ];

int main() {
  fred[2]= x;
  printf("%d\n", fred[2]);
  return(0);
}
```

Conclusion and What's Next

In the next part of our compiler writing journey, I will probably feed more of the compiler source to itself and see what we still have to implement. Next step