Part 57: Mopping Up, part 3

In this part of our compiler writing journey, I fix up a few more small issues with the compiler.

No -D Flag

Our compiler doesn't have a run-time `-D` flag to define a symbol to the pre-processor, and it would be somewhat complicated to add it in. But we use this in the `Makefile` to set the location of the directory where our header files are.

I've rewritten the `Makefile` to write this location into a new header file:

```
# Define the location of the include directory
INCDIR=/tmp/include
...

incdir.h:
        echo "#define INCDIR \"$(INCDIR)\"" > incdir.h
```

and in `defs.h` we now have:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```
#include <ctype.h>
#include "incdir.h"
```

This ensures that the location of this directory is known to the source code.

Loading Extern Variables

I've added these three external variables in `include/stdio.h` :

```
extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;
```

but when I tried to use them they were being treated as local variables! It turns out my logic to choose a global variable was wrong. In `genAST()` in `gen.c` , we now have:

```
    case A_IDENT:
      // Load our value if we are an rvalue
      // or we are being dereferenced
      if (n->rvalue || parentASTop == A_DEREF) {
        if (n->sym->class == C_GLOBAL || n->sym->class == C_STATIC
            || n->sym->class == C_EXTERN) {
          return (cgloadglob(n->sym, n->op));
        } else {
          return (cgloadlocal(n->sym, n->op));
        }
```

with the `C_EXTERN` alternative being added.

Problems with the Pratt Parser

Way back in part 3 of this journey, I introduced the Pratt parser which has a table of precedence values associated with each token. We've been using it ever since as it works.

However, I've introduced tokens that don't get parsed by the Pratt parser: prefix operators, postfix operators, casts, array element access etc. And along the way I broke the chain that ensures the Pratt parser knows the precedence of the previous operator token.

Here is the basic Pratt algorihm again, as shown by the code in `binexpr()` in `expr.c` :

```
  // Get the tree on the left.
  // Fetch the next token at the same time.
```

```
  left = prefix();
  tokentype = Token.token;

  // 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)) {
    // Fetch in the next integer literal
    scan(&Token);

    // Recursively call binexpr() with the
    // precedence of our token to build a sub-tree
    right = binexpr(OpPrec[tokentype]);

    // Join that sub-tree with ours (code not given)

    // Update the details of the current token.
    // Leave the loop if a terminating token (code not given)
    tokentype = Token.token;
  }

  // Return the tree we have when the precedence
  // is the same or lower
  return (left);
```

We must ensure that `binexpr()` gets called with the precedence of the previous token. Now let's look at how this got broken.

Consider this expression that checks if three pointers are valid:

```
if (a == NULL || b == NULL || c == NULL)
```

The `==` operator has higher precedence that the `||` operator, so the Pratt parser should treat this the same as:

```
if ((a == NULL) || (b == NULL) || (c == NULL))
```

Now, NULL is defined as this expression, and it includes a cast:

```
#define NULL (void *)0
```

So let's look at the call chain of the IF line above:

- `binexpr(0)` is called from `if_statement()`
- `binexpr(0)` parses the `==` (which has precedence 40) and calls `binexpr(40)`
- `binexpr(40)` calls `prefix()`
- `prefix()` calls `postfix()`
- `postfix()` calls `primary()`
- `primary()` sees the left parenthesis at the start of the `(void *)0` and calls `paren_expression()`
- `paren_expression()` sees the `void` token and calls `parse_cast()`. Once the cast is parsed, it calls `binexpr(0)` to parse the `0`.

And that's the problem. The value of NULL, i.e. `0` should still be at precedence level 40, but `paren_expression()` just reset it back to zero.

This means that we will now parse `NULL || b`, making an AST tree out of it instead of parsing `a == NULL` and building that AST tree.

The solution is to ensure that the previous token precedence is passed through the call chain all the way from `binexpr()` up to `paren_expression()`. This means that:

- `prefix()`, `postfix()`, `primary()` and `paren_expression()`

all now take an `int ptp` argument and this is passed on.

The program `tests/input143.c` checks that this change now works for `if (a==NULL || b==NULL || c==NULL)`.

Pointers, += and -=

A while back, I realised that if we were adding an integer value to a pointer, we needed to scale the integer by the type size that the pointer points at. For example:

```
int list[]= {3, 5, 7, 9, 11, 13, 15};
int *lptr;

int main() {
  lptr= list;
  printf("%d\n", *lptr);
  lptr= lptr + 1; printf("%d\n", *lptr);
}
```

should print the value at the base of `list`, i.e. 3. The `lptr` should be incremented by the *size* of `int`, i.e. 4, so that it now points at the next element in the `list`.

Now, we do this for the `+` and `-` operators, but I forgot to implement it for the `+=` and `-=` operators. Fortunately this was easy to fix. At the bottom of `modify_type()` in `types.c`, we now have:

```
// We can scale only on add and subtract operations
if (op == A_ADD || op == A_SUBTRACT ||
    op == A_ASPLUS || op == A_ASMINUS) {

  // Left is int type, right is pointer type and the size
  // of the original type is >1: scale the left
  if (inttype(ltype) && ptrtype(rtype)) {
    rsize = genprimsize(value_at(rtype));
    if (rsize > 1)
      return (mkastunary(A_SCALE, rtype, rctype, tree, NULL, rsize));
    else
      return (tree);          // Size 1, no need to scale
  }
}
```

You can see I've added A_ASPLUS and A_ASMINUS to the list of operations where we can scale an int value.

Conclusion and What's Next

That's enough mopping up for now. When I fixed up the `+=` and `-=` problem, it highlighted a big issue with the `++` and `--` operators (prefix and postfix) as applied to pointers.

In the next part of our compiler writing journey, I will tackle this issue. Next step