Part 47: A Subset of `sizeof`

In a real C compiler, the `sizeof()` operator gives the size in bytes of:

- a type definition, and
- the type of an expression

I looked at the code in our compiler and I'm only using `sizeof()` for the first of the two options above, so I'm only going to implement the first one. This makes things a bit easier as we can assume that the tokens inside the `sizeof()` are a type definition.

New Token and Keyword

We need a "sizeof" keyword and a new token, T_SIZEOF. As per usual, I'll let you look at the changes to `scan.c` .

Now, when adding new tokens, we also have to update the:

```
// List of token strings, for debugging purposes
char *Tstring[] = {
  "EOF", "=", "+=", "-=", "*=", "/=",
  "||", "&&", "|", "^", "&",
  "==", "!=", ",", ">", "<=", ">=", "<<", ">>",
  "+", "-", "*", "/", "++", "--", "~", "!",
  "void", "char", "int", "long",
  "if", "else", "while", "for", "return",
  "struct", "union", "enum", "typedef",
    "extern", "break", "continue", "switch",
    "case", "default", "sizeof",
    "intlit", "strlit", ";", "identifier",
    "{", "}", "(", ")", "[", "]", ",", ".",
    "->", ":"
  };
```

I initially forgot to do this, and when debugging I was seeing the "wrong" token description for the tokens after "default". Oops!

Changes to the Parser

The `sizeof()` operator is part of expression parsing, as it takes an expression and returns a new value. We can do things like:

```
    int x= 43 + sizeof(char);
```

Thus, we are going to modify `expr.c` to add `sizeof()`. It isn't a binary operator, and it's not a prefix or postfix operator, so the best place to add `sizeof()` is as part of parsing primary expressions.

In fact, once I found my silly bugs, the amount of new code to do `sizeof()` was small. Here it is:

```
  // Parse a primary factor and return an
  // AST node representing it.
  static struct ASTnode *primary(void) {
    struct ASTnode *n;
    int id;
    int type=0;
    int size, class;
    struct symtable *ctype;

    switch (Token.token) {
    case T_SIZEOF:
      // Skip the T_SIZEOF and ensure we have a left parenthesis
      scan(&Token);
      if (Token.token != T_LPAREN)
        fatal("Left parenthesis expected after sizeof");
      scan(&Token);

      // Get the type inside the parentheses
      type= parse_stars(parse_type(&ctype, &class));
      // Get the type's size
      size= typesize(type, ctype);
      rparen();
      // Return a leaf node int literal with the size
      return (mkastleaf(A_INTLIT, P_INT, NULL, size));
      ...
    }
    ...
  }
```

We already have a `parse_type()` function to parse a type definition, and we already have a `parse_stars()` function to parse any following asterisks. Finally, we already have a `typesize()` function which returns the number of bytes in a type. All we have to do is scan the tokens in, call these three functions, build a leaf AST node with an integer literal in it, and return it.

Yes, I know there are a bunch of subtleties that go with `sizeof()`, but I'm following the "KISS principle" and doing enough to make our compiler self-compiling.

Testing the New Code

The file `tests/input115.c` has a set of tests for the primitive types, a pointer and for the structures in our compiler:

```
struct foo { int x; char y; long z; };
typedef struct foo blah;

int main() {
  printf("%ld\n", sizeof(char));
  printf("%ld\n", sizeof(int));
  printf("%ld\n", sizeof(long));
  printf("%ld\n", sizeof(char *));
  printf("%ld\n", sizeof(blah));
  printf("%ld\n", sizeof(struct symtable));
  printf("%ld\n", sizeof(struct ASTnode));
  return(0);
}
```

At present, the output from our compiler is:

```
1
4
8
8
13
```

```
64
48
```

I'm wondering if we need to pad the `struct foo` struct to be 16 bytes instead of 13. We'll cross that bridge when we get to it.

Conclusion and What's Next

Well, `sizeof()` turned out to be simple, at least for the functionality that we need for our compiler. In reality, `sizeof()` is quite complicated for a full-blown production C compiler.

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