Part 20: Character and String Literals

I've been wanting to print out "Hello world" with our compiler for quite a while so, now that we have pointers and arrays, it's time in this part of the journey to add character and string literals.

These are, of course, literal values (i.e. immediately visible). Character literals have the definition of a single character surrounded by single quotes. String literals have a sequence of characters surrounded by double quotes.

Now, seriously, character and string literals in C are just completely crazy. I'm only going to implement the most obvious backslashed-escaped characters. I'm also going to borrow the character and string literal scanning code from SubC to make my life easier.

This part of the journey is going to be short, but it will end with "Hello world".

A New Token

We need a single new token for our language: T_STRLIT. This is very similar to T_IDENT in that the text associated with the token is stored in the global `Text` and not in the token structure itself.

Scanning Character Literals

A character literal starts with a single quote, is followed by the definition of a single character and ends with another single quote. The code to interpret that single character is complicated, so let's modify `scan()` in `scan.c` to call it:

```
      case '\'':
      // If it's a quote, scan in the
      // literal character value and
      // the trailing quote
      t->intvalue = scanch();
      t->token = T_INTLIT;
      if (next() != '\'')
        fatal("Expected '\\'' at end of char literal");
      break;
```

We can treat a character literal as an integer literal of type `char`; that is, assuming that we limit ourselves to ASCII and don't try to deal with Unicode. That's what I'm doing here.

The Code for `scanch()`

The code for the `scanch()` function comes from SubC with a few simplifications:

```
// Return the next character from a character
// or string literal
static int scanch(void) {
  int c;

  // Get the next input character and interpret
  // metacharacters that start with a backslash
  c = next();
  if (c == '\\') {
    switch (c = next()) {
      case 'a':  return '\a';
      case 'b':  return '\b';
      case 'f':  return '\f';
      case 'n':  return '\n';
      case 'r':  return '\r';
      case 't':  return '\t';
      case 'v':  return '\v';
      case '\\': return '\\';
      case '"':  return '"' ;
      case '\'': return '\'';
      default:
        fatalc("unknown escape sequence", c);
      }
    }
    return (c);                   // Just an ordinary old character!
  }
```

The code recognises most of the escaped character sequences, but it doesn't try to recognise octal character codings or other difficult sequences.

Scanning String Literals

A string literal starts with a double quote, is followed by zero or more characters and ends with another double quote. As with character literals, we need to call a separate function in `scan()`:

```
    case '"':
      // Scan in a literal string
      scanstr(Text);
      t->token= T_STRLIT;
      break;
```

We create one of the new T_STRLIT and scan the string into the `Text` buffer. Here is the code for `scanstr()`:

```
// Scan in a string literal from the input file,
// and store it in buf[]. Return the length of
// the string.
static int scanstr(char *buf) {
  int i, c;

  // Loop while we have enough buffer space
  for (i=0; i<TEXTLEN-1; i++) {
    // Get the next char and append to buf
    // Return when we hit the ending double quote
    if ((c = scanch()) == '"') {
      buf[i] = 0;
      return(i);
    }
    buf[i] = c;
  }
  // Ran out of buf[] space
  fatal("String literal too long");
  return(0);
}
```

I think the code is straight-forward. It NUL terminates the string that is scanned in, and ensures that it doesn't overflow the `Text` buffer. Note that we use the `scanch()` function to scan in individual characters.

Parsing String Literals

As I mentioned, character literals are treated as integer literals which we already deal with. Where can we have string literals? Going back to the BNF Grammar for C written by Jeff Lee in 1985, we see:

```
primary_expression
        : IDENTIFIER
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
        ;
```

and thus we know that we should modify `primary()` in `expr.c` :

```
// Parse a primary factor and return an
// AST node representing it.
static struct ASTnode *primary(void) {
  struct ASTnode *n;
  int id;


  switch (Token.token) {
  case T_STRLIT:
    // For a STRLIT token, generate the assembly for it.
    // Then make a leaf AST node for it. id is the string's label.
    id= genglobstr(Text);
    n= mkastleaf(A_STRLIT, P_CHARPTR, id);
    break;
```

Right now, I'm going to make an anonymous global string. It needs to have all the characters in the string stored in memory, and we also need a way to refer to it. I don't want to pollute the symbol table with this new string, so I've chosen to allocate a label for the string and store the label's number in the AST node for this string literal. We also need a new AST node type: A_STRLIT. The label is effectively the base of the array of characters in the string, and thus it should be of type P_CHARPTR.

I'll come back to the generation of the assembly output, done by `genglobstr()` soon.

An Example AST Tree

Right now, a string literal is treated as an anonymous pointer. Here's the AST tree for the statement:

```
    char *s;
    s= "Hello world";

    A_STRLIT rval label L2
    A_IDENT s
  A_ASSIGN
```

They are both the same type, so there is no need to scale or widen anything.

Generating the Assembly Output

In the generic code generator, there are very few changes. We need a function to generate the storage for a new string. We need to allocate a label for it and then output the string's contents (in `gen.c`):

```
int genglobstr(char *strvalue) {
  int l= genlabel();
  cgglobstr(l, strvalue);
  return(l);
}
```

And we need to recognise the A_STRLIT AST node type and generate assembly code for it. In `genAST()`,

```
    case A_STRLIT:
        return (cgloadglobstr(n->v.id));
```

Generating the x86-64 Assembly Output

We finally get to the actuall new assembly output functions. There are two: one to generate the string's storage and the other to load the base address of the string.

```
// Generate a global string and its start label
void cgglobstr(int l, char *strvalue) {
  char *cptr;
  cglabel(l);
```

```
    for (cptr= strvalue; *cptr; cptr++) {
      fprintf(Outfile, "\t.byte\t%d\n", *cptr);
    }
    fprintf(Outfile, "\t.byte\t0\n");
  }

  // Given the label number of a global string,
  // load its address into a new register
  int cgloadglobstr(int id) {
    // Get a new register
    int r = alloc_register();
    fprintf(Outfile, "\tleaq\tL%d(\%%rip), %s\n", id, reglist[r]);
    return (r);
  }
```

Going back to our example:

```
    char *s;
    s= "Hello world";
```

The assembly output for this is:

```
  L2:     .byte   72              # Anonymous string
          .byte   101
          .byte   108
          .byte   108
          .byte   111
          .byte   32
          .byte   119
          .byte   111
          .byte   114
          .byte   108
          .byte   100
          .byte   0
          ...
          leaq    L2(%rip), %r8   # Load L2's address
          movq    %r8, s(%rip)    # and store in s
```

Miscellaneous Changes

When writing the test program for this part of the journey, I uncovered another bug in the existing code. When scaling an integer value to match the type size that a pointer points to, I forgot to do nothing when the scale was 1. The code in `modify_type()` in `types.c` is now:

```
      // 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, tree, rsize));
        else
          return (tree);          // Size 1, no need to scale
      }
```

I'd left the `return (tree)` out, thus returning a NULL tree when trying to scale `char *` pointers.

Conclusion and What's Next

I'm so glad that we can now output text:

```
$ make test
./comp1 tests/input21.c
cc -o out out.s lib/printint.c
./out
10
Hello world
```

Most of the work this time was extending our lexical scanner to deal with the character and string literal delimiters and the escaping of characters inside them. But there was some work done on the code generator, too.

In the next part of our compiler writing journey, we'll add some more binary operators to the language that the compiler recognises. Next step