rzaharia Updated all readme files to contain links to the next step last year

329 lines (263 loc) · 7.95 KB

Preview Code Blame Raw

Part 1: Introduction to Lexical Scanning

We start our compiler writing journey with a simple lexical scanner. As I mentioned in the previous part, the job of the scanner is to identify the lexical elements, or *tokens*, in the input language.

We will start with a language that has only five lexical elements:

- the four basic maths operators: `*`, `/`, `+` and `-`
- decimal whole numbers which have 1 or more digits `0` .. `9`

Each token that we scan is going to be stored in this structure (from `defs.h`):

```
// Token structure
struct token {
  int token;
  int intvalue;
};
```

where the `token` field can be one of these values (from `defs.h`):

```
// Tokens
enum {
  T_PLUS, T_MINUS, T_STAR, T_SLASH, T_INTLIT
};
```

When the token is a `T_INTLIT` (i.e. an integer literal), the `intvalue` field will hold the value of the integer that we scanned in.

Functions in `scan.c`

The `scan.c` file holds the functions of our lexical scanner. We are going to read in one character at a time from our input file. However, there will be times when we need to "put back" a character if we have read too far ahead in the input stream. We also want to track what line we are currently on so that we can print the line number in our debug messages. All of this is done by the `next()` function:

```
// Get the next character from the input file.
static int next(void) {
  int c;

  if (Putback) {                // Use the character put
    c = Putback;                // back if there is one
    Putback = 0;
    return c;
  }

  c = fgetc(Infile);            // Read from input file
  if ('\n' == c)
    Line++;                     // Increment line count
  return c;
}
```

The `Putback` and `Line` variables are defined in `data.h` along with our input file pointer:

```
extern_ int     Line;
extern_ int     Putback;
extern_ FILE    *Infile;
```

All C files will include this where `extern_` is replaced with `extern`. But `main.c` will remove the `extern_`; hence, these variables will "belong" to `main.c`.

Finally, how do we put a character back into the input stream? Thus:

```
// Put back an unwanted character
static void putback(int c) {
  Putback = c;
}
```

Ignoring Whitespace

We need a function that reads and silently skips whitespace characters until it gets a non-whitespace character, and returns it. Thus:

```
// Skip past input that we don't need to deal with,
// i.e. whitespace, newlines. Return the first
// character we do need to deal with.
static int skip(void) {
  int c;

  c = next();
  while (' ' == c || '\t' == c || '\n' == c || '\r' == c || '\f' == c) {
    c = next();
  }
  return (c);
}
```

Scanning Tokens: `scan()`

So now we can read characters in while skipping whitespace; we can also put back a character if we read one character too far ahead. We can now write our first lexical scanner:

```
// Scan and return the next token found in the input.
// Return 1 if token valid, 0 if no tokens left.
int scan(struct token *t) {
  int c;

  // Skip whitespace
  c = skip();

  // Determine the token based on
  // the input character
  switch (c) {
  case EOF:
    return (0);
  case '+':
    t->token = T_PLUS;
    break;
  case '-':
    t->token = T_MINUS;
    break;
  case '*':
    t->token = T_STAR;
    break;
  case '/':
```

```
      t->token = T_SLASH;
      break;
    default:
      // More here soon
    }

    // We found a token
    return (1);
  }
```

That's it for the simple one-character tokens: for each recognised character, turn it into a token. You may ask: why not just put the recognised character into the `struct token`? The answer is that later we will need to recognise multi-character tokens such as `==` and keywords like `if` and `while`. So it will make life easier to have an enumerated list of token values.

Integer Literal Values

In fact, we already have to face this situation as we also need to recognise integer literal values like `3827` and `87731`. Here is the missing `default` code from the `switch` statement:

```
    default:

      // If it's a digit, scan the
      // literal integer value in
      if (isdigit(c)) {
        t->intvalue = scanint(c);
        t->token = T_INTLIT;
        break;
      }

      printf("Unrecognised character %c on line %d\n", c, Line);
      exit(1);
```

Once we hit a decimal digit character, we call the helper function `scanint()` with this first character. It will return the scanned integer value. To do this, it has to read each character in turn, check that it's a legitimate digit, and build up the final number. Here is the code:

```
// Scan and return an integer literal
// value from the input file. Store
// the value as a string in Text.
static int scanint(int c) {
  int k, val = 0;

  // Convert each character into an int value
```

```
  while ((k = chrpos("0123456789", c)) >= 0) {
    val = val * 10 + k;
    c = next();
  }

  // We hit a non-integer character, put it back.
  putback(c);
  return val;
}
```

We start with a zero `val` value. Each time we get a character in the set `0` to `9` we convert this to an `int` value with `chrpos()`. We make `val` 10 times bigger and then add this new digit to it.

For example, if we have the characters `3`, `2`, `8`, we do:

- `val= 0 * 10 + 3`, i.e. 3
- `val= 3 * 10 + 2`, i.e. 32
- `val= 32 * 10 + 8`, i.e. 328

Right at the end, did you notice the call to `putback(c)`? We found a character that's not a decimal digit at this point. We can't simply discard it, but luckily we can put it back in the input stream to be consumed later.

You may also ask at this point: why not simply subtract the ASCII value of '0' from `c` to make it an integer? The answer is that, later on, we will be able to do `chrpos("0123456789abcdef")` to convert hexadecimal digits as well.

Here's the code for `chrpos()`:

```
// Return the position of character c
// in string s, or -1 if c not found
static int chrpos(char *s, int c) {
  char *p;

  p = strchr(s, c);
  return (p ? p - s : -1);
}
```

And that's it for the lexical scanner code in `scan.c` for now.

Putting the Scanner to Work

The code in `main.c` puts the above scanner to work. The `main()` function opens up a file and then scans it for tokens:

```
void main(int argc, char *argv[]) {
  ...
  init();
  ...
  Infile = fopen(argv[1], "r");
  ...
  scanfile();
  exit(0);
}
```

And `scanfile()` loops while there is a new token and prints out the details of the token:

```
// List of printable tokens
char *tokstr[] = { "+", "-", "*", "/", "intlit" };

// Loop scanning in all the tokens in the input file.
// Print out details of each token found.
static void scanfile() {
  struct token T;

  while (scan(&T)) {
    printf("Token %s", tokstr[T.token]);
    if (T.token == T_INTLIT)
      printf(", value %d", T.intvalue);
    printf("\n");
  }
}
```

Some Example Input Files

I've provided some example input files so you can see what tokens the scanner finds in each file, and what input files the scanner rejects.

```
$ make
cc -o scanner -g main.c scan.c

$ cat input01
2 + 3 * 5 - 8 / 3

$ ./scanner input01
Token intlit, value 2
Token +
Token intlit, value 3
Token *
Token intlit, value 5
Token -
```

```
Token intlit, value 8
Token /
Token intlit, value 3

$ cat input04
23 +
18 -
45.6 * 2
/ 18

$ ./scanner input04
Token intlit, value 23
Token +
Token intlit, value 18
Token -
Token intlit, value 45
Unrecognised character . on line 3
```

Conclusion and What's Next

We've started small and we have a simple lexical scanner that recognises the four main maths operators and also integer literal values. We saw that we needed to skip whitespace and put back characters if we read too far into the input.

Single character tokens are easy to scan, but multi-character tokens are a bit harder. But at the end, the `scan()` function returns the next token from the input file in a `struct token` variable:

```
struct token {
  int token;
  int intvalue;
};
```

In the next part of our compiler writing journey, we will build a recursive descent parser to interpret the grammar of our input files, and calculate & print out the final value for each file.
Next step