Scanning and Parsing

Lab 4 - COMP211 - Spring 2020

Abstract

In this lab you will implement a `Scanner` and `Parser` for a simplified shell language. This lab is a continuation from the previous lab where you built abstractions for dynamic vectors and strings.

Grading

Autograding will make up 90% of your grade on this lab. The autograder will be based on the same unit tests you are provided below can run locally on your container. The remaining 10% will be based on style and correctly freeing a `Node` tree's memory in part 2.

Part 0. Testing your `Str`

Be sure your `Str` implementation from the previous lab is passing all of the unit tests. You can download the grader unit tests as shown below. From the working directory of your `thsh` project, you can download its support files from GitHub by redirecting `curl`'s output:

```
repo=https://raw.githubusercontent.com/comp211-20f/thsh-2/master
curl "${repo}/test/unit/StrGrader.cpp" >test/unit/StrGrader.cpp
```

Aside: The `curl` program downloads data from a URL and writes it to standard output. Notice, in the commands above, you're making use of a shell variable assignment, parameter (variable) substitution in a string, and output redirection to save a file from the internet to your project.

You can run the tests with `make unit-test` and should pass all `StrGrader` tests before continuing to scanning.

Part 1. Scanning for Lexemes to Produce Tokens

Support Code `CharItr`

A character iterator abstraction is provided as support code you'll want to bring into your project and become familiar with. Download the `CharItr` files into your project:

```
repo=https://raw.githubusercontent.com/comp211-20f/thsh-2/master
curl "${repo}/include/CharItr.h" >include/CharItr.h
curl "${repo}/src/CharItr.c" >src/CharItr.c
```

You should familiarize yourself with `CharItr.h` as you will need to make use of its functions while Scanning.

Lexemes

The `thsh` shell grammar has three kinds of lexemes:

A `word` is a sequence of ASCII characters not including `SPACE, \t, \n, |, \0, EOF`.

A `pipe` is a `|`

An `end` is `EOF`, `\0`

Expressed as a *regular definition* using extended regular expression syntax:

```
word ::= [^ \t\n|\0]+
pipe ::= |
end  ::= [EOF\0]
```

Note that the `^` at the beginning of the `word` definition's character class is a *negation* or *complement* meaning *not* the charcters `\n`, `\t`, `|`, or `|0`. As a reminder, the `+` means "one or more". Whitespace characters, *spaces*, *tabs*, and *new lines* are ignored and not a part of any lexeme.

Tokens and Scanner

The definitions for `TokenType`, `Token`, and `Scanner` are found in `include/Scanner.h` which you will want to download:

```
curl "${repo}/include/Scanner.h" >include/Scanner.h
curl "${repo}/src/Scanner.c" >src/Scanner.c
curl "${repo}/test/unit/ScannerSpec.cpp" >test/unit/ScannerSpec.cpp
```

Open and read through `include/Scanner.h` to get a sense of the `enum TokenType`, `struct Token`, and `struct Scanner` with related functions.

Your job in this part of the lab is to implement `src/Scanner.c`. Skeleton implementations of the four `Scanner` related functions are provided. You will need to reimplement each of them to satisfy the documentation comments in `include/Scanner.h`.

The `next` member of `Scanner` should always hold the next peekable `Token` value. You should define `static` helper functions to decompose the concerns of scanning and reduce unnecessary duplication between functions.

Unit tests for `Scanner` are provided in the `test/unit/ScannerSpec.cpp` file. You are welcome to add your own tets, but we have decided to not make it necessary for this lab given our time constraints. Once you are passing all `ScannerSpec` tests, you can assume your `Scanner` is adequately functional.

We've created a sample `main.c` program you can use to interact with your `Scanner`, as well. It reads a line of input into a `Str`, and then prints out the scanned `Token` values. You can download it and overwrite your current `main.c` to tinker: `curl "${repo}/src/scanner-main.c >main.c`

Part 2 - Parsing

The structrual grammar of your Parser follows. Note that the *start symbol* is `Node`, non-terminals are `Capitalized` and terminal tokens are `lowercase`:

```
Node        ::= Command | Pipe
Pipe        ::= Node pipe Node
Command     ::= word+
```

A `Node` is either a `Pipe` or a `Command`. A `Pipe` is, recursively, a `Node` followed by a `pipe Token` (whose lexeme is a `|`), followed by a `Node`. Finally, a command is a sequence of one or more `word Tokens`.

Unfortunately, the above grammar features left-recursion in the `Pipe` production rule because its first symbol is `Node`. It's also ambiguous. For a recursive descent parser, you want to *rewrite* a grammar featuring left recursion and ambiguity. The above grammar was presented first because it most closely conveys the structure of the tree of `Node`s your `Parser` will produce. However, for the purposes of implementation, your Parser should be guided by the following rewritten grammar:

```
Node          ::= Command (pipe Node)?
Command       ::= word+
```

A `Node` is a `Command`, optionally (because of the zero-or-one question mark) followed by a `pipe Token` followed by a `Node` recurisvely. Notice the left recursion is removed through a slight of hand. The cost is the first production rule is more nuanced than in the original grammar. You, as the implementor, will either produce a `COMMAND_NODE` or a `PIPE_NODE`, but you will not know which until peeking ahead *after* you've parsed the leading `Command`.

Code

There are a few supporting files to help get you started on this part.

```
curl "${repo}/include/Node.h" >include/Node.h
curl "${repo}/src/Node.c" >src/Node.c

curl "${repo}/include/StrVec.h" >include/StrVec.h
curl "${repo}/src/StrVec.c" >src/StrVec.c

curl "${repo}/include/Parser.h" >include/Parser.h
curl "${repo}/src/Parser.c" >src/Parser.c
curl "${repo}/test/unit/ParserSpec.cpp" >test/unit/ParserSpec.cpp
```

Before starting, you should familiarize yourself with `Node.h` and the `struct Node`. Pay particular attention to the `enum NodeType`, `union NodeValue`, `struct CommandValue`, and `struct PipeValue`. You should be able to draw direct connections between these definitions and the structure of the grammar we're attempting to parse. As an exercise, you are encouraged to diagram out how you imagine a line like `ls | grep foo | less` parsing into a tree of `Node` values.

Notice the `Node` constructors defined at the bottom of the `Node.h` header file. You should read through their implementations in `Node.c` to understand what they're doing for you.

Notice a `CommandValue` is made up of `StrVec` named `words`. In the previous midterm, you implemented the functions of `StrVec` found in `src/StrVec.c`. Working implementations are provided for you here.

Your primary task is to implement parsing in `Parser.c`. You should define additional static helper function(s) as necessary to implement a simple recursive descent parser for the grammar given above. Refer to the lecture on parsing for the process of implementing a top-down parser from a grammar. Unit tests for your `parse` function are provided in the `test/unit/ParserSpec.cpp` file.

Your final task is to complete the implementation of `Node_drop` in `src/Node.c` to free all owned memory of a `Node` on the heap. Consider that both the `Node` values themselves, as well as any *owned* memory need to be freed. For pipe nodes, this is a recursive process. A runner program that prints out the results of your `parse` function is also being provided which can help you test memory leaks. You can download it from the repo like you did other files above using the URL `"${repo}/src/parser-main.c"` and redirecting to `src/main.c`.