Assignment 3

CSCI 3136: Principles of Programming Languages

Due Mar 4, 2019

This is the first in a sequence of three programming assignments that culminate in an interpreter for Scheme--, a small subset of Scheme. This assignment focuses on implementing a scanner (lexical analyzer) for Scheme--. I begin by describing the lexical structure of Scheme--.

Lexical Structure of Scheme--

Whitespace and token types. Tokens cannot include space characters (with the exception of string literals). Thus, space characters are used to delimit tokens. Two tokens can be separated by one or more spaces.

In the following, a token is considered to be a **single-character token** if it cannot consist of more than one character. It is a **multi-character token** if its definition *allows* it to consist of multiple characters **even if the actual token has only one character in it**. Two examples: A parenthesis is a single-character token; there is no string of two or more characters the scanner would consider to be a parenthesis. An identifier is a multi-character token because, for example, `aname` is a valid identifier. Thus, even though `x` has only one character, it is considered a multi-character token because it is an identifier.

Multi-character tokens (e.g., two identifiers or an identifier and a number) **must** be separated by at least one space. If at least one of two consecutive tokens is a single-character token, the two tokens do not need to be separated by a space. (E.g., a parenthesis or identifier can immediately follow a parenthesis.)

Beyond serving as separators between tokens, spaces have no meaning in Scheme--.

Comments. A semicolon (`;`) starts a comment except when inside a string literal or character literal. All characters from the semicolon to the end of the line are part of the comment. Comments must be ignored and treated as if they were spaces by the scanner.

Parentheses. Parentheses (`()[]{}`) are special characters in Scheme--. Every parenthesis is a single-character token by itself. These tokens are `OPENRD`, `CLOSERD`, `OPENSQ`, `CLOSESQ`, `OPENCU`, and `CLOSECU`, representing opening and closing round, square, and curly parentheses.

Semantically, there is no difference between the different types of paretheses, but the syntax definition in the next assignment requires parenthesis types to match (an opening square bracket must be closed by a matching closing square bracket, etc.), so the parser needs information about the type of parenthesis encountered. Hence, the 6 separate token types.

As an example, the character sequence `(]{` defines a sequence of three parenthesis tokens `OPENRD`, `CLOSESQ`, `OPENCU`. The parser will complain about the round parenthesis and the square parenthesis not matching, but lexically, this sequence is perfectly acceptable.

Numbers literals. Every number literal is to be represented as a multi-character token `NUMBER`. With Scheme-- being dynamically typed, from a lexical and syntactic standpoint, there is no need to distinguish whether this number is an integer or floating point number.

Scheme-- knows two types of number literals: integers and floating point numbers.

An integer literal is any maximal-length character sequence that matches one of the following regular expressions:

- `[-+]?[0-9]+` (decimal signed integer)
- `0x[0-9a-fA-F]+` (hexadecimal unsigned integer)
- `0b[01]+` (binary unsigned integer)

A floating-point literal is any maximal-length character sequence that matches one of the following regular expressions:

- `[-+]?[0-9]*\.[0-9]+([eE][-+]?[0-9]+)?`
- `[-+]?[0-9]+[eE][-+]?[0-9]+`

Boolean literals. A Boolean literal, represented by the token name `BOOL`, is one of two strings: `#t` (true) or `#f` (false). This is a multi-character token.

Character literals. A character literal, represented by the token name `CHAR`, is any maximal-length character sequence that matches one of the following regular expressions:

- `#\\.` (any printable character)
- `#\\newline` (the newline character)
- `#\\space` (a space)
- `#\\tab` (the tab character)
- `#\\[0-3][0-7]{2}` (a character identified by its 3-digit octal character code)

For simplicity, we disallow unicode characters and only allow characters in the extended ASCII range 0–255, which in octal notation is 0–377.

String literals. A string literal, represented by the token name `STRING`, is any character sequence that matches the following regular expression:

- `"([^\"]|\\([tn]|[0-3][0-7]{2}))*"`

Keywords. There are a small number of keywords: `lambda`, `define`, `let`, `cond`, `if`, `begin`, `quote`. Your scanner should represent these using the tokens `LAMBDA`, `DEFINE`, `LET`, `COND`, `IF`, `BEGIN`, `QUOTE`. These are multi-character tokens.

Identifiers. An identifier is any maximal-length character sequence that matches the following regular expression and is not a keyword.

- `[^][(){};0-9'"#][^][(){};'"#]*`

This is a multi-character token.

Quotes. A quote is the single character `'`. This is a single-character token. Scheme-- does not support quasiquotation.

Scanner Implementation

Your scanner must be implemented in Java or R6RS Scheme. In both cases, your code must compile and be executable using the Java or Chez Scheme environment on `bluenose`. For the Java implementation, this implies in particular that only Java 8 constructs are valid because the Java version on `bluenose` is Java 8. A caveat that may influence your choice of programming language is that I do not have the time to provide sample implementations in both languages. Given that a significantly larger number of students indicated they would prefer to implement their Scheme-- interpreter in Java, I will make Java sample implementations available. This is important to take into account in case you get stuck in one of the assignments and do not produce a fully functional implementation. Since the scanner, parser, and interpreter assignments build on top of each other, the sample implementations provide reset points that allow you to attempt the next assignment even if you did not fully complete the previous assignment. Thus, if you attempt your implementation in Scheme and do not succeed, you will be forced to switch to Java in subsequent assignments.

The following are the main requirements for your scanner implementation.

- The top-level program must be a scheme program `scanner.ss` that can be run as

  ```
  ./scanner.ss <scheme-- source file>
  ```

  or a Java class `Scanner.java` that can be compiled and run as

  ```
  java Scanner <scheme -- source file>
  ```

- Given a Scheme-- source file, your scanner should output the sequence of tokens in the input, one token per line. Each token should be annotated with the position of the token in the input. (More details in the example below.)
- If the input is lexically incorrect, your scanner should not produce any output except a single error message corresponding to the first incorrect token. The error message should indicate what is wrong and the position in the input where the error was found. After reporting this error, the scanner should not continue to try to analyze the remainder of the input (because this requires error recovery strategies beyond the scope of this course).

Two examples to illustrate these requirements:

Correct input: On the following input

```
(define (fibonacci n)
  (let fib ([prev 0]
            [cur 1]
            [i 0])
    (if (= i n)
        cur
        (fib cur (+ prev cur) (+ i 1)))))
```

your scanner should output the following token stream:

```
OPENRD 1:1
DEFINE 1:2
OPENRD 1:9
IDENTIFIER 1:10
IDENTIFIER 1:20
CLOSERD 1:21
OPENRD 2:3
LET 2:4
IDENTIFIER 2:8
OPENRD 2:12
OPENSQ 2:13
IDENTIFIER 2:14
NUMBER 2:19
CLOSESQ 2:20
OPENSQ 3:13
IDENTIFIER 3:14
NUMBER 3:18
CLOSESQ 3:19
OPENSQ 4:13
IDENTIFIER 4:14
NUMBER 4:16
CLOSESQ 4:17
CLOSERD 4:18
OPENRD 5:5
IF 5:6
OPENRD 5:9
IDENTIFIER 5:10
IDENTIFIER 5:12
IDENTIFIER 5:14
CLOSERD 5:15
IDENTIFIER 6:9
OPENRD 7:9
IDENTIFIER 7:10
IDENTIFIER 7:14
OPENRD 7:18
IDENTIFIER 7:19
IDENTIFIER 7:21
IDENTIFIER 7:26
CLOSERD 7:29
OPENRD 7:31
IDENTIFIER 7:32
IDENTIFIER 7:34
NUMBER 7:36
CLOSERD 7:37
CLOSERD 7:38
CLOSERD 7:39
CLOSERD 7:40
CLOSERD 7:41
```

Incorrect input: On the following input

```
(define (fibonacci n)
  (let 0fib ([prev 0]
            [cur 1]
            [i 0])
    (if (= i n)
        cur
        (fib cur (+ prev cur) (+ i 1)))))
```

your scanner should output the following error message:

```
LEXICAL ERROR [2:8]: Invalid token `0fib'
```

Specifically, if an error is encountered upon reading the first character of a token, the error should be treated as an invalid single-character token and only the first character should be included in the error message. If an error is encountered after reading the first character, the invalid token should be treated as a multi-character token. Thus, the entire text from the beginning of the token to the next space character should be reported as the text of the incorrect token. In the above example, a "0" can be read without error because this may be the beginning of a number literal, but reading the letter "f" is an error. Thus, the input is an invalid multi-character token and the entire substring "0fib" should be reported as the incorrect token.

Implementation requirements: In terms of the implementation requirements, your scanner must meet two conditions:

- You must not use any regular expression library to recognize tokens.
- You must implement a DFA to split your input into tokens. You can do this in two ways: you can hand-code the scanner or you can implement functions that convert a set of regular expressions for the different tokens into a DFA that accepts these tokens. The second option is the more instructive one. The first option is the easier one for a simple language as Scheme--.

Submission Instructions

Place your code in a folder `a3` of your SVN repository and submit using the appropriate combination of `svn add` and `svn commit`.