CSE2260 - Principles of Programming Languages

PROJECT 1 – Part 1 (Due 14/04/2023 23:59)

In this project, you will implement a lexical analyzer (scanner) for the given specific programming language, i.e., PPLL. The details of the PPLL are given below.

Whitespaces cannot exist within tokens. They are used to separate tokens.

Comments: A tilde (~) starts a comment except when inside a string literal or character literal. All characters from the tilde to the end of the line are considered as a comment. Scanner must ignore comments and treat as if they are spaces.

The tokens in the language:

Brackets: Brackets (()[]{}) are special characters. Every bracket is a single-character token by itself. These tokens are:

- **LEFTPAR** for (,
- **RIGHTPAR** for),
- **LEFTSQUAREB** for [,
- **RIGHTSQUAREB** for],
- **LEFTCURLYB** for {, and
- **RIGHTCURLYB** for }

Number literals: There are two types of numbers: integer and floating-point number. The lexical analyzer will not determine whether a number is an integer or floating-point number, all numbers will be recognized as **NUMBER** token.

Regular expressions for integer numbers:

- *Decimal signed integer:* (**-**|**+**|ε)(DEC_DIGIT)$^+$
- *Hexadecimal unsigned integer:* **0x**(HEX_DIGIT)$^+$
- *Binary unsigned integer:* **0b**(BIN_DIGIT)$^+$

Regular expressions for floating-point numbers:

- (**-**|**+**|ε) (DEC_DIGIT)***.**(DEC_DIGIT)$^+$(((**e**|**E**)(**-**|**+**|ε)(DEC_DIGIT)$^+$) | ε)
- (**-**|**+**|ε) (DEC_DIGIT)$^+$(**e**|**E**)(**-**|**+**|ε)(DEC_DIGIT)$^+$

- BIN_DIGIT : [0 | 1]
- DEC_DIGIT : [0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9]
- HEX_DIGIT : [0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | A | B | C | D | E | F]

Boolean literals: A Boolean, represented by the token name **BOOLEAN**, is one of two strings: **true** or **false**. This is a multi-character token.

Character literals: A character literal is a single Unicode character enclosed within two single-quote (') characters, except for single-quote itself, which must be escaped by a preceding backslash (\) character.
Characters are represented by the token name **CHAR**. The regular expression for character literal is: **'** (**\'** | \\ | ALL_UNICODE) **'**

String literals: A string literal, represented by the token name **STRING**, is any character sequence between two double-quotes (**"**). This is a multi-character token. The regular expression for string literal is: **"** (**\"** | \\ | ALL_UNICODE)$^{+}$ **"**

Keywords: There are a small number of keywords: define, let, cond, if, begin. Your scanner should represent these using the tokens **DEFINE**, **LET**, **COND**, **IF**, **BEGIN**. These are multi-character tokens. Keywords are case-sensitive (i.e., all small letters).

Identifiers: An **IDENTIFIER** is any maximal-length character sequence that matches the following regular expression and is not a keyword. This is a multi-character token. The regular expression for identifiers is:
((+ | - | .) | ((! | * | / | : | < | = | > | ? | LETTER) (LETTER | DEC_DIGIT | (. | + | -))*))

- LETTER: [a | b | c | ... | z]

Your scanner must be implemented in Java or C.

• Given a PPLL 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.

IMPORTANT: You must not use any regular expression library to recognize tokens.

Sample Input/Output

1. Correct input: On the following input your scanner should output the following token stream:

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

Corresponding Output:

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

2. **Incorrect input:** On the following input your scanner should output the following error message:

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

Corresponding Output:
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" after "0" is an error. Thus, the input is an invalid multi-character token and the entire substring "0fib" should be reported as the incorrect token.