444 Compilers — Project #1 — Lexer

Project #1: Lexer

Introduction

The objective of this assignment is to write a lexer for the A4 language lexicon (i.e., legal "words"). The lexer transforms an A4 high-level program into a list of tokens for that program (in a special format). For convenience, this lexer will take input from standard-input (stdin) and send output to standard-output (stdout). Coding is in Java.

Note, this project will form the basis for the next project, so apply Rule #4 as appropriate.

Team

The team may contain from two to four members. Pick a three-letter name for the team. (If two teams pick the same TLA, I will disambiguate them.)

Guidelines

The **A4 Lexcon** is given, below. It has a regex section for complicated words and a more straight forward simple string section (which is also regex, but trivial). Each token type has an ID number. Recall that a lexer is built via a DFA, (usually in the form of a set of nested switch statements) and by calling a couple character functions: peek() and advance(). Peek() returns a copy of the next character in the input stream. Advance() moves past that next character.

In addition to the Lexicon, shown below, it is to be assumed that whitespace delimits tokens, but whitespace is itself not a token; meaning that encountering a whitespace char will terminate a token (e.g., an identifier). Further, a comment (see the lexicon) is treated as whitespace.

The lexer must be hand-built, rather than built by using a lexer generator program (e.g., LEX).

Small sample A4 programs and their corresponding Token records as output are shown, below.

Each lexer is to be built by a team of up to 3 students).

Note, a period character not in quotes is a regex operator that matches any character except newline.

A4 Lexicon

```
    comment = '//' .*  // Treat as whitespace up to newline char; like C/C++/Java.
2 id = LU LUD *  // identifier.
  LU = '_' | [a-zA-Z]  // Letter-Underscore.
  LUD = LU | [0-9]  // Letter-Underscore-Digit.
3 int = SIGN ? DIGITS // integer.
4 float = int [ '.' DIGITS ] ? // float.
5 string = '"' .* '"' // Cannot contain a double-quote char.
  SIGN = plus | minus
  DIGITS = [0-9] +

// Unpaired delimiters
 6 comma = ','
 7 semi = ';'

// Keywords
10 kprog = "prog"
11 kmain = "main"
12 kfcn = "fcn"
13 kclass = "class"
15 kfloat = "float"
16 kint = "int"
17 kstring = "string"
18 kif = "if"
19 kelseif = "elseif"
20 kelse = "else"
21 kwhile = "while"
22 kinput = "input"
23 kprint = "print"
24 knew = "new"
25 kreturn = "return"
26 kvar = "var"

// Paired delimeters
31 angle1 = '<'
32 angle2 = '>'
33 brace1 = '{'
34 brace2 = '}'
35 bracket1 = '['
36 bracket2 = ']'
37 parens1 = '('
38 parens2 = ')'

// Other punctuation tokens
41 aster = '*'
42 caret = '^'
43 colon = ':'
44 dot = '.'
45 equal = '='
46 minus = '-'
47 plus = '+'
48 slash = '/'
```

```
// Multi-char operators
51 oparrow = "->"
52 opeq = "=="
53 opne = "!="
54 ople = "<="
55 opge = ">="
56 opshl = "<<"
57 opshr = ">>"

// Miscellaeous
99 error // Unknown token.
 0 eof // End-of-Input.\
```

A4 Token Output Format

```
(Tok: <ID#> line= <line> str= "<token>" [int= <int> | float= <float>]?)
```

The ID# is that of the A4 Lexicon, described above.

For a string token (id==5), the string token's input double-quotes are **not included** in the str field. Those double-quotes are only used to help delimit and identify the string's character sequence.

For an int (id==3) or a float (id==4), include the optional "int=" or "float=" field. You compute the actual value from the string of digits.

The token's line field contains the source line number (1-based) where the token starts.

Examples (see more examples below):

```
(Tok:  3 line= 1 str= "65" int= 65)
(Tok:  7 line= 1 str= ";")
```

The first example shows a token with ID #3 on line #1, its an int token. The second example shows a semicolon token with ID #7 on line #1.

Development Note: For teams, pick small functions/methods and agree on a small quick API. Create sample expected input output (EIO) that represents a valid use of the API. Make sure your small pieces work according to that API. Don't assume that your partners will get things correct. Check back with them at least every other day.

Submission

Your submission must, at a minimum, include a plain ASCII text file called **`README.txt`**, all necessary source files, (and if you use them: libraries, and build configuration files) to allow the submission to be built and run independently by the instructor. [For this project, no unusual files are expected.]

All files must include a comment header identifying the author(s), author contact information, and a brief description of the files submitted. The **`README`** file must describe any bugs and features along with a description of what was or was not completed in the solution. The **`README`** must also describe the use of the program (and if there are any external dependencies). Finally, for a pair team, the **`README`** must indicate which functions/methods were authored by whom.

Do not include any object files, binary executables, or other superfluous files.

Place your submission files (e.g., the **`README`** and .jar files) in a **folder named** `444-p1_teamID`. For example, if your team name were KAZ, then your folder would be named `444-p1_KAZ`. Then zip up this folder. Name the .zip file the **same as the folder name**.

Turn in by 11pm on the due date (in the bulletin board post) by **sending me email** with the zip file attached. [NB, If your emailer will not email a .zip file, then change the file extension from .zip to .zap, and tell me so in the email.] The email subject title should also include **the folder name**. Please include your name(s) and campus ID(s) at the end of the email (because some email addresses don't make this clear).

Grading

- 75% for compiling and correctly executing with no errors or warnings
- 15% for clean and well-documented code
- 5% for a clean and reasonable **`README`** file
- 5% for correctly following the submission guidelines (below)

```
prog main { print( "ASCII:", " A= ", 65, " Z= ", 90 ); }
```

A4-sample-1.alex format // .alex is the extension for an A4 lexer token file.

```
(Tok: 10 line= 1 str= "prog")
(Tok: 11 line= 1 str= "main")
(Tok: 33 line= 1 str= "{")
(Tok: 23 line= 1 str= "print")
(Tok: 37 line= 1 str= "(")
(Tok:  5 line= 1 str= "ASCII:")
(Tok:  6 line= 1 str= ",")
(Tok:  5 line= 1 str= " A= ")
(Tok:  6 line= 1 str= ",")
(Tok:  3 line= 1 str= "65" int= 65)
(Tok:  6 line= 1 str= ",")
(Tok:  5 line= 1 str= " Z= ")
(Tok:  6 line= 1 str= ",")
(Tok:  3 line= 1 str= "90" int= 90)
(Tok: 38 line= 1 str= ")")
(Tok:  7 line= 1 str= ";")
(Tok: 34 line= 1 str= "}")
(Tok:  0 line= 1 str= "")
```

EX #2: A4-sample-2.acod program

```
prog main { // Find the circumference of a circle.
  pi = 3.14;
  print( "Input radius> " );
  rx = input ( float );
  circum = 2 * pi * rx;
  print( "Circumf= ", circum );
}
```

A4-sample-2.alex format

```
(Tok: 10 line= 1 str= "prog")
(Tok: 11 line= 1 str= "main")
(Tok: 33 line= 1 str= "{")
(Tok:  2 line= 2 str= "pi")
(Tok: 45 line= 2 str= "=")
(Tok:  4 line= 2 str= "3.14" float= 3.14)
(Tok:  7 line= 2 str= ";")
(Tok: 23 line= 3 str= "print")
(Tok: 37 line= 3 str= "(")
(Tok:  5 line= 3 str= "Input radius> ")
(Tok: 38 line= 3 str= ")")
(Tok:  7 line= 3 str= ";")
(Tok:  2 line= 4 str= "rx")
(Tok: 45 line= 4 str= "=")
(Tok: 22 line= 4 str= "input")
(Tok: 37 line= 4 str= "(")
(Tok: 15 line= 4 str= "float")
(Tok: 38 line= 4 str= ")")
(Tok:  7 line= 4 str= ";")
(Tok:  2 line= 5 str= "circum")
(Tok: 45 line= 5 str= "=")
(Tok:  3 line= 5 str= "2" int= 2)
(Tok: 41 line= 5 str= "*")
(Tok:  2 line= 5 str= "pi")
(Tok: 41 line= 5 str= "*")
(Tok:  2 line= 5 str= "rx")
(Tok:  7 line= 5 str= ";")
(Tok: 23 line= 6 str= "print")
(Tok: 37 line= 6 str= "(")
(Tok:  5 line= 6 str= "Circumf= ")
(Tok:  6 line= 6 str= ",")
(Tok:  2 line= 6 str= "circum")
(Tok: 38 line= 6 str= ")")
(Tok:  7 line= 6 str= ";")
(Tok: 34 line= 7 str= "}")
(Tok:  0 line= 7 str= "")
```

EX #3: A4-sample-3.acod program

```
prog main { // Find the hypotenuse of a right triangle.
  print( "Input legs> " );
  a = input( int );
  b = input( int );
  print( "Hypotenuse= ", ( a * a + b * b ) ^ 0.5 );
}
```

A4-sample-3.alex format

```
(Tok: 10 line= 1 str= "prog")
(Tok: 11 line= 1 str= "main")
(Tok: 33 line= 1 str= "{")
(Tok:  2 line= 2 str= "print")
(Tok: 37 line= 2 str= "(")
(Tok:  5 line= 2 str= "Input legs> ")
(Tok: 38 line= 2 str= ")")
(Tok:  7 line= 2 str= ";")
(Tok:  2 line= 3 str= "a")
(Tok: 45 line= 3 str= "=")
(Tok: 22 line= 3 str= "input")
(Tok: 37 line= 3 str= "(")
(Tok: 15 line= 3 str= "int")
(Tok: 38 line= 3 str= ")")
(Tok:  7 line= 3 str= ";")
(Tok:  2 line= 4 str= "b")
(Tok: 45 line= 4 str= "=")
(Tok: 22 line= 4 str= "input")
(Tok: 37 line= 4 str= "(")
(Tok: 16 line= 4 str= "int")
(Tok: 38 line= 4 str= ")")
(Tok:  7 line= 4 str= ";")
(Tok: 23 line= 5 str= "print")
(Tok: 37 line= 5 str= "(")
(Tok:  5 line= 5 str= "Hypotenuse= ")
(Tok:  6 line= 5 str= ",")
(Tok: 37 line= 5 str= "(")
(Tok:  2 line= 5 str= "a")
(Tok: 41 line= 5 str= "*")
(Tok:  2 line= 5 str= "a")
(Tok: 47 line= 5 str= "+")
(Tok:  2 line= 5 str= "b")
(Tok: 41 line= 5 str= "*")
(Tok:  2 line= 5 str= "b")
(Tok: 38 line= 5 str= ")")
(Tok: 42 line= 5 str= "^")
(Tok:  4 line= 5 str= "0.5" float= 0.5)
(Tok: 38 line= 5 str= ")")
(Tok:  7 line= 5 str= ";")
(Tok: 34 line= 6 str= "}")
(Tok:  0 line= 6 str= "")
```

Error Handling

Use Rule #0, (which is also reflected in the use-case main scenario: No Frills), to get a working version early. If you detect a mistake in A4 source program input, then output an error message and abort the lexer program.

Testing

Test that your lexer works on the above three programs. Also, create two further A4 "programs" of your own, to see if it works on yours. Add the pairs of tested A4 source input and A4 token output files to your test directory folder and include that folder in your submission .zip file.

Readme File

You should provide a README.txt text file. Be clear in your instruction on how to build and use the project by providing instructions a novice programmer would understand. If there are any external dependencies for building, the README must also list them and how to find and incorporate them. Usage should include an example invocation. A README would cover the following:

- Class number
- Project number and name
- Team name and members
- Intro (including the algorithm used)
- Contents: Files in the .zip submission
- External Requirements (None?)
- Setup and Installation (if any)
- Sample invocation & results to see
- Features (both included and missing)
- Bugs (if any)

444 Compilers — Project #1 — Lexer

Academic Rules

Correctly and properly attribute all third party material and references, lest points be taken off.

Submission

All Necessary Files: Your submission must, at a minimum, include a plain ASCII text file called **`README.txt`**, all necessary source files to allow the submission to be built and run independently by the instructor. [For this project, no unusual files are expected.] Note, the instructor not use use your IDE or O.S.

Headers: All source code files must include a comment header identifying the author, author's contact info (please, no phone numbers), and a brief description of the file.

No Binaries: Do not include any IDE-specific files, object files, binary **executables**, or other superfluous files.

Project Folder: Place your submission files in a **folder named** `444-p1_teamname`. For example, if your team name is ABC then name the folder `"444-p1_ABC"`.

Project Zip File: Then zip up this folder. Name the .zip file the **same as the folder name**. (Note, I spend a bit of time renaming wayward project files, and I'd rather not have to.) Turn in by 11pm on the due date (as specified in the bulletin-board post) by **sending me email** (see the Syllabus for the correct email address) **with the zip file attached**. The email subject title should include **the folder name**.

ZAP file: If your emailer will not email a .**zip** file, then change the file extension **from .zip to .zap**, attach that, and tell me so in the email.

Email Body: Please include your team members' names and campus IDs at the end of the email.

Project Problems: If there is a problem with your project, don't put the problem description in the email body – put it in the README.txt file.

The Cloud: Do not provide a link to Dropbox, Gdrive, or other cloud storage. Note, cloud files (e.g., G-drive) are not accepted.

Grading

- 75% for compiling and executing correctly with no errors or warnings
- 10% for clean and well-documented code (Rule #4)
- 10% for a clean and reasonable **`README`** file
- 5% for successfully following Submission rules