## CMPE 152 Programming Project (Part I: Lexical Analyzer)

## Project Introduction

In this project, you will write a lexical analyzer to recognize a given set of tokens for a programming language that we will introduce later.

## Due date

The assignment is due 11:59 pm PST on Sep. 27th

## Token Definition

Token speciﬁcation in the below table (Table 1) defines the tokens that must be recognized by your lexical analyzer. The first 18 tokens in the table are multi-symbol tokens in that each of them consists multiple characters. For example, ID (token # 10 in the table) is a type of multi-symbol token that represents an identiﬁer, which is a sequence of (upper or lower case) letters or digits, beginning with a letter. ID is case sensitive and thus upper and lower cases are not the same. Some other tokens represent one or more reserved keywords in the target programming language, and those include BASE\_TYPE (represents *int*, *float*, or *bool*), BREAK (represents *break*), DO, ELSE, etc. All multi-symbol tokens are separated from other multi- or single-symbol tokens by delimiters such as spaces, tabs, and newlines. Tokens numbered 19 ~ 30 in the table are single-symbol tokens. We do not provide names for those tokens, and you may consider that they don’t need to be named or they are named using their literal values.

|  |  |  |
| --- | --- | --- |
| # | Token Name | Symbolic Pattern |
| 1 | AND | && |
| 2 | BASE\_TYPE | int|float|bool |
| 3 | BREAK | break |
| 4 | DO | do |
| 5 | ELSE | else |
| 6 | EQ | == |
| 7 | FALSE | false |
| 8 | FOR | for |
| 9 | GE | >= |
| 10 | ID | [A-Za-z][A-Za-z0-9\_]\* |
| 11 | IF | if |
| 12 | LE | <= |
| 13 | NE | != |
| 14 | NUM | [0-9]+ |
| 15 | OR | || |
| 16 | REAL | [0-9]+.[0-9]+ |
| 17 | TRUE | true |
| 18 | WHILE | while |
| 19 |  | ; |
| 20 |  | = |
| 21 |  | < |
| 22 |  | > |
| 23 |  | ( |
| 24 |  | ) |
| 25 |  | { |
| 26 |  | } |
| 27 |  | + |
| 28 |  | - |
| 29 |  | \* |
| 30 |  | / |
| 31 | EOF |  |

Table 1

Note that in addition to the “visible” tokens (# 1 ~ 30) defined in the table, you need to define an **EOF** token to indicate the end of input is reached.

## Token Implementation

Implementing each token using one dedicated class would result in too many classes. Instead, you may implement one base class Token and use an integer (or a String) field to denote which kind of token a Token object is. In addition, you may want to use a field to keep track of the “value” of each token. There are two possible ways to achieve this:

* The first approach is to use another String field (e.g., lexeme) in your Token class to record the actual string content of a token. For example, a REAL token that represents the real number *2.56* has lexeme set to “*2.56*”. As another example, an ID token that represents a variable *tmp* has its lexeme field equal to “*tmp*”. The benefit of using this approach is you only need one Token class to represent all type of tokens, which may have actual values of different types (a string, a real number, or an integer). The downside is that you need to convert the string value to its actual type when you need.
* An alternative approach is to use three subclasses that inherit the Token base class, to represent three possible value types for tokens in our target language. This way each subclass has value field of an appropriate type for corresponding tokens. In particular, the subclass to represent REAL token needs a float value field, the subclass for NUM token needs an int value field, and the subclass for the rest of the tokens (those for reserved keywords, operators, etc.) can use a string value.

## Lexer Implementation

Based on the token classes, implement your *Lexer* class, in which you provide a public method (e.g., *getNextToken()*) to scan the input and return the next token. Your processing logic should be aware of delimiters (spaces, tabs, and newlines) and skip them when necessary. In your scanning logic, you should NOT use a regular expression or similar high level string manipulation package. You are encouraged to process the input characters one by one. You may use preliminary character and input processing libraries and utilities in Java or other language you choose.

Upon a valid token as defined in the table, your *getNextToken()* method should return a corresponding token object. The method should return null or a special token (indicating an error) when the input cannot be recognized.

For certain types of tokens (reserved keywords and operators), the token values are always the same for all objects of that token. In these cases, it is efficient to predefine static and final objects for those tokens, or put those token objects into a table. When a token of such type is encountered while processing the input, we can return the predefined object or lookup in the table to avoid creating and returning multiple Token objects needlessly.

You should implement all of your token related classes as well as the *Lexer* class into one package.

## Testing & Output

To test your lexer, you need to write a driver class (in another package) with a main function to invoke your lexer. Below is an example:

public static void main(String[] args) throw Exception {

Lexer mylexer = new Lexer();

while(true) {

Token tok = mylexer.getNextToken();

if(tok == null) {

// print error message

break;

} else if (tok.tag == EOF) {

break;

} else {

// print token name and value (if your token doesn’t have a name, add logic to print value twice)

}

}

}

Test your lexer with the below input:

{

int b; b = 1;

{

int a; a = 2;

do a = a+1; while(a < 100);

}

}

The expected output should be:

{ {

BASE\_TYPE int

ID b

; ;

ID b

= =

NUM 1

; ;

{ {

BASE\_TYPE int

ID a

; ;

ID a

= =

NUM 2

; ;

DO do

ID a

= =

ID a

+ +

NUM 1

; ;

WHILE while

( (

ID a

< <

NUM 100

) )

; ;

} }

} }

You may want to test your program with more inputs before submission.

## Project Submission:

Submit the below files before the project is due:

1. All of your source files
2. Sample test files and its corresponding outputs
3. A readme file that describes how to build and run your driver that invokes your lexer

Organize all of the above files in a folder hierarchy and zip the top level folder as one single zip file named as such: project1-team-name.zip

Submit into Canvas: Project #1. Lexical Analyzer

## Project Grading:

90% of your score is determined by:

* Whether your lexer can handle ID, NUM, and REAL tokens correctly (20%)
* EOF Handling (15%)
* Handling of other tokens in Table 1 (20%)
* Whether your lexer has basic error handling (15%)
* Output required format (20%)

The rest 10% depends on the impression on the code quality and documentation (e.g., comments and readme).

Note that if you use a library to handle string patterns rather than scanning characters one by one, you lose 50% of the score.