Lexer Documentation

I. Implementation Choice:

We have chosen the JLex scanner generator for various reasons :

- Our team members are more familiar with Java, and the in-depth explanation provided about JLex at the end of the project handout also encouraged us to opt for that option.
- We refrained from working in C and using a regular expression library with it because we did not know how to integrate the library (header file) in our code although we followed some tutorials.
- As for Python, none of the team members has a heavy background about the language.

II. Lexer Description:

Our implementation with JLex consists of the following elements:

1. Lexer:

This file contains the Lexer specifications. It declares a function **getNextToken()** that uses the regular expressions to perform an action. The action can either be: returning an instance of the class Token, or throwing a **Lexical Error Exception** when the token is not identified. In the first case, when the Token object is created, three parameters are passed to the constructor. The first parameter is the token id, it is retrieved from the class **sym.java**, the second parameter is the value of the token, which is retrieved from the method yytext() that is predefined in JLex. The third and last parameter is the line at which the token was found, this parameter is yyline, which is a predefined attribute in JLex.

In the lexer specification, the **main()** method is defined. Here are the essential components of the main() method:

- **inp**: an instance of the class FileInputStream. It is the file containing the input to the lexer, which is the source code written in our programming language.
- **out**: a PrintWriter instance that refers to the token stream generated by the lexer (Token Stream.txt).
- **s**: a PrintWriter instance that refers to the Symbol Table generated by the lexer (Symbol Table.txt).

- **lit**: a PrintWriter instance that refers to the Literal Table generated by the lexer (Literal Table.txt).

In the main() method, we have within a try{}catch() block the call to the function getNextToken() that returns the tokenized input. The received token is then verified: if it is an identifier **(that has not been declared before)** or a data type, it should be placed in the symbol table; otherwise, if it is a numeric, string, or character literal, it gets added to the literal table. Then the token info is placed in the output token stream. In case the token was not recognized, the lexical error exception is caught, and the program is terminated. Once reaching the end of the input, the symbol table and the literal table are copied to output files.

2. Lexer.java:

This is a java class that gets generated when compiling the lexer specification using JLex. This class is what contains the whole lexical analyzer.

3. Token.java:

This is the class that defines tokens. It has as attributes for the token, its ID, its value, and the line at which the token occurred in the input file.

4. sym.java:

This class assigns to each token an ID that can be used when creating the object Token.

5. LexicalErrorException.java:

This exception class is used to report any lexical analysis error that can occur in the input file. When this exception is caught, the following message gets displayed in the token stream: `ERROR: Invalid input at line` … ,and the execution is terminated.

Our lexer takes the source code as input file, and generates three output files:

1. Symbol Table.txt

   Has the following format for variables:

   ```
   data_type user_identifier
   ```

   and the following format for scopes:

   ```
   %scope_name
   ```

   Example: (sum here is the name of a function)

   ```
    %sum
   entier var
   %Main

   car c
   entier number
   ```

2. Literal Table.txt

   Has the following format:

   ```
   Literal_type literal_value
   ```

   Example:

   ```
   Numeric Literal 10
   Character Literal 'a'
   String Literal "Hello"
   ```

3. Token Stream.txt

   Has the following format:

   ```
   Line line_number Token toenail: token_value
   ```

   Example:

   ```
   Line 1 Token 1: demarre
   Line 2 Token 3: debut
   Line 3 Token 8: entier
   Line 3 Token 34: *var
   Line 3 Token 23: ,
   Line 3 Token 34: b
   Line 3 Token 25: :
   Line 3 Token 37: -8
   Line 3 Token 24: .
   Line 4 Token 4: fin
   ```

III. Lexer User Manual:

After unzipping the submitted file, please follow these steps to run the lexer:

First: Please open the folder "Input and Output files", then move all the text files in it to your computer's home directory

(for Windows OS: C:\Users\{current_user_name})

(for Mac OS: /Users/{current_user_name})

Open your Java IDE (we use NetBeans) and create a new project, you can name it "2As2Bs Lexer". Then copy to the <default package> the classes stored in the folder "Lexical Analyzer Classes" (Token.java; sym.java; LexicalErrorException.java; Lexer.java).

Now, you can run the class Lexer.java. Then, please go back to your home directory and check the output files to see the token stream, symbol table and the literal table generated.

N.B. The reason we did not generate a zip java project is that it did not work for all the team members since they had different NetBeans versions, so we chose to opt for the safest option that works for all versions.

IV. Requirements Check:

Elements	Example	Lexer
Constants	- Numeric literals : 10, -34... - Character Literals : ‘a’, ‘b’... - String Literals : “hello”...	- Reads (\+|-)?[0-9]+ and generates the token *tNumLiteral.* - Reads **'.'** and generates the token *tCharLiteral.* - Reads \".*\" and generates the token *tStringLiteral*
Types	- Integers : entier - Characters : car - Adresses : entier/car	The regular expressions of the datatypes are put at the beginning of the Lexer so that it does not consider them user-defined idenfiers or other language elements. The Lexer treats them as the following : - Reads *entier* and generates the token *tEntier*. - Reads *car* and generates the token *tCar*.
Variables	- User-defined Identifiers : sum, index, number... - User-defined Identifiers as Addresses : *sum, *index,*number...	**Its Regex is put <u>after</u> the Regexs of reserved words** so that reserved words do not get tokenized as user-defined identifiers. The Lexer treats user-defined identifiers it as follows: - Reads *?[a-zA-Z]+[0-9a-zA-Z]* and generates the token *tIdentifier*
Functions	*Function Definition format*: fonction function_name(parameters) renvoie datatype debut...fin	- Reads *fonction* as a reserved word and generates its token *tFonction.* - Reads function_name and treats it as an identifier, [a-zA-Z]+[0-9a-zA-Z]* and generates the token *tIdentifier* - Reads parameters as datatypes + user-defined identifiers, then generates their correspondant tokens. - Reads *renvoie* as a reserved word and generates the token *tRenvoie.* - Reads the datatype that the function returns and generates its correspondant token. - Reads *debut* which indicates the start of the function definition as a reserved word then generates the token *tDebut* - After tokenizing the body of the

		function, it reads *fin* which indicates the end of the function definition as a reserved word then generates the token *tFin*
Operators	- Assignment op: : - Arithemic ops: +,*,-, / - Logical ops : pas, et, ou - Relational ops : <, >, =, ?	- Reads : and generates *tColon* - Reads +,*,-, or / and generates *tPlus, tStar, tMinus*, or *tSlash* - Reads pas, et, or ou and generates *tPas*, *tEt*, or *tOu* - Reads <, >,=, or ? And generates *tLessThan*, *tGreaterThan*, *tEquals*, or *tNotEquals*
Expressions	- Arithmetic Expression - Relational Expression - Logical Expression	- Reads first operand which can be either a Numeric literal or User-defined Identifer then returns *tNumLiteral* or *tIdentifier.* Then it reads the arithmetic operator and returns its corresponding token. Then it reads the second operand which can be either a Numeric literal or User-defined Identifer then returns *tNumLiteral* or *tIdentifier.* It continues reading the expression as long as there are more operands and operators. - Same goes for the other types of expressions.
Statements	- Assignment statement - Conditiona statement format: si(...) debut..... fin sinon si (…) debut..... fin sinon debut..... fin - Repitition statement format: tantque(...)debut. . fin	**Assignments** - Reads the regular expressions that represent Num literal or Id then returns their corresponding tokens. - Next, it reads the assignment operator ':' and generates the token *tColon* - Reads Num literal, Id, or an arithmetic Expression then returns their corresponding tokens - Reads the last lexeme : the dot which signifies the end of the assignment statement then returns the token *tDot* **Conditional Statement** - Reads *si* as a reserved word and generates the token *tSi* - Reads '(' and ')' and returns the tokens *tLParen* and t*RParen*

		- Reads *debut*, *fin* as reserved words and returns *tDebut*, and *tFin* **Repitition Statement** - Reads *tantque* as reserved word and generates the token *tTanque* - Reads ‘(’ and ‘)’ and returns the tokens *tLParen* and t*RParen* - Reads *debut*, *fin* as reserved words and returns the tokens *tDebut*, and *tFin*
Program Structure	Main structure format: demarre debut …fin	- Reads *demarre* as a reserved word and generates the token *tDemarre* - Reads *debut*, *fin* as reserved words and returns the tokens *tDebut*, and *tFin*
Comments	Commen format example: ~This is a comment~	- Once reading ‘~’, the lexer ignores the line.