ALGONQUIN COLLEGE

ASSIGNMENT 2 SPECIFICATION[1] - SCANNER

General View

Due Date: prior or on **October 14th 2020 (midnight)**

- **1st week late submission** (until November 21st midnight): 30% off.
- **2nd week late submission** (until October 28th midnight): 50% off.
- **Later:** 100% off.

Earnings: 20% of your course grade (plus 1% bonus)

Development: Activity can be done **individually** or in teams (**only 2 students** allowed).

Purpose: Development of a Scanner, using dynamic definition, RE (Regular Expressions) and FDA (Finite Deterministic Automata) implementation.

- This is an important activity from front-end compiler and it will use several advanced datatypes, as well as function pointer in C coding style, incrementing the concepts used in programming techniques, data types and structures, memory management, and simple file input/output. The activity will use also the **buffer** previously defined by students. This assignment will be also an exercise in "*excessively defensive programming*".
- You are going to write functions that are required to the front-end compiler and should be used by **parser** to identify tokens and will use a dynamic way to recognize tokens using tables and functions. To complete the assignment, you should fulfill some tasks presented later.
- The current version of code requires *Camel Code style*. Use it appropriately.

[1] Adapted from resources developed by Prof. Svillen Ranev (Algonquin College, 2019)

Task 1: RE, TD and TT for PLATYPUS (5 marks)

Do the activity described in **RETDTT_20F** document (check inside **A2_for_students.zip**). In this part you have to write **regular expressions** for AVID (Arithmetic Variable Identifiers), SVID (String Variable Identifiers), IL (Integer Literals). FPL (Float Point Literals), SL (String Literals), design the TD (**Transition Diagram**) and complete the TT (**Transition Table**).

- In this course you will have a gratifying experience to write the front-end of a compiler for a programming language named ***PLATYPUS*** (or, simply "PLATYPUS").
    - The PLATYPUS informal language specification is given in **PlatypusILS_20F** document.
    - The PLATYPUS formal specification (detailing **Grammar** and **BNF**) is given in **PlatypusBNFGR_20F**.
- In order to write a compiler, the informal language specification must be converted to a formal language specification. Since PLATYPUS is a simple, yet complete, programming language it can be described formally with a context-free grammar (**BNF**) notation.
- The PLATYPUS grammar has two parts: a lexical grammar and a syntactic grammar.
    - The **lexical grammar** will define the lexical part of the language: the character set and the input elements such as white space, comments and tokens. In Part 2 of the assignment you will use the lexical grammar to implement a lexical analyzer (scanner).
    - The **syntactic grammar** has the tokens defined by the lexical grammar as its terminal symbols. It defines a set of productions. The productions, starting from the start symbol <program>, describe how a sequence of tokens can form syntactically correct PLATYPUS statements and programs. This part of the grammar will be used to implement a syntax analyzer (parser) in one of the following assignments.
- You will find the complete lexical and syntactical grammar for the PLATYPUS language in the document **PlatypusBNFGR_20F**. Read very carefully the informal language specification (**PlatypusILS_20F**) and then see how the informal language specification has been converted to a formal specification using a BNF grammar notation.
- Part of the Scanner will be implemented using a Deterministic Finite Automaton (DFA) based on a Transition Table (TT) (see **Part 2** of this document).
    - The TT is partially given to you in file **CST8152_RETDTT_20F.doc** (model task). You need to complete it and include in your code (see table.h). To do this, you must:
        - Convert the lexical grammar into RE (Regular Expressions).
        - Then using the regular expressions, you must draw a Transition Diagram (TD).

- Finally, using the Transition Diagram you can create the TT (Transition Table) that must be competed in your **CST8152_RETDTT_20F.doc** and code (table.h).

Task 2: Scanner Implementation (15 marks)

2.1. GENERAL VIEW

In Part 1, you had analyzed the grammar for the PLATYPUS programming language. Now, in Part 2, you are to create a lexical analyzer (**scanner**) for the PLATYPUS programming language.

Note 1: Remembering Scanner

*The scanner reads a source program from a text file and produces a stream of token representations. Actually, the scanner does not need to recognize and produce all the tokens before next phase of the compilation (the parsing) takes action. That is why, in almost all compilers, the scanner is actually a function that recognizes language tokens and produces token representations one at a time when called by the syntax analyzer (the **parser**).*

The scanner reads a source program from a text file and produces a stream of token representations. It does not need to recognize and produce all the tokens before next phase of the compilation (the parsing) takes action. That is why, in almost all compilers, the scanner is actually a function that recognizes language tokens and produces token representations one at a time when called by the syntax analyzer (the parser).

- In your implementation, the input to the lexical analyzer is a source program written in PLATYPUS language and seen as a stream of characters (symbols) loaded into an input buffer.
  - The output of each call to the Scanner is a single Token, to be requested and used, in a later assignment, by the Parser.
  - You need to use a data structure to represent the Token.
  - Your scanner will be a mixture between **token driven scanner** and **transition-table driven** (DFA)
    - In **token-driven scanner** you have to write code for every token recognition.
    - **Transition-table driven scanners** are easy to implement with scanner generators.
  - The token is processed as a separate exceptional case (exception or case driven scanners).
  - They are difficult for modifications and maintenance (but in some cases could be faster and more efficient).

- **Transition-table driven part** of your scanner is to recognize only variable identifiers (including keywords), both arithmetical or string, integer literals (decimal constants), floating-point literals, and string literals.
    - To build transition table for those tokens you have to transform their grammar definitions into regular expressions and create the corresponding transition diagram(s) and transition table.
    - As you already know, **Regular Expressions** are a convenient means of specifying (describing) a set of strings.

2.2. IMPLEMENTATION OVERVIEW

- In Part 2 your task is to write a scanner program (set of functions). Three files are provided for you on Brightspace LMS: **token.h**, **table.h**, and **scanner.c** (see **A2_for_students.zip**). Where required, you have to write a C code that provides the specified functionality. Your scanner program (project) consists of the following components:

testScanner.c – The main function. This program and the test files are provided for you on Brightspace in a separate file (**A2_for_students.zip**).

token.h – Provided **complete**. It contains the declarations and definitions describing different tokens. Do not modify the declarations and the definitions. **Do not add anything to that file**.

table.h – Provided **incomplete**. It contains transition table declarations necessary for the scanner. All of them are incomplete. You must initialize them with proper values. It must also contain the function prototypes for the accepting functions. You are to complete this file. You will find the additional requirements within the file. If you need named constants you can add them to that file.

scanner.c - Provided **incomplete**. It contains a few declarations and definitions necessary for the scanner. You will find the additional requirements within the file.

- The definition of the ***initScanner()*** is complete and you must not modify it. The function performs the initialization of the scanner input buffer and some other scanner components.
- You are to write the function ***processToken()*** which performs the token recognition (*the original idea is from the concept:* **m**atch-**a**-**l**exeme-**a**nd-**r**eturn = "*malar*").
    - It "reads" the lexeme from the input stream (in our case from the input buffer) one character at a time, and returns a token structure any time it finds a token pattern (as defined in the lexical grammar) which matches the lexeme found in the stream of input symbols.

- The token structure contains the **token code** and the **token attribute**.
- The token attribute can be an attribute code, an integer value, a floating-point value (for the floating-point literals), a lexeme (for the variable identifiers and the errors), an offset (for the string literals), an index (for the keywords), or source-end-of file value.
- Remember that:
  - The scanner ignores the white space.
  - The scanner ignores the comments as well. It ignores all the symbols of the comment including line terminator.
- The function consists of **two implementation parts** (see section 2.1 from this document).
  - Part 1: token driven (special case or exception driven) processing.
  - Part 2: transition table driven processing.
- You are to write both parts. The tokens which must be processed one by one (special cases or exceptions) are defined in **table.h**.
- **Note**: You must build the transition table for recognizing the variable identifiers (including keywords), integer literals, floating-point literals, and string literals.

Note 2: Progressive assignment
Remember that you need to use the code previously developed in your buffer:
** **buffer.h**: Completed in Assignment 1. It contains buffer structure declarations, as well as function prototypes for the buffer structure.*
** **buffer.c**: Completed in Assignment 1. It contains the function definitions for the functions written in Assignment 1.*

The scanner is to perform some **rudimentary error handling** – **error detection** and **error recovery**.

- ***Error handling in comments***. If the **comment construct** is not lexically correct (as defined in the grammar), the scanner must return an error token.
  - For example, if the scanner finds the symbol **!** but the symbol is not followed by the symbol **!** it must return an **error token**.
  - The attribute of the comment error token is a **C-type string** containing the **!** symbol and the symbol following **!**.
  - Before returning the error token the scanner **must ignore** all of the symbols of the wrong comment to the end of the line.

- ***Error handling in strings*.** In the case of **illegal strings**, the scanner must return an error token.
  - The erroneous string must be stored as a C-type string in the attribute part of the error token (***not in the string literal table***).
  - If the erroneous string is longer than 20 characters, you must store the first 17 characters only and then append three dots (…) at the end.
- ***Error handling in case of illegal symbols*.** If the scanner finds an **illegal symbol** (out of context or not defined in the language alphabet) it returns an error token with the erroneous symbol stored as a C-type string in the attribute part of the token.
- ***Error handling of runtime errors*.** In a case of **run-time error**, the function must store a non-negative number into the global variable **errorNumber** and return a run-time error token. The error token attribute must be the string "RUN TIME ERROR:".
  - **TIP_1:** It is important to be sure that you are incrementing the line number when you find a new line char in your buffer.
- The definition of the ***getNextState()*** is **complete** and you must not modify it.
- The function ***nextTokenClass()*** is **incomplete** and you must return the **column index** for the column in the transition table that represents a character or character class / type.
  - For example, the representation for letters in the RE (Regular Expression) is L = [a-zA-Z] and must return the "0" because the order of TT (see **CST8152_RETDTT_20F.doc**).
- Additionally, you have to write the definitions of the **accepting functions** and some other functions (see **scanner.c**).
  - Remember that you need to **accept** (recognize) the tokens defined as AVID, SVID, IL, FPL, SL:
    - Token ***aStateFuncAVID***(char* lexeme);
    - Token ***aStateFuncSVID***(char* lexeme);
    - Token ***aStateFuncIL***(char* lexeme);
    - Token ***aStateFuncFPL***(char* lexeme);
    - Token ***aStateFuncSL***(char* lexeme);
  - You must also create the function to set the Error Token (when DFA is not finishing with the correct token recognition):
    - Token ***aStateFuncErr***(char* lexeme);
  - **Note:** You may implement your own functions if needed.

Note 3: Spec violation manipulating Buffer

In the scanner implementation you are not allowed to manipulate directly any of the Buffer structure data members. You must use appropriate functions provided by the buffer implementation. Direct manipulation of data members will be considered an error against the functional specifications and will render your Scanner non-working.

2.3. IMPLEMENTATION STEPS

1. Firstly, be sure your **buffer** (Assignment 1) is working fine (at least for standard tests).
    a. *Problems with buffer will directly affect your next assignments*.
    b. *However, it is possible to start Assignment 2 in parallel, focusing on the Task 1 (models for PLATYPUS).*

- ***IMPORTANT NOTE_1:*** The answer for buffer is not provided and it is required that the student / team has developed it previously.

2. Start Task 1 (**5 marks**):
    a. It is required to answer the questions in **CST8152_RETDTT_20F** *before starting the development* of **Task 2**.
    b. To this, read the language specification – both informal and BNF that you can find respectively on **PlatypusILS_20F** and **PlatypusBNFGR_20F**.
3. Start Task 2 (**15 marks**): Complete all "**TODO**" sections in your files:
    a. On **table.h**:
        i. Define the constants, the elements (transitionTable, stateType, finalStateTable) and headers for functions to be implemented.
        ii. **TODO_01:** Follow the standard and adjust the file header.
        iii. **TODO_02:** Following PLATYPUS spec define constants for EOF (two situations must be considered).
        iv. **TODO_03:** Define constants for Token Errors and illegal state;
        v. **TODO_04:** Define values missing on transitionTable;
        vi. **TODO_05:** Define values for accepting states types;
        vii. **TODO_06:** Define list of acceptable states;

        To do this, consider that the numbers for **ASWR**, **ASNR** and **NOAS** are only "categories" (with distinct labels) for the final states. They will be required to complete the stateType table.

        viii. **TODO_07:** Declare accepting states functions

        To do this, include all the function definitions that return Token when receiving a specific lexeme – they are already started in the buffer.c (for instance, Token FuncName(char lexeme[])), responsible to recognize AVID, SVID, IL, FPL, SL.

ix. **TODO_08:** Define finalStateTable.

x. **TODO_09:** Define the number of Keywords from the language.

xi. **TODO_10:** Define the keywordTable.

b. On **scanner.c**: continue the development:

i. **TODO_01:** Follow the standard and adjust the file header.

ii. **TODO_02:** Follow the standard and adjust all the function headers.

iii. Implement the two parts of processToken(): token driver scanner (where you are detecting some terminals) and transition table driver scanner (where you call the FDA functions.

iv. **TODO_03:** processToken() part 1: Token driven scanner implementation using switch

THE PART 1 OF PROCESS TOKEN IS SUPPOSED IS BASICALLY THE switch CASE WHEN YOU ARE READING CHARS.
A. SEVERAL TIMES, WITH ONE SINGLE CHAR, YOU CAN DECIDE ABOUT THE TOKEN TO BE CLASSIFIED: IT CAN BE DIRECTLY (FOR INSTANCE, '(').
B. BUT IN OTHER CASES, IT IS NECESSARY TO READ A SEQUENCE OF CHARS (EX: '_','O','R','_' TO MATCH WITH THE TOKEN "_OR_"). YOU CAN USE bufferSetMarkPosition TO THIS.
C. YOU MUST ADJUST THE code (IN currentToken).
D. IN SOME CASES, ADDITIONAL INFO (FOR INSTANCE, attribute FIELD) SHOULD BE UPDATED BECAUSE YOU NEED TO SPECIFY WHAT IS THE VALUE FOR UNION THAT YOU ARE USING.
E. REMEMBER TO ADJUST THE LINE (GLOBAL VARIABLE line) WHEN NECESSARY.
F. CONSIDER ERROR SITUATIONS AND USE DEFENSIVE CODE.

v. **TODO_04:** processToken() part 2: Transition driven scanner implementation inside default

THE PART 2 OF PROCESS TOKEN IS SUPPOSED TO HAPPEN IN THE "default" CASE OF THE SWITCH. IN THIS PART, WE WILL USE SEVERAL VARIABLES THAT ARE DECLARED IN THE BEGINNING: (state, lexStart, lexEnd, lexLength, lexemeBuffer) AS WELL AS SOME BUFFER FUNCTIONS THAT WERE NOT USED (YET) IN THE 1st ASSIGNMENT, BUT SHOULD BE IMPLEMENTED.
A. USE THE state TO GET THE NEXT STATE (CALLING getNextState)
B. USE lexStart TO GET THE INITIAL POSITION OF THE LEXEME (USING bufferGetCPostiion)
C. YOU MUST MARK THIS POSITION CALLING bufferSetMarkPosition
D. NOW, IT IS TIME TO CREATE A LOOP THAT WILL STOP UNTIL YOU HAVE FOUND AN ACCEPTABLE STATE: SO USE THE "NOAS" TO REPEAT THE PROCESS:
  D.1. GETTING THE NEXT CHAR
  D.2. GETTING THE NEXT STATE
E. WHEN YOU HAVE FOUND A FINAL STATE, IF IT IS "ASWR", YOU NEED TO RETRACT – CALLING bufferRetract.
F. SET THE lexEnd AND CALCULATE THE LENGTH OF LEXEME (USING lexLength)
G. YOU NEED TO CREATE A TEMPORARY BUFFER TO LEXEME (CALLING bufferCreate in ;f; MODE, USING THE APPROPRIATE SIZE)
H. RESET THE BUFFER (CALLING bufferReset) TO MOVE TO THE SPECIFIC MARK POSITION.
I. COPY THE LEXEME TO lexemeBuffer.
J. CALL THE APPROPRIATE FUNCTION TO RETURN THE TOKEN BY USING finalStateTable AND USING THE lexemeBuffer
K. RETURN THE currentToken
L. CONSIDER ERROR SITUATIONS AND USE DEFENSIVE CODE.

vi. Adjust your nextTokenClass(), to identify the column in the TT.

vii. **TODO_05:** the logic to return the next column in TT

viii. Implement all functions required to recognize the tokens: ***aStateFuncAVID()***, ***aStateFuncSVID()***, ***aStateFuncIL()***, ***aStateFuncFPL()*** and ***aStateFuncSL()***;

ix. **TODO_06:** Implement the method to recognize AVID

```
WHEN CALLED THE FUNCTION MUST
1. CHECK IF THE LEXEME IS A KEYWORD.
   IF YES, IT MUST RETURN A TOKEN WITH THE CORRESPONDING ATTRIBUTE
   FOR THE KEYWORD. THE ATTRIBUTE CODE FOR THE KEYWORD
   IS ITS INDEX IN THE KEYWORD LOOKUP TABLE (kw_table in table.h).
   IF THE LEXEME IS NOT A KEYWORD, GO TO STEP 2.
2. SET a AVID TOKEN.
   IF THE lexeme IS LONGER than VID_LEN (see token.h) CHARACTERS,
   ONLY FIRST VID_LEN CHARACTERS ARE STORED
   INTO THE VARIABLE ATTRIBUTE ARRAY vid_lex[](see token.h) .
   ADD \0 AT THE END TO MAKE A C-type STRING.
```

x. **TODO_07:** Implement the method to recognize SVID

```
WHEN CALLED THE FUNCTION MUST
1. SET a SVID TOKEN.
   IF THE lexeme IS LONGER than VID_LEN characters,
   ONLY FIRST VID_LEN-1 CHARACTERS ARE STORED
   INTO THE VARIABLE ATTRIBUTE ARRAY vid_lex[],
   AND THEN THE $ CHARACTER IS APPENDED TO THE NAME.
   ADD \0 AT THE END TO MAKE A C-type STRING.
```

xi. **TODO_08:** Implement the method to recognize IL

```
THE FUNCTION MUST CONVERT THE LEXEME REPRESENTING A DECIMAL CONSTANT
TO A DECIMAL INTEGER VALUE, WHICH IS THE ATTRIBUTE FOR THE TOKEN.
THE VALUE MUST BE IN THE SAME RANGE AS the value of 2-byte integer in C.
IN CASE OF ERROR (OUT OF RANGE) THE FUNCTION MUST RETURN ERROR TOKEN
THE ERROR TOKEN ATTRIBUTE IS  lexeme. IF THE ERROR lexeme IS LONGER
than ERR_LEN characters, ONLY THE FIRST ERR_LEN-3 characters ARE
STORED IN err_lex. THEN THREE DOTS ... ARE ADDED TO THE END OF THE
err_lex C-type string.
BEFORE RETURNING THE FUNCTION MUST SET THE APROPRIATE TOKEN CODE
```

xii. **TODO_09:** Implement the method to recognize FPL

```
THE FUNCTION MUST CONVERT THE LEXEME TO A FLOATING POINT VALUE,
WHICH IS THE ATTRIBUTE FOR THE TOKEN.
THE VALUE MUST BE IN THE SAME RANGE AS the value of 4-byte float in C.
IN CASE OF ERROR (OUT OF RANGE) THE FUNCTION MUST RETURN ERROR TOKEN
THE ERROR TOKEN ATTRIBUTE IS  lexeme. IF THE ERROR lexeme IS LONGER
than ERR_LEN characters, ONLY THE FIRST ERR_LEN-3 characters ARE
STORED IN err_lex. THEN THREE DOTS ... ARE ADDED TO THE END OF THE
err_lex C-type string.
BEFORE RETURNING THE FUNCTION MUST SET THE APROPRIATE TOKEN CODE
```

xiii. **TODO_10:** Implement the method to recognize SL

```
THE FUNCTION MUST STORE THE lexeme PARAMETER CONTENT INTO THE STRING
LITERAL TABLE(str_LTBL)
FIRST THE ATTRIBUTE FOR THE TOKEN MUST BE SET.
THE ATTRIBUTE OF THE STRING TOKEN IS THE OFFSET FROM
THE BEGINNING OF THE str_LTBL char buffer TO THE LOCATION
WHERE THE FIRST CHAR OF THE lexeme CONTENT WILL BE ADDED TO THE BUFFER.
USING buffer add char function. COPY THE lexeme content INTO str_LTBL.
THE OPENING AND CLOSING " MUST BE IGNORED DURING THE COPING PROCESS.
ADD '\0' AT THE END MAKE THE STRING C-type string
IF THE STING lexeme CONTAINS line terminators THE line COUNTER MUST BE
INCTREMENTED.
SET THE STRING TOKEN CODE.
```

xiv. Implement the ***aStateFuncErr()***

xv. **TODO_11:** Implement the method to deal with Error Token

```
THE FUNCTION SETS THE ERROR TOKEN. lexeme[] CONTAINS THE ERROR
THE ATTRIBUTE OF THE ERROR TOKEN IS THE lexeme CONTENT ITSELF
AND IT MUST BE STORED in err_lex. IF THE ERROR lexeme IS LONGER
than ERR_LEN characters, ONLY THE FIRST ERR_LEN-3 characters ARE
STORED IN err_lex. THEN THREE DOTS ... ARE ADDED TO THE END OF THE
err_lex C-type string.
IF THE ERROR lexeme CONTAINS line terminators THE line COUNTER MUST BE
INCTREMENTED.
BEFORE RETURNING THE FUNCTION MUST SET THE APROPRIATE TOKEN CODE
```

xvi. **TODO_12:** Implement the function that checks if a string is a keyword, returning the position in the list.

xvii. If necessary, create additional functions (remember to include definition in table.h).

4. Finally, start testing with the files (see files on **A2_for_students.zip**).:
   a. The sequence suggested to your tests is:
      i. a2empty.pls: Must match with a2empty.sout
      ii. a2r.pls: Must match with a2r.sout
      iii. a2w.pls: Must match with a2w.sout
      iv. a2error.pls: Must match with a2error.sout
   b. **TIP_2:** Create additional scenarios to test your scanner and how the **error handler** is working. For instance, imagine what happens when you are opening files with problems in string, comments, identifiers, numbers, etc.

- **TIP_3: Check all comments included in the files (.h and .c) that you are downloading.**

Submission Details

❖ **Digital Submission:** Here are the general orientation. Any problems, contact your lab professor:

- **Compress** into a **zip** file all the files used in the project (for instance, **buffer.h**, **buffer.c**, **table.h**, **token.h**. **scanner.c**, **testScanner.c** as well as the **files required** in the Marking Sheet for A1 (for instance, the document with **answers from the Part 1** – RE, TD and TT)
  - Any **additional files** related to project- your additional input/output test files if you have any can be included (not required).
- Please check the documentation required (as shown in **CST8152_A2MarkingSheet_20F**):
  - For instance, a **Cover page** and a **Test Plan**. Check the A1 Marking Sheet for it, as well as Submission Standard document.

- The submission must follow the course **submission standards**. You will find the Assignment Submission Standard (**CST8152_ASSAMG_20F)** for the Compilers course on the Brightspace.
- **Upload** the zip file on Brightspace. The file must be submitted prior or on the due date as indicated in the assignment.
- ***IMPORTANT NOTE_2:*** The name of the file must be **Your Last Name** followed by the last three digits of your student number followed by your lab **section number**. For example: **Sousa123_s10.zip**.
  - If you are working in teams, please, include also your partner info. For instance, something like: **Sousa123_Melo456_s10.zip**.
  - **Remember:** Only students from the same section can constitute a specific team.

Note 4: About Teams
*You can submit individually or in teams (2 students only). This team can be different from the previous assignment, but it is required to inform your lab professor previously. In this case, it is required to submit one page detailing who was responsible for which function (as required in the header function). Some methods, such as **processToken()** can have two authors, but NOT all methods. For this reason, it is possible to find different marks for each student even in the same team, when the assignment is evaluated.*

- ***IMPORTANT NOTE_3:*** **Assignments will not be marked if there are not source files in the digital submission.** Assignments could be late, but the lateness will affect negatively your mark: see the Course Outline and the Marking Guide. All assignments must be successfully completed to receive credit for the course, even if the assignments are late.

- **Evaluation Note:** Make your functions as efficient as possible.
  - If your program compiles, runs, and produces correct output files, it will be considered a ***working program***.
  - Additionally, I will try my best to "crash" your functions using a modified main program, which will test all your functions including calling them with "invalid" parameters.
  - I will use also some additional test files (for example, a large file). So, test your code as much as you can!
  - This can lead to fairly big reduction of your assignment mark (see **CST8152_ASSAMG** and **MarkingSheetA2** documents).

- ***IMPORTANT NOTE_4:*** In case of emergency (BS LMS is not working) submit your zip file via e-mail to your **lab professor**.

About Lab Demos

- **Main Idea:** This semester, you can get **bonuses** when you are demonstrating your evolution in labs. The marks are reported in CSI.
  - **Note:** The demo during lab sessions is now required to get marks when you do your lab submissions.

- **How to Proceed:** You need to demonstrate the expected portion of code to your Lab Professor in **private Zoom Sections**.
  - If you are working in teams, you and your partner must do it together, otherwise, only the student that has presented can get the bonus marks.
  - Eventual questions can be posed by the Lab professor for any explanation about the code developed.
  - Each demo is related to a **specific lab** in one specific week. If it is not presented, no marks will be given later (even if the activity has been done).

Finally, another motivation thought that prof **Svillen Ranev** used to share…

"There are two kinds of people, those who do the work and those who take the credit. Try to be in the first group; there is less competition there."

Indira Gandhi

Murphy's laws (1 and 2)
If anything can go wrong, it will.
If there is a possibility of several things going wrong, the one that will cause the most damage will be the first one to go wrong.

File update: Sep 5th 2020 by Paulo Sousa.

- *Changes:*
  - Additional tips in section 2.3 (Implementation steps).
  - Adjusting the variable name "scerrnum" to ***errorNumber*** in error handling strategy on page 6.
  - *Tips (step-by-step to be done in the 1st and 2nd part of* **processToken** *(that deals with the function pointers to accepting functions – see "TODO_03" and "TODO_04" on page 08.*

By: Paulo Sousa.

Good Luck with Assignment 2!