ALGONQUIN COLLEGE

ASSIGNMENT 2.2 – LEXICAL ANALIZER (Scanner)

General View

Due Date: prior or on **Nov 13th 2021 (midnight)**

- **2nd Due date** (until 20h Nov) – NO DEDUCTION this time.

Earnings: 15% of your course grade.

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

Note 1: About Teams
Only teams already defined can continue working. In this case, only one student is required to submit the solution.

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.
- We are progressively implementing the front-end compiler defined for your language (see your **A11**). The activity will also use the **buffer** (**A12**) previously defined by students and uses all elements developed in the models (**A21**). 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** (next assignments activities: **A31** and **A32**) 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.

- ***IMPORTANT NOTE_1:*** This activity follows the specification done in **A21** (Language Model).
  - The tables to be implemented on code must match with your **TT** (Transition Table) that comes from **TD** (Transition Diagram = Automata) which maps the **RE** (Regular Expression) previously defined.
  - The code provided is NOT complete (even without any compilation problem). It means that you need to continue the development in order to obey your specification.
  - The use of any other definition will be considered a plagiarism and will not be accepted.

Task: Scanner Implementation (10 marks)

In short, you can see the example provided in the BS (Brightspace) that can recognize the "Hello World" in SOFIA:

```
@ SOFIA Example
  Using Strings @
&MAIN {
        DATA {
        }
        CODE {
                WRITE("Hello world!");
        }
}
```

Basically, once you execute in SOFIA, you can see this output:

```
[Debug mode: 0]
Reading file a22B_Hello.sof ....Please wait

Printing buffer parameters:

The capacity of the buffer is:  200
The current size of the buffer is:  92

Printing buffer contents:

@ SOFIA Example
  Using Strings @
&MAIN {
      DATA {
      }
      CODE {
```

```
            WRITE("Hello world!");
      }
}

Scanning source file...

Token          Attribute
----------------------------------
MNID_T         &MAIN
LBR_T
KW_T           DATA
LBR_T
RBR_T
KW_T           CODE
LBR_T
KW_T           WRITE
LPR_T
STR_T          0       Hello world!
RPR_T
EOS_T
RBR_T
RBR_T
SEOF_T         0

Printing string table...
----------------------------------
Hello world!
----------------------------------
```

Note that you need to use your own files respecting your language specification (review your A11 definition and eventually, update it).

2.1. GENERAL VIEW

In A21, you had analyzed the grammar for one specific language (for instance, SOFIA). Now, in A22, you need to create a lexical analyzer (**scanner**) for the same programming language that you have defined.

Note 2: A little about Scanner

*The scanner reads a source program from a text file and produces a stream of token representations. The scanner does not need to recognize and produce all the tokens before next phase of the compilation (the parsing) acts. That is why, in almost all compilers, the scanner is 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) acts. That is why, in almost all compilers, the scanner is 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 the language (ex: SOFIA) 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 must 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 must 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

- Your task is to **write a scanner program** (set of functions). The files are provided for you on Brightspace LMS:
    - Scanner files: **MainScanner.c, Scanner.h**, and **Scanner.c**.

Note 3: About other files

Remember that you need to use all previous code developed – using Compilers.h and Compilers.c, but also the Buffer.h and Buffer.c. However, you need to use "mainScanner" instead of "mainBuffer" (adjust the main() function.

- The scanner is responsible to show / print tokens, and, eventually, inform errors with the corresponding lines.
- Your scanner program (project) consists of the following components:
  - startScanner (already done): responsible for basic definitions;
  - tokenizer (to be updated): responsible for basic token classification, divided into 2 parts:
    - Part I: most of "switch-case" logic, when you need to classify the tokens directly;
    - Part II: basic "default" when you are supposed to simulate the automata.
  - nextState (already done): when you return the next state;
  - nextColumn (to be updated): when you return the column according to the TT.
  - *FuncNAME* (already initialized): when you are defining functions to classify the tokens. Ex: funcMID (Method identifier), funcSL (String literal), funcKEY (Keywords) and funcERR (Errors – with or without retract).
  - printToken (already initialized, to be continued): when you print the information from tokens.
- Some constants are used – ex: VID_LEN, where you are defining the size of a length, ERR_LEN, related to length of error messages and NUM_LEN, the size of a number.
- The scanner is to perform some **rudimentary error handling** – **error detection** and **error recovery**.
  - ***Error messages.*** If the erroneous string is longer than ERR_LEN characters, you must store the first ERR_LEN-3 characters only and then append three dots (**…**) at the end.
  - ***Error handling of runtime errors.*** In SOFIA, there is basically one kind of runtime error: when it is not possible to add char in StringLiteralTable. 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:".

The purpose of accepting functions in scanner is to classify the lexemes received.

- Remember that you need to **accept** (recognize) the tokens defined as VID (IVID, FVID or SVID), numerical literals (IL, FPL), keywords (KEY) or strings (SL):
  - Token ***funcMID*** (char* lexeme);
  - Token ***funcSL*** (char* lexeme);

- Token ***funcKEY*** (char* lexeme);
- Token ***funcErr*** (char* lexeme);
- **Note:** You need to implement your own functions for **Final States**.

2.3. IMPLEMENTATION STEPS

1. Firstly, be sure your **buffer** (Assignment 1) is working fine. *Problems with buffer will directly affect your next assignments*.

- ***IMPORTANT NOTE_2:*** The code for buffer is not provided and it is required that the student / team has developed it previously.

2. Change the code to adapt for your language and complete all "**TO_DO**" sections in your files:
   a. On **Scanner.h**:
      i. TO_DO: Define your own list of tokens (enum TOKEN_CLASS):
         - **NOTE 1**: Remember that they must correspond to different tokens from your language (for instance, if all VIDs are using the same expression, you must have just one token for variables).
         - **NOTE 2**: Some classes must be included to obey your specification (TO_DO).
      ii. TO_DO: Adjust your attributes (according to your language) on:
         - TokenType (use your language typedefs):
         - TypeAttributes (use your language typedefs);
      iii. TO_DO: Adjust your values for:
         - TABLE_COLUMNS: The number of columns to be used in the TT.
         - COLUMN_NUMBERS: Adjust eventually the **prefixes** and **suffixes** used in the code.
      iv. **TO_DO**: Define your transitionTable (follow your definitions from last assignment): define the TT.

Example[1] (Sofia code):

```
/*        [A-z], [0-9], _,    &,    ",    SEOF, other  */
/*        L(0),  D(1),  U(2), M(3), Q(4), E(5), O(6)   */
/* S0 */ {5,     7,     7,    1,    3,    8,    7},  /* NOAS */
/* S1 */ {1,     1,     1,    2,    2,    2,    2},  /* NOAS */
/* S2 */ {IS,    IS,    IS,   IS,   IS,   IS,   IS}, /* ASWR (MNID) */
/* S3 */ {3,     3,     3,    3,    4,    8,    3},  /* NOAS */
```

[1] Note that most IL is a label for "Illegal State" (-1). Additionally, most of errors are NOT retracting (state 7), but when you are finding EOF, you can retract.

```
        /* S4 */ {IS,    IS,    IS,   IS,   IS,   IS,   IS}, /* ASNR (SL) */
        /* S5 */ {5,     6,     6,    6,    6,    8,    6},  /* NOAS */
        /* S6 */ {IS,    IS,    IS,   IS,   IS,   IS,   IS}, /* ASWR (KEY) */
        /* S7 */ {IS,    IS,    IS,   IS,   IS,   IS,   IS}, /* ASNR (Err1) */
        /* S8 */ {IS,    IS,    IS,   IS,   IS,   IS,   IS}  /* ASWR (Err2) */
```

v. TO_DO: Define your stateType (follow your definitions from last assignment – last column from TT).

vi. TO_DO: Adjust your accepting functions (remember that all signatures should respect the same definition. Ex:

Example (Sofia code):

```
        Token funcNAME      (sofia_chr lexeme[]);
```

vii. TO_DO: Adjust your finalStateTable (defining the functions to be invoked in each final state).

- NOTE: Your accepting states must match with your TD;
- NOTE: Your Pointer to functions should invoke the functions according to your TD definition.

viii. TO_DO: Adjust your keyword list:

- Define the correct size (in KWT_SIZE)
- Define all keywords from your language using strings (be aware about being CASE-SENSITIVE).

b. On **Scanner.c**:

i. TO_DO: Define your variables according to your datatypes.

ii. TO_DO: Adjust startScanner to your datatypes.

iii. In tokenizer:

- Mandatory: Update / change the **part I**, defining all tokens that are directly detected;
- Eventually, update / change the **part II**, where you are invoking the functions (the "default" code can be ok, but you need to define the corresponding functions.

iv. In nextColumn:

- Observe the correct sequence of all columns to be returned (check your TT).

i. **TO_DO**: Adjust all your accepting functions (similar to "funcNAME"):

- NOTE: Use the final states that you have defined in your diagram.

ii. TO_DO: Create your own functions (remember to include declarations in Scanner.h).

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

Note 4: Language Modification
Your language can be changed – since you are implementing it and because new challenges can appear, one possible solution is to modify the grammar. However, it is required that you can submit the NEW specification (see the document created in the A11) with some notes about reason for these updates[2].

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 (including previous buffer)
    - Any **additional files** related to project- your additional input/output test files.
- Check the A12 Marking Sheet to include all elements:
  - In "Submission Pattern", it is mentioned that you must include:
    - The code with the adaptation for your language.
    - Adaptation of all function definitions (respecting your datatypes and constants).
    - Inputs, standard outputs and error output files.
    - **Test Plan**: The way you can show / describe how to execute your program (ex: script / batch or simply command line arguments).
    - **NOTE**: A batch file is provided (as well as inputs using SOFIA language).
- The submission must follow the course **submission standards**. You will find the Assignment Submission Standard (**CST8152_ASSAMG**) for the Compilers course on the Brightspace.
    - For instance, a **Cover page** and a **Test Plan**. Check the A22 Marking Sheet for it, as well as Submission Standard document.
- **Upload** the zip file on Brightspace. The file must be submitted prior or on the due date as indicated in the assignment.

[2] You are completely free to do modifications, but remember that your input files (required to any compiler implementation) should also reflect these new ideas.

- ***IMPORTANT NOTE_3:*** 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.
- ***IMPORTANT NOTE_4:*** **Assignments will not be marked if there are no source files or input 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, your scripts). So, test your code as much as you can!
- ***IMPORTANT NOTE_5:*** 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).

Appendix – Explaining Functionalities

- ***IMPORTANT NOTE:*** **Here are some ideas about how to recognize literals in SOFIA**. Remember to check your specification about literals.

Example 1: funcIL (Integer Literals)

Suppose that your language is composed by the following RE: "D+". See one example code:

```
Token funcIL(sofia_chr lexeme[]) {
	Token currentToken = { 0 };
	sofia_lng tlong;
	if (lexeme[0] != '\0' && strlen(lexeme) > NUM_LEN) {
		currentToken = (*finalStateTable[ES])(lexeme);
	}
	else {
		tlong = atol(lexeme);
		if (tlong >= 0 && tlong <= SHRT_MAX) {
			currentToken.code = INL_T;
			currentToken.attribute.intValue = (sofia_int)tlong;
		}
		else {
			currentToken = (*finalStateTable[ES])(lexeme);
		}
	}
	return currentToken;
}
```

Brief explanation (about example above[3]):

- Basically, the token is initialized without any information.
- Then, it is evaluated if there is a valid string and if the length is higher than the max size.
- If so, there is a conversion from lexeme and the result is using a datatype beyond the default range (short): it is required because otherwise, casting will be done and the value can be truncated. Doing this, you can check if the value is outside the datatype and error can be treated.
- In this approach, note that there is no negative numbers (only positive are considered). The negative value can be considered as a kind of "unary" operation.
- Finally, if there is an error, the ES (Error State) will be invoked to show the error message.

[3] Note that this example is using SOFIA and the corresponding datatypes and RE for IL (Integer Literals). Remember to adapt it to your own language.

Scanner evaluation

Marking Rubric

Maximum Deduction (%)	Deduction Event
CRICTICAL	**Severe Errors**
Up to 100	Plagiarism detection
Up to 100	Does not compile in the environment (considering **Visual Studio 2019**)
Up to 100	Compile but crashes after launch (fix by debug[4])
PATTERN	**Non-compliance**
Up to 20	Language adaptation (missing elements in the buffer)
Up to 10	Missing files (input files – see examples provided but create yours[5])
Up to 10	Missing script (test plan)
BASIC	**Execution**
Up to 30	Problems with small files (empty, hello, basic numerical)
Up to 30	Common errors (EOF, comments, tokens mismatching, invalid tokens[6])
ADDITIONAL	**Small problems**
1 per each	Warnings
Up to 20	Missing defensive programming
Up to 10	Other minor errors (ex: inconsistency between automata and code)
Up to 10	Bonus: clean code, elegant code, enhancements, discretionary points.
Final Mark	**Formula: 15***((100- ∑ penalties + bonus)/100), max score 15%**.**

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

File update: Oct 25th 2021.

Good Luck with Assignment 2.2!

[4] The better strategy is doing tests with small files and step by step include additional tokens. Ex: "void main() {}".
[5] Different from buffer, where any text file provided could be consider, this time, your inputs must match with your language definition, since you are classifying your tokens. Examples can include text (and SVID), integers and floats (and AVID).
[6] Common examples: test what is happen when you are not ending multi-line comments, and also non-ending strings, incorrect regular expressions, etc.