# ASSIGNMENT 2.2 – LEXICAL ANALIZER (Scanner)

## General View

**Due Date:** prior or on **Nov 6th 2022 (midnight)**

- **2nd Due date** (until **Nov 13th 2022**) - 50% off.

**Earnings:** 15% of your course grade.

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

**Purpose: Development of a Scanner**

- ❖ We are progressively implementing the front-end compiler defined for your language (see your **A11**). The activity will also use the **reader** (**A12**) previously defined by students and uses all elements developed in the models (**A21**). This assignment will be also an exercise in "*defensive programming*".
- ❖ The **Scanner** to be implemented will use **RE** (Regular Expressions) and the **TD** (Transition Table) developed in the **A21** specification.
  - At the same time, it will use several advanced datatypes, as well as *function pointers* in C coding style, incrementing the concepts used in programming techniques, data types and structures, memory management, and simple file input/output.
  - It will be necessary 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.
- ❖ The current version of code requires *Camel Code style*. Use it appropriately.

- ❖ ***IMPORTANT NOTE_1***
  - 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 functional but not complete (additional elements will be necessary to be included). 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 **Boa Language** (**INPUT1_Hello.boa**):

```
# BOA Example 2:
  The program is "lexically" correct
  and should not generate any error #
main& {
        data {
        }
        code {
                print&('Hello world!');
        }
}
```

Basically, once you execute in **Boa Language**, you can see this output:

```
_________________________________
|                                 |
| ....... 'BOA' LANGUAGE ........ |
|                                 |
|     __    __    __    __        |
|    /  \  /  \  /  \  /  \       |
| __/  __\/  __\/  __\/  __\__    |
| _/  /__/  /__/  /__/  /_____|   |
|  \_/ \   / \   / \   / \  \___  |
|       \_/   \_/   \_/   \___o_> |
|                                 |
| .. ALGONQUIN COLLEGE - 2022F .. |
|_________________________________|

[Option 'S': Starting SCANNER ....]

[Debug mode: 0]
Reading file INPUT1_Hello.boa ....Please wait

Printing buffer parameters:

The capacity of the buffer is:  250
The current size of the buffer is:  151

Printing buffer contents:

# BOA Example 2:
  The program is "lexically" correct
  and should not generate any error #
main& {
      data {
      }
```

List of strings from the code

```
      code {
            print&('Hello world!');
      }
}

Scanning source file...

Token                 Attribute
----------------------------------
MNID_T                main&
LBR_T
KW_T                  data
LBR_T
RBR_T
KW_T                  code
LBR_T
MNID_T                print&
LPR_T
STR_T        0        Hello world!
RPR_T
EOS_T
RBR_T
RBR_T
SEOF_T       0

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

Token classification:
- Token type – Value (attribute)

Code printed

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

***Note 1:* About Scanner**

*The scanner reads a source program from a text file (in the **Reader / Buffer**) 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.*

- *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 will be a mixture between:*

    - *In token-driven scanner you must write code for every token recognition.*

    - *In the transition-table driven scanners (using DFA: Deterministic Finite Automaton) are easy to implement with scanner generators.*

- In your implementation, the input to the lexical analyzer is a source program written in the language (ex: **Boa Language**) and seen as a stream of characters (symbols) loaded into an input (**reader**).

- 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.
- 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).

- The **transition-table driven part** of your scanner must recognize only variable identifiers (including keywords), both arithmetical or string, literals (decimal constants / floating-point literals), etc.
  - 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 2:* *About the Project Files***

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

*The code for reader (buffer) is not provided and it is required that the student / team has developed it previously*

- 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 (basically done): responsible for basic definitions;
    - **TODO**: Adjust this code to use your own language definition.
  - tokenizer (incomplete): responsible for basic token classification, divided into 2 parts:
    - **Part I**: most of "switch-case" logic, when you need to classify the tokens directly;
      - **TODO**: All cases of tokens (not described in the RE / automaton) should be recognized here.

- **Part II**: basic "default" when you are supposed to simulate the automaton.
  - **TODO**: Implement the recognizer of RE in this part.
- nextState (partial): moves from one state to another, when reading one char;
- nextClass (partial): when you determine the lexeme class related to one specific char.
- *FuncNAME*[1] (partial): when you are defining functions to classify the tokens.
  - **Methods provided**: funcMID (Method identifier), funcSL (String literal), funcKEY (Keywords) and funcERR (Errors – with or without retract).
- printToken (partial): 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.

***Note 3:* About error handler**

*It is also time to start checking errors. Remember that we have always an output: it can be the correct sequence of messages or errors.*

*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 **Boa**, 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 (some can be used for variables or method names: FNID), keywords (KEY) or strings (SL):
  - Token **funcID** (char* lexeme);
  - Token **funcSL** (char* lexeme);
  - Token **funcKEY** (char* lexeme);
  - Token **funcErr** (char* lexeme);
    - **TODO:** You need to implement functions to **Final States**. For example, variable names (IVID, FVID, SVID) or numerical literals (IL, FPL).

## 2.3. IMPLEMENTATION STEPS

1. Firstly, be sure your **reader** (**A11** and **A12**) is working fine. *Problems with reader will affect your directly next assignments*.

[1] The "NAME" in "funcNAME" is supposed to match with proper tokens – ex: INTG, REAL, etc..

2. Change the code to adapt for your language and complete all "**TODO**" sections in your files:
    a. On **Scanner.h**:
        i. **TODO**: Define your own list of tokens (enum TOKENS):
            - **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. **TODO**: Adjust your attributes (according to your language) on:
            - Token(use your language typedefs);
            - IdAttributes (use your language typedefs).
        iii. **TODO**: Adjust constants - your values for:
            - TABLE_COLUMNS: The number of columns to be used in the TT.
            - Adjust the eventual symbols for PREFIXES / SUFIXES to be used. These values can also be used by your nextClass.
        iv. **TODO**: Define your transitionTable (follow your definitions from last assignment): define the TT.
            - Adjust the values for ES / ER / IR;
            - Adjust (eventually) the special columns to be used in

**Example**[2] (**Boa code**):

```
static boa_intg transitionTable[][TABLE_COLUMNS] = {
  /*   [A-z], [0-9],    _,    &,    ‘, SEOF, other
         L(0),  D(1), U(2), M(3), Q(4), E(5),  O(6) */
      {     1,  ESNR, ESNR, ESNR,    4, ESWR, ESNR}, // S0: NOAS
      {     1,     1,    1,    2, ESNR, ESWR,    3}, // S1: NOAS
      {    FS,    FS,   FS,   FS,   FS,   FS,   FS}, // S2: ASNR (MVID)
      {    FS,    FS,   FS,   FS,   FS,   FS,   FS}, // S3: ASWR (KEY)
      {     4,     4,    4,    4,    5, ESWR,    4}, // S4: NOAS
      {    FS,    FS,   FS,   FS,   FS,   FS,   FS}, // S5: ASNR (SL)
      {    FS,    FS,   FS,   FS,   FS,   FS,   FS}, // S6: ASNR (ES)
      {    FS,    FS,   FS,   FS,   FS,   FS,   FS}  // S7: ASWR (ER)
};
```

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

[2] 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.

```
/* TODO: Define list of acceptable states */
static boa_intg stateType[] = {
	NOFS, /* 00 */
	NOFS, /* 01 */
	FSNR, /* 02 (MID) - Methods */
	FSWR, /* 03 (KEY) */
	NOFS, /* 04 */
	FSNR, /* 05 (SL) */
	FSNR, /* 06 (Err1 - no retract) */
	FSWR  /* 07 (Err2 - retract) */
};
```

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

```
Token funcSL        (boa_char lexeme[]);
Token funcID        (boa_char lexeme[]);
Token funcKEY       (boa_char lexeme[]);
Token funcErr       (boa_char lexeme[]);
```

**Example (Boa code)**:

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

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

- **NOTE**: Your accepting states must match with your TD and functions pointers should invoke the functions according to your TD definition.

```
static PTR_ACCFUN finalStateTable[] = {
	NULL,		/* -    [00] */
	NULL,		/* -    [01] */
	funcID,		/* MNID [02] */
	funcKEY,	/* KEY  [03] */
	NULL,		/* -    [04] */
	funcSL,		/* SL   [05] */
	funcErr,	/* ERR1 [06] */
	funcErr		/* ERR2 [07] */
};
```

viii. **TODO**: 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*).

```
/* TO_DO: Define the number of Keywords from the language */
#define KWT_SIZE 10

/* TO_DO: Define the list of keywords */
static boa_char* keywordTable[KWT_SIZE] = {
	"data",
	"code",
	"int",
	"real",
	"string",
	"if",
	"then",
	"else",
	"while",
	"do"
};
```

ix. **TODO**: Adjust your structure for extra language Attributes:

- Define symbols to be used (ex: INDENT).
- Define attributes in a “languageAttributes” to be used by you later).

```
/* TO_DO: Define the number of Keywords from the language */
/*
 * Scanner attributes to be used (ex: including: intendation data
 */

#define INDENT '\t'  /* Tabulation */

/* TO_DO: Should be used if no symbol table is implemented */
typedef struct languageAttributes {
	boa_char indentationCharType;
	boa_intg indentationCurrentPos;
	/* TO_DO: Include any extra attributes to your scanner (OPTIONAL and FREE) */
} LanguageAttributes;
```

b. On **Scanner.c**:

i. **TODO**: Define your variables according to your datatypes.

ii. **TODO**: 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. **TODO**: Adjust all your accepting functions (similar to “funcNAME”):

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

ii. **TODO**: 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**

*As happened in the previous assignments, your language definition can be changed – since you are implementing it and because new challenges can appear, one possible solution is to modify the grammar. You are completely free to do modifications but remember that your input files (required to any compiler implementation) should also reflect these new ideas.*

## 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 reader)
    - Any **additional files** related to project- your additional input/output test files.
- Do not forget to include all elements:
  - 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 **Boa** language).
- The submission must follow the course **submission standards**. You will find the Assignment Submission Standard (**Submission Standard and Marking Guide** at "Assignments > Standards" section) 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.
  - 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.
  - Assignments will not be marked if there are no source files or input files in the digital submission.
- **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!

- 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

- Here are some ideas about how to recognize literals in **Boa Language**. 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(boa_char lexeme[]) {
	Token currentToken = { 0 };
	boa_long tlong;
	if (lexeme[0] != '\0' && strlen(lexeme) > NUM_LEN) {
		currentToken = (*finalStateTable[ESNR])(lexeme);
	}
	else {
		tlong = atol(lexeme);
		if (tlong >= 0 && tlong <= SHRT_MAX) {
			currentToken.code = INL_T;
			currentToken.attribute.intValue = (boa_intg)tlong;
		}
		else {
			currentToken = (*finalStateTable[ESNR])(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 are 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.

## Scanner evaluation

***Note 5: About Teams***

*Only teams already defined can continue working. In this case, only one student is required to submit the solution.*

### Marking Rubric

| Maximum Deduction (%) | Deduction Event |
|---|---|
| **CRICTICAL** | **Severe Errors** |
| 15 pt (100%) | Plagiarism detection |
| 15 pt (100%) | Does not compile in the environment (considering **Visual Studio 2019/2022**) |
| 7.5 pt (50%) | Compile but crashes after launch (fix by debug[4]) |
| **PATTERN** | **Non-compliance** |
| 7.5 pt (50%) | Language adaptation (missing elements in the scanner) |
| 2 pt | Missing files (input files – see examples provided but create yours[5]) |
| 1 pt | Missing script (test plan) |
| **BASIC** | **Execution** |
| 2 pt | Problems with input files (empty, hello, basic numerical) |
| 1 pt | Common errors (EOF, comments, tokens mismatching, invalid tokens[6]) |
| **ADDITIONAL** | **Small problems** |
| 0.1pt each | Warnings |

[3] Note that this example is using **Boa Language** and the corresponding datatypes and RE for IL (Integer Literals). Remember to adapt it to your own language.
[4] The better strategy is doing tests with small files and step by step include additional tokens. Ex: "void main() {}".
[5] Different from **reader**, 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 – String Variable ID), integers and floats (and AVID – Arithmetic Variable ID).
[6] Common examples: test what is happen when you are not ending multi-line comments, and also non-ending strings, incorrect regular expressions, etc.

| | |
|---|---|
| 1.0 | Missing defensive programming |
| 0.5 | 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%**.** |

**File update**: Oct 17th 2022.

**Good Luck with A22!**