ALGONQUIN COLLEGE

ASSIGNMENT 3.2 – SYNTAX ANALYZER (Parser)

1. General View

Due Date: prior or on **Dec 11th 2021 (midnight)**

- **Note:** Because this is the last week, there is no a "second" due date.

Earnings: 10% of your course grade.

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

Note 1: About Teams
Only teams previously defined can continue working. In this case, only one student is required to submit the solution (complete code, batch file, input files).

Purpose: Development of a Parser, using Scanner previously defined and with FIRST set definition.

- This is the last activity from front-end compiler development: this time, we are simply simulating the execution of a front-end defined by a Grammar (BNF style), and using routines, detect if the sequence of tokens is respecting the defined grammar.
- The code requires all previous assignments (**Buffer** and **Scanner**) and uses basically the tokenizer function and several procedures that should correspond to each kind of non-terminal defined in the grammar.
- 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 **A31** (BNF Grammar and tokens defined in **A22** (Scanner implementation).
    - Each non-terminal function must obey exactly the sequence of tokens that defines one production.

- For example, in **Sofia**, a basic program to print the **HelloWorld** is something like this:

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

- ***IMPORTANT NOTE_2:*** About code demo provided to this assignment:
  - The code is NOT complete (even without any compilation problem). It means it is recognizing only this basic "Hello" program.
  - As shown later, the code requested must be able to recognize variable definitions, variable initialization and arithmetic expressions.
  - The use of any other definition will be considered a plagiarism and will not be accepted.

2. Task: Parser Implementation (10 marks)

Here are the main definitions to be followed when you are implementing the Parser…

2.1. GENERAL VIEW

In **A31**, you created the grammar for one specific language (for instance, "**Sofia**" / "Rio", etc…). Now, in A32, we will create functions to simulate the execution of a code that is written in your language following the grammar rules that you have defined.

Note 2: A little about Parser

*The **parser** (our syntactic analyzer) uses the **buffer** (where you are putting the content of a file) and the **scanner** (where you are able to recognize tokens) and check if the sequence of tokens is obeying the rules that you are defining for this grammar. What you will see is in fact a test if one specific code is obeying the BNF. In order to do this, you will use a top-down predictive parser that takes one token per time – a language in LL(1) model – and detect the rule to be followed, using the **First Set** of the grammar.*

- In your implementation, the **input** to the syntactic analyzer is a source program written in your language (for instance, I am providing **".sof"** files written in **Sofia**). They can be exactly the same input files that you have used in the lexical analyzer (**Scanner**).

Note 3: Grammar update

Your language can be updated to prepare it better for the implementation. But be careful about some effects. For instance, if you are changing the BNF to define a new order for tokens, be sure that your grammar is respecting this.

- The **output** the parser is a sequence of messages that indicates that the file is following the BNF rules. Typically, the process shows the outputs in a kind of pos-fixed sequence.

2.2. IMPLEMENTATION OVERVIEW

2.2.1. BASIC IMPLEMENTATION

- Your task is to **write a parser program** that uses the top-down analysis to check the correct sequence of tokens to be recognized by one specific **non-terminal**.
- Each non-terminal must be implemented as a procedure (void function) that simply print messages for each successful sequence.
  - Parser files: **MainParser.c, Parser.h**, and **Parser.c**.

Note 4: 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** and **Scanner.h** and **Scanner.c**. However, you need to use "MainParser" now in the **main()** function.*

- The parser is responsible to show / print the recognized grammar rules, and, eventually, inform errors with the corresponding lines.
- Your **Parser.h** contains all basic definitions to be used. Check the TO_DO comments:
  - `/* TO_DO: Adjust all datatypes to your language */`
  - `/* TO_DO: Adjust your language name */`
  - `/* TO_DO: Create ALL constants for keywords (sequence given in table.h) */`
  - `/* TO_DO: Place ALL non-terminal function declarations */`
- Your **Parser.c** consists of the following functions that are supposed to be similar to any language:
  - BASIC FUNCTIONS:
    - startParser: that invokes the "program" code;

- matchToken: that checks the main classes of tokens to print their own attributes;
- syncErrorHandler: that implements a basic panic-mode error handler;
- printError: where you print proper messages for any token class to be recognized in the parser.

- GRAMMAR FUNCTIONS:
  - Where you need to define all functions that are supposed to match with non-terminal productions from your grammar, starting from "program()".
    - Each function has the same signature:

      nullLanguageType ***grammarFunctionName*** *()*
    - All functions must also have their signatures defined in the **parser.h**.

2.2.2. NON-TERMINAL FUNCTIONS

It is time to create **ALL functions** that are supposed to map one specific non-terminal. Check one example end continue the development in the other cases…

- Suppose the function called "production()" defined by the following non-terminal:

<production> → TEST { <content> };

The corresponding implementation can be given by (considering "**TEST**" as a keyword):

```
nullDatatype production() {
	matchToken(KW_T, TEST);
	matchToken(LBR_T, NO_ATTR);
	content();
	matchToken(RBR_T, NO_ATTR);
	matchToken(EOS_T, NO_ATTR);
	printf("%s%s\n", STR_LANGNAME, ": production parsed");
}
```

Observe that:

- Each token recognized by the Scanner in this production is supposed to match perfectly using the order.
- In the end of most common productions, a basic message is printed (using standard *printf()* routines).

- Note that this process continues: "content()" will be the next production, and so on…

Basic steps:

- To implement the Parser, you **must use** the grammar created on A31.
- Then, start developing the functions (you can use the **Sofia** code as a model).
- If the production consists of a single production rule (no alternatives), write the corresponding function without using the **FIRST** set. Otherwise, you need to be able to identify what is the option according to the lookahead Token.
- If you use the ***lookahead*** to verify in advance whether to proceed with the production and call the ***printError()*** function.

Note 5: About Minimum Grammar

The minimum grammar that you are supposed to develop is that one that can manipulate inputs / outputs and can recognize float arithmetic expressions. Additional features, such as extra datatypes, additional statements (ex: for loop or switch-case) can be considered for additional marks (bonus).

Part III – How to Test

- **Main Idea:** Create different INPUTS (using the BNF of your language). Consider at least these three scenarios:
  - **Case 1:** Using an empty code: the program should not crash and it is required to see the EOF_T.

Sofia input:

```
```

Sofia output:

```
Sofia: Source file parsed
```

  - **Case 2:** A basic Hello World, using only a print message from your language.

Sofia Input:

```
@ SOFIA Hello @
 &MAIN {
    DATA { }
    CODE {
         WRITE("Hello World!");
     }
}
```

Sofia Output:

```
Sofia: Optional Variable List Declarations parsed
Sofia: Data Session parsed
Sofia: Output variable list parsed
Sofia: Output statement parsed
Sofia: Statement parsed
Sofia: Statements parsed
Sofia: Optional statements parsed
Sofia: Code Session parsed
Sofia: Program parsed
Sofia: Source file parsed
```

- **Case 3:** A basic program that uses expressions.

Sofia Example:

```
@ SOFIA Example (Volume of a sphere) @
&MAIN {
        DATA {
                FLOAT %PI, %r, %Vol;
        }
        CODE {
                %PI=3.14;
                READ(%r);
                %Vol = 4.0 / 3.0 * %PI * (%r * %r * %r);
                WRITE(%Vol);
        }
}
```

Sofia Output:

```
Sofia: Float Variable identifier parsed
Sofia: Float Variable identifier parsed
Sofia: Float Variable identifier parsed
Sofia: Float Var List parsed
Sofia: Float Var List Declaration parsed
Sofia: Variable List Declaration parsed
Sofia: Variable List Declarations parsed
Sofia: Optional Variable List Declarations parsed
Sofia: Data Session parsed
Sofia: Float Variable identifier parsed
Sofia: Primary arithmetic expression parsed
Sofia: Multiplicative arithmetic expression parsed
Sofia: Additive arithmetic expression parsed
Sofia: Arithmetic expression parsed
Sofia: Assignment expression parsed
Sofia: Assignment statement parsed
Sofia: Statement parsed
Sofia: Variable identifier parsed
```

```
Sofia: Variable list parsed
Sofia: Input statement parsed
Sofia: Statement parsed
Sofia: Float Variable identifier parsed
Sofia: Primary arithmetic expression parsed
Sofia: Primary arithmetic expression parsed
Sofia: Primary arithmetic expression parsed
Sofia: Primary arithmetic expression parsed
Sofia: Primary arithmetic expression parsed
Sofia: Primary arithmetic expression parsed
Sofia: Multiplicative arithmetic expression parsed
Sofia: Additive arithmetic expression parsed
Sofia: Arithmetic expression parsed
Sofia: Primary arithmetic expression parsed
Sofia: Multiplicative arithmetic expression parsed
Sofia: Additive arithmetic expression parsed
Sofia: Arithmetic expression parsed
Sofia: Assignment expression parsed
Sofia: Assignment statement parsed
Sofia: Statement parsed
Sofia: Variable identifier parsed
Sofia: Variable list parsed
Sofia: Output variable list parsed
Sofia: Output statement parsed
Sofia: Statement parsed
Sofia: Statements parsed
Sofia: Optional statements parsed
Sofia: Code Session parsed
Sofia: Program parsed
Sofia: Source file parsed
```

- ***IMPORTANT NOTE_3:*** About these outputs:
  - The two first examples (1 and 2) are already working fine in the Parser implementation.
  - However, the last example (3) requires that you need to adjust the grammar to use float point variables and expressions.
  - Finally, remember that your **Buffer** and **Scanner** must be fully functional to let you to get the expected results.

2.2.3. EXTRA MARKS (BONUS)

The code that you are supposed to recognize in your BNF is basically using inputs, outputs and arithmetic expressions.

- Extra marks will be given if you are able to recognize:
  - Additional datatypes (ex: integers and strings, including literals);
  - Complex structures, for instance, conditional (**if-else** clauses) and iteration statements (**do-while** loops).
- Example: see the following **Sofia** code (this is the same code you can see in the end of Lecture Notes):

```
@ SOFIA Example
 The program is "lexically" correct
 and should not generate any error @
 &MAIN {
    DATA {
        INT #i;
        FLOAT %a, %sum008;
        STRING $text;
    }
   CODE {
        %a=+1.2;
        %sum008 = 7.87050 ;
        READ(%a,%sum008);
        #i=0;
        WHILE (#i < 32767 _|_ #i == 32767)DO{
            #i = #i + 2;
            %a=
                %a*#i/0.5 ;
            %sum008 = %sum008 + %a - 1 ;
        };
        IF ($text == "")THEN {
            $text = "prog" ++ "ram";
        }
        ELSE {
            $text = $text ++ "ram";
        };
        WRITE("\* This is a program -:)-<-<-- \*");
        WRITE($text);
        IF ($text <> "program"_&_%sum008==8.0_|_#i>10)THEN {
            WRITE(%sum008);
            WRITE();
        }
        ELSE{};
    }
}
```

Part IV - 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 following rules when you submit:
  - The submission must follow the course **submission standards**. You will find the Assignment Submission Standard (**CST8152_ASSAMG_F21**) for the Compilers course on the Brightspace.
    - The Code Pattern with the adaptation for your language.
    - See also the Integral Submission (including inputs and batch 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).
- **Upload** the zip file on Brightspace. The file must be submitted prior or on the due date as indicated in the assignment.
- ***IMPORTANT NOTE_4:*** 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_5:*** In case of emergency (BS LMS is not working) submit your zip file via e-mail to your **professor**.

- **How to Proceed:** You need to demonstrate your progress to your Professor in **private Zoom Sections** during Lab sessions.
  - 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.

- For this **last** assignment, the demo must be done in before the due date.

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[1])
PATTERN	**Non-compliance**
Up to 20	Language adaptation (missing elements in the parser)
Up to 10	Missing files (input files – see examples provided but create yours[2])
Up to 10	Missing script (test plan)
BASIC	**Execution**
Up to 30	Problems with outputs (correct inputs not recognized)
Up to 20	Small errors (missing outputs for specific non-terminal functions[3])
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 30	**Bonus**: recognition of additional statements, such as if clauses and loops (using relational expressions and logical operators).
Final Mark	**Formula: 10***((100- ∑ penalties + bonus)/100), max score 15%**.**

Final Message

Remember that your language must have a proper grammar (different from **Sofia**). In this final assignment, you are simply simulating the execution of a front-end compiler that can recognize your BNF specification.

File update: Nov 29th 2021.

Good luck with A31!

[1] The better strategy is doing tests with small files and step by step include additional tokens. Ex: "program {}".
[2] Remember that your input files must obey your BNF definition.
[3] Common examples: after productions, no messages are found.