TC3048 Compiler
Project II
Syntax Analysis Phase
(Pascal - -)

I. Introduction

The evaluation of this project's phase is constituted by two parts. The first part is the evaluation of the functioning software, whose weight is 50% of the total evaluation. The other 50% will be formed by the quality correctness of the written report. The evaluation metrics for each part are shown in the following sections. For the evaluation of the functionality of the software, there will be an individual one-to-one session with each student, in which an **oral exam** will be applied to the student, and it will be applied as a **multiplier** of the total evaluation grade.

Table #1 provides a summary of how the evaluation is formed for the project. Specific evaluation metrics for each part of the evaluation are provided in the following pages of this document.

	Student Assists to Final Presentation
Evaluation	Weight
Software	50%
Written Report	50%
Oral Exam	x 100%

Table #1: Evaluation Summary

The syntactical and semantical conventions for the language that will be used to develop the compiler are described on Section II and III respectively. Section IV enumerates the project's deliverables. The evaluation metrics for the parser software are presented on Section V. Section VI describes the Oral Exam part of the evaluation as well as its components. Finally, Section VII presents the evaluation metrics for the written report.

PROJECT'S ACADEMIC INTEGRITY (CHEATING)

All work involved in the construction of the project **MUST** be **ENTIRELY developed** by the individual student. The project **MUST** represent the student's **INTELLECTUAL WORK**.

SOFTWARE REUSE: It is a Software Engineering strategy in which the development process is strongly based on existing software components, libraries, applications or/and algorithms[1].

➜ The student is ***ONLY*** allowed to reuse software that **has been developed by the student him/her self on any activity of the TC-3048 Compiler Design course such as Lab Practices or Class Exercises**.

OPEN-SOURCE SOFTWARE: Software which code is available to the public for review, inspection, modification, enhance and use by anyone with interest and permission[2].

➜ The student is **STRICTLY FORBIDDEN** to use open-source software, code freely available from the internet, or any software part **NOT developed entirely** by the student.

BE AWARE, any violation to the restrictions, stablished in this document for the development of the Project's software components, will be considered an act that attempts against the Institution's Academic Integrity Regulation. Hence, the **student will be accountable** and the corresponding measures and actions will be applied accordingly.

Cheating will generate a value of 0 (ZERO) assigned as the FINAL GRADE for the course.

1. Ian Sommerville, *Software Engineering*, 9th. edition (Addison-Wesley, 2011), 426
2. OpenSource.org, *Coining Open Source,* accessed December 15 2017; available from https://www.opensource.org/history; Internet

II. Pascal - - Language Syntactical Specification

The syntax analyzer or parser is the second phase of the translation process of a compiler for any given programming language. The main purpose of the syntax phase is to get the tokens or valid member strings of the programming language found by the scanner, in order to construct a syntactical tree that verifies the grammatical structure of the token stream.

For this project, the student will have to develop a parser for the ***Pascal - -*** programming language, which is essentially a subset of ***Pascal*** language. A **DRAFT** of the grammatical conventions for the language is proposed as follows:

Syntax Conventions:

The syntax of Pascal - - language is described by the following BNF grammar.

1. *start* → **program ID ;** *declaration_block main_block*
2. *declaration_block* → *vars_block functions_block procedures_block*
3. *main_block* → *compound_statement* **.**
4. *vars_block* → **var** *var_declaration* | ε
5. *var_declaration* → *var_declaration var_list* **:** *type_specifier* **;** | *var_list* **:** *type_specifier* **;**
6. *var_list* → *var_list* **, ID** | **ID**
7. *type_specifier* → *basic_type* | *array_type*
8. *basic_type* → **integer** | **real** | **string**
9. *array_type* → **array [NUMBER . . NUMBER] of** *basic_type*
10. *functions_block* → *functions_block function_declaration* | ε
11. *function_declaration* → **function ID (** *params* **) :** *type_specifier* **;** *local_declarations compound_stmt* **;**
12. *procedures_block* → *procedures_block procedure_declaration* | ε
13. *procedure_declaration* → **procedure ID (** *params* **) ;** *local_declarations compound_stmt* **;**

14. *params* → *param_list* | ε
15. *param_list* → *param_list var_list* **:** *type_specifier* **;** | *var_list* **:** *type_specifier* **;**
16. *local_declarations* → *vars_block* | ε
17. *compound_stmt* → **begin** *statement_list* **end**
18. *statement_list* → *statement_list statement* | *statement*
19. *statement* → *assignment_stmt* | *call _stmt* | *compound_stmt* | *selection_stmt* | *for_stmt* | *repeat_stmt* | *input_stmt* | *output_stmt*
20. *assignment_stmt* → *var* **:=** *arithmetic_expression* **;** | *var* **:= STRING ;**
21. *call_stmt* → *call*
22. *selection_stmt* → **if (** *logic_expression* **) then** *statement* **;** | **if (** *logic_expression* **) then** *statement* **else** *statement* **;**
23. *repeat_stmt* → **repeat** *statement_list* **until (** *logic_expression* **)**
24. *for_stmt* → **for ID := NUMBER to NUMBER do** *statement* **;**
25. *input_stmt* → **readln (** *var_list* **)**
26. *output_stmt* → **writeln (** *output_list* **)**
27. *output_list* → *output_list* **,** *output* | *output*
28. *output* → *arithmetic_expression* | **STRING**
29. *var* → **ID** | **ID [** *arithmetic_expression* **]**
30. *logic_expression* → *arithmetic_expression relop arithmetic_expression*
31. *relop* → **<=** | **<** | **>** | **>=** | **==** | **!=**
32. *arithmetic _expression* → *arithmetic_expression arithmetic_operator arithmetic_expression* | **(***arithmetic_expression* **)** | *var* | *call* | **NUMBER**
33. *arithmetic_operator* → **+** | **-** | ***** | **/**
34. *call* → **ID (** *args* **) ;**
35. *args* → *arg_list* | ε
36. *arg_list* → *arg_list* **,** *arithmetic_expression* | *arithmetic_expression*

III. Semantics Conventions:

Every Pascal-- program has a *heading statement* which is used to name the program, a *declaration part* and an *execution part* **STRICTLY** **in this order**. The source code has the following general program structure:

- **Program Name** ← *heading statement*
- **Global Variables Declarations**
- **Functions Declarations**
- **Procedures Declarations**

declaration part (optional)

- **Main Program Block**
- **Statements and Expressions within each program block**
- **Comments**

execution part

1. Program Name:

This section uses the `program` **keyword** followed by the *name* and by semicolon (;). This section **MUST** be the first statement, and has the following structure:

```
program name of the program ;
```

Restrictions:

- The *name of the program* **MUST** be a valid **identifier** and **MUST** be unique, i.e., it **CAN NOT** be used anywhere else.
- The name of the **source file** **MUST** be the same as the **identifier** used for the *name of the program*.

2. Main Program Block:

Every statement block in Pascal-- is enclosed within the **begin** and **end** **keywords** that ends with semicolon (;). A block can be empty, i.e., there may be no other statements between the **begin** and **end** **keywords** .

After the *declaration part* comes the *execution part* which is the **Main Program Block**. This block **IS NOT** optional, it **MUST** always be present.

Thus, the general structure for the Pascal-- **Main Program Block** is the following:

```
program  name of the program ;

Global Variable Declaration part (optional)
Function Declaration part (optional)
Procedure Declaration part (optional)

begin
......
end.
```

Restrictions:

- In contrast with all the other statement blocks, the **end** **keyword** of the main program block **MUST** end with dot (.)

3. Variable Declarations:

The definition of a variable is set in a block that starts with the **`var`** **keyword**, followed by the *variables definition* as follows:

```
var
variable_1, variable_2 ..... : Variable_Type ;
```

The scope of a variable in a Pascal-- program is the region where a variable is declared and has its existence; beyond this region the variable can not be accessed. There are three locations where variables can be declared:

- *Global variables*, those that are outside of all procedures and functions.
- *Local variables*, those that are inside a procedure or function.
- As a *parameter* of a procedure or function.

Restrictions:

- Each variable **MUST** be a valid **Identifier**.
- The declaration may be a single variable declaration or a list of variables of the same type. If it is a list, each variable is separated by comma (,). At the end of the variable list, a colon (:) **MUST** be followed by the variable type and ended with semicolon (;).

- Variables can be of type: **integer**, **real**, or **string**.
- Variable initialization **IS NOT** allowed during its declaration.
- Variables **MUST** be declared before used.
- **example:**

```
var
   a,b: integer;
   c: real;
   d,e: string;
```

4. Arrays:

Pascal-- provides a data structure called **array**. An **array** can store a fixed-size sequential collection of elements of the same data type. All arrays consist of contiguous memory locations, in which the lowest address corresponds to the first element of the array and the highest address corresponds to the last element.

In order to declare an **array** variable in Pascal--, the variable must be included under the **var** section.

A one-dimensional array starts with the **`array`** **keyword**, followed by an *index-subscript* that indicates the length of the array and which is enclosed between square brackets, followed by the **`of`** **keyword**, the *data-type,* and ends with semicolon (;). The general form of a *one-dimensional* array is the following:

```
var
variable_1: array[ index-subscript ] of data-type;
```

Restrictions:

- Pascal-- allows **only** **one-dimensional** arrays.
- The *index-subscript* can by any integer scalar, e.g., **1..10**, which denotes that the size of the array is 10, with 1 as the index of the first element, and 10 as the index of the last element.
- The *index-subscript* starts with an integer as lower bound index, followed by two sequential dots (..) and an integer as the upper bound index.
- The *data-type* can be any of the basic data types: **`integer`**, **`real`**, or **`string`**.
- The declaration can have more than one variable for the same array.
- Arrays **CAN NOT** be initialized during their declaration.

Accessing Array Elements:

An array element is accessed by indexing the array variable. This is done by placing the index within square brackets after the variable. The index value can be given by any arithmetic expression.

Example:

```
program example;
var
   x: integer;
   a,b: array[1..4] of integer;    { size:4, Index: from 1 to 4 }
   c: array[0..9] of real;         { size:10, Index: from 0 to 9 }
   d,e: array[3..8] of string;     { size:6, Index: from 3 to 8 }

(* main program body *)
begin
   a[1] := 0;
   c[0] := 3.17;
   for x := 3 to 8 do
      d[x] := '  ';
end .
```

5. **Functions**: A function definition consists of a function **header**, local **declarations**, and a function **body**.
   - The function header consist of the **`function` keyword** and a *name* given to the function.
   - The header may optionally have a list of parameters. The list of parameters follows the same syntax as for variable declarations, but without the **`var` keyword**. Parameters can be of any type and can also be arrays. Parameters are passed by value. It is allowed to have no parameters at all.
   - All functions **MUST** return a value, so all functions **MUST** be assigned a type, which is the data type of the value that the function returns.
   - The function body contains a collection of statements that defines what the function does. It **MUST** always be enclosed between the **`begin`** and **`end`** **keywords**. Inside the body there **MUST** be at least one assignment statement of the form **name := expression;** which assigns a value to the function name.

The general form of a function definition is as follows:

```
function name ( argument(s): type1; argument(s): type2; ..... ): function_type ;
Local Variable Declaration part (optional)

begin
   .....
   <statements>
   .....
   name := expression;
end;
```

Calling a Function:

When a program calls a function, program control is transferred to the function called. When the last end statement of the procedure is reached, it returns the control back to the calling program. To call a function, it is only needed to pass the required arguments along with the function name. Because the function returns a value, then it needs to be stored into a variable of the same data type as the function's. Functions can be recursive.

Consideration:

Each argument can be a valid *arithmetic expression*.

Example:

Function that returns the maximum between any two integer numbers.

```
program example;
var
   a, b, ret : integer ;

function max(num1, num2: integer): integer;
(* local variable declaration *)
var
   result: integer;

 (* function's body*)
begin
  if (num1 > num2) then
      result := num1
  else
      result := num2;
   max := result;
end ;

(* main program body *)
begin
   a := 20;
   b := 35;
   ret := max(a, b);
   writeln ( 'Maximum value is : ', ret );

end .
```

6. Procedures:

A procedure definition consists of a function **header**, local **declarations**, and a procedure **body**.

- The procedure header consist of the **procedure keyword** and a *name* given to the procedure.
- The header may optionally have a list of parameters. The list of parameters follows the same syntax as for variable declarations, but without the **var keyword**. Parameters can be of any type and can also be arrays. Parameters are passed by value. It is allowed to have no parameters at all.
- The procedure body contains a collections of statements that defines what the procedure does. It **MUST** always be enclosed between the **begin** and **end keywords**.
- The general form of a procedure definition is as follows:

```
procedure  name ( argument(s): type1; argument(s): type2; ..... ) ;
Local Variable Declaration part (optional)

begin
   statement_1 ;
   statement_2 ;
    .....
    .....
  statement_n ;
end;
```

Calling a Procedure:

When a program calls a procedure, program control is transferred to the procedure called. When the last end statement of the procedure is reached, it returns the control back to the calling program. To call a procedure, it is only needed to pass the required arguments along with the procedure name. Procedures can be recursive

Consideration:

Each argument can be a valid *arithmetic expression*.

Example:

Procedure that finds the minimum among three values.

```
program example;
var
   a, b, c, min : integer ;

procedure findMin(x, y, z: integer);
 (* function's body*)
begin
  if (x < y) then
     min := x
  else
     min := y;
  if (z < min) then
     min := z;
end ;    { end of procedure findMin }

(* main program body *)
begin
   writeln('  Enter three numbers:  ');
   readln(a, b, c);
   findMin(a, b, c);
   writeln('  Minimum:  ', min);

end .
```

7. Compound statement:

The compound statement is a group of valid statements enclosed between the **begin** and **end** **keywords** followed by semicolon (**;**):

```
begin
    statement_1 ;
    statement_2 ;
     .....
     .....
    statement_n ;
end;
```

8. Assignment statement:

The assignment statement has the following syntax:

```
id := expression;
```

```
id[expression] := expression;
```

```
id := string;
```

```
id[expression] := string;
```

- It can start with a simple variable represented by a valid **id** or with an array variable represented by a valid **id** followed by its index which is a valid *arithmetic expression* enclosed by square brackets.
- The assignment operator is **:=**
- The value to be assigned can be any valid value given by an *arithmetic expression* or a **string**. The data type of both, the variable and the value **MUST** be the same.

9. Writeln statement:

The **writeln** statement has the following syntax:

```
writeln( arguments );
```

- The **writeln** statement displays the value of the *arguments* into standard output (computer screen).
- The *arguments* can be any list of valid expressions, or constants of the basic data types, e.g., **integers**, **reals**, or **strings** all separated by commas (**,**).
- If the *arguments* are empty, then a new-line and carriage-return is displayed.
- Example:

```
program example;
var
   a: integer ;

(* main program body *)
begin
   a := 24;
   writeln('  The value of variable a is:  ', a);

end .
```

10. Readln statement:

The **readln** statement has the following syntax:

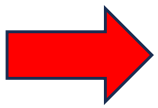
```
readln( arguments );
```

- The **readln** reads from standard input (computer keyboard) and stores them into the *arguments*.
- The *arguments* can be a single variable or any list of variables, separated by comma (**,**) .
- The *arguments* **CAN NOT** be empty, it **MUST** have at least one argument.
- Example:

```
program example;
var
   a: array [1..5] of integer ;
   b: real ;
   c: string ;

(* main program body *)
begin
    writeln('provide an integer, a real and a string:');
    readln(a[1], b, c);
    writeln('a[1] = ', a[1]);
    writeln('b= ', b);
    writeln('c= ', c);

end .
```



```
>
provide an integer, a real and a string:
23  1.5  'hola'
a[1] = 23
b = 1.5
c = 'hola'
```

11. If-then statement:

The If-then statement has the following syntax:

```
if (condition) then statement ;
```

- Where *condition* is a boolean or simple (one relational operator) relational condition, and *statement* can be a simple or compound statement.

12. If-then-else statement:

The If-then-else statement has the following syntax:

```
if (condition) then statement else statement ;
```

- Where *condition* is a boolean or simple (one relational operator) relational condition, and *statement* can be a simple or compound statement.
- Observe that the `then` *statement*, either simple or compound, **DOES NOT** end with semicolon (`;`).

13. for-do statement:

The for-do statement has the following syntax:

```
for ID := initial_value to final_value do statement;
```

- Where **ID** is any **integer** variable, *initial_value* and *final_value* are an **integer** constant, where *initial_value* **MUST** be less than *final_value*.
- *statement* can be a single statement or a compound statement.
- Example:

```
program example;
var
   a: array [1..5] of integer ;
   b: integer ;

(* main program body *)
begin
    for b:=1 to 1 do  a[b]:=b;
    for b:=1 to 1
        begin
            writeln('nunber?: ');
            readln(a[b]);
        end;

end .
```

14. Repeat-until statement:

The repeat-until statement has the following syntax:

```
repeat
   statement_1 ;
   statement_2 ;
   .....
   .....
   statement_n ;
until ( condition ) ;
```

- Where *statement_i* is any simple statement or compound statement, and *condition* is a boolean or simple (one relational operator) relational condition.
- Notice that the conditional expression appears at the end of the loop, so the loop body is executed at least once before the condition is tested.
- If the condition is false, the flow of control jumps back up to the repeat, and the body in the loop executes again. This process repeats until the given condition becomes true.

- Example:

```
program example;
var
   a: array [1..5] of integer ;
   b: integer ;

(* main program body *)
begin
    b:=1;
    repeat
        writeln('nunber?: ');
        readln(a[b]);
        b:=b+1;
    until ( b>5);

end .
```

15. Sample programs in Pascal--:

Example #1:

```
{ Example #1 }
{ This is the typical "Hello World" }

program HelloWorld;

(* This is the main program block *)
begin
     writeLn( ' Hello World ' );
end. (* This is the end of the main
          program block *)
```

Example #2:

```
{ Example #2 }

program Ejemplo2;

var
a: integer;
b: real;

(* This is a procedure block*)
procedure assign (x: integer; y: real);
begin
   a := x;
   b := y;
end;

(* This is the main program block *)
begin
    assign(27, 3.1416);
    writeLn( '  a = ', a );
    writeLn( '  b = ', b );
end. (* This is the end of the main
          program block *)
```

Example #3:

```
{ Example #3 }

program Ejemplo3;

(* Var declaration section*)
var
    a, b : integer;
    x, y : real;
    n : array [1..10] of integer;
    s: string;

function calc (w, z : real) : integer;
begin
   if (w >= z) then
      calc := 5
   else
      calc := 0;
end;

procedure arrayInit (w: integer; z: real);
begin
   for i := 1 to 10 do
       begin
           n[i] := 1 * 5;
           writeLn( 'n[', i, '] =', n[i]);
       end;
end;
```

```
procedure assign (w, z : real);
var
    temp: real;
begin
   temp := w;
   repeat
      temp := temp –z;
   until (temp <=0);
   if (temp = 0) then
      begin
           a := 10;
           b := 20;
       end
    else
       begin
           a := 0;
           b := 0;
        end;
end;

begin
    s := 'The end';
    writeLn( '  x = ' );
    readLn(x);
    writeLn( '  y = ‘ );
    readLn(y);
    if (calc(x,y) = 5) then
        assign(x,y)
    else
        writeLn(s);
end.
```

IV. Project Outcomes:

A. Parser Output:

The Parser **shall** provide the following **outputs**:

1. **Corresponding Syntactical Errors.**
2. **The semantics update of the corresponding Symbol Tables.**

B. Deliverables:

The project **must** include the following **deliverables**:

1. **The correct grammar for Pascal - - in order to generate a Top-Down Predictive Parser. You MUST explain all the steps and decision taken for the generation of the correct grammar.**
2. **The First, follow, and first+ sets.**
3. **Symbol Table management: Description of semantic aspects that were updated during the syntax analysis.**
4. **Error messages generated by the Parser.**
5. **Example of the Parser Outputs**

C. RESTRICTIONS:

1. The parser **CAN NOT** be implemented by using any kind of **Context-Free Grammar Libraries** or **APIs** native to the programming language used to develop the project.
2. The parser **MUST** be implemented by programming the corresponding **Top-Down Predictive** parser **algorithm** as seen in class.

V. Parser Software Evaluation Metrics

	Points						
Aspect	**100**	**80**	**60**	**40**	**20**	**0**	**Weight**
Software Complies with Requirements	1. Software runs. 2. Parser implements the correct grammar for the given language. 3. Parser recognizes invalid syntactical structures and provides the corresponding error message.	Software runs and implements the correct grammar for the given language; however, it does not provide valid error messages for syntax errors.	Software runs and implements the grammar for the given language; however, **it is based on an erroneous grammar**.	Does not apply	Does not apply	Software **does not run**, or failed to implement the correct grammar.	70
Comments in Source Code	1. Syntax analyzer source code is extraordinarily explained and documented 2. All source code files include a description of its functionality and relation with other source code files. 3. All functions/methods and/or classes are clearly and completely described. 4. Comments are unambiguous, complete, and correct with respect to analysis and design.	All source code is commented, however, is hard to understand the meaning and support that the comments provide to each section.	Comments are incomplete, ambiguous, or incorrect.	Does not apply	Extremely poor comments description.	Source code does not include comments.	10
Traceability	1. Every single functional requirement can be mapped directly to a specific piece of code. 2. The code complies with the design. 3. The design can be mapped directly to the implementation	Does not apply	Does not apply	Does not apply	Does not apply	**Poor traceability**.	10
Testing	The Parser software passes all test cases given by the professor.	Does not apply	Does not apply	Does not apply	Does not apply	The parser software does not pass all test cases given by the professor.	10

VI.Oral Exam Evaluation Metrics

	Points						
Aspect	100	80	60	40	20	0	Weight
Software Questions	The student proves that has complete knowledge of the code, and is able to answer any questions from the professor regarding: 1. Functionality. 2. Code. 3. Source files.	Does not apply	The student fails to answer in a correct manner any question.	Does not apply	The student does not prove complete knowledge of the code. However, there is no evidence of cheating.	**Cheating**	60
Software Modifications	The student is able to on-the-fly modify the code with regard to any change or new functional requirement given by the professor during the session.	Does not apply	Does not apply	Does not apply	The student is not able to on-the-fly modify the code. However, there is no evidence of cheating.	**Cheating**	30
Development process.	The student is able to answer any questions from the professor regarding: 1. Functional Requirements. 2. Analysis. 3. Design. 4. Implementation.	Does not apply	Does not apply	Does not apply	The student is not able to answer all the questions. However, there is no evidence of cheating.	**Cheating**	10

Important Remarks

- **The above points are maximum margins. The 100 will obtained if and only if all the items are satisfied in an excellent manner accordingly to the professor criteria.**

VII. Written Report

Once all the previous problem's features are being clearly and concisely formulated and stated as seen during lectures, the following step is to implement the development of the software system project process model. All development process models include in one way or another the following phases: ***Analysis***, ***Design***, ***Implementation***, ***Testing***, and ***Deployment***. The development of the scanner should be based on the IEEE-830 standard.

The structure of report for the scanner must include the following sections:

1. Introduction

1.1.- Summary

Brief description of the contents of this report.

1.2.- Notation

Give a brief description of Context Free Grammars.

- Explain about the model used for the development of the analysis and design phases.
- Justification regarding the selected model.
- Justification of the programming language used for the implementation.

2. Analysis

The analysis model it's a bridge between the system level description that describes overall system's functionalities and the system design. The primary focus of this model is on the ***whats not the hows***. What I/O the system manipulates (data), what functions the system must perform, what are the behaviors that the system exhibit, what interfaces are defined, and what constraints apply. The analysis model shall achieve three primary objectives: 1) describe ***what*** the customer requires, 2) establish the basis for the creation of the software system design, and 3) define a set of requirements that can be validated (tested) once the software system is built.

In this section, the student shall describe the requirements of the system which are represented by the five deliverables for the scanner. It must include all the steps that are required to generate the complete set of formal specifications for them. It must include a concise and precise explanation of every step of the process, by making clear "what is required to do" and "why".

In summary, the analysis section shall include all the steps performed in order to obtain the final working grammar, i.e., you **MUST** provide:

- The final unambiguous, left-recursion-free and left-factor free CGF.
- All the decisions that you made to generate an unambiguous and correct grammar.
- All the processing steps to make the grammar Left-Recursion and Left-Factoring free.
- All the simplifications implemented as well as the justifications for them, regarding *unit productions*, *epsilon productions*, and *useless symbols*.

3. Design

Design and development represent the process of turning the specification (analysis model) into reality (the product). It's an iterative process through which the requirements are translated into a "blueprint" of "how" to construct the system.

There are several characteristics that represent a good design:

- The design must implement all the explicit requirements contained in the analysis model.
- The design must be a readable and understandable guide for the developers and testers.
- The design must provide a complete ***"picture"*** of the system, addressing the data, functional, and behavioral domains from an implementation perspective.
- A design should exhibit an architecture that depicts its modularity, that is, it must show how the system is subdivided into subsystems, how the different requirements are assigned to these subsystems, and which system functionalities are attached to hardware and which to software.

The design **MUST** be conformed by a complete and consistent set of design diagrams (***state*** and ***flow diagrams***, ***module diagrams***, etc). ***Pseudo code*** **MUST** be used to complement state and flow diagrams.

Furthermore, the design model **MUST BE TRACEABLE TO THE ANALYSIS MODEL,** that is, for every functional requirement specified during analysis, the design model shall explicitly describe ***"how"*** this requirement will be implemented. Therefore, the design model becomes the blueprints of how the software system will be implemented. It must be a self sufficient, complete, accurate, consistent, traceable, and maintainable document, whose purpose is to guide and tell the programmer how to develop the code.

The implementation **MUST** be completely based on the design, and traceable to it. The ***"acid test"*** for the design is to consider that if you deliver your design to a completely different developer team, each member of the new team will be able to understand your document and use it to generate the code with out further interaction with you.

In summary,, the design section shall describe how each of the requirements for the the five deliverables of the parser, is implemented. The design **MUST** include:

- The ***First***, ***Follow***, and ***First*** + sets
- The justification of which Top-Down algorithm was selected, ***Recursive Descend*** or ***LL(1)***.
  - For ***Recursive Descend***, provide and explain all the procedures-functions-methods for each non-terminal symbols.
  - For ***LL(1)***, provide the Parsing Table and explain how it was obtained, the data structures required to implement the main components of the parser.
- A preliminary architecture of the complete compiler software shall be provided, by making an special emphasis on the syntax component.

4. Implementation

A complete printout of the source code for the syntax analyzer must be provided. The source code shall be completely and accurately commented.

During and after the implementation process, the system being developed must be checked to ensure that it meets its specification and delivers the functionality expected by the customer. Verification and Validation (V&V) is the name given to these checking processes. V&V starts with requirements reviews and continues through design reviews and hardware and code inspections up to product testing.

5. Verification and Validation

The student must present a Test model. The test model consists of the set of test cases that are developed during the test case design.

During and after the implementation process, the system being developed must be checked to ensure that it meets its specification and delivers the functionality expected by the customer. Verification and Validation (V&V) is the name given to these checking processes. V&V starts with requirements reviews and continues through design reviews and hardware and code inspections up to product testing.

Verification and validation is not the same thing, verification deals with "***are we building the product right?***" while validation deals with "***are we building the right product?***" Verification involves checking that the system conforms to its specification. The developer must check that it meets its specified functional and non-functional requirements. However, validation aims to ensure that the system meets the expectation of the customer. The ultimate goal of the V&V process is to establish confidence that the system is good enough for its intended use.

In order to perform the V&V, the developer must implement a Test Case Design phase. The test cases are part of system and component testing where the developer designs the test cases (inputs and predicted outputs) that test the system. The goal of this phase is to create a set of test cases that are effective in discovering hardware and software defects and showing that the system meets its requirements. To design a test case, the developer must select a particular feature of the system or component that is to be tested. Then, the developer must select a set of inputs that execute that feature, document the corresponding outputs, and check that the actual and expected outputs are the same.

Provide your own set of test files and their expected results. Your implementation MUST pass your test files as well as the professor's test cases. Be sure to include snapshots of the parser's output for your test cases, together with the corresponding explanation.

6. References

Any information that is used to develop this document must be listed on a standard bibliography format.

With respect to **bibliography**, the ***IEEE Reference Style*** **must be used**. This style incorporates common practices of bibliographic references of the scientific and technical fields. This style uses numeric references enclosed on square brackets inserted into the text, whenever the writer needs to link the text to a bibliography entry. The bibliography list must include all the references used on the text. The general structure of an input on the bibliography list is the following:

- **Author** or **authors**, begins with the first name followed by the last.
- **Title**: Every main word starts with a capital letter and all are italic. If the source is not a book or an article, a description of the source must be included.
- **Publisher information**: editor and year.
- **Page numbers**:

In case of articles from scientific journals, the name of the author is followed by the title of the article. The title of the article must be enclosed between quotation marks. Following, the complete name of the journal must be written in italics. Immediately, the volume number as well as the issue number must be included. Finally, the date enclosed in parenthesis, and followed by colon and the pages numbers.

Example of a book entry to the bibliography list:

1. Noam Chomsky and Morris Halle, *The Sound Pattern of English*, (Prentice Hall, 1968), 77-81

Example of a journal article entry to the bibliography list:

2. Keith A. Nelson, R.J Swayne Millar, and Michael D. Fayer, "Optical Generation of Tunable Ultrasonic Waves", *Journal of Applied Physics* 53, no 2 (February 1982): 11-29.

Example of internet references entries to the bibliography list:

3. William J. Mitchel, *City of Bits: Space, Place, and the Infobahn* [book on-line] (Cambridge, Mass: MIT press, 1995, accessed 29 September 1995); available from http://www-mitpress.mit.edu:80/city_of_Bits/Pulling_Glass/Index.html; Internet.
4. Joanne C. Baker and Richard W. Hunstead, "Revealing the effects of Orientation in Composite Quasar Spectra", *Astrophysical Journal* 452, 20 October 1995 [journal on-line]; available from http://www.aas.org/ApJ/v452n2/5309/5309.html; Internet; accessed 29 September 1995.

Example of lecture notes references entry to the bibliography list:

5. R. Castelló, Class Lecture, Topic: "Chapter 2 – Lexical Analysis." TC3048, School of Engineering and Science, ITESM, Chihuahua, Chih, April, 2020.

Example of text's citation/reference:

TEXT:

The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements. No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later. [2]

Bibliography:

1. Noam Chomsky and Morris Halle, *The Sound Pattern of English*, (Prentice Hall, 1968), 77-81
2. Frederick P. Brooks, Jr. *The Mythical Man-Month*, Addison Wesley, 1995.

You can find the complete IEEE Reference Style guide, in the following web pages:

- http://libraryguides.vu.edu.au/ieeereferencing/home
- https://ieeeauthorcenter.ieee.org/wp-content/uploads/IEEE-Reference-Guide.pdf

Writen Report Evaluation Metrics

	Points						
Aspect	100	80	60	40	20	0	Weight
General aspects of the report	1. Title Page. 2. Introduction. 3. Analysis. 4. Design. 5. Testing. 6. Work Plan. 7. References.	Does not apply	Does not apply	Does not apply	Does not apply	Incomplete Document	3
Document Presentation and Format	1. Computer edited. 2. Quality of printing. 3. Page number. 4. Sections and subsections. 5. Figure and Tables MUST have Figure/Table number and a subtitle. 6. Figures and Tables MUST be referenced in the text. 7. Correct distribution of text, figures, and tables. 8. All references must be included in bibliography, using Chicago Manual Style. 9. Professionalism and Quality of Document's Presentation; i.e., delivered in spiral binding or in professional folder.	Does not comply with only one aspect.	Does not comply with only two aspects.	Does not comply with only three aspects.	Does not comply with more than three aspects.	Incomplete Document	2
Orthography and Typography	0 errors	Does not apply	Does not apply	Does not apply	Does not apply	Any Error	10
Introduction	The section is extremely well developed; it is congruent and relevant, including summary and notation.	The section is congruent and relevant, including summary and notation.	The section is poorly developed, but it is complete.	The section is poorly developed and incomplete.	The section is ambiguous, incoherent, and poorly related to the work.	The section does not exist.	2

Aspect	100	80	60	40	20	0	Weight
Analysis	Complete formal specification of the syntax requirements for Pascal-- : 1. The final unambiguous, left-recursion-free and left-factor free CGF. 2. Clear and complete description of all the steps applied to obtain the unambiguous, complete and correct grammar 3. All the processing steps to make the grammar Left-Recursion and Left-Factoring free, 4. All the simplifications implemented as well as the justifications for them, regarding *unit productions*, *epsilon productions*, and *useless symbols*	1. There are minor discrepancies with the specification of the requirements. However, the information provided is sound and it is traceable for the complete development process. 2. The phase has a poor informal description.	1. There are mayor discrepancies with the specification of the requirements. 2. It is very difficult to trace this specification for the complete development process.	NA	NA	Specification is **ambiguous**, or **inconsistent**, or **incomplete**, or **incorrect**.	40
Design	1. Calculation of sets First, Follow, and First+ sets. 2. Justification of which Top-Down algorithm was selected, ***Recursive Descend*** or ***LL(1)***: – For ***Recursive Descend***, provide and explain all then procedure-functions-methods for each non-terminal symbols. – For ***LL(1)***, provide the Parsing Table and explain how it was obtained, the data structures required to implement the main components of the parser. 3. Architectural or modular design.	Traceability to the analysis model is not clear.	NA	NA	NA	Design is **ambiguous**, or **inconsistent**, or **incomplete**, or **incorrect**.	40

Aspect	100	80	60	40	20	0	Weight
Testing	1. Informal description. 2. Software Test Cases design. 3. Justification.	NA	NA.	NA	The section is poorly developed and/or incomplete.	The section does not exist.	3

Important Remarks

- **Everything MUST be completely justified.**
- **The above points are maximum margins. The 100 will obtained if and only if all the items are satisfied in an excellent manner accordingly to the professor criteria.**