TC3048 Compiler
Project II
Syntax Analysis Phase

Compiler’s Project - Syntax Analysis Phase

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 conventions for the language that will be used to develop the compiler are described on Section II. The evaluation metrics for the parser software are presented on Section III. Section IV describes the Oral Exam part of the evaluation as well as its components. Finally, Section IV 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. C-- 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 ***C-Minus*** programming language, which is essentially a subset of ***C*** language. The grammatical conventions for the language are the following:

A. Syntax Conventions:

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

1. *program* → *declaration_list*
2. *declaration_list* → *declaration_list declaration* | *declaration*
3. *declaration* → *var_declaration* | *fun_declaration*
4. *var_declaration* → *type_specifier* **ID ;** | *type_specifier* **ID [NUMBER] ;**
5. *type_specifier* → **int** | **float** | **string** | **void**
6. *fun_declaration* → *type_specifier* **ID (** *params* **)** *compound_stmt*
7. *params* → *param_list* | **void**
8. *param_list* → *param_list* **,** *param* | *param*
9. *param* → *type_specifier* **ID** | *type_specifier* **ID [ ]**
10. *compound_stmt* → **{** *local_declarations statement_list* **}**
11. *local_declarations* → *local_declarations var_declaration* | ε
12. *statement_list* → *statement_list statement* | ε
13. *statement* → *assignment_stmt* | *call* | *compound_stmt* | *selection_stmt*
    | *iteration_stmt* | *return_stmt* | *input_stmt* | *output_stmt*

14. *assignment_stmt* → *var* **=** *expression* **;** | **STRING ;**
15. *call_stmt* → *call* **;**
16. *selection_stmt* → **if (** *expression* **)** *statement*
    | **if (** *expression* **)** *statement* **else** *statement*
17. *iteration_stmt* → **while (** *expression* **)** *statement*
18. *return_stmt* → **return ;** | **return** *expression* **;**
19. *input_stmt* → **input** *var* **;**
20. *output_stmt* → **output** *expression* **;**
21. *var* → **ID** | **ID [** *arithmetic_expression* **]**
22. *expression* → *arithmetic_expression relop arithmetic_expression*
    | *arithmetic_expression*
23. *relop* → **<=** | **<** | **>** | **>=** | **==** | **!=**
24. *arithmetic _expression* → *arithmetic_expression addop term* | *term*
25. *addop* → **+** | **-**
26. *term* → *term mulop factor* | *factor*
27. *mulop* → ***** | **/**
28. *factor* → **(***arithmetic_expression* **)** | *var* | *call* | **NUMBER**
29. *call* → **ID (** *args* **)**
30. *args* → *arg_list* | ε
31. *arg_list* → *arg_list* **,** *arithmetic_expression* | *arithmetic_expression*

B. Semantics Conventions:

A short explanation of the meaning of each grammar rule is provided.

1. *program* → *declaration_list*
2. *declaration_list* → *declaration_list declaration* | *declaration*
3. *declaration* → *var_declaration* | *fun_declaration*

A program consists of a sequence of declarations. The sequence may have function or variable declaration, and the order does not matter. There has to be at least one declaration.

The following restrictions MUST be complied:

- All variables and functions must be declared before they are used.
- The last declaration in a program MUST be a function declaration of the form **void** main**(void)**.
- Observe that C Minus does not have "prototypes", therefore, there is no distinction between declarations and prototypes.

4. *var_declaration* → *type_specifier* **ID ;** | *type_specifier* **ID [NUMBER] ;**
5. *type_specifier* → **int** | **float** | **string** | **void**

A variable declaration states either a simple variable of integer type or an array variable of integer type with indices from 0 to NUMBER-1. In C Minus, the only basic types are **int**, **float**, and **string**.

The following restrictions MUST be complied:

- In a variable declaration, only the type specifiers **int**, **float**, and **string** can be used, **void** is for function declarations.
- Only one variable can be declared per declaration.

6. *fun_declaration* → *type_specifier* **ID (** *params* **)** *compound_stmt*
7. *params* → *param_list* | **void**
8. *param_list* → *param_list* **,** *param* | *param*
9. *param* → *type_specifier* **ID** | *type_specifier* **ID [ ]**

A function declaration consists of a return type specifier, an identifier, and a comma-separated list of parameters inside parentheses, followed by a compound statement that contains the code of the function. If the return type of the function is **void**, then the function returns no value. Parameters of a function are either **void** (i.e., there are no parameters) or a list that represents the function's parameters. Parameters followed by brackets are array parameters whose size can vary.

The following restrictions MUST be complied:

- Simple integer, float, and string parameters are passed by value.
- Array parameters are passed by reference (i.e., as pointers), and MUST be matched by an array variable during a call.
- There are no parameters of type function.
- The parameters of a function have scope equal to the compound statement of the function declaration.
- Functions may be recursive.

10. *compound_stmt* → **{** *local_declarations* *statement_list* **}**

A compound statement consists of curly brackets surrounding a set of declarations and statements. A compound statement is executed by executing the statement sequence in the order provided. The local declarations have scope equal to the statement list of the compound statement and supersede any global declarations.

11. $local\_declarations \rightarrow local\_declarations \quad var\_declaration \mid \varepsilon$
12. $statement\_list \rightarrow statement\_list \quad statement \mid \varepsilon$

Both, the local declarations and the statement list may be empty.

13.
$$\begin{aligned} statement \rightarrow\ & assignment\_stmt \\ \mid\ & call\_stmt \\ \mid\ & compound\_stmt \\ \mid\ & selection\_stmt \\ \mid\ & iteration\_stmt \\ \mid\ & return\_stmt \\ \mid\ & input\_stmt \\ \mid\ & output\_stmt \end{aligned}$$

14. $assignment\_stmt \rightarrow var = expression \ \textbf{;} \mid \textbf{STRING ;}$

An assignment statement has two options. One, assigns the value of an expression to a variable followed by semicolon. The expression to the right side of the assignment is evaluated, and its result is stored in the location that the variable represents. The variable data type shall correspond to the data type of the result value from the expression The second one, assigns a string constant to the variable, the variable type shall be STRING.

15. $call\_stmt \rightarrow call \ \textbf{;}$

A call statement executes a function and ends with semicolon.

16.
$$\begin{aligned} selection\_stmt \rightarrow\ & \textbf{if} \ \textbf{(} \ expression \ \textbf{)} \ statement \\ \mid\ & \textbf{if} \ \textbf{(} \ expression \ \textbf{)} \ statement \ \textbf{else} \ statement \end{aligned}$$

The if-statement has the usual semantics. The expression is evaluated, a non-zero value causes execution of the first part of the statement, while a zero value causes the execution of the **else** part of statement if it exists.

17. *iteration_stmt* → **while** **(** *expression* **)** *statement*

The while-statement is the only iteration statement for C Minus. It is executed by repeatedly evaluating the expression and then executing the statement if the expression evaluates to non-zero. It ends its execution when the expression evaluates to zero.

18. *return_stmt* → **return** **;** | **return** *expression* **;**

A return statement may either return a value or not. Functions not declared as void MUST return a value. Functions declared as voids MUST not return a value. A return causes the transfer of control back to the caller (or termination of the program if it is inside **main**).

19. *input_stmt* → **read** *var* **;**
20. *output_stmt* → **write** *expression* **;**

The output-statement is used to display into standard output (screen) either the value of a variable, the result of a logical or arithmetic expression, or a constant. The input-statement assigns the string obtained from the standard input into the location represented by variable.

The following restriction MUST be complied:

- Only integer values can be assigned to var trough the usage of the input-statement.
- If a non-integer value is obtained from standard input, the program MUST stop.

21. *var* → **ID** | **ID** **[** arithmetic_*expression* **]**

A variable is either a simple (integer) variable, or a subscripted array variable. The following restrictions MUST be complied:

- A negative subscript value MUST cause the program to stop.
- The upper bound of the subscript MUST be checked, if the value of the subscript exceeds the upper bound, then the program MUST stop.

22. *expression* → arithmetic_*expression* *relop* arithmetic_*expression*
| arithmetic_*expression*

23. *relop* → <= | < | > | >= | == | **!**=

An expression is usually evaluated and the result used on the behavior of the program. An expression consists of relational operators that do not associate (i.e., an un-parenthesized expression can only have one relational operator). The value of an expression is either the value of its arithmetic expression, if it contains no relational operators, or 1 if the relational operator evaluates to true; 0 if it evaluates to false.

24. *arithmetic_expression* → *arithmetic_expression addop term* | *term*
25. *addop* → + | **-**
26. *term* → *term mulop factor* | *factor*
27. *mulop* → * | /

Arithmetic expressions represent the typical associative and precedence of arithmetic operators. The / symbol represents the division operation for both INTEGER and FLOAT, i.e., if the variable that is going to receive the result of the division is of type INTEGER, hence, any reminder is truncated.

28. *factor* → **(** *arithmetic_expression* **)** | *var* | *call* | **NUMBER**

A factor is an *arithmetic_expression* enclosed in parentheses. It can also be a variable, which evaluates to the value that it holds. It can also be a call to a function, which evaluates to its the returned value. Finally, it can be a number, whose value is computed by the scanner and it is held in the corresponding Symbol Table.

29. *call* → **ID (** *args* **)**
30. *args* → *args_list* | **ε**
31. *args_list* → *args_list* **,** *arithmetic_expression* | *arithmetic_expression*

A function call consists of an ID which represents the name of the function, followed by parentheses enclosing its arguments. Arguments are either empty or consist of a comma-separated list of *arithmetic_expression*, representing the values to be assigned as parameters during a call.

The following restrictions MUST be complied:

- Functions MUST be declared before they are called
- An array parameter in a *function declaration* MUST be matched with an expression consisting of a single identifier that represents an array variable.

F. Sample Program in C Minus:

The following program inputs a list of 10 integers, multiplies each element by a float number, sorts them by selection sort and outputs the resulting sorted float numbers array.

```
/* Program that reads a 10 element array of
integers, and then multiply each element of
the array by a float, stores the result into an
array of floats. Subsequently, the array of
floats is sorted and display it into standard
output.*/

int x[10];
string s;
float f1;
float f2[10];

int miniloc(float a[], int low, int high){
  int i; float y; int k;

   k = low;
   y = a[low];
   i = low + 1;
   while (i < high){
         if (a[i] < x){
             y = a[i];
             k = i;
          }
          i = i + 1;
    }
    return k;
}/* END of miniloc() */
```

```
void sort(float a[], int low, int high){
  int i; int k;

   i = low;
   while (i < high - 1){
         float t;
         k = miniloc(a,i,high);
         t = a[k];
         a[k] = a[i];
         a[i] = t;
         i = i +1;
   }
   return;
}/* END of sort() */


void readArray(void){
  int i;

   s = "Enter a float number: ";
  write(s);
   read(f1);
   while (i < 10){
         s = "Enter an integer number: ";
        write(s);
        read x[i];
        f2[i] = x[i]*f1;
        i = i + 1;
    }
    return;
}/* END of readArray() */
```

```
void writeArray(void){
   int i;
    i = 0;
    while (i < 10){
         write f2[i];
         i = i + 1;
    }
    return;
}/* END of writeArray() */


void main(void){
    s = "Reading Information…..";
    write(s);
    readArray();
    s = "Sorting…..";
    write(s);
    sort(f2,0,10);
    s = "Sorted Array:";
    write(s);
    writeArray();
    return;
}/* END of main() */
```

G. Parser Output:

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

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

H. Deliverables:

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

1. **The correct grammar for C Minus 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. **Symbol Table management: Description of semantic aspects that were updated during the syntax analysis.**
3. **Error messages generated by the Parser.**
4. **Example of the Parser Outputs**

I. 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.

III. 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.	Does not apply	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

IV. 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.**

V. <u>Written Report</u>

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 finite state machines, regular expressions, and transition tables:

- 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, this section shall provide a brief description and explanation of the lexical components of the programming language for which the project is being developed. It shall include the DFA and Transition Table of the Scanner. You must describe all the considerations taken in order to develop these models. Be sure to explain every part of the DFA and how it complies with the Lexical Definition of the project's language.

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 this section, the student shall describe how each of the requirements for the the four deliverables of the parser, is implemented. The design must include the justification of which Top-Down algorithm was selected, ***Recursive Descend*** or ***LL(1)***. 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 functional requirements for the: 1. Clear and complete description of all the steps applied to obtain the complete and correct grammar for C-Minus, i.e., making the grammar unambiguous, elimination of Left Recursion and Left Factoring, reducing useless symbols, epsilon productions, and unit productions, calculation of sets First, Follow, and First+. 2. Justification whether you developed a Recursive Descent or LL(1) parser.	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 specification.	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. Informal description. 2. Architectural or modular design. 3. Justification. 4. Software Algorithm and data structures. 5. Pseudo code. 6. Traceable to the analysis model.	Traceability to the analysis model is not clear.	NA	NA	NA	Design is **ambiguous**, or **inconsistent**, or **incomplete**, or **incorrect**.	40
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.**