

Final Report

|  |
| --- |
| School of Computing  Faculty of Engineering AND PHYSICAL SCIENCES |

Automated Recursive Descent Parser Generation from a Context-Free Grammar.

Joseph Barrington

Submitted in accordance with the requirements for the degree of  
BSc Computer Science

**2024/25**

**COMP3932 Synoptic Project (EPA)**

The candidate confirms that the following have been submitted*:*

*<As an example>*

|  |  |  |
| --- | --- | --- |
| **Items** | **Format** | **Recipient(s) and Date** |
| *Final Report* | *PDF file* | *Uploaded to Minerva (DD/MM/YY)* |
| *Scanned participant consent forms* | *PDF file / file archive* | *Uploaded to Minerva (DD/MM/YY)* |
| *Link to online code repository* | *URL* | *Sent to supervisor and assessor (DD/MM/YY)* |
| *User manuals* | *PDF* | *Sent to client and supervisor (DD/MM/YY)* |

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)

© 2025 The University of Leeds and Joseph Barrington

# Summary

*<Concise statement of the problem you intended to solve and main achievements (no more than one A4 page)>*

This project aims to design and develop a tool that can automatically produce a recursive descent parser from a given grammar. Writing parsers manually can be complex and time consuming. This project aims to simplify this process by creating a tool that takes a formal grammar in BNF or EBNF notation as an input and generates working code for a recursive descent parser.

# Acknowledgements

*<This page should contain any acknowledgements to those who have assisted with your work. Where you have worked as part of a team, you should, where appropriate, reference to any contribution made by others to the project.>*

*Note that it is not acceptable to solicit assistance on ‘proof reading’ which is defined as “the systematic checking and identification of errors in spelling, punctuation, grammar and sentence construction, formatting and layout in the text”; see*

https:://www.leeds.ac.uk/secretariat/documents/proof\_reading\_policy.pdf

# Table of Contents (example of how to format)

Summary iii

Acknowledgements iv

Table of Contents (example of how to format) v

Chapter 1 Introduction and Background Research 1

1.1 Introduction 1

1.2 Compiler Overview 1

1.2.1 The Compilation Process 2

1.3 Lexical Analysis 2

1.3.1 Implementation Approaches 2

1.3.1.1 Regular Expression-Based Approach 3

1.3.1.2 Deterministic Finite Automata Approach 3

1.4 Syntax Analysis 4

1.4.1 Context-Free Grammars 4

1.4.2 Implementation Approaches 4

1.4.2.1 Top-Down Parsing 5

1.4.2.2 Bottom-Up Parsing 5

1.5 Existing Parser Generation Tools 5

1.5.1 YACC/Bison Evaluation 6

1.5.2 ANTLR Evaluation 7

Chapter 2 Methodology 8

2.1 Table example 8

2.2 Figure example 9

Chapter 3 Implementation and Validation 10

Chapter 4 Results and Evaluation 11

Chapter 5 Conclusions 12

5.1 Conclusions 12

5.2 Future work 12

List of References 13

Appendix A Self-appraisal 14

A.1 Critical self-evaluation 14

A.2 Personal reﬂection and lessons learned 14

A.3 Legal, social, ethical and professional issues 14

A.3.1 Legal issues 14

A.3.2 Social issues 14

<Discussion of social issues> 14

A.3.3 Ethical issues 14

<Discussion of ethical issues> 14

A.3.4 Professional issues 14

Appendix B External Materials 15

**Notes from John:**

* Lit review throughout the research not separate.
* Last section be on the products that are out there doesn’t matter if it is not original just talk about them and evaluate them
* Methodology is more of the design chapter. Talk about the grammar designs etc.
* Implementation is more about how I implemented it maybe talk about sprints etc.
* Results and evaluation talk about the different grammars I have tested and how they do.
* In self appraisal the legal social and ethical issues may not apply to my project but still talk about them.

Amys notes

* Version control in implementation.
* Design In method
* Sprints in implementation
* In self-appraisal talk about them and say what I would of done if there was any.
* Chapter 3 heading make it methodology implementation
* Have a first section talking about how it was iterations and why it

# Chapter 1 Introduction and Background Research

## 1.1 Introduction

Computers do not understand programs in the same way as humans. When we write in high level languages like *Python* or *Java,* the computer must translate the code into a lower level language to execute the program. To complete this action the computer uses a special type of software called a compiler.

Writing a compiler is a complex process, with many different parts, one of the most important parts is the parser. The role of a parser is to read through the code and check if it follows the rules of the programming language. If the code doesn’t follow the rules, the parser flags an error. Writing a parser manually can be time consuming, especially if the high level language is complex.

Every programming language has a set of rules, which describes the syntactical structure of the language. These rules are called a grammar. The parser uses the languages grammar to verify the correctness of the syntax being compiled. This process allows the compiler to flag any syntax errors within the file.

This project aims to create a parser generator – a tool that can automatically generate a parser from a context free grammar. Instead of writing the parser manually, a user can simply provide the set of grammar rules of their language, and the tool will instantly generate the program to parse it.

The complete parser generator should accept a grammar, in a specified format, and allow the user to choose which programming language the parser will be written in. The tool will have three programming languages to choose from being: *Python, Java* and *C.*

## 1.2 Compiler Overview

Computers are designed to execute basic instructions at very high speeds. These basic instructions are called machine language, which consist of raw binary instructions, and are essential for execution. However, programming directly in machine language is both difficult and time consuming, hence why developers use high level languages, allowing them to write code in a more efficient and human-readable way.

A compiler is a specialised software tool, designed to translate high-level programming language into machine code. This process is not just a direct translation; compilers also analyse, optimize, and verify the correctness of the code before execution (1). Without compilers modern software development would be significantly more time consuming and challenging, severely limiting the pace of innovation we see in the world today.

### 1.2.1 The Compilation Process

As mentioned earlier, the compiler does not only directly translate the source code. This process consists of many stages, each responsible for transforming and optimizing the code. The main stages include:

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Optimization
6. Code Generation

While presented as discrete stages, the phases work in collaboration to complete the compilation process (2).

This project will primarily focus on the first two stages: Lexical Analysis and Syntax Analysis. These stages form the foundation of language parsing, and are necessary for recognizing the structure of programming languages. The tool can generate parsers that accurately validate code without the additional complexity of semantic analysis or code generation.

Furthermore, later stages of compilation, for example Semantic Analysis, often depend heavily on language-specific features, making them less suitable for an automated parser generator.

## 1.3 Lexical Analysis

Lexical Analysis is the first stage of the compilation process and is also known as tokenization or scanning. This key stage transforms the source code into a sequence of lexical units, also known as tokens or symbols (3). Tokens are all of the fundamental components that make up the source code, for example: keywords, operators, literals and comments.

This transformation is an important preprocessing step, making it significantly easier for later compilation stages to analyse the programs structure, without having to deal with low level character processing.

### 1.3.1 Implementation Approaches

There are many different approaches for creating a lexer. All implementations follow the same basic principles; read past white spaces and comments and check if the next token is a keyword, operator, number etc.

For this project, we identified two primary techniques and evaluated both, in order to make an educated decision that would best fit the parser generator tool.

#### 1.3.1.1 Regular Expression-Based Approach

Regular expressions, also known as regex, provide a powerful method for pattern matching in text. In terms of lexical analysis, this method offers a concise way of defining the patterns that identify different token types. A basic example of this is identifying integers; using an alphabet of all digits [0-9], the lexer can easily identify all tokens of type integer (1).

A collection of regex patterns are defined, each pattern is associated with a token type. The lexer uses this connection to scan through the source file, matching tokens to their respective pattern. Each pattern is checked in a specific order, usually based on priority, the longest matching pattern is then selected as the token type. The lexer moves to the next position and repeats the process (4).

This method is effective and is easily implemented due to the widespread support in many languages because of regular expression libraries. However, regular expressions alone are not equipped to handle all of the challenges that come with lexical analysis.

#### 1.3.1.2 Deterministic Finite Automata Approach

Deterministic Finite Automata (DFA) uses state machines with precisely defined transitions to recognise patterns in an input file. A DFA consists of: a finite set of states, a defined start state, a set of accepting state and a transition function that maps each state and input character to a next state.

In lexical analysis, the automaton begins in the start state and processes input characters one at a time, transitioning between states according to the transition function. Once the automaton has reached an accepting state, it recognizes a token of the type associated with that state.

A diagram of a diagram

AI-generated content may be incorrect.

**Figure 1 – Example DFA processing comments beginning and ending with braces.**

This approach is robust and efficient for lexical analysis. However, using DFA’s can be complex and less accessible for specifying token patterns (4). Due to the nature of this project, this approach alone is also not feasible.

## 1.4 Syntax Analysis

Syntax analysis, also known as parsing, is the second stage of the compilation process. Once the lexer has transformed the source code into tokens, the parser determines if they form a valid structure according to the grammar rules of the language.

In order for the parser to successfully complete this task, it attempts to construct a syntax tree (1). This tree forms the structure of the input file and allows the parser to efficiently detect syntax errors, if the source code does not follow the structure defined by the grammar rules.

Syntax analysis relies heavily on formal grammar specifications, particularly on context-free grammars to define the structure of the language. The grammar provides the rules that the parser uses to determine if the sequence of tokens forms a valid program.

### 1.4.1 Context-Free Grammars

Context-free grammars (CFGs) provide a formal notation for describing the syntax of a programming language. They consist of four major components: terminals, non-terminals, start symbol and productions (5).

Terminals are the basic symbols of the language that appear in the program as text. For example; in *Python* the keywords ‘if’ or ‘else’ would be terminal symbols. Non-terminal symbols represent patterns in language. For example; in *Python* the entire ‘if statement’ would be a non-terminal symbol.

Start symbols are a non-terminal symbol that represents a complete program. Finally, production rules are the grammar rules that define how terminals and non-terminals are combined. On the left of the rule is a non-terminal and on the right side is the sequence of terminals and non-terminals.

### 1.4.2 Implementation Approaches

There are many different approaches to implementing a parser, however all of these approaches are broadly categorized into two main types: Top-down parsing and Bottom-up parsing. The main difference in the two categories is how the context-free grammar is parsed.

#### 1.4.2.1 Top-Down Parsing

As the name suggests, top-down parsers begin at the start symbol of the grammar and attempt to parse the input string by expanding non-terminals. This category of parsers tend to have a more straightforward implementation, with better error handling than most bottom-up parsers.

One of the parser implementations from this category is the recursive descent parser. In this parser each non-terminal symbol is implemented as a function, with these functions calling each other recursively to parse the input file according to the grammar rules.

This approach is used throughout this project, due to the direct translation of the grammar leading to a straightforward parser generator. Furthermore, this approach has an ideal complexity level for the aim of the project, unlike some of the other methods evaluated.

One disadvantage of the recursive descent parser is its inability to handle left-recursive grammar rules – this is when a non-terminal contains itself first in the rule (For example: expr = expr | 1), causing a recursive loop (1). Despite this flaw, the decision was made to use this approach due to the many advantages it possesses.

#### 1.4.2.2 Bottom-Up Parsing

Bottom-up parsing takes the opposite approach to top-down parsing. They begin with the input token and gradually translate them to higher level constructs, determined by the grammar, until the starting symbol is reached.

Bottom-up parsers typically rely on parsing tables that determine when to read the next token or apply a grammar rule to the input (5). These tables require algorithms to analyse the grammar for potential conflicts and create appropriate state transitions.

This style of parsers allow for a wider range of grammars to be parsed compared to top-down parsers, this includes grammars that contain left-recursive rules. Additionally, these parsers are more efficient than top-down parsers and handle grammar ambiguity in a more robust manner.

Despite the advantages of this type of parser, it is not appropriate for this project due to the vast differences in complexity level between the two. Creating a bottom-up parser generator from a grammar would be challenging enough, but also including properties such as parsing tables would significantly exceed the scope of this project.

## 1.5 Existing Parser Generation Tools

Parser generation has evolved significantly since its creation, with many tools offering different approaches to transform formal grammars into functioning parsers. The thesis behind this project was inspired by these tools however, it diverges in one key aspects that addresses a limitation in existing solutions.

Ortin et al. (2022) conducted an empirical study comparing the YACC and ANTLR parser generation tools in an educational context. Their research evaluated both tools when used by third-year Software Engineering students. The empirical data demonstrated that ANTLR’s more accessible approach led to “significantly higher performance” compared to the more complex approach of YACC.

Powerful parser generators like YACC, Bison and ANTLR all require a complex grammar specification, with YACC and Bison requiring C code intermixed with grammar rules. In contrast, this project adopts a simplistic approach in terms of the required grammar specification.

A black background with white text

AI-generated content may be incorrect.

**Figure 2 – Basic example of the grammar specification used to generate a parser in this project.**

### 1.5.1 YACC/Bison Evaluation

YACC (Yet Another Compiler Compiler) is one of the most influential parser generators in the industry. Developed by Stephen C. Johnson in the 1970’s (7), many modern generation tools still use the patterns established in YACC. Bison is an example of this, created with direct inspiration from YACC and still maintained today (8).

Both YACC and Bison generate Bottom-up parsers, whereas the focus of this project is Top-down parsers. Implementing a Bottom-up parser is much more complex, but allows the generation of a wider range of programming languages. Additionally, this type of parser can efficiently handle left recursion, while Top-down parsers cannot.

Figures 2 and 3 offer a comparison between the input grammar specifications for this project and YACC/Bison. Both examples are for the same language. Figure 3 presents an elaborate specification using BNF (Backus-Naur Form) including C code and explicit token declaration (8). In comparison, figure 2 displays a simplistic approach without the use of a programming language and token declaration is not necessary, as the program interprets the tokens through the use of speech marks.

In conclusion, YACC/Bison offer a very powerful parser generation tool, which can handle complicated grammars, including left recursion. However, this power comes at the cost of simplicity for the user.

### 1.5.2 ANTLR Evaluation

ANTLR (Another Tool for Language Recognition), developed by Terrence Parr, has evolved through many iterations and become one of the most widely used modern parser generators. Unlike YACC/Bison, ANTLR generates a recursive descent parser (9), the similar top-down approach used in this project.

A computer screen with white text

AI-generated content may be incorrect.

**Figure 4 - Basic example of the grammar specification used to generate a parser in ANTLR.**

The grammar specification used by ANTLR is much simpler than that of YACC/Bison, still it remains more complex than our approach. As shown in figure 4, the grammar specification for ANTLR uses EBNF (Extended Backus-Naur Form) and includes a separate section for the lexer rules. This functionality allows the user to control token recognition, reduce ambiguity and improve error handling.

Overall, ANTLR is a very powerful recursive descent parser. The tool has found a great balance between power and simplicity. Furthermore, ANTLR offers an extensive list of tools for developers to take advantage of when interacting with the program.

# Chapter 2 Methodology

<Everything that comes under the `Methodology' criterion in the mark scheme should be described in one, or possibly more than one, chapter(s).>

* First prototype was developed using a EBNF grammar for sentences which looked like this:

sentence\_grammar = """

sentence = subject , verb , object ;

subject = article , noun ;

object = article , noun ;

article = "the" | "a" ;

noun = "cat" | "dog" | "bird" ;

verb = "chases" | "catches" | "watches" ;

"""

* Next prototype was done on the expression grammar. This included grouping, and repetition which made the parsing of the grammar much more complicated.
* Next was using the JACK language grammar. This was a big step as it also involved needing the use of a lexer.

## 2.1 Table example

|  |  |  |
| --- | --- | --- |
| **Heading One** | **Heading Two** | **Heading Three** |
| 1.1 | 1.2 | 1.3 |
| 1.21 | 1.22 | 12.3 |
| 12.31 | 12.32 | 12.33 |

Text before table. Text before table. Text before table. Text before table. Text before table. Text before table. Text before table. Text before table. Text before table. Text before table.

**Table 2.1** This is the table description in the ‘table description’ style.

## 2.2 Figure example

Figures can be added using the Illustrations section of the Insert tab.



**Figure 2.1** This is the figure description in the ‘figure description’ style.

# Chapter 3 Implementation and Validation

<Everything that comes under the `Implementation and Validation' criterion in the mark scheme should be described in one, or possibly more than one, chapter(s).>

# Chapter 4 Results and Evaluation

<Results, evaluation (including user evaluation) *etc*. should be described in one or more chapters. See the `Results and Evaluation' criterion in the mark scheme for the sorts of material that may be included here.>

# Chapter 5 Conclusions

<Outcomes of your project. Discussion of the aim and context of your project and what you achieved in that regard. Everything that comes under the `Results and Discussion' criterion in the mark scheme that has not been addressed in an earlier chapter should be included in this final chapter. The following section headings are suggestions only.>

## 5.1 Conclusions

<Text in 11-point size and 1.5 line spacing.>

## 5.2 Future work

<Text in 11-point size and 1.5 line spacing.>

# List of References

*<It is expected that the list would reflect the breadth and depth of scholarly research undertaken by the student during the course of the project*

*The format of referencing is nor prescribed, but you must use one style appropriately and consistently>*

1. <https://cse.sc.edu/~mgv/csce531sp20/basics_lulu2.pdf> - reference from page
2. <https://books.google.co.uk/books?id=uvN-EAAAQBAJ&lpg=PP1&ots=yGwiThrjpb&dq=Compiler%20process&lr&pg=PA3#v=onepage&q=Compiler%20process&f=false>
3. <https://books.google.co.uk/books?id=NTIkJAuytiwC&lpg=PT9&ots=hfWd5dG3_1&dq=lexical%20analysis%20compiler&lr&pg=PT11#v=onepage&q=lexical%20analysis%20compiler&f=false>
4. <https://arxiv.org/abs/2412.13581>
5. <https://elib.vku.udn.vn/bitstream/123456789/2542/1/2007.%20Compilers-Principles%2C%20Techniques%2C%20and%20Tools%20%282nd%20Edition%29.pdf>
6. Ortin F, Quiroga J, Rodriguez-Prieto O, Garcia M (2022) An empirical evaluation of Lex/Yacc and ANTLR parser generation tools. PLOS ONE 17(3): e0264326. <https://doi.org/10.1371/journal.pone.0264326>
7. Johnson, S.C., 1975. *Yacc: Yet another compiler-compiler* (Vol. 32). Murray Hill, NJ: Bell Laboratories.
8. Levine, J., 2009. *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.".
9. Parr, T.J. and Quong, R.W., 1995. ANTLR: A predicated‐LL (k) parser generator. *Software: Practice and Experience*, *25*(7), pp.789-810.

# Appendix A Self-appraisal

<This appendix must contain everything covered under the ’self-appraisal’ criterion in the mark scheme. Although there is no length limit for this section, 2-4 pages will normally be suﬃcient. The format of this section is not prescribed, but you may like to consider the following sections and subsections.>

## A.1 Critical self-evaluation

## A.2 Personal reﬂection and lessons learned

## A.3 Legal, social, ethical and professional issues

<Refer to each of these issues in turn. If one or more is not relevant to your project, you should still explain *why* you think it was not relevant.>

### A.3.1 Legal issues

<Discussion of legal issues>

### A.3.2 Social issues

### <Discussion of social issues>

### A.3.3 Ethical issues

### <Discussion of ethical issues>

### A.3.4 Professional issues

<Discussion of professional Issues>

# Appendix B External Materials

<This appendix should provide a brief record of materials used in the solution that are not the student's own work. Such materials might be pieces of codes made available from a research group/company or from the internet, datasets prepared by external users or any preliminary materials/drafts/notes provided by a supervisor. It should be clear what was used as ready-made components and what was developed as part of the project. This appendix should be included even if no external materials were used, in which case a statement to that effect is all that is required.>

A screenshot of a computer program

AI-generated content may be incorrect.