ISSN 1870-4069

Comparison of Parsing Techniques

Nida Hafeez[1,2], Muhammad Ateeb Ather[2], Abdollah Abadian[1], José Luis Oropeza Rodríguez[1]

[1] Instituto Politécnico Nacional (IPN), Centro de Investigación en Computación (CIC), Mexico City, Mexico

[2] Bahria University, Lahore, Department of Computer Sciences, Pakistan

03-134211-022@student.bahria.edu.pk

Abstract: This review paper provides a thorough examination of compiler building parsing methods. It emphasized the significance of parsing in compiler construction and the requirement for effective and precise parsing algorithms. The paper examines a number of parsing strategies, including top-down and bottom-up parsing, discussing both traditional approaches and more contemporary developments. The evaluation considers variables including time complexity, error recovery, support for ambiguous grammar, and simplicity of implementation. Additionally, the effects of grammar traits and problem-solving techniques are covered. In order to increase the effectiveness and dependability of compiler construction, this paper offers researchers and practitioners useful insights and recommendations for selecting and implementing parsing strategies in compiler projects.

Keywords: Parsing, top-down, bottom-up, semantics, syntax, compiler, predictive parsing.

1 Introduction

Compiler converts a high-level language program to a low-level language program which a computer can understand and interpret. To achieve this conversion, the compiler passes the program through various phases. Each one of these phases has a unique role to play during the conversion. One of them is Parsing. Parsing in simple English language means to break down the sentence into grammatical parts and then identifying those parts and their relation to one another just like that, Parsing in Compiler Construction is breaking down the program into the grammatical rules of the high language to identify the syntax of the program.

Parsing is done in order to generate the syntax of the targeted grammar and then the syntax can also be verified using the same technique. Parsing is done by a part of the compiler named Parser. According to Hassan. Et. Al [2]. The main aim of parsing is to determine whether the string is part of the grammar or not. This can be achieved through various parsing techniques.

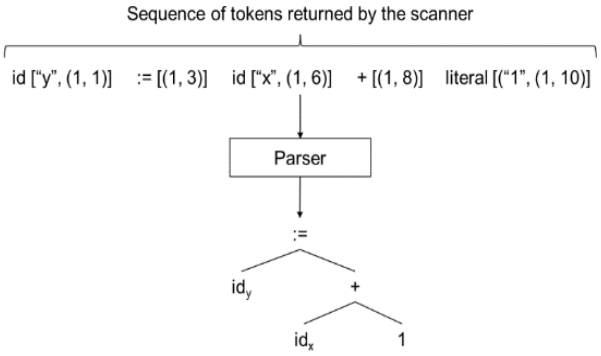

Fig. 1 Sequence of tokens returned by the scanner.

Out of all the parsing techniques two which are most widely used and discussed are top- down parsing and bottom-up parsing, based on these approaches the parser designed are either top-down parsers or bottom-up. Both the techniques are different depending on their names, the top-down parsing starts tracking down from the start of the production to the bottom whereas the bottom-up technique winds up the tracking from the end of the production to the start. Different techniques of parsing having their own pros and cons are not only used in compiler construction but are widely used in other fields of computer sciences such as Natural Language Processing (NLP), Robotics, Machine Learning(ML), Databases and even in Web Development. Due to the importance of parsing in multiple domains of Computer Science it is one of the most researched topics of Compiler Construction.

2 Literature Review

A. Parser

The role of the parser is to verify the syntax of the language used. The main function of the parser is to take in the final output from the lexical analyzer and then it uses the parsing techniques to verify the syntax. If the program is not syntactically correct, then the parser throws out the syntax error and if the syntax is verified then it generates the parsing tree of the CFG. The parser must have an error detection mechanism to detect errors in the syntax.

There are two approaches to define the grammars:

i. Context-Free Grammar,

ii. Parsing Expression grammar.

Parsing Expression grammar or PEG is a formal grammar to define formal languages using a set of rules to recognize that the string belongs to the language. PEG has only one valid abstract syntax tree for each grammar. This makes sure that the grammar is not ambiguous [4].

The work of the parser can be visualized.

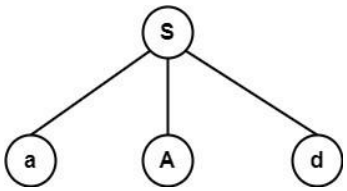

Fig. 2(a) shows the first production i.e., a Ad.

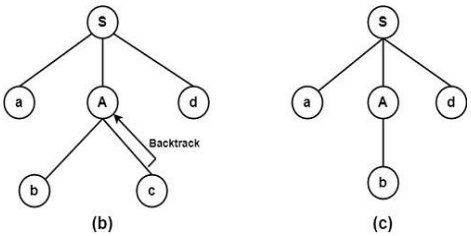

Fig. 2(b) shows that the production A → b c.

B. Top-Down Parsing

Top- down parsing is the parsing technique in which the parser starts to match the statement from the top of the parse tree and tries to match with the rules of the grammar. Top-down parsing involves the left derivation of the tokens and utilizes the expansion of terminal symbols. Backtracking is vital to have optimal solution using the top-down parsing.

Following are the types of top-down parsing techniques:

a. Backtracking

Backtracking is a technique in which the alternatives are used to expand the non-terminals. If one alternative does not work the other one is adopted. The mechanism is that if the opted set of rules does not verify the syntax, then the analyzer restarts, this time with a different set of rules which weren't used before. Consider the following grammar:

Input: S → aAd

A → bc | b

"abd" is the required string and "abcd" is not required.

It is used and the string abcd is formed, which is not a required string so it backtracks. Fig.2(b) shows that the alternative production which is A → b is used to obtain the required string "abd".

Backtracking is easier to implement. To backtrack to the previous state the backtracking technique keeps the states saved. The code is also small in size. Solving the large problem with backtracking is not effective as the technique becomes quite slow for the large problems and also it requires so much space to store all the states to backtrack the alternative rules. Due to the investigation of several paths, backtracking parsers may behave in a non- deterministic manner. As a result, the parsing outcomes could be less predictable. The chosen alternative might rely on how the options are explored or on other variables. Error reporting and recovery may become more difficult as a result. Due to all these reasons parsing techniques without backtracking are preferred.

b. Predictive Parsing

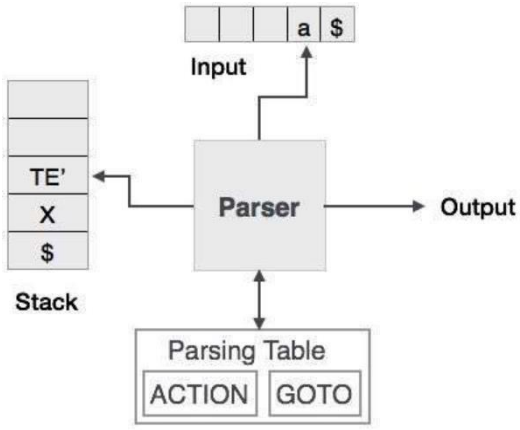

Fig. 3. Shows the predictive parsing components.

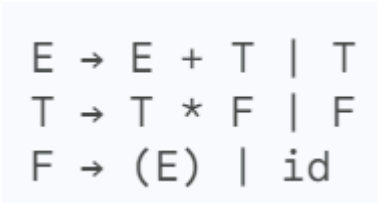

Fig. 4. (a) shows the grammar which can be used in predictive parsing

Fig. 4. (b) shows grammar which cannot be used in predictive parsing.

Predictive parsing is recursive descent parsing with the aim of predicting which of the productions is best to substitute the input string. Parse tree is constructed from the topmost symbol, input string is read from leftward side towards right side. This technique iteratively construes the input string and then the parsing tree is generated [5].

Predictive parsing has a stack to store the production symbols in the stack, whereas the input contains the input string that has to be verified. The third thing that the table contains is the production. If the top of the stack and the top of the input stack are the same, then we pop it out from both stacks and this way if both stacks have $ sign left then the string is verified and if input stack contain one or more than one such symbol which cannot be popped when compared with the stack then the string cannot be verified using the grammar.

Predictive parsing does not need backtracking and thus it uses only grammar which are backtrackless. To achieve this, the left recursion and the left factoring has to be removed from the grammar, basically making it backtrackless, which makes the process extensive. The only grammar that predictive parsing can use is LL(k) grammar where there is no ambiguity and left recursion.

Predictive parsing is a strong tool and provides flexibility to try out other alternatives but the fact that it cannot use all the grammar makes it to be used in certain cases only.

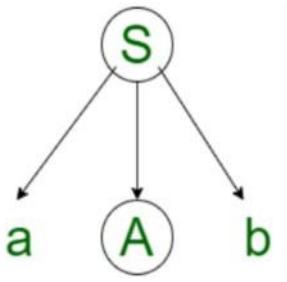

Fig. 5. (a) shows that the first production is used to start the parse tree.

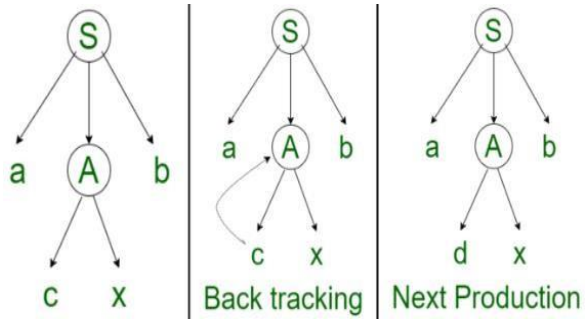

Fig. 5. (b) shows that the next production.

c. Recursive Descent Parsing

Recursive descent parsing is the type of parsing in which the parser is implemented using a set of recursive procedures, each corresponding to the non-terminal of the grammar. The parser starts with the top-level non-terminal and recursively invokes the corresponding procedures to parse the input string, making decisions based on the current input symbol.

Predictive parsing can also be called a special type of recursive descent parsing in which the backtracking is not possible. But both techniques have few differences such as the predictive parsing use of the first and follow sets, also the table is required to parse and predict which of the next production has to be used.

The recursive descent can or cannot use backtracking depending on what kind of grammar is being used to parse, whereas backtracking is not possible in predictive parsing. The recursive descent parsing use the left most input symbol to recursively analyze and then the parse tree is generated [5].

For instance:

Input string:

adxb S → aAb|aBb

A → cx | dx

B → xe

A → cx is used but this production does not verify the production so in Fig.5(c) shows that the backtracking is done and in Fig.5(d) the next production i.e. A → dx is used which verify the input string.

By using this technique, we cannot use objects other than global semantics, as recursion is to be done so the previous state has to be stored which requires memory.

	First	Follow
S –> A/a	{ a }	{ $ }
A –>a	{ a }	{ $ }

Fig. 6. (a) shows the first and follow set of the above grammar.

	a	$
S	S –> A, S –> a	
A	A –> a	

Fig. 6. (b) shows the table for the LL parsing.

d. LL Parsing

LL parsing is the left-left parsing, in which the first left defines that the input will be scanned from left to the right and the second left shows that the left derivation is used for parsing trees.

In this parsing first the first and follow sets of the productions are formed and after that the table is formed in which the rows are composed of non-terminals whereas the columns are made up of terminals. Any cell with no entry corresponds to different firsts and follow sets. If any cell contains more than one production that means that the grammar is ambiguous, and the LL parsing can't be done on that ambiguous grammar [3].

Consider the following grammar:

$$S \rightarrow A \mid a \; A \rightarrow a$$

As one of the cells contains two production rules that means the grammar is ambiguous and the LL parsing of this grammar is not possible until and unless the ambiguity is removed

LL parsing is also known as LL(k) parsing if it uses K tokens of look-ahead when parsing a sentence. Looking ahead at the next symbol helps in deciding which choice to make. The top-down parser does the prediction of input, and this prediction has a terminal symbol in front [6].

e. Bottom-up Parsing

As the name suggests, bottom-up parsing is a parsing technique in which the grammar productions are used in the opposite way, i.e., starting from the right and working its way up to the starting symbol. Due to the bottom to top approach used in this parsing it is known as bottom-up parsing. It is commonly used in the implementation of compilers and language processors. Bottom-up parsing uses a stack- based approach to identify and reduce groups of terminals and non-terminals until the input string is fully parsed.

It contains the following:

- Stack (to store),
- Output (provide results),
- Input (given to parser),
- Driver program (Same for all LR Parsers),
- Parsing Table (has two functions “Action”, “Go To”, varies from parser to parser).

Bottom-up parsing is often based on the shift- reduce parsing strategy. The parser scans the input string from left to right, shifting terminals onto a stack until it can reduce a production rule and replaces them with the corresponding non- terminal.

The bottom-up parser maintains a stack to store the symbols encountered during the parsing process. It also maintains an input buffer containing the remaining input symbols that are yet to be parsed. Group of symbols on the stack to a non-terminal using a production rule [10].

It is required to have a handle recognizer which will assist in deciding appropriate action, by perusing the stack. FSM (finite-state-machine) operates as a recognizer. Nevertheless, here terminals and non-terminals are included in language symbols. Last state or final state indicates success or reduction by dropping it out, when following any rules [5].

There are two basic actions in Bottom-up- parsing:

- Move present input token in stack also, read next token.
- Reduce by rule of production.

Bottom-up parsing is powerful and can handle a wide range of context-free grammar, including ambiguous and left-recursive grammar. It provides greater expressiveness and flexibility compared to some other parsing techniques. Bottom-up parsers are generally better at error recovery compared to top-down parsers. They can often continue parsing after an error and recover to a valid state, reducing the impact of syntax errors on the overall parsing process.

f. Precedence Parsing

Precedence parsing is a technique for code analysis that employs operator precedence to resolve conflicts between many potential code meanings. Because the order of operations and the associativity of operators must be considered when parsing expressions and arithmetic operations, it is especially helpful in these situations.

There are two type of precedence parsing:

a) Operator Precedence

Operator Precedence Parsing is a method of precedence parsing that utilizes an operator precedence table. The table specifies the precedence and associativity of operators in the grammar. It is constructed based on the grammar's production rules and the operators used.

b) Simple Precedence

Simple Precedence Parsing is another type of precedence parsing that uses a simple precedence relation between adjacent non- terminals. It is a more relaxed version of Operator Precedence Parsing and can handle a wider range of grammars.

g. LR Parsing

LR parsing is the most commonly known bottom-up parsing technique used in compiler construction to construct the tree. It generally stands for Left Right derivation, it is of different types like LALR, SLR etc. [11].

1. SLR

SLR stands for simple Left Right derivation and it's the most simple and efficient one from the rest of the LR parsing techniques. It is based on LR(0) and uses a simplified LR parsing table. It is widely used in compiler construction because it's easier to implement yet a reasonably powerful technique [9].

2. LALR

Look-ahead ahead LR is the parsing technique that has a look-ahead pointer and is used to create a compact yet efficient compiler for context-free grammar. LALR is the extension of SLR, addressing the limitations of the SLR by increasing the power of the parsing yet keeping it quite efficient [8]. LALR can resolve more shift-reduce and reduce-reduce conflicts, making it much more powerful than SLR. LALR basically uses a small parsing table, therefore decreasing the need for the memory needed to store the states of the table. But still

LALR is not as powerful as the expressive LR(1) parsing technique.

3. GLR

It is used commonly for the following reasons:

1. Great parsing efficiency.

2. Effective error recovery [7].

GLR has active support for recovery against error. The foremost contributions are

1. GLR provides fast parsing speed with LALR (1).

2. Additional fault retrieval mechanisms [3].

4. CYK (Cocke-Younger-Kasami)

CYK is not appropriate as it has large time space complexity. It does not have a static direction of scanning and a complete source is essential to be read as an early step while parsing. It means, it cannot be utilized in parsing huge legacy-systems. Ideal algorithm must be able to:

- Parse any given context-free grammar.

Table 1. Shows the comparison of Parsing Technique and their applicability.

Technique	Advantage	Limitation	Applicability
Recursive Descent Parsing	Easy to implement and understand, suitable for LL(k) grammars	Inefficient for left-recursive grammars, struggles with ambiguity	Simple grammars, LL(k) grammars
LL(k)	Efficient for LL(k) grammars, top-down parsing, good error reporting	Limited support for left- recursive grammars, k-factor lookahead restriction	LL(k) grammars, non- left-recursive grammars
LR(k)	Efficient for LR(k) grammars, handles left-recursion, broad language coverage	Complex implementation, potentially ambiguous grammars require resolution	LR(k) grammars, general-purpose parsing
LALR(1)	Efficient, compact parsing tables, broader language coverage than LL(k)	Efficient, compact parsing tables, broader language coverage than LL(k)	LR(0) grammars, practical applications
GLR	Handles ambiguous grammars, multiple parse tree generation, broad language coverage	Increased memory usage, potentially slower due to ambiguity resolution	Ambiguous grammars, natural language processing
Packrat	Linear time parsing for grammars with limited backtracking, good error reporting	Increased memory usage, not suitable for all grammars	Parsing with limited backtracking, PEG grammars

- Image input from left to right.
- Eliminate backtracking.
- GLR rightly fulfills these requirements [2].

5. Recursive Ascent Parser

a. Packrat parser

It provides flexibility as well as ability of backtracking and unrestricted lookahead, but nevertheless guarantees linear parse time [5]. It does not require any special lexical analyzer.

3 Differences

Two popular methods for creating parsers are top-down and bottom-up parsing. While bottom-up parsing starts with the input and groups terminals into non-terminals, top-down parsing starts with the root and extends non- terminals to match the input string.

Top-down parsing offers clearer error signals and is more practical for simpler grammar. Bottom-up parsing, on the other hand, is more potent, effective for bigger grammar, and better at error recovery.

The complexity and requirements of the grammar determine which of the two approaches is best.

In contrast to bottom-up parsing, which is better for complex grammars with left-recursion, ambiguity, or LR(k) features, top-down parsing is recommended for less complicated, unambiguous, and LL(k) grammars.

The final choice ought to be dependent on the particular grammar and parsing requirements.

4 Future Work

Future work on this article includes investigating novel parsing techniques, improving performance through sophisticated algorithms and parallel processing, creating reliable error recovery and correction methods, handling complex grammars with specialized techniques, integrating parsing with other compiler phases, and performing empirical studies on real-world applications. These research directions seek to deepen our comprehension of parsing methods, better compiler performance generally, and facilitate the creation of more dependable and efficient compilers for a variety of programming languages and applications.

5 Conclusion

In conclusion, the different kinds of parsers used in real-time applications have been discussed in this study. We have reviewed a number of parsing algorithms; in terms of comparison, bottom-up approaches are the most effective but difficult to implement; backtracking is not required, but handling is implemented. However, top-down parsing

is straightforward whereas backtracking is occasionally necessary, where both parsing methods are appropriate, backtracking can be removed using predictive parsing, making top-down parsing simple to use.

References

1. Angelo Borsotti, L. B.: General parsing with regular expression matching. Journal of Computer Languages, 74, (2022). doi:10.1016/j.cola.2022.10 1176.
2. Hamna Baqai, A. I. Comparison of Parsing Techniques (2020)
3. Hassan Ali, M. S.: LL(1) (2020)
4. Parser versus GNF inducted LL(1) Parser on Arithmetic Expressions. Quest research journal (2010)
5. Hnin Myat Soe, D. M.: Implementation of Learning for Syntax Analyzer. *Fourth* Local Conference on Parallel and Soft Computing (2009)
6. Ismail A. Ismail, N. A.: Internet of things -application and future. 114, pp. 115–124 (2019)
7. Moore, J.: Introduction to Compiler Design. pp. 14–32 (2019)
8. van Binsbergen, L.T.: Purely functional GLL parsing. Journal of Computer Languages, 58, (2020). doi:10.1016/j.cola.2020.10 0945.
9. Oudshoorn, M.: A Drop-in Replacement for LR[1] Table-Driven Parsing. Journal of Advances in Computing and Engineering, 1(2) (2021)
10. Pankaj Sharma, N. M. Parsing Techniques: A Review. International Journal of Advanced Research in, 2(10) (2019)
11. Wyk, E. V.: Context in Parsing: Techniques and Application (2023). doi:10.4230/OASIcs.EVCS.2023.18.
12. Zimmerman, J.: Practical LR Parser Generation. Formal Languages and Automata theory (2022)