iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 10 **Issue:** IX **Month of publication:** September 2022

DOI: https://doi.org/10.22214/ijraset.2022.46588

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 10 Issue IX Sep 2022- Available at www.ijraset.com

Survey on Various Syntax Analyzer Tools

Spurthi Bhat[1], Rutuja Bhirud[2], Vaishnavi Bhokare[3]
Department of Computer Engineering, Vishwakarma Institute of Technology, Pune

Abstract: ***Syntax analysis forms the second phase of a compiler. Syntax Analyzer basically takes the input of tokens from the lexical analyzer and parses the source code according to the production rules to detect errors in the code. The syntax analyzer gives an output in the form of a parse tree. The syntax analysis techniques can be classified as top-down parsing and bottom-up parsing. These categories can be further subdivided into recursive descent, LL (1), operator precedence, LR (0), SLR (1), CLR (1) and LALR (1) respectively. Various parser generators can generate parsers of these types which have been studied and analyzed in this paper such as Beaver, Tatoo, APG etc. This paper provides an overview of all these tools with respective to their working, advantages and features. These syntax analyzer tools can be used for different purposes according to the user.***
Keywords: Bottom-Up Parser, Compiler, Survey, Syntax Analyzer, Top-Down Parser

I. INTRODUCTION

Syntax analysis or parsing forms the second phase of a compiler. Input from a lexical analyzer is taken by a parser or a syntax analyzer as token streams. The source code (token stream) is analyzed by the parser in context of the production rules to detect any errors in the code. The output of this phase is generated in the form of a parse tree. This way, the parser performs two tasks, i.e., parsing the code, finding errors and generating a parse tree as the output of the phase. Parsers are supposed to successfully parse the complete code despite of some errors exist in any part of the program. Fig. 1 shows the working of a syntax analyzer.

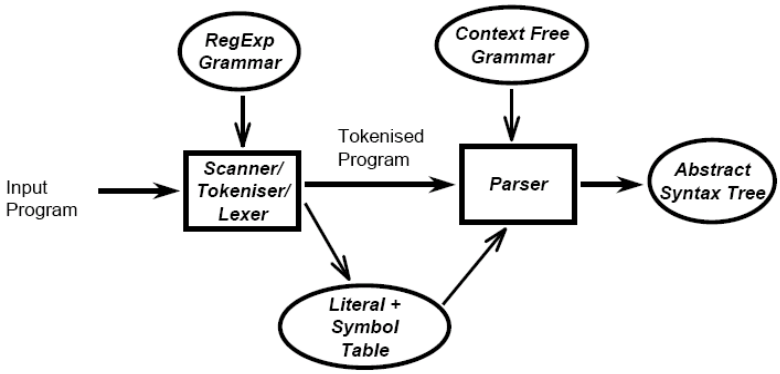


Fig. 1 Working of a Syntax Analyzer

Parsing techniques are segregated into two different categories i.e., Top-Down Parsing and Bottom-Up Parsing. Fig. 2 shows different types of parsers.

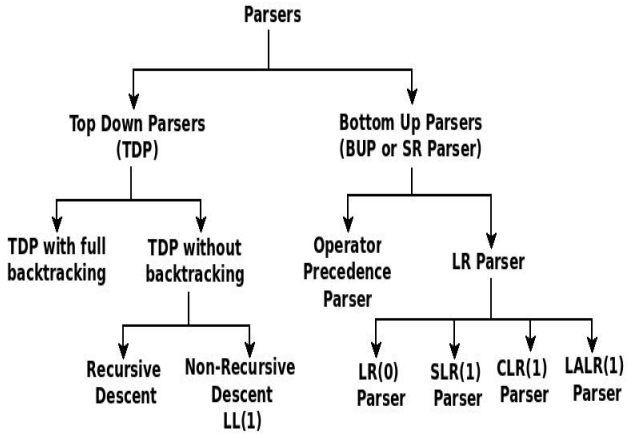


Fig. 2 Types of Parsers

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 10 Issue IX Sep 2022- Available at www.ijraset.com

II. SURVEY

A. Top-Down Parsing

Construction of the parse tree starts at the root and then proceeds towards the leaves in top-down parsing. Two types of Top-down parsing are:

1) Predictive Parsing

Predictive parsers can predict which production should be used to replace the specific input string. The point called look-ahead point, which points towards next input symbols is used by the predictive parsers. Backtracking is not a problem with this parsing technique. It is known as LL (1) Parser.

Some of the parser generators which generate parsers of LL (1) type are as follows:

a) *LLGEN:* [1] LLgen provides a tool for generating an efficient recursive descent parser with no backtrack from an Extended Context Free syntax. Extended LL (1) parsers are considered to be an extension of LL (1) parsers. The ECF syntax specification forms major part of a LLgen grammar specification. Names in this syntax specification refer to either tokens or nonterminal symbols. LLgen requires token names to be declared as such. This way it can be avoided that a typing error in a nonterminal name causes it to be accepted as a token name. A name will be regarded as a nonterminal symbol, unless it is declared as a token name. If there is no production rule for a nonterminal symbol, LLgen will throw an error. A grammar specification may also include some C routines, for instance the lexical analyzer and an error reporting routine. Thus, a grammar specification file can consist of declarations, grammar rules and C code. LLgen allows arbitrary insertions of actions within the right-hand side of a production rule in the ECF syntax. An action comprises of a number of C statements, enclosed within the brackets "{" and "}". LLgen generates a precise parsing routine for each rule in the grammar. LLgen generates a number of files: one for each input file, and two other files: Lpars.c and Lpars.h. Lpars.h contains "#-define"s for the tokennames. Lpars.c contains the error recovery routines and tables that are sometimes needed to rectify errors. LLgen has successfully been used to create recognizers for Pascal, C, and Modula-2.

b) *COCO/R:* [2] Coco/R is a compiler generator, which takes an attributed grammar for a source language and generates a scanner and a recursive descent parser. The user has to provide a main class that calls the parser and the semantic classes that are used by semantic actions in the parser. This is shown in Fig. 3.

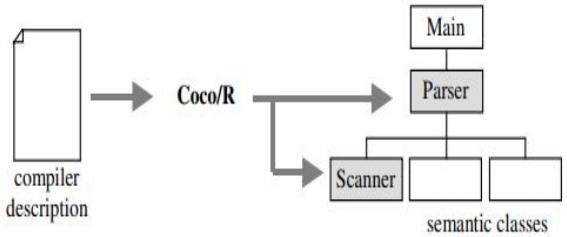


Fig. 3 Input and Output of COCO/R

The parser is specified by a set of EBNF productions which have attributes and semantic actions. The productions allow for different alternatives, repetition and optional parts. Coco/R converts the productions into a recursive descent parser which is compact and efficient. Nonterminal symbols may have a number of input and output attributes. Terminal symbols are not allowed to have explicit attributes, but the tokens returned by the scanner contain information that can be viewed as attributes. All attributes are evaluated during parsing. Semantic actions can be placed anywhere and everywhere in the grammar. They may contain arbitrary statements or declarations that are written in the language of the generated parser (e.g., C# or Java). In principle, the grammar must be LL (1). However, Coco/R is capable to handle non-LL (1) grammars by using so-called resolvers that make a parsing decision based on a multi-symbol lookahead or on semantic information. Coco/R checks the grammar specifically for completeness, consistency and non-redundancy. It also reports LL (1) conflicts. Coco/R has been used by various people for the following applications:

- An analyzer for the static complexity of programs. The analyzer evaluates the kind of operators and statements, the nesting of corresponding statements and expressions as well as the use of local and global variables to gauge a measure of the program complexity and a notion of whether the program is well structured.
- A program that builds a repository of symbols and their relations in a program.

2) Recursive Descent Parsing:

This parsing method recursively parses the input in a recursive manner to develop a parse tree. It comprises of different small functions. A parser generator which generates a parser of Recursive Descent type is as follows:

a) APG: [3] APG – an ABNF Parser Generator – was originally designed to generate recursive-descent parsers directly from the ABNF grammar defining the sentences or phrases to be parsed. The approach is to recognize that ABNF defines a tree with seven types of nodes and that each node represents an operation that can guide a depth-first traversal of the tree – that is, a recursive-descent parse of the tree. APG reduces ambiguity by always selecting only a single, well-defined parse tree from the multitude. From here it was quickly realized that this method of defining a tree of node operations did not in any way require that the nodes correspond to the ABNF-defined tree. They could be expanded and enhanced in any way that might be convenient and efficient for the problem at hand. The first expansion is to add the "look ahead" nodes i.e., operations that look ahead for a specific phrase and then continue or not depending on whether the phrase is found. Next nodes with user-defined operations were introduced. That is, a specific phrase-matching problem is performed by a hand-written node operation. Finally, to develop an ABNF-based pattern-matching engine similar to regular expressions, regex, a number of new node operations have been added: look behind, back referencing, and begin and end of string anchors.

The parser generation begins with an SABNF grammar. This may be obtained from a file, cpApiInFile(), or a string, cpApiInString(). Each successive call after the first call, concatenates its input to that of the previous calls.

The first step is to validate the input grammar characters, vApiInValidate(). SABNF is fully described by the printing ASCII character set, plus tab, line feed and carriage return. The syntax phase parses the input SABNF grammar, verifies that the syntax is correct and generates an AST for translation. The semantic phase checks for any semantic errors and translates the grammar syntax into the rule, UDT and opcode information required by an APG parser.

Once the grammar has been processed, there are then two methods for generating a parser object, vApiOutput() and vpApiOutputParser().

vApiOutput() will generate two grammar files. A base name is provided, say mygrammar, and two files defining the rules, UDTs, opcodes and all necessary supporting data will be generated – mygrammar.h and mygrammar.c. mygrammar.h must be included with the application and mygrammar.c must be compiled with it, to create a parser object from these files. A parser can then be constructed using the data pointer vpMygrammarInit supplied in mygrammar.h. vpApiOutputParser() will return a pointer to a parser object directly.

Today, APG is one of the most versatile, and a well-tested generator of parsers. And because it is based on ABNF, it is especially well suited to parsing the languages of many Internets' technical specifications. Several large Telecom companies use APG. Previous versions of APG have been developed to generate parsers in C/C++, Java and JavaScript.

B. Bottom-Up Parsing

In the bottom-up parsing technique, the construction of the parse tree starts with the leave, and then it progresses towards its root, which is also known as shift-reduce parsing. Bottom-up parsing can be classified into various parsing. These are as follows:

- Shift-Reduce Parsing
- Operator Precedence Parsing
- Table Driven LR Parsing: LR (1), SLR (1), CLR (1), LALR (1)

1) Operator Precedence Parsing:

An operator precedence parser works like a bottom-up parser that interprets an operator grammar. This parser can be employed only for operator grammars. *Ambiguous grammars are not allowed* in any parser with the exception of operator precedence parser. Operator precedence parsers usually do not store the precedence table with the relations; rather they are implemented in a unique way. Operator precedence parsers use precedence functions that map terminal symbols to integers, and the precedence relations between the symbols are implemented by numerical comparison. One of the popular parser generators which generates operator precedence parser is:

a) PAPAGENO: [4] Papageno is a new parallel parser generator which exploits the properties of operator precedence grammars, also known as Floyd grammars. Such grammars are expressive enough to be used for many existing languages with little or no adjustment. The tool generates automatically a C implementation of the parser given a grammar description provided in Bison compatible syntax, and an additional parameter indicating how many threads are desired. Fig. 4 shows a typical Papageno toolchain.

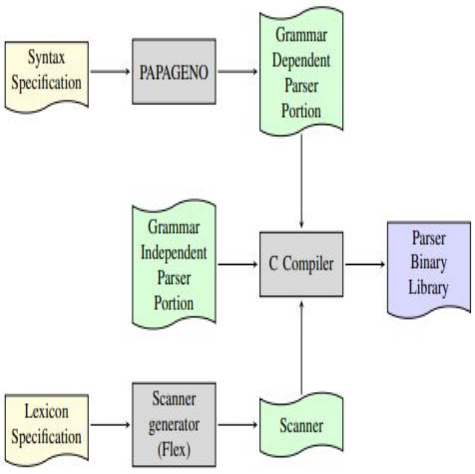


Fig. 4 Papageno toolchain

The implementation of the parser is combined with a lexical scanner gained from the standard scanner generator Flex, to obtain a fully functional parser library, which can be connected with the main application being developed. The parsing process is initiated by invoking the parse call, which receives two parameters: a reference to the input character stream, and the number of workers onto which the parsing process should be split. As depicted in Figure 2, the typical workflow to employ Papageno is analogous to the one of common parser generators such as Yacc/Bison. The user can write two files: a grammar specification, that describes the grammar rules and any semantic actions that are to be performed jointly with reductions, and a lexical specification that describes the terminals or tokens used by the grammar. Papageno can thus be employed as a drop-in replacement for common parser generators, provided that the user checks the form of the language grammar, and removes the possible presence of precedence conflicts in the rules. The syntax of the grammar specification file follows closely the one used by Yacc/Bison. To guide the user during the process of representing the grammar in Floyd form, Papageno offers diagnostic messages pinpointing any existing precedence conflicts between terminal symbols, and it outputs a printable form of the precedence matrix. [5] Papageno has able to successfully generate a full JSON parallel parser, together with a straightforward lexer, proving the practicality of parallel parsing through OPGs of data description languages. Contrary to common belief, it is noted that the parallelization of the lexing phase becomes relevant when dealing with operator precedence parsing, as the running times of the parser and the lexer are comparable for some lightweight syntax languages such as JSON. Papageno has also been able to tackle the parsing of the Lua programming language.

2) Table Driven LR Parsing:

a) LR Parsing: An LR parser reads input text from left to right without backing up and produces a rightmost derivation in reverse: it does a bottom-up parse. The name LR is usually followed by a numeric qualifier, as in LR (1) or sometimes LR(k). In order to avoid backtracking, the LR parser is allowed to peek ahead at k lookahead input symbols prior to deciding how to parse earlier symbols. One of the popular LR (1)/ LR (k) Parser generators is as follows:

- *Essence:* [5] Essence is a generator for LR(k) and SLR(k) parsers in Scheme. The generated parsers perform error recovery, and are highly efficient and accurate. Testing and debugging a parser do not require an edit—generate—compile—test cycle. The generated parsers are a result of the general parser by an automatic program transformation called partial evaluation. This guarantees consistency and ensures correctness. At the heart of parsing is, as usual, a context-free grammar. Essence provides a new syntactic form define-grammar which embeds a language for attributed context-free grammars into Scheme. Given a grammar, parsing can proceed in one of two modes: A general parse procedure will accept a grammar, a parsing method (SLR or LR), a lookahead size, and an input, and produces the result of parsing and attribute evaluation. This mode of operation enables instant turnaround, but also parses very slowly. It is apt for incremental development, but impractical for production parsers. A parser generator produces a unique, specialized parser from a grammar, a parsing method, and a lookahead. The specialized parser only accepts an input as an argument, but is otherwise similar in operation to the general parser. It is highly efficient, and therefore apt for production use. Essence parsers can perform recovery from parsing errors in the manner of Yacc and Bison. The basic idea is that the author of a grammar can specifies special error productions at critical places in a grammar designed to “catch” parsing errors. This allows printing specially tailored error messages, and some control over attribute

evaluation in such a case. The Essence distribution comes with a stand-alone parser generator as well as a batch front-end for easy operation which can be used for Scheme parsing.

b) *Simple LR Parser (SLR):* A Simple LR or SLR parser is a type of LR parser that has small parse tables. The other type of LR (1) parser, an SLR parser, is quite efficient at finding the single correct bottom-up parse in a single left-to-right scan over the input stream, without backtracking. The parser is generated from a formal grammar for the particular language.

- *Tatoo:* [6] Tatoo is a new parser generator. More precisely, given a set of regular expressions describing tokens, a formal specification of a grammar and several semantic hints, Tatoo can generate a lexer, a parser and several implementation glues which allow to run a complete analyzer that creates trees or computes simpler values. Due to a clean separation between specifications, the lexer and the parser may be used independently. Tatoo is written in Java 1.5 and heavily uses parameterized types. For lexing, depending on the character encoding, Tatoo provides three implementations: interval based, table based and switch-case based. For the parsing, depending on the grammar, SLR, LR (1) and LALR (1) implementations are available. For the front-end, the Tatoo engine depends on reified lexer or parser specifications where regular expressions or productions are Java objects. Thus, everything may be implemented in Java. One of the main features of the Java back-end resides in its ability to work in presence of non-blocking inputs such as network connections. Therefore, the Tatoo runtime supports push lexing and parsing. Another innovative feature of Tatoo is its support of language versions. This is provided by two distinct mechanisms. First, productions can be tagged with a version and the Tatoo engine produces a combined parser table tagged with these versions. Then, given a version provided at runtime, the parser selects the correct actions to be performed dynamically. Second, the glue code which links the lexer and the parser, supports dynamic activation of lexer rules according to the parser context. Moreover, Tatoo integrates various other interesting features such as a pluggable error recovery mechanism and multiple character sets optimized support. By default, the lexer and the parser are generated offline by the Tatoo generators using XML specifications. However, it is possible to construct them fully at runtime mentioning everything in Java. Indeed, XML specifications are only available for convenience. Internally, rules or productions are made into Java objects that the developer may or may not construct directly. The parsing process is divided into three steps: specification of the language to be parsed and of several semantic hints, automated implementation of parsing mechanism according to the chosen method and target language and implementation of the semantics which is similar to most of the parsers in existence till date.

c) *Canonical LR Parsing:*A canonical LR parser or LR (1) parser is an LR(k) parser for k=1, i.e., with a single lookahead terminal. The special feature of this parser is that any LR(k) grammar with k>1 can be transformed into an LR (1) grammar. LR(k) can handle all deterministic context-free languages. Recently, a ‘minimal LR (1) parser’ whose space requirements are near to LALR parsers, is offered by several parser generators.

d) *LALR Parsing:* An LALR parser or Look-Ahead LR parser is a simplified version of a canonical LR parser, to parse a text with respect to a set of production rules fixed by a formal grammar for a computer language. ("LR" means left-to-right, rightmost derivation.) The LALR parser is a memory-efficient alternative to the LR (1) parser for languages that are LALR. Some of the LALR Parser generators are as follows:

- *Gold:* [7] GOLD is an acronym for Generalized Object-oriented Language Developer. According to words of Devin Cook, the author of the tool, the GOLD Parser is a free, pseudo-open-source parser generator that one can use to develop one’s own programming languages, scripting languages and interpreters. The actual LALR and DFA algorithms are easy to implement since they depend on tables to determine actions and to traverse between states. Consequently, it is the computation of these tables that is both time-consuming and complex. Unlike usual practise established in common compiler-compilers, the GOLD Parser does not need the developer to embed source code directly into the grammar and mix them. Instead, they stay separated, the application analyzes the grammar and then saves the parse tables to a separate file called compiled grammar file. This file can be subsequently loaded by the actual parser engine and used to parse input. Since the parse tables are programming language independent, the parser engine can be implemented in different programming languages. As a result, the GOLD Parser supports a myriad of programming languages and can be used on multiple platforms. The GOLD parser engine is capable to parse text string and retrieve the information as discussed in previous section. GOLD Parser actually analyze the syntax and identify the classes by tokenized the reserved words and symbols and atoms of the language from the source strings before determining the syntax are sequence and structurally valid.[11] The tool has many advantages such as:

➢ Windows standard feel and look.

- Support the full Basic Multi-lingual Plane of the Unicode characters and as a result parser is not limited to 256 characters in ASCII.
- Export complete tables and sets in HTML format.
- Test the grammar. GOLD Parser Builder offers interactively testing capabilities. One can enter an input string to test the grammar. Further- more it builds and draws the syntax tree in case of success. Input grammar is written using regular expression and is expected in BackusNaur form.
- It is free.

It has many disadvantages which are as follows:

- ✓ A lack of %left %right %non assoc notations for operator precedence and associativity that exist in Yacc. Operator precedence actually comprises of a series of rules. In the case with Yacc, the extra rules needed to implement the proper logic are created "behind the scene". This makes perfect sense for YACC since the additional rules can be hidden from the programmer and the unique logic needed for the parser engine is already implemented.
- ✓ An absence of actions accompanying rule which consist of code executed each time an instance of rule is recognized as it exists in Yacc. GOLD Parser Builder does not support such code as its basic intention to be generator of tables, and not of code.

- *Beaver:* [8] Beaver may be a programme generator for generating LALR parsers from AN EBNF descriptive linguistics specification. LALR stands for Look-Ahead Left-to-right, right derivation and describes however the programme works to use the assembly rules of a language. The number within the parenthesis indicates the number of lookahead tokens, with the foremost common variant being only 1. LALR was developed as another to the LR (1) parser, with the ad-vantage of a smaller memory demand at the expense of some language recognition power. The latest version of Beaver was free in Dec 2012. The beaver specification uses'%' before its directives. These square measure at the start of the specification and specify what terminals and non-terminals the programme uses, in addition as their kind and that pro-duction is that the goal/start production. This is often followed by the specific productions. one thing to notice regarding Beaver specifications is that terminal precedence square measure listed from high to low, and linguistics actions feature a come statement.

III.CONCLUSION

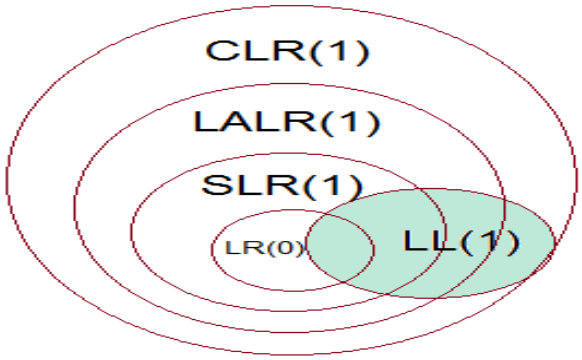


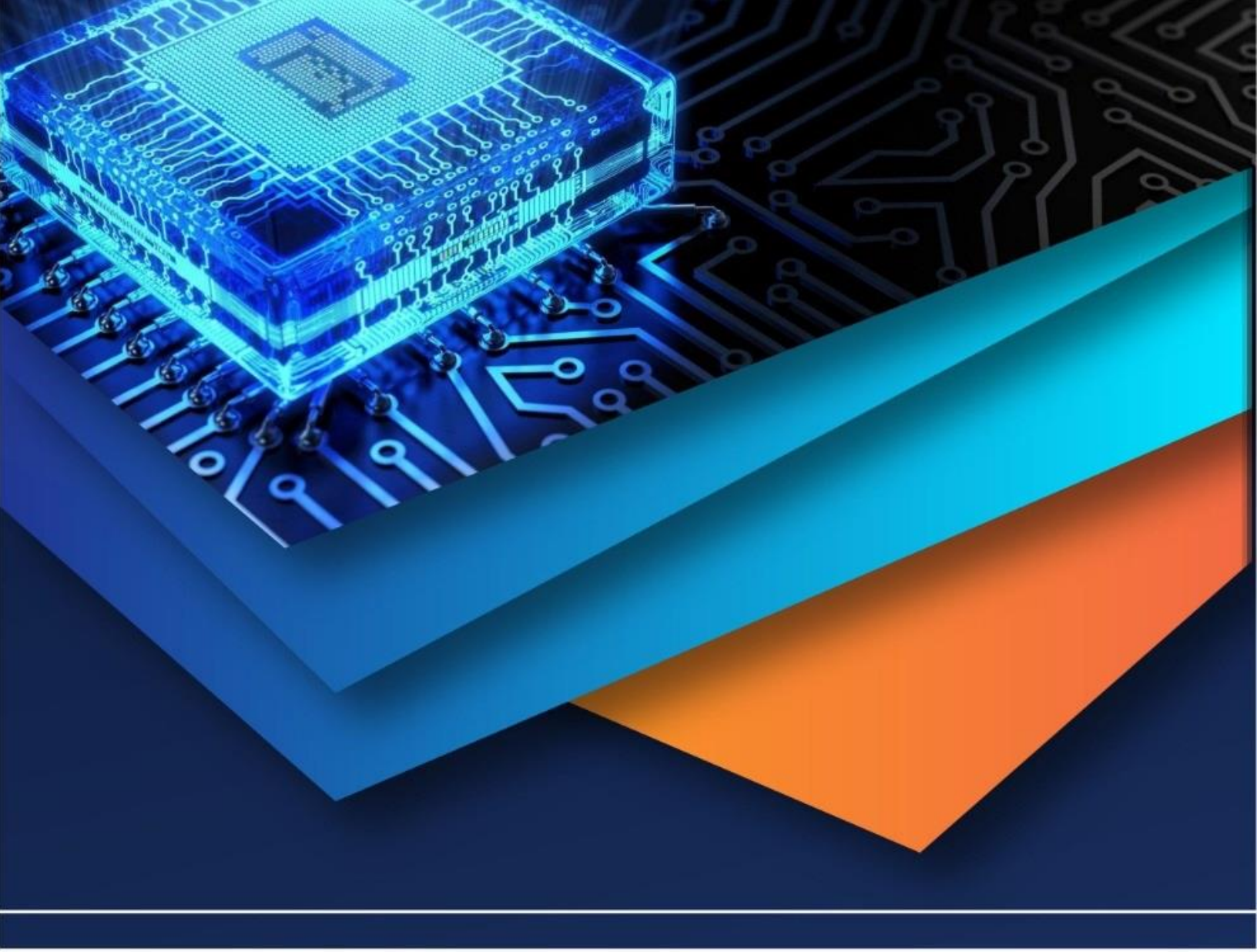
Fig. 5 Power of Parsers

The main conclusion drawn from all the syntax analyzer tools mentioned above is that the influence of the grammar cannot be underestimated. The parser that works best for one grammar may turn out to be the most inefficient one for a different grammar. If one chooses a strong parser generator, one can code the grammar with no worries about specific properties. (LA)LR means one doesn't have to worry about left recursion. GLR means one need not have to be concerned about local ambiguity. An overview of the power of each type of the parser is shown in Fig. 5.

The bottom-up parsers are very efficient. So, once a person has paid the cost of complex machinery, it is easier to write grammars and the parsers perform well. One should expect to see this kind of choice wherever there is some programming construct that commonly occurs: if it is easier to specify, and it performs pretty well, even if the machinery is complicated to work with, complex machinery will win. A crucial factor in determining the best parser is the actual use one wants to make of the parser.

REFERENCES

[1] C. J. H. Jacobs, "LLgen, an extended LL(1) parser generator," retrieved from http://tack.sourceforge.net/olddocs/LLgen.pdf
[2] H. M. Johannes, "The compiler generator Coco/R extended user manual," retrieved from http://norayr.am/tmp/cocor/CocoManual.pdf, June 2004.
[3] L. D. Thomas, "APG, an ABNF parser generator," version 7.0, retrieved from https://sabnf.com/docs/doc7.0/intro.html, June 5, 2022.
[4] A. Barenghi, E. Viviani, S. C. R. D. Mandrioli, and M. Pradella, "PAPAGENO: A parallel parser generator for operator precedence grammars," retrieved from https://re.public.polimi.it/retrieve/handle/11311/709734/217143/main.pdf, 2014.
[5] M. Sperber and P. Thiemann, "Essence - An LR parser generator for scheme," version 2.0, retrieved from https://s48.org/essence/doc/essence.pdf, 2008.
[6] J. Cervelle, R. Forax, and G. Roussel, "Tatoo: An innovative parser generator," 4th International Conference on Principles and Practices of Programming in Java, pp. 13-20, Sept. 1, 2006.
[7] L. S. Ping, W. B. Lin, and N. Jali, "A parsing approach for system behaviour modelling," IADIS International Conference Applied Computing, 2007.
[8] K. Raghavendra, R. Mithuna, and S. A. Alex, "CUP Parser generator for JustAdd(EDAN70)," International Journal of Research in Engineering, Science and Management, vol.2, issue 5, May 2019.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY