International Journal of Research (IJR) Vol-1, Issue-8, September2014 **ISSN 2348-6848**

Parsing – A Brief Study

Tanya Sharma, Sumit Das & Vishal Bhalla

Research Scholars, Computer science and engineering department, MDU Rohtak, India

ABSTRACT

Parsing is a technique to determine how a string might be derived using productions (rewrite rules) of a given grammar. It can be used to check whether or not a string belongs to a given language. When a statement written in a programming language is input, it is parsed by a compiler to check whether or not it is syntactically correct and to extract components if it is correct. Finding an efficient parser is a nontrivial problem and a great deal of research has been conducted on parser design. This paper basically serves the purpose of elaborating the concept of parsing.

1) INTRODUCTION

Parsing or syntactic analysis is the process of analysing a string of symbols, either in natural language or in computer languages, according to the rules of a formal grammar. The term parsing comes from Latin pars (orationis), meaning part (of speech).

The term has slightly different meanings in different branches of linguistics and computer science. Traditional sentence parsing is often performed as a method of understanding the exact meaning of a sentence, sometimes with the aid of devices such as sentence diagrams. It usually emphasizes the importance of grammatical divisions such as subject and predicate.

Within computational linguistics the term is used to refer to the formal analysis by a computer of a sentence or other string of words into its constituents, resulting in a parse tree showing their syntactic relation to each other, which may also contain semantic and other information.

The term is also used in psycholinguistics when describing language comprehension. In this context, parsing refers to the way that human beings analyze a sentence or phrase (in spoken language or text) "in terms of grammatical constituents, identifying the parts of speech, syntactic relations, etc." This term is especially common when discussing what linguistic cues help speakers to interpret garden-path sentences.

Within computer science, the term is used in the analysis of computer languages, referring to the syntactic analysis of the input code into its component parts in order to facilitate the writing of compilers and interpreters.

2) NATURAL LANGUAGE PARSING

2.1) TRADITIONAL METHODS

The traditional grammatical exercise of parsing, sometimes known as clause analysis, involves breaking down a text into its component parts of speech with an explanation of the form, function, and syntactic relationship of each part. This is determined in large part from study of the language's conjugations and declensions, which can be quite intricate

for heavily inflected languages. To parse a phrase such as 'man bites dog' involves noting that the singular noun 'man' is the subject of the sentence, the verb 'bites' is the third person singular of the present tense of the verb 'to bite', and the singular noun 'dog' is the object of the sentence. Techniques such as sentence diagrams are sometimes used to indicate relation between elements in the sentence.

Parsing was formerly central to the teaching of grammar throughout the English-speaking world, and widely regarded as basic to the use and understanding of written language. However the teaching of such techniques is no longer current.

2.2) COMPUTATIONAL METHODS

In some machine translation and natural language processing systems, written texts in human languages are parsed by computer programs. Human sentences are not easily parsed by programs, as there is substantial ambiguity in the structure of human language, whose usage is to convey meaning (or semantics) amongst a potentially unlimited range of possibilities but only some of which are germane to the particular case. So an utterance "Man bites dog" versus "Dog bites man" is definite on one detail but in another language might appear as "Man dog bites" with a reliance on the larger context to distinguish between those two possibilities, if **indeed** that difference was of concern. It is difficult to prepare formal rules to describe informal behaviour even though it is clear that some rules are being followed.

In order to parse natural language data, researchers must first agree on the grammar to be used. The choice of syntax is affected by both linguistic and computational concerns; for instance some parsing systems use lexical functional grammar, but in general, parsing for grammars of this type is known to be NP-complete. Head-driven phrase structure grammar is another linguistic formalism which has been popular in the parsing community, but other research efforts have focused on less complex formalisms such as the one used in the Penn Treebank. Shallow parsing aims to find only the boundaries of major constituents such as noun phrases. Another popular strategy for avoiding linguistic controversy is dependency grammar parsing.

Most modern parsers are at least partly statistical; that is, they rely on a corpus of training data which has already been annotated (parsed by hand). This approach allows the system to gather information about the frequency with which various constructions occur in specific contexts. Approaches which have been used include straightforward PCFGs (probabilistic context-free grammars), maximum entropy, and neural nets. Most of the **more** successful systems use lexical statistics (that is, they consider the identities of the words involved, as well as their part of speech). However such systems are vulnerable to over-fitting and require some kind of smoothing to be effective.

Parsing algorithms for natural language cannot rely on the grammar having 'nice' properties as with manually designed grammars for programming languages. As mentioned earlier some grammar formalisms are very difficult to parse computationally; in general, even if the desired structure is not context-free, some kind of context-free approximation to the grammar is used to perform a first pass. Algorithms which use context-free

grammars often rely on some variant of the CKY algorithm, usually with some heuristic to prune away unlikely analyses to save time. However some systems trade speed for accuracy using, e.g., linear-time versions of the shift-reduce algorithm. A somewhat recent development has been parse re-ranking in which the parser proposes some large number of analyses, and a more complex system selects the best option.

3) COMPUTER LEVEL PARSING

3.1) PARSER

A parser is a software component that takes input data (frequently text) and builds a data structure – often some kind of parse tree, abstract syntax tree or other hierarchical structure – giving a structural representation of the input, checking for correct syntax in the process. The parsing may be preceded or followed by other steps, or these may be combined into a single step. The parser is often preceded by a separate lexical analyser, which creates tokens from the sequence of input characters; alternatively, these can be combined in scannerless parsing. Parsers may be programmed by hand or may be automatically or semi-automatically generated by a parser generator. Parsing is complementary to templating, which produces formatted *output.* These may be applied to different domains, but often appear together, such as the scanf/printf pair, or the input (front end parsing) and output (back end code generation) stages of a compiler.

The input to a parser is often text in some computer language, but may also be text in a natural language or less structured textual data, in which case generally only certain parts of the text are extracted, rather than a parse tree being constructed. Parsers range from very simple functions such as scanf, to complex programs such as the frontend of a C++ compiler or the HTML parser of a web browser. An important class of simple parsing is done using regular expressions, where a regular expression defines a regular language, and then the regular expression engine automatically generates a parser for that language, allowing pattern matching and extraction of text. In other contexts regular expressions are instead used prior to parsing, as the lexing step whose output is then used by the parser.

The use of parsers varies by input. In the case of data languages, a parser is often found as the file reading facility of a program, such as reading in HTML or XML text; these examples are markup languages. In the case of programming languages, a parser is a component of a compiler or interpreter, which parses the source code of a computer programming language to create some form of internal representation; the parser is a key step in the compiler frontend. Programming languages tend to be specified in terms of a deterministic context-free grammar because fast and efficient parsers can be written for them. For compilers, the parsing itself can be done in one pass or multiple passes – see one-pass compiler and multi-pass compiler.

The implied disadvantages of a one-pass compiler can largely be overcome by adding fix-ups, where provision is made for fix-ups during the forward pass, and the fix-ups are applied backwards when the current program segment has been recognized as having been completed. An example where such a fix-up mechanism would be useful would be a forward GOTO statement, where the target of the GOTO is unknown until the

program segment is completed. In this case, the application of the fix-up would be delayed until the target of the GOTO was recognized. Obviously, a backward GOTO does not require a fix-up.

Context-free grammars are limited in the extent to which they can express all of the requirements of a language. Informally, the reason is that the memory of such a language is limited. The grammar cannot remember the presence of a construct over an arbitrarily long input; this is necessary for a language in which, for example, a name must be declared before it may be referenced. More powerful grammars that can express this constraint, however, cannot be parsed efficiently. Thus, it is a common strategy to create a relaxed parser for a context-free grammar which accepts a superset of the desired language constructs (that is, it accepts some invalid constructs); later, the unwanted constructs can be filtered out at the semantic analysis (contextual analysis) step.

3.2) OVERVIEW OF PROCESS

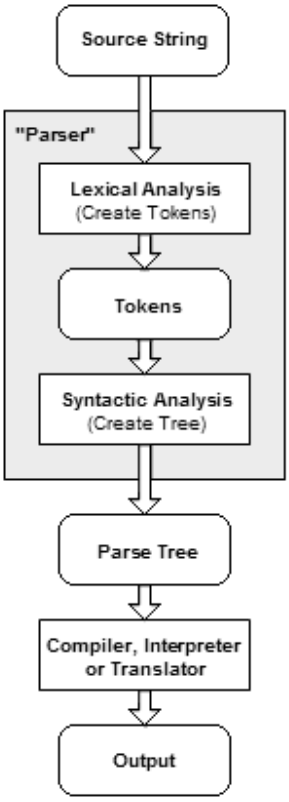

Fig 1)

4) TYPES OF PARSING

The task of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. This can be done in essentially two ways:

- **TOP-DOWN PARSING**- top-down parsing is a parsing strategy where one first looks at the highest level of the parse tree and works down the parse tree by using the rewriting rules of a formal grammar. LL parsers are a type of parser that uses a top-down parsing strategy.
  Top-down parsing is a strategy of analyzing unknown data

relationships by hypothesizing general parse tree structures and then considering whether the known fundamental structures are compatible with the hypothesis. It occurs in the analysis of both natural languages and computer languages.

Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse-trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules.

Simple implementations of top-down parsing do not terminate for left-recursive grammars, and top-down parsing with backtracking may have exponential time complexity with respect to the length of the input for ambiguous CFGs. However, more sophisticated top-down parsers have been created by Frost, Hafiz, and Callaghan which do accommodate ambiguity and left recursion in polynomial time and which generate polynomial-sized representations of the potentially exponential number of parse trees.

- **Bottom-up parsing** - parsing reveals the grammatical structure of linear input text, as a first step in working out its meaning. Bottom-up parsing identifies and processes the text's lowest-level small details first, before its mid-level structures, and leaving the highest-level overall structure to last.

4.1) TOP DOWN VERSUS BOTTOM UP PARSING

The bottom-up name comes from the concept of a parse tree, in which the most detailed parts are at the bushy bottom of the (upside-down) tree, and larger structures composed from them are in successively higher layers, until at the top or "root" of the tree a single unit describes the entire input stream. A bottom-up parse discovers and processes that tree starting from the bottom left end, and incrementally works its way upwards and rightwards. A parser may act on the structure hierarchy's low, mid, and highest levels without ever creating an actual data tree; the tree is then merely implicit in the parser's actions. Bottom-up parsing lazily waits until it has scanned and parsed all parts of some construct before committing to what the combined construct is.

The opposite of this are top-down parsing methods, in which the input's overall structure is decided (or guessed at) first, before dealing with mid-level parts, leaving the lowest-level small details to last. A top-down parser discovers and processes the hierarchical tree starting from the top, and incrementally works its way downwards and rightwards. Top-down parsing eagerly decides what a construct is much earlier, when it has only scanned the leftmost symbol of that construct and has not yet parsed any of its parts. Left corner parsing is a hybrid method which works bottom-up along the left edges of each subtree, and top-down on the rest of the parse tree.

If a language grammar has multiple rules that may start with the same leftmost symbols but have different endings, then that grammar can be efficiently handled by a deterministic bottom-up parse but cannot be handled top-down without guesswork and backtracking. So bottom-up parsers handle a somewhat larger

range of computer language grammars than do **deterministic** top-down parsers.

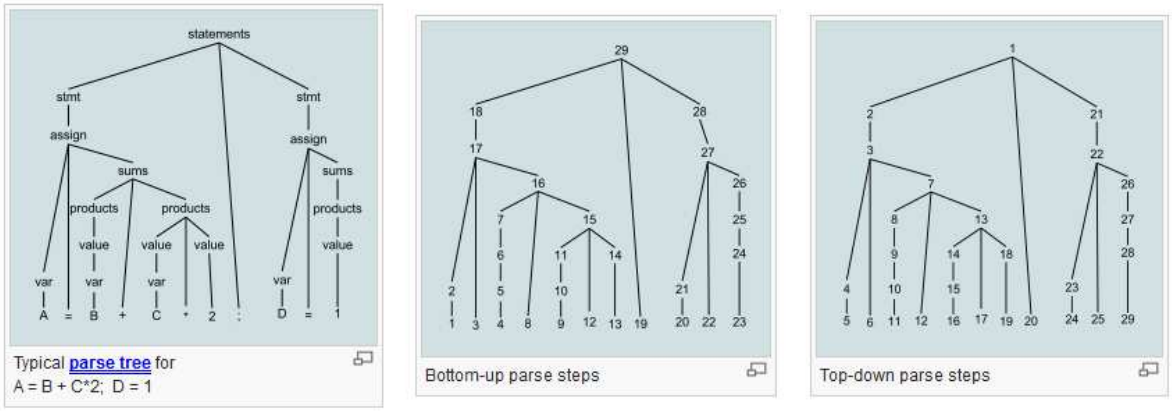

Fig 2)

5) TYPES OF PARSERS

5.1) TOP-DOWN PARSERS

Some of the parsers that use top-down parsing include:

- **Recursive descent parser**- a recursive descent parser is a kind of top-down parser built from a set of mutually recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

  A predictive parser is a recursive descent parser that does not require backtracking. Predictive parsing is possible only for the class of LL(k) grammars, which are the context-free grammars for which there exists some positive integer k that allows a recursive descent parser to decide which production to use by examining only the next k tokens of input. (The LL(k) grammars therefore exclude all ambiguous grammars, as well as all grammars that contain left recursion. Any context-free grammar can be transformed into an equivalent grammar that has no left recursion, but removal of left recursion does not always yield an LL(k) grammar.) A predictive parser runs in linear time.

  Recursive descent with backtracking is a technique that determines which production to use by trying each production in turn. Recursive descent with backtracking is not limited to LL(k) grammars, but is not guaranteed to terminate unless the grammar is LL(k). Even when they terminate, parsers that use recursive descent with backtracking may require exponential time.

  Although predictive parsers are widely used, and are frequently chosen if writing a parser by hand, programmers often prefer to use a table-based parser produced by a parser generator, either for an LL(k) language or using an alternative parser, such as LALR or LR. This is

particularly the case if a grammar is not in LL(k) form, as transforming the grammar to LL to make it suitable for predictive parsing is involved. Predictive parsers can also be automatically generated, using tools like ANTLR.

- **LL parser (Left-to-right, Leftmost derivation)** - An **LL parser** is a top-down parser for a subset of the context-free grammars. It parses the input from **L**eft to right, and constructs a **L**eftmost derivation of the sentence (hence LL, compared with LR parser). The class of grammars which are parsable in this way is known as the *LL grammars*.
  The remainder of this article describes the table-based kind of parser, the alternative being a recursive descent parser which is usually coded by hand (although not always; see e.g. ANTLR for an LL(*) recursive-descent parser generator).
  An LL parser is called an LL(*k*) parser if it uses *k* tokens of lookahead when parsing a sentence. If such a parser exists for a certain grammar and it can parse sentences of this grammar without backtracking then it is called an LL(*k*) grammar. A language that has an LL(k) grammar is known as an LL(k) language. There are LL(k+n) languages that are not LL(k) languages. A corollary of this is that not all context-free languages are LL(k) languages.
  LL(1) grammars are very popular because the corresponding LL parsers only need to look at the next token to make their parsing decisions. Languages based on grammars with a high value of *k* have traditionally been considered to be difficult to parse, although this is less true now given the availability and widespread use of parser generators supporting LL(*k*) grammars for arbitrary *k*.
  An LL parser is called an LL(*) parser if it is not restricted to a finite *k* tokens of lookahead, but can make parsing decisions by recognizing whether the following tokens belong to a regular language (for example by use of a Deterministic Finite Automaton).

5.2) BOTTOM-UP PARSERS

Some of the parsers that use bottom-up parsing include:

- **Operator-precedence parser** - While ad-hoc methods are sometimes used for parsing expressions, a more formal technique using operator precedence simplifies the task. For example an expression such as

  (x*x + y*y)^.5

  is easily parsed. Operator precedence parsing is based on bottom-up parsing techniques and uses a precedence table to determine the next action. The table is easy to construct and is typically hand-coded. This method is ideal for applications that require a parser for expressions and where embedding compiler technology, such as yacc, would be overkill.

- **LR PARSER (LEFT-TO-RIGHT, RIGHTMOST DERIVATION)**

  - **Simple LR (SLR) parser** - The most prevalent type of bottom-up

parser today is based on a concept called LR(k) parsing; the "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions. The cases k = 0 or k = 1 are of practical interest, and we shall only consider LR parsers with k ≤ 1 here. When (k) is omitted, k is assumed to be 1.

Some familiarity with the basic concepts is helpful even if the LR parser itself is constructed using an automatic **parser generator**. We begin with "items" and "parser states"; the diagnostic output from an LR parser generator typically includes parser states, which can be used to isolate the sources of parsing conflicts.

LR parsing is attractive for a variety of reasons:

1. LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written. NonLR context-free grammars exist, but these can generally be avoided for typical programming-language constructs.
2. The LR-parsing method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other, more primitive shift-reduce methods (see the bibliographic notes).
3. An LR parser can detect a syntactic **error** as soon as it is possible to do so on a left-to-right scan of the input.
4. The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with **predictive** or LL methods. For a grammar to be LR(k), we must be able to recognize the occurrence of the right side of a production in a right-sentential form, with k input symbols of lookahead. This requirement is far less stringent than that for LL(k) grammars where we must be able to recognize the use of a production seeing only the first k symbols of what its right side derives. Thus, it should not be surprising that LR grammars can describe more languages than LL grammars.

- **LALR parser** - LALR parsers are based on a finite-state-automata concept, from which they derive their speed. The data structure used by an LALR parser is a pushdown automaton (PDA). A deterministic PDA is a deterministic-finite automaton (DFA) that uses a stack for a memory, indicating which states the parser has passed through to arrive at the current state. Because of the stack, a PDA can recognize grammars that would be impossible with a DFA; for example, a PDA can determine

whether an expression has any unmatched parentheses, whereas an automaton with no stack would require an infinite number of states due to unlimited nesting of parentheses.

LALR parsers are driven by a parser table in a finite-state machine (FSM) format. An FSM is very difficult for humans to construct and therefore an LALR parser generator is used to create the parser table automatically from a grammar in Backus–Naur Form which defines the syntax of the computer language the parser will process. The parser table is often generated in source code format in a computer language (such as C++ or Java). When the parser (with parser table) is compiled and/or executed, it will recognize text files written in the language defined by the BNF grammar.

LALR parsers are generated from LALR grammars, which are capable of defining a larger class of languages than SLR grammars, but not as large a class as LR grammars. Real computer languages can often be expressed as LALR(1) grammars, and in cases where a LALR(1) grammar is insufficient, usually an LALR(2) grammar is adequate. If the parser generator handles only LALR(1) grammars, then the LALR parser will have to interface with some hand-written code when it encounters the special LALR(2) situation in the input language.

- **Canonical LR (LR(1)) parser** - A canonical LR parser or LR(1) parser is an LR parser whose parsing tables are constructed in a similar way as with LR(0) parsers except that the items in the item sets also contain a *lookahead*, i.e., a terminal that is expected by the parser after the right-hand side of the rule. For example, such an item for a rule A → B C might be

  which would mean that the parser has read a string corresponding to B and expects next a string corresponding to C followed by the terminal 'a'. LR(1) parsers can deal with a very large class of grammars but their parsing tables are often very big. This can often be solved by merging item sets if they are identical except for the lookahead, which results in so-called LALR parsers.

- **GLR parser** - A **GLR parser** (GLR standing for "generalized LR", where L stands for "left-to-right" and R stands for "rightmost (derivation)") is an extension of an LR parser algorithm to handle nondeterministic and ambiguous grammars. The theoretical foundation was provided in a 1974 paper by Bernard Lang (along with other general Context-Free parsers such as GLL). It describes a systematic way to produce such algorithms, and provides uniform results regarding correctness proofs, complexity with respect to grammar classes, and optimization techniques.

SUMMARY

Parsing' is the term used to describe the process of automatically building syntactic analyses of a sentence in terms of a given grammar and lexicon. The resulting

syntactic analyses may be used as input to a process of semantic interpretation,(or perhaps phonological interpretation, where aspects of this, like prosody, are sensitive to syntactic structure). Occasionally, 'parsing' is also used to include both syntactic and semantic analysis. We use it in the more conservative sense here, however. In most contemporary grammatical formalisms, the output of parsing is something logically equivalent to a tree, displaying dominance and precedence relations between constituents of a sentence, perhaps with further annotations in the form of attribute-value equations ('features') capturing other aspects of linguistic description. However, there are many different possible linguistic formalisms, and many ways of representing each of them, and hence many different ways of representing the results of parsing. We shall assume here a simple tree representation, and an underlying context-free grammatical (CFG) formalism. However, all of the algorithms decribed here can usually be used for more powerful unification based formalisms, provided these retain a context-free 'backbone', although in these cases their complexity and termination properties may be different.

Parsing algorithms are usually designed for classes of grammar rather than tailored towards individual grammars. There are several important properties that a parsing algorithm should have if it is to be practically useful. It should be 'sound' with respect to a given grammar and lexicon; that is, it should not assign to an input sentence analyses which cannot arise from the grammar in question. It should also be 'complete'; that it, it should assign to an input sentence all the analyses it can have with respect to the current grammar and lexicon. Ideally, the algorithm should also be 'efficient', entailing the minimum of computational work consistent with fulfilling the first two requirements, and 'robust': behaving in a reasonably sensible way when presented with a sentence that it is unable to
Full, analyze successfully.

DISCLOSURE STATEMENT
There is no financial support for this research work from the funding agency.

ACKNOWLEDGMENTS
We thank our guide for his timely help, giving outstanding ideas and encouragement to finish this research work successfully.

SIDE BAR

Comparison: it is an act of assessment or evaluation of things side by side in order to see to what extent they are similar or different. It is used to bring out similarities or differences between two things of same type mostly to discover essential features or meaning either scientifically or otherwise.

Content: The amount of things contained in something. Things written or spoken in a book, an article, a programme, a speech, etc.

DEFINITION

- **Mutual** - held in common by two or more parties.
- **Hierarchical** - a system in which members of an organization or society are ranked according to relative status or authority.
- **Segment** - each of the parts into which something is or may be divided.
- **Component** - a part or element of a larger whole, especially a part of a machine or vehicle.

REFERENCES

[1] "Bartleby.com homepage". Retrieved 28 November 2010.

[2] "parse". dictionary.reference.com. Retrieved 27 November 2010.

[3] "Grammar and Composition".

[4] Aho, A.V., Sethi, R. and Ullman ,J.D. (1986) " Compilers: principles, techniques, and tools." *Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.*

[5] Frost, R., Hafiz, R. and Callaghan, P. (2007) " Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars ." *10th International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSE* , Pages: 109 - 120, June 2007, Prague.

[6] Frost, R., Hafiz, R. and Callaghan, P. (2008) " Parser Combinators for Ambiguous Left-Recursive Grammars." *10th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN* , Volume 4902/2008, Pages: 167 - 181, January 2008, San Francisco.

[7] Knuth, D. E. (July 1965). "On the translation of languages from left to right". *Information and Control* **8** (6): 607–639. doi:10.1016/S0019-9958(65)90426-2. Retrieved 29 May 2011. edit

[8] Language theoretic comparison of LL and LR grammars

[9] Engineering a Compiler (2nd edition), by Keith Cooper and Linda Torczon, Morgan Kaufman 2011.

[10] Crafting and Compiler, by Charles Fischer, Ron Cytron, and Richard LeBlanc, Addison Wesley 2009.

[11] Flex & Bison: Text Processing Tools, by John Levine, O'Reilly Media 2009.

[12] Compilers: Principles, Techniques, and Tools (2nd Edition), by Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman, Prentice Hall 20