Journal Of Harmonized Research (JOHR)

Journal Of Harmonized Research in Engineering
1(2), 2013, 73-79

ISSN 2347 – 7393

Original Research Article

PARSING: PROCESS OF ANALYZING WITH THE RULES OF A FORMAL GRAMMAR

Nikita Chhillar, Nisha Yadav, Neha Jaiswal

Department of Computer Science and Engineering,
Dronacharya College of Engineering, Khentawas,
Farukhnagar, Gurgaon, India

Abstract:
Parsing is the process of structuring a linear representation in accordance with a given grammar. The "linear representation" may be a sentence, a computer program, knitting pattern, a sequence of geological strata, a piece of music, actions in ritual behavior, in short any linear sequence in which the preceding elements in some way restrict† the next element. For some of the examples the grammar is well-known, for some it is an object of research and for some our notion of a grammar is only just beginning to take shape.

Keywords: Parsing, Grammar, Bottom-up parsing, Top-down parsing, Parser.

Introduction:

Parsing or **syntactic analysis** is the process of analyzing 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* 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. Within computational linguistics the term is used to refer to the formal analysis by 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.

For Correspondence:
nikitachhillarATyahoo.com
Received on: October 2013
Accepted after revision: December 2013
Downloaded from: www.johronline.com

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.

1) Traditional methods:

The traditional grammatical exercise of parsing, are known as *clause analysis*, involves splitting a text into its component parts of speech with an explanation of the form, purpose, and syntactic relationship of each part. This is determined in many part from study of the language's conjugations and declensions, which can be quite complex for heavily inflected languages. To parse a phrase such as 'Kittu saw monkey' involves noting that the singular noun 'Kittu' is the subject of the sentence, the verb 'saw' is the third person singular of the past tense of the verb 'to see', and the singular noun 'monkey' is the object of the sentence. Techniques such as sentence diagrams are used to indicate relation between elements in the sentence.

2) Computational methods:

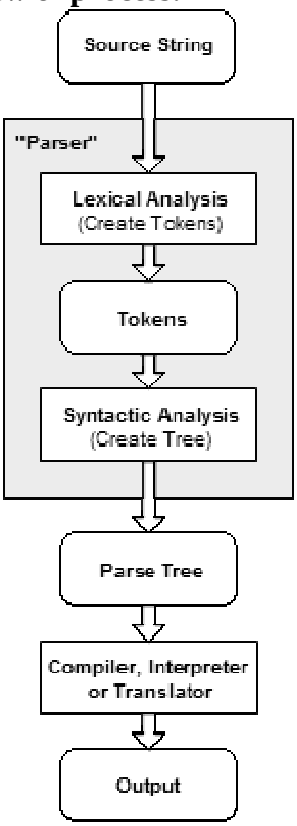
In some machine translation and natural language processing systems, written texts in human languages are parsed by computer programs. Human sentences are not easily parsed via programs, as there is substantial ambiguity inside the structure of human language, whose usage is to convey meaning (or semantics) in a potentially unlimited range of possibilities however only some of which are germane to the particular case. So an utterance "Kittu saw monkey" versus "Monkey saw Kittu" is definite on one detail but in another language might appear as "Kittu monkey saw" 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 behavior even though it is clear that some rules are being followed.

To parse natural language data, researchers should first have same opinion on the grammar to be used. The selection of syntax is affected by both linguistic and computational concerns; for example some parsing systems make use of lexical functional grammar, whereas in general, parsing for grammars of this type is identified as NP-complete. Head-driven phrase structure grammar is another linguistic formalism that has been accepted in the parsing community, but further research efforts have focused on simple formalisms such as the one used in the Penn Treebank. Shallow parsing aims to locate only the boundaries of major constituents like noun phrases. Another admired strategy for avoiding linguistic controversy is dependency grammar parsing.

Most modern parsers are at least partially statistical; that is, they rely on a body of training data which has already been interpreted (parsed by hand). This approach permits the system to gather information about the frequency with which different constructions occur in specific contexts. Approaches which have been used consist of straightforward PCFGs (probabilistic context-free grammars), maximum entropy, and neural nets. Most of the 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 need some kind of smoothing to be effective.

Parsing algorithms used for natural language cannot rely on the grammar having 'good' properties as with manually designed grammars for programming languages. As mentioned before some grammar formalisms are very difficult to parse computationally; in general, even if the desired structure is not context-free, some type of context-free approximation to the grammar is used to perform a first pass. Algorithms which make use context-free grammars often rely on some alternative of the CKY algorithm, usually with some heuristic to prune away unlikely analyses to keep time. However some systems trade speed for accurateness using, example linear-time versions of the shift-reduce algorithm. A recent development has been parse reranking that the parser proposes several large numbers of analyses, and a more complex system picks the best option.

3) Overview of process:



These examples demonstrate the general case of parsing a computer language by two levels of grammar: lexical and syntactic.

The first step is the token generation, or lexical analysis, in which the input character stream is split into meaningful symbols defined with a grammar of regular expressions. For example, a calculator program would come across an input such as "12*(3+4^2" and split this into the tokens 12, *, (, 3, +, 4,), ^, 2, each of which is a significant symbol in the context of an arithmetic expression. The lexer would contain rules that tell it, the characters *, +, ^, (and) mark the beginning of a new token, so meaningless tokens such as "12*" or "(3" will not be generated).

The next stage is parsing or syntactic analysis, which examines that the tokens form an acceptable expression. This is generally done with reference to a context-free grammar which recursively defines parts that can make up an expression and the arrangement in which they must appear. However, not all rules defining programming languages can be conveyed by context-free grammars alone, for example type validity and proper declaration of identifiers. These rules can be formally conveyed with attribute grammars.

The last phase is semantic parsing or analysis that works out the implications of the expression just validated and taking the suitable action. Calculator or interpreter evaluates the expression or program, a compiler, would generate some kind of code. Attribute grammars can also be used to describe these actions.

Types of parsing:

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

- Top-down parsing- Top-down parsing can be viewed as an approach to find left-most origin of an input-stream by searching for parse trees using a top-down extension of the given formal grammar rules. Tokens are used from left to right. Complete choice is used to hold ambiguity by expanding every alternative right-hand-side of grammar rules.
- Bottom-up parsing - A parser can initiate with the input and approach to rewrite it to the start symbol. Intuitively, the parser attempts to place the most basic elements, then the elements include these, and so on. LR parsers are instances of bottom-up parsers. Another term used for this sort of parser is Shift-Reduce parsing.

LL parsers and recursive-descent parser are examples of top-down parsers that cannot accommodate left recursive production rules. Although it has been assumed that simple implementations of top-down parsing cannot hold direct and indirect left-recursion and may need exponential time and space complexity as parsing ambiguous context-free grammars, extra sophisticated algorithms for top-down parsing have been produced by Frost, Hafiz, and Callaghan that accommodate ambiguity and left recursion in polynomial time and that generate polynomial-size depictions of the potentially exponential number of parse trees. Their algorithm is capable to produce both left-most and right-most derivations of an input with respect to a given CFG (context-free grammar).

An important dissimilarity with regard to parsers is whether a parser produces a *leftmost derivation* or a *rightmost derivation.* LL parsers will produce a leftmost derivation and LR parsers will produce a rightmost derivation (although usually in reverse).

Top-down parsing:

Top-down parsing is a parsing approach where one begins with the top most level of the parse tree and works downward by using the rewriting rules for a formal grammar. LL parsers are a kind of parser that makes use of top-down parsing approach.

- Top-down parsing is a strategy of studying unknown data relationships by hypothesizing general parse tree structure and then considering the known fundamental structures are well-suited with the hypothesis. It occurs in the study of both natural languages and computer languages.
- Top-down parsing can be viewed as an shot 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 used from left to right. Inclusive choice is used to hold ambiguity by expanding all unconventional right-hand-sides of grammar rules.
- Simple executions of top-down parsing do not end for left-recursive grammars, and top-down parsing by backtracking may have exponential time complexity with respect to the length of the input for ambiguous CFGs. However, more complicated top-down parsers have been created by Frost, Hafiz, and Callaghan which do hold ambiguity and left recursion within polynomial time and which generate polynomial-sized illustrations of the potentially exponential number of parse trees

Programming language application:

A compiler parses input to assembly language from a programming language or an internal representation by harmonizing the incoming symbols to production rules. Production rules are defined using Backus-Naur form. An LL parser is a kind of parser that does top-down parsing by applying each production rule to the received symbols, working from the left-most symbol give way a production rule and then proceeding to the next production rule for every non-terminal symbol encountered. In this way the parsing begins on the Left of the result side (right side) of the production rule and calculates non-terminals from the Left first and, thus, moves down the parse tree for every new non-terminal before continuing to the another symbol for a production rule.

For example:

- $A \rightarrow aBC$
- $B \rightarrow c \mid cd$
- $C \rightarrow df \mid eg$

would match $A \rightarrow aBC$ and attempt to match $B \rightarrow c \mid cd$ next. Then $C \rightarrow df \mid eg$ would be tried. As one may suppose, some languages are extra ambiguous than others. For a non-ambiguous language that has all productions for non-terminal produce different strings: the string produced by one production will not begin with the similar symbol as the string produced by another production. A non-ambiguous language might be parsed by an LL (1) grammar where the (1) implies the parser reads ahead one token at a time. For an ambiguous language to be parsed by an LL parser, the parser must look ahead more than 1 symbol, e.g. LL (3).

The common way out to this problem is to use an LR parser, which is a kind of shift-reduce parser, and does bottom-up parsing.

Accommodating left recursion in top-down parsing:

A formal grammar which contains left recursion cannot be parsed by a naive recursive descent parser if they are not converted to a weakly equivalent right-recursive form. However, recent research reveals that it is possible to hold left-recursive grammars (along with all other forms of general CFGs) in a more complicated top-down parser by use of curtailment. A detection algorithm which accommodates ambiguous grammars and curtails an ever-emergent direct left-recursive parse by imposing depth limitations with respect to input length and recent input position, is

described by Frost and Hafiz in 2006. That algorithm was extended to a absolute parsing algorithm to accommodate indirect (by comparing earlier computed context with current context) with direct left-recursion in polynomial time, and to generate compact polynomial-size demonstrations of the potentially exponential number of parse trees for highly ambiguous grammars by Frost, Hafiz and Callaghan in 2007. The algorithm has since been executed as a set of parser combinatory written in the Haskell programming language.

Time and space complexity of top-down parsing:

When top-down parser tries to parse an ambiguous input with respect to an ambiguous CFG, it may require exponential number of ladder (with respect to the length of the input) to try all substitutes of the CFG in order to create all possible parse trees, which in due course would require exponential memory space. The problem of exponential time complexity in top-down parsers created as sets of mutually recursive functions has been explained by Norvig in 1991. His technique is similar to the use of dynamic programming and state-sets in Earley's algorithm (1970), and tables of the CYK algorithm of Cocke, Younger and Kasami.

The main idea is to store results of applying a parser p at point j in a memorable and to reuse result whenever the same situation arises. Frost, Hafiz and Callaghan also use memorization for refraining redundant computations for accommodating any form of CFG in polynomial time (Θ (n^4) for left-recursive grammars and Θ (n^3) for non left-recursive grammars). Their top-down parsing algorithm also needs polynomial space for potentially exponential ambiguous parse trees by 'compressed representation' and 'local ambiguities grouping'.

Bottom-up parsing:

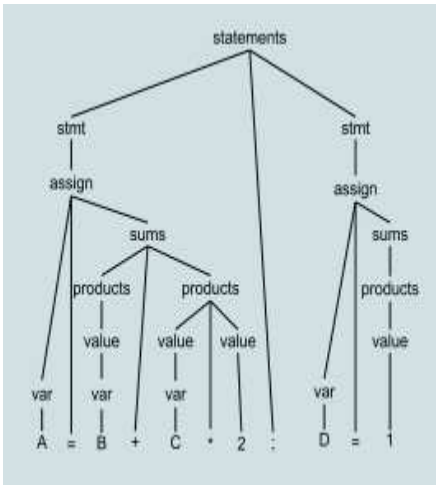
Parsing tells the grammatical structure of linear input text, as a primary step in working out its meaning. **Bottom-up parsing** discovers and processes the text's lowest-level small elements first, before its mid-level structures, and leaving the highest-level whole structure to last.

Bottom-up parsing works in the reverse direction from top-down parsing. A top-down parser initiate with the start symbol at the top of the parse tree and works downward, driving productions in forward direction until it gets to the terminal leaves. A bottom-up parse begins with the string of terminals itself and build from the leaves upward, working in reverse to the start symbol by applying the productions. Then a bottom-up parser searches for substrings of the working string which matches the right side of some production. When it finds such a substring, it substitutes the left side non-terminal for the matching right side. The goal is to lessen all the way till the start symbol and report a successful parsing. Generally, bottom-up parsing algorithms are more authoritative than top-down methods, but not surprisingly, the construction required is also more complex. It's difficult to write a bottom-up parser by hand for anything but trivial grammar, but fortunately, there are excellent parser generator tools like **yacc** that build a parser from an input specification, not unlike the way **lex** build a scanner to your spec. *Shift-reduce* parsing is the commonly used and powerful of the bottom-up techniques. Bottom-up parsing takes as input a stream of tokens and develops the list of productions used for building the parse tree, but the production is discovered in reverse order of a top down parser. Like a table-driven predictive parser, it makes use of a stack to keep track of the position in the parse and a parsing table to determine what to do next.

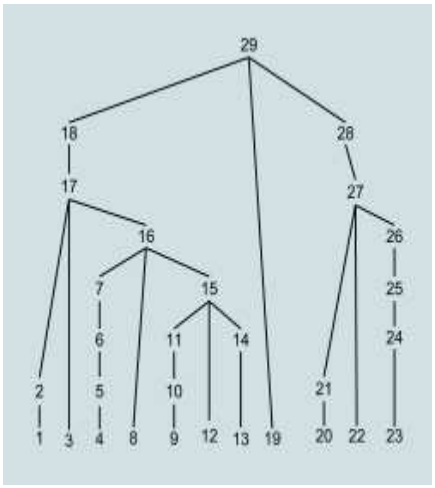
Bottom-up Versus Top-down:

The bottom-up name originally comes from the concept of a parse tree, in which the detailed parts are at the bushy bottom of the (upside-down) tree, and larger structures collected from them are in successively higher layer, until at the top or "root" of the tree a single unit explains the entire input stream. A bottom-up parse processes that tree starting from the bottom left end, and incrementally work 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.

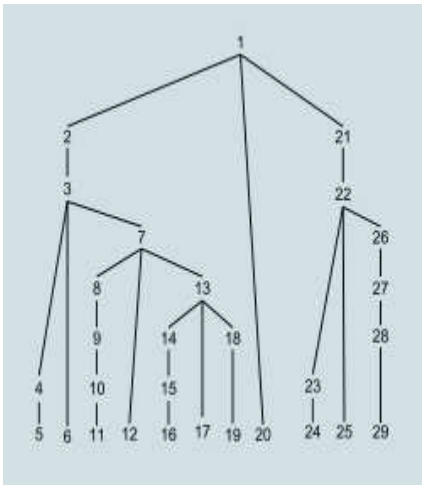
Bottom-up parsing lazily waits until it has scanned and parsed all parts of some construct before committing to what the combined construct is.



Typical parse tree for A = B + C*2; D = 1



Bottom-up parse steps



Top-down parse steps

The opposite of **bottom-up parsing methods** are **top-down parsing** methods, in which the input's almost structure is decided (or guessed at) first, before dealing with mid parts, leaving the lowest small details to last. In top-down parse processes the hierarchical tree starting from the top, and incrementally work downwards and rightwards. Top-down parsing keenly decides what a construct is much earlier, when it has only scanned the leftmost symbols 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 edge of each subtree, and top-down on the rest of the parse tree.

If grammar has multiple rules which start with the same leftmost symbol but have different endings, then the grammar can be handled by a deterministic bottom-up parse but cannot be handled by top-down without guesswork and backtracking. So bottom-up parsers handle a larger range of computer language grammar than do deterministic top-down parsers.

Bottom-up parsing is every now and then done by backtracking. But generally, bottom-up parsing is done by a **shift-reduce parser** such as a LALR parser.

4) Top-down parsers:

Parsers which use top-down parsing are:

- Recursive descent parser
- LL parser (**L**eft-to-right, **L**eftmost derivation)
- Earley parser

5) Bottom-up parsers:

Parsers which use bottom-up parsing are:

- Precedence parser
  - Operator-precedence parser
  - Simple precedence parser
- BC (bounded context) parsing
- LR parser (**L**eft-to-right, **R**ightmost derivation)
  - Simple LR (SLR) parser
  - LALR parser
  - Canonical LR (LR(1)) parser
  - GLR parser
- CYK parser
- Recursive ascent parser

Conclusions:

Parsing splits a sequence of characters or letters into smaller parts. Parsing is also used for recognizing characters or letters that occur in a specific order. In addition to providing a strong, readable, and maintainable approach to regular expression pattern matching, parsing enable you to create your own custom languages for specific purposes.

References:

1. Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D. (1986). *Compilers, principles, techniques, and tools* (Rep. with

corrections. Ed.). Addison-Wesley Pub. Co. ISBN 978-0201100884.
2. Aho, Alfred V.; Ullman, Jeffrey D. (1972). *The Theory of Parsing, Translation, and Compiling (Volume 1: Parsing.)* (Repr. Ed.). Englewood Cliffs, NJ: Prentice-Hall. ISBN 978-0139145568.
3. 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.
4. 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.
5. Frost, R. and Hafiz, R. (2006) "A New Top-Down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time." *ACM SIGPLAN Notices*, Volume 41 Issue 5, Pages: 46 - 54.
6. Norvig, P. (1991) "Techniques for automatic memoisation with applications to context-free parsing." *Journal - Computational Linguistics* Volume 17, Issue 1, Pages: 91 - 98.
7. Tomita, M. (1985) "Efficient Parsing for Natural Language." *Kluwer, Boston, MA.*
8. "Bartleby.com homepage". Retrieved 28 November 2010.
9. "Parse". Dictionary.reference.com. Retrieved 27 November 2010.
10. "Grammar and Composition".
11. Aho, A.V., Sethi, R. and Ullman, J.D. (1986) " Compilers: principles, techniques, and tools." *Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.*
12. 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.
13. 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.
14. shproto.org
15. Compilers: Principles, Techniques, and Tools (2nd Edition), by Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman, Prentice Hall 2006.