International Journal of Advanced Research in Engineering and Applied Sciences
ISSN: 2278-6252

PARSING TECHNIQUES: A REVIEW

Pankaj Sharma*

Naveen Malik*

Naeem Akhtar*

Rahul*

Hardeep Rohilla*

Abstract: *'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.*

*Student, CSE, Dronachraya Collage of Engineering, Gurgaon

1. 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 (ōrātiōnis), 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 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." [2] 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. HISTORY

Parsing is the process of structuring a linear representation in accordance with a given grammar. This definition has been kept abstract on purpose, to allow as wide an interpretation as possible. The “linear representation” may be a sentence, a computer program, a 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. For each grammar, there are generally an infinite number of linear representations (“sentences”) that can be structured with it. That is, a finite-size grammar can supply structure to an infinite number of sentences. This is the main strength of the grammar paradigm and indeed the main source of the importance of grammars: they

summarize succinctly the structure of an infinite number of objects of a certain class. There are several reasons to perform this structuring process called parsing. One reason derives from the fact that the obtained structure helps us to process the object further. When we know that a certain segment of a sentence in German is the subject, that information helps in translating the sentence. Once the structure of a document has been brought to the surface, it can be converted more easily. A second is related to the fact that the grammar in a sense represents our understanding of the observed sentences: the better a grammar we can give for the movements of bees, the deeper our understanding of them is. A third lies in the completion of missing information that parsers, and especially error-repairing parsers, can provide. Given a reasonable grammar of the language, an error-repairing parser can suggest possible word classes for missing or unknown words on clay tablets.

3. TYPES OF PARSING TECHNIQUES

There are several other dimensions on which is useful to characterize the behavior of parsing algorithms. One can characterize their search strategy in terms of the characteristic alternatives of depth first or breadth first (q.v.). Orthogonally, one can characterize them in terms of the direction in which a structure is built: from the words upwards ('bottom up'), or from the root node downwards ('top down'). A third dimension is in terms of the sequential processing of input words: usually this is left-to-right, but right-to-left or 'middle-out' strategies are also feasible and may be preferable in some applications (e.g. parsing the output of a speech recognizer).

3.1. 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.

3.1.1 Accommodating left recursion in top-down parsing:-

A formal grammar that contains left recursion cannot be parsed by a naive recursive descent parser unless they are converted to a weakly equivalent right-recursive form. However, recent research demonstrates that it is possible to accommodate left-recursive grammars (along with all other forms of general CFGs) in a more sophisticated top-down parser by use of curtailment. A recognition algorithm which accommodates ambiguous grammars and curtails an ever-growing direct left-recursive parse by imposing depth restrictions with respect to input length and current input position, is described by Frost and Hafiz in 2006.[5] That algorithm was extended to a complete parsing algorithm to accommodate indirect (by comparing previously computed context with current context) as well as direct left-recursion in polynomial time, and to generate compact polynomial-size representations of the potentially exponential number of parse trees for highly ambiguous grammars by Frost, Hafiz and Callaghan in 2007.[3] The algorithm has since been implemented as a set of parser combinatory written in the Haskell programming language. The implementation details of these new set of combinatory can be found in a paper [4] by the above-mentioned authors, which was presented in PADL'08. The X-SAIGA site has more about the algorithms and implementation details

3.1.2. 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 need exponential number of steps (with respect to the length of the input) to try all alternatives of the CFG in order to produce all possible parse trees, which eventually would require exponential memory space. The problem of exponential time complexity in top-down parsers constructed as sets of mutually recursive functions has been solved by Norvig in 1991. His technique is similar to the use of dynamic programming and state-sets in Earley's algorithm (1970), and tables in the CYK algorithm of Cocke, Younger and Kasami.

The key idea is to store results of applying a parser p at position j in a memotable and to reuse results whenever the same situation arises. Frost, Hafiz and Callaghan[3][4] also use memoization for refraining redundant computations to accommodate any form of CFG in polynomial time ($\Theta(n^4)$ for left-recursive grammars and $\Theta(n^3)$ for non left-recursive grammars). Their top-down parsing algorithm also requires polynomial space for potentially exponential ambiguous parse trees by 'compact representation' and 'local ambiguities grouping'. Their compact representation is comparable with Tomita's compact representation of bottom-up parsing.

3.2. Bottom up parsing:-

The basic idea of a bottom-up parser is that we use grammar productions in the opposite way (from right to left). Like for predictive parsing with tables, here too we use a stack to push symbols. If the first few symbols at the top of the stack match the ruse of some rule, then we pop out these symbols from the stack and we push the lhs (left-hand-side) of the rule. This is called a *reduction*. For example, if the stack is x * E + E (where x is the bottom of stack) and there is a rule E ::= E + E, then we pop out E + E from the stack and we push E; ie, the stack becomes x * E. The sequence E + E in the stack is called a *handle*. But suppose that there is another rule S ::= E, then E is also a handle in the stack. Which one to choose? Also what happens if there is no handle? The latter question is easy to answer: we push one more terminal in the stack from the input stream and check again for a handle. This is called *shifting*. So another name for bottom-up parsers is shift-reduce parsers. There two actions only:

1. shift the current input token in the stack and read the next token, and
2. reduce by some production rule.

Consequently the problem is to recognize when to shift and when to reduce each time, and, if we reduce, by which rule. Thus we need a recognizer for handles so that by scanning the stack we can decide the proper action. The recognizer is actually a finite state machine exactly the same we used for REs. But here the language symbols include both terminals and non terminal (so state transitions can be for any symbol) and the final states indicate either reduction by some rule or a final acceptance (success).

A DFA though can only be used if we always have one choice for each symbol. But this is not the case here, as it was apparent from the previous example: there is an ambiguity in

recognizing handles in the stack. In the previous example, the handle can either be E + E or E. This ambiguity will hopefully be resolved later when we read more tokens. This implies that we have multiple choices and each choice denotes a valid potential for reduction. So instead of a DFA we must use a NFA, which in turn can be mapped into a DFA. These two steps (extracting the NFA and map it to DFA) are done in one step using *item sets.*

3.3. Chart Parsing:-

- Chart parsing uses charts based upon a "well-formed substring table," or "wfsst." A chart is represents the interaction between "edges" and "vertices," wherein vertices are the position of words in a sentence and an edge is the underlying rule. In programming, chart parsing can get very complex, involving long and intricate algorithms. Chart parsing is most useful when dealing with complex sentences or language structures that involve many rules working in tandem.

A **chart parser** is a type of parser suitable for ambiguous grammars (including grammars of natural languages). It uses the dynamic programming approach—partial hypothesized results are stored in a structure called a chart and can be re-used. This eliminates backtracking and prevents a combinatorial explosion.

- Chart parsing is generally credited to Martin Kay.[1]

3.3.1. Types of chart parsers

A common approach is to use a variant of the Viterbi algorithm. The Earley parser is a type of chart parser mainly used for parsing in computational linguistics, named for its inventor. Another chart parsing algorithm is the Cocke-Younger-Kasami (CYK) algorithm.

Chart parsers can also be used for parsing computer languages. Earley parsers in particular have been used in compiler compilers where their ability to parse using arbitrary Context-free grammars eases the task of writing the grammar for a particular language. However their lower efficiency has led to people avoiding them for most compiler work.

In bidirectional chart parsing, edges of the chart are marked with a direction, either forwards or backwards, and rules are enforced on the direction in which edges must point in order to be combined into further edges.

In incremental chart parsing, the chart is constructed incrementally as the text is edited by the user, with each change to the text resulting in the minimal possible corresponding change to the chart.

We can distinguish top-down and bottom-up chart parsers, and active and passive chart parsers.

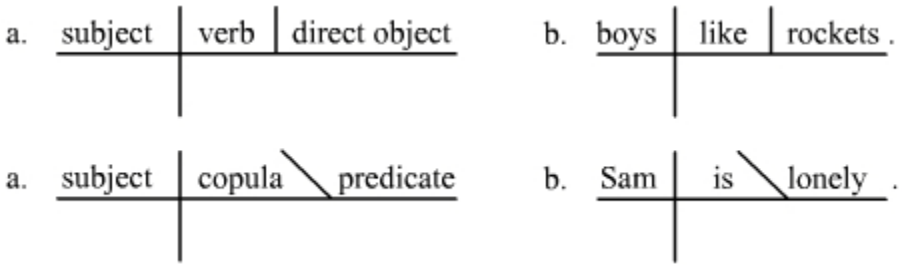
3.4. Sentence Diagramming

Students who are tasked with sentence diagramming in school may not realize they're actually studying a variant of parsing as well. X-bar theory, for example, was developed in the 1970s and is widely used by linguistics to parse a language's lexicon. Parts of speech are assigned one of three levels, X, X-bar and X-double bar, and each sentence has a "head" on which it is based from which subsequent levels follow. For example, a sentence may be "headed" by a verb, from which the X-shaped parsing emerges.

a sentence diagram or *parse tree* is a pictorial representation of the grammatical structure of a sentence. The term "sentence diagram" is used more in pedagogy, where sentences are *diagrammed*. The term "parse tree" is used in linguistics (especially computational linguistics), where sentences are *parsed*. The purpose of sentence diagrams and parse trees is to have a model of the structure of sentences. The model is informative about the relations between words and the nature of syntactic structure and is thus used as a tool to help predict which sentences are and are not possible.

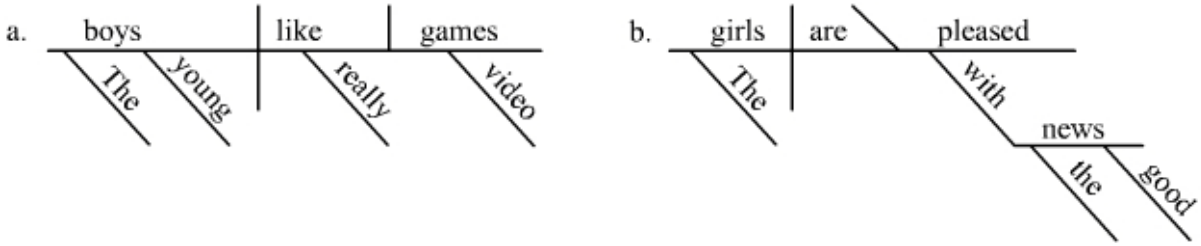
3.4.1. The Reed-Kellogg System

Simple sentences in the Reed-Kellogg system are diagrammed in accordance with the following basic schemata:



The diagram of a simple sentence begins with a horizontal line called the *base*. The subject is written on the left, the predicate on the right, separated by a vertical bar which extends through the base. The predicate must contain a verb, and the verb either requires other sentence elements to complete the predicate, permits them to do so, or precludes them from doing so. The verb and its object, when present, are separated by a line that ends at the baseline. If the object is a direct object, the line is vertical. If the object is a predicate noun or adjective, the line looks like a backslash, \, sloping toward the subject.

Modifiers of the subject, predicate, or object dangle below the base line:

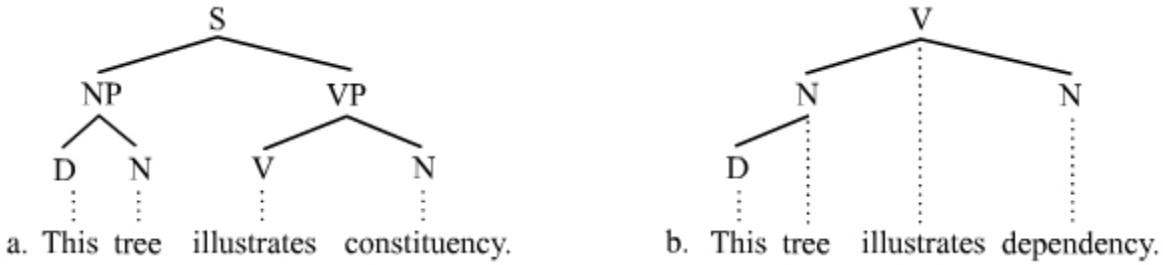


Adjectives (including articles) and adverbs are placed on slanted lines below the word they modify. Prepositional phrases are also placed beneath the word they modify; the preposition goes on a slanted line and the slanted line leads to a horizontal line on which the object of the preposition is placed.

These basic diagramming conventions are augmented for other types of sentence structures, e.g. for coordination and subordinate clauses.

3.4.2. Constituency and dependency

The connections to modern principles for constructing parse trees are present in the Reed-Kellogg diagrams, although Reed and Kellogg were certainly unknowingly employing these principles. The principles are now understood as the constituency relation of phrase structure grammars and the dependency relation of dependency grammars. These two relations are illustrated here adjacent to each other for comparison:



(D = Determiner, N = Noun, NP = Noun Phrase, S = Sentence, V = Verb, VP = Verb Phrase)

Constituency is a one-to-one-or-more relation; every word in the sentence corresponds to one or more nodes in the tree diagram. Dependency, in contrast, is a one-to-one relation; every word in the sentence corresponds to exactly one node in the tree diagram. Both parse trees employ the convention where the category acronyms (e.g. N, NP, V, VP) are used as the labels on the nodes in the tree. The one-to-one-or-more constituency relation is capable of increasing the amount of sentence structure to the upper limits of what is possible. The result can be very "tall" trees, such as those associated with X-bar theory. Both

constituency-based and dependency-based theories of grammar have established traditions.

4. DETERMINISM

A parsing procedure which, in a particular state, is faced with a choice as to what to do next, is called 'non-deterministic'. It has been argued (Marcus 1980) that natural languages are almost deterministic, given the ability to look ahead a few constituents: i.e., that in general if there is not sufficient information to make a decision based on the input so far, there usually will be within the next few words. Such a property would have obvious functional advantages for the efficient processing of language using minimal short term memory resources. This claim, although hotly contested, has been used to try to explain certain types of preferred interpretation or otherwise mysterious difficulties in interpreting linguistic constructs. For example, it is argued, if the human parsing system is

deterministic, and making decisions based on limited look ahead, we would have an explanation for the fact that sentences like: The horse raced past the barn fell are perceived as rather difficult to understand. Under normal circumstances, the tendency is to be 'led down the garden path', assembling 'the horse raced past the barn' into a sentence, then finding an apparently superfluous verb at the end. However, there are many other factors involved in the comprehension of sentences, and when all of these are taken into account, the determinism hypothesis is by no means completely satisfactory as an explanation. (For a survey and further discussion, see Pulman 1987; Briscoe, 1987)

5. DIFFICULTIES IN PARSING

The main difficulty in parsing is non determinism. That is, at some point in the derivation of a string more than one productions are applicable, though not all of them lead to the desired string, and one can not tell which one to use until after the entire string is generated. For example in the parsing of *aababaa* discussed above, when S is at the top of the stack and *a* is read in the top-down parsing, there are two applicable productions, namely $S \longrightarrow aSa$ and $S \longrightarrow a$. However, it is not possible to decide which one to choose with the information of the input symbol being read and the top of the stack. Similarly for the bottom-up parsing, it is impossible to tell when to apply the production $S \longrightarrow a$ with the same information as for the top-down parsing. Some of these non determinisms are due to the particular grammar being used and they can be removed by transforming grammars

to other equivalent grammars while others are the nature of the language the string belongs to. Below several of the difficulties are briefly discussed.

5.1.Factoring:

Consider the following grammar:

$$S \to T;\quad T \to aTb \mid abT \mid ab.$$

With this grammar when string *aababaa* is parsed top-down, after S is replaced by T in the first step, there is no easy way of telling which production to use to rewrite T next. However, if we change this to the following grammar which is equivalent to this grammar, this nondeterminism disappears:

$$S \to aU;\quad U \to Sb \mid bT;\quad T \to S \mid \Lambda.$$

This transformation operation is called **factoring** as *a* on the right hand side of productions for T in the original grammar are factored out as see n in the new grammar.

5.2. Left-recursion:

Consider the following grammar:

$$S \to Sa \mid Sb \mid a$$

When a string, say *aaba*, is parsed top-down for this grammar, after S is pushed into the stack, it needs to be replaced by the right hand side of some production. However, there is no simple way of telling which production to use and a parser may go into infinite loop especially if it is given an illegal string (a string which is not in the language). This kind of grammar is called **left-recursive**. Left-recursions can be removed by replacing left-recursive pairs of productions with new pairs of productions as follows:

If $X \to X\alpha_1 \mid X\alpha_2 \mid \beta_1 \mid \beta_2$ are left-recursive productions, where β's don't start with X, then replace them with $X \to \beta_1 X' \mid \beta_2 X'$ and $X' \to \alpha_1 X' \mid \alpha_2 X' \mid \Lambda$.

For example the left-recursive grammar given above can be transformed to the following non-recursive grammar:

$$S \to aS';\quad S' \to aS' \mid bS' \mid \Lambda$$

5.3.Ambiguous grammar:

A context-free grammar is called **ambiguous** if there is at least one string that has more than one distinct derivations (or, equivalently, parse trees). For example, the grammar $S \rightarrow S + S \mid S * S \mid (S) \mid id$, where id represents an identifier, produces the following two derivations for the expression $x + y * z$:

$$S \Rightarrow S + S \Rightarrow id + S \Rightarrow id + S * S \Rightarrow id + id * S \Rightarrow id + id * id$$

, which corresponds to $x + (y * z)$ and

$$S \Rightarrow S * S \Rightarrow S + S * S \Rightarrow id + S * S \Rightarrow id + id * S \Rightarrow id + id * id$$

, which corresponds to $(x + y) * z$.

Though some context-free languages are inherently ambiguous and no unambiguous grammars can be constructed for them, it is often possible to construct unambiguous context-free grammars for unambiguous context-free languages. For example, for the language of algebraic expressions given above, the following grammar is unambiguous: $S \Rightarrow S + X \mid X$; $X \Rightarrow X * Y \mid Y$; $Y \Rightarrow (S) \mid id$.

5.4. Nondeterministic language :

Lastly there are context-free languages that can not be parsed by a deterministic PDA. This kind of languages need nondeterministic PDAs. Hence guess work is necessary in selecting the right production at certain steps of their derivation. For example take the language of palindromes. When parsing strings for this language, the middle of a given string must be identified. But it can be shown that no deterministic PDA can do that.

6. CONCLUSION

In this dissertation we have studied techniques for the principled combination of multiple textual software languages into a single, composite language. The applications we have studied motivate the need for composing languages, e.g.

for syntactic abstraction, syntactic checking of meta-programs with concrete object syntax, and the prevention of injection attacks. We have extensively evaluated the application of

modular syntax definition, scanner less parsing, generalized parsing, and parse table composition for composing languages. Our case studies provide strong evidence that the aggregate of these techniques is the technique for the principled combination of languages into a single, composite language. First, using these techniques we have been able to formally define the syntax of a series of composite languages, in particular Aspect, a complex, existing language conglomerate for which no formal syntax definition was available. Second, we have explained in detail how employing scanner less and generalized parsing techniques elegantly deals with the issues in parsing such combinations of languages, in particular context sensitive lexical syntax. Third, we have shown for various applications that we can avoid the need to spend considerable effort on crafting specific combinations of languages. The resulting genericity of these applications is due to techniques that allow the syntax of a composite language to be defined as the principled combination of its sub-languages. Fourth, we have introduced and evaluated parse table composition as a technique that enables the separate compilation and deployment of language extensions. This allows the separate deployment of syntax embeddings as plug-ins to a compiler. Hence, end-programmers can select syntax embeddings for use in their source programs, but do not have to wait for the parse table to be compiled from scratch.

REFERENCES:-

[1]. http://en.wikipedia.org/wiki/Parsing

[2].http://books.google.co.in/books?id=05xA_d5dSwAC&printsec=frontcover&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false

[3].M. Tomita 1987:- An efficient augmented context-free parsing algorithm Computational Linguistics.

[4].M. Marcus 1980:- A Theory of Syntactic Recognition for Natural Language, MIT Press.

[5].S. G. Pulman 1987:-Computational Models of Parsing, in A. Ellis (ed), Progress in the Psychology of Language, Vol III.

[6].E. J. Briscoe 1987:-Modelling Human Speech Comprehension: a Computational Approach, Ellis Horwood, Chichester, and Wiley and Sons, N.Y.J. Kimball 1973

[7].L. Frazier and J. D. Fodor 1978:-The Sausage Machine: a new two-stage parsing model.

[8].Principles of Compiler Design Reading, Mass: Addison Wesley.

[9]H. S. Thompson and G. D. Ritchie 1984 Implementing Natural Language Parsers