ATTACHMENT

Top-down parsing

Reference from: http://en.wikipedia.org/wiki/Top-down_parsing

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. [1]

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 [2] . However, more sophisticated top-down parsers have been created by Frost, Hafiz, and Callaghan [3] [4] which do accommodate ambiguity and left recursion in polynomial time and which generate polynomial-sized representations of the potentially-exponential number of parse trees.

Programming language application

A compiler parses input from a programming language to assembly language or an internal representation by matching the incoming symbols to Backus-Naur form production rules. An LL parser, also called a **top-bottom parser** or **top-down parser**, applies each production rule to the incoming symbols by working from the left-most symbol yielded on a production rule and then proceeding to the next production rule for each non-terminal symbol encountered. In this way the parsing starts on the Left of the result side (right side) of the production rule and evaluates non-terminals from the Left first and, thus, proceeds down the parse tree for each new non-terminal before continuing to the next symbol for a production rule.

For example:

- $A \rightarrow aBC$

- $B \rightarrow c|cd$
- $C \rightarrow df|eg$

would match $A \rightarrow aBC$ and attempt to match $B \rightarrow c|cd$ next. Then $C \rightarrow df|eg$ would be tried. As one may expect, some languages are more ambiguous than others. For a non-ambiguous language in which all productions for a non-terminal produce distinct strings: the string produced by one production will not start with the same symbol as the string produced by another production. A non-ambiguous language may be parsed by an LL(1) grammar where the (1) signifies the parser reads ahead one token at a time. For an ambiguous language to be parsed by an LL parser, the parser must lookahead more than 1 symbol, e.g. LL(3).

The common solution is to use an LR parser (also known as bottom-up or shift-reduce parser).

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 combinators written in the Haskell programming language. The implementation details of these new set of combinators 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.

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.[6] 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.[7]

Using PEG's, another representation of grammars, packrat parsers provide a elegant and powerful parsing algorithm. See Parsing expression grammar.