Left recursion

In the formal language theory of computer science, **left recursion** is a special case of recursion where a string is recognized as part of a language by the fact that it decomposes into a string from that same language (on the left) and a suffix (on the right). For instance, $1+2+3$ can be recognized as a sum because it can be broken into $1+2$, also a sum, and $+3$, a suitable suffix.

In terms of context-free grammar, a nonterminal is left-recursive if the leftmost symbol in one of its productions is itself (in the case of direct left recursion) or can be made itself by some sequence of substitutions (in the case of indirect left recursion).

Contents

Definition

A grammar is left-recursive if and only if there exists a nonterminal symbol A that can derive to a sentential form with itself as the leftmost symbol.[1] Symbolically,

$$A \Rightarrow^{+} A\alpha,$$

where $\Rightarrow^{+}$ indicates the operation of making one or more substitutions, and α is any sequence of terminal and nonterminal symbols.

Direct left recursion

Direct left recursion occurs when the definition can be satisfied with only one substitution. It requires a rule of the form

$$A \to A\alpha$$

where α is a sequence of nonterminals and terminals. For example, the rule

$$\mathit{Expression} \to \mathit{Expression} + \mathit{Term}$$

is directly left-recursive. A left-to-right recursive descent parser for this rule might look like

```
void Expression() {
  Expression();
  match('+');
  Term();
}
```

and such code would fall into infinite recursion when executed.

Indirect left recursion

Indirect left recursion occurs when the definition of left recursion is satisfied via several substitutions. It entails a set of rules following the pattern

$$\begin{array}{l} A_0 \to \beta_0 A_1 \alpha_0 \\ A_1 \to \beta_1 A_2 \alpha_1 \\ \cdots \\ A_n \to \beta_n A_0 \alpha_n \end{array}$$

where $\beta_0, \beta_1, \ldots, \beta_n$ and $\alpha_0, \alpha_1, \ldots, \alpha_n$ are any sequence of terminal and nonterminal symbols. Note that these sequences may be empty. The derivation

$$A_0 \Rightarrow \beta_0 A_1 \alpha_0 \Rightarrow^+ A_1 \alpha_0 \Rightarrow \beta_1 A_2 \alpha_1 \alpha_0 \Rightarrow^+ \cdots \Rightarrow^+ A_0 \alpha_n \ldots \alpha_1 \alpha_0$$

then gives A_0 as leftmost in its final sentential form.

Removing left recursion

Left recursion often poses problems for parsers, either because it leads them into infinite recursion (as in the case of most top-down parsers) or because they expect rules in a normal form that forbids it (as in the case of many bottom-up parsers, including the CYK algorithm). Therefore, a grammar is often preprocessed to eliminate the left recursion.

Removing direct left recursion

The general algorithm to remove direct left recursion follows. Several improvements to this method have been made.[2] For a left-recursive nonterminal A, discard any rules of the form $A \to A$ and consider those that remain:

$$A \to A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

where:

- each α is a nonempty sequence of nonterminals and terminals, and
- each β is a sequence of nonterminals and terminals that does not start with A.

Replace these with two sets of productions, one set for A:

$$A \to \beta_1 A' \mid \ldots \mid \beta_m A'$$

and another set for the fresh nonterminal A' (often called the "tail" or the "rest"):

$$A' \to \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \epsilon$$

Repeat this process until no direct left recursion remains.

As an example, consider the rule set

$$\mathit{Expression} \to \mathit{Expression} + \mathit{Expression} \mid \mathit{Integer} \mid \mathit{String}$$

This could be rewritten to avoid left recursion as

$$\begin{array}{l} \mathit{Expression} \to \mathit{Integer}\,\mathit{Expression}' \mid \mathit{String}\,\mathit{Expression}' \\ \mathit{Expression}' \to +\,\mathit{Expression}\,\mathit{Expression}' \mid \epsilon \end{array}$$

Removing all left recursion

By establishing a topological ordering on nonterminals, the above process can be extended to also eliminate indirect left recursion

Inputs *A grammar: a set of nonterminals* $A_1, \ldots, A_n$ *and their productions*
Output *A modified grammar generating the same language but without left recursion*

1. *For each nonterminal* A_i*:*
    1. *Repeat until an iteration leaves the grammar unchanged:*
        1. *For each rule* $A_i \rightarrow \alpha_i$*,* α_i *being a sequence of terminals and nonterminals:*
            1. *If* α_i *begins with a nonterminal* A_j *and* $j < i$*:*
                1. *Let* β_i *be* α_i *without its leading* A_j*.*
                2. *Remove the rule* $A_i \rightarrow \alpha_i$*.*
                3. *For each rule* $A_j \rightarrow \alpha_j$*:*
                    1. *Add the rule* $A_i \rightarrow \alpha_j \beta_i$*.*
    2. *Remove direct left recursion for* A_i *as described above.*

Note that this algorithm is highly sensitive to the nonterminal ordering; optimizations often focus on choosing this ordering well.

Pitfalls

Although the above transformations preserve the language generated by a grammar, they may change the parse trees that witness strings' recognition. With suitable bookkeeping, tree rewriting can recover the originals, but if this step is omitted, the differences may change the semantics of a parse.

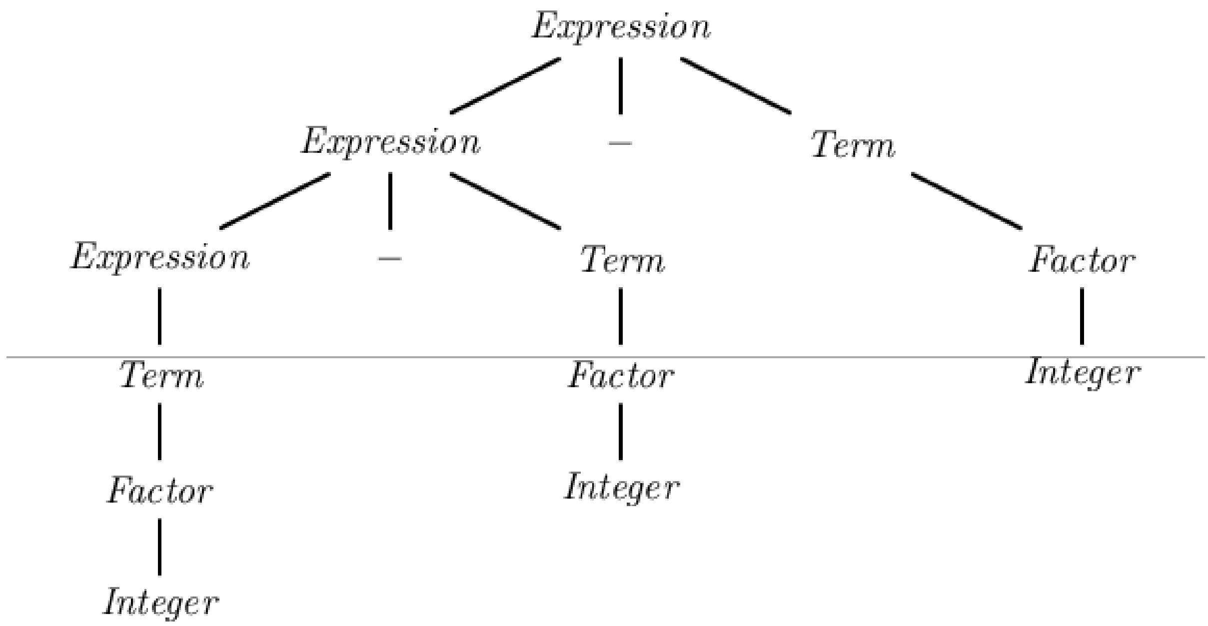
Associativity is particularly vulnerable; left-associative operators typically appear in right-associative-like arrangements under the new grammar. For example, starting with this grammar:

$$\begin{aligned} &Expression \rightarrow Expression\ -\ Term \mid Term \\ &Term \rightarrow Term\ *\ Factor \mid Factor \\ &Factor \rightarrow (Expression) \mid Integer \end{aligned}$$

the standard transformations to remove left recursion yield the following:

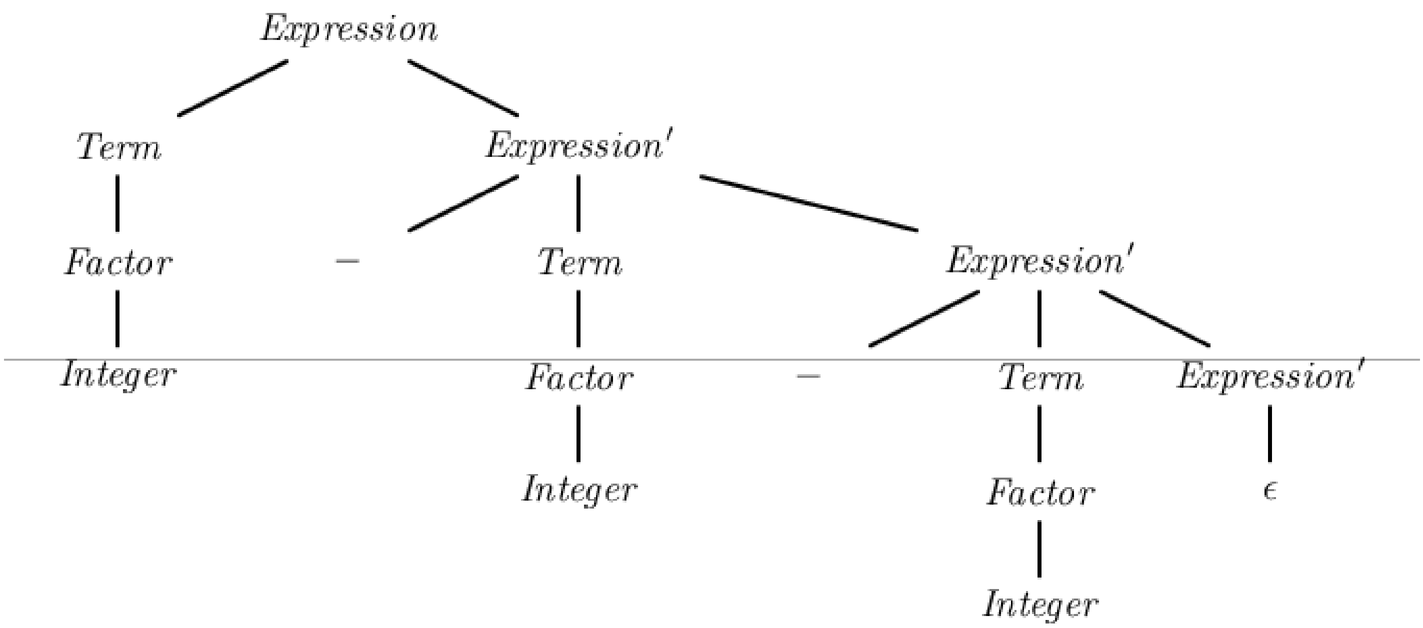
$$\begin{aligned} &Expression \rightarrow Term\ Expression' \\ &Expression' \rightarrow\ -\ Term\ Expression' \mid \epsilon \\ &Term \rightarrow Factor\ Term' \\ &Term' \rightarrow\ *\ Factor\ Term' \mid \epsilon \\ &Factor \rightarrow (Expression) \mid Integer \end{aligned}$$

Parsing the string "1 - 2 - 3" with the first grammar in an LALR parser (which can handle left-recursive grammars) would have resulted in the parse tree:



This parse tree groups the terms on the left, giving the correct semantics *(1 - 2) - 3*.

Parsing with the second grammar gives



which, properly interpreted, signifies *1 + (-2 + (-3))*, also correct, but less faithful to the input and much harder to implement for some operators. Notice how terms to the right appear deeper in the tree, much as a right-recursive grammar would arrange them for *1 - (2 - 3)*.

Accommodating left recursion in top-down parsing

A formal grammar that contains left recursion cannot be parsed by a LL(k)-parser or other naive recursive descent parser unless it is converted to a weakly equivalent right-recursive form. In contrast, left recursion is preferred for LALR parsers because it results in lower stack usage than right recursion. However, more sophisticated top-down parsers can implement general context-free grammars by use of curtailment. In 2006, Frost and Hafiz described an algorithm which accommodates ambiguous grammars with direct left-recursive production rules.[3] That algorithm was extended to a complete parsing algorithm to accommodate indirect 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.[4] The authors then implemented the algorithm as a set of parser combinators written in the Haskell programming language.[5]

See also

- Tail recursion

References

1. Notes on Formal Language Theory and Parsing (http://www.cs.may.ie/~jpower/Courses/parsing/parsing.pdf#search='indirect%20left%20recursion') James Power, Department of Computer Science National University of Ireland, Maynooth Maynooth, Co. Kildare, Ireland.JPR02
2. Moore, Robert C. (May 2000). "Removing Left Recursion from Context-Free Grammars" (http://research.microsoft.com/pubs/68869/naacl2k-proc-rev.pdf) (PDF). *6th Applied Natural Language Processing Conference*: 249–255.
3. Frost, R.; R. Hafiz (2006). "A New Top-Down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time" (http://portal.acm.org/citation.cfm?id=1149988). *ACM SIGPLAN Notices* **41** (5): 46–54. doi:10.1145/1149982.1149988 (https://doi.org/10.1145/1149982.1149988), available from the author at http://hafiz.myweb.cs.uwindsor.ca/pub/p46-frost.pdf
4. Frost, R.; R. Hafiz; P. Callaghan (June 2007). "Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars" (https://web.archive.org/web/20110527032954/http://acl.ldc.upenn.edu/W/W07/W07-2215.pdf) (PDF). *10th International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSE*. Prague: 109–120. Archived from the original (http://acl.ldc.upenn.edu/W/W07/W07-2215.pdf) (PDF) on 2011-05-27.
5. Frost, R.; R. Hafiz; P. Callaghan (January 2008). "Parser Combinators for Ambiguous Left-Recursive Grammars" (http://cs.uwindsor.ca/~richard/PUBLICATIONS/PADL_08.pdf) (PDF). *10th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN*. Lecture Notes in Computer Science. **4902** (2008): 167–181. doi:10.1007/978-3-540-77442-6_12 (https://doi.org/10.1007/978-3-540-77442-6_12). ISBN 978-3-540-77441-9.

External links

- Practical Considerations for LALR(1) Grammars

Retrieved from "https://en.wikipedia.org/w/index.php?title=Left_recursion&oldid=849701820"