



Compiler Construction

Lecture Notes

Syntax Analysis

Agenda

Introduction

Error Handling

Context Free Grammar

Writing a Grammar

Introduction

Every programming language has precise rules that prescribe the syntactic structure of well-formed programs. In C, for example, a program is made up of functions, a function out of declarations and statements, a statement out of expressions, and so on. The syntax of programming language constructs can be specified by context-free grammars.

Grammars

A grammar gives a precise syntactic specification of a programming language. An efficient parser can be constructed from the grammar that determines the syntactic structure of a source program.

As a side benefit, the parser-construction process can reveal syntactic ambiguities and trouble spots that might have slipped through the initial design phase of a language.

The structure imparted to a language by a properly designed grammar is useful for translating source programs into correct object code and detection of errors.

A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks. These new constructs can be integrated more easily into an implementation that follows the grammatical structure of the language.

Role of the Parser

The parser obtains a string of tokens from the lexical analyzer, and verifies that the string of token names can be generated by the grammar for the source language. Parser can report any syntax errors in input code and recover from commonly occurring errors to continue processing the remainder of the program.

There are three general types of parsers for grammars: *universal*, *top-down*, and *bottom-up*. Universal parsing methods are too inefficient to use in production compilers.

The methods commonly used in compilers can be classified as being either top-down or bottom-up. *Top-down* methods build parse trees from the top (root) to the bottom (leaves), while *bottom-up* methods start from the leaves and work their way up to the root. In either case, the input to the parser is scanned from left to right, one symbol at a time.

Assume that the output of the parser is some representation of the parse tree for the stream of tokens that comes from the lexical analyzer. In practice, there are a number of tasks that might be conducted during parsing, such as collecting information about various tokens into the symbol table, performing type checking and other kinds of semantic analysis, and generating intermediate code.

Representative Grammars

Constructs that begin with keywords like `while` or `int`, are relatively easy to parse, because the keyword guides the choice of the grammar production that must be applied to match the input. Expressions present more challenge, because of the associativity and precedence of operators.

Associativity and precedence are captured in the following grammar for describing expressions, terms, and factors. E represents expressions consisting of terms separated by `+` signs, T represents terms consisting of factors separated by `*` signs, and F represents factors that can be either parenthesized expressions or identifiers:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \text{id} \end{array}$$

Figure 1: Simple expression grammar

Expression grammar above belongs to the class of LR grammars that are suitable for bottom-up parsing. This grammar can be adapted to handle additional operators and additional levels of precedence. However, it cannot be used for top-down parsing because it is left recursive.

The following non-left-recursive variant of the expression grammar above will be used for top-down parsing:

$$\begin{array}{lcl} E & \rightarrow & T E' \\ E' & \rightarrow & + T E' \mid \epsilon \\ T & \rightarrow & F T' \\ T' & \rightarrow & * F T' \mid \epsilon \\ F & \rightarrow & (E) \mid \text{id} \end{array}$$

Figure 2: Non-left-recursive variant of the expression grammar

The following grammar treats + and * alike, so it is useful for illustrating techniques for handling ambiguities during parsing:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

Figure 3: Simple grammar for the expression

Here, E represents expressions of all types. Grammar above permits more than one parse tree for expressions like $a + b * c$.

Syntax Error Handling

A compiler is expected to assist the programmer in locating and tracking down errors that inevitably creep into programs.

Most programming language specifications do not describe how a compiler should respond to errors; error handling is left to the compiler designer. Planning the error handling from the start can simplify the structure of a compiler and improve its handling of errors.

Common programming errors can occur at many different levels.

Lexical errors include misspellings of identifiers, keywords, or operators - e.g., missing quotes around text intended as a string.

Syntactic errors include misplaced semicolons or extra or missing braces; that is, `\{"` or `\}.`

Semantic errors include type mismatches between operators and operands, e.g., the return of a value in a method with result type void.

Logical errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator `=` instead of the comparison operator `==`. The program containing `=` may be well formed; however, it may not reflect the programmer's intent.

Several parsing methods, such as the LL and LR methods, detect an error as soon as possible; that is, when the stream of tokens from the lexical analyzer cannot be parsed further according to the grammar for the language. More precisely, they have the viable-prefix property, meaning that they detect that an error has occurred as soon as they see a prefix of the input that cannot be completed to form a string in the language.

Error recovery during parsing is also important because many errors appear syntactic, and are exposed when parsing cannot continue. A few semantic errors, such as type mismatches, can also be detected efficiently; however, accurate detection of semantic and logical errors at compile time is in general a difficult task.

The error handler in a parser has goals that are simple to state but challenging to implement:

Report the presence of errors clearly and accurately.

Recover from each error quickly enough to detect subsequent errors.

Add minimal overhead to the processing of correct programs.

At the very least, error handler must report the place in the source program where an error is detected, because there is a good chance that the actual error occurred within the previous few tokens. A common strategy is to print corresponding line with a pointer to the position at which an error is detected.

Error-Recovery Strategies

The simplest approach is for the parser to quit with an informative error message when it detects the first error.

Additional errors are often uncovered if the parser can restore itself to a state where processing of the input can continue with next token. If errors pile up, it is better for the compiler to give up after exceeding some error limit than to produce a big list of errors.

Panic-Mode Recovery

With this method, on discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found.

The synchronizing tokens are usually delimiters, such as semicolon or `}`. Compiler designer must select the synchronizing tokens appropriate for the source language. While panic-mode correction often skips a considerable amount of input without checking it for additional errors, it has the advantage of simplicity, and is guaranteed not to go into an infinite loop.

Phrase-Level Recovery

On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue. A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon.

Global Correction

There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction. Given an incorrect input string x and grammar G , these algorithms will find a parse tree for a related string y , such that the number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible.

These methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest. A closest correct program may not be what the programmer had in mind.

Context-Free Grammars

Grammars describe the syntax of programming language constructs like expressions and statements. Using a syntactic variable *stmt* to denote statements and variable *expr* to denote expressions, the production

$$stmt \rightarrow \text{if } (expr) \text{ stmt else stmt}$$

Figure 4: Grammar for conditional statement production

specifies the structure of this form of conditional statement. Other productions then define precisely what an *expr* is and what else a *stmt* can be.

Formal Definition of Context-Free Grammar

A context-free grammar (grammar for short) consists of terminals, nonterminals, a start symbol, and productions.

1. **Terminals** are the basic symbols from which strings are formed. The term “token name” is a synonym for “terminal”. We will use word “token” for terminal when it is clear that we are talking about just the token name.

2. **Nonterminals** are syntactic variables that denote sets of strings. `stmt` and `expr` are nonterminals. The sets of strings denoted by nonterminals help define the language generated by the grammar. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.

3. One nonterminal is marked as **start symbol**, and the set of strings it denotes is the language generated by the grammar. Productions of start symbol are listed first.

4. **Productions** of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each production consists of:

(a) A nonterminal called the head or left side of the production; this production defines some of the strings denoted by the head.

(b) The symbol `!`. Sometimes `::=` has been used in place of the arrow.

(c) A body or right side consisting of zero or more terminals and non-terminals. The components of the body describe one way in which strings of the nonterminal at the head can be constructed.

Example: The grammar below defines simple arithmetic expressions. In this grammar, the terminal symbols are *id* + - * / ()

The nonterminal symbols are expression, term and factor, and expression is the start symbol

$$\begin{array}{ll}
 \textit{expression} & \rightarrow \textit{expression} + \textit{term} \\
 \textit{expression} & \rightarrow \textit{expression} - \textit{term} \\
 \textit{expression} & \rightarrow \textit{term} \\
 \textit{term} & \rightarrow \textit{term} * \textit{factor} \\
 \textit{term} & \rightarrow \textit{term} / \textit{factor} \\
 \textit{term} & \rightarrow \textit{factor} \\
 \textit{factor} & \rightarrow (\textit{expression}) \\
 \textit{factor} & \rightarrow \mathbf{id}
 \end{array}$$

Figure 5: Grammar for simple arithmetic expressions

Notational Conventions

To avoid having to state that “these are the terminals,” “these are nonterminals,” and so on, the following notational conventions for grammars will be used:

1. The terminals:

- (a) Lowercase letters early in the alphabet, such as *a*, *b*, *c*.
- (b) Operator symbols such as +, * and so on.
- (c) Punctuation symbols such as parentheses, comma, and so on.
- (d) The digits 0, 1, ..., 9.
- (e) Boldface strings such as *id*, *if*, each of which representing single terminal symbol.

2. The nonterminals:

- (a) Uppercase letters early in the alphabet, such as *A*, *B*, *C*.
- (b) The letter *S* is usually the start symbol.
- (c) Lowercase, italic names such as *expr* or *stmt*.
- (d) In programming constructs, uppercase letters may be used to represent nonterminals for the constructs. For example, nonterminals for expressions, terms, and factors are often represented by *E*, *T*, and *F*, respectively.

3. Uppercase letters late in the alphabet, such as *X*, *Y*, *Z*, represent grammar symbols; either nonterminals or terminals.

4. Lowercase letters like *u*, *v*, ..., *z*, represent strings of terminals.

5. Lowercase Greek letters, α , β for example, represent strings of grammar symbols.

6. A set of productions $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ... $A \rightarrow \alpha_k$ with a common head *A* (or *A*-productions), may be written $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$

7. Unless stated otherwise, the head of the first production is the start symbol.

Example : Using these conventions, the grammar of the expressions can be rewritten concisely as:

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T / F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

The notational conventions show that E, T, and F are nonterminals, with E the start symbol. The remaining symbols are terminals.

Derivations

Beginning with the start symbol, each step replaces a nonterminal by the body of one of its productions. This derivational view corresponds to the top-down construction of a parse tree. Bottom-up parsing is related to a class of derivations known as “rightmost” derivations, in which the rightmost nonterminal is rewritten at each step.

For example, consider the following grammar, with a single nonterminal E , which adds a production $E \rightarrow - E$ to the grammar :

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$$

The production $E \rightarrow - E$ signifies that if E denotes an expression, then $- E$ must also denote an expression. The replacement of a single E by $- E$ will be described by:

$$E \Rightarrow -E$$

read as “ E derives $-E$.” The production $E \rightarrow (E)$ can be applied to replace any instance of E in any string of grammar symbols by (E) , e.g., $E * E \Rightarrow (E) * E$ or $E * E \Rightarrow E * (E)$. We can take a single E and repeatedly apply productions in any order to get a sequence of replacements. For example,

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

Such a sequence of replacements is called a derivation of $-(\mathbf{id})$ from E . This derivation is a proof that the string $-(\mathbf{id})$ is one particular instance of an expression.

A sentence of G is a sentential form with no nonterminals. The language generated by a grammar is its set of sentences. Thus, a string of terminals w is in $L(G)$, the language generated by G , if and only if w is a sentence of G (or $S \rightarrow^* w$). A language that can be generated by a grammar is said to be a *context-free language*. If two grammars generate the same language, the grammars are said to be *equivalent*.

(See page 200 of the book)

Parse Trees and Derivations

A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals. Each interior node of a parse tree represents the application of a production. The interior node is labeled with the nonterminal A in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation.

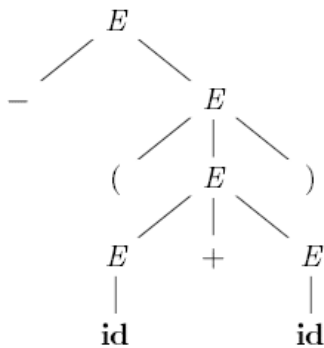


Figure 6: Parse tree for $-(id + id)$

In the first step of the derivation, $E \rightarrow -E$. To model this step, add two children, labeled $-$ and E , to the root E of the initial tree. The result is the second tree.

In the second step of the derivation, $-E \rightarrow -(E)$. This requires three children, labeled $($, E , and $)$, to the leaf labeled E of the second tree, to obtain the third tree with yield $-(E)$. Continuing this process results in the complete parse tree for the expression.

For leftmost or a rightmost derivation, there is a one-to-one relationship between parse trees and either leftmost or rightmost derivations. Both leftmost and rightmost derivations pick a particular order for replacing symbols in. Every parse tree has associated with it a unique leftmost and a unique rightmost derivation.

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathbf{id} + E) \Rightarrow -(\mathbf{id} + \mathbf{id})$$

Derivation for the expression $-(\mathbf{id} + \mathbf{id})$

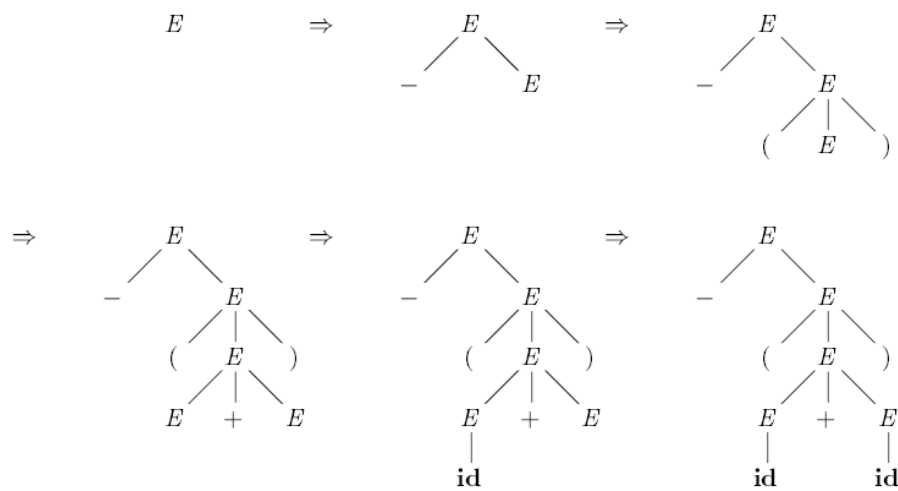


Figure 7: Sequence of parse trees for the derivation

Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguous. Put another way, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

Example : The arithmetic expression grammar

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \text{id}$$

permits two distinct leftmost derivations for the sentence $\text{id} + \text{id} * \text{id}$:

$E \Rightarrow E + E$	$E \Rightarrow E * E$
$\Rightarrow \text{id} + E$	$\Rightarrow E + E * E$
$\Rightarrow \text{id} + E * E$	$\Rightarrow \text{id} + E * E$
$\Rightarrow \text{id} + \text{id} * E$	$\Rightarrow \text{id} + \text{id} * E$
$\Rightarrow \text{id} + \text{id} * \text{id}$	$\Rightarrow \text{id} + \text{id} * \text{id}$

Corresponding parse trees are shown in figure below:

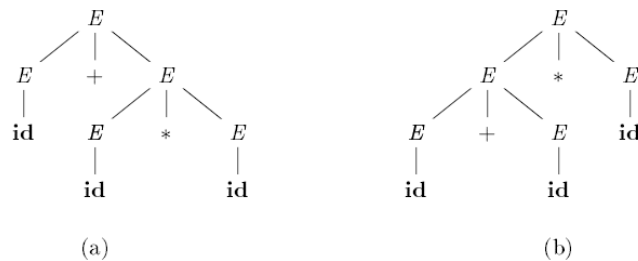


Figure 8: Two parse trees for $\text{id} + \text{id} * \text{id}$ with respect to the grammar above

The parse tree of Figure 8(a) reflects the commonly assumed precedence of $+$ and $*$, while the tree of Figure 8(b) does not. The operator $*$ has higher precedence than $+$, as an expression like $a + b * c$ is evaluated as $a + (b * c)$, rather than as $(a + b) * c$.

Most parsers require the grammar to be unambiguous, for if it is not, we cannot uniquely determine which parse tree to select for a sentence. In other cases, it is convenient to use carefully chosen ambiguous grammars, together with disambiguating rules that “throw away” undesirable parse trees, leaving only one tree for each sentence.

Verifying the Language Generated by a Grammar

Verifying confirms that a given set of productions generates a particular language. Troublesome constructs can be studied by writing a concise, abstract grammar and studying the language that it generates.

A proof that a grammar G generates a language L has two parts: show that every string generated by G is in L , and conversely that every string in L can indeed be generated by G .

Context-Free Grammars Versus Regular Expressions

Grammars are a more powerful notation than regular expressions. Every construct that can be described by a regular expression can be described by a grammar, but not vice-versa. That is, every regular language is a context-free language, but not vice-versa.

The regular expression $(a|b)^*abb$ and the grammar

$$\begin{aligned} A_0 &\rightarrow aA_0 \mid bA_0 \mid aA_1 \\ A_1 &\rightarrow bA_2 \\ A_2 &\rightarrow bA_3 \\ A_3 &\rightarrow \epsilon \end{aligned}$$

recognize the same language, the set of strings of a's and b's ending in abb.

Finite automata cannot count," meaning that a finite automaton cannot accept a language like $\{a^n b^n \mid n \geq 1\}$ that would require it to keep count of the number of a's before it sees the b's.

Writing a Grammar

Grammars are capable of describing most, but not all, of the syntax of programming languages. For instance, the requirement that identifiers be declared before they are used, cannot be described by a context-free grammar. Therefore, the sequences of tokens accepted by a parser form a superset of the programming language; subsequent phases of the compiler must analyze the output of the parser to ensure compliance with the language rules not checked by the parser.

Lexical Versus Syntactic Analysis

Everything that can be described by a regular expression can also be described by a grammar. Then why use regular expressions to define the lexical syntax of a language?

There are several reasons.

1. Separating the syntactic structure of a language into lexical and non-lexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.
2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.
3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.
4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

There are no guidelines as to what to put into the lexical rules, as opposed to the syntactic rules. Regular expressions are most useful for describing the structure of constructs such as identifiers, constants, keywords, and white space. Grammars, on the other hand, are most useful for describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's, and so on. These nested structures cannot be described by regular expressions.

Eliminating Ambiguity

Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. As an example, eliminate the ambiguity from the following “dangling-else” grammar:

```

stmt → if expr then stmt
      | if expr then stmt else stmt
      | other
    
```

Here “other” stands for any other statement. According to this grammar, the compound conditional statement

if E_1 then S_1 else if E_2 then S_2 else S_3

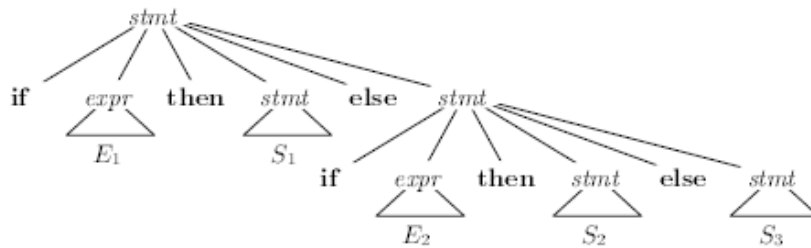


Figure 9: Parse tree for a conditional statement

has the parse tree shown in Fig 9, the Grammar is ambiguous since the string *if E_1 then if E_2 then S_1 else S_2* has the two parse trees shown in Fig 10.

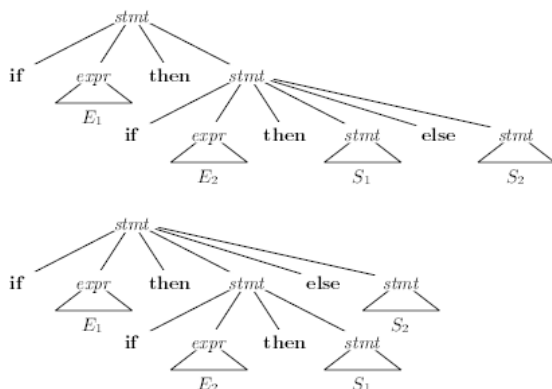


Figure 10: Two parse trees for an ambiguous sentence

In all programming languages with conditional statements of this form, the first parse tree is preferred. The general rule is, “Match each else with the closest unmatched then”.

The dangling-else grammar can be rewritten as the following unambiguous grammar. A statement appearing between a then and an else must be “matched”; that is, the interior statement must not end with an unmatched or open then. A matched statement is either an if-then-else statement containing no open statements or it is any other kind of unconditional statement. This grammar generates the same strings as the dangling-else grammar but it allows only one parsing for string, the one that associates each else with the closest previous unmatched then.

```

    stmt  →  matched_stmt
           |  open_stmt
matched_stmt →  if expr then matched_stmt else matched_stmt
           |  other
open_stmt  →  if expr then stmt
           |  if expr then matched_stmt else open_stmt

```

Figure 11: Unambiguous grammar for if-then-else statements

Elimination of Left Recursion

A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \rightarrow^+ A\alpha$ for some string α . Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.

Example : The non-left-recursive expression grammar

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \epsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid \epsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

is obtained by eliminating immediate left recursion from the expression grammar. The left-recursive pair of productions $E \rightarrow E + T \mid T$ are replaced by $E \rightarrow T E'$ and $E' \rightarrow + T E' \mid \epsilon$. The new productions for T and T' are obtained similarly by eliminating immediate left recursion.

Immediate left recursion can be eliminated by the following technique, which works for any number of A -productions. First, group the productions as

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

where no β_i begins with an A . Then, replace the A -productions by

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

The nonterminal A generates the same strings as before but is no longer left recursive. This procedure eliminates all left recursion from the A and A_0 productions (provided no α_i is ϵ), but it does not eliminate left recursion involving derivations of two or more steps. For example, consider the grammar

$$\begin{array}{l} S \rightarrow A a \mid b \\ A \rightarrow A c \mid S d \mid \epsilon \end{array}$$

The nonterminal S is left recursive because $S \rightarrow Aa \rightarrow Sda$, but it is not immediately left recursive.

Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

For example, if we have the two productions

$$\begin{array}{lcl} stmt & \rightarrow & \text{if } expr \text{ then } stmt \text{ else } stmt \\ & | & \text{if } expr \text{ then } stmt \end{array}$$

on seeing the input `if`, we cannot immediately tell which production to choose to expand `stmt`. In general, if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two A-productions, and the input begins with a nonempty string derived from α , we do not know whether to expand A to $\alpha\beta_1 \mid \alpha\beta_2$. However, this decision can be deferred by expanding A to $\alpha A'$. After seeing the input derived from α , expand A' to β_1 or to β_2 . That is, left-factored, the original productions become

$$\begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2 \end{array}$$

Non-Context-Free Language Constructs

A few syntactic constructs found in typical programming languages cannot be specified using grammars alone.

Example: The problem of checking that identifiers are declared before they are used in a program. The language consists of strings of the form wcw , where the first w represents the declaration of an identifier w , c represents any program fragment, and the second w represents the use of the identifier.

Example : The non-context-free language in this example abstracts the problem of checking that the number of formal parameters in the declaration of a function agrees with the number of actual parameters in a use of the function. This language is not context free.

Typical syntax of function declarations and uses does not concern itself with counting the number of parameters. For example, a function call in C-like language might be specified by:

$$\begin{array}{ll} \text{stmt} & \rightarrow \text{id (expr_list)} \\ \text{expr_list} & \rightarrow \text{expr_list , expr} \\ & | \text{expr} \end{array}$$

with suitable productions for expr . Checking that the number of parameters in a call is correct is done during the semantic-analysis phase.