BNF Grammars, Parse Trees, Ambiguity

A **BNF (Backus-Naur Form) grammar** is a triple < T, N, R > where

- T is a finite set of *terminal symbols*;
- N is a finite set of *non-terminal symbols*;
- R is a finite set of *production rules*.

When the BNF grammar is used for a formal description of lexical items ("tokens") and syntax of programming languages, terminal symbols are ASCII or Unicode characters and non-terminal symbols designate *syntactic categories*. In this course, the notation <X> will be used for syntactic categories where X is a mnemonic identifier or phrase. Terminal symbols will be simply referred to as *terminals*. Each production rule takes the form:

$$\langle X \rangle \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n, \quad n \geq 1$$

where each α_i is a string of terminals and/or syntactic categories; α_i may be the empty string ε.

Example Let T = { 0, 1, ..., 9, a, b, ..., z, A, B, ..., Z, +, −, *, /, (,), "." }, N = { <digit>, <letter>, <int>, <id>, <rest>, <float>, <exponent>, <eSign>, <sign>, <E> }, and let R contain the following production rules:

<digit> → 0 | 1 | ⋯ | 9
<letter> → a | b | ⋯ | z | A | B | ⋯ | Z
<int> → <digit> | <digit> <int>
<id> → <letter> <rest>
<rest> → ε | <digit> <rest> | <letter> <rest>
<float> → <int> "." <int> | <int> "." <int> <exponent>
<exponent> → <eSign> <int> | <eSign> <sign> <int>
<eSign> → "e" | "E"
<sign> → + | −
<E> → <id> | <int> | <float> | <E> + <E> | <E> − <E> | <E> * <E> | <E> / <E> | "(" <E> ")"

This BNF defines three token categories <int>, <id>, <float>; the categories <digit>, <letter>, <rest>, <exponent>, <eSign>, and <sign> are not tokens but merely auxiliary categories to assist the definitions of <int>, <id>, <float>. The category <E> of arithmetic expressions is then defined from the three token categories, the arithmetic operators +, −, *, /, and the parentheses.

Normally tokens are contiguous strings of non-whitespace characters. The usual assumption, therefore, is that the right side of any production rule for tokens may not be broken by whitespaces between any two terminals and/or syntactic categories. For example, we tacitly assume that no whitespace may appear between <digit> and <int> for <int>, between <letter> and <rest> for <id>, etc. (Obvious exceptions are character and string literals which may contain whitespaces, like ' ' and "a b"; in this case explicit syntactic categories are needed for symbols, including whitespaces, that can appear in character and string literals). Non-token syntactic categories, however, may have whitespaces between tokens/syntactic categories on the right sides of production rules. So we tacitly assume that there may be whitespaces between <E> and +, * and <E>, "(" and <E>, etc. For example, "A1 + 123 / CS316" is a valid string in <E>.

Derivations Let β<X>γ be any string with an occurrence of <X> where β and γ are strings of terminals and/or syntactic categories which may be ε. Let <X> → α_i be any choice from a production rule for <X>. Then β<X>γ is said to *derive* βα_iγ *in one step*, formally denoted β<X>γ ⇒ βα_iγ.

Example The following is a sequence of one-step derivations of "CS316" from <id> (the syntactic category being replaced is underlined):

<id> ⇒
<letter><rest> ⇒
C<rest> ⇒
C<letter><rest> ⇒
CS<rest> ⇒
CS<digit><rest> ⇒
CS3<rest> ⇒
CS3<digit><rest> ⇒
CS31<rest> ⇒
CS31<digit><rest> ⇒
CS316<rest> ⇒
CS316ε =
CS316

In this derivation sequence, the leftmost syntactic category is replaced in each step. In general any of the syntactic categories appearing in each step may be replaced. Let the notation α ⇒* β mean α derives β in a finite number of steps. Let Γ be a BNF grammar and <X> any syntactic category in Γ. Then the *language defined by* <X> *in* Γ is defined as the set:

{ β | <X> ⇒* β, β is a string of terminals in Γ }

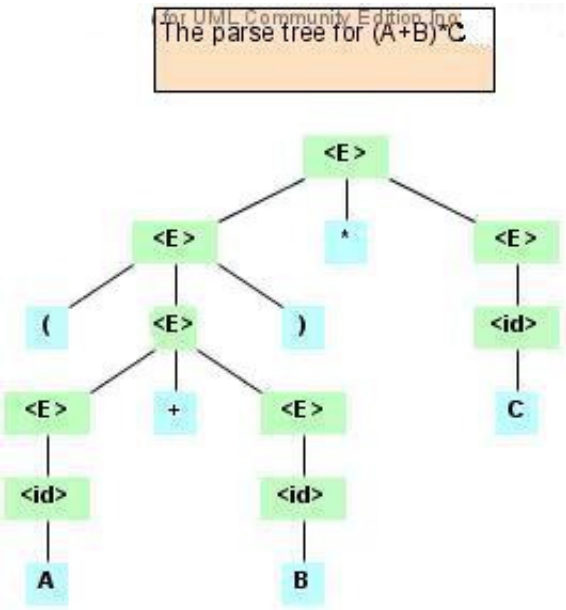
Parse Trees Linear derivation sequences are a useful concept for the formal definition of the languages defined by syntactic categories. They are also useful for deriving token strings, like "CS316", "76453", and "764.34E-5". However, it is very difficult to identify syntactic structures of higher-level, non-token strings in linear derivation sequences, such as the arithmetic expression "CS316+76453−764.34E-5/k+(a*b)". A much better way of identifying and analyzing syntactic structures of strings is to construct derivations in tree form, called **parse trees**. A parse tree is defined as follows:

- Each non-leaf node is labeled by a syntactic category;
- Each leaf node is labeled by a terminal;
- Let *n* be any non-leaf node, <X> the syntactic category labeling it. Then there must be a production rule <X> → α_i such that the children of *n* are labeled by the terminals and syntactic categories in α_i from left to right.

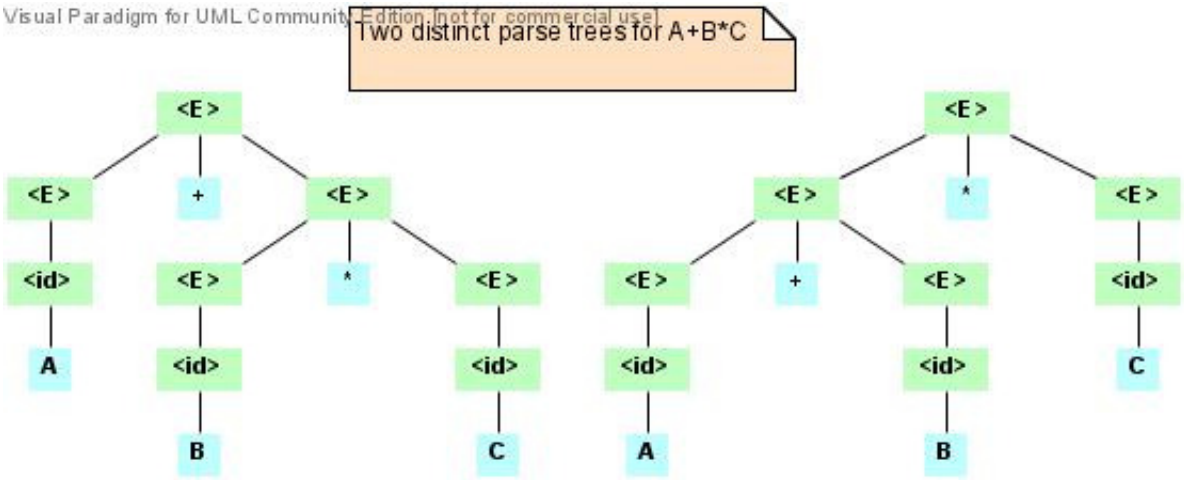
Parse trees can be constructed for tokens or strings of higher-level categories like <E>, and their leaf nodes would be labeled by terminals according to this definition. However, actual compilers' parsers construct parse trees from streams of tokens and do not parse tokens themselves in tree form; here tokens are regarded as "high-level terminal symbols" extracted by lexical analyzers. In accordance with this, whenever constructing parse trees

for non-token categories, we will derive tokens directly from their categories. All leaf nodes of such parse trees will then be labeled by tokens rather than terminals.

Example The following is the parse tree for "(A+B)*C" from the category <E>:



The Ambiguity Problem A syntactic category <X> in a BNF grammar is said to be *ambiguous* if there exists a string α of terminals/tokens such that there exist two or more parse trees for α from <X>. The category <E> defined above is ambiguous as the following distinct parse trees for "A+B*C" demonstrate:



The problem with ambiguity is that distinct parse trees for the same string do not in general represent equivalent semantic information. The first parse tree above represents A+(B*C) while the second represents (A+B)*C. Of course we would tend to assume the first tree is the "right" one based on the conventional precedence rules for the arithmetic operators, but both trees are valid according to the above production rule for <E> since it does not incorporate precedence rules in any way. The problem is clearly aggravated if we are looking at ambiguous syntactic categories for operators, control structures, etc. in a new programming language we are trying to learn – we can never be sure that expressions and statements we write down will be parsed as we intend. The same problem also occurs in compiler construction. The compiler builds parse trees for expressions/statements and then generates intermediate code based on the semantic information encoded in the trees. The first tree above

would generate intermediate code to compute A+(B*C) while the second would generate code to compute (A+B)*C. So the compiler writer suffers the same problem: Which parse tree is supposed to be built? Clearly, formal descriptions of programming language syntax should use unambiguous BNF grammars or, failing that, ambiguous categories should be provided with separate disambiguation specifications (e.g., concise English comments).

The following is an unambiguous BNF grammar for <E>:

<E> → <term> | <term> + <E> | <term> − <E>
<term> → <primary> | <primary> * <term> | <primary> / <term>
<primary> → <id> | <int> | <float> | "(" <E> ")"

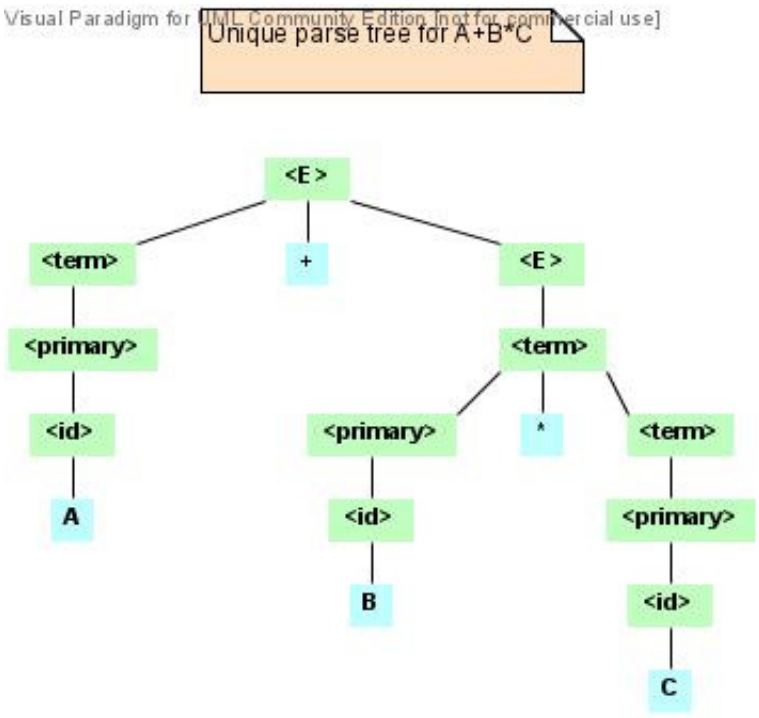
We can read these production rules informally as follows:

An expression is a sequence of one or more terms separated by + or −.
A term is a sequence of one or more primaries separated by * or /.
A primary is an id, int, float, or a parenthesized expression.

This grammar disambiguates expressions by two factors:

- it introduces a 3-level hierarchy that incorporates the usual precedence rule: * and / have higher precedence than + and −;
- at the same precedence level, the operators associate to right.

A term cannot contain + or − unless it is enclosed in (⋯), and a primary cannot contain any operator unless it is enclosed in (⋯).



Extended BNF (EBNF) Grammars The extended BNF grammar provides concise notations for multiple choices and iterations that can be used on the right-hand sides of production rules. It does not, however, provide

additional definitional power since any EBNF grammar can always be translated into an equivalent one in the original BNF.

- **Mandatory Multiple Choice:** $(\alpha_1 \mid \cdots \mid \alpha_n)$ denotes a mandatory choice of exactly one α_i.
- **Optional Multiple Choice:** $[\alpha_1 \mid \cdots \mid \alpha_n]$ denotes an optional choice of at most one α_i.
- **Zero or More Iterations:** $\{ \alpha \}$ denotes zero or more iterations of α, i.e., ε, α, αα, ααα, ..., α^i, ...
- **One or More Iterations:** $\{ \alpha \}^+$ denotes one or more iterations of α, i.e., α, αα, ααα, ..., α^i, ...

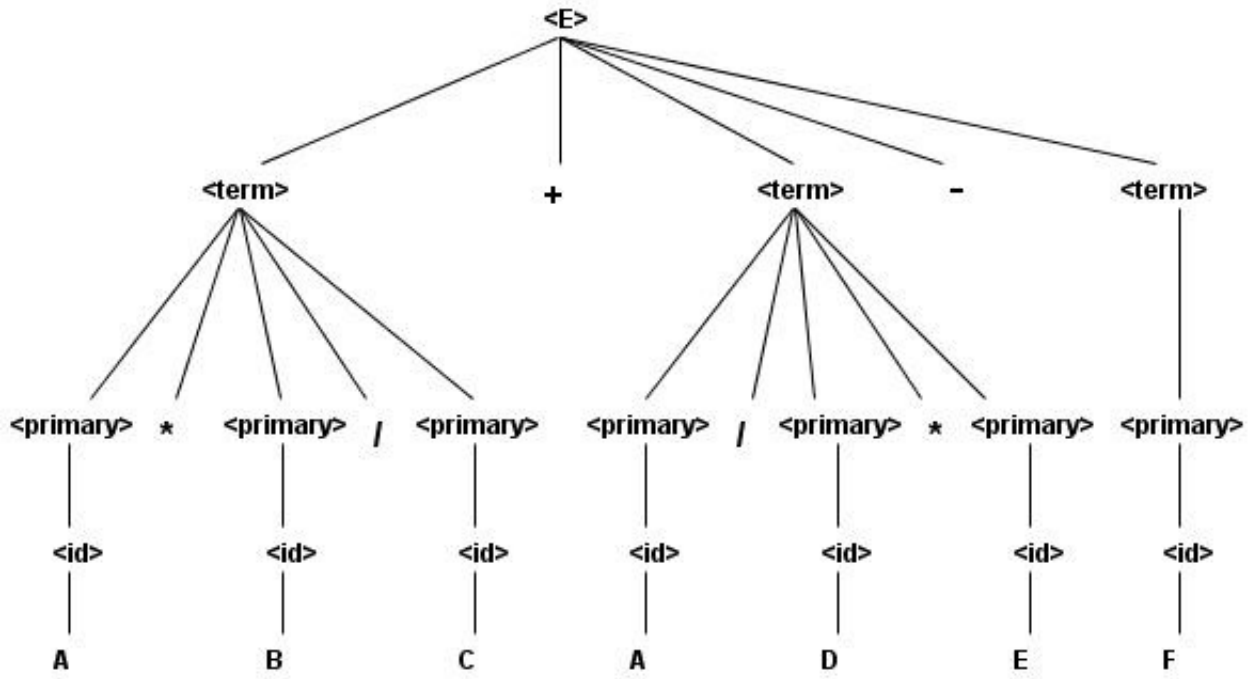
$\{ (\alpha_1 \mid \cdots \mid \alpha_n) \}$ and $\{ (\alpha_1 \mid \cdots \mid \alpha_n) \}^+$ may be abbreviated, respectively, to $\{ \alpha_1 \mid \cdots \mid \alpha_n \}$ and $\{ \alpha_1 \mid \cdots \mid \alpha_n \}^+$.

Example

<int> → { <digit> }$^+$
<id> → <letter> { <letter> | <digit> }
<float> → { <digit> }$^+$ "." { <digit> }$^+$ [(E|e) [+|−] { <digit> }$^+$]
<E> → <term> { (+|−) <term> }
<term> → <primary> { (*|/) <primary> }
<primary> → <id> | <int> | <float> | "(" <E> ")"

Here the left or right associativity of the arithmetic operators must be commented separately, since the iterative definitions of <E> and <term>, unlike the recursive definitions, do not incorporate operator associativity.

In parts of parse trees that use production rules involving {α} or {α}$^+$, the necessary number of iterations of α are displayed at the same level. The following is the parse tree for "A*B/C+A/D*E−F":



Meta Symbols The following characters are called meta symbols (or special symbols) of the BNF/EBNF grammar since they have predefined, fixed meanings throughout all BNF/EBNF grammars: |, <, >, (,), [,], {, }. If any of these needs to be used as a terminal symbol, it must be enclosed in double quotes "..." to avoid confusion. Double quotes may also be used for any terminal symbols — especially easily missed terminals like "." and "," — to make them stand out clearly.

Example

<S> → <id> = <E>; | **if** "(" <B> ")" <S> [**else** <S>] | **while** "(" <B> ")" <S> | **do** <S> **while** "(" <B> ")" | "{" <S List> "}"
<S List> → { <S> }$^{+}$

Note: In the 2-branch conditionals, each **else** matches with the closest preceding unmatched **if**.