BNF, EBNF, and Parse Trees

BNF

We have seen previously seen the notation for formal grammars. However, that notation does not work well in practical grammars where elements of the grammar need to be defined not by single characters by character strings. Moreover, the "➔"is not easy to type in. A common alternative notation is BNF (Backus-Naur Form) as in

<if stmt> ::= **if** <logic expr> **then** <stmt> |
if <logic expr> **then** <stmt> **else** <stmt>

In that notation "::=" is used instead of "➔" and the variables (i.e. non-terminals) are denoted by angle brackets (< ... >). The terminal symbols are denoted without any special markings (but generally bolded in my notes). As we have seen previously, "|" is used to separate rules for the same variable.

We also saw that derivations describe how a sentence can be generated by the grammar.

Let us consider the following grammar:

<assign> ::= <id> **=** <expr>
<id> ::= **A** | **B** | **C**
<expr> ::= <id> + <expr> |<id> * <expr> | **(** <expr> **)** | <id>

The leftmost derivation for the following sentence:

A = B * (A + C) is

<assign> ⇒ <id> **=** <expr> ⇒ **A =** <expr> ⇒ **A =** <id> * <expr> ⇒
A = B * <expr> ⇒ **A = B * (** <expr> **)** ⇒ **A = B * (** <id> + <expr> **)** ⇒
A = B * (A + <expr> **)** ⇒ **A = B * (A** + <id> **)** ⇒ **A = B * (A** + **C)**

while the rightmost derivation is:

<assign> ⇒ <id> **=** <expr> ⇒ <id> **=**<id> * <expr> ⇒
<id> **=** <id> * **(** <expr> **)** ⇒ <id> **=** <id> * **(** <id> + <expr> **)** ⇒ <id> **=**
<id> * **(** <id> + <id> **)** ⇒ <id> **=** <id> * **(** <id> + **C)** ⇒
<id> **=** <id> * **(A** + **C)** ⇒ <id> **= B** * **(A** + **C)** ⇒ **A = B** * **(A** + **C)**

Notice the arrow we use at each step of the derivation. It is "⇒" which is different from the arrow used in productions.

The notation $\overset{*}{\Rightarrow}$ denotes multiple steps. As in

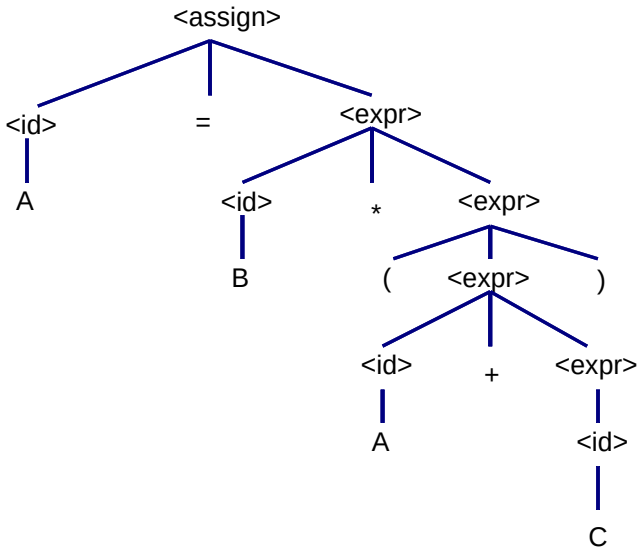
<assign> $\overset{*}{\Rightarrow}$ <id> **=** <id> * **(** <id> + **C)**

Think of it as <assign> ⇒ . . . ⇒ <id> **=** <id> * **(** <id> + **C)**

We will now examine another way to describe the structure of a sentence.

Parse Tree

Another, somewhat more convenient, way to display the same information is via a parse tree. Parse trees have the advantage that, since they are two dimensional, the order in which the rules are applied is not reflected in them. The parse tree for the sentence above is:



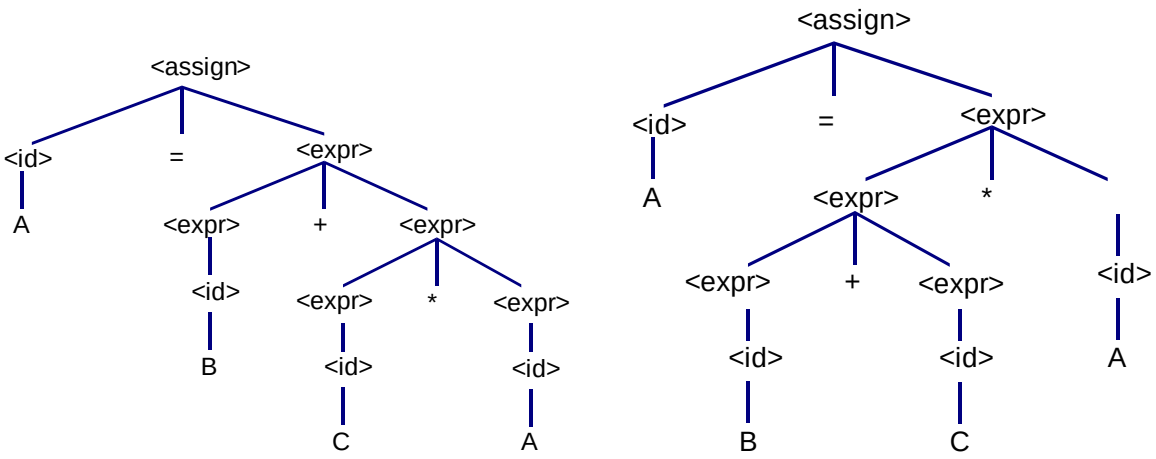
Some things to notice about parse trees:

- The start symbol is always at the root of the tree
- Every leaf node is a terminal
- Every interior node is a non-terminal
- The sentence appears in a left-to-right traversal of the leaves.
- Each parse tree corresponds to a rightmost derivation and to a left most derivation (i.e. there is a one-to-one correspondence between them)

When designing a grammar one must be careful not to create ambiguity. This is when two or more parse trees (or leftmost derivations, or rightmost derivations) could exists. For example:
Consider the following grammar:

<assign> ::= <id> **=** <expr>
<id> ::= **A** | **B** | **C**
<expr> ::= <expr> + <expr> | <expr> * <expr> | **(** <expr> **)**
| <id>

and the sentence **A = B + C * A** which has two distinct parse trees (from what we saw earlier it would also have two leftmost derivations and two rightmost derivations):



The left tree corresponds to the interpretation of the string as A = B + (C * A), while the right tree corresponds to A = (B + C) * A. As you can see the results will be quite different. Assuming A = 4, B = 3, and C = 6. The first interpretation will set A to 27 and the second to 36. *Which one is the one used by most languages?*

We can rewrite the grammar to reflect the precedence of operators + and * and eliminate this ambiguity

<assign> ::= <id> **=** <expr>
<id> ::= **A** | **B** | **C**
<expr> ::= <expr> + <expr> | <factor>
<factor> ::= <factor> * <factor> | **(** <expr> **)** | <id>

The string **A = B + C * A** has now a single parse tree. However, the string **A = B + C + A** has two parse trees as the above grammar does not reflect the associativity of the + operator. This does not cause serious problems with addition but it would with subtraction. For example:

5 - 3 - 2 is equal to 0 if "-" is left associative [i.e (5 - 3) - 2] but is equal to 4 if it is right associative [5 - (3 - 2)].

Note: addition and subtraction are left associative as are division and mod. power is right associative

The following grammar eliminates this ambiguity:

<assign> ::= <id> **=** <expr>
<id> ::= **A** | **B** | **C**
<expr> ::= <expr> + <term> | <term>
<term> ::= <term> * <factor> | <factor>
<factor> ::= **(** <expr> **)** | <id>

This grammar reflects left associativity (associated with left recursive rule). Right associativity is associated with right recursion.

Extended BNF (EBNF)

Three extensions are commonly included (highlighted in red):

- optional part of an RHS
  <selection>::= **if (** <expression> **)** <statement> **[else** <statement> **]**
- indefinite repeat
  <ident list> ::= <identifier> **{ ,** <identifier> **}**
- multiple choice option
  <term> ::= <term> **(** * **|** / **|** % **)** <factor>

Note that by using the braces you may lose some information:

<expr> ::= <term> { + <term> }

does not imply left associativity the way the BNF rule

<expr> ::= <expr> + <term>

does. This can be addressed by enforcing the correct associativity during the semantic analysis phase.

	BNF		EBNF	
1	<a>	::= A | AB	<a>	::= A [B]
2	<snum>	::= - <num> | <num>	<snum>	::= [-] <num>
3	<a>	::= <a> A | A	<a>	::= A { A }
4	<a>	::= <a> A | λ	<a>	::= { A }
5	<block> <stmts>	::= begin <stmts> end ::= <cmd> | <cmd> ; <stmts>	<block>	::= begin <cmd> { ; <cmd> } end
6	<snum>	::= <num> | + <num> | - <num>	<snum>	::= [(+ | -)] <num>

EBNF has the effect of

- removing most “or’s”(|) which reduces the number of rules
- removing most recursion which is replaced by { } loops
- removing occurrences of the null string (λ)

It is also used to build syntax diagrams.