Context Free Grammars

A context-free grammar (CFG) is defined as:

- A finite terminal set V_t;
  these are the tokens produced by the scanner.
- A set of intermediate symbols, called non-terminals, V_n.
- A start symbol, a designated non-terminal, that starts all derivations.
- A set of productions (sometimes called rewriting rules) of the form

  $A \rightarrow X_1 \ldots X_m$

  X_1 to X_m may be any combination of terminals and non-terminals.
  If m =0 we have $A \rightarrow \lambda$
  which is a valid production.

Example

Prog $\rightarrow$ **{ Stmts }**
Stmts $\rightarrow$ **Stmts ; Stmt**
Stmts $\rightarrow$ **Stmt**
Stmt $\rightarrow$ **id = Expr**
Expr $\rightarrow$ **id**
Expr $\rightarrow$ **Expr + id**

Often more than one production shares the same left-hand side.

Rather than repeat the left hand side, an "or notation" is used:

Prog $\rightarrow$ **{ Stmts }**
Stmts $\rightarrow$ **Stmts ; Stmt**
| **Stmt**
Stmt $\rightarrow$ **id = Expr**
Expr $\rightarrow$ **id**
| **Expr + id**

Derivations

Starting with the start symbol, non-terminals are rewritten using productions until only terminals remain.

Any terminal sequence that can be generated in this manner is syntactically valid.

If a terminal sequence can't be generated using the productions of the grammar it is invalid (has syntax errors).

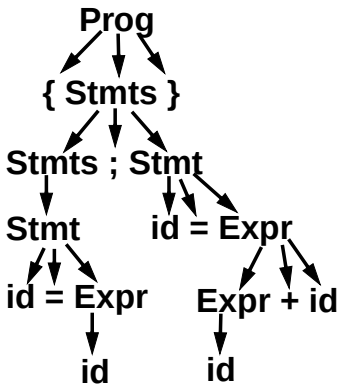
The set of strings derivable from the start symbol is the *language* of the grammar (sometimes denoted L(G)).

For example, starting at **Prog** we generate a terminal sequence, by repeatedly applying productions:

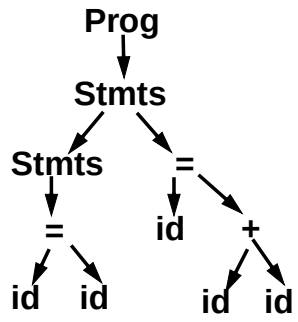
Prog
{ Stmts }
{ Stmts ; Stmt }
{ Stmt ; Stmt }
{ id = Expr ; Stmt }
{ id = id ; Stmt }
{ id = id ; id = Expr }
{ id = id ; id = Expr + id}
{ id = id ; id = id + id}

Parse Trees

To illustrate a derivation, we can draw a derivation tree (also called a parse tree):



An *abstract syntax tree* (AST) shows essential structure but eliminates unnecessary delimiters and intermediate symbols:



If $A \rightarrow \gamma$ is a production then
$\alpha A \beta \Rightarrow \alpha\gamma\beta$
where $\Rightarrow$ denotes a one step derivation (using production $A \rightarrow \gamma$).

We extend $\Rightarrow$ to $\Rightarrow^+$ (derives in one or more steps), and $\Rightarrow^*$ (derives in zero or more steps).

We can show our earlier derivation as

Prog $\Rightarrow$
{ Stmts } $\Rightarrow$
{ Stmts ; Stmt } $\Rightarrow$
{ Stmt ; Stmt } $\Rightarrow$
{ id = Expr ; Stmt } $\Rightarrow$
{ id = id ; Stmt } $\Rightarrow$
{ id = id ; id = Expr } $\Rightarrow$
{ id = id ; id = Expr + id} $\Rightarrow$
{ id = id ; id = id + id}

Prog $\Rightarrow^+$ **{ id = id ; id = id + id}**

When deriving a token sequence, if more than one non-terminal is present, we have a choice of which to expand next.

We must specify, at each step, which non-terminal is expanded, and what production is applied.

For simplicity we adopt a convention on what non-terminal is expanded at each step.

We can choose the leftmost possible non-terminal at each step.

A derivation that follows this rule is a *leftmost derivation*.

If we know a derivation is leftmost, we need only specify what

productions are used; the choice of non-terminal is always fixed.

To denote derivations that are leftmost,

we use $\Rightarrow_L$, $\Rightarrow_L^+$, and $\Rightarrow_L^*$

The production sequence discovered by a large class of parsers (the top-down parsers) is a leftmost derivation, hence these parsers produce a *leftmost parse*.

Prog $\Rightarrow_L$

{ Stmts } $\Rightarrow_L$

{ Stmts ; Stmt } $\Rightarrow_L$

{ Stmt ; Stmt } $\Rightarrow_L$

{ id = Expr ; Stmt } $\Rightarrow_L$

{ id = id ; Stmt } $\Rightarrow_L$

{ id = id ; id = Expr } $\Rightarrow_L$

{ id = id ; id = Expr + id} $\Rightarrow_L$

{ id = id ; id = id + id}

Prog $\Rightarrow_L^+$ **{ id = id ; id = id + id}**

Rightmost Derivations

An alternative to a leftmost derivation is a rightmost derivation, in which the rightmost possible non-terminal is always expanded.

This derivation sequence may seem less intuitive given our normal left-to-right bias, but it corresponds to an important class of parsers (the bottom-up parsers, including CUP).

As a bottom-up parser discovers the productions used to derive a token sequence, it discovers a rightmost derivation, but in *reverse order*.

The last production applied in a rightmost derivation is the first that is discovered, while the first production used, involving the start symbol, is the last to be discovered.

The sequence of productions recognized by a bottom-up parser is a rightmost parse.

It is the exact reverse of the production sequence that represents a rightmost derivation.

For derivations that are rightmost, we use the notation $\Rightarrow_R$, $\Rightarrow_R^+$, and $\Rightarrow_R^*$

Prog $\Rightarrow_R$

{ Stmts } $\Rightarrow_R$

{ Stmts ; Stmt } $\Rightarrow_R$

{ Stmts ; id = Expr } $\Rightarrow_R$

{ Stmts ; id = Expr + id } $\Rightarrow_R$

{ Stmts ; id = id + id } $\Rightarrow_R$

{ Stmt ; id = id + id } $\Rightarrow_R$

{ id = Expr ; id = id + id } $\Rightarrow_R$

{ id = id ; id = id + id}

Prog $\Rightarrow^+$ **{ id = id ; id = id + id}**

You can derive the same set of tokens using leftmost and rightmost derivations; the only difference is the order in which productions are used.

Ambiguous Grammars

Some grammars allow more than one parse tree for the same token sequence. Such grammars are *ambiguous*. Because compilers use syntactic structure to drive translation, ambiguity is undesirable—it may lead to an unexpected translation.

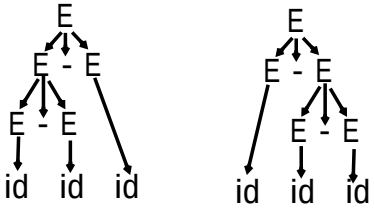
Consider

E $\rightarrow$ **E - E**

| **id**

When parsing the input a-b-c (where a, b and c are scanned as identifiers)

we can build the following two parse trees:



The effect is to parse a-b-c as either (a-b)-c or a-(b-c). These two groupings are certainly not equivalent.

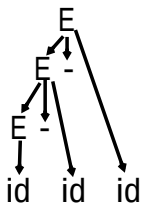
Ambiguous grammars are usually voided in building compilers; the tools we use, like Yacc and CUP, strongly prefer unambiguous grammars.

To correct this ambiguity, we can use

E → E - id

| id

Now a-b-c can only be parsed as:



Operator Precedence

Most programming languages have *operator precedence* rules that state the order in which operators are applied (in the absence of explicit parentheses). Thus in C and Java and CSX, a+b*c means compute b*c, then add in a.

These operators precedence rules can be incorporated directly into a CFG.

Consider

E → E + T

| T

T → T * P

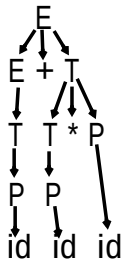
| P

P → id

| (E)

Does a+b*c mean (a+b)*c or a+(b*c)?

The grammar tells us! Look at the derivation tree:



The other grouping can't be obtained unless explicit parentheses are used.

(Why?)