Example

Prog → { Stmts }
Stmts →Stmts ; Stmt
Stmts →Stmt
Stmt →id = Expr
Expr →id
Expr →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 → { Stmts }
Stmts →Stmts ; Stmt
| Stmt
Stmt →id = Expr
Expr →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).

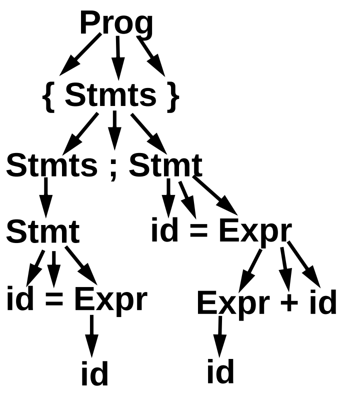
The set of strings derivable from the start symbol is the *language* of the grammar (sometimes denoted L(G)).

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

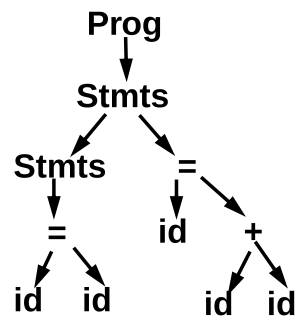
Prog
{ Stmts }
{ Stmts ; Stmt }
{ Stmt ; Stmt }
{ id = Expr ; Stmt }
{ id = id ; Stmt }
{ id = id ; id = Expr }
{ id = id ; id = Expr + id}
{ id = id ; id = id + id}

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

A rightmost derivation is an alternative to a leftmost derivation. Now the rightmost 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. The first production used, involving the start symbol, is discovered last.

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 rightmost derivations, 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

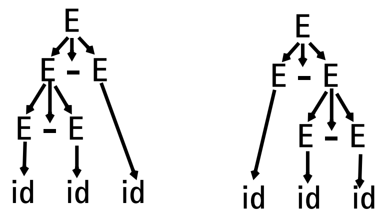
Some grammars allow more than one parse tree for the same token sequence. Such grammars are *ambiguous*. Because compilers use syntactic structure to drive translation, ambiguity is undesirable—it may lead to an unexpected translation.

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.

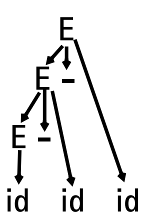
Ambiguous grammars are usually voided in building compilers; the tools we use, like Yacc and CUP, strongly prefer unambiguous grammars.

To correct this ambiguity, we use

E $\rightarrow$ **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 $\rightarrow$ **E + T**

| **T**

T $\rightarrow$ **T * P**

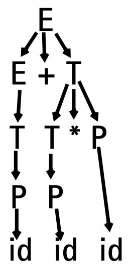
| **P**

P $\rightarrow$ **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?)

Java CUP

Java CUP is a parser-generation tool, similar to Yacc.

CUP builds a Java parser for LALR(1) grammars from production rules and associated Java code fragments.

When a particular production is recognized, its associated code fragment is executed (typically to build an AST).

CUP generates a Java source file **parser.java**. It contains a class **parser**, with a method

```
Symbol parse()
```

The **Symbol** returned by the parser is associated with the grammar's start symbol and contains the AST for the whole source program.

The file **sym.java** is also built for use with a JLex-built scanner (so that both scanner and parser use the same token codes).

If an unrecovered syntax error occurs, **Exception()** is thrown by the parser.

CUP and Yacc accept exactly the same class of grammars—all LL(1) grammars, plus many useful non-LL(1) grammars.

CUP is called as

```
java java_cup.Main < file.cup
```


Java CUP Specifications

Java CUP specifications are of the form:

- Package and import specifications
- User code additions
- Terminal and non-terminal declarations
- A context-free grammar, augmented with Java code fragments

Package and Import Specifications

You define a package name as:

```
package name ;
```

You add imports to be used as:

```
import java_cup.runtime.*;
```

User Code Additions

You may define Java code to be included within the generated parser:

```
action code {: /*java code */ :}
```

This code is placed within the generated action class (which holds user-specified production actions).

```
parser code {: /*java code */ :}
```

This code is placed within the generated parser class .

```
init with{: /*java code */ :}
```

This code is used to initialize the generated parser.

```
scan with{: /*java code */ :}
```

This code is used to tell the generated parser how to get tokens from the scanner.

Terminal and Non-terminal Declarations

You define terminal symbols you will use as:

terminal classname $\mathbf{name_1}$**,** $\mathbf{name_2}$**, ...**

classname is a class used by the scanner for tokens (**CSXToken**, **CSXIdentifierToken**, etc.)

You define non-terminal symbols you will use as:

non terminal classname $\mathbf{name_1}$**,** $\mathbf{name_2}$**, ...**

classname is the class for the AST node associated with the non-terminal (**stmtNode**, **exprNode**, etc.)

Production Rules

Production rules are of the form

name ::= $\mathbf{name_1}$ $\mathbf{name_2}$ **... action ;**

or

name ::= $\mathbf{name_1}$ $\mathbf{name_2}$ **...**
$\mathbf{action_1}$
    | $\mathbf{name_3}$ $\mathbf{name_4}$ **...** $\mathbf{action_2}$
    | **...**
    ;

Names are the names of terminals or non-terminals, as declared earlier.

Actions are Java code fragments, of the form

```
{: /*java code */ :}
```

The Java object assocated with a symbol (a token or AST node) may be named by adding a **:id** suffix to a terminal or non-terminal in a rule.

RESULT names the left-hand side non-terminal.

The Java classes of the symbols are defined in the terminal and non-terminal declaration sections.

For example,

```
prog ::= LBRACE:l stmts:s RBRACE
   {: RESULT =
        new csxLiteNode(s,
          l.linenum,l.colnum); :}
```

This corresponds to the production

prog $\rightarrow$ **{ stmts }**

The left brace is named **l**; the stmts non-terminal is called **s**.

In the action code, a new **CSXLiteNode** is created and assigned to **prog**. It is constructed from the AST node associated with **s**. Its line and column numbers are those given to the left brace, **l** (by the scanner).

To tell CUP what non-terminal to use as the start symbol (**prog** in our example), we use the directive:

```
start with prog;
```


Example

Let's look at the CUP specification for a small subset of CSX. The CFG is

program → **{ stmts }**
stmts → **stmt stmts**
| λ
stmt → **id = expr ;**
| **if (expr) stmt**
expr → **expr + id**
| **expr - id**
| **id**

The corresponding CUP specification is:

```
/***
This Is A Java CUP Specification For a
Small Subset of The CSX Language.
 ***/

/* Preliminaries to set up and use the
scanner.  */

import java_cup.runtime.*;
parser code {:
 public void syntax_error
  (Symbol cur_token){
   report_error(
    "CSX syntax error at line "+
    String.valueOf(((CSXToken)
       cur_token.value).linenum),
    null);}
:};

init with {:              :};
scan with {:
   return Scanner.next_token();
:};
```



```
/* Terminals (tokens returned by the
scanner). */
terminal CSXIdentifierToken IDENTIFIER;
terminal CSXToken SEMI, LPAREN, RPAREN,
ASG, LBRACE, RBRACE;
terminal CSXToken PLUS, MINUS, rw_IF;

/* Non terminals */
non terminal csxLiteNode prog;
non terminal stmtsNode   stmts;
non terminal stmtNode    stmt;
non terminal exprNode    exp;
non terminal nameNode    ident;


start with prog;

prog::= LBRACE:l stmts:s RBRACE
 {: RESULT=
     new csxLiteNode(s,
          l.linenum,l.colnum); :}
;

stmts::= stmt:s1  stmts:s2
 {: RESULT=
     new stmtsNode(s1,s2,
       s1.linenum,s1.colnum);
 :}
```

```
|
 {: RESULT= stmtsNode.NULL; :}
;
stmt::= ident:id ASG exp:e SEMI
 {: RESULT=
       new asgNode(id,e,
           id.linenum,id.colnum);
 :}

| rw_IF:i LPAREN exp:e RPAREN  stmt:s
 {: RESULT=new ifThenNode(e,s,
            stmtNode.NULL,
            i.linenum,i.colnum); :}
;
exp::=
 exp:leftval PLUS:op ident:rightval
 {: RESULT=new binaryOpNode(leftval,
      sym.PLUS, rightval,
      op.linenum,op.colnum); :}

| exp:leftval MINUS:op ident:rightval
 {: RESULT=new binaryOpNode(leftval,
           sym.MINUS,rightval,
           op.linenum,op.colnum); :}
| ident:i
 {: RESULT = i; :}
;
```



```
ident::= IDENTIFIER:i
 {: RESULT = new nameNode(
   new identNode(i.identifierText,
                i.linenum,i.colnum),
   exprNode.NULL,
   i.linenum,i.colnum); :}
;
```


Let's parse

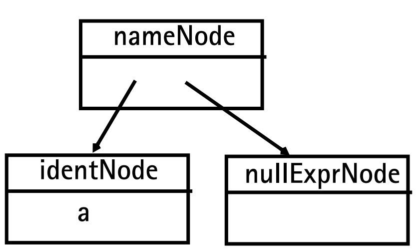
{ a = b ; }

First, **a** is parsed using

```
ident::= IDENTIFIER:i
 {: RESULT = new nameNode(
   new identNode(i.identifierText,
                i.linenum,i.colnum),
   exprNode.NULL,

   i.linenum,i.colnum); :}
```

We build

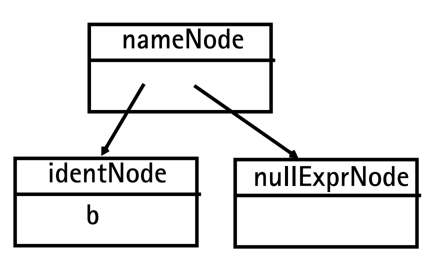


Next, **b** is parsed using

```
ident::= IDENTIFIER:i
 {: RESULT = new nameNode(
   new identNode(i.identifierText,
                i.linenum,i.colnum),
   exprNode.NULL,

   i.linenum,i.colnum); :}
```

We build



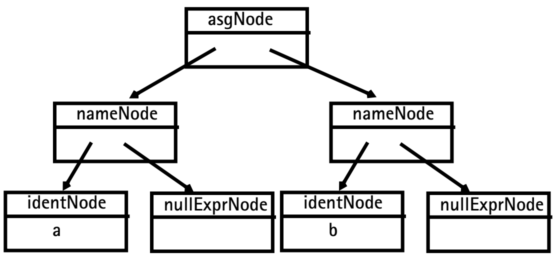
Then `b`'s subtree is recognized as an **exp**:

```
| ident:i
 {: RESULT = i; :}
```

Now the assignment statement is recognized:

```
stmt::= ident:id ASG exp:e SEMI
 {: RESULT=
       new asgNode(id,e,
           id.linenum,id.colnum);
 :}
```

We build

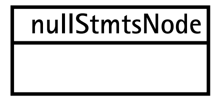


The **stmts** → λ production is matched (indicating that there are no more statements in the program).

CUP matches

```
stmts::=
 {: RESULT= stmtsNode.NULL; :}
```

and we build



Next,

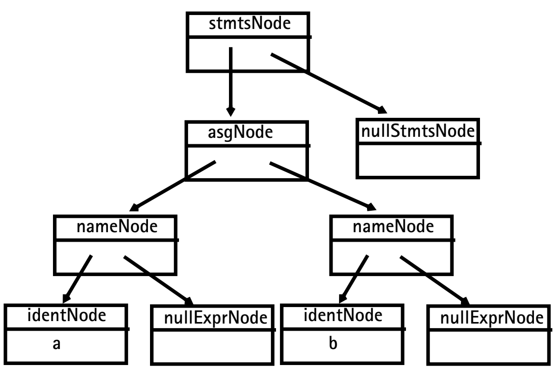
stmts → **stmt** **stmts**

is matched using

```
stmts::= stmt:s1  stmts:s2
 {: RESULT=
     new stmtsNode(s1,s2,
       s1.linenum,s1.colnum);

 :}
```


This builds



As the last step of the parse, the parser matches

program → **{** **stmts** **}**

using the CUP rule

```
prog::= LBRACE:l stmts:s RBRACE
 {: RESULT=
     new csxLiteNode(s,
          l.linenum,l.colnum); :}
;
```


The final AST reurned by the parser is