

PROPERTIES OF REGULAR EXPRESSIONS AND FINITE AUTOMATA

- Some token patterns *can't* be defined as regular expressions or finite automata. Consider the set of balanced brackets of the form $[[[...]]]$. This set is defined formally as

$$\{ [^m]^m \mid m \geq 1 \}.$$

This set is *not* regular.

No finite automaton that recognizes *exactly* this set can exist.

Why? Consider the inputs $[$, $[[$, $[[[$, ...

For two different counts (call them i

and j) $[^i$ and $[^j$ must reach the same state of a given FA! (Why?)

Once that happens, we know that if $[^i]^i$ is accepted (as it should be), the $[^j]^i$ will also be accepted (and that should not happen).

- $\bar{R} = V^* - R$ is regular if R is.

Why?

Build a finite automaton for R . Be careful to include transitions to an “error state” s_E for illegal characters.

Now invert final and non-final states. What was previously accepted is now rejected, and what was rejected is now accepted. That is, \bar{R} is accepted by the modified automaton.

- **Not all subsets of a regular set are themselves regular.** The regular expression $[^+]^+$ has a subset that isn't regular. (What is that subset?)

- Let R be a set of strings. Define R^{rev} as all strings in R , in reversed (backward) character order.

Thus if $R = \{abc, def\}$

then $R^{\text{rev}} = \{cba, fed\}$.

If R is regular, then R^{rev} is too.

Why? Build a finite automaton for R .

Make sure the automaton has only one final state. Now *reverse* the direction of all transitions, and interchange the start and final states. What does the modified automation accept?

- If R_1 and R_2 are both regular, then $R_1 \cap R_2$ is also regular. We can show this two different ways:
 1. Build two finite automata, one for R_1 and one for R_2 . Pair together states of the two automata to match R_1 and R_2 simultaneously. The paired-state automaton accepts only if both R_1 and R_2 would, so $R_1 \cap R_2$ is matched.
 2. We can use the fact that $R_1 \cap R_2$ is $\overline{\overline{R_1} \cup \overline{R_2}}$. We already know union and complementation are regular.

READING ASSIGNMENT

- Read Chapter 4 of **Crafting a Compiler**

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
$$A \rightarrow X_1 \dots X_m$$
 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 }

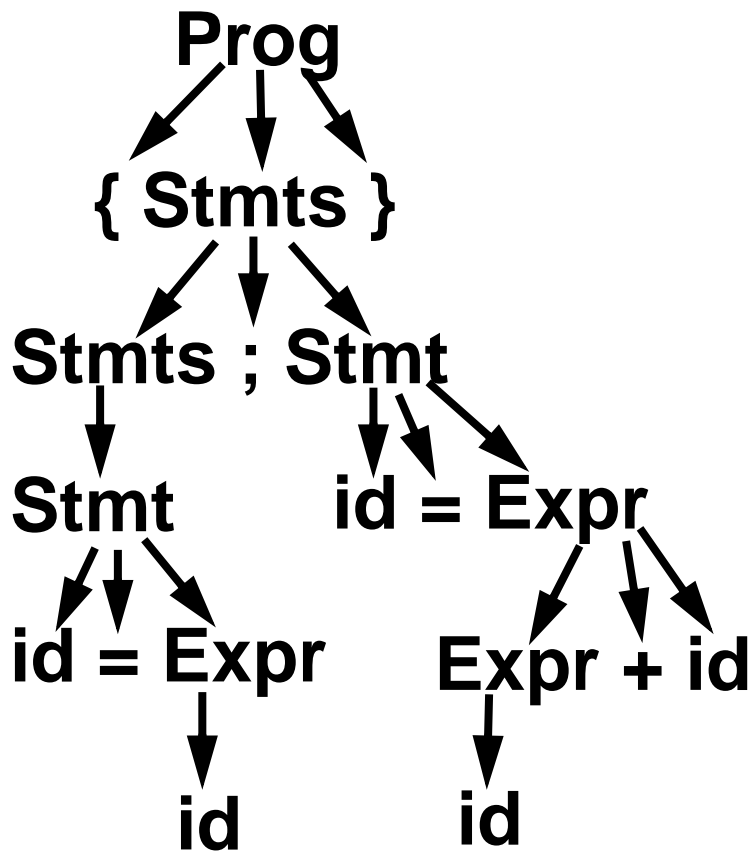
{ id = id ; id = Expr }

{ id = id ; id = Expr + id }

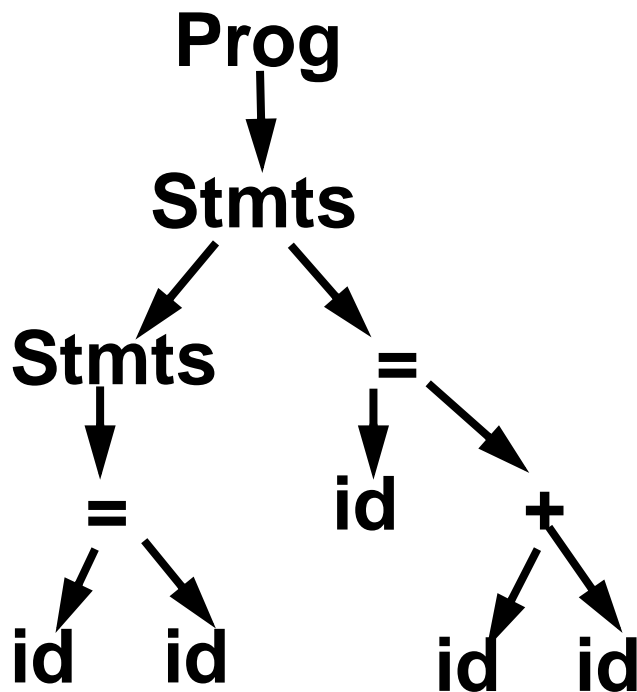
{ id = id ; id = id + id }

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

{ Stmt } \Rightarrow

{ Stmt ; 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

{ Stmt } \Rightarrow_L

{ Stmt ; 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.

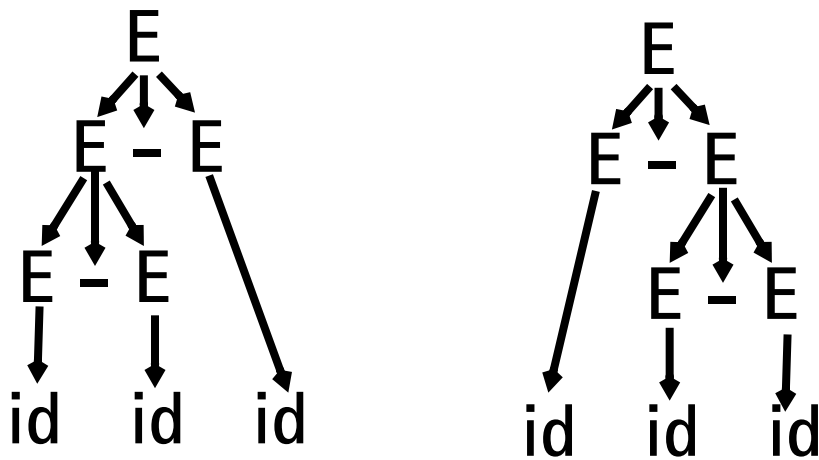
Ambiguous Grammars

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

$$\begin{array}{l} \mathbf{E} \rightarrow \mathbf{E} - \mathbf{E} \\ \quad | \quad \mathbf{id} \end{array}$$

When parsing the input a-b-c (where a, b and c are scanned as identifiers) we can build the following two parse trees:



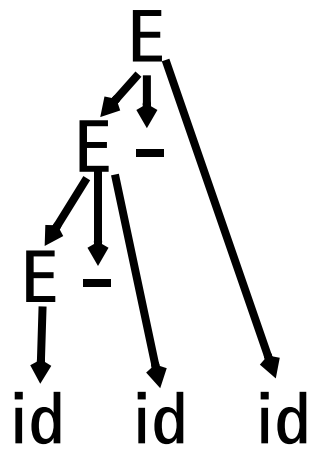
The effect is to parse a-b-c as either (a-b)-c or a-(b-c). These two groupings are certainly not equivalent.

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

$$\begin{array}{l}
 \mathbf{E} \rightarrow \mathbf{E - id} \\
 \quad \quad \quad \mathbf{| \ id}
 \end{array}$$

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**.

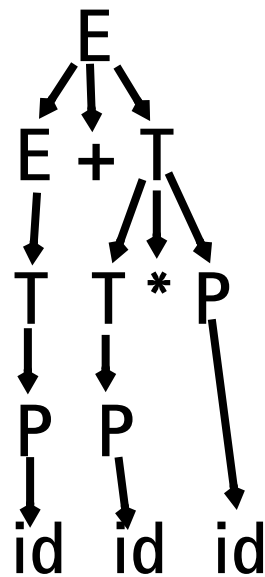
These operators precedence rules can be incorporated directly into a CFG.

Consider

$$E \rightarrow E + T$$
$$| T$$
$$T \rightarrow T * P$$
$$| P$$
$$P \rightarrow \text{id}$$
$$| (E)$$

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 name1, name2, ...
```

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 name1, name2, ...
```

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 ::= name1 name2 ... action ;
```

or

```
name ::= name1 name2 ...
```

```
action1
```

```
| name3 name4 ... action2
```

```
| ...
```

```
;
```

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 associated 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:1 stmts:s RBRACE  
{: RESULT =  
    new csxLiteNode(s,  
        1.linenum,1.colnum); :}
```

This corresponds to the production

prog → { **stmts** }

The left brace is named **1**; 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, **1** (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;
```