Parsing

RE macros:

digits = [0-9]+
sum = (digits "+")* digits

7 + 11 + 42

How about:

digits = [0-9]+
sum = (expr "+")* expr
expr = digits | "(" sum ")"

7 + (11 + 42)

not regular language

Context-Free Grammar

Functionality:

- similar to RE
- plus recursion

Top-level alternation:

expr = a b (c|d) e

$\Downarrow$

aux = c|d
expr = a b aux e

$\Downarrow$

aux = c
aux = d
expr = a b aux e

CFG

Recursion instead of Kleene closure

$$expr = (a\ b\ c)^*$$

$$\Downarrow$$

$$expr = (a\ b\ c)\ expr$$

$$expr = \varepsilon$$

Context-Free Grammar

- terminal symbols (tokens)
- non-terminal symbols
- start symbol
- rules of the form

$$N \to X^* \quad \text{where}$$

N nonterminal

X (non-)terminal

Example:

digit = 0

⋮

digit = 9

digits = digit

digits = digit digits

sum = expr "+" ~~expr~~ sum

sum = expr

expr = digits

expr = "(" sum ")"

Regular Languages

Right-recursive

$$N \to t$$
$$N \to t\,N$$

Left-recursive

$$N \to t$$
$$N \to N\,t$$

Example

$\underline{S} \rightarrow S ; S$

$S \rightarrow id := E$

$S \rightarrow print(L)$

$E \rightarrow id$

$E \rightarrow num$

$E \rightarrow E + E$

$E \rightarrow (S, E)$

$L \rightarrow E$

$L \rightarrow L, E$

Example (cont.)

$L \rightarrow E$ left-recursive

$L \rightarrow L, E$

or

$L \rightarrow E$ right-recursive

$L \rightarrow E, L$

or

$L \rightarrow E$ ambiguous

$L \rightarrow L, L$

styles of rules

left-recursive

doesn't work with
top-down parsing
(e.g., hand-written parser)
$O(n^2)$

right-recursive

may be hard for
bottom-up parsing (e.g., CUP)
$O(n^2)$

ambiguous

are a pain
$O(n^3)$ Earley's algorithm

Parser Types

LL(1) – recursive descent

LR(1)

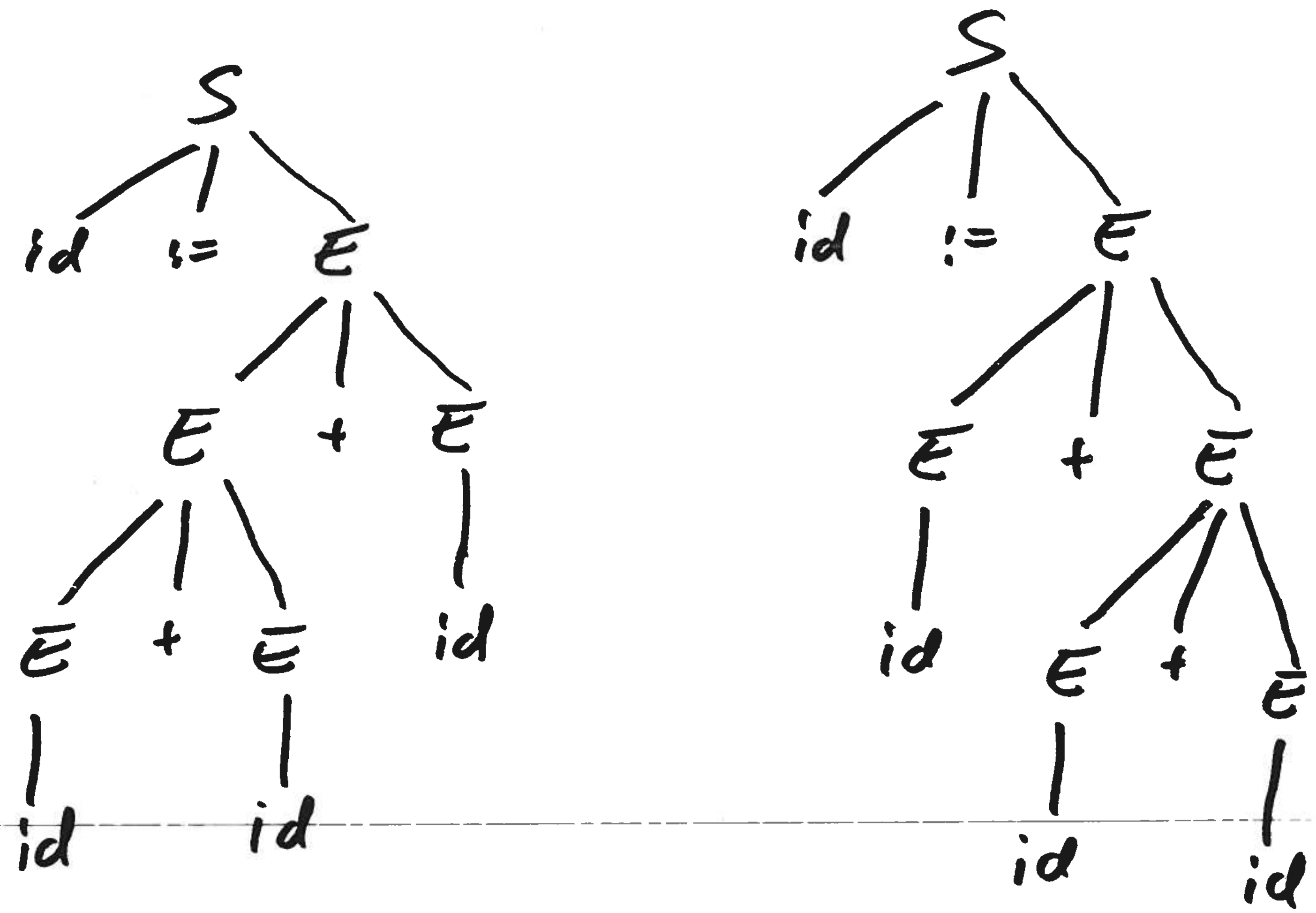
LL(k) – Java CC, ANTLR

LALR(1) – yacc, bison, CUP

see p. 68

Parse Trees

id := id + id + id



Derivations

$\underline{S}$

$S \,;\, \underline{S}$

$\underline{S} \,;\, id := E$

$id := \underline{E} \,;\, id = E$

$id := num \,;\, id := E$

$\vdots$

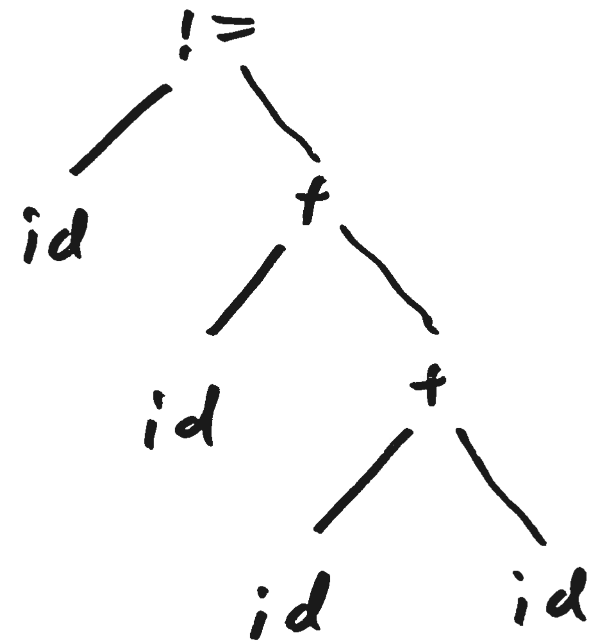
left-most derivation

= top-down parsing

right-most derivation

= bottom-up parsing

Abstract Parse Trees



Ambiguous Grammars

$E \to id$

$E \to num$

$E \to E * E$

$E \to E / E$

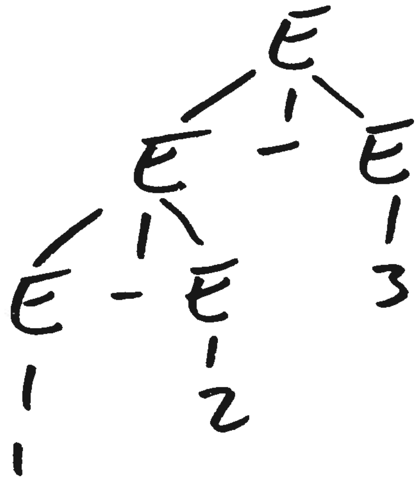
$E \to E + E$

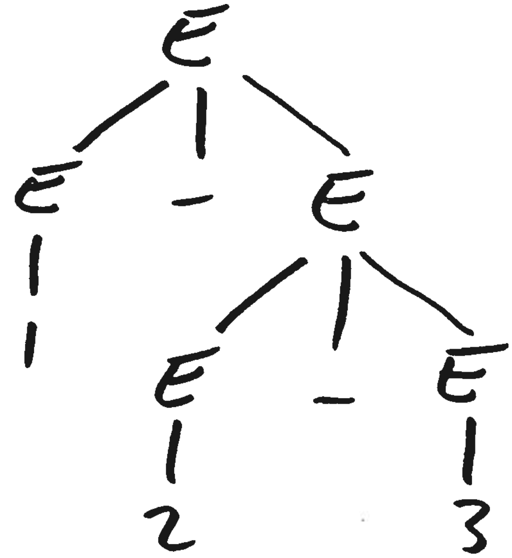
$E \to E - E$

$E \to (E)$

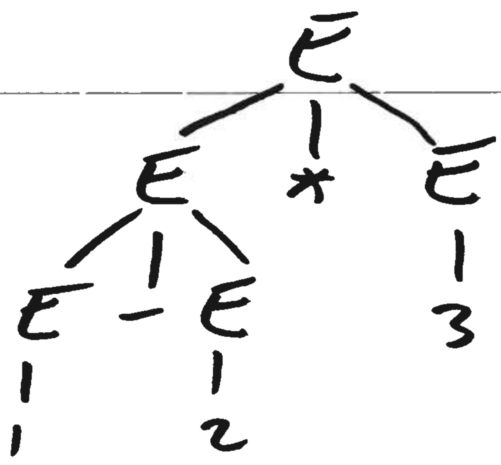
Ambiguous Grammars

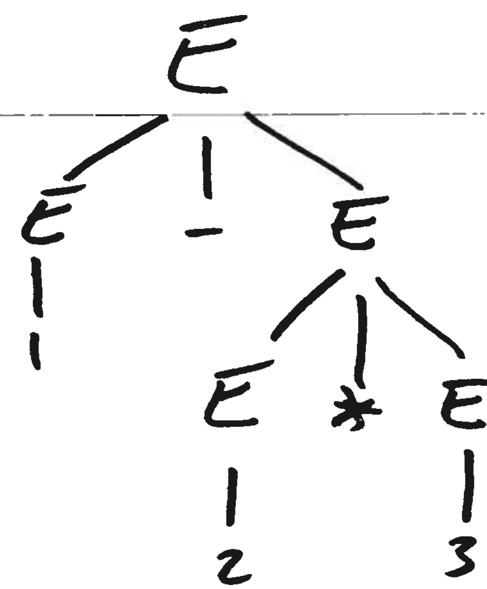
1 - 2 - 3





1 - 2 * 3





Modify Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

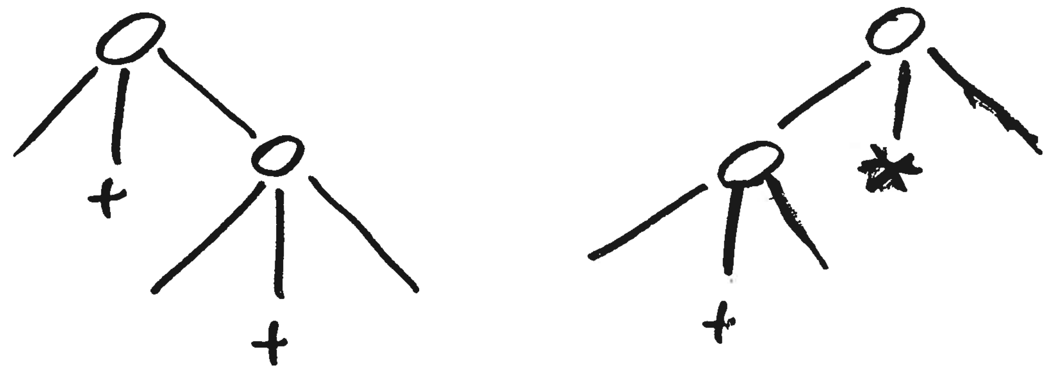
$T \rightarrow F$

$F \rightarrow id$

$F \rightarrow num$

$F \rightarrow (E)$

We don't get



IF Statement

$S \to$ if E then S else S

$S \to$ if E then S

$S \to$ other

$\Downarrow$

$S \to M$

$S \to U$

$M \to$ if E then M else M

$M \to$ other

$U \to$ if E then S

$U \to$ if E then M else U

CUP: Precedence Directives

```
precedence nonassoc  EQ, NEQ;
precedence left      PLUS, MINUS;
precedence left      TIMES, DIV;
precedence right     EXP;
precedence left      UMINUS;
```

```
exp ::= INT
      | exp PLUS exp
      | exp MINUS exp
      | exp TIMES exp
      | MINUS exp
        %prec UMINUS
```

LR - Parsing

LR(k)

k tokens lookahead

right-most derivations

left-to-right parse

$\underbrace{\mathrm{LA}}_{\text{lookahead}}\mathrm{LR}(1)$

Parse Engine

stack

DFA – applied to the stack
– edges labeled with (non) terminals

Actions

s_n	shift and goto state n
g_n	goto state n
r_k	reduce by rule k
a	accept (shift EOF)
–	error

Example (p. 58)

Stack	Input	Action
$_1$	a := 7 \$	shift 4
$_1$ id_4	:= 7 \$	shift 6
$_1$ id_4 $:=_6$	7 \$	shift 10
$_1$ id_4 $:=_6$ num_{10}	\$	reduce $E \rightarrow num$
$_1$ id_4 $:=_6$ E_{11}	\$	reduce $S \rightarrow id := E$
$_1$ S_2	\$	accept

$E \rightarrow num$

$E \rightarrow id$

$S \rightarrow id := E$

Parsing Table

	id	num	print	;	,	+	:=	(	)	$	S	E	L
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7			⋮										
8													
9													
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
⋮				⋮									

Conflicts

shift-reduce	resolved by shift
reduce-reduce	resolved by order of rules

CUP output

state 17: shift/reduce conflict (shift ELSE, reduce 4)

stm ::= IF ID THEN stm •

stm ::= IF ID THEN stm • ELSE stm

ELSE	shift 19
•	reduce by rule 4

Actions

shift (n) - eat one token
- shift state n onto stack

reduce (k) - pop n states off stack
where n is #tokens on
RHS of rule k
- in state on top of stack,
look up X (LHS of rule k)
in goto (m)
- push m onto stack

accept - stop, report success

error - stop, report error

Reduce / Reduce Conflicts

```
precedence  left  OR;
precedence  left  AND;
precedence  left  PLUS;

stm  ::=  ID  ASSIGN  ae
       |  ID  ASSIGN  be;

be   ::=  be  OR  be
       |  be  AND  be
       |  ae  EQUAL  ae
       |  ID;
                        R/R
ae   ::=  ae  PLUS  ae  conflict
       |  ID;
```

Reduce/Reduce Conflict

state 5: R/R conflict
between rule 6 and 4
on EOF

be ::= ID.
ae ::= ID.

PLUS	R6
AND	R4
OR	R4
EQUAL	R6
EOF	R4 ←
.	error

Solution: push decision to semantic analysis

```
stm ::= ID ASSIGN E;
E   ::= E OR E
      | E AND E
      | E EQUAL E
      | E PLUS E
      | ID;
```

Solution: push decision to scanner

g++ : IDENT
TYPEIDENT
LABELIDENT

C x (a, b, c);

Shift-Reduce Errors

use precedence declarations
for operators

```
%precedence left PLUS
%precedence right ASSIGN
%precedence nonassoc EQUAL
```

use %prec if no appropriate
operators available

```
%precedance left UMINUS

E ::= MINUS E %prec UMINU
```

Error Recovery

On parse error:

- pop stack until there is a shift on error
- shift error token
- discard input until we are in a state with non-error action
- resume parsing

Error Recovery

$$E \to ID$$

$$E \to E + E$$

$$E \to (E)$$

$$E \to (\text{error})$$

$$Es \to E$$

$$Es \to Es ; E$$

$$Es \to \text{error} ; E$$

Global Error Repair

Example:

```
let type a := intArray [10] of 0;
```

Solutions:

- delete type...0 (error prod.)
- replace := with =
  (local repair)
- replace type with var
  (global repair)

Burke-Fisher Error Repair

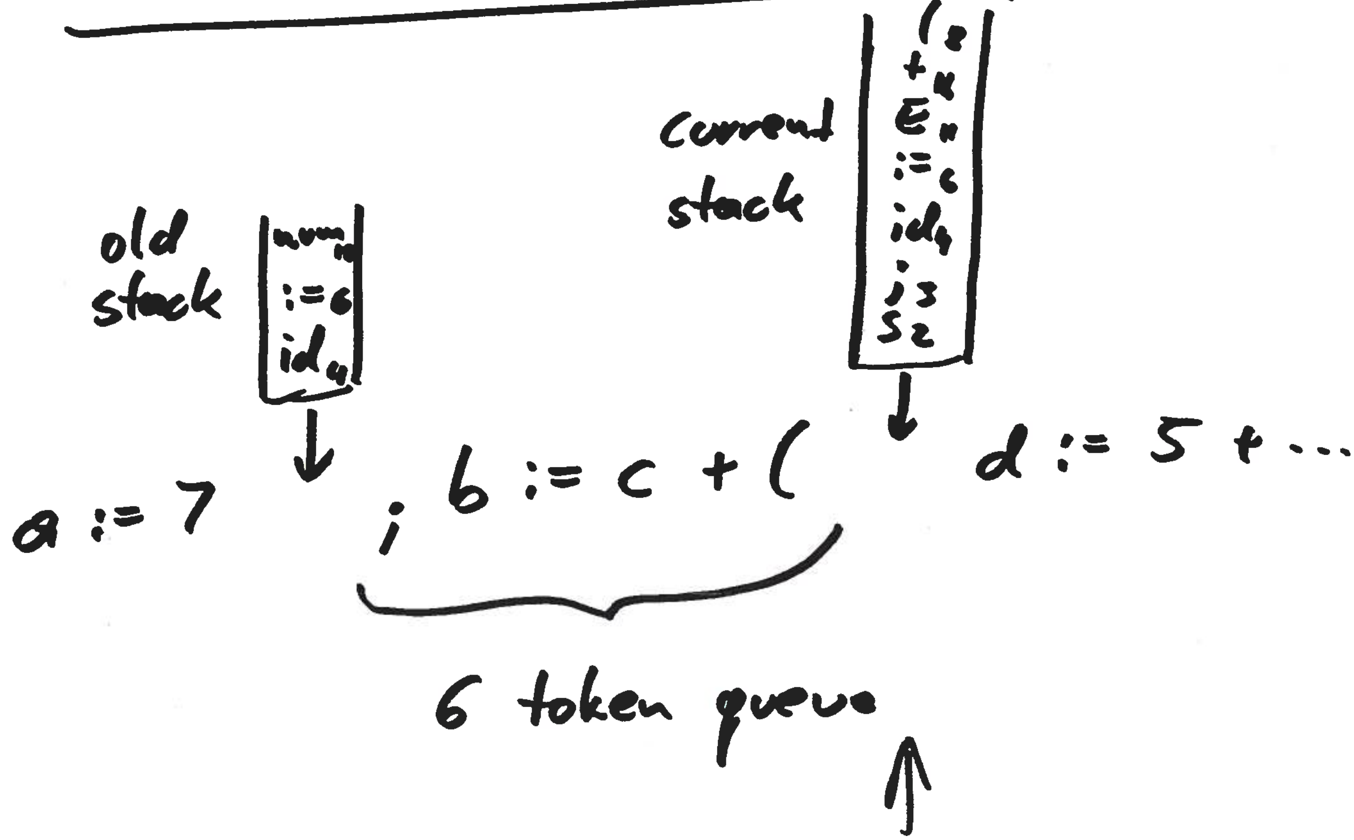
- consider possible single token insertions/deletions/substitutions in last K tokens ($K = 15$)
- use the repair that gets us the farthest, preferably at least R tokens ($R = 4$)

Grammar Rules for Error Reporting

```
Decl ::= Type ID LBRACK INT RBRACK
         ASSIGN LBRACE Explist RBRACE
         SEM
         {: ... :}

       | ....
         ASSIGN LBRACE RBRACE SEM
         {: error("...");
            RESULT = NULL;
         :}
```

Burke - Fisher error repair



Cost

for window size K and

N tokens

$$\underbrace{K}_{\text{deletions}} + \underbrace{N \cdot K}_{\text{insertions}} + \underbrace{N \cdot K}_{\text{substitutions}}$$

ML-Yacc

Semantic actions for insertions

```
%value ID ("bogus")
%value INT (1)
%value STRING ("")
```

Programmer-specified substitutions

```
%change EQ -> ASSIGN
      | ASSIGN -> EQ
      | SEMI ELSE -> ELSE
      | -> IN INT END
```

Resolving Shift-Reduce Conflicts

```
Exp ::= Var
      | ID LBRACK Exp RBRACK OF Exp

Var ::= ID
      | Var LBRACK Exp RBRACK
```

Example: S-R Conflict

```
Stm ::= VarDec
      | Assign;

VarDec ::= Type ID SEM;
Assign ::= LVal EQ Exp SEM;

Type ::= QualName
       | BuiltInType;

LVal ::= QualName
       | LVal LBRACK Exp RBRACK
       | LVal DOT ID;

QualName ::= ID
           | QualName DOT ID;
```

LR Parsing

Grammar

↓

? CUP

↓

Parse Tables
+
Shift-Reduce Parser

Read pp. 60-68

LR(0)
SLR
→ LALR(1)
LR(1)

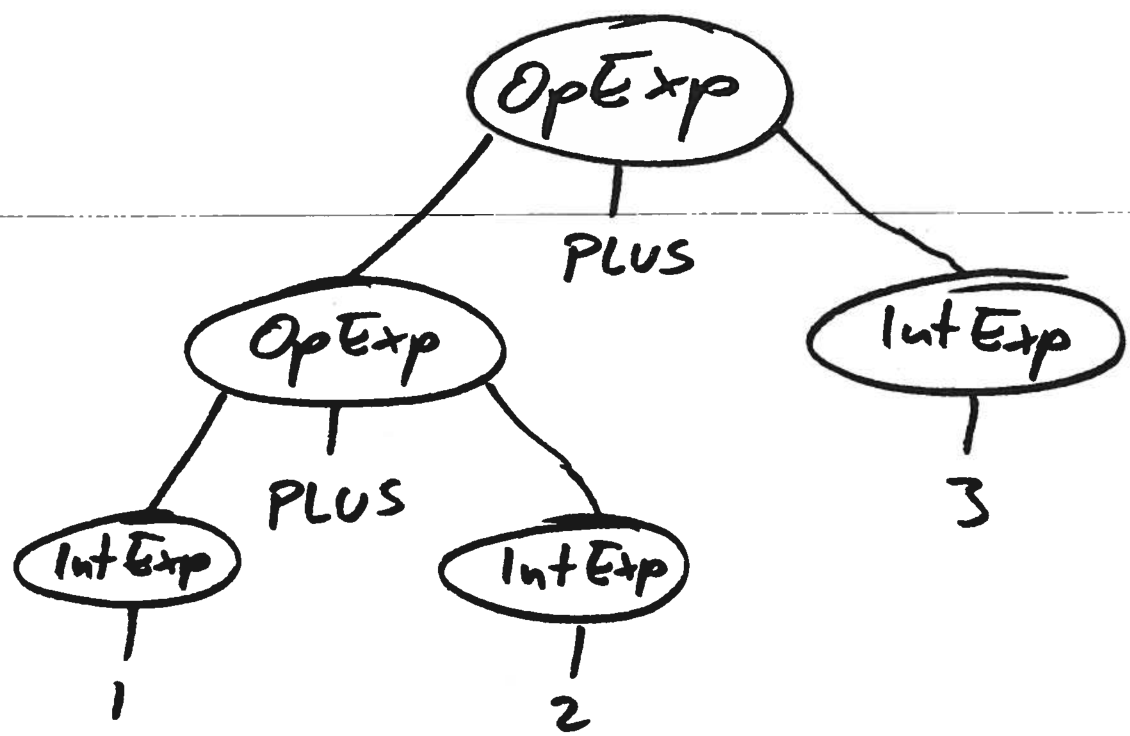
Building Parse Trees

```
precedence left PLUS

Exp ::= INT:e
        {: RESULT = new IntExp(e); :}

      | Exp:e1 PLUS Exp:e2
        {: RESULT =
             new OpExp(e1, PLUS, e2); :}
```

Input: 1+2+3



Building Parse Trees

```
Exp ::= LPAREN Exp:e RPAREN
      {: RESULT = e; :}
    ;

Foolist ::=
      {: RESULT = null; :}
    | Foos:l
      {: RESULT = l; :}
    ;
```

LL - Parsing

Recursive Descent

```
void S() {
  switch (tok) {
    case IF:
        eat(IF);
        E();
        eat(THEN);
        S();
        eat(ELSE);
        S();
        break;
    case BEGIN:
        ⋮
        break;
    default: error();
  }
}
```

Problems in LL Parsing

Left recursion:

$$\begin{aligned} X &\rightarrow X\ y \\ &\mid z \end{aligned}$$

Common left factors:

$$\begin{aligned} X &\rightarrow a\ Y \\ &\mid a\ Z \end{aligned}$$

Rules starting with nonterms:

$$\begin{aligned} X &\rightarrow Y \\ &\mid Z \end{aligned}$$

Empty RHS:

$$\begin{aligned} X &\rightarrow \\ &\mid Y \end{aligned}$$

LL Parsing

- must chose alternatives (rules) based on lookahead
- must know FIRST, FOLLOW for each rule

Nullable, FIRST, FOLLOW

Nullable:

can X derive emty string?

FIRST:

set of terminals that can begin strings derived from X

FOLLOW:

set of terminals that can follow X

Example

$$
\begin{aligned}
Z &\to d \\
&\mid X\,Y\,Z \\
Y &\to \\
&\mid c \\
X &\to Y \\
&\mid a
\end{aligned}
$$

Example

	nullable	FIRST	FOLLOW
X	yes	a c	a c d
Y	yes	c	a c d
Z	no	a c d	

Construction of Predictive Parser

Enter production $X \to \gamma$
in row X, column T
for each $T \in \text{FIRST}(\gamma)$;

if γ is nullable, enter
production in row X, col. T

for each $T \in \text{FOLLOW}(X)$

Example

	a	c	d
X	$X \to a$ $X \to Y$	$X \to Y$	$X \to Y$
Y	$Y \to$	$Y \to$ $Y \to c$	$Y \to$
Z	$Z \to XYZ$	$Z \to XYZ$	$Z \to d$ $Z \to XYZ$

not LL(1)

Summary: Parsing

- recognize context-free syntactic structures on token stream
- syntax description using grammar
- bottom-up parsing
  - grammar $\xrightarrow{\text{CUP}}$ table-driven parser
  - uses push-down automaton
  - LR(0), SLR, LALR(1), LR(1), LR(k)
- top-down parsing
  - recursive descent (hand-written)
  - predictive parser (tool)
  - LL(1), LL(k) (JavaCC)
- one-pass vs. multi-pass