Parser

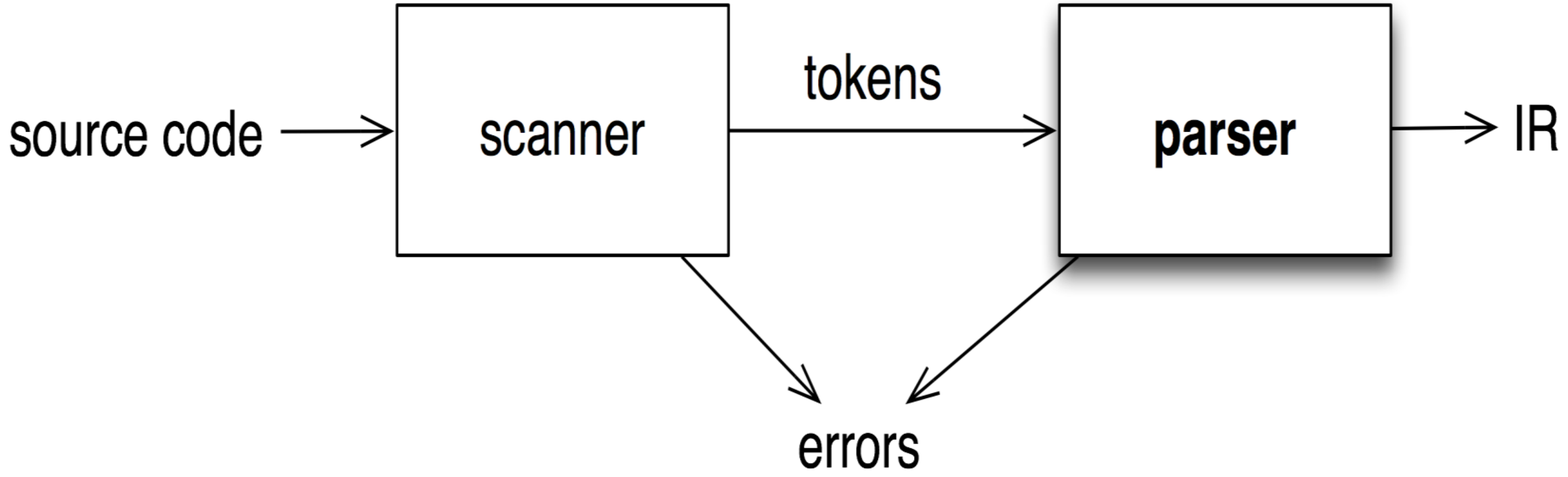
Second Phase of the compiler

- Parser – Typically integrated with the lexical phase of the compiler
- Top Down Parser
- Bottom Up Parser

Functions of the Parser

- Validate the syntax of the programming language
- Points out errors in the statements

Role of the Parser



Role of the Parser

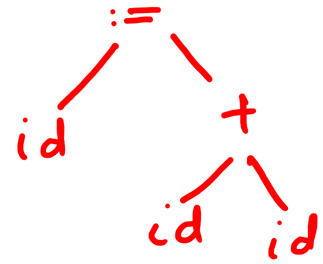
x = y z

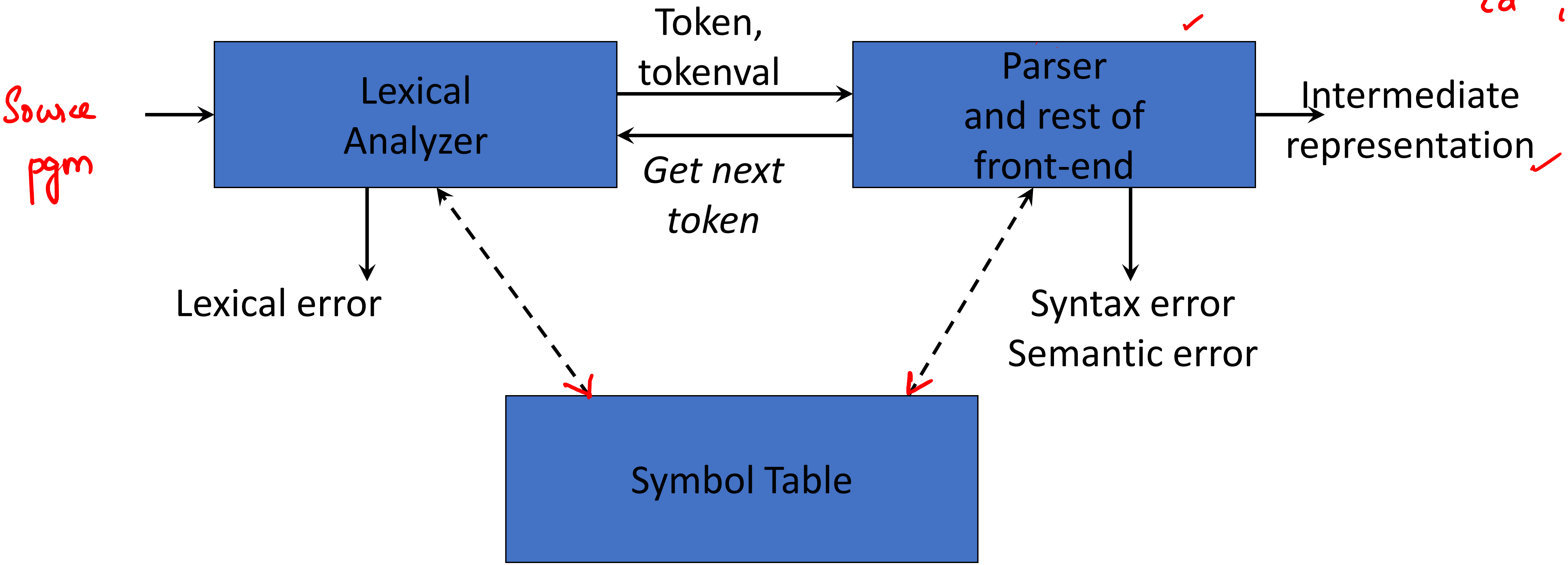
x = y+z

id = id+id

id op

id = id id





General Types of Parsers

- Universal Parsers
  - Cocke- Younger-Kasami
  - Earley's Algorithm
- Top-Down Parsers
- Bottom Up Parsers

Universal Parsers

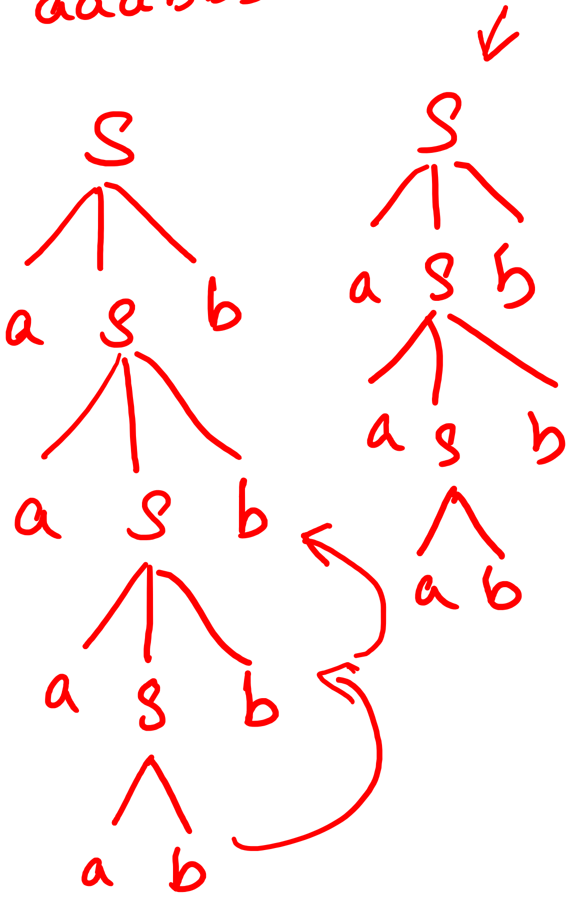
- Can parse any Grammar
- Use in NLP
- But too inefficient in Compilers

Top Down Parsers

- Build the parse trees from the top to the bottom
- Recursive Descent parsers – requires backtracking
- LL Parsers – No Backtracking

$S \rightarrow aSb \mid ab$

aaabbb ✓



Example

Consider the Grammar

$S \to c\,A\,d$

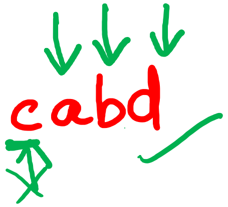
$A \to ab \mid a$

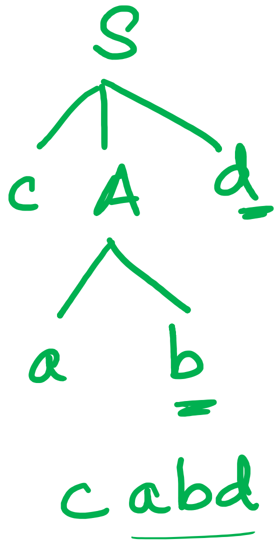
Let the input be "*cad*"

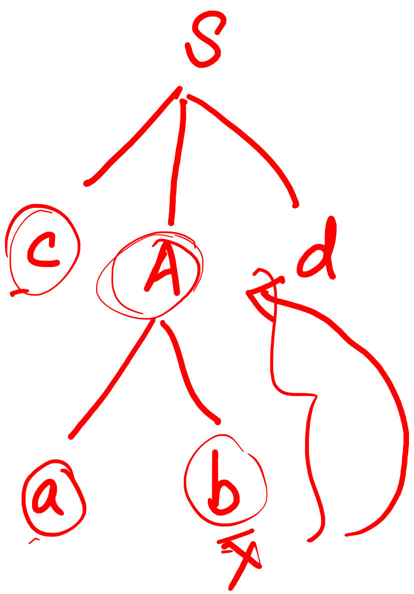
$S \to cAd \mid dC$

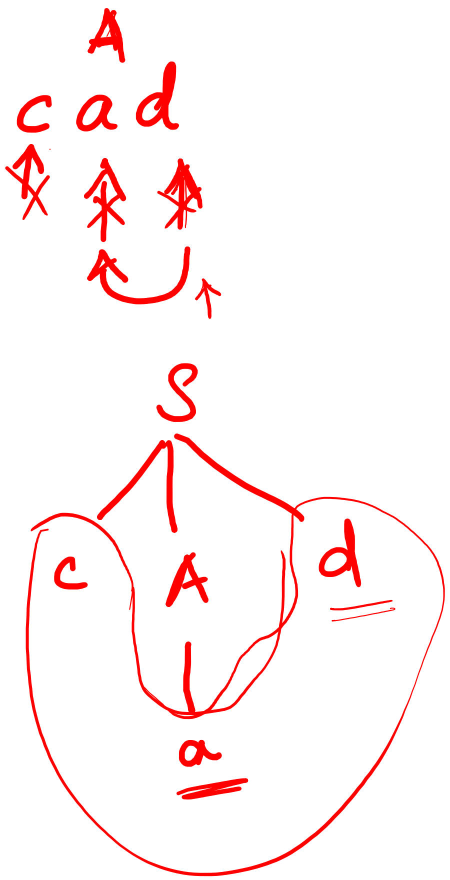
$A \to ab \mid C$

$C \to c \mid \epsilon$



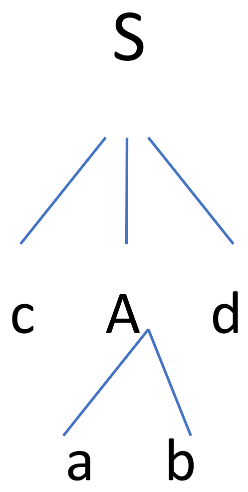


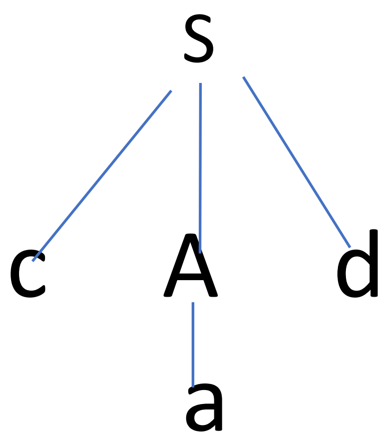




Parsing (Recursive Descent)

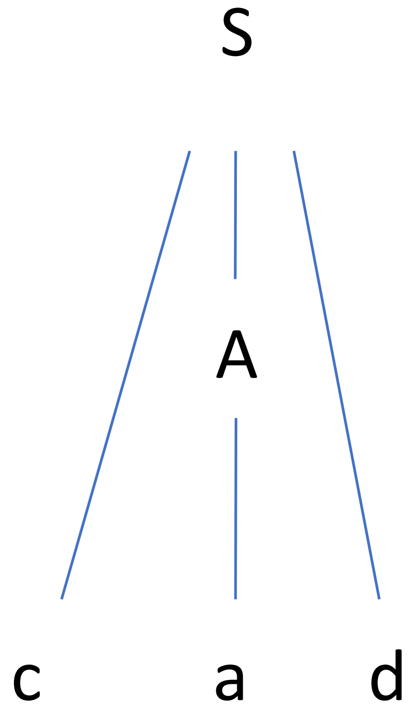
Expand A using the first alternative A → ab





Bottom up Parsers

- Start from the bottom and work up to the root for parsing a string
- LR parsers are bottom up parsers



Parsing

- Both Top Down and Bottom up parsers parse the string based on a viable-prefix property
- This property states that before the string the is fully processed, if there is an error, the parser will identify it and recovers

Context Free Grammars - CFG

- Programming language constructs are defined using context free grammar
- For example

  $E \rightarrow E + E \mid E * E \mid (E) \mid id$

  Expression grammar involving the operators, +, *, ( )

id + id * id

Context Free Grammars

- Defined formally as (V, T, P, S)

  V – Variables / Non-terminals

  T – Terminals that constitute the string

  P – Set of Productions that has a LHS and RHS

  S – Special Symbol, subset of V

$NT \rightarrow \alpha$

$A \rightarrow A\alpha$

$A \rightarrow B\beta$

$A \rightarrow A\alpha \mid B\beta$

Context Free Grammar

- Example

stmt → if E then stmt else stmt

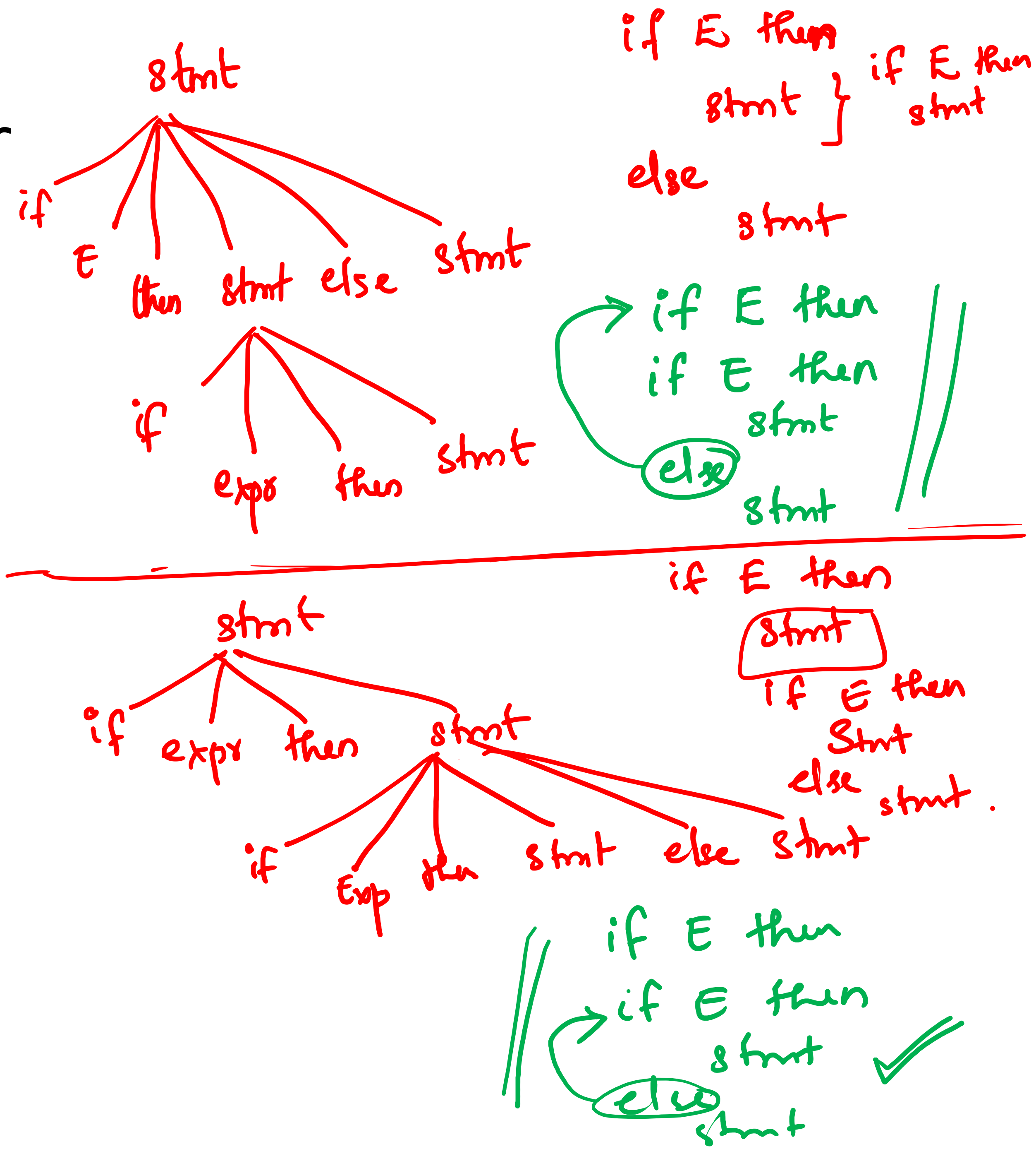
stmt → if E then stmt

stmt → a

E → b

Here, stmt, E are Non-terminals,

if, then, else, a, b are all terminals



Grammar - notations

- Terminals - $a,b,c,\ldots \in T$
  - specific terminals: **0**, **1**, **id**, **+**
- Non-terminals - $A,B,C,\ldots \in$
  - specific non-terminals: *expr*, *term*, *stmt*
- Grammar symbols - $X,Y,Z \in (N \cup T)$
- Strings of terminals

  $u,v,w,x,y,z \in T^*$
- Strings of grammar symbols

  $\alpha,\beta,\gamma \in (N \cup T)^*$

Derivation

- The *one-step derivation* is defined by

  $$\alpha\, A\, \beta \Rightarrow \alpha\, \gamma\, \beta$$

  where $A \to \gamma$ is a production in the grammar
- In addition, we define
  - $\Rightarrow$ is *leftmost* $\Rightarrow_{lm}$ if α does not contain a nonterminal
  - $\Rightarrow$ is *rightmost* $\Rightarrow_{rm}$ if β does not contain a nonterminal
  - Transitive closure $\Rightarrow^*$ (zero or more steps)
  - Positive closure $\Rightarrow^+$ (one or more steps)

Derivation

- The *language generated by G* is defined by

  $L(G) = \{w \mid S \Rightarrow^{+} w\}$

Derivation

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow - E$
$E \rightarrow \mathbf{id}$

$$E \Rightarrow - E \Rightarrow - \mathbf{id}$$

$$E \Rightarrow_{rm} E + E \Rightarrow_{rm} E + \mathbf{id} \Rightarrow_{rm} \mathbf{id} + \mathbf{id}$$

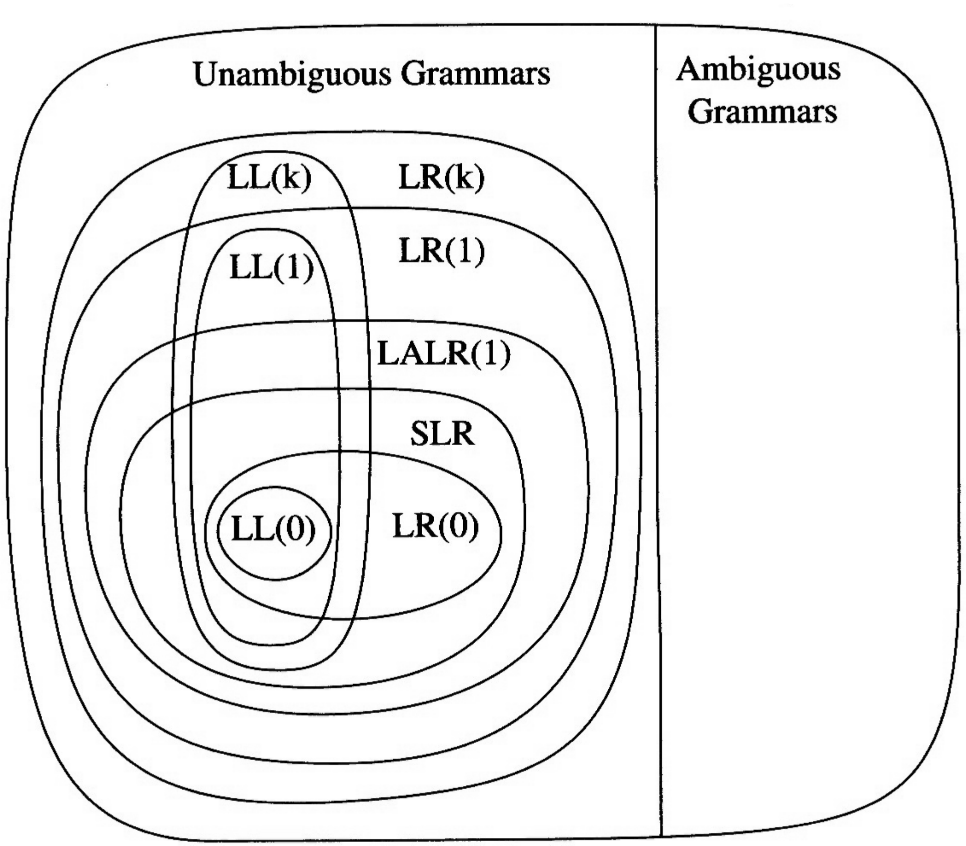
$$E \Rightarrow^{*} E \qquad E \Rightarrow^{+} \mathbf{id} * \mathbf{id} + \mathbf{id}$$

$$E \Rightarrow_{lm} E + E \Rightarrow id + E \Rightarrow id + id$$

Parsers

- Context Free grammars are already defined for all programming constructs
- All strings that are part of the programming language will be based on this construct
- Hence, parsers are designed keeping in mind the CFG

Hierarchy of Grammar Classes



Hierarchy

- **LL(*k*):**
— **L**eft-to-right, **L**eftmost derivation, *k* tokens lookahead
- **LR(*k*):**
— **L**eft-to-right, **R**ightmost derivation, *k* tokens lookahead
- **SLR:**
— **S**imple **LR** (uses “follow sets”)
- **LALR:**
— **L**ook**A**head **LR** (uses “lookahead sets”)

Top Down Parsers

- LL methods (Left-to-right, Leftmost derivation) and recursive-descent parsing

Grammar:

$E \to T \mathbf{+} T$

$T \to \mathbf{(}\ E\ \mathbf{)}$

$T \to \mathbf{-}\ E$

$T \to \mathbf{id}$

Leftmost derivation:

$$E \Rightarrow_{lm} T \mathbf{+} T$$
$$\Rightarrow_{lm} \mathbf{id} \mathbf{+} T$$
$$\Rightarrow_{lm} \mathbf{id} \mathbf{+} \mathbf{id}$$

Top Down Parsers – LL (1) Parsers

$\boxed{G} \to G' \to \boxed{G''}$

- LL parsers cannot handle
    - Left Recursive Grammar
    - Left Factoring

Left Recursive Grammar

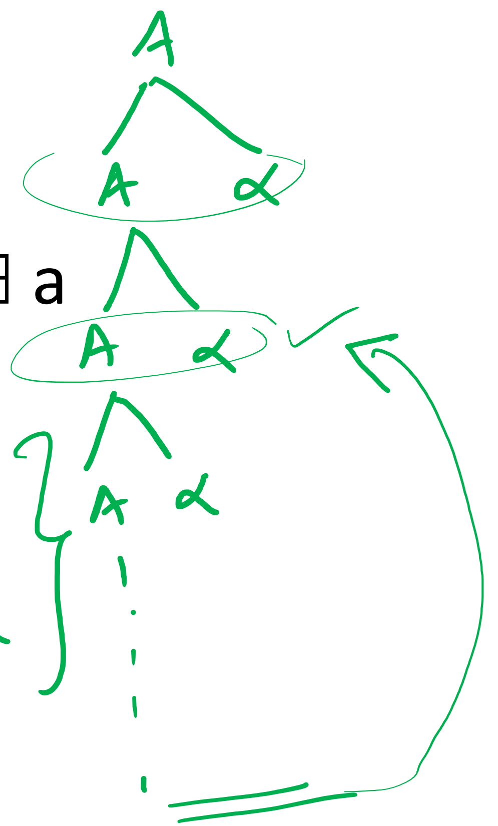
- *Formally, a grammar* is *left recursive* if $\exists\, A \in NT$ such that $\exists$ a derivation $A \Rightarrow^{+} A\alpha$, for some string $\alpha \in (NT \cup T)^{+}$
- $A \to A\,\alpha \,/\, \beta \mid \gamma$

$\boxed{A \to A\alpha}$

$A \to \beta$

$A \to \gamma$

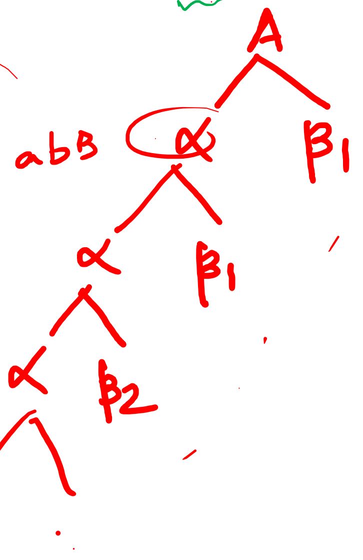
$$A \Rightarrow A\alpha \Rightarrow AA\alpha\alpha \Rightarrow AAA\alpha\alpha\alpha$$



Left Factor

- When a nonterminal has two or more productions whose right-hand sides start with the same grammar symbols the grammar is said to have left-factor property
- *Example* $A \rightarrow \alpha\, \beta_1 \,/\, \alpha\, \beta_2 \,/\, \ldots \mid \alpha\, \beta_n \mid \gamma$

$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$



$A \rightarrow \boxed{a}B \mid \boxed{a}S \mid cb$

$A \rightarrow \underbrace{abB}_{\alpha}\overbrace{Cb}^{\beta_1} \mid \underbrace{abB}_{\alpha}\overbrace{Sd}^{\beta_2} \mid cb$

$\boxed{ab\,B\,abB\,abB\,abB}\ \beta_1$

Pre-requisites for Top-Down Parser

- Eliminate Left Recursion
- Left Factor the grammar

$A \gamma \beta$

$A \rightarrow A\alpha \mid B\beta$

$B \rightarrow A\gamma \mid \varepsilon$

Eliminating Left Recursion

Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$
  for $i = 1, \ldots, n$ **do**
      for $j = 1, \ldots, i\text{-}1$ **do**
          replace each

$$A_i \to A_j \gamma$$

          with

$$A_i \to \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$$

          where

$$A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$$

      end
      eliminate the immediate left recursion in A_i
  end

Eliminate Left Recursion

- Rewrite every left-recursive production

  $$A \rightarrow A\,\alpha \;/\; \beta \mid \gamma \mid A\,\delta$$

- into a right-recursive production:

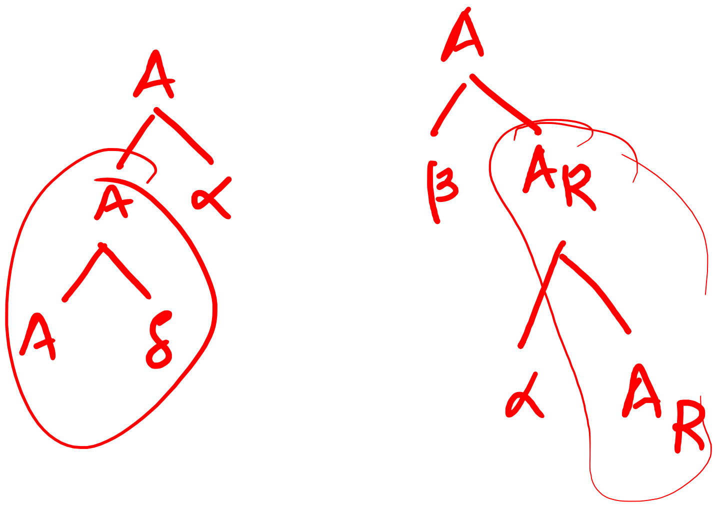
  $$A \rightarrow \beta\, A_R \;/\; \gamma\, A_R$$
  $$A_R \rightarrow \alpha\, A_R \;/\; \delta\, A_R \;/\; \varepsilon$$

$$A \rightarrow A\alpha \mid A\delta \mid \beta \mid \gamma$$

$$A \rightarrow \beta A_R \mid \gamma A_R$$

$$A_R \rightarrow \alpha A_R \mid \delta A_R \mid \varepsilon =$$

$$A \rightarrow \beta \mid \gamma \mid$$



$$A \rightarrow A\alpha$$
$$\Rightarrow AA\alpha\alpha$$
$$\Rightarrow AAA\alpha\alpha$$
$$\Rightarrow \cancel{AA}\beta\alpha\alpha$$

Example

- $A \rightarrow B\ C \mid \mathbf{a}$ ✓
  $B \rightarrow C\ A \mid A\ \mathbf{b}$ ✓
  $C \rightarrow A\ B \mid C\ C \mid \mathbf{a}$

i	j
1	
2	1
3	1 → 2

A - 1 B - 2 C - 3

① $A \rightarrow BC \mid a$ ✓

$B \rightarrow CA \mid Ab$

$B \rightarrow CA \mid BCb \mid ab$

② $B \rightarrow CAB_R \mid abB_R$ ✓

③ $B_R \rightarrow CbB_R \mid \varepsilon$

$C \rightarrow AB \mid CC \mid a$

$C \rightarrow BCB \mid aB \mid CC \mid a$

$C \rightarrow CAB_RCB \mid abB_RCB \mid aB \mid CC \mid a$

① ② ③ ④ ⑤

④ $C \rightarrow abB_RCBC_R \mid aBC_R \mid aC_R$

⑤ $C_R \rightarrow AB_RCBC_R \mid CC_R \mid \varepsilon$

- i = 1: nothing to do

  i = 2, j = 1: $B \rightarrow C\,A \mid \underline{A}\,\mathbf{b}$

  $\Rightarrow$ $B \rightarrow C\,A \mid \underline{B\,C}\,\mathbf{b} \mid \underline{\mathbf{a}}\,\mathbf{b}$

  $\Rightarrow_{\text{(imm)}}$ $B \rightarrow C\,A\,B_R \mid \mathbf{a}\,\mathbf{b}\,B_R$

  $B_R \rightarrow C\,\mathbf{b}\,B_R \mid \varepsilon$

  i = 3, j = 1: $C \rightarrow \underline{A}\,B \mid C\,C \mid \mathbf{a}$

  $\Rightarrow$ $C \rightarrow \underline{B\,C}\,B \mid \underline{\mathbf{a}}\,B \mid C\,C \mid \mathbf{a}$

- $i = 3, j = 2$: $C \rightarrow \underline{B}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow \quad C \rightarrow \underline{C\ A\ B_R}\ C\ B \mid \underline{\mathbf{a}\ \mathbf{b}\ B_R}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow_{\text{(imm)}} \quad C \rightarrow \mathbf{a}\ \mathbf{b}\ B_R\ C\ B\ C_R \mid \mathbf{a}\ B\ C_R \mid \mathbf{a}\ C_R$

$C_R \rightarrow A\ B_R\ C\ B\ C_R \mid C\ C_R \mid \varepsilon$

Example - Expression Grammar

G.

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

Modified Grammar

G'

- E → TE'
- E' → +TE' | ε
- T → FT'
- T' → *FT' | ε
- F→ (E) | id

Left Factoring

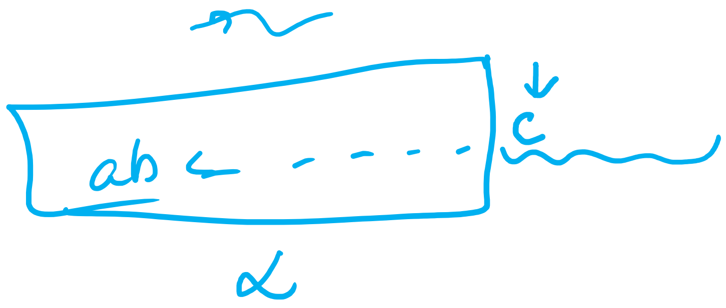
- Replace productions

$$A \rightarrow \alpha\,\beta_1 \,/\, \alpha\,\beta_2 \,/\, \ldots \mid \alpha\,\beta_n \mid \gamma$$

with

$$A \rightarrow \alpha\,A_R \mid \gamma$$

$$A_R \rightarrow \beta_1 \,/\, \beta_2 \,/\, \ldots \,/\, \beta_n$$



Left Factoring

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$
\begin{aligned}
A &\to \alpha A' \mid \gamma \\
A' &\to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n
\end{aligned}
$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. $\square$

Left Factoring - Example

$S \rightarrow$

$S \rightarrow iCtS \mid iCtSeS \mid a$

$eS \mid \epsilon$

$C \rightarrow b$

Left Factoring

- $S \rightarrow iCTSS' \mid a$
- $S' \rightarrow eS \mid \varepsilon$
- $C \rightarrow b$

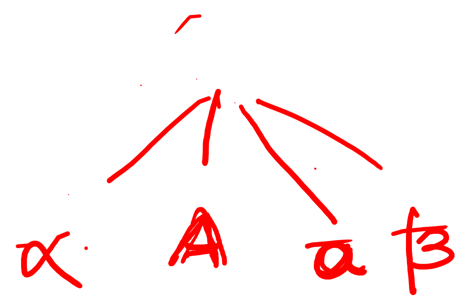
LL (1) Parser – Predictive parser

- L – input is scanned from left to right
- L – left derivation
- (1) – looking at 1 input symbol

Predictive Parser LL (1)

$A \rightarrow aBb \mid cb \mid BC$

- Eliminate left recursion from grammar
- Left factor the grammar
- Compute FIRST and FOLLOW
- Two variants:
  - Recursive (recursive calls)
  - Non-recursive (table-driven)



$\alpha A a \beta$

Recursive descent with Recursive calls

- Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input
- One procedure is associated with each nonterminal of a grammar
- A simple form of recursive-descent parsing, called predictive parsing, in which the lookahead symbol unambiguously determines the flow of control through the procedure body for each nonterminal
- The sequence of procedure calls during the analysis of an input string implicitly defines a parse tree for the input, and can be used to build an explicit parse tree, if desired.

void S() {

if 'c'

match('c')

call A()

if 'd' match('d')

$S \to \underline{c}Ad$

$A \to a \mid cd$

cad

match(t)

{

if(t==l)

l++;

else

error;

}

```
void A() {
    Choose an A-production, A → X1 X2 ··· Xk;
    for ( i = 1 to k ) {
        if ( Xi is a nonterminal )
            call procedure Xi();
        else if ( Xi equals the current input symbol a )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}
```

Recursive descent with Recursive calls

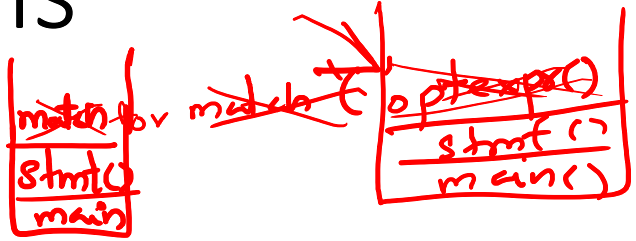
$\Rightarrow E+T$

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{expr} \; ; \\
& | & \textbf{if} \; (\; \textbf{expr} \;) \; stmt \\
& | & \textbf{for} \; (\; optexpr \; ; \; optexpr \; ; \; optexpr \;) \; stmt \\
& | & \textbf{other} \\
\\
optexpr & \rightarrow & \epsilon \; ✓ \\
& | & \textbf{expr} \; ✓
\end{array}
$$

for (i=0 ; i<n ; i++)

for (; i<n ; i++)

Recursive descent with Recursive calls



for(expr ;expr;)

```
void stmt() {
    switch ( lookahead ) {
    case expr:
        match(expr); match(';'); break;
    case if:
        match(if); match('('); match(expr); match(')'); stmt();
        break;
    case for:
        match(for); match('(');
        optexpr(); match(';'); optexpr(); match(';'); optexpr();
        match(')'); stmt(); break;
    case other:
        match(other); break;
    default:
        report("syntax error");
    }
}
```

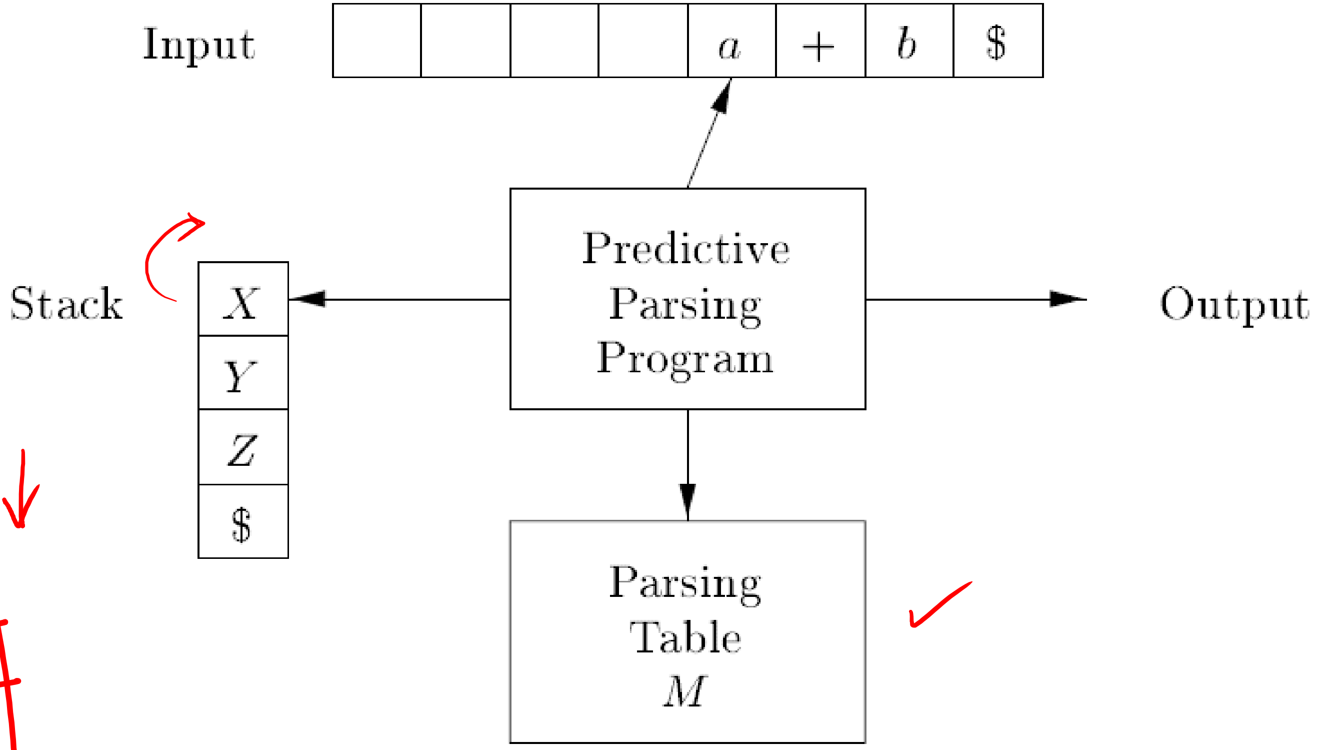
Recursive descent with Recursive calls

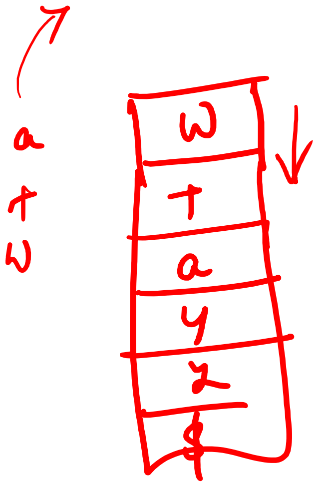
```
void optexpr() {
        if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
        if ( lookahead == t ) lookahead = nextTerminal;
        else report("syntax error");
}
```

Non-Recursive Predictive Parsing

$(X,a) \rightarrow X \rightarrow WTa$





FIRST ()

- FIRST function is computed for all terminals and non-terminals
- FIRST(α) = the set of terminals that begin all strings derived from α

FIRST(a)

FIRST (A)

FIRST (aAB)

FIRST ( )

- $\text{FIRST}(a) = \{a\}$ if $a \in T$
  $\text{FIRST}(\varepsilon) = \{\varepsilon\}$
  $\text{FIRST}(A) = \cup_{A\to\alpha} \text{FIRST}(\alpha)$
  for $A\to\alpha \in P$

$A \to \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid$

$\text{FIRST}(A) = \text{FIRST}(\alpha_1) \cup \text{FIRST}(\alpha_2) \cup \text{FIRST}(\alpha_3)$

FIRST () – Algorithm

- FIRST($X_1X_2…X_k$) =
  if for all j = 1, …, i-1 : $\varepsilon \in$ FIRST(X_j) **then**
  add non-ε in FIRST(X_i) to FIRST($X_1X_2…X_k$)
  if for all j = 1, …, k : $\varepsilon \in$ FIRST(X_j) **then**
  add ε to FIRST($X_1X_2…X_k$)

bcd

cd

d

ε

X_1 X_2 X_3

$A \to BCD$

$B \to b \mid \varepsilon$

$C \to c \mid \varepsilon$

$D \to d \mid \varepsilon$

FIRST(A) = FIRST(BCD)

= {b, c, d, ε}

FIRST(B) = {b, ε}

FIRST(C) = {c, ε}

FIRST(D) = {d

FOLLOW

- FOLLOW(A) = the set of terminals that can immediately follow non-terminal A

FOLLOW - Algorithm

- FOLLOW(A) =

  if A is the start symbol S **then**

  add **$** to FOLLOW($A$)

  for all $(B \to \alpha\ A\ \beta) \in P$ **do**

  add FIRST(β)\{ε} to FOLLOW(A) ✓

  for all $(B \to \alpha\ A\ \beta) \in P$ and $\varepsilon \in$ FIRST(β) **do**

  add FOLLOW(B) to FOLLOW(A)

  for all $(B \to \alpha\ A) \in P$ **do**

  add FOLLOW(B) to FOLLOW(A)

$\stackrel{*}{\Rightarrow} \ldots cBd \ldots$

$\Rightarrow \ldots c\alpha Ad \ldots$

$\Rightarrow \ldots c\alpha A\beta d \ldots$

$c\alpha Ad$

ε

$B \to \alpha A\beta$

$B \to \alpha A$

Example

- E → TE’
- E’ → +TE’ | ε
- T → FT’
- T’ → *FT’ | ε
- F→ (E) | id

FI(+) = {+} FI(*) = {*} FI(() = {(}

FI()) = {)} FI(id) = {id}

FI(E) = FI(T) = {(, id}

FI(T) = FI(F) = {(, id}

FI(F) = {(, id}

FI(E') = {+, ε}

FI(T') = {*, ε}

FO(E) = {), $ }

FO(T) = FI(E') ∪ FO(E) ∪ FO(E') = { +,), $ }

FO(F) = FI(T') ∪ FO(T) ∪ FO(T') = { *, +,), $ }

FO(E') = FO(E) = {), $ }

FO(T') = { +,), $ }

(id+id)

id+id $

FIRST

- FIRST (E) = FIRST (T) = FIRST(F)

  = {(, id}
- FIRST (E’) = { +, ε}
- FIRST (T’) = { *, ε}

FOLLOW

- FOLLOW(E) = FIRST(‘)’)

  ={$,) }
- FOLLOW(T) = FIRST(E’) U FOLLOW(E)

  ={+, $,) }
- FOLLOW(F) = { *, +, $,) }

  FIRST(T’) U FOLLOW(T)

- FOLLOW (E’) = FOLLOW (E)

  = {$,) }
- FOLLOW (T’) = FOLLOW (T)

  = {$, +,) }

Another Example

- Ambiguous grammar
  $S \rightarrow \overset{\alpha}{\mathbf{i}}\, \overset{A}{C}\, \mathbf{t}\, S\, S' \mid \mathbf{a}$
  $S' \rightarrow \mathbf{e}\, S \mid \varepsilon$
  $C \rightarrow \mathbf{b}$

$FO(S) = \{\$, e\}$

$FO(S') = \{\$, e\}$

$FO(C) = \{t\}$

$FI(C) = \{b\}$

$FI(S') = \{e, \varepsilon\}$

$FI(S) = \{i, a\}$

$R \rightarrow (T) \mid c$

$T \rightarrow T, R \mid R$

$R \rightarrow (T) \mid c$

$T \rightarrow RT'$

$T' \rightarrow , RT' \mid \varepsilon$

$FI(R) = \{(, c\}$

$FI(T) = \{c, (\}$

$FI(T') = \{\varepsilon, ,\}$

$FO(R) = \{\$, ,,)\}$

$FO(T) = \{)\}$

$FO(T') = \{)\}$

- First (S) = {i,a}
- First (S’) = {e, ε }
- First (C) = {b}
- Follow (S) ={$, e}
- Follow(S’) = {$, e}

Predictive Parsing Table

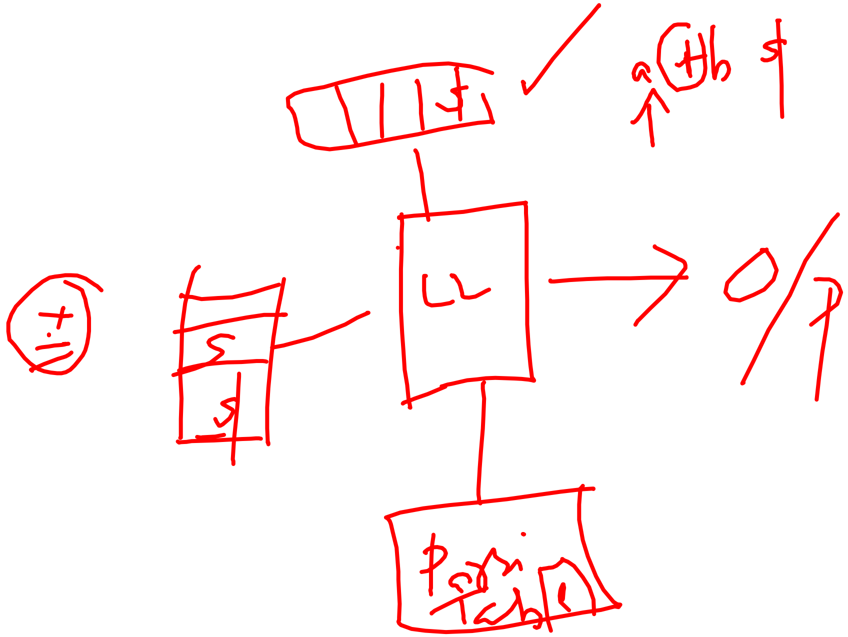
- Row for each non-terminal
- Column for each terminal symbol

  Table[NT, symbol] = Production that matches the [NT, symbol]

  if First(NT) has ε, then add production

  NT → ε in all [NT, a] for all ‘a’ in FOLLOW(NT)

Parsing action



- push(**$**)
  push(S)
  a := *lookahead*
- **repeat**
  X := pop()
  if X is a terminal or X = **$** **then**
  match(X) // move to next token, a := *lookahead*
  else if $M[X,a] = X \rightarrow Y_1Y_2\ldots Y_k$ **then**
  push($Y_k, Y_{k-1}, \ldots, Y_2, Y_1$) // such that Y_1 is on top
  produce output and/or invoke actions
  else error()
  endif
  until X = **$**

Parsing action Example

$E \to TE'$

Stack	Input String	Action
$E	id + id* id $	[E, id]
$E'T	id + id *id $	[T, id] $E \to TE'$
$E'T'F	id + id *id $	[F, id] $T \to FT'$
$E'T'id	id + id *id $	id, id -> pop stack and move input
$E'T'	+ id *id$	[T', +] -> replace with ε
$E'	+ id *id$	[E', +]
$E'T+	+ id *id$	+, + → pop stack and move
$E'T	id * id $	[T, id]
$E'T'F	id *id$	[F, id]

$E id*$
$E'T id*$
$E'T'F id*$
$E'T' *$
$E'T'F* *$
$E'T'F $
Error

Stack	Input String	Action
$E’T’id	id *id$	id, id –> pop
$E’T’	*id $	[T’, *]
$E’T’F*	*id $	*,* -> pop, and move
$E’T’F	id$	[F, id]
$E’T’id	id $	id, id → pop
$E’T’	$	T’, $ -> replace with ε
$E’	$	E’, $ -> replace with ε
$	$	Accept

Error Recovery in LL (1) parser

- *Panic mode*
  - Discard input until a token in a set of designated synchronizing tokens is found
- *Phrase-level recovery*
  - Perform local correction on the input to repair the error
- *Error productions*
  - Augment grammar with productions for erroneous constructs
- *Global correction*
  - Choose a minimal sequence of changes to obtain a global least-cost correction

Error Recovery

- Panic Mode
  - Add synchronizing actions to undefined entries based on FOLLOW
- Phrase Mode
  - Change input stream by inserting missing +, *, (, or)
    For example: **id id** is changed into **id * id or id + id**

Error Recovery

- Error Production
  - Add productions that will take care of incorrect input combinations

Error Recovery

	id	+	*	(	)	$
E	E → TE’			E → TE’	synch	synch
E’		E’→ +TE’			E’ → ε	E’ → ε
T	T → FT’	synch		T → FT’	synch	synch
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε
F	F → id	synch	synch	F → (E)	synch	synch

LL (1)

- A grammar G is LL(1) if for each collections of productions
  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
  for nonterminal A the following holds:

  1. FIRST(α_i) $\cap$ FIRST(α_j) = $\emptyset$ for all $i \neq j$
  2. if $\alpha_i \Rightarrow^* \varepsilon$ then
     2.a. $\alpha_j \Rightarrow^* \varepsilon$ for all $i \neq j$
     2.b. FIRST(α_j) $\cap$ FOLLOW(A) = $\emptyset$
     for all $i \neq j$

If then Grammar

- The if then grammar has multiple entries in the parsing table.
- So, confusion on which production to apply
- Ambiguous grammar hence not LL (1)