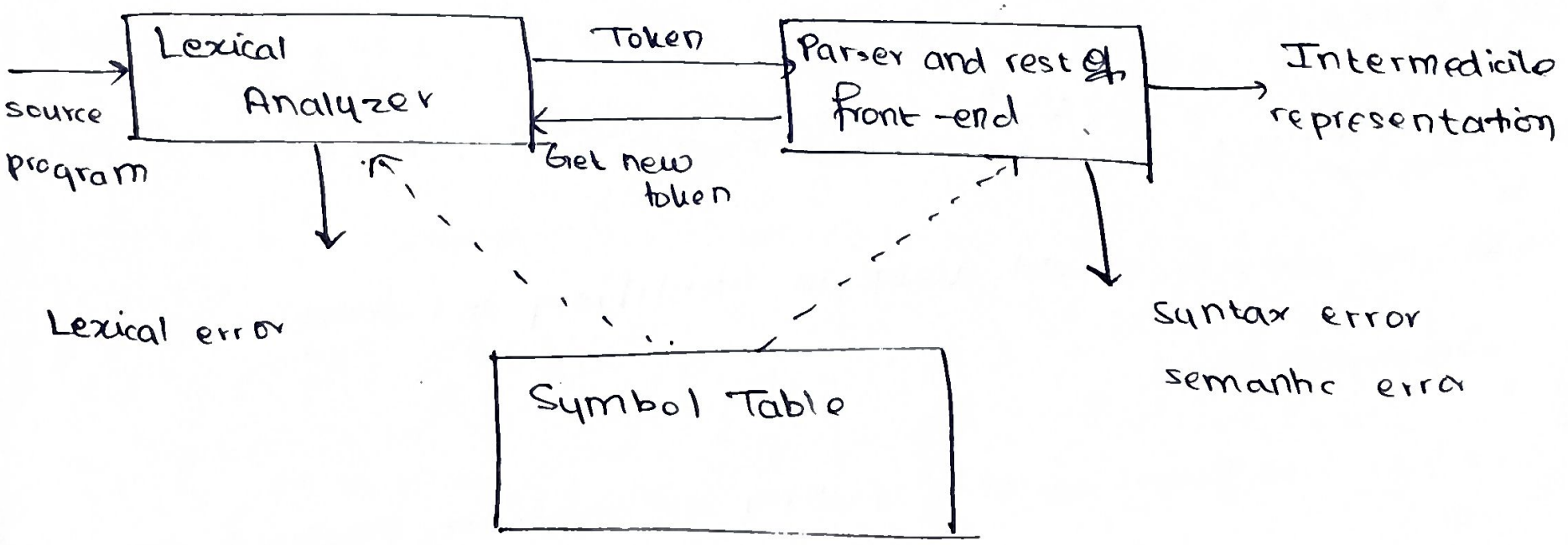
UCS2702 COMPILER DESIGN

Unit2

Syntax Analysis

Role of parser - writing grammars for language constructs - Types of grammar : Ambiguity - Deterministic & Recursive ; Top down parsers : Recursive decent parser - predictive parser ; Bottom up parsers : SLR Parser - CLR Parser - LALR Parser ; Error handling & recovery in syntax analyzer ; Syntax analyzer generator : Structure of yacc program - creating yacc lexical analyzers with lex

* Role of Parser



* Syntax Analyzer (also called parser)

→ Creates the syntactic structure of the given source program

→ The syntactic structure is mostly a parse tree

→ The syntax of a programming lang. is described by a CFG

→ The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a CFG or not

(i) If it satisfies → the parser creates the parse tree of that program

(ii) Else, gives an error message

* Kinds of Parsers

1. Top-Down Parser - the parse tree is created top to bottom, starting from the root

2. Bottom-Up Parser - the parse is created bottom to top, starting from the leaves.

→ Both parsers scan the input from the left to right (one symbol at a time).

* Error Handling

→ A good compiler should assist in identifying and locating errors.

→ Errors can be:

(i) Lexical Errors - misspelling an identifier, keyword or operator

(ii) Syntax Errors - such as an arithmetic expression with unbalanced ~~expressi~~ parantheses

(iii) Semantic Errors - such as an operator applied to an incompatible operand

(iv) Logical Errors : such as an indefinitely recursive call.

* Error Recovery Strategies

(i) Panic Mode - Discard input until a token in a set of designated synchronizing tokens is found

(ii) Phase-Level Recovery - Perform local correction on the input to repair the error

(iii) Error productions - Augment grammar with productions for erroneous constructs

(iv) Global Correction - choose a minimal sequence of changes to obtain a global least-cost correction

* Grammars

→ Inherently recursive structures of a programming language are defined by a context-free grammar, as a 4-tuple

$$G = (N, T, P, S)$$

N → non-terminals
T → terminals
P → products
S → start symbol

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

Derivation → A sequence of replacements of non-terminal symbols is called a derivation

$\Rightarrow$: derives in one step

$\overset{*}{\Rightarrow}$: derives in zero or more steps

$\overset{+}{\Rightarrow}$: derives in one or more steps

(i) If we always choose the left-most non-terminal in each derivation step, this derivation is called as the left-most derivation

(ii) If we always choose the right-most non-terminal in each derivation step, this derivation is called the right-most derivation

Example : Consider the grammar :

$E \rightarrow E+E$

$E \rightarrow E-E$

$E \rightarrow E*E$

$E \rightarrow E/E$

$E \rightarrow -E$

$E \rightarrow (E)$

$E \rightarrow id$

Use LMD & RMD for $-(id+id)$

LMD

$E \rightarrow -E$

$\Rightarrow -(E)$

$\Rightarrow -(E+E)$

$\Rightarrow -(id+E)$

$\Rightarrow -(id+id)$

RMD

$E \Rightarrow -E$

$\Rightarrow -(E)$

$\Rightarrow -(E+E)$

$\Rightarrow -(E+id)$

$\Rightarrow -(\overset{id}{E}+id)$

* Constructing CFGs from Regular Expressions

Algorithm

For each state i of the NFA, create a non-terminal Ai

Begin

(i) If state i has a transition to j on symbol a,

introduce production $A_i \rightarrow aA_j$

(ii) If state i goes to state j on ϵ,

introduce production $A_i \rightarrow A_j$

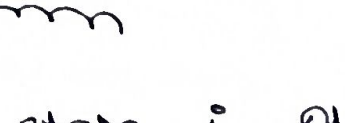
End

If i is an accepting state, introduce $A_i \rightarrow \epsilon$

If i is the start state, make Ai the start symbol for the grammar.

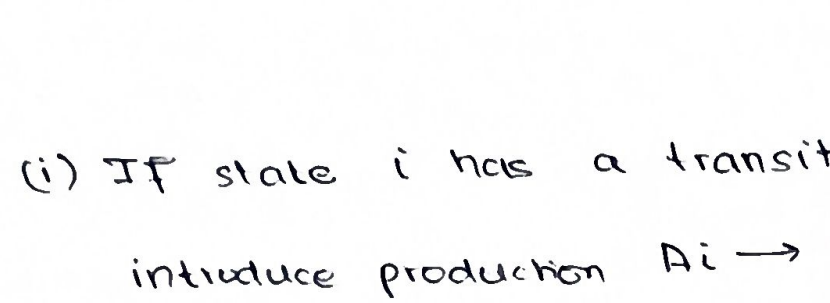
RE - NFA Rules from TOC (Thompson's Construction)

a

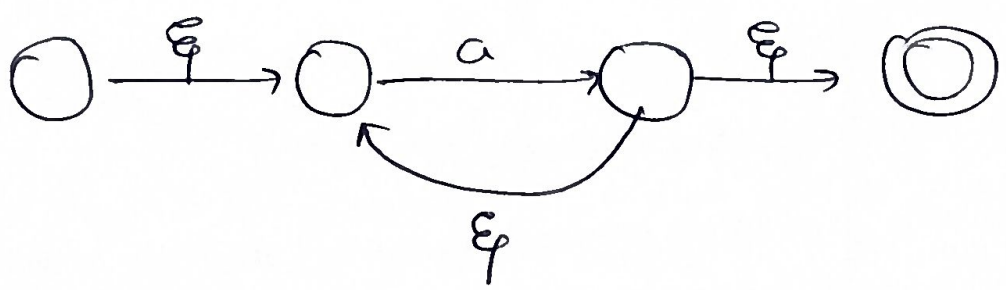


ab

a|b

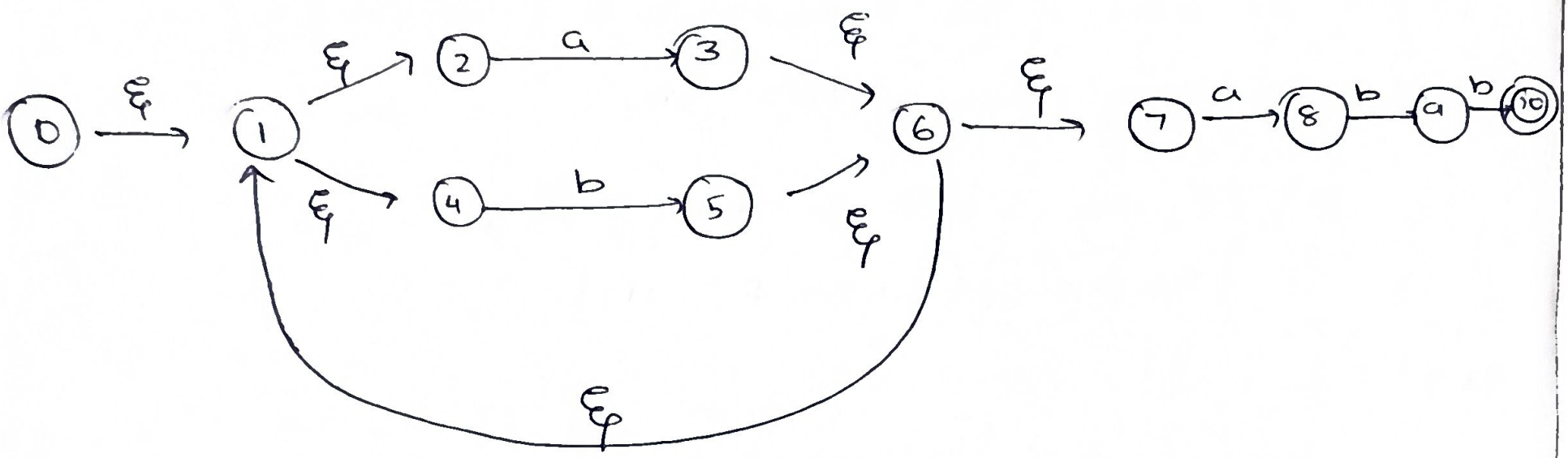


a*



Example Construct a grammar for the following regular expressions.

(1) $(a|b)^* abb$



$A_0 \rightarrow A_1$

$A_1 \rightarrow A_2 \mid A_4$

$A_2 \rightarrow aA_3$

$A_4 \rightarrow bA_5$

$A_3 \rightarrow A_6$

$A_5 \rightarrow A_6$

$A_6 \rightarrow A_7 \mid A_1$

$A_7 \rightarrow aA_8$

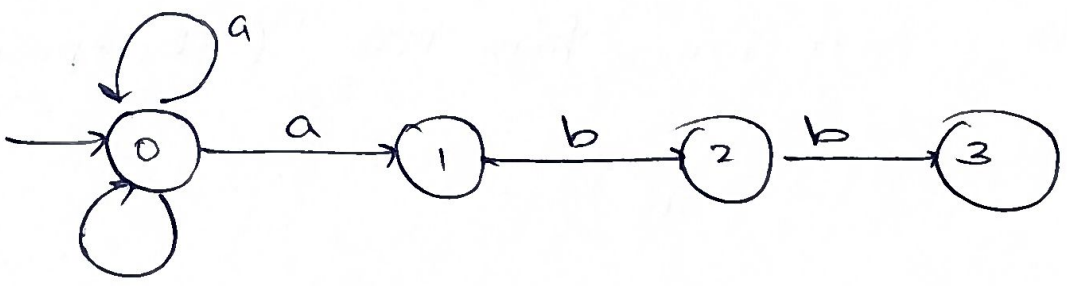
$A_8 \rightarrow bA_9$

$A_9 \rightarrow bA_{10}$

$A_{10} \rightarrow \varepsilon$

But for simplicity

DO THIS



$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \varepsilon$

A_0 = start state

A_3 = accepting state

* Ambiguity

→ If more than one parse tree can be constructed for a sentence, then the grammar is called ambiguous.

Example: Check if the following grammar is ambiguous.

$E \rightarrow E+E$
$E \rightarrow E-E$
$E \rightarrow E*E$
$E \rightarrow E/E$
$E \rightarrow -E$
$E \rightarrow (E)$
$E \rightarrow id$

for using the sentence

id + id * id

$E \rightarrow E+E$

$\Rightarrow E+E*E$

$\Rightarrow id+E*E$

$\Rightarrow id+id*E$

$\Rightarrow id+id*id$

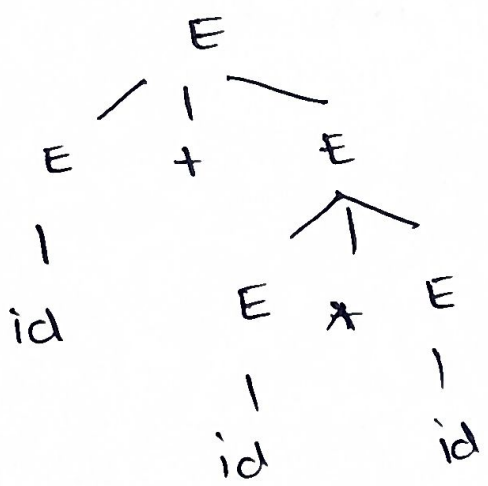
$E \Rightarrow E*E$

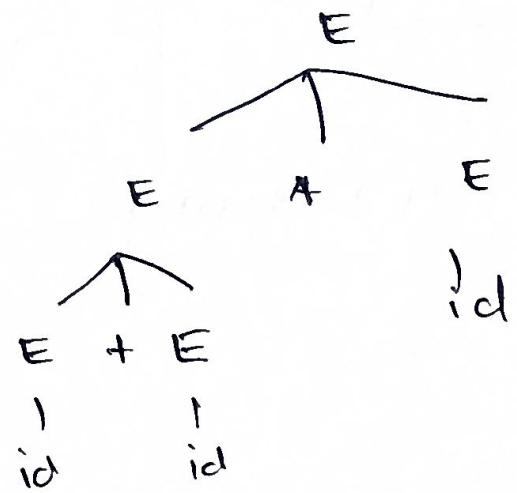
$\Rightarrow E+E*E$

$\Rightarrow id+E*E$

$\Rightarrow id+id*E$

$\Rightarrow id+id*id$





Two different parse trees ⇒ ambiguous

→ For most parsers, the grammar must be unambiguous, such that there is a unique selection in the parse tree for a sentence.

→ Must prefer one of the parse trees of a sentence to disambiguate the grammar, and restrict to this choice.

Example Consider the following CFG.

S → aABe
A → Abc | b
B → d

Parse abbcde using LMD & RMD

LMD

S → aABe
⇒ ~~aAbcBe~~ ⇒ aAde
⇒ aAbcde
~~⇒ aabcBe~~
⇒ abbcde

RMD

S → aABe
⇒ aAbcBe
⇒ abbcBe
⇒ abbcde

* Left Recursion and Left Factoring?

* Types of Grammar

1. Ambiguous Grammar
2. Unambiguous Grammar
3. Recursive Grammar
4. Non-deterministic grammar
5. Deterministic Grammar

* Left Recursion

→ A grammar is left-recursive if it has a non-terminal A such that there is a derivation

$$A \Rightarrow A\alpha$$

→ Top-down parsing techniques cannot handle left-recursive grammars

→ The left-recursion may appear in a single step of the derivation (immediate left-recursion), or may appear in more than one-step of the derivation

Handling left Recursion (Immediate)

$$A \to A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \ldots \beta_n$$

Then (βs do not start with A)

$$A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \beta_n A'$$

$$A' \to \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

Example 1 Eliminate left-recursion

$$E \to E+T \mid T$$

$$T \to T*F \mid F$$

$$F \to id \mid (E)$$

Ans LR in $E \to E+T$ and $T \to T*F$

The grammar becomes:

$E \rightarrow \overset{A}{E+T} \mid \overset{\beta}{T}$ & $T \rightarrow T*F \mid F$

Anc w/o LR :

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id \mid (E)$

* Removing Left Recursion that is not immediate

→ A grammar may not be immediately left recursive, but it can still be left-recursive

eg. $S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$

Not immediately left-recursive,

but $S \Rightarrow Aa \Rightarrow Sca \Rightarrow$ left recursion

∴ Need to eliminate all left recursion from the grammar

Algorithm

Input : Grammar G with no cycles or ε-productions

Arrange the non-terminals in some order $A_1, A_2, \ldots A_n$

for $i = 1 \ldots n$ do

  for $j = 1 \ldots n$ do

    replace each

      $A_i \rightarrow A_i\gamma$

with

$$A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \delta_k\gamma$$

where

$$A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \delta_k$$

end do

eliminate the immediate left recursion in A_i.

end do

Example 1 Remove left recursion in:

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid f$$

Assume that the ordering of non-terminals is S, A

1. $S \rightarrow Aa \mid b$ → no LR

2. $A \rightarrow Ac \mid Sd \mid f$

⇓

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Aad \mid \underset{\beta_1}{bd} \mid \underset{\beta_2}{f}$

⇓

Ans

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid fA'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Example

$S \to Aa \mid b$

$A \to Ac \mid Sd \mid f$

Assume that the ordering of non-terminals is A, S

Ans

1. $S \to Aa \mid b$
2. $A \to Ac \mid Sd \mid f$ $\to$ LR

$\Downarrow$

For A

1. $S \to Aa \mid b$
   $A \to SdA' \mid fA'$
   $A' \to cA' \mid \epsilon$

Ans

1. $S \to Aa \mid b$

$\to$ LR

$\Downarrow$

1. $S \to bA'$
2. $A' \to aA'$
3. $A \to Ac \mid Sd \mid f$

2. $S \to bA'$
   $A' \to aA'$
   $A \to Ac \mid bA'd \mid f\epsilon$

For S

$S \to SdA'a \mid fA'a \mid b$ (B1, B2) $\to$ LR

$\Downarrow$

$S \to fA'aS' \mid bS'$
$S' \to dA'aS' \mid \epsilon$

∴ The final grammar becomes:

$S \to fA'aS' \mid bS'$

$S' \to dA'aS' \mid \epsilon$

$A \to SdA' \mid fA'$

$A' \to cA' \mid \epsilon$

Example 3 Remove LR

$A \rightarrow Abd \mid Aa \mid a$

$B \rightarrow Be \mid b$

1. $A \rightarrow aA'$
2. $A' \rightarrow bdA' \mid aA' \mid \varepsilon$
3. $B \rightarrow bB'$
4. $B \rightarrow eB' \mid \varepsilon$

Example 4 Remove RR

$S \rightarrow (L) \mid a$

$L \rightarrow ,S \mid S$

1. $S \rightarrow (L) \mid a$

   $L \rightarrow SL'$

   $L' \rightarrow ,SL' \mid \varepsilon$

* Non-Deterministic Grammar

→ Grammar with a common prefix between at least two different productions coming from the same LHS.

eg. $S \rightarrow \underline{a}Sb \mid \underline{a}A \mid b$

eg. $S \rightarrow \underline{ab} \mid \underline{ab}A$

$A \rightarrow \underline{bab} \mid \underline{bab}c$

Disadvantage: During parsing - needs a lot of back tracking (time consuming)

Deterministic Grammar → Grammar without any common prefix in any of the different productions from the same LHS

☆☆☆ To make the grammar suitable for predictive or top-down parsing, we need to convert the non-deterministic grammar into deterministic grammar using Left Factoring

* Left Factoring

→ A predictive parser (a top-down parser without backtracking) insists that the grammar must be left-factored.

If

stmt → if expr then sten. | if expr the stmt else stmt

When we see if, we would not know which production rule to choose.

To left factor :

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ (which α to choose?)

rewrite as β_1 & β_2 are different

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$

Algorithm

$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \ldots \mid \gamma_m$

convert to $A \rightarrow \alpha A' \mid \gamma_1 \mid \gamma_2 \ldots \mid \gamma_m$ & $A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$

Example 1 left - facto :

$A \rightarrow abB \mid aB \mid cdg \mid caeB \mid cafB$

Ans $A \rightarrow aA'$

$A' \rightarrow bB \mid B$

$A \rightarrow cdA''$

$A'' \rightarrow g \mid eB \mid fB$

Example 2 left - facto :

$A \rightarrow aa \mid a \mid ab \mid abc \mid b$

Ans $A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

⇓

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$

Example 3 Left Factor : $S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

Ans $S \rightarrow iEtSS' \mid a$

$S' \rightarrow \varepsilon \mid eS$

$E \rightarrow b$

Example : left factor :

$S \rightarrow aSSbS \mid aSaSb \mid abb \mid b$

Ans $S \rightarrow aS' \mid b$

$S' \rightarrow SSbS \mid SaSb \mid bb$

$S \rightarrow aS' \mid b$

$S' \rightarrow SS'' \mid bb$

$S'' \rightarrow SbS \mid aSb$

* Top-Down Parser → The parse tree is created top to bottom

→ can be : (i) recursive decent parsing

(ii) predictive parsing

Difficulties with Top Down Parsing

(i) Left Recursion

(ii) Backtracking

(iii) Selection of Alternatives

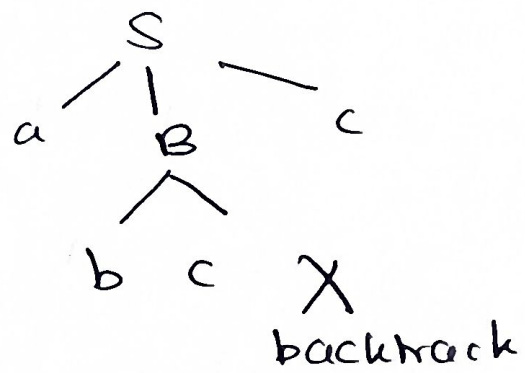
(iv) Error Reporting

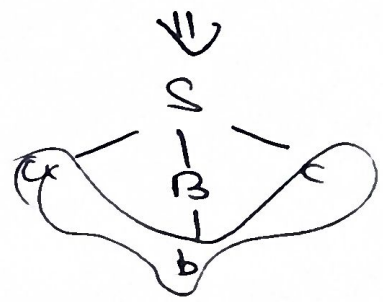
→ Consider the grammar :

$S \rightarrow aBc$

$B \rightarrow bc \mid b$

For abc





A. **Recursive Decent Parsing**

→ Backtracking is needed - if choice of production rule doesn't work, backtrack to other alternatives

→ A general parsing technique, but not used widely

→ Not efficient

B. **Predictive Parsing**

→ no backtracking needed

→ efficient

→ needs a special form of grammar called LL(1) grammar

→ Recursive Predictive Parsing is like (A) but no backtracking, also called the LL(1) parser.

Example1 : Consider the following grammar:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid id$$

Construct functions for a recursive descent parser and parse the following:

(i) id + id * id

(ii) id - id

Ans

① Procedure E()

```
Begin
    T()
    EPrime()
End
```

② Procedure EPrime()

```
If input symbol = '+' then
Begin
    Advance()
    T()
    EPrime()
End
```

③ Procedure T()

```
Begin
    F()
    TPrime()
End
```

④ Procedure TPrime()

```
If input symbol = '*' then
Begin
    Advance()
    F()
    TPrime()
End
```

⑤ Procedure F()

```
If input symbol = 'id' then
    Advance()
else if inputsymbol = '(' then
begin
    Advance()
    E()
    if inputsymbol = ')' then
        Advance()
    else
        Error()
end
else
    error();
```

(i) id + id

Function call

Input

id + id.

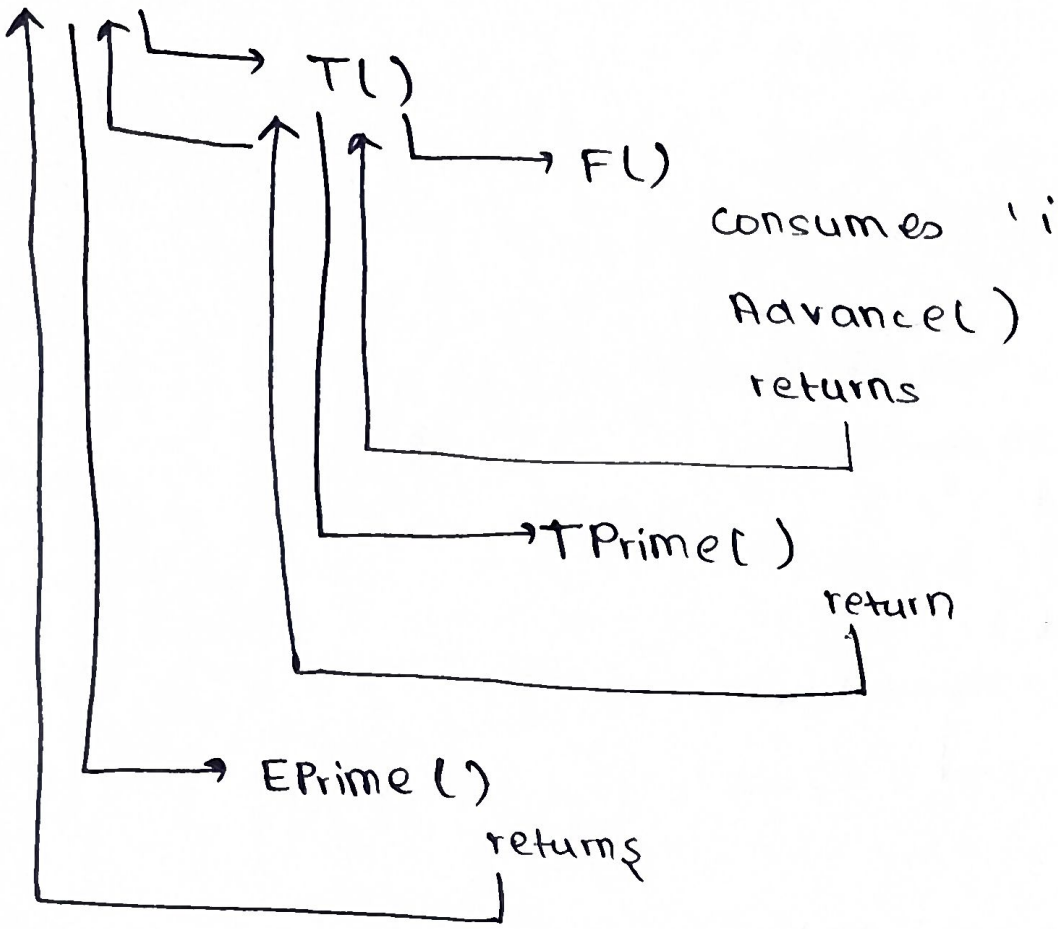
1. E()
   - → T()
     - → F()
       - → consumes the first id
       - Advance()
     - → TPrime()
       - nothing happens
   - → EPrime()
     - → consumes '+'
     - → Advance()
     - → T()
       - → F()
         - → consumes id
         - Advance()
     - → EPrime()
       - nothing returns
   - → Terminates (Exit)
     - → string successfully parsed

id + id

id + id

id + id

(ii) id - id



id - id
↑

can try

no valid parse

(iii) id + id * id

* Predictive Parser

Steps

1. Eliminate left-recursion if any
2. Compute FIRST
3. Compute FOLLOW
4. Construct parsing table
5. Use parsing algo to check if syntactically correct or not

COMPUTE FIRST Start from bottom

(i) X is terminal - FIRST(X) = X

(ii) X is a NT , $X \to \varepsilon$ FIRST(X) = ε

(iii) X is a NT , $X \to ABCD$ FIRST(X) = FIRST(A)

(see first symbol)

if FIRST(A) = ε,

find FIRST(B) and so on

if all FIRST(A_i) = ε FIRST(X) = ε

COMPUTE FOLLOW → For non-terminals (Start from top) start symbol

(i) If start symbol, X FOLLOW(X) has the symbol $ too.

(ii) IF $A \to \alpha \underline{B\,\beta}$ (B is squished between 2 characters)

then FOLLOW(B) = FIRST(β)

(no-ε)

If ε, then add FOLLOW(A)

(iii) If $A \to \alpha B$ FOLLOW(B) = FOLLOW(A)

CONSTRUCT PARSE TREE

Rows = NT

Columns = T + $

For each production rule $A \to \alpha$

For each terminal a in FIRST(α) Add $A \to \alpha$ to M[A, a]

If ε in FIRST(α), for each terminal a in FOLLOW(A)

add $A \to \alpha$ to M[A, a]

If ε in FIRST(α) and $ in FOLLOW(A)

Add $A \to \alpha$ to M[A, $]

Example1 : Construct a predict parser for the following grammar.

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

① FIRST(E) = FIRST(T) = FIRST(F) = { (, id }

FIRST(E') = { +, ε }

FIRST(T') = { *, ε }

② FOLLOW(E) = { $,) }

FOLLOW(T) = { +, $,) }

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

FOLLOW(E') = { $,) }

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

③ Construction of Parsing Table

NT	+	*	(	)	id	$
E			$E \to TE'$		$E \to TE'$	
E'	$E' \to +TE'$			$E' \to \varepsilon$		$E' \to \varepsilon$
T			$T \to FT'$		$T \to FT'$	
T'	$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$		$T' \to \varepsilon$
F			$F \to (E)$		$F \to id$	

NT on top of stack & input
⇒ see table
T on stack ⇒ match w/ input & ppcut

(4) Parse input string $w = id * id + id$

STACK	INPUT	OUTPUT
$\$E$	$id * id + id\$$	$E \to TE'$
$\$E'T$	$id * id + id\$$	$T \to FT'$
$\$E'T'F$	$id * id + id\$$	$F \to id$
$\$E'T'\underline{id}$	$\underline{id} * id + id\$$	
$\$E'T'$	$* id + id\$$	$T' \to *FT'$
$\$E'T'F*$	$\underline{*} id + id\$$	
$\$E'T'F$	$id + id\$$	$F \to id$
$\$E'T'\underline{id}$	$\underline{id} + id\$$	
$\$E'T'$	$+ id\$$	$T' \to \varepsilon$
$\$E'$	$+ id\$$	$E' \to +TE'$

$ E' T +	+ id $	
$ E' T	id $	T → F T'
$ E' T' F	id $	F → id.
$ E' T' id	id $	
$ E' T'	$	T' → ε
$ E'	$	E' → ε
$	$	

stack & input are empty ⟶ parse is valid

Example 2 : Compute FIRST & FOLLOW for the following grammar

S → A

A → aB | Ad

B → b

C → g

Ans Remove LR

S → A

A → aBA'

A' → aA' | ε

B → b

C → g

FIRST

FIRST(C) = {q}

FIRST(B) = {b}

FIRST(A') = {d, ε}

FIRST(BA) = {a}

FIRST(S) = FIRST(A) = {a}

Find the NTs

FOLLOW

~~FOLLOW(B) =~~

FOLLOW(S) = {$}

FOLLOW(A) = {$}

FOLLOW(A') = {$}

FOLLOW(A)

FOLLOW(B) = FIRST(A') = {d, $}

FOLLOW(C) = { }

Example 3 : Construct a predictive parser for the following grammar.

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

Parse the string : (a, a)

Ans Eliminate left recursion

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L' \rightarrow ,SL' \mid \varepsilon$

The grammar is:

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L' \rightarrow , SL' \mid \varepsilon$

FIRST (S) = { (, a }

FIRST (L) = FIRST (S) = { (, a }

FIRST (L') = { , , ε }

~~FIRST(S)~~

~~FOLLOW (S) = $, , (, a~~

~~FOLLOW: FIRST(L'), FOLLOW(L)~~

~~FOLLOW(S) = $, , $,)~~

FOLLOW (L), FOLLOW (L')

FOLLOW(S) = { $, , ,) }

FOLLOW(L) = {) }

FOLLOW(L') = {) }

see first symbol of each production and find FIRST( ) if ε FOLLOW( )

NT	(	)	a	,	$
S	$S \rightarrow (L)$		$S \rightarrow a$		
L	$L \rightarrow SL'$		$L \rightarrow SL'$		
L'		$L' \rightarrow \varepsilon$		$L' \rightarrow , SL'$	

Parsing string (a,a)

Stack	Input	Output
$ S	(a,a)$	S → (L)
$) L (	(a,a)$	
$) L	(a,a)$	L → SL'
$) L' S	(a,a)$	S → a
$) L' a	(a,a)$	
$) L'	,a)$	L' → ,SL'
$) L' S ,	a)$	
$) L' S	a)$	S → a
$) L' a	)$	
$) L'	)$	L' → ε
$)	)$	
$	$	

String parsed successfully

*/Yr/Grammar and Error Recovery Techniques

Example 4 : Construct a predictive parser for the following grammar.

$S \rightarrow iEtSS' | a$

$S' \rightarrow eS | \epsilon$

$E \rightarrow b$

Ans First (E) = b

First (S') = {e, ε}

FIRST (S) = {i, a}

FOLLOW (S) = {$, e}

FOLLOW (S') = FOLLOW(S) = {$, e}

FOLLOW (E) = {t}

FOLLOW(S), FOLLOW(S')

	i	t	a	e	b	$
S	S → iEtSS'		S → a			
S'				S' → eS S' → ε		S' → ε
E					E → b	

* LL1 Grammar

→ A grammar whose parsing table has no multiply-defined entries is said to be an LL(1) Grammar.

LL(1)
- ↙ input scanned from L→R
- ↳ → left most derivation

* What to do if parse table has multiply defined entries?

1. Eliminate left recursion
2. Left factor grammar
3. If the new grammar's parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.

* Properties of LL(1) Grammar

→ A grammar G is LL(1) iff the following conditions hold for two distinctive production rules

$$A \to \alpha \quad \text{and} \quad A \to \beta$$

→ Both α & β cannot derive symbols starting w/ the same terminals

→ At most one of α & β can derive to ε

* Error Recovery in Predictive Parsing?

An error may occur in predictive parsing if:

(i) the terminal symbol on top of the stack doesn't match with the current input symbol

(ii) If the top of the stack is a non-terminal A, the current input symbol is a, and the parsing table entry M[A,a] is empty

* What should the parser do in case of an error?

1. An error msg as meaningful as possible
2. Be able to recover from that error case and continue w/ parsing with the rest of the input.

* Error Recovery Techniques

1. **Panic Mode Error Recovery**

→ Skipping the input symbols until a synchronizing token is found

2. **Phrase-level Error Recovery**

→ Each empty entry in the parsing table is filled w/ a pointer to a specific error routine to take care of that error case.

3. **Error Productions**

→ Augment the grammar with productions that generate erroneous constructs

→ When an error production is used by the parser, generate appropriate error diagnostics.

→ Since it is almost impossible to know all the errors that can be made, not practical

4. **Global Correction**

→ Ideally, we want the compiler to make as few changes as possible in processing incorrect inputs.

→ Have to globally analyze the input to find the error.

→ Expensive method, not used in practice.

* Panic-Mode Error Recovery → pro - can be automated
  con - error msgs needed

→ Skip all the input tokens until a synchronizing token is found.

→ Synchronizing Token - All the terminal symbols in the follow set of a non-terminal can be used a synchronizing token set for that NT

∴ In a simple system:

(1) All the empty entries are marked as synch

(2) Parser will skip all the input symbols until a symbol in the follow set of the non-terminal A is on top of the stack

(3) Then pop that NT-A from the stack

(4) Parsing continues from that state.

(5) To handle unmatched terminal symbols, the parser pops that unmatched terminal symbol from the stack & issues an error msg. about the unmatched terminal symbol.

Example : Use predictive parsing to construct a parse tree for the following grammar. Parse the strings aab & cedb and use panic-mode error recovery.

$S \rightarrow Abs \mid \epsilon \mid e$

$A \rightarrow a \mid cAd$

FIRST(A) = { a, c }

Ans FIRST(S) = { a, c, e, ε }

FOLLOW(A) = { b, d }

FOLLOW(S) = { $ }

NT	a	b	e	c	d	$
S	S→Abs	sync	S→e	S→Abs	sync	S→ε
A	A→a	sync	sync	A→cAd	sync	sync

for aab

STACK	INPUT	OUTPUT
$S	aab$	S → abS
$ Sba	aab$	
$ Sb	ab $	

no match-error
since M [S, b] = sync

} In this case there is ~~ST~~ - ~~ST~~ mismatch pop ^b from stack, and ignore that anything happened

~~$S b~~	~~a b $~~	
	↑	

Take off the b

$ S	ab $	S → AbS
$ SbA	ab$	A → a
$ Sba	ab$	
$ Sb	b$	
$ S	$	S → ε
$	$	

accept

for ceadb

Stack	Input	Output
$S	ceadb$	S→AbS
$SbA	ceadb$	~~S→AbS~~ A→cAd
$SbaAc	ceadb$	
$SbdA	eadb$	

error
take off A
keep taking of input until you get = {b,d}

FOLLOW(A) = {b,d}

(since here a NT is on top of the stack)

$Sbd	db$	
$Sb	b$	S→ε
$S	$	
$	$	

accepted

* Phrase Level Error Recovery → pro: can be automated
→ con: recovery not intuitive

→ Each empty entry in the parse table is filled with a pointer to a special error routine which will take care of that error case.

→ These error routines may

- change / insert / delete i/p symbols
- issue appropriate error msgs
- pop items from the stack

→ Must be careful while designing these error routines as we may put the parser into an ∞ loop

* Error Productions

→ augment / ignore and parse

pro : powerful recovery method

con : cannot be automated

eg. add error production

$T' \rightarrow FT'$

to ignore missing *, eg id id

* YACC – Yet Another Compiler -Compiler

→ YACC is a tool designed to generate a parser based on a gramma you define.

→ used for LALR(1) grammars

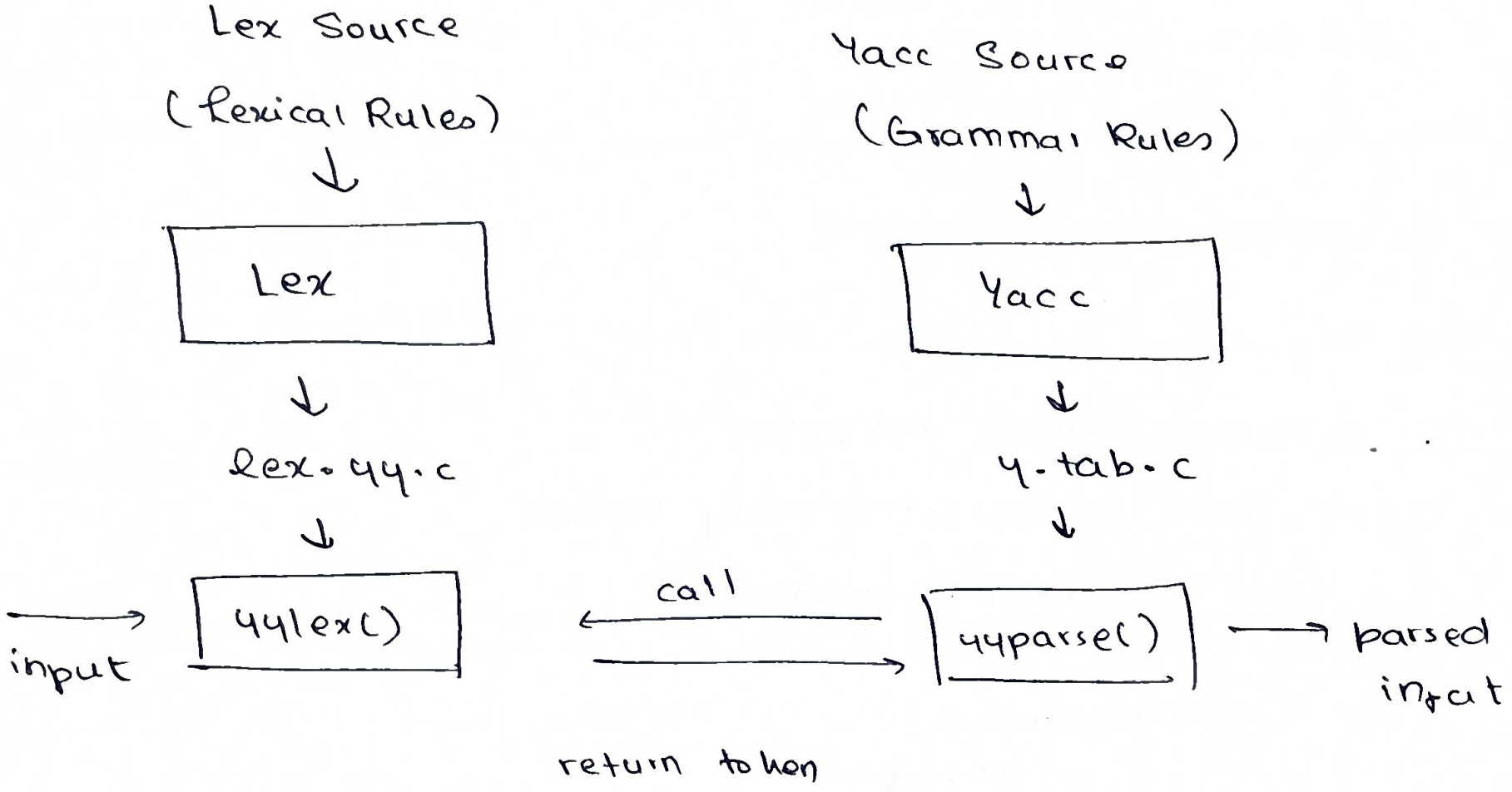
→ produces the source code for the parser (syntactic analyzer) of a language

* How YACC works with Lex

1. Lex → for lexical analysis - scans the input and produces tokens
2. Yacc uses the tokens to perform syntactic analysis & checks if the tokens follow the grammar rules.

yylex() → lex fn. to return tokens

yyparse() → Yacc function to parse i/p



* YACC File Format

```
%{
    C Declarations
%}
    yacc declarations
%%
    Grammar rules
%%
    Additional c code
```

* Example of Grammar Rules

```
expr:   expr '+' term      { $$ = $1 + $3 }
      | term
      ;

term :  term '*' factor    {                }
      | factor
      ;

factor : '(' expr ')'      {                }
       | ID
       | NUM
       ;
```

→ Each production rule has an action enclosed in { }

$$ = value of current non-terminal

$1, $2, $3 values of element on the RHS of the prod rule.

* Special Yacc Declarations

%start → start symbol

%token → token symbols

%type → data type for non-terminal symbols

%right, %left & %nonassoc → associativity & precedence for operators

* Bottom-up Parsing

→ Builds the parse-tree from the leaves up to the root

→ This approach tries to find the right-most derivation of the input in reverse order.

→ Starts with the input string & attempts to reduce it step-by-step to the start symbol

of Two-Types

- shift-reduce parsing
- operator-precedence parsing

Efficient Methods

→ LR methods (Left-to-right, rightmost derivation in reverse)

- SLR, Canonical LR, LALR

* Shift-Reduce Parsing

→ reduce given input string into the start symbol

→ Two main operations:

(i) Shift: Move the next input symbol onto a stack

(ii) Reduce: Replace a sequence of symbols on the stack, which matches the RHS of a production rule, with the corresponding non-terminal on the LHS

There are four actions the parser can take:

1. Shift: Push the next input symbol onto the stack
2. Reduce: Replace the top symbols on the stack with the LHS of a production rule.
3. Accept: If the input has been parsed successfully, the parser halts.
4. Error: If the parser detects an error, it calls an error recovery routine.

Example: Parse abbcde using shift-reduce method. The grammar is:

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

Input	Stack	Operation / Production Rule
abbcde	$	-
bbcde	$a	Shift
bcde	$ab	Shift
cde	$abb	Shift
de	$abbc	Shift
de	$aA	$A \rightarrow Abc$ $A \rightarrow b$

de	$aA	
e	$aAd	shift
e	$aABe	B → d
-	$aABe	shift
	$S	S → aABe.

* Handle

→ A handle of a string is a substring that matches the RHS of a production

→ Unambiguous forms of the grammar would have exactly one handle for every right sentential form.

→ Handle pruning is the process of reducing handles repeatedly in a right-most derivation in reverse order.

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \gamma_{n-1} \Rightarrow \gamma_n = w$$

Example for S → aABe
A → Abc | b
B → d

abbcde
aAbcde
aAde
aABe
S

underlined terms are handles

however
aAbcde
is not a handle → result is not in sentential form
(cannot be derived from S)

Example Use a shift-reduce parser for the string id + id * id for the grammar:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Ans Right-most Derivation.

$E \rightarrow E+T$

$\Rightarrow E+T*F$

$\Rightarrow E+T*id$

$\Rightarrow E+F*id$

$\Rightarrow E+id*id$

$\Rightarrow T+id*id$

$\Rightarrow F+id*id$

$\Rightarrow id+id*id$.

Input	Stack	Operation / Production Rule
id + id * id	$	-
+ id * id	$ id	shift
+ id * id	$ F	reduce $F \rightarrow id$
+ id * id	$ T	reduce $T \rightarrow F$
+ id * id	$E	$E \rightarrow T$
id * id	$E +	shift
* id	$E + id	shift
* id	$E + F	reduce $F \rightarrow id$

*id	$E+T	reduce $T \to F$
id	$E+T*	shift
$	$E+T*id	shift
$	$E+T*F	reduce $F \to id$
$	$E+T	$T \to T*F$
$	$E	$E \to E+T$

↙

start symbol

* Conflicts during Shift-Reduce Parsing:

→ There are CFGs for which shift-reduce parsers cannot be used.

→ Stack contents and the next input symbol may not decide:

(i) shift/reduce conflict - whether to make a shift operation or a reduction

(ii) reduce/reduce conflict - The parser cannot decide which of several reductions to make.

Shift-Reduce Conflicts

$S \to$ if E then S |
if E then S else S |
other.

Stack	Input	Action
⋮	⋮	
$ ... if E then s	else ... $	shift or reduce?

Soln: resolve in favor of shift, so else matches closest if.

Reduce - Reduce Conflicts

Grammar: $C \to AB$

$A \to a$

$B \to a$

Stack	Input	Action
$	aa $	shift
$a	a $	reduce to $A \to a$ or $B \to a$?

* Simple LR Parser

↳ A kind of shift-reduce parser that can handle a large class of context-free grammars.

L: left-to-right scanning of input

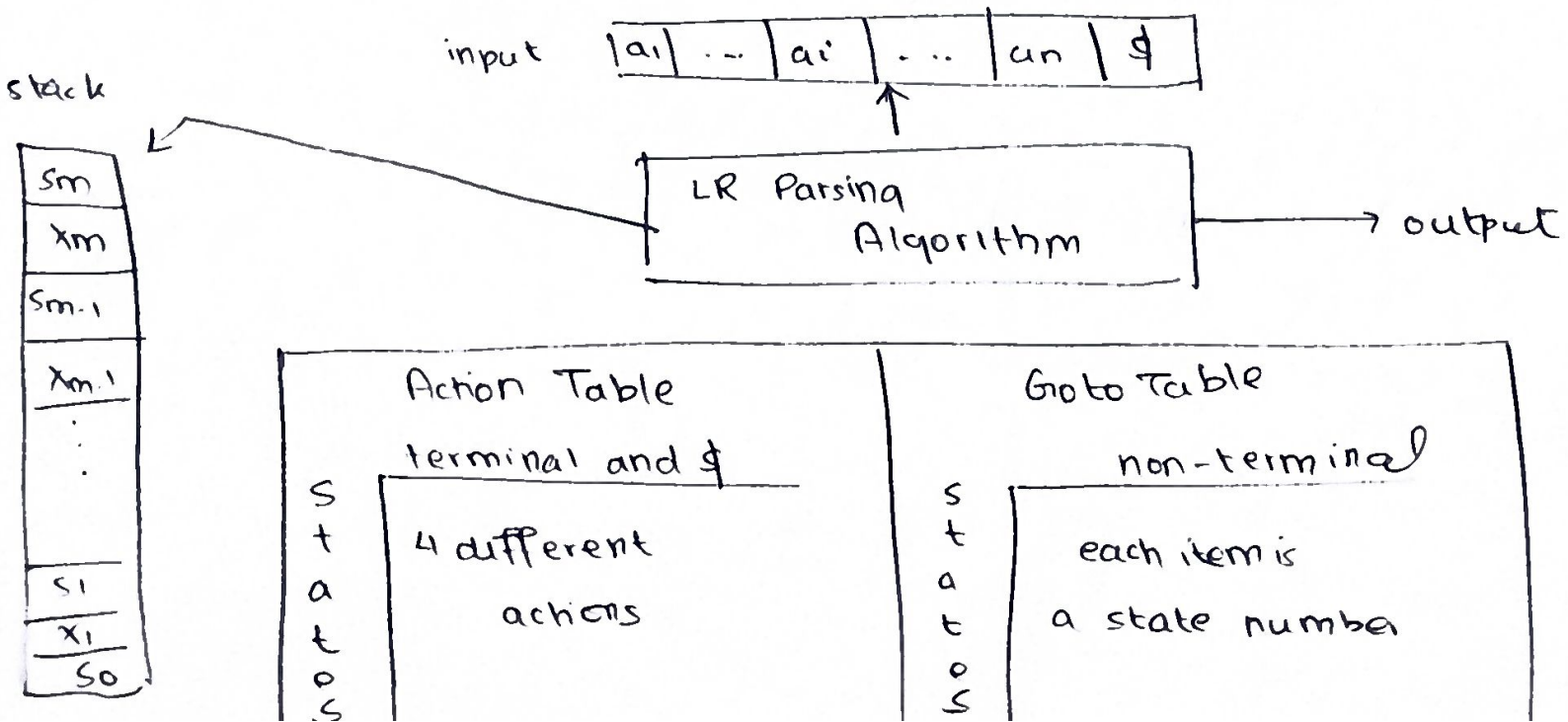
R: rightmost derivation in reverse

Types → SLR

→ CLR

→ LALR

(lookahead)



Steps

1. Construct an augmented grammar G' for any grammar G
2. Construct LR(0) collection of items
3. Construct DFA
4. Find FOLLOW for the non-terminals
5. Construct the parsing table

Example: Construct an SLR parser for the following grammar.

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Parse the string id + id * id

Step 1: Construct the augmented grammar

$E' \rightarrow E$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Step 2: Construct the canonical LR(0) collection

Use the closure & GOTO operations

I_0 : $E' \to .E$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

I_1 : GOTO(I_0, E)

$E' \to E.$

$E \to E.+T$

I_2 : GOTO(I_0, T)

$E \to T.$

$T \to T.*F$

I_3 : GOTO(I_0, F)

$T \to F.$

I_4 : GOTO(I_0, ()

$F \to (.E)$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

I_5 : GOTO(I_0, id)

$F \to id.$

I_6 : GOTO(I_1, +)

$E \to E+.T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

I_7 : GOTO(I_2, *)

$T \to T*.F$

$F \to .(E)$

$F \to .id$

I_8 : GOTO(I_4, E)

$F \to (E.)$

$E \to E.+T$

I_9 : GOTO(I_6, T)

$E \to E+T.$

$T \to T.*F$

I_{10} : GOTO(I_7, F)

$T \to T*F.$

I_{11} : GOTO(I_8,))

$F \to (E).$

Grammar

① $E' \to E$

② $E \to E+T$

③ $E \to T$

④ $T \to T*F$

⑤ $T \to F$

⑥ $F \to (E)$

⑦ $F \to id$

Step 3 Construct the parse table

State	action						goto		
	+	*	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3
1	S_6					acc			
2	R_2	S_7		R_2		R_2			
3	R_4	R_4		R_4		R_4			
4			S_4		S_5		8	2	3
5	R_6	R_6		R_6		R_6			
6			S_4		S_5			9	3
7			S_4		S_5				10
8	S_6			S_{11}					
9	R_1	S_7		R_1		R_1			
10	R_3	R_3		R_3		R_3			
11	R_5	R_5		R_5		R_5			

FOLLOW (E) = {$, +,)}

FOLLOW (T) = {$, +,), *}

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

Grammar

1. E → E+T
2. E → T
3. T → T*F
4. T → F
5. F → (E)
6. F → id

For action table: look for • T

write the no, where it becomes T.

For GOTO table look for • NT

write the no when it becomes NT.

For reduce operation, look for NT •

write prod no

Example 2 : Construct a parse tree for the following grammar using SLR parsing

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

Ans Augmented Grammar.

$S' \rightarrow S$

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

FOLLOW(S) = {$}

FOLLOW(L) = {=, $}

FOLLOW(R) = {$, =}

Canonical LR(0) Collection

I_0: closure
- $S' \rightarrow .S$
- $S \rightarrow .L = R$
- $S \rightarrow .R$
- $L \rightarrow .*R$
- $L \rightarrow .id$
- $R \rightarrow .L$

I_1 : GOTO(I_0, S)
- $S' \rightarrow S.$

I_2 : GOTO(I_0, L)
- $S \rightarrow L. = R$
- $R \rightarrow L.$

I_3 : GOTO(I_0, R)
- $S \rightarrow R.$

I_4 : GOTO(I_0, *)
- $L \rightarrow *.R$
- $R \rightarrow .L$
- $L \rightarrow .id$
- $L \rightarrow .*R$

I_5 : GOTO(I_0, id)
- $L \rightarrow id.$

I_6 : GOTO(I_2, =)
- $S \rightarrow L = .R$
- $R \rightarrow .L$
- $L \rightarrow .*R$
- $L \rightarrow .id$

I_7 : GOTO(I_4, R)
- $L \rightarrow *R.$

I_8 : GOTO(I_4, L)
- $R \rightarrow L.$

I_9 : GOTO(I_6, R)
- $S \rightarrow L = R.$
- $L \rightarrow *R.$

I_{10} : GOTO(I_6, *)
- $L \rightarrow *.R$
- $R \rightarrow .L$
- $L \rightarrow .*R$
- $L \rightarrow .id$

I_4

Parsing Table

	ACTION				GOTO	
	=	*	id	$	S	R	L
0		S4	S5		1	2	3
1				Acc			
2	S6, R5			R5			
3				R2			
4		S4	S5			7	8
5	R4			R4			
6		S4	S5			9	8
7	R3			R3			
8	R5			R5			
9	R3			R1, R3			

GRAMMAR

① S → L=R

② S → R

③ L → *R

④ L → id

⑤ R → L

Example 3 : Construct a parse table using an SLR parser for:

$E \to E+T \mid T$

$T \to TF \mid F$

$F \to F* \mid a \mid b$

Ans Augmented Grammar

$E' \to E$

(1) $E \to E+T$

(2) $E \to T$

(3) $T \to TF$

(4) $T \to F$

(5) $F \to F*$

(6) $F \to a$

$F \to b$

(7)

Canonical LR(0) Collection

Closure

I_0: $E' \to .E$

$E \to .E+T$

$E \to .T$

$T \to .TF$

$T \to .F$

$F \to .F*$

$F \to .a$

$F \to .b$

$I_1 = GOTO(I_0, E)$

$E' \to E.$

$E \to E.+T$

$I_2 = GOTO(I_0, T)$

$E \to T.$

$T \to T.F$

$F \to .F*$

$I_3 = GOTO(I_0, F)$

$T \to F.$

$F \to F.*$

$I_4 = GOTO(I_0, a)$

$F \to a.$

$I_5 = GOTO(I_0, b)$

$F \to b.$

$I_6 : GOTO(I_1, +)$

$E \to E+.T$

$T \to .TF$

$T \to .F$

$F \to .F*$

$F \to .a$

$F \to .b$

$I_7 = GOTO(I_2, F)$

$T \to TF.$

$F \to F.*$

$I_8 = GOTO(I_3, *)$

$F \to F* .$

$I_9 = GOTO(I_6, T)$

$E \to E+T.$

$T \to T.F$

$F \to .F*$

$F \to .a$

$F \to .b$

$I_{10} = GOTO(I_6, F)$

$T \to F.$

$F \to F.*$

~~$I_{11} = GOTO($~~

State	ACTION					GOTO		
	+	*	a	b	$	E	T	F
0			S_4	S_5		1	2	3
1	S_6				acc			
2	R_2		S_4	S_5	R_2			7
3	R_4	S_8	R_4	R_4	R_4			
4	R_6	R_6	R_6	R_6	R_6			
5	R_7	R_7	R_7	R_7	R_7			
6			S_4	S_5			9	10
7	R_3	S_8	R_3	R_3	R_3			
8	R_5	R_5	~~S_8~~ R_5	~~S_5~~ R_5	R_5			
9	R_1		S_4	S_5	R_1			7
10	R_4	S_8	R_4	R_4	R_4			

FOLLOW(E) = {$, +}

FOLLOW(T) = {$, +, a, b}

FOLLOW(F) = {*, a, b, $, +}

FIRST(F)

* Canonical LR Parser

Example 1 : Construct a CLR parser for the following grammar.

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$

Ans ① Augmented Grammar

1. $S' \rightarrow S$
2. $S \rightarrow CC$
3. $C \rightarrow cC$
4. $C \rightarrow d$

② Calculating LR(1) items

I_0: closure

$S' \rightarrow .S, \$$

$S \rightarrow .CC, \$$

$C \rightarrow .cC, c/d$

$C \rightarrow .d, c/d$

I_1 : GOTO (I_0, S)

$S' \rightarrow S., \$$

I_2 : GOTO (I_0, C)

$S \rightarrow C.C, \$$

$C \rightarrow .cC, \$$

$C \rightarrow .d, \$$

I_3: GOTO (I_0, c)

$C \rightarrow c.C, c/d$

$C \rightarrow .cC, c/d$

$C \rightarrow .d, c/d$

I_4 : GOTO (I_0, d)

$C \rightarrow d., c/d$

I_5 : GOTO (I_2, C)

$S \rightarrow CC., \$$

I_6 : GOTO (I_2, c)

$C \rightarrow c.C, \$$

$C \rightarrow .cC, \$$

$C \rightarrow .d, \$$

I_7 : GOTO (I_2, d)

$C \rightarrow d., \$$

I_8 : GOTO (I_3, C)

$C \rightarrow cC., c/d$

I_9 : GOTO (I_3, c)

$C \rightarrow c.C, c/d$ (I_3)

$C \rightarrow .d, c/d$

$C \rightarrow c.C, c/d$

I_{10} : GOTO (I_3, d)

$C \rightarrow d.$ (I_4)

I_9 : GOTO (I_6, C)

$C \rightarrow cC., \$$

(3) Constructing a Parse Table

State	Action			GOTO		
	c	d	$	S	B	C
0	S3	S4		1		2
1			acc			
2	S6	S7				5
3	S3	S4				8
4	R3	R3				
5			R4			
6	S6	S7				9
7			R3			
8	R2	R2				
9			R2			

No need to find follow for reduce.