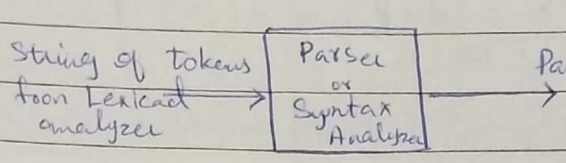
Compiler Construction Practice

Date 7-2-2024

Ch: 3 :: Syntax Analysis

- Second phase of compiler.



Types

① Universal.

② Top-Down.

Builds parse tree from top (root) to the bottom (leaves).

③ Bottomup:

Builds parse tree from the leaves and work their way up to the root.

In any case, input to the parser is scanned from left-to-right, one symbol at a time.

Grammar:

- A powerful tool for describing & analyzing the correct syntax of a language.
- A set of rules by which valid sentences in a language are constructed.
- Non-terminal: a grammar symbol that can be ~~constructed~~ replaced/expanded to a sequence of symbols. They are syntactic variables, intermediate.
- Terminal: an actual word of a language, symbols that cannot be replaced. They are tokens, leaves in a parse tree.
- Production: A grammar rule that describes how to replace/expand a symbol/non-terminal
- Start Symbol: A designation of one of the non-terminal

تیرا دیوانہ تیرے شہر میں ہے

Date 20

Example: All non-empty strings that start and end with the same symbol.

$S \rightarrow aXa \mid bXb \mid a \mid b$

$X \rightarrow aX \mid bX \mid \in$

Example: All strings with more a's than b's.

$S \rightarrow Aa \mid MS \mid SMA$

$A \rightarrow Aa \mid \in$

$M \rightarrow MM \mid bMa \mid aMb \mid \in$

Example: ~~All~~ abbaababa

$S \rightarrow MS$
$\rightarrow aMbS$
$\rightarrow abMabS$
$\rightarrow abbMaabS$
$\rightarrow abbaabS$
$\rightarrow abbaabMS$
$\rightarrow abbaabaMbS$
$\rightarrow abbaababS$
$\rightarrow abbaababAa$
$\rightarrow abbaababa$

bbaabab

$S \rightarrow MS$
$\rightarrow bMaS$
$\rightarrow bbMaaS$
$\rightarrow bbaaS$
$\rightarrow bbaaMS$
$\rightarrow bbaabMaS$
$\rightarrow bbaababa$

can't be parsed

Example: All palindromes (a palindrome is a string that reads the same forwards & backwards).

$S \rightarrow aSa \mid bSb \mid a \mid b \mid \in$

① Define a grammar for the language of strings with one or more a's followed by zero or more b's.

$S \rightarrow aA$

$A \rightarrow aA \mid bA \mid \in$

② Define a grammar for the even-length palindromes.

$S \rightarrow aSa \mid bSb \mid \in$

Date 20

⑤ Define a grammar for strings where the number of a's is equal to number of b's.

abaaba

$S \to aB \mid bA$
$B \to bS \mid b \mid aBB$
$A \to aS \mid a \mid bAA$

$S \to SaSbS \mid SbSaS \mid \in$

$S \to SaSbS$
$S \to ab$

④ Define a grammar for strings where the number of a's is not equal to the number of b's. (Hint: think about it as two separate cases).

× $S \to SaSbS \mid SbSaS \mid a \mid b$ ×

$S \to S_1 \mid S_2$
$S_1 \to aS_1 \mid S_1a \mid a.$
$S_1 \to bS_1S_1 \mid S_1bS_1 \mid S_1S_1b.$ } a's are greater
$S_2 \to bS_2 \mid S_2b \mid b.$
$S_2 \to aS_2S_2 \mid S_2bS_2 \mid S_2S_2b.$ } b's are greater

The grammar defines simple arithmetic expressions.

expression → expression + term
expression → expression − term
expression → term
term → term * factor
term → term / factor
term → factor
factor → (expression)
factor → id

OR

$E \to E+T \mid E-T \mid T$
$T \to T*F \mid T/F \mid F$
$F \to (E) \mid id$

A grammar for balanced parantheses

$S \to (S)S \mid \in$

Date ____ 20__

Ambiguity:

A grammar which generates more than one parse tree for a given string is said to be ambiguous.

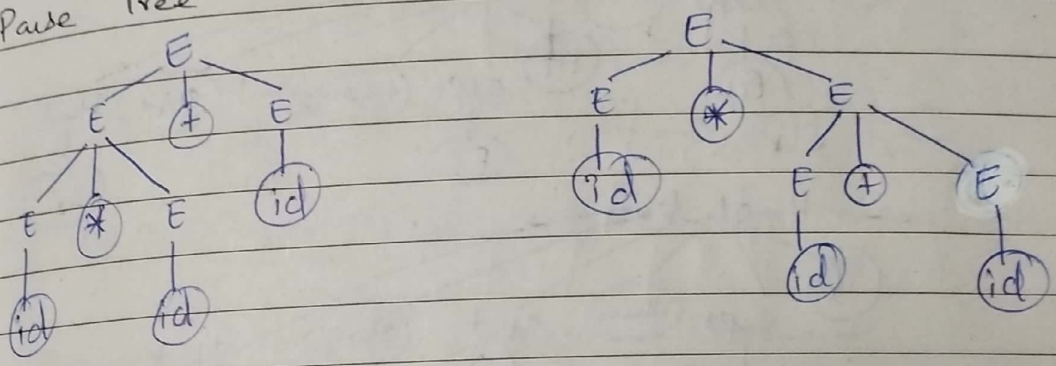
Example.

Grammar rules:

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

String id * id + id.

Parse Tree



Hence, the grammar is ambiguous.

Solution for ambiguity:

Associativity of operators:

mostly operators : +, −, *, / are left-associative.

left-associative means, if an operand has same operators on its left & right

e.g 9 + 5 + 2, 5 has + operator on left & right, 5 first belongs to its left side operator.

Precedence of Operators:

* & / operators have more precedence than + & -

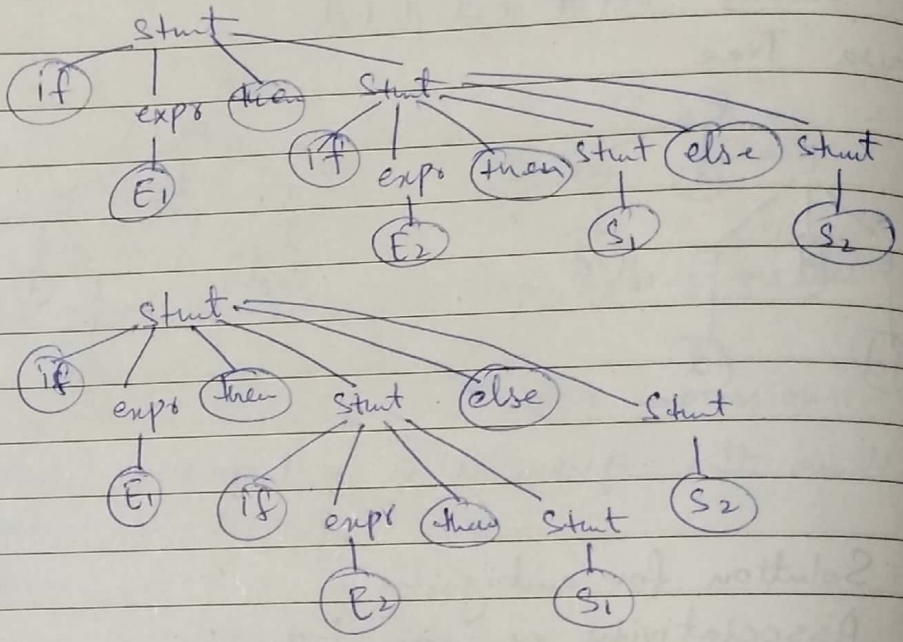
Operators on the same line have same associativity & precedence.

Date

Ambiguous Grammar.

stmt → if expr then stmt ①
| if expr then stmt else stmt ②
| other ③

String :: if E_1 then if E_2 then S_1 else S_2.



Two parse trees, grammar is ambiguous.

- The idea is that a stmt appearing between a then and an else ② must be 'matched', i.e. the interior stmt ① must not end with an unmatched then
- A matched stmt is either an if – then – else ② statement containing no open statements or it is any other kind of unconditional statement.

Unambiguous Grammar.

stmt → matched_stmt | open_stmt

matched_stmt → if expr then matched_stmt else matched_stmt | other

open_stmt → if expr then stmt |
if expr then matched_stmt else open_stmt

Date

Ambiguity Practice

① bExp → bExp AND bExp |
bExp OR bExp |
NOT bExp | True | False.

After removing ambiguity.

bExp → bExp OR bExp1 | bExp1.
bExp1 → bExp1 AND bExp2 | bExp2.
bExp2 → NOT bExp2 | True | False.

② R → R + R | R . R | R* | a | b | c

After removing ambiguity.

R → R + R1 | R1
R1 → R1 . R2 | R2
R2 → R2* | a | b | c

R → R R' | a | b | c
R' → + R | . R | *

& A → AA | (A) | ∈

rexp → rexp "|" rexp | rexp rexp
| rexp "*" | (rexp) | letter

rexp → rexp "|" rexp2. | rexp2.
rexp2 → rexp2 rexp3 | rexp3.
rexp3 → rexp4 "*" | rexp4.
rexp4 → "(" rexp4 ")" | letter.

(ab|b)*

Date ____ 20__

Elimination of Left Recursion.

- A grammar is left-recursive if it has a non-terminal A such that $A \rightarrow A\alpha$.
- Top-down parsing cannot handle left-recursion

General form of left-recursive Grammar.

$$A \rightarrow A\alpha \mid \beta.$$

Here α and β can be a terminal or non-terminals.
This form of production could be replaced by the non-left-recursive productions.

$$A \rightarrow A\alpha \mid \beta \longrightarrow \begin{matrix} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \epsilon \end{matrix}$$

This rule by itself suffices for many grammars

Example: Arithmetic Expressions Grammar.

$E \rightarrow E+T \mid E-T \mid T$ → left recursive
$T \rightarrow T*F \mid T/F \mid F$ → left recursive.
$F \rightarrow (E) \mid id$

$E \rightarrow E+T \mid E-T \mid T$ ∵ $A \rightarrow A\alpha \mid \beta$.
$E \rightarrow TE'$ $A = E$
$E' \rightarrow +TE' \mid -TE' \mid \epsilon$ $\alpha = +T, -T$.
$\beta = T$

$T \rightarrow T*F \mid T/F \mid F$
$T \rightarrow FT'$ $A = T$
$T' \rightarrow *F \mid /F \mid \epsilon$. $\alpha = *F, /F$
$\beta = F$

Now, the non-left-recursive Grammar is.

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid -TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid /FT' \mid \epsilon$
$F \rightarrow (E) \mid id$

Date 20

Eliminating Left recursion..

E → int | int * E | int / E | E - E

① expr → expr + term | term

term → term factor | factor

factor → factor * | primary

primary → a | b

Eliminate left recursion.

① expr → expr + term | term

A α β

∵ A → Aα | β

A → β A'

A' → α A' | ∈

expr → term expr'

expr' → + term | ∈

② term → term factor | factor

A α β

term → factor term'

term' → factor term' | ∈

③ factor → factor * | primary

A α β

factor → primary factor'

factor' → * factor' | ∈

Grammar:

S → Aa | b

A → Ac | Sd | ∈ → contains left-recursion.

S contains recursion as well.

- first substitute S in A's production

A → Ac | Aad | bd | ∈

now remove L.R.

A → bd A'

A' → cA' | adA' | ∈

S → Aa | b

A → bd A'

A' → cA' | adA' | ∈

Date ____ 20

Left-Factoring:

- Useful for producing a grammar suitable for predictive, or top-down parsing.
- When the choice b/w two alternative A-production isn't clear, we can rewrite the productions to defer the decision until enough of the input has been seen that we can make right choice for..

Ex:→ $A \rightarrow aAb \mid aAc$.

which production to choose isn't clear.

on reading a, we can't be assured that we're choosing the right production.

General form which may require left-factoring

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

α, β_1 and β_2 can be any terminals or non-terminals

↓

$A \rightarrow \alpha A'$

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

Example:

Grammar: $S \rightarrow iEtS \mid iEtSeS \mid a$.

$E \rightarrow b$.

$S \rightarrow iEtSS' \mid a$.

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$.

$\alpha \Rightarrow iEt$.

$\beta_1 \Rightarrow S$

$\beta_2 \Rightarrow SeS$

Example: Left factor the following grammar.

$E \rightarrow int \mid int + E \mid int - E \mid E - (E)$.

$E \rightarrow int\ E' \mid E - (E)$

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

$\alpha \Rightarrow int$

$\beta_1 = \epsilon$

$\beta_2 = +E$

$\beta_3 = -E$

Date ____ 20__

Q: Left factor the given grammar

rexpr → rexpr + rterm | rterm.
rterm → rterm rfactor | rfactor.
rfactor → rfactor * | rprimary.
rprimary → a | b

No left factoring needed.

Q: Eliminate left recursion and left factor the grammar (if needed).

① S → S S + | S S * | a.

Removing left recursion & Left factoring

S → S S S' | a.
S' → + | *

Removing Left recursion:

S → a S''
S'' → a S'' S' S'' | ∈
S' → + | *

S → S S S' | a
S' → + | *
S → a S''
S'' → S S' S'' | ∈

② S → 0 S 1 | 0 1

Left factoring

S → 0 S'
S' → S 1 | 1

② S → 0 S 1 | 0 1

Not left recursive & no left factor is needed

③ S → + S S | * S S | a.

Not left recursive & no left factoring needed

④ S → S (S) S | ∈

Removing left recursion.

S → S'
S' → (S) S S' | ∈.

No left factoring needed.

Q. S → a S S b S | a S a S b | a b b | b.

S → a S' | b
S' → S S b S | S a S b | b b

S → a S' | b.
S' → S'' | b b
S'' → S b S | a S b

Date 20

(5) $S \rightarrow S+S \mid SS \mid (S) \mid S* \mid a$.

Left factoring

$S \rightarrow SS_1 \mid (S) \mid a$.

$S_1 \rightarrow +S \mid S \mid *$

Eliminating left recursion.

$S \rightarrow (S)S' \mid aS'$

$S' \rightarrow +SS' \mid SS' \mid *S' \mid \epsilon$

Top-Down Parsing

- Constructing a parse tree for the input string, starting from root & creating the nodes of the parse tree in preorder
- Left-most derivation of an input string.
- Recursive descent parsing, a general form of top-down parsing, which may require back-tracking to find the correct production
- Predictive parsing, is a special case of recursive descent parsing, where no backtracking is required.

It chooses correct production by looking ahead at the input (one at a time)

Ex: Grammar.

$E \rightarrow TE'$

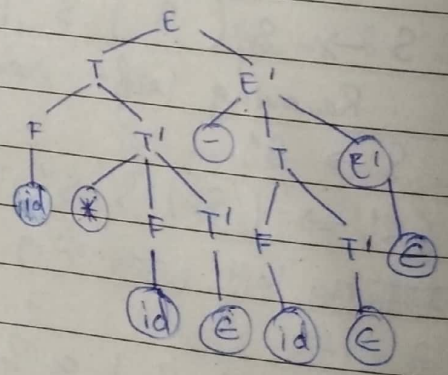
$E' \rightarrow +TE' \mid \epsilon \mid -TE'$

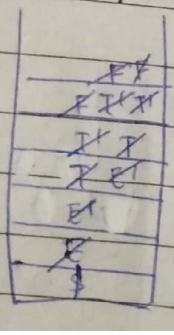
$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$.

id * id - id $





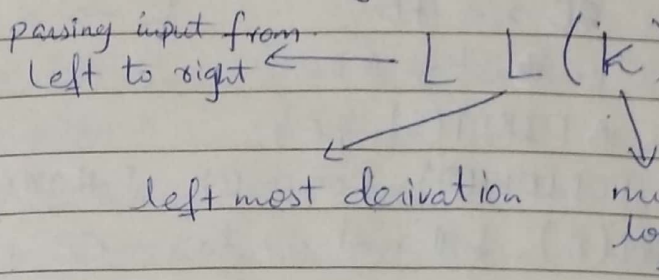
LL PARSER.

Date 20

Predictive parsers (recursive descent parsers with no backtracking) can be constructed for a class of LL grammar.

LL Parser accepts LL grammar, a subset of context-free grammar with some restrictions

LL Parser is denoted by LL(k)



The construction of this parser is aided by two functions FIRST() and FOLLOW() associated with Grammar.

During top-down parsing, these functions allow us to choose which production to apply, based on the next input symbol

FIRST()

First(A), A is any symbol in the grammar, terminal or non-terminal

FIRST(∈) = {∈}.

FIRST (terminal) = {terminal}.

Ex:- A → CB.

FIRST(A) = {c}.

FOLLOW(r):

To compute Follow(A) for all non-terminals A.

- Place $ in the start non-terminal is FOLLOW(S).
- For any non-terminal (for ex A), for which you are finding its FOLLOW(A), check that on the expanded side of product what comes after that non-terminal 'A' 'A' (on
- If it is terminal, then add it in the set

Date ____ 20

- if it is a non-terminal (for ex: B), then everything in the FIRST(B), will be in FOLLOW(A).
- if there is nothing after the non-terminal A, then or if the non-terminal B after A has ϵ in it FIRST(B), then check the FOLLOW of that rule's non-terminal, it'll become FOLLOW(A).

Ex:- $C \to aAB$ — in this production A has B as FIRST(B) = {ϵ}. then FOLLOW(A) = FOLLOW(C)

$C \to aA$ — in this production there is nothing after A, then FOLLOW(A) = FOLLOW(C)

- FOLLOW(A) will not contain ϵ.

Example: Grammar.

$E \to TE'$

$E' \to +TE' \mid \epsilon \mid -TE'$

$T \to FT'$

$T' \to *FT' \mid /FT' \mid \epsilon$

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

Find FIRST

FIRST(E) = FIRST(T) = {(, id}.

FIRST(E') = {+, -, ϵ}.

FIRST(T) = FIRST(F) = {(, id}.

FIRST(T') = {*, /, ϵ}.

FIRST(F) = {(, id}

Find FOLLOW

FOLLOW(E) = {$,)}

FOLLOW(E') = FOLLOW(E) = {$,)}.

FOLLOW(T) = (FIRST(E') - ϵ) U FOLLOW(E)) U FOLLOW(E') = {+, -, $,)}

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

FOLLOW(F) = (FIRST(T') - ϵ) U FOLLOW(T) U

Date 20

How to check If a grammar is LL(1) or not.

Let $G \Rightarrow A \to \alpha \mid \beta$.

- First(α) and FIRST(β) are disjoint sets, means their intersection is ϕ or $FIRST(\alpha) \cap FIRST(\beta) = \phi$
- If ϵ is in First(β) => $FIRST(\alpha) \cap FOLLOW(A) = \phi$
  & same logic for ϵ in FIRST(α).

LL(1) Parsing Table.

① Make a table consisting of total non-terminals (row) and total terminals/input symbols (columns)

② Now check for the non-terminal (in a row) e.g. S and terminal (in column), if that terminal is in the set of FIRST(S) of S then, write the rule which accepts that terminal in the cell.

③ If ϵ is in FIRST(S), then check FOLLOW(S) & fill the cells with that rule (epsilon rule).

LL(1) Parsing Table for Arithmetic Expression Grammar

Non Terminals	Terminals / Input Symbols. id	+	−	*	/	(	)	$
E	E→TE'					E→TE'		
E'		E'→+TE'	E'→-TE'				E'→ϵ	E'→ϵ
T	T→FT'					T→FT'		
T'		T'→ϵ	T'→ϵ	T'→*FT'	T'→/FT'		T'→ϵ	T'→ϵ
F	F→id					F→(E)		

Date

Ex: The following grammar, written abstractly for if-else statement:

$S \rightarrow i\ E\ t\ S\ S' \mid a$

$S' \rightarrow e\ S \mid \epsilon$

$E \rightarrow b.$

Find ~~Follow~~ FIRST

$FIRST(S) = \{i, a\}.$

$FIRST(S') = \{e, \epsilon\}$

$FIRST(E) = \{b\}$

Find FOLLOW

$FOLLOW(S) = (FIRST(S') - \epsilon) \cup FOLLOW(S) \cup FOLLOW(S')$

$= \{e, \$\}.$

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

$FOLLOW(E) = \{t\}.$

Passing table.

Non-terminals	Terminals.					
	i	t	a	e	b	$
S	$S \rightarrow iEtSS'$		$S \rightarrow a$			
S'				$S' \rightarrow eS$ $S' \rightarrow \epsilon$		$S' \rightarrow \epsilon$
E	$E \rightarrow b$					

The grammar is not LL(1) parsable, since.

the rule $S' \rightarrow eS \mid \epsilon$

if $\epsilon \Rightarrow FIRST(\beta)$, according to second point.

$FIRST(eS) \cap FOLLOW(S').$

$\{e\} \cap \{e, \$\} = \{e\}.$

and $\{e\} \neq \phi$

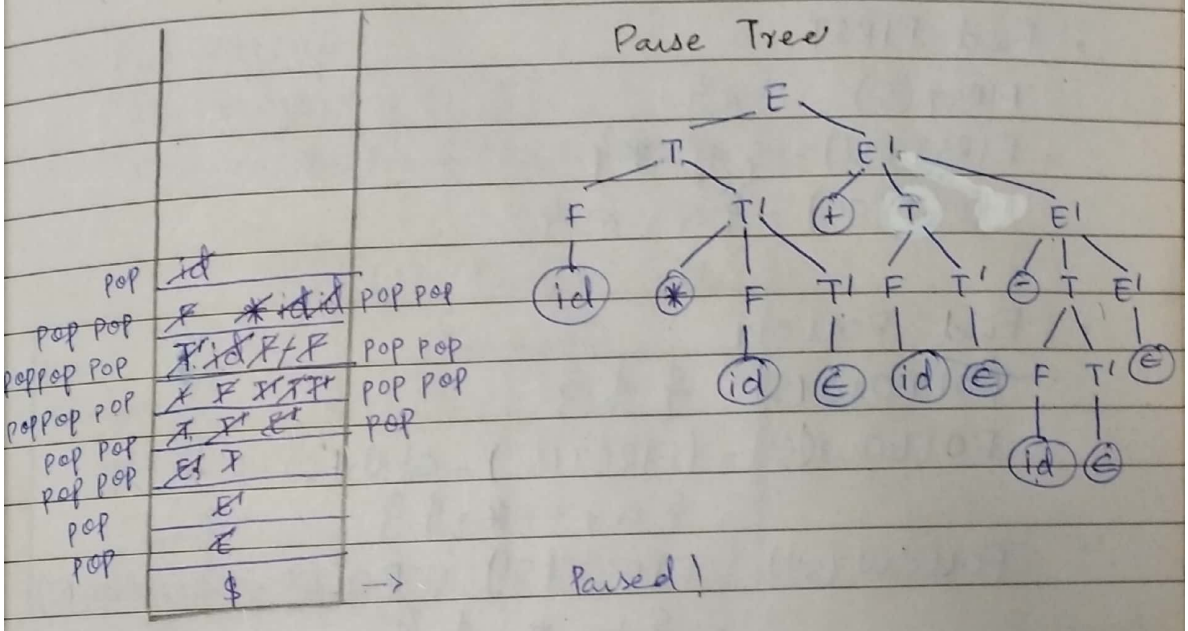
Date 20

How to parse an input string using LL(1) Parse Table.

- An input string, ending with $
- A stack, which has $ initially at its top.
- A parsing table.

Let's parse → id * id + id - id, using arithmetic grammar.

Input: id * id + id - id $



① Put the starts non-terminal at the top and check for it's possible rules according to the lookahead "L" pointing on the input & Make nodes for the symbols recognized.

② The rule, which is to be applied, place its symbol in the stack from right to left. & make their nodes.

③ after each placement in stack, check if a terminal is at top, if yes, pop it & move lookahead forward.

Pop the non-terminal, if it is done.

Date 20

Grammar: Postfix expression with operand a

$S \rightarrow SS+ \mid SS* \mid a$.

- After Left-factoring.

$S \rightarrow SSS' \mid a$.

$S' \rightarrow + \mid *$

- After eliminating left-recursion.

$S \rightarrow a\ S''$

$S'' \rightarrow S\ S'\ S'' \mid \epsilon$ $\Rightarrow$ $S'' \rightarrow a\ S''\ S'\ S'' \mid \epsilon$

$S' \rightarrow + \mid *$ $\quad S' \rightarrow + \mid *$

- Find FIRST

FIRST(S) = {a}.

FIRST(S') = {+, *}

FIRST(S'') = {a, ∈}

- Find FOLLOW

FOLLOW(S) = {$}

FOLLOW(S') = (FIRST(S'') − ∈) ∪ FOLLOW(S'')
{a, +, *, $}.

FOLLOW(S'') = FIRST(S') ∪ FOLLOW(S) ∪ FOLLOW(S'')
= {+, *, $}

LL(1) Parsing Table.

Non-Terminals	Terminals			
	a	+	*	$
S	$S \rightarrow a\ S''$			
S'		$S' \rightarrow +$	$S' \rightarrow *$	
S''	$S'' \rightarrow a\ S''\ S'\ S''$	$S'' \rightarrow \epsilon$	$S'' \rightarrow \epsilon$	$S'' \rightarrow \epsilon$

Date ____ 20__

Grammar

$S \to 0\,S\,1 \mid 0\,1$.

after left-factoring

$S \to 0\,S'$

$S' \to S\,1 \mid 1$

Find ~~Follow~~. FIRST.

FIRST(S) = {0}.

FIRST(S') = FIRST(S) ∪ {1} = {0, 1}.

Find FOLLOW.

FOLLOW(S) = {1, $}.

FOLLOW(S') = FOLLOW(S) = {1, $}.

LL(1) Parsing Table

Non-Terminals	Terminals.	
	0	1.
S	$S \to 0S'$	
S'	$S' \to S1$	$S' \to 1$

Example: Prefix expression with operand a.

$S \to +SS \mid *SS \mid a$.

Find FIRST

FIRST(S) = {+, *, a}.

Find FOLLOW

FOLLOW(S) = FIRST(S) ∪ FOLLOW(S) = {+, *, a, $}.

LL(1) Parsing Table

Non Terminal	Terminal.			
	+	*	a	$
S	$S \to +SS$	$S \to *SS$	$S \to a$.	