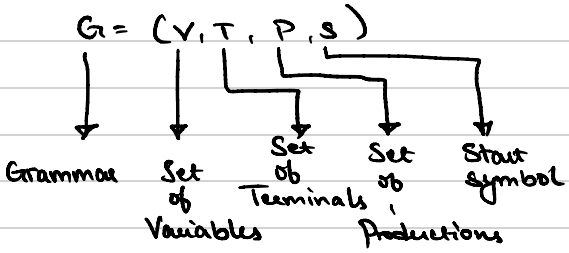
Compiler Design - II

REPRESENTATIVE GRAMMAR



* TERMINAL: Symbols from which strings are formed.
  - lowercase letter, operators, punchtuation, Digits, bold-face.

* NON-TERMINAL: Syntactic Variables that denote a set of string.
  - Uppercase letter, lowercase-italics

* START-SYMBOL: Head of production, Stated first in grammar.

* PRODUCTION: It's of the form LHS → RHS.

Error Recovery Strategy

Panic-mode Recovery	Phrase-level Recovery	Error Production	Global Correction
• Error Detected • enter panic mode • Discard input token one-by-one • until delimiter reached	• Error Detected • Parser make local changes to input - including a missing token • Continue Parsing	• Add/Extend Special rules to grammar to specify common errors. • Error Detected • Matches input based on Special rule and provide feedback.	• look at entire input & gives various possible correction. • Uses set of algo to make input correct (minimal cost)

DERIVATIONS → Rewriting Rules to generate a string.

Left-most Derivations

* E → E+E | E * E | id
w = id + id * id

E → E+E
E → id+E
E → id + E * E
E → id+id*E
E → id+id+id

* Leftmost Non-terminal is Expanded

Right-most Derivations

E → E+E | E*E | id
w = id + id * id

E → E+E
E → E+E*E
E → E+E*id
E → E+id*id
E → id+id*id

* Right-most non terminal is expande

AMBIGUOUS GRAMMAR

* Grammar is said to be Ambiguous if it can generate more than one-parse tree for the given input string.
  - parse tree → Diagrammatic Representation of Derivation process.

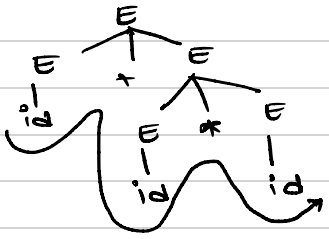
* Here,
  - Root node — start symbol
  - Internal nodes — non-terminals
  - Leaf — Terminals

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

LMD:

$E \rightarrow E+E$
$E \rightarrow id+E$
$E \rightarrow id+E*E$
$E \rightarrow id+id*E$
$E \rightarrow id+id*id$

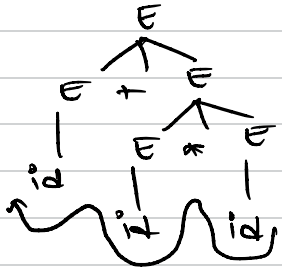
PARSE TREE:



RMD:

$E \rightarrow E+E$
$E \rightarrow E+E*E$
$E \rightarrow E+E*id$
$E \rightarrow E+id*id$
$E \rightarrow id+id*id$

→ Ambiguous



#2- $S \rightarrow aSbS$
$S \rightarrow bSaS$
$S \rightarrow \epsilon$

LMD:

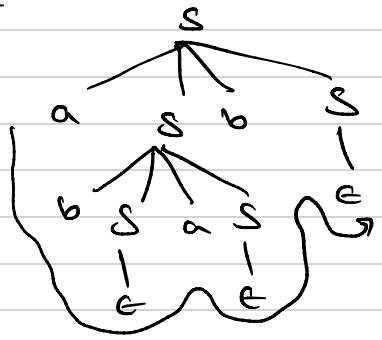
$S \rightarrow aSbS$
$\rightarrow abSaSbS$
$\rightarrow ab\epsilon a\epsilon b\epsilon$
$\rightarrow abab$

LMD:

$S \rightarrow aSbS$
$\rightarrow a\epsilon bS$
$\rightarrow abS$
$\rightarrow abaSbS$
$\rightarrow aba\epsilon b\epsilon$
$\rightarrow abab$

→ Ambiguous

abab



$ab\epsilon a\epsilon b\epsilon$
abab

Left Recursion & Left factoring

Production of the form $A \to A\alpha$

Transformation Technique to remove common prefixes.

LEFT- RECURSION:

$A \to A\alpha \mid \beta$

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

#1- Eliminate Left- Recursion

$E \to E+T \mid T$ (A, $A\alpha$, β)

$T \to T*F \mid F$ (A, $A\alpha$, β)

$F \to (E) \mid id$

$E \to TE'$

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

$E \to TE'$

$E \to *FT' \mid \epsilon$

LEFT- FACTORING:

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots$

$A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2 \cdots$

#1- Eliminate left factoring

$S \to iEtS \mid iEtSeS \mid a$ (α, $E \to \beta_1$, β_2)

$E \to b$

$S \to iEtSS' \mid a$

$S' \to \epsilon \mid eS$

$E \to b$

Recursive Decent Parser – top- Down Parser. → cannot process left Recursive grammar

$E \to iE'$

$E' \to +iE' \mid \epsilon$

} 2 Non terminals

```
E()
{ if (input==i)
    input++;
  EPRIME();
}
```

```
EPRIME()
{ if (input == '+') {
    input++;
    if (input == 'i')
      input++;
    EPRIME();
  }
  else
    return;
}
```

Predictive / LL(1) Parser.

→ At a time, reading only one input symbol.

- Scanning input string from left to Right
- using Left-most Derivation.

CONSTRUCTION:

#1 - Eliminate Left-Recursion
#2 - Eliminate Left-factoring
#3 - Calculate First AND Follow
#4 - Construct Parsing table
#5 - Check if input string accepted by Parser or not.

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

Step 1:

$A \to A\alpha \mid \beta$
$A \to \beta A'$
$A' \to \alpha A' \mid \varepsilon$

$E \to TE'$ (1)
$E' \to +TE' \mid \varepsilon$ (2)
$T \to FT'$ (3)
$T' \to *FT' \mid \varepsilon$ (4)
$F \to (E) \mid id$ (5)

Step 2: No Left Factoring

Step 3:

- First(E) = { (, id }
- First(E') = { +, ε }
- First(T) = { (, id }
- First(T') = { *, ε }
- First(F) = { (, id }

→ look at Left Production

- Follow(E) = { $,) }
- Follow(E') = { $,) }
- Follow(T) = { +, $,) }
- Follow(T') = { +, $,) }
- Follow(F) = { *, +, $,) }

→ look at Right production → start symbol

Step 4:

	+	*	(	)	id	$
E			1			1
E'	2			2		2
T			3		3	
T'	4	4		4		4
F			5		5	

Step 5: input: id + id $

STACK	INPUT	ACTION
$E	id+id $	E→TE'
$E'T	id+id $	T→FT'
$E'T'F	id+id $	F→id
$E'T'id	id+id $	POP
$E'T'	+id$	T'→ε
$E'	+id$	E'→+TE'
$E'T+	+id $	POP
$E'T	id $	T→FT'
$E'T'F	id $	F→id
$E'T'id	id $	POP
$E'T'	$	
$E'	$	
$	$	

(until stack = $)

Use to Construct Parse tree.