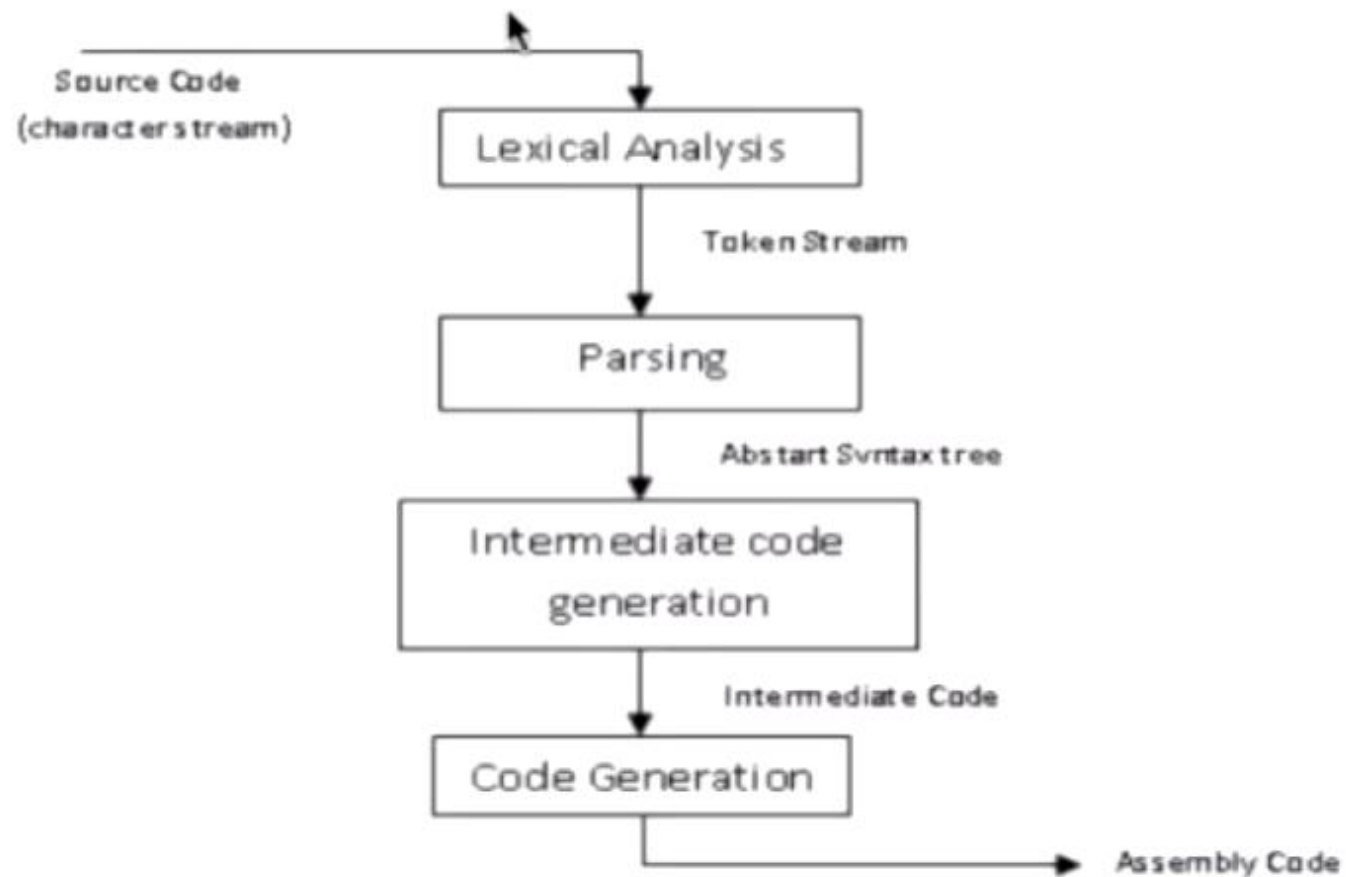


Syntax Analysis / Parsing

The big picture



Syntactic Analysis

- Lexical Analysis was about ensuring that we extract a set of valid **words** (i.e., tokens/lexemes) from the source code
- But nothing says that the words make a coherent **sentence** (i.e., program)
- Example:
 - “for while i == == == 12 + for (abcd)”
 - Lexer will produce a stream of tokens: <TOKEN_FOR> <TOKEN_WHILE>
<TOKEN_IDENT, “i”> <TOKEN_COMPARE> <TOKEN_COMPARE> <TOKEN_COMPARE>
<TOKEN_NUMBER, “12”> <TOKEN_OP, “+”> <TOKEN_FOR> <TOKEN_OPAREN>
<TOKEN_ID, “abcd”> <TOKEN_CPAREN>
 - But clearly we do not have a valid program
 - This program is lexically correct, but syntactically incorrect

Grammar

- How do we determine that a sentence is syntactically correct?
- Answer: We check against a **grammar**!
- A grammar consists of rules that determine which sentences are correct
- Example in English:
 - A sentence must have a verb
- Example in C:
 - A "{" must have a matching "}"

Context-Free Grammars

- A context-free grammar (CFG) consists of a set of **production rules**
- Each rule describes how a **non-terminal symbol** can be “replaced” or “expanded” by a string that consists of **nonterminal symbols** or by **terminal symbols**
 - Terminal symbols are really tokens
 - Rules are written with syntax like regular expressions
- Rules can then be applied recursively
- Eventually one reaches a string of only terminal symbols, or so one hopes
- This string is syntactically correct according to the grammatical rules!

CFG Example

- Set of non-terminals: A, B, C (uppercase initial)
- Start non-terminal: S (uppercase initial)
- Set of terminal symbols: a, b, c, d
- Set of production rules:
 - $S \rightarrow A \mid BC$
 - $A \rightarrow Aa \mid a$ (Extended Backus-Naur form - EBNF)
 - $B \rightarrow \underline{bBCb} \mid b$
 - $C \rightarrow dCcd \mid c$
- We can now start producing syntactically valid strings by doing **derivations**
- Example derivations:
 - $S \rightarrow BC \rightarrow bBCbC \rightarrow \underline{bbdCcdbC} \rightarrow \underline{bbdccbC} \rightarrow bbdccdbc$
 - $S \rightarrow A \rightarrow Aa \rightarrow Aaa \rightarrow Aaaa \rightarrow aaaa$

A Grammar for Expressions

<u>Expr</u>	→ <u>Expr</u> Op <u>Expr</u>
<u>Expr</u>	→ Number Identifier
Identifier	→ Letter Letter Identifier
Letter	→ a-z
Op	→ "+" "-" "*" "/"
Number	→ Digit Number Digit
Digit	→ 0 1 2 3 4 5 6 7 8 9

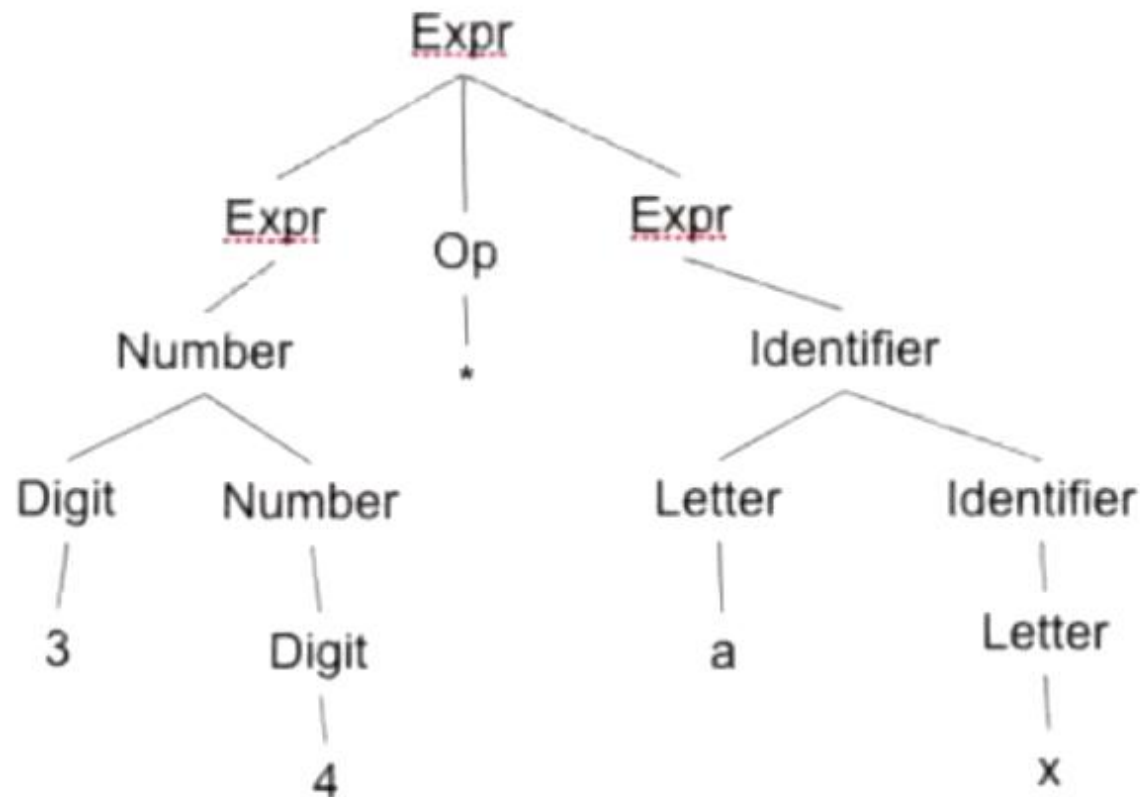
Expr → Expr Op Expr → Number Op Expr → Digit Number Op Expr → 3
Number Op Expr → 34 Op Expr → 34 * Expr → 34 * Identifier → 34 *
Letter Identifier → 34 * a Identifier → 34 * a Letter → 34 * ax

What is Parsing?

- What we just saw is the process of, starting with the start symbol and, through a sequence of rule derivation obtain a string of terminal symbols
 - We could generate all correct programs
- When we say we can't parse a string, we mean that we can't find any legal way in which the string can be obtained from the start symbol through derivations
- Parser : a program that takes in a string of tokens (terminal symbols) and discovers a derivation sequence, thus validating that the input is a syntactically correct program

Derivations as Trees

- A convenient and natural way to represent a sequence of derivations is a **syntactic tree** or **parse tree**
- Example:
- $\text{Expr} \rightarrow \text{Expr Op Expr} \rightarrow \text{Number Op Expr} \rightarrow \text{Digit Number Op Expr} \rightarrow 3 \text{ Number Op Expr} \rightarrow 34 \text{ Op Expr} \rightarrow 34 * \text{Expr} \rightarrow 34 * \text{Identifier} \rightarrow 34 * \text{Letter Identifier} \rightarrow 34 * a \text{ Identifier} \rightarrow 34 * a \text{ Letter} \rightarrow 34 * ax$

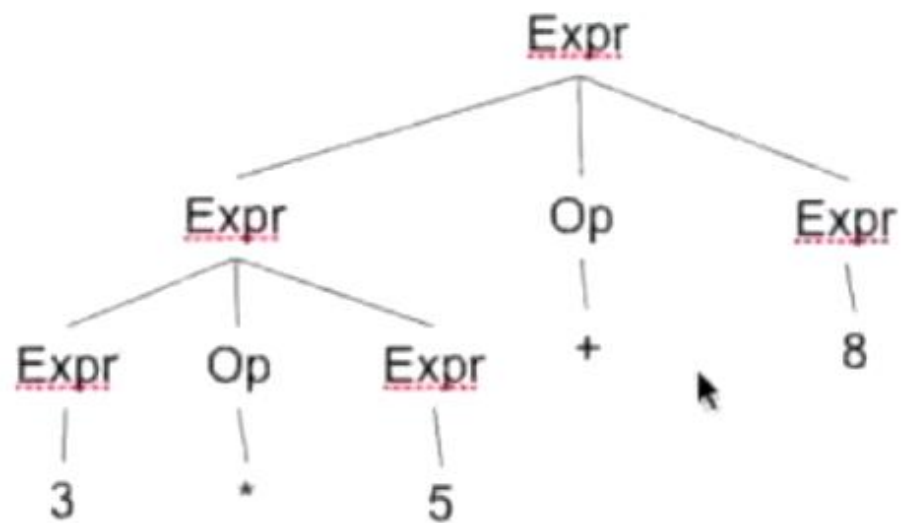


Ambiguity

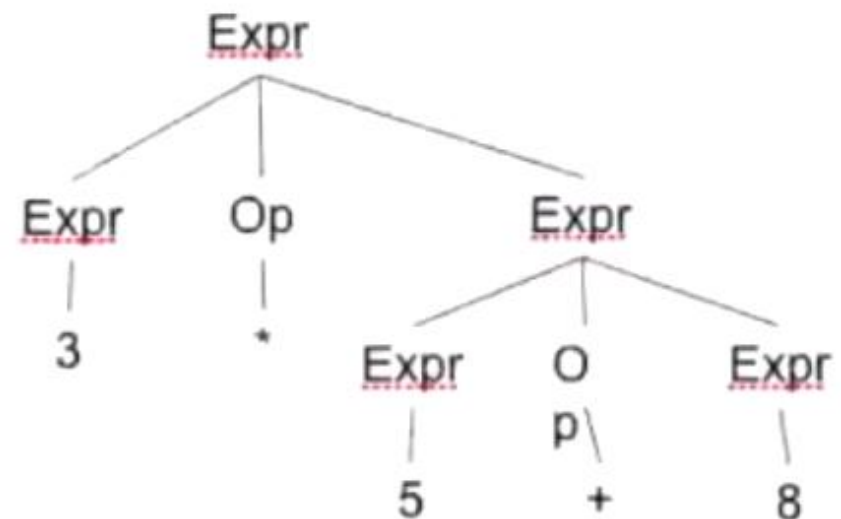
- We call a grammar **ambiguous** if a string of terminal symbols can be reached by two different derivation sequences
- In other terms, a string can have more than one parse tree.
- It turns out that our expression grammar is ambiguous!
- Let's show that string $3*5+8$ has two parse trees

Ambiguity

Expr → Expr Op Expr
Expr → Number | Identifier
Identifier → Letter | Letter Identifier
Letter → a-z
Op → "+" | "-" | "*" | "/"
Number → Digit Number | Digit
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



"left parse-tree"



"right parse-tree"

Problems with Ambiguity

- The problem is that the syntax impacts meaning (for the later stages of the compiler)
- For our example string, we'd like to see the left tree because we most likely want * to have a higher precedence than +
- We don't like ambiguity because it makes the parsers difficult to design because we don't know which parse tree will be discovered when there are multiple possibilities
- So we often want to disambiguate grammars.
- It turns out that it is possible to modify grammars to make them non-ambiguous
 - By adding precedence (work for many operators and many precedence relations)
 - By adding non-terminals (eliminate left recursion)
 - By adding/rewriting production rules (left factoring)

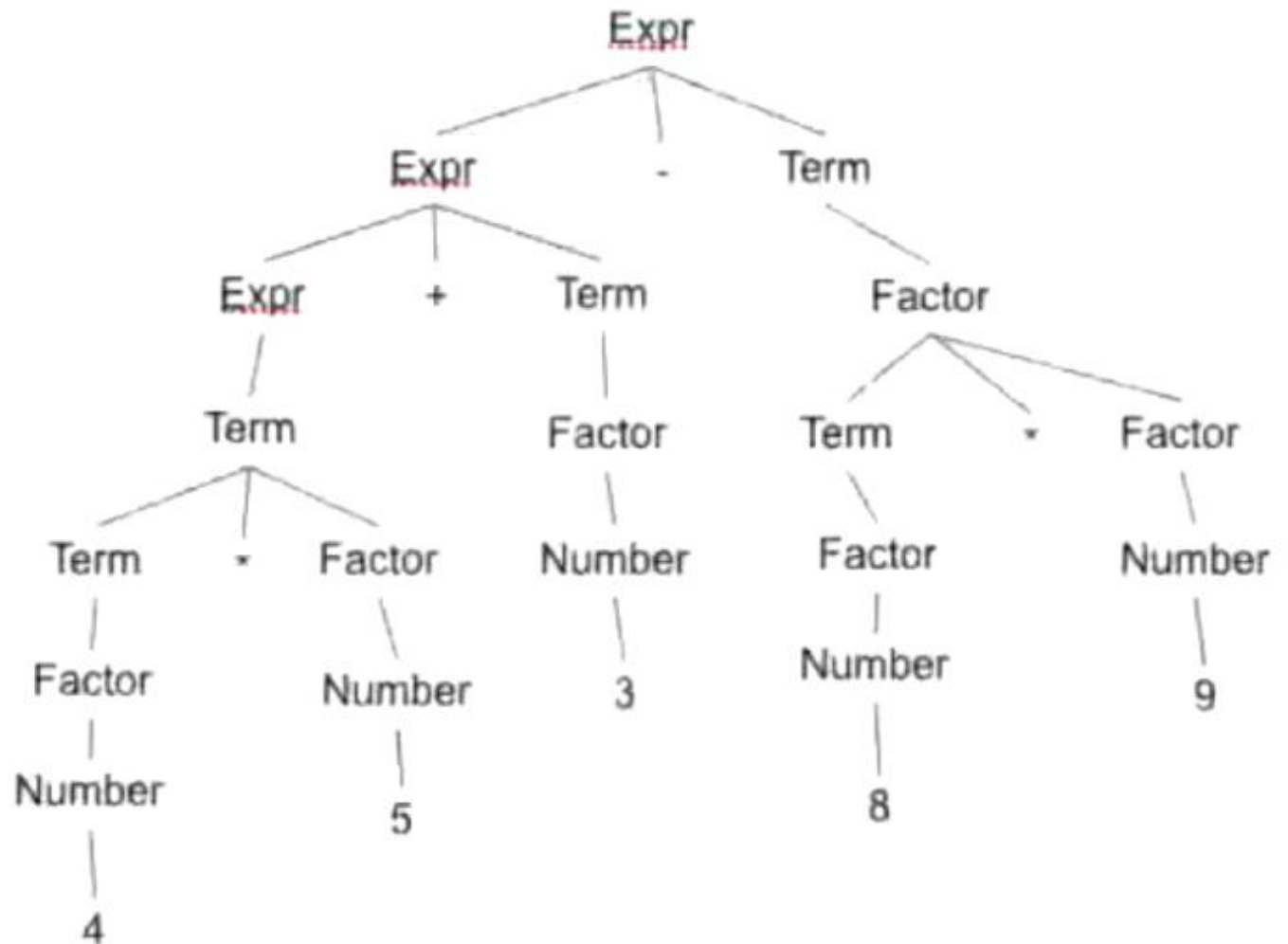
Non-Ambiguous Grammar

Expr \rightarrow Term | Expr + Term | Expr - Term

Term \rightarrow Term * Factor | Term / Factor | Factor

Factor \rightarrow Number | Identifier

Example: $4*5+3-8*9$



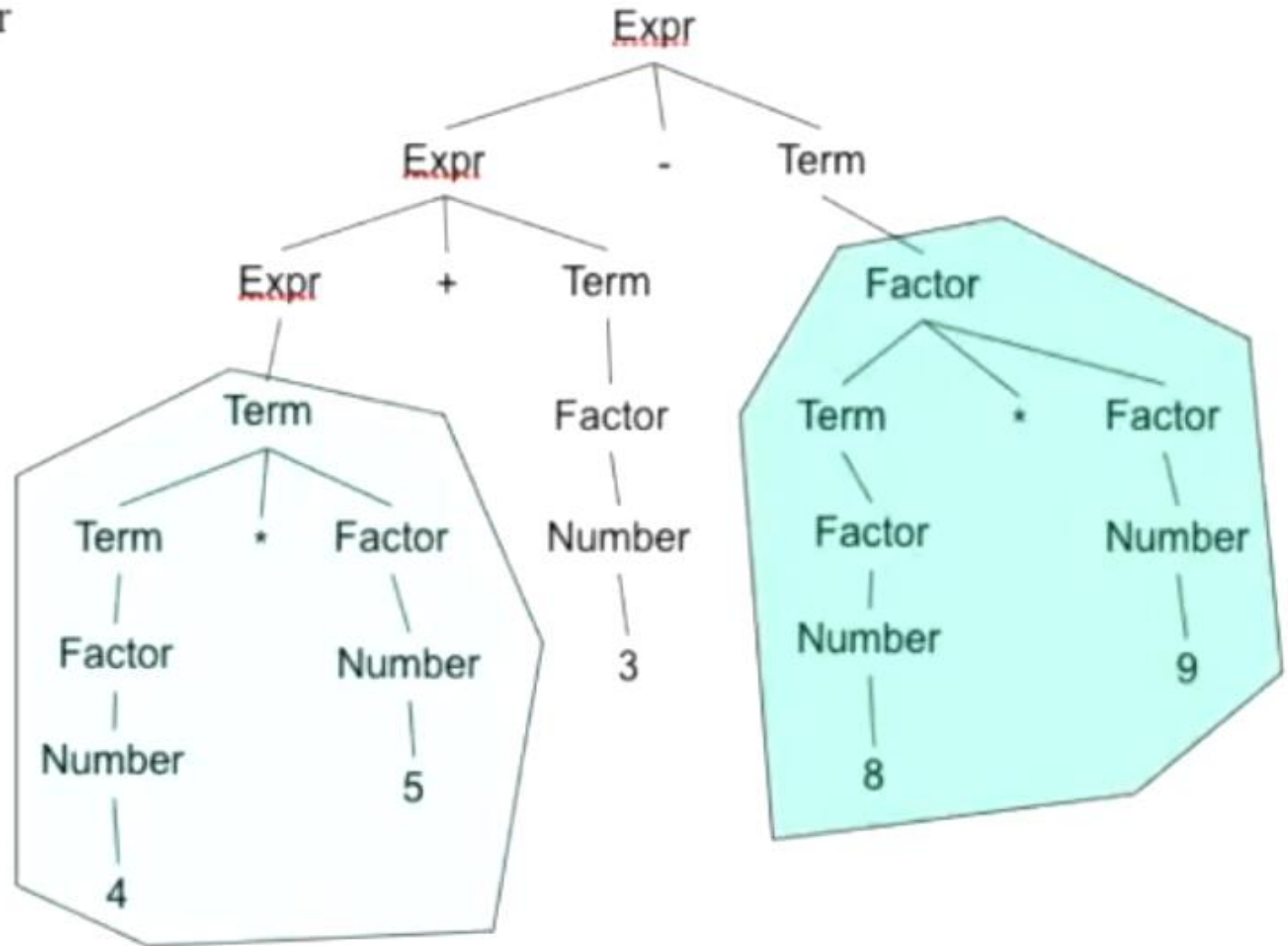
Non-Ambiguous Grammar

Expr \rightarrow Term | Expr + Term | Expr - Term

Term \rightarrow Term * Factor | Term / Factor | Factor

Factor \rightarrow Number | Identifier

Example: 4*5+3-8*9



Exercise

- Consider the CFG:

$$S \rightarrow (L) \mid a$$
$$L \rightarrow L, S \mid S$$

Draw parse trees for:

(a, a)

(a, ((a, a), (a, a)))

Exercise

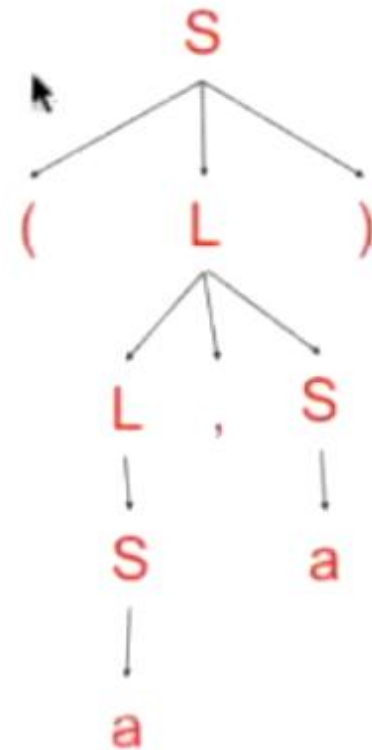
- Consider the CFG:

$$S \rightarrow (L) \mid a$$
$$L \rightarrow L, S \mid S$$

Draw parse trees for:

(a, a)

$(a, ((a, a), (a, a)))$



Exercise

- Consider the CFG:

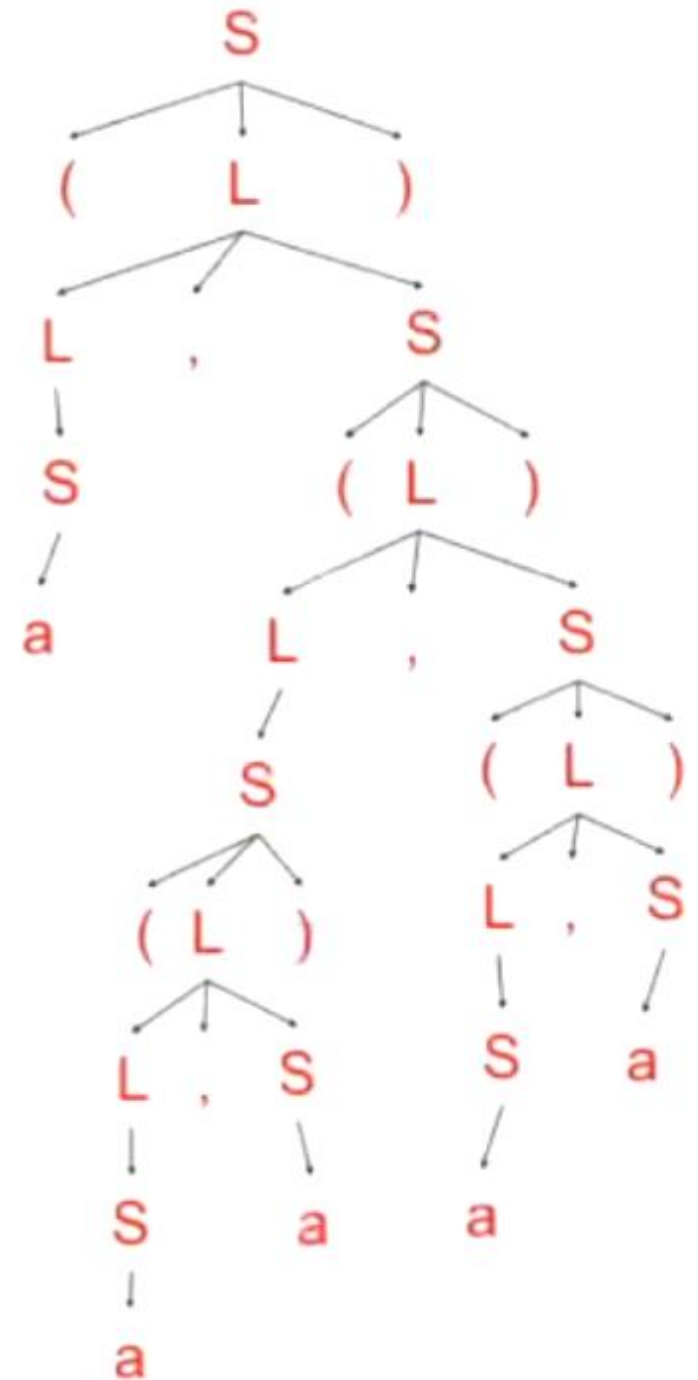
$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

Draw parse trees for:

(a, a)

$(a, ((a, a), (a, a)))$



Exercise

- Write a CFG grammar for the language of well-formed parenthesised expressions
 - $()$, $(())$, $(())()$, $(())()$, etc.: OK
 - $()$, $)()$, $(())$, $((($, etc.: not OK

$P \rightarrow () \mid PP \mid (P)$

Exercise

- Is the following grammar ambiguous?

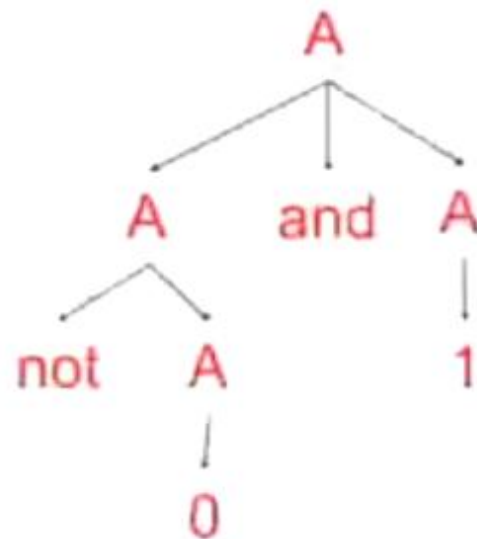
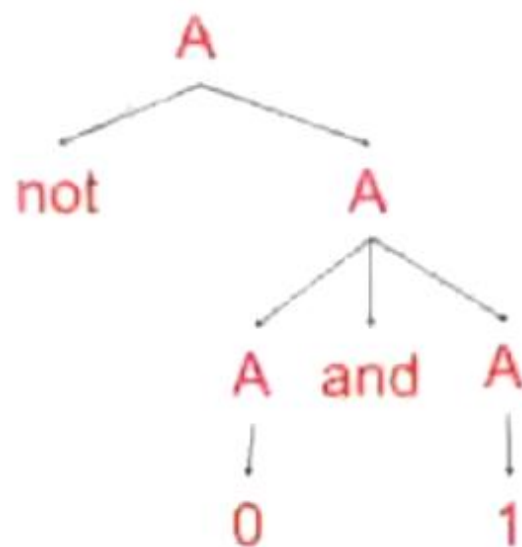
$A \rightarrow A \text{ "and" } A \mid \text{"not" } A \mid \text{"0"} \mid \text{"1"}$



Exercise

- Is the following grammar ambiguous?

$A \rightarrow A \text{ "and" } A \mid \text{not } A \mid 0 \mid 1$



Full Language Grammar Sketch

Program \rightarrow VarDeclList FuncDeclList

VarDeclList $\rightarrow \epsilon \mid$ VarDecl \mid VarDecl VarDeclList

VarDecl \rightarrow Type IdentCommaList “;”

IdentCommaList \rightarrow Ident \mid Ident “,” IdentCommaList

Type \rightarrow int \mid char \mid float

FuncDeclList $\rightarrow \epsilon \mid$ FuncDecl \mid FuncDecl FuncDeclList

FuncDecl \rightarrow Type Ident “(“ ArgList “)” “{“ VarDeclList StmtList “}”

StmtList $\rightarrow \epsilon \mid$ Stmt \mid Stmt StmtList

Stmt \rightarrow Ident “=” Expr “;” \mid ForStatement \mid ...

Expr \rightarrow ...

Ident \rightarrow ...

So What Now?

- We want to write a compiler for a given language
- We come up with a definition of the tokens embodied in regular expressions
- We build a lexer as a DFA (previous class)
- We come up with a definition of the syntax embodied in a context-free grammar
 - not ambiguous
 - enforces relevant operator precedence and associativity
- Question: How do we build a parser?
 - i.e., a program that given an input source file produces a parse tree

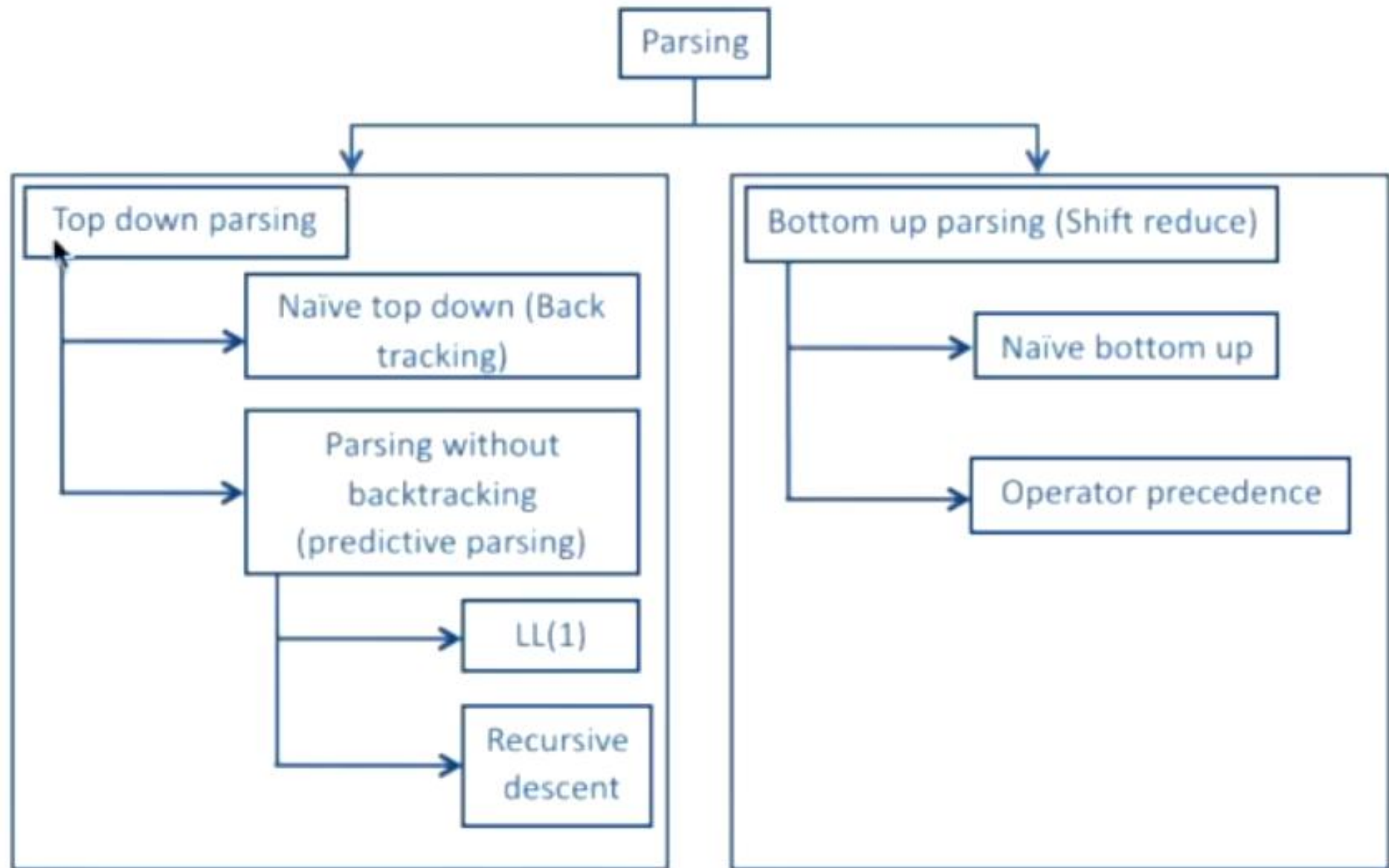
So What Now?

- We want to write a compiler for a given language
- We come up with a definition of the tokens embodied in regular expressions
- We build a lexer as a DFA (previous class)
- We come up with a definition of the syntax embodied in a context-free grammar
 - not ambiguous
 - enforces relevant operator precedence and associativity
- Question: How do we build a parser?
 - i.e., a program that given an input source file produces a parse tree

How do we build a Parser?

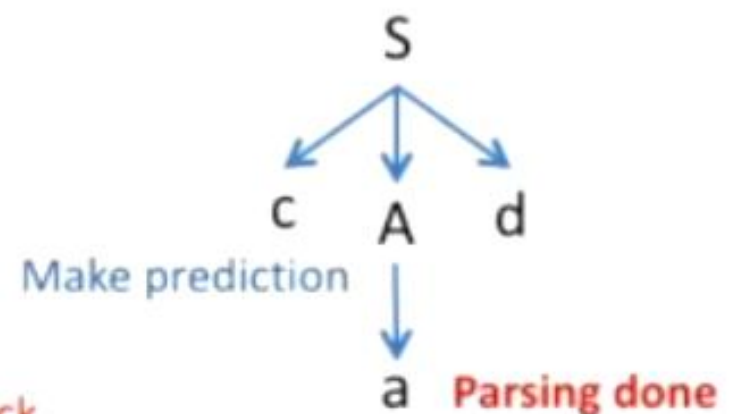
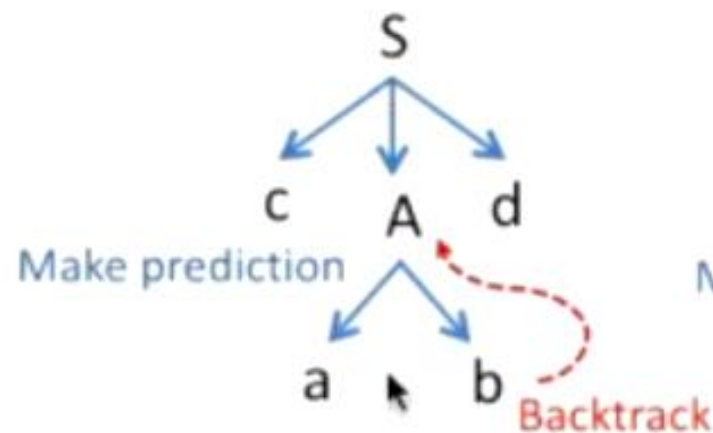
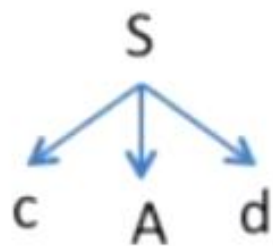
- Here we're going to see a very high-level view of parsing
- There are two approaches for parsing:
 - **Top-Down**: Start with the start symbol and try to expand it using derivation rules until you get the input source code
 - **Bottom-Up**: Start with the input source code, consume symbols, and infer which rules could be used
- Both techniques does not work for all CFGs
 - CFGs must have some properties to be parsable with our parsing algorithms

Classification of parsing methods



Backtracking

- In backtracking, expansion of nonterminal symbol we **choose one alternative** and **if any mismatch occurs** then we **try another alternative**.
- Grammar: $S \rightarrow cAd$ Input string: cad
 $A \rightarrow ab \mid a$



Exercise

Perform backtracking on following grammar:

$$E \rightarrow 3+T \mid 3-T$$

$$T \rightarrow V \mid V*V \mid V+V$$

$$V \rightarrow a \mid b$$

(Input String: 3-a+b)

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down and is the method widely used in practice
- The idea is very simple:
 - Look at the string of tokens, from left to right
 - Look for “things that look like” the right-hand side of production rules
 - Replace the tokens

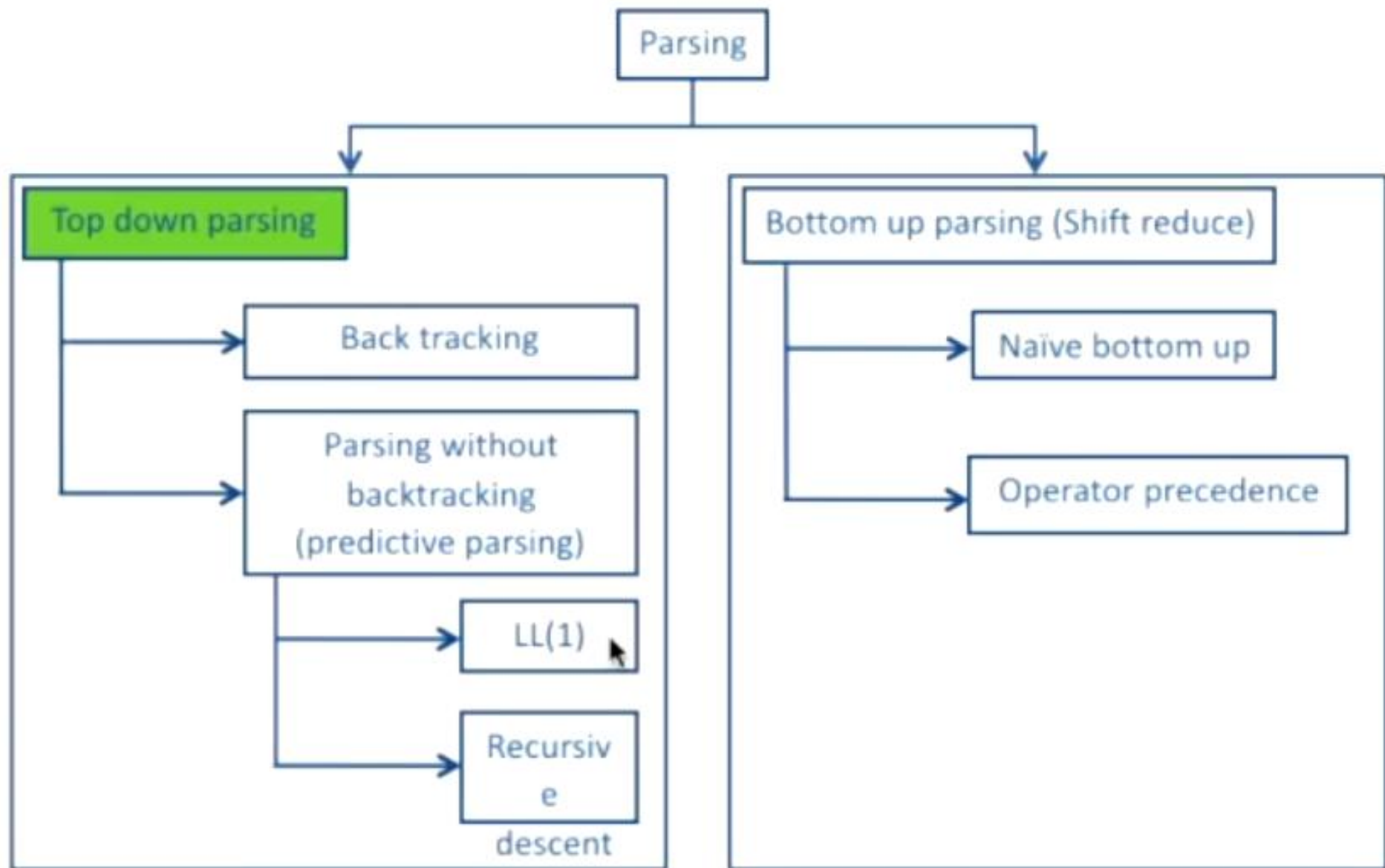
Bottom-Up Parsing Example

- A simple grammar
 - (R1) $\text{Expr} \rightarrow \text{Number} + \text{Expr}$
 - (R2) $\text{Expr} \rightarrow \text{Number}$
 - (R3) $\text{Expr} \rightarrow \text{Number} * \text{Expr}$
 - (R4) $\text{Number} \rightarrow 0-9$
- Let's parse string "3*4+5"
 - Number * 4 + 5 (R4)
 - Number * Number + 5 (R4)
 - Expr + 5 (R3)
 - Expr + Number (R4)
 - Expr (R5) [done]

Bottom-Up Parsing

- The previous example made it look very simple, but this doesn't always work.
- Turns out there is a way to do this (“shift-reduce” parsing) that is guaranteed to work for any non-ambiguous grammar
 - Uses a stack to do some backtracking

Parsing methods



Top-Down Parsing

- A simple recursive algorithm
 - Start with the start symbol
 - Pick one of the rules to expand and expand it
 - If the leftmost symbol is a non-terminal and matches the current token of the input source, great
 - If there is no match, then backtrack and try another rule
 - Repeat for all non-terminal symbols
 - Success if we get all terminals
 - Failure if we've tried all productions without getting all terminals.

Example : Top-Down Parsing

- A simple grammar

(R1) $\text{Expr} \rightarrow \text{Number} + \text{Expr}$

(R2) $\text{Expr} \rightarrow \text{Number}$

(R3) $\text{Expr} \rightarrow \text{Number} * \text{Expr}$

(R4) $\text{Number} \rightarrow 0-9$

- Let's parse string "3*4+5"

(R1) **Number** + Expr

(R4) 3 + Expr [backtrack]

(R2) **Number**

(R4) 3 [backtrack]

(R3) **Number** * Expr

(R4) 3 * Expr

(R1) 3 * **Number** + Expr

(R4) 3 * 4 + Expr

(R1) 3 * 4 + Expr + Expr

(R2) 3 * 4 + **Number** + Expr

(R4) 3 * 4 + 5 + Expr [backtrack]

(R2) 3 * 4 + **Number**

(R4) 3 * 4 + 5 [done!]

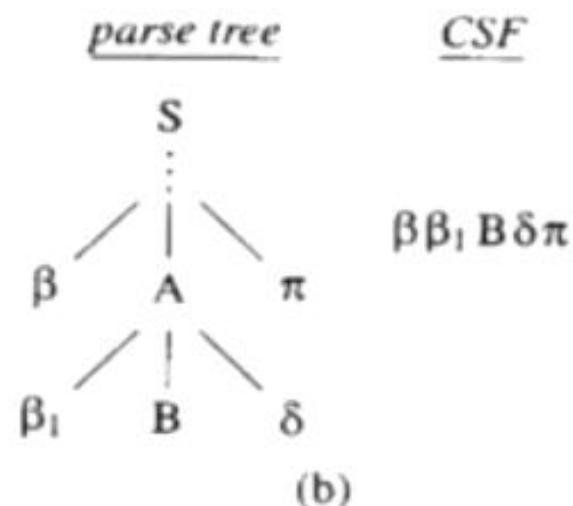
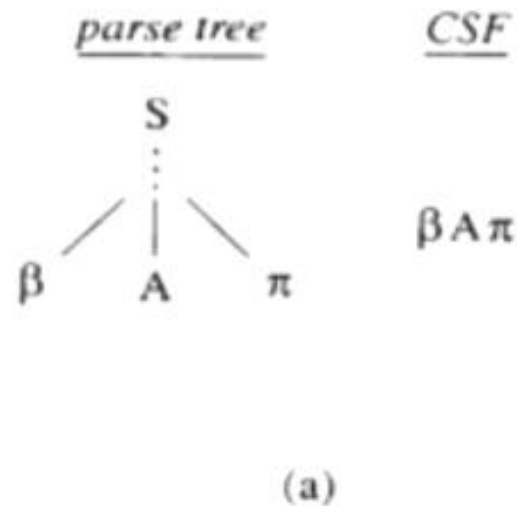
Left-Recursion

- One problem for the Top-Down approach is **left-recursive rules**
- Example: $\text{Expr} \rightarrow \text{Expr} + \text{Number}$
 - The Parser will expand the leftmost Expr as Expr + Number to get: “Expr + Number + Number”
 - And again: “Expr + Number + Number + Number”
 - And again: “Expr + Number + Number + Number + Number”
 - Ad infinitum. . .
 - Since the leftmost symbol is never a non-terminal symbol the parser will never check for a match with the source code and will be stuck in an infinite loop
- Remove left-recursion

Top Down Parsing

- Top down parsing according to a grammar G attempts to derive a string matching a source string through a sequence of derivations starting with the distinguished symbol of G .
- For a valid source string a , a top down parse thus determines a derivation sequence

$$S \rightarrow \dots \rightarrow \dots \rightarrow \alpha$$



Implementing top down parsing

- The following features are needed to implement top down parsing:
 1. Source string marker (SSM): SSM points to the first unmatched symbol in the source string.
 2. Prediction making mechanism: This mechanism systematically selects the RHS alternatives of a production during prediction making. It must ensure that any string in LG can be derived from S.
 3. Matching and backtracking mechanism: This mechanism matches every terminal symbol generated during a derivation with the source symbol pointed to by SSM. (This implements the incremental continuation check.) Backtracking is performed if the match fails. This involves resetting CSF and SSM to earlier values.

Top Down Parsing

- Lexically analyzed version of the source string $a+b*c$, viz. $\langle id \rangle + \langle id \rangle * \langle id \rangle$ is to be parsed according to the grammar
 - $E ::= T + E \mid T$
 - $T ::= V * T \mid V$
 - $V ::= \langle id \rangle$
- The prediction making mechanism selects the RHS alternatives of a production in a left-to-right manner.

Final Steps in the parse :

- Prediction : Predicted Sentential Form

$E \Rightarrow T + E$	$T + E$
$T \Rightarrow V$	$V + E$
$V \Rightarrow \langle id \rangle$	$\langle id \rangle + E$
$E \Rightarrow T$	$\langle id \rangle + T$
$T \Rightarrow V * T$	$\langle id \rangle + V * T$
$V \Rightarrow \langle id \rangle$	$\langle id \rangle + \langle id \rangle * T$
$T \Rightarrow V$	$\langle id \rangle + \langle id \rangle * V$
$V \Rightarrow \langle id \rangle$	$\langle id \rangle + \langle id \rangle * \langle id \rangle$

Top Down Parsing

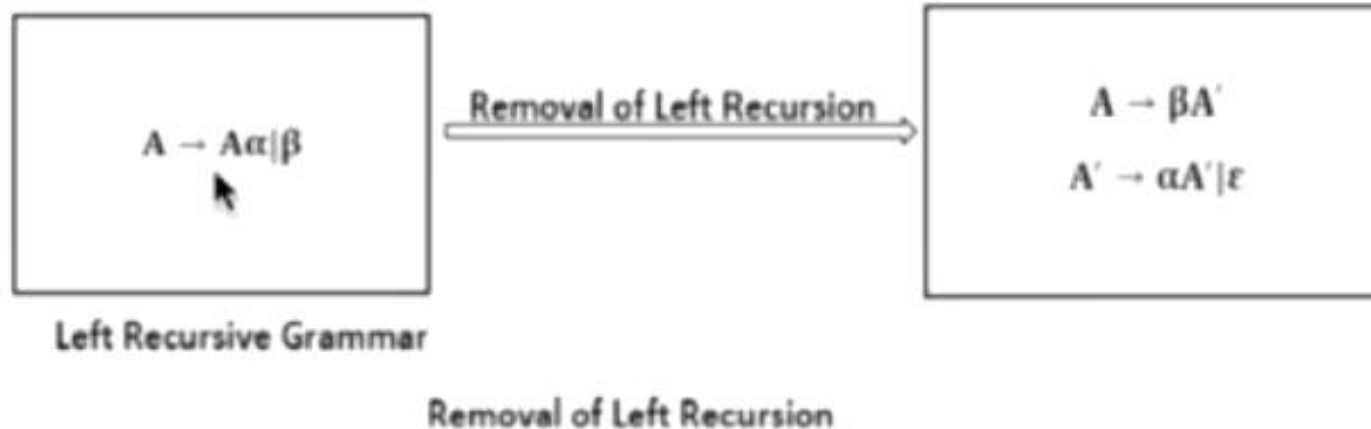
- Two problems arise due to the possibility of backtracking.
 - First, semantic actions cannot be performed while making a prediction. The actions must be delayed until the prediction is known to be a part of a successful parse.
 - Second, precise error reporting is not possible. A mismatch merely triggers backtracking. A source string is known to be erroneous only after all predictions have failed. This makes it impossible to pinpoint the violations of PL specification.
- Grammars containing left recursion are not amenable to top down parsing. For example, consider parsing of the string $\langle \text{id} \rangle + \langle \text{id} \rangle * \langle \text{id} \rangle$ according to the grammar $E = E + T$.
- The first prediction would be $E = E + T$ which makes E the leftmost NT in CSF once again. Thus, the parser would enter an infinite loop of prediction making.

Eliminate left-recursion

- A Grammar G is left recursive if it has a production in the form.
- $A \rightarrow A\alpha \mid \beta$.
- The above Grammar is left recursive because the left of production is occurring at a first position on the right side of production.
- It can eliminate left recursion by replacing a pair of production with
- $A \rightarrow \beta A'$
- $A \rightarrow \alpha A' \mid \epsilon$

Eliminate left-recursion

- Left Recursion can be eliminated by introducing new non-terminal A' .



- This type of recursion is also called Immediate Left Recursion.

Eliminate left-recursion

- In Left Recursive Grammar, expansion of A will generate $A\alpha$, $A\alpha\alpha$, $A\alpha\alpha\alpha$ at each step, causing it to enter into an infinite loop.

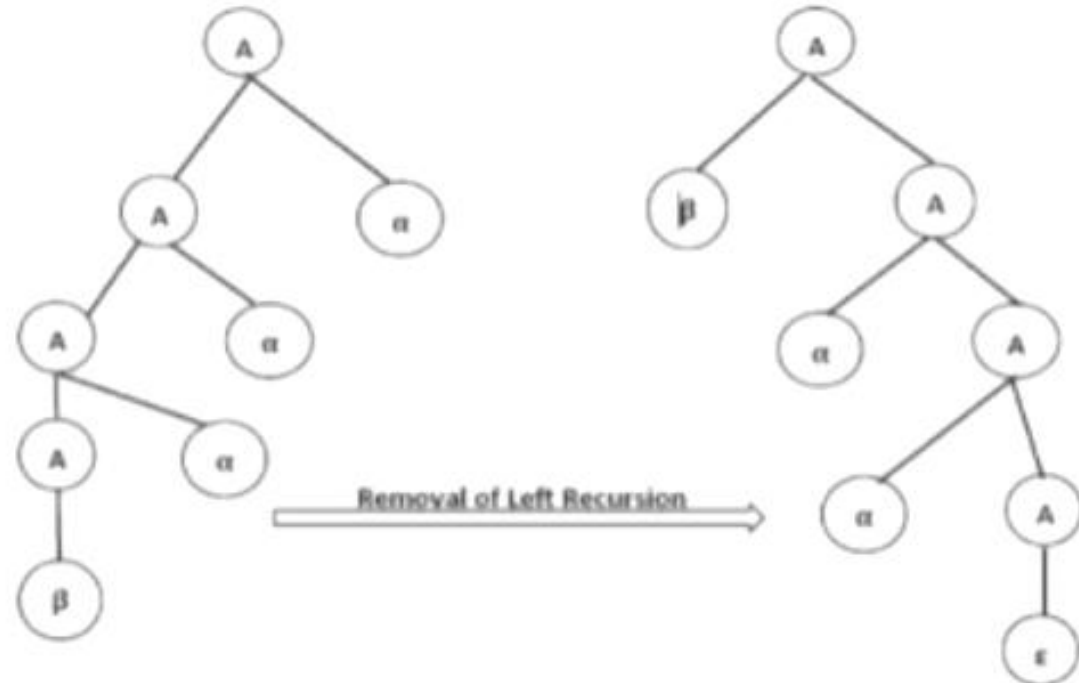
The general form for left recursion is

$$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_m | \beta_1 | \beta_2 | \dots | \beta_n$$

can be replaced by

$$A \rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_n A'$$

$$A \rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_m A' | \epsilon$$



Example : Eliminate Left Recursion

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Eliminate immediate left recursion from the Grammar.

Comparing $E \rightarrow E + T | T$ with $A \rightarrow A \alpha | \beta$

E	\rightarrow	E	+T		T
A	\rightarrow	A	α		β

$\therefore A = E, \alpha = +T, \beta = T$

$\therefore A \rightarrow A \alpha | \beta$ is changed to $A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' | \epsilon$

$\therefore A \rightarrow \beta A'$ means $E \rightarrow TE'$

$A' \rightarrow \alpha A' | \epsilon$ means $E' \rightarrow +TE' | \epsilon$

Comparing $T \rightarrow T * F | F$ with $A \rightarrow A \alpha | \beta$

T	\rightarrow	T	*F		F
A	\rightarrow	A	α		β

$\therefore A = T, \alpha = * F, \beta = F$

$\therefore A \rightarrow \beta A'$ means $T \rightarrow FT'$

$A \rightarrow \alpha A' | \epsilon$ means $T' \rightarrow * FT' | \epsilon$

Production $F \rightarrow (E) | id$ does not have any left recursion

Example : Eliminate Left Recursion

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

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

Eliminate immediate left recursion from the Grammar.

∴ Combining productions 1, 2, 3, 4, 5, we get

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Example : Eliminate Left Recursion

$$S \rightarrow a|(T)$$

$$T \rightarrow T, S|S$$

Solution

We have immediate left recursion in T-productions.

Comparing $T \rightarrow T, S|S$ With $A \rightarrow A\alpha|\beta$ where $A = T$, $\alpha = , S$ and $\beta = S$




Therefore,



Double-click to edit

∴ Complete Grammar will be

 $S \rightarrow a | ^{(T)}$
 $T \rightarrow ST'$
 $T' \rightarrow , ST' \mid \epsilon$

Eliminate left-recursion

- $A \rightarrow Aa \mid b$
- Repeated application of this production builds up sequence of **a**'s to the right of A. When A is finally replaced by **b**.
- We have **b** followed by sequence of zeros or more **a**'s.
- Same effect is achieved by following :

$$A \rightarrow bR$$

$$R \rightarrow aR \mid \epsilon$$

Eliminate left-recursion

- $A \rightarrow Aa \mid b$ $E \rightarrow E+T \mid T$

$$A = E, a = +T, b = T$$

$$A \rightarrow bR$$

$$R \rightarrow aR \mid \varepsilon$$

$$E \rightarrow TE'$$

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

Top down parsing without backtracking **(Predictive Parser)**

- Elimination of backtracking in top down parsing would have several advantages—
 - Parsing would become more efficient.
 - It would be possible to perform semantic actions and precise error reporting during parsing.
 - Prediction making becomes very crucial when backtracking is eliminated.
 - The parser must use some contextual information from the source string to decide which prediction to make for the leftmost NT.

(Predictive Parser)

- The complete rewritten form of Grammar is
 - $E ::= TE''$
 - $E'' ::= +E \mid \varepsilon$
 - $T ::= VT''$
 - $T'' ::= *T \mid \varepsilon$
 - $V ::= \langle \text{id} \rangle$
- Note that grammar does not need left factoring since its RHS alternatives produce unique terminal symbols in the first position.

$$\begin{aligned} E &::= T+E \mid T \\ T &::= V*T \mid V \\ V &::= \langle \text{id} \rangle \end{aligned}$$

Example Parsing of $\langle id \rangle + \langle id \rangle * \langle id \rangle$

- according to Grammar proceeds as shown in Table

<i>Sr. No.</i>	<i>CSF</i>	<i>Symbol</i>	<i>Prediction</i>
1.	E	$\langle id \rangle$	$E \Rightarrow T E''$
2.	$T E''$	$\langle id \rangle$	$T \Rightarrow V T''$
3.	$V T'' E''$	$\langle id \rangle$	$V \Rightarrow \langle id \rangle$
4.	$\langle id \rangle T'' E''$	+	$T'' \Rightarrow \epsilon$
5.	$\langle id \rangle E''$	+	$E'' \Rightarrow + E$
6.	$\langle id \rangle + E$	$\langle id \rangle$	$E \Rightarrow T E''$
7.	$\langle id \rangle + T E''$	$\langle id \rangle$	$T \Rightarrow V T''$
8.	$\langle id \rangle + V T'' E''$	id	$V \Rightarrow \langle id \rangle$
9.	$\langle id \rangle + \langle id \rangle T'' E''$	*	$T'' \Rightarrow * T$
10.	$\langle id \rangle + \langle id \rangle * T E''$	$\langle id \rangle$	$T \Rightarrow V T''$
11.	$\langle id \rangle + \langle id \rangle * V T'' E''$	$\langle id \rangle$	$V \Rightarrow \langle id \rangle$
12.	$\langle id \rangle + \langle id \rangle * \langle id \rangle T'' E''$	-	$T'' \Rightarrow \epsilon$
13.	$\langle id \rangle + \langle id \rangle * \langle id \rangle E''$	-	$E'' \Rightarrow \epsilon$
14.	$\langle id \rangle + \langle id \rangle * \langle id \rangle$	-	-

A recursive descent parser

- A recursive descend (RD) parser is a variant of top down parsing without backtracking.
- It uses a set of recursive procedures to perform parsing.
- Salient advantages of recursive descent parsing are its simplicity and generality. It can be implemented in any language supporting recursive procedures.
- To implement recursive descent parsing, a left-factored grammar is modified to make repeated occurrences of strings more explicit.
- Grammar is rewritten as
 - $E ::= T \{ + T \}^*$
 - $T ::= V \{ * V \}^*$
 - $V ::= \langle \text{id} \rangle$

- **Procedure** proc_E : (tree_root);
- /* This procedure constructs a syntax tree for 'E' and returns a pointer to its root */
- **Var** a, b : pointer to a tree node;
- **begin**
- proc_T (a);
- /* Returns a pointer to the root of tree for T */
- **while** (nextsyms = ' + ') **do**;
- match (' + ');
- **proc_T(b);**
- a := treebuild (' + ' , a, b);
- tree_root := a;
- **return**;
- **end** proc_E;

- **Procedure** proc_T : (tree_root);
- **var** a, b : pointer to a tree node;
- **begin**
- proc_V (a);
- **while** (nextsyms = ' * ') **do**;
- match (' * ');
- **proc_V(b);**
- a := treebuild (' * ' , a, b);
- tree_root := a;
- **return**;
- **end** proc_T;

- **Procedure** proc_V : (tree_root);
- **var** a : pointer to a tree node;
- **begin**
- **if** (nextsyms = < id >) **then**;
- tree_root := treebuild (< id > , - , -);
- **else** print " Error ! ";
- **return**;
- **end** proc_V;

Top-Down Parsing

- The parse tree is created top to bottom.
- Top-down parser
 - Recursive-Descent Parsing
 - It is a general parsing technique, but not widely used.
 - Not efficient
 - Predictive Parsing
 - no backtracking
 - efficient
 - needs a special form of grammars (LL(1) grammars).
 - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
 - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

Abstract Syntax Tree

- The parser accepts *syntactically* correct programs and produces a full parse tree.
- Unfortunately, being syntactically correct is a necessary condition for the program to be correct (i.e., compilable), but it is not sufficient.
- Let's see this on a simple example
- Say we want to write a compiler for the Ada language
- The Ada language requires that procedures be written as:

```
procedure my_func  
...  
end my_func
```

- An incorrect program:

```
procedure my_func  
...  
end some_other_name
```

- Problem: There is no way to express the “both names should be the same” requirement in a CFG!
 - Both are seen as a TOKEN_IDENT token

Attributed Syntax Tree

- To perform such checks we need to associate attributes to nodes in the Syntax Tree and to define rules about these attributes
- You can really see this as adding tons of little pieces of code associated with grammar rules
- Example #1:
 - The Ada Example
 - ProcDecl → procedure Ident
 ProcBody
 end Ident
- Whenever this rule is used, run the piece of code:

```
if (Ident[1].lexeme != Ident[2].lexeme) {  
    fprintf(stderr, "Syntax error: non-matched procedure names\n");  
    exit(1); }
```

Attributed Syntax Tree

- **Example #2 : Type Checking**

Say we have a language in which the body of a function can declare variables

$\text{VarDecl} \rightarrow \text{Type Ident} ";"$

Each time we see this we execute the following code:

`Symbol_Table.insert(Ident.lexeme, type.lexeme);`

(updates some table that keeps track of variables and their types)

$\text{Sum} \rightarrow \text{Ident} "=" \text{Number} + \text{Number}$

Each time we see this we execute the following code:

```
if ((Number[1].type == "float") && (Number[1].type == "float")) {  
    if (Symbol_Table.lookupType(Ident.lexeme) != "float") {  
        fprintf(stderr, "Syntax error: float must be assigned to float\n");  
        exit(1);  
    }  
}
```

LL(1) Parsing

Predictive Parser

A grammar \rightarrow a grammar suitable for predictive parsing (a LL(1) grammar)

eliminate	left
left recursion	factor

- When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

$A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$

input: ... a

|

current token

LL(1) Grammars

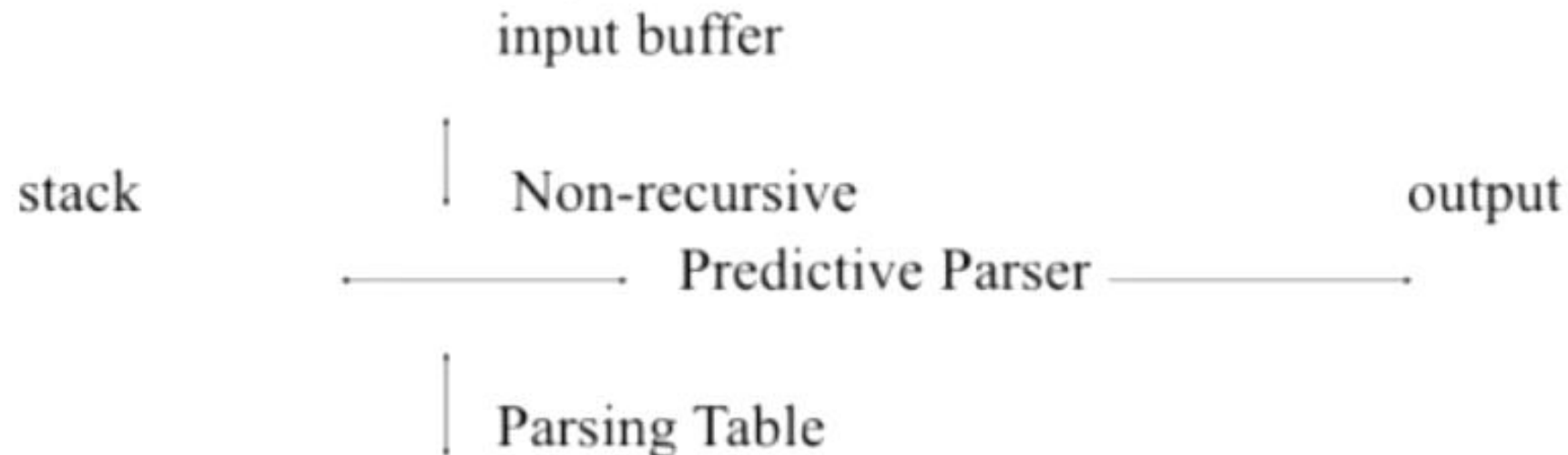
- A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

one input symbol used as a look-head symbol to determine parser action
LL(1) left most derivation
input scanned from left to right

- The parsing table of a grammar may contain more than one production rule. In this case, we say that it is not a LL(1) grammar.

Non-Recursive Predictive Parsing -- LL(1) Parser

- It parses the input from **Left** to right, and constructs a **Leftmost** derivation of the sentence.
- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser.



LL(1) Parser

input buffer

- our string to be parsed. We will assume that its end is marked with a special symbol \$.

output

- a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

stack

- contains the grammar symbols
- at the bottom of the stack, there is a special end marker symbol \$.
- initially the stack contains only the symbol \$ and the starting symbol S.
- SS is initial stack
- when the stack is emptied (ie. only \$ left in the stack), the parsing is completed.

parsing table

- A two-dimensional array $M[A,a]$
- Each entry holds a production rule.

LL(1) Parser – Parser Actions

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
- There are four possible parser actions.
 1. If X and a are $\$$ \rightarrow parser halts (successful completion)
 1. If X and a are the same terminal symbol (different from $\$$)
 \rightarrow parser pops X from the stack, and moves the next symbol in the input buffer.
 3. If X is a non-terminal
 \rightarrow parser looks at the parsing table entry $M[X, a]$. If $M[X, a]$ holds a production rule $X \rightarrow Y_1 Y_2 \dots Y_k$, it pops X from the stack and pushes Y_k, Y_{k-1}, \dots, Y_1 into the stack. The parser also outputs the production rule $X \rightarrow Y_1 Y_2 \dots Y_k$ to represent a step of the derivation.
- none of the above \rightarrow error
 - all empty entries in the parsing table are errors.
 - If X is a terminal symbol different from a , this is also an error case.

LL(1) Parser – Example1

$S \rightarrow aBa$

$B \rightarrow bB \mid \epsilon$

LL(1) Parsing Table

	a	b	\$
S	$S \rightarrow aBa$		
B	$B \rightarrow \epsilon$	$B \rightarrow bB$	

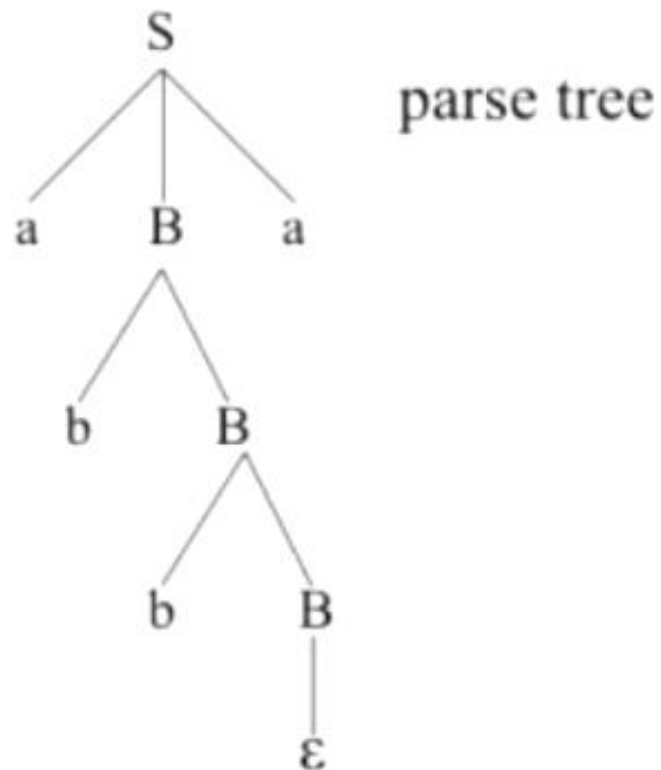
<u>stack</u>	<u>input</u>	<u>output</u>
\$S	abba\$	$S \rightarrow aBa$
\$aBa	abba\$	
\$aB	bba\$	$B \rightarrow bB$
\$aBb	bba\$	
\$aB	ba\$	$B \rightarrow bB$
\$aBb	ba\$	
\$aB	a\$	$B \rightarrow \epsilon$
\$a	a\$	
\$	\$	accept, successful completion

LL(1) Parser – Example1 (cont.)

Outputs: $S \rightarrow aBa$ $B \rightarrow bB$ $B \rightarrow bB$ $B \rightarrow \varepsilon$

Derivation(left-most):

$S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



LL(1) Parser – Example2

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) \mid \text{id}$$

	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

LL(1) Parser – Example2

<u>stack</u>	<u>input</u>	<u>output</u>
SE	$id+id\$$	$E \rightarrow TE'$
$SE'T$	$id+id\$$	$T \rightarrow FT'$
$SE'TF$	$id+id\$$	$F \rightarrow id$
$SE'Tid$	$id+id\$$	
$SE'T$	$+id\$$	$T' \rightarrow \epsilon$
SE'	$+id\$$	$E' \rightarrow +TE'$
$SE'T+$	$+id\$$	
$SE'T$	$id\$$	$T \rightarrow FT'$
$SE'TF$	$id\$$	$F \rightarrow id$
$SE'Tid$	$id\$$	
$SE'T$	$\$$	$T' \rightarrow \epsilon$
SE'	$\$$	$E' \rightarrow \epsilon$
S	$\$$	accept

Constructing LL(1) Parsing Tables

- Two functions are used in the construction of LL(1) parsing tables:
 - FIRST FOLLOW
- **FIRST(α)** is a set of the terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols.
- if α derives to ϵ , then ϵ is also in FIRST(α) .
- **FOLLOW(A)** is the set of the terminals which occur immediately after (follow) the *non-terminal* A in the strings derived from the starting symbol.
 - a terminal a is in FOLLOW(A) if $S \Rightarrow_* \alpha A a \beta$
 - \$ is in FOLLOW(A) if $S \Rightarrow_* \alpha A$

Compute FIRST for Any String X

1. If X is a terminal symbol
 1. $\rightarrow \text{FIRST}(X) = \{X\}$
2. If X is a non-terminal symbol and $X \rightarrow \epsilon$ is a production rule
 1. $\rightarrow \text{FIRST}(X) = \{\epsilon\}$.
3. If X is a non-terminal symbol and $X \rightarrow Y_1 Y_2 \dots Y_n$ is a production rule
 1. \rightarrow if a terminal **a** in $\text{FIRST}(Y_i)$ and ϵ is in all $\text{FIRST}(Y_j)$ for $j=1, \dots, i-1$ then **a** is in $\text{FIRST}(X)$.
 2. \rightarrow if ϵ is in all $\text{FIRST}(Y_j)$ for $j=1, \dots, n$ then ϵ is in $\text{FIRST}(X)$.

LL(1) Grammar

- Find first of following grammar.

Grammar Rule

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) \mid \text{id}$$

FIRST & FOLLOW Set

1. If X is a terminal symbol
 1. $\rightarrow \text{FIRST}(X) = \{X\}$
2. If X is a non-terminal symbol and $X \rightarrow \epsilon$ is a production rule
 1. $\rightarrow \text{FIRST}(X) = \{\epsilon\}$.
3. If X is a non-terminal symbol and $X \rightarrow Y_1 Y_2 \dots Y_n$ is a production rule
 1. \rightarrow if a terminal a in $\text{FIRST}(Y_i)$ and ϵ is in all $\text{FIRST}(Y_j)$ for $j=1, \dots, i-1$ then a is in $\text{FIRST}(X)$.
 2. \rightarrow if ϵ is in all $\text{FIRST}(Y_j)$ for $j=1, \dots, n$ then ϵ is in $\text{FIRST}(X)$.

Grammar Rule

$E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \epsilon$
 $F \rightarrow (E) \mid \text{id}$

FIRST Set

$\text{FIRST}(F) = \{ (, \text{id} \}$ -- Rule 1, 1
 $\text{FIRST}(T') = \{ *, \epsilon \}$ -- Rule 1, 2
 $\text{FIRST}(T) = \{ (, \text{id} \}$ -- Rule 3(1)
 $\text{FIRST}(E') = \{ +, \epsilon \}$ -- Rule 1, 2
 $\text{FIRST}(E) = \{ (, \text{id} \}$ -- Rule 3(1)

FIRST Example

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) \mid \text{id}$$

$$\text{FIRST}(F) = \{ (, \text{id} \}$$

$$\text{FIRST}(T') = \{ *, \varepsilon \}$$

$$\text{FIRST}(T) = \{ (, \text{id} \}$$

$$\text{FIRST}(E') = \{ +, \varepsilon \}$$

$$\text{FIRST}(E) = \{ (, \text{id} \}$$

$$\text{FIRST}(TE') = \{ (, \text{id} \}$$

$$\text{FIRST}(+TE') = \{ + \}$$

$$\text{FIRST}(\varepsilon) = \{ \varepsilon \}$$

$$\text{FIRST}(FT') = \{ (, \text{id} \}$$

$$\text{FIRST}(*FT') = \{ * \}$$

$$\text{FIRST}(\varepsilon) = \{ \varepsilon \}$$

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

$$\text{FIRST}(\text{id}) = \{ \text{id} \}$$

Compute FOLLOW (for non-terminals)

1. If S is the start symbol
 1. $\rightarrow S$ is in $\text{FOLLOW}(S)$

1. if $A \rightarrow \alpha B \beta$ is a production rule
 1. \rightarrow Everything in $\text{FIRST}(\beta)$ is $\text{FOLLOW}(B)$ except ϵ

1. If ($A \rightarrow \alpha B$ is a production rule) or ($A \rightarrow \alpha B \beta$ is a production rule and ϵ is in $\text{FIRST}(\beta)$)
 1. \rightarrow everything in $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$.

We apply these rules until nothing more can be added to any follow set.

LL(1) Grammar

- Find FOLLOW of following grammar.

Grammar Rule

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) \mid \text{id}$$

FOLLOW set

Grammar Rule

1. If S is the start symbol
 1. \rightarrow \$ is in FOLLOW(S)
1. if $A \rightarrow \alpha B \beta$ is a production rule
 1. \rightarrow Everything in FIRST(β) is FOLLOW(B) except ϵ
1. If ($A \rightarrow \alpha B$ is a production rule) or ($A \rightarrow \alpha B \beta$ is a production rule and ϵ is in FIRST(β))
 1. \rightarrow everything in FOLLOW(A) is in FOLLOW(B).

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

FOLLOW Set

$$\text{FOLLOW}(E) = \{ \$,) \} \text{ -- Rule 1, 2}$$

$$\text{FOLLOW}(E') = \{ \$,) \} \text{ -- Rule 3}$$

$$\text{FOLLOW}(T) = \{ +,), \$ \} \text{ -- Rule 2,3,3}$$

$$\text{FOLLOW}(T') = \{ +,), \$ \} \text{ -- Rule 3}$$

$$\text{FOLLOW}(F) = \{ +, *,), \$ \} \text{ -- Rule 3,2,3,3}$$

$$F \rightarrow (E)$$

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

$$T' \rightarrow *FT'$$

FOLLOW Example

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) \mid \text{id}$$

$$\text{FOLLOW}(E) = \{ \$,) \}$$

$$\text{FOLLOW}(E') = \{ \$,) \}$$

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

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

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

FIRST & FOLLOW Set

FIRST Set

Grammar Rule

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

$FIRST(F) = \{ (, id \}$ -- Rule 1 , 1

$FIRST(T') = \{ *, \epsilon \}$ -- Rule 1 , 2

$FIRST(T) = \{ (, id \}$ -- Rule 3

$FIRST(E') = \{ +, \epsilon \}$ -- Rule 1 , 2

$FIRST(E) = \{ (, id \}$ -- Rule 3

FOLLOW Set

$FOLLOW(E) = \{ \$,) \}$ -- Rule 1 , 2

$FOLLOW(E') = \{ \$,) \}$ -- Rule 3

$FOLLOW(T) = \{ +,), \$ \}$ -- Rule 2,3,3

$FOLLOW(T') = \{ +,), \$ \}$ -- Rule 3

$FOLLOW(F) = \{ +, *,), \$ \}$ -- Rule 3,2,3,3

$F \rightarrow (E)$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow *FT'$

FIRST & FOLLOW Set

FIRST Set

$\text{FIRST}(F) = \{ (, id \}$

$\text{FIRST}(T') = \{ *, \varepsilon \}$

$\text{FIRST}(T) = \{ (, id \}$

$\text{FIRST}(E') = \{ +, \varepsilon \}$

$\text{FIRST}(E) = \{ (, id \}$

FOLLOW Set

$\text{FOLLOW}(E) = \{ \$,) \}$

$\text{FOLLOW}(E') = \{ \$,) \}$

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

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

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

Constructing LL(1) Parsing Table -- Algorithm

- for each production rule $A \rightarrow \alpha$ of a grammar G
 1. for each terminal a in $\text{FIRST}(\alpha)$
 \rightarrow add $A \rightarrow \alpha$ to $M[A, a]$
 2. If ϵ in $\text{FIRST}(\alpha)$, then for each terminal b in $\text{FOLLOW}(A)$,
 \rightarrow add $A \rightarrow \alpha$ to $M[A, b]$
 3. If ϵ in $\text{FIRST}(\alpha)$ and $\$$ in $\text{FOLLOW}(A)$
add $A \rightarrow \alpha$ to $M[A, \$]$
- All other undefined entries of the parsing table are error entries.

Constructing LL(1) Parsing Table -- Example

$E \rightarrow TE'$	$FIRST(TE') = \{ (, id \}$	$\boxed{?} E \rightarrow TE' \text{ into } M[E, (] \text{ and } M[E, id]$
$E' \rightarrow +TE'$	$FIRST(+TE') = \{ + \}$	$\boxed{?} E' \rightarrow +TE' \text{ into } M[E', +]$
$E' \rightarrow \epsilon$	$FIRST(\epsilon) = \{ \epsilon \}$ but since ϵ in $FIRST(E')$ and $FOLLOW(E') = \{ \$,) \}$	$\boxed{?}$ none $\boxed{?} E' \rightarrow \epsilon \text{ into } M[E', \$] \text{ and } M[E',)]$
$T \rightarrow FT'$	$FIRST(FT') = \{ (, id \}$	$\boxed{?} T \rightarrow FT' \text{ into } M[T, (] \text{ and } M[T, id]$
$T' \rightarrow *FT'$	$FIRST(*FT') = \{ * \}$	$\boxed{?} T' \rightarrow *FT' \text{ into } M[T', *]$
$T' \rightarrow \epsilon$	$FIRST(\epsilon) = \{ \epsilon \}$ but since ϵ in $FIRST(T')$ and $FOLLOW(T') = \{ \$,), + \}$	$\boxed{?}$ none $\boxed{?} T' \rightarrow \epsilon \text{ into } M[T', \$], M[T',)] \text{ and } M[T', +]$
$F \rightarrow (E)$	$FIRST((E)) = \{ (\}$	$\boxed{?} F \rightarrow (E) \text{ into } M[F, (]$
$F \rightarrow id$	$FIRST(id) = \{ id \}$	$\boxed{?} F \rightarrow id \text{ into } M[F, id]$

Constructing LL(1) Parsing Table -- Example

$E \rightarrow TE'$	$\text{FIRST}(TE') = \{ (, \text{id} \}$	$\boxed{?} E \rightarrow TE' \text{ into } M[E, (] \text{ and } M[E, \text{id}]$
$E' \rightarrow +TE'$	$\text{FIRST}(+TE') = \{ + \}$	$\boxed{?} E' \rightarrow +TE' \text{ into } M[E', +]$
$E' \rightarrow \epsilon$	$\text{FIRST}(\epsilon) = \{ \epsilon \}$ but since ϵ in $\text{FIRST}(E')$ and $\text{FOLLOW}(E') = \{ \$,) \}$	$\boxed{?}$ none $\boxed{?} E' \rightarrow \epsilon \text{ into } M[E', \$] \text{ and } M[E',)]$
$T \rightarrow FT'$	$\text{FIRST}(FT') = \{ (, \text{id} \}$	$\boxed{?} T \rightarrow FT' \text{ into } M[T, (] \text{ and } M[T, \text{id}]$
$T' \rightarrow *FT'$	$\text{FIRST}(*FT') = \{ * \}$	$\boxed{?} T' \rightarrow *FT' \text{ into } M[T', *]$
$T' \rightarrow \epsilon$	$\text{FIRST}(\epsilon) = \{ \epsilon \}$ but since ϵ in $\text{FIRST}(T')$ and $\text{FOLLOW}(T') = \{ \$,), + \}$	$\boxed{?}$ none $\boxed{?} T' \rightarrow \epsilon \text{ into } M[T', \$], M[T',)] \text{ and } M[T', +]$
$F \rightarrow (E)$	$\text{FIRST}((E)) = \{ (\}$	$\boxed{?} F \rightarrow (E) \text{ into } M[F, (]$
$F \rightarrow \text{id}$	$\text{FIRST}(\text{id}) = \{ \text{id} \}$	$\boxed{?} F \rightarrow \text{id} \text{ into } M[F, \text{id}]$

LL(1) Parser – Example2



<u>stack</u>	<u>input</u>	<u>output</u>
\$E	id+id\$	$E \rightarrow TE'$
\$E'T	id+id\$	$T \rightarrow FT'$
\$E'TF	id+id\$	$F \rightarrow id$
\$E'Tid	id+id\$	
\$E'T	+id\$	$T' \rightarrow \epsilon$
\$E'	+id\$	$E' \rightarrow +TE'$
\$E'T+	+id\$	
\$E'T	id\$	$T \rightarrow FT'$
\$E'TF	id\$	$F \rightarrow id$
\$E'Tid	id\$	
\$E'T	\$	$T' \rightarrow \epsilon$
\$E'	\$	$E' \rightarrow \epsilon$
\$	\$	accept

Example

- Is following grammar LL(1) ?

$S \rightarrow i C t S E \mid a$

$E \rightarrow e S \mid \varepsilon$

$C \rightarrow b$

A Grammar which is not LL(1)

$S \rightarrow i C t S E \mid a$

$E \rightarrow e S \mid \epsilon$

$C \rightarrow b$

$\text{FOLLOW}(S) = \{ \$, e \}$

$\text{FOLLOW}(E) = \{ \$, e \}$

$\text{FOLLOW}(C) = \{ t \}$

$\text{FIRST}(iCtSE) = \{ i \}$

$\text{FIRST}(a) = \{ a \}$

$\text{FIRST}(eS) = \{ e \}$

$\text{FIRST}(\epsilon) = \{ \epsilon \}$

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

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iCtSE$		
E			$E \rightarrow e S$ $E \rightarrow \epsilon$			$E \rightarrow \epsilon$
C		$C \rightarrow b$				

two production rules for $M[E, e]$

Problem \rightarrow ambiguity

A Grammar which is not LL(1) (cont.)

- What do we have to do if the resulting parsing table contains multiply defined entries?
 - If we didn't eliminate left recursion, eliminate the left recursion in the grammar.
 - If the grammar is not left factored, we have to left factor the grammar.
 - If its (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.
- A left recursive grammar cannot be a LL(1) grammar.
 - $A \rightarrow A\alpha \mid \beta$
 - ☐ any terminal that appears in $\text{FIRST}(\beta)$ also appears in $\text{FIRST}(A\alpha)$ because $A\alpha \Rightarrow \beta\alpha$.
 - ☐ If β is ϵ , any terminal that appears in $\text{FIRST}(\alpha)$ also appears in $\text{FIRST}(A\alpha)$ and $\text{FOLLOW}(A)$.
- A grammar is not left factored, it cannot be a LL(1) grammar
 - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
 - ☐ any terminal that appears in $\text{FIRST}(\alpha\beta_1)$ also appears in $\text{FIRST}(\alpha\beta_2)$.
- An ambiguous grammar cannot be a LL(1) grammar.

Properties of LL(1) Grammars

- A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$
 1. Both α and β cannot derive strings starting with same terminals.
 1. At most one of α and β can derive to ϵ .
 1. If β can derive to ϵ , then α cannot derive to any string starting with a terminal in FOLLOW(A).

Error Recovery in Predictive Parsing

- An error may occur in the predictive parsing (LL(1) parsing)
 - if the terminal symbol on the top of stack does not match with the current input symbol.
 - if the top of 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 an error case?
 - The parser should be able to give an error message (as much as possible meaningful error message).
 - It should be able to recover from that error case, and it should be able to continue the parsing with the rest of the input.

Error Recovery Techniques

- **Panic-Mode Error Recovery**
 - Skipping the input symbols until a synchronising token is found.
- **Phrase-Level Error Recovery**
 - Each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case.
- **Error-Productions**
 - If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
 - When an error production is used by the parser, we can generate appropriate error diagnostics.
 - Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.
- **Global-Correction**
 - Ideally, we would like a compiler to make as few change as possible in processing incorrect inputs.
 - We have to globally analyse the input to find the error.
 - This is an expensive method, and it is not in practice.

Panic-Mode Error Recovery in LL(1) Parsing

- In panic-mode error recovery, we skip all the input symbols until a synchronising token is found.
- What is the synchronising token?
 - All the terminal-symbols in the follow set of a non-terminal can be used as a synchronising token set for that non-terminal.
- So, a simple panic-mode error recovery for the LL(1) parsing:
 - All the empty entries are marked as *synch* to indicate that the parser will skip all the input symbols until a symbol in the follow set of the non-terminal A which on the top of the stack. Then the parser will pop that non-terminal A from the stack. The parsing continues from that state.
 - To handle unmatched terminal symbols, the parser pops that unmatched terminal symbol from the stack and it issues an error message saying that that unmatched terminal is inserted.

Phrase-Level Error Recovery

- Each empty entry in the parsing table is filled with a pointer to a special error routine which will take care that error case.
- These error routines may:
 - change, insert, or delete input symbols.
 - issue appropriate error messages
 - pop items from the stack.
- We should be careful when we design these error routines, because we may put the parser into an infinite loop.

Example - Error Recovery

$S \rightarrow AbS \mid e \mid \epsilon$

$A \rightarrow a \mid cAd$

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

$FOLLOW(A) = \{b, d\}$

	a	b	c	d	e	\$
S	$S \rightarrow AbS$	<i>sync</i>	$S \rightarrow AbS$	<i>sync</i>	$S \rightarrow e$	$S \rightarrow \epsilon$
A	$A \rightarrow a$	<i>sync</i>	$A \rightarrow cAd$	<i>sync</i>	<i>sync</i>	<i>sync</i>

<u>stack</u>	<u>input</u>	<u>output</u>
SS	aab\$	$S \rightarrow AbS$
SSbA	aab\$	$A \rightarrow a$
SSba	aab\$	
SSb	ab\$	Error: missing b, inserted
SS	ab\$	$S \rightarrow AbS$
SSbA	ab\$	
SSba	ab\$	
SSb	b\$	
SS	\$	$S \rightarrow \epsilon$
\$	\$	accept

<u>stack</u>	<u>input</u>	<u>output</u>
SS	ceadb\$	$S \rightarrow AbS$
SSbA	ceadb\$	$A \rightarrow cAd$
SSbdAc	ceadb\$	
SSbdA	eadb\$	Error: unexpected e (illegal A)
(Remove all input tokens until first b or d, pop A)		
$A \rightarrow a$	SSbd	db\$
SSb	b\$	
SS	\$	$S \rightarrow \epsilon$
\$	\$	accept