Syntax Analysis
2nd Phase of Compiler Construction

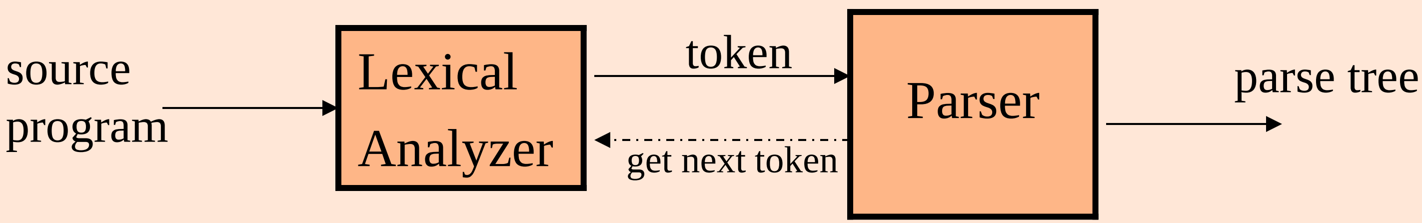
Section 2.1: Context Free Grammar

Syntax Analyzer

- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.
  - If it satisfies, the parser creates the parse tree of that program.
  - Otherwise the parser gives the error messages.
- It creates the syntactic structure of the given source program.
- This syntactic structure is mostly a *parse tree*.
- Syntax Analyzer is also known as *parser*.
- The syntax of a programming is described by a *context-free grammar (CFG).*
- A context-free grammar
  - gives a precise syntactic specification of a programming language.
  - the design of the grammar is an initial phase of the design of a compiler.
  - a grammar can be directly converted into a parser by some tools.

PARSER

- Parser works on a stream of tokens.
- The smallest item is a token.



PARSERS (CONT.)

- We categorize the parsers into two groups:

1. **Top-Down Parser**
   - Parse-trees built is build from root to leaves (top to bottom).
   - Input to parser is scanned from left to right one symbol at a time
2. **Bottom-Up Parser**
   - Start from leaves and work their way up to the root.
   - Input to parser scanned from left to right one symbol at a time

- Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
  - LL for top-down parsing
  - LR for bottom-up parsing

WHY DO WE NEED A GRAMMAR?

Grammar defines a Language.

There are some rules which need to be followed to express or define a language.

These rules are laid down in the form of Production rules (P).

Context-free grammar (CFG) is used to generate a language called Context Free Language (L)

Context-Free Grammars (CFG)

CFG G consist of 4 symbol (T,V, S, P):

- **T**: A finite set of terminals
- **V**: A finite set of non-terminals (also denoted by N)
- **S**: A start symbol (Non-terminal symbol with which the grammar starts)
- **P**: A finite set of productions rules

CONTEXT-FREE GRAMMARS (CFG)

Consider the Grammar:

$S \rightarrow aAa/b$

$A \rightarrow a$

G = (T,V, S, P)

{a, b}

S, A

$S \rightarrow aAa$

$S \rightarrow b$

$A \rightarrow a$

TERMINALS SYMBOLS

Terminals include:

- **Lower case letters early in the alphabets**
- **Operator symbols, +, %**
- **Punctuation symbols such as ( ) , ;**
- **Digits 0,1,2, …**
- Boldface strings **id or if**

Consider the Grammar:

S→ aAa

S→ b;c

A→ aA/ **ε**

Here Terminal Symbols are {a, b, c, ; , ε}

NON TERMINALS SYMBOLS

Non - Terminals include:

- **Uppercase letters early in the alphabet**
- **The letter S, start symbol**
- **Lower case italic names such as *expr* or *stmt***

Consider the Grammar:

S→ aAa
S→ bB
A→ aA/ ε
B→ b

Here Non- Terminal Symbols are {A, B, S}

PRODUCTION RULES

Production Rules include:

- **Set of Rules which define the grammar G**

Consider the Grammar:

$S \rightarrow aAa$

$A \rightarrow aA / \mathbf{a}$

Here we have three production rules

i. $S \rightarrow aAa$

ii. $A \rightarrow aA$

iii. $A \rightarrow a$

DERIVATION OF A STRING

String 'w' of terminals is *generated* by the grammar if:

Starting with the start variable, one can apply productions and end up with 'w'.

A sequence of replacements of non-terminal symbols or a sequence of strings so obtained is a ***derivation*** of 'w'.

Consider the Grammar:

S→ aAa

A→ aA/ **a**

We can derive sentence 'aaa' from this grammar.

S→aAa
S→ aaa (A→a)

DERIVATION OF A STRING

In general a derivation step is:

$\alpha A\beta \Rightarrow \alpha\gamma\beta$ if there is a production rule $A\rightarrow\gamma$ in a grammar
where α and β are arbitrary strings of terminal and non-terminal symbols

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)

$\Rightarrow$: derives in one step

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

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

DERIVATION OF A STRING

Consider the Grammar:

S→ aSa/b/aA

A→ **a**

Derived in one step	⇒	**S→b**
Derived in two steps	⇒	**S→aSa → aba**
Derived in multiple steps	⇒	**S→aSa → aaSaa→aaaSaaa→aaabaaa**

SENTENCE AND SENTENTIAL FORM

A **sentence** of L(G) is a string of terminal symbols only.

A **sentential form** is a combination of terminals and non-terminals.

Say, we have a production

$$S \Rightarrow \alpha$$

If α contains non-terminals, it is called as a *sentential* form of G.

$$S \rightarrow aSa \rightarrow aaSaa \rightarrow aaaSaaa \rightarrow \text{aaabaaa}$$

If α does not contain non-terminals, it is called as a *sentence* of G.

Left-Most and Right-Most Derivations

We can derive the grammar in two ways:

- Left-Most Derivation
- Right- Most Derivation

In **Left Most Derivation** , we start deriving the string 'w' from the left side and convert all non terminals into terminals.

In **Right Most Derivation**, we start deriving the string 'w' from the right side and convert all non terminals into terminals.

Left-Most Derivations

Consider the Grammar:

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

Derive the string ‘id+id *id’

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

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

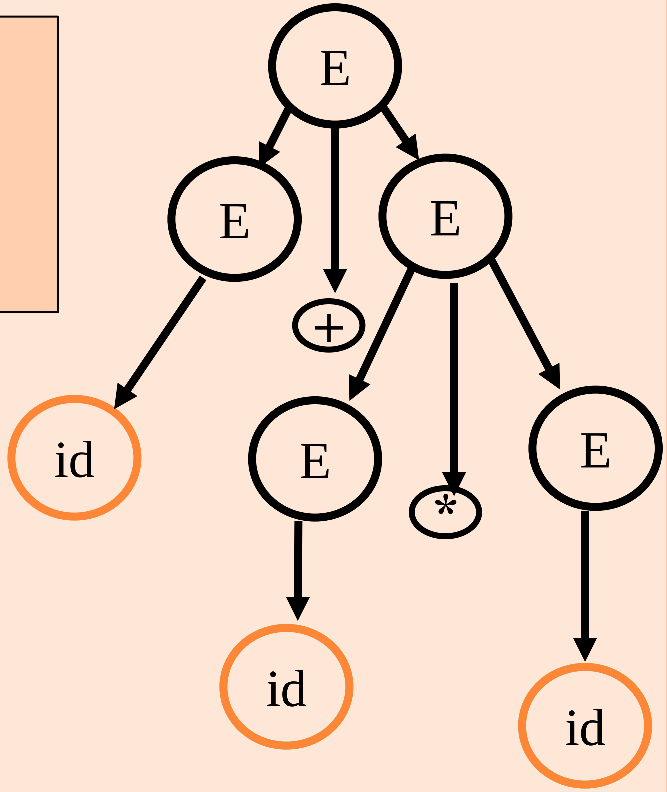
PARSE TREE FOR LEFT-MOST DERIVATIONS

Consider the Grammar:

E→ E+E/E-E/E*E/E/(E)/id

Derive the string 'id+id *id'

E→E+E	(E→E+E)
E→id+E	(E→id)
E→id+E*E	(E→E*E)
E→id+id*E	(E→id)
E→id+id*id	(E→id)



RIGHT-MOST DERIVATIONS

Consider the Grammar:

E→ E+E/E-E/E*E/E/(E)/id

Derive the string ‘id+id *id’

E→E*E	(E→E*E)
E→E*id	(E→id)
E→E+E*id	(E→E+E)
E→E+id*id	(E→id)
E→id+id*id	(E→id)

~~E→E*E~~	~~(E→E+E)~~
~~E→E+E*E~~	~~(E→E+E)~~
~~E→E+E*id~~	~~(E→id)~~
~~E→E+id*id~~	~~(E→id)~~
~~E→id+id*id~~	~~(E→id)~~

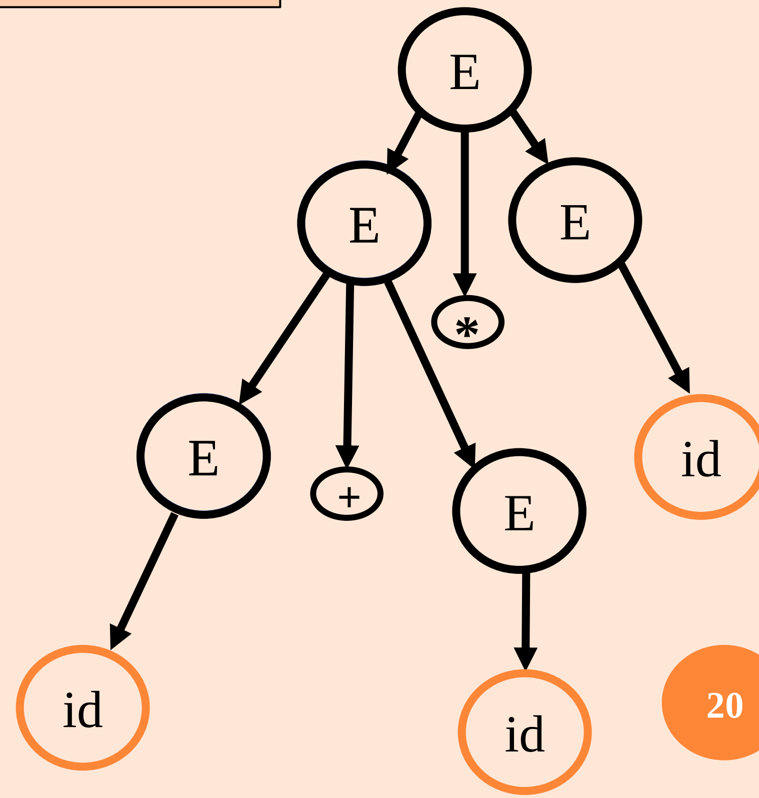
RIGHT-MOST DERIVATIONS

Consider the Grammar:

E→ E+E/E-E/E*E/E/(E)/id

Derive the string 'id+id *id'

E→E*E	(E→E+E)
E→E*id	(E→id)
E→E+E*id	(E→E+E)
E→E+id*id	(E→id)
E→id+id*id	(E→id)



Section 2.2: Ambiguous Grammar

AMBIGUOUS GRAMMAR

A grammar is Ambiguous if it has:

More than one left most or more than one right most derivation for a given sentence *i.e.* it can be derived by more then one ways from LMD or RMD.

Consider the Grammar:

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

Derive the string 'id+id *id'

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

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

More than one leftmost derivations

Ambiguous Grammar

AMBIGUOUS GRAMMAR

A grammar is Ambiguous if it has:

More than one left most or more than one right most derivation for a given sentence *i.e.* it can be derived by more then one ways from LMD or RMD.

Consider the Grammar:

E→ E+E/E-E/E*E/E/(E)/id

Derive the string ‘id+id *id’

E→E*E	(E→E*E)
E→E*id	(E→id)
E→E+E*id	(E→E+E)
E→E+id*id	(E→id)
E→id+id*id	(E→id)

E→E+E	(E→E+E)
E→E+E*E	(E→E*E)
E→E+E*id	(E→id)
E→E+id*id	(E→id)
E→id+id*id	(E→id)

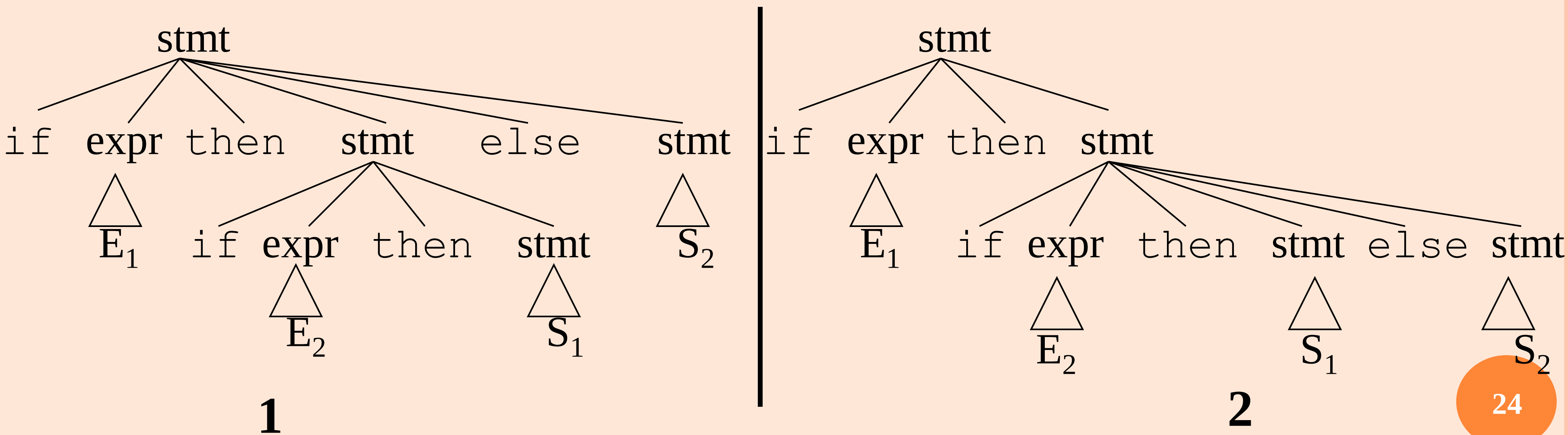
More than one rightmost derivations

Ambiguous Grammar

AMBIGUITY (CONT.)

stmt → `if` expr `then` stmt |
`if` expr `then` stmt `else` stmt | otherstmts

`if` E_1 `then` `if` E_2 `then` S_1 `else` S_2



Ambiguity (cont.)

- We prefer the second parse tree (else matches with closest if).
- So, we have to disambiguate our grammar to reflect this choice.

- The unambiguous grammar will be:

stmt → matchedstmt | unmatchedstmt

matchedstmt → `if` expr `then` matchedstmt `else` matchedstmt | otherstmts

unmatchedstmt → `if` expr `then` stmt |
`if` expr `then` matchedstmt `else` unmatchedstmt

Section 2.3: Left recursion and left factoring

Left Recursion

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

$$A \overset{+}{\Rightarrow} A\alpha \qquad \text{for some string } \alpha$$

- Top-down parsing techniques **cannot** handle left-recursive grammars.
- So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.
- 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.

Immediate Left-Recursion

$$\boxed{A \rightarrow A\,\alpha \mid \beta}$$

where β does not start with A

Eliminate immediate left recursion $\Rightarrow$

$$\boxed{\begin{array}{l} A \rightarrow \beta\, A' \\ A' \rightarrow \alpha\, A' \mid \varepsilon \end{array}}$$

An equivalent grammar

In general,

$$\boxed{A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n \text{ where } \beta_1 \ldots \beta_n \text{ do not start with } A}$$

$\Downarrow$ **Eliminate immediate left recursion**

$$\boxed{\begin{array}{l} A \rightarrow \beta_1\, A' \mid \ldots \mid \beta_n\, A' \\ A' \rightarrow \alpha_1\, A' \mid \ldots \mid \alpha_m\, A' \mid \varepsilon \end{array}}$$

an equivalent grammar

REMOVING IMMEDIATE LEFT-RECURSION

$A \rightarrow A\ \alpha |\ \beta \longrightarrow \begin{cases} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' | \varepsilon \end{cases}$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

Immediate Left Recursion In

E→E+T|T

T→T*F|F

No Immediate left recursion in

F→ id|(E)

E→E+T|T (A→A α| β)

A is E; α is +T and β is T

Applying Rule we get

E → T E' (A → β A ')

E' → +T E' | ε (A ' → α A '| ε)

T→T*F|F (A→A α| β)

A is T; α is *F and β is F

Applying Rule we get

T → F T' (A → β A')

T' → *F T' | ε (A' → α A'|ε)

E → T E'

E' → +T E' | ε

T → F T'

T' → *F T' | ε

F → id | (E)

Final Output

No immediate Left-Recursion but grammar is left recursive

Consider the Grammar

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$

No Immediate left recursion in the grammar

Substitution

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$

or

$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$

Immediate left recursion in the grammar

We need to check and eliminate both Immediate left recursion and Left recursion

No immediate Left-Recursion but grammar is left recursive

Consider the Grammar
$S \to Aa \mid b$
$A \to Ac \mid Sd \mid f$

No Immediate left recursion in S

Order of non-terminals: S, A
for S:
- there is no immediate left recursion in S.

Substitute A→Sd with Aad|bd

$S \to Aa \mid b$
$A \to Ac \mid Aad \mid bd \mid f$

Immediate left recursion in A

α_1 is c; α_2 is ad; β_1 is bd and β_2 is f

Applying Rule

$A \to A\alpha \mid \beta \Longrightarrow \begin{cases} A \to \beta A' \\ A' \to \alpha A' \mid \varepsilon \end{cases}$

We get:
$A \to bdA' \mid fA'$
$A' \to cA' \mid adA' \mid \varepsilon$

$S \to Aa \mid b$
$A \to bdA' \mid fA'$
$A' \to cA' \mid adA' \mid \varepsilon$

Final Output

No immediate Left-Recursion but grammar is left recursive

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

$\Downarrow$

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

Consider the Grammar
$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid f$

Order of non-terminals: A, S

for A:

Eliminate the immediate left-recursion in A

$$A \rightarrow SdA' \mid fA'$$
$$A' \rightarrow cA' \mid \varepsilon$$

$A \rightarrow Ac \mid Sd \mid f$
α is c; β_1 is Sd and β_2 is f

for S:

- Replace $S \rightarrow Aa$ with $S \rightarrow SdA'a \mid fA'a$

So, we will have $S \rightarrow SdA'a \mid fA'a \mid b$

Eliminate the immediate left-recursion in S

$$S \rightarrow fA'aS' \mid bS'$$
$$S' \rightarrow dA'aS' \mid \varepsilon$$

$S \rightarrow SdA' \mid fA'a \mid b$
α is dA' a; β_1 is fA'a and β_2 is b

Final Output

$$S \rightarrow fA'aS' \mid bS'$$
$$S' \rightarrow dA'aS' \mid \varepsilon$$
$$A \rightarrow SdA' \mid fA'$$
$$A' \rightarrow cA' \mid \varepsilon$$

Practice Question: Left recursion

Remove the left recursion from the grammar given below

A → B **x y** | **x**
B → C D
C → A | **c**
D → **d**

Eliminate Left-Recursion -- Algorithm

- Arrange non-terminals in some order: $A_1 \dots A_n$

- for i **from** 1 **to** n **do** {

  - **for** j **from** 1 **to** i-1 **do** {

    replace each production

    $$A_i \rightarrow A_j \gamma$$

    by

    $$A_i \rightarrow \alpha_1 \gamma \mid \dots \mid \alpha_k \gamma$$

    where $A_j \rightarrow \alpha_1 \mid \dots \mid \alpha_k$

  }

  - eliminate immediate left-recursions among A_i productions

}

LEFT-FACTORING

Consider the Grammar

$S \rightarrow Aa \mid A\,b$

OR

$stmt \rightarrow$ `if` expr `then` stmt `else` stmt |
`if` expr `then` stmt

When we see A or `if`, we cannot determine which production rule to choose to expand ***S*** or ***stmt*** since both productions have same left most symbol at the starting of the production.

(A in first example and *if* in second example)

Left-Factoring (cont.)

If there is a grammar

$$\mathbf{A \rightarrow \alpha\beta_1 \mid \alpha\beta_2}$$

where α is non-empty and the first symbols of β_1 and β_2 (if they have one)are different.

Re-write the grammar as follows:

$$\mathbf{A \rightarrow \alpha A'}$$

$$\mathbf{A' \rightarrow \beta_1 \mid \beta_2}$$

Now, we can immediately expand A to αA'

This rewriting of the grammar is called LEFT FACTORING

LEFT-FACTORING -- ALGORITHM

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

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

convert it into

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

LEFT-FACTORING – EXAMPLE1

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \Longrightarrow \begin{cases} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2 \end{cases}$

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$ **α is a; β_1 is bB; β_2 is B**

$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$

$A' \rightarrow bB \mid B$

$\Downarrow$ **α is cd; β_1 is g; β_2 is eB; β_3 is fB**

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

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

LEFT-FACTORING – EXAMPLE2

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \Longrightarrow \begin{cases} A \rightarrow \alpha A' \\ A' \rightarrow \beta 1 \mid \beta_2 \end{cases}$

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

$$\Downarrow$$

α is a; β_1 is d; β_2 is ε ; β_3 is b, β_4 is bc

$$A \rightarrow aA' \mid b$$

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

$$\Downarrow$$

α is b; β_1 is ε ; β_2 is c

$$A \rightarrow aA' \mid b$$

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

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

Non-Context Free Language Constructs

- There are some language constructions in the programming languages which are not context-free. This means that, we cannot write a context-free grammar for these constructions.

- $L1 = \{ \omega c \omega \mid \omega \text{ is in } (a|b)^* \}$ is not context-free

  ➔ Declaring an identifier and checking whether it is declared or not later. We cannot do this with a context-free language. We need semantic analyzer (which is not context-free).

- $L2 = \{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1 \}$ is not context-free

  ➔ Declaring two functions (one with n parameters, the other one with m parameters), and then calling them with actual parameters.

ERRORS

- Lexical errors include misspellings of identifiers, keywords, or operators -e.g., the use of an identifier elipsesize instead of ellipsesize - and missing quotes around text intended as a string.
- Syntactic errors include misplaced semicolons or extra or missing braces; that is, "{" or "}". As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).

ERRORS –CONTD.

- Semantic errors include type mismatches between operators and operands.An example is a return statement in a Java method with result type void.
- Logical errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==. The program containing = may be well formed; however, it may not reflect the programmer's intent.

Challenges of Error Handler

- The error handler in a parser has goals that are simple to state but challenging to realize:
  - Report the presence of errors clearly and accurately.
  - Recover from each error quickly enough to detect subsequent errors.
  - Add minimal overhead to the processing of correct programs.

Error Recovery Strategies

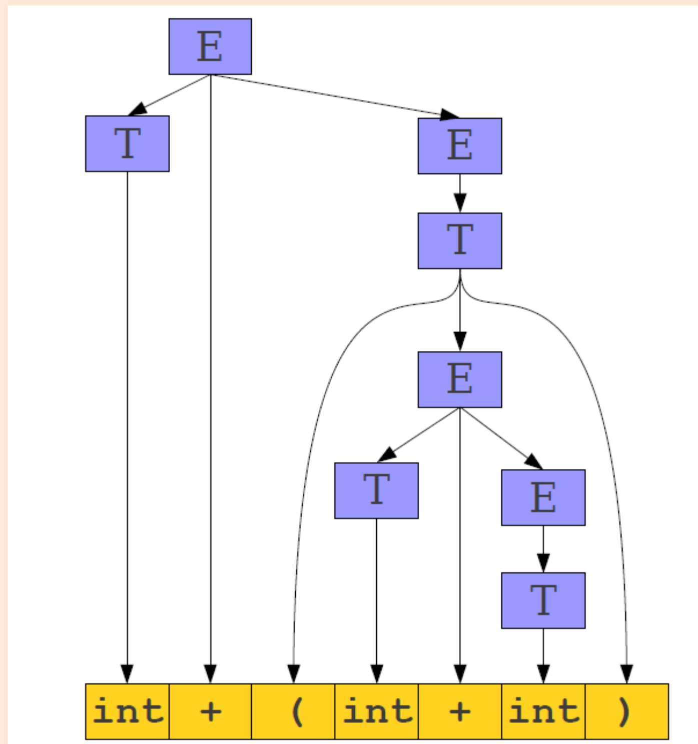
- Panic Mode
  - Parser discards input symbols one at a time until one of a designated set of synchronizing(e.g. ‘;’) token is found.
- Phrase Level
  - Parser performs local correction on the remaining input.
  - Replacement can correct any input string, but has drawback
- Error Productions
  - Augmenting the error productions to construct a parser
  - Error diagnostics can be generated to indicate the erroneous construct.
- Global correction
  - Minimal sequence of changes to obtain a globally least cost correction

Section 2.4 : Top Down Parsing

Top-Down Parsing

- Beginning with the start symbol, try to guess the productions to apply to end up at the user's program.

E → T
E → T + E
T → int
T → (E)



CHALLENGES IN TOP-DOWN PARSING

- Top-down parsing begins with virtually no information.
- Begins with just the start symbol, which matches *every* program.
- How can we know which productions to apply?
- In general, we can't.
- There are some grammars for which the best we can do is guess and backtrack if we're wrong.
- If we have to guess, how do we do it?

Top-Down Parsing

- Top-down parser
  - **Recursive-Descent Parsing**
    - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
    - 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.

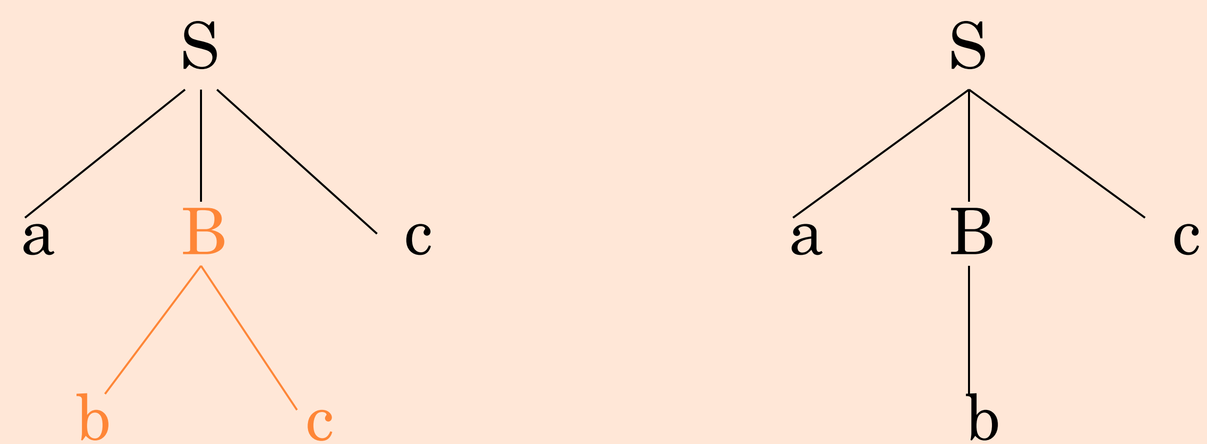
RECURSIVE-DESCENT PARSING (USES BACKTRACKING)

- Backtracking is needed.
- It tries to find the left-most derivation.

$$S \rightarrow aBc$$

$$B \rightarrow bc \mid b$$

Input: abc



fails, backtrack

Recursive Predictive Parsing

- Each non-terminal corresponds to a procedure.

Ex: $A \rightarrow aBb$ (This is only the production rule for A)

```
proc A {
    - match the current token with a, and move to the next token;
    - call 'B';
    - match the current token with b, and move to the next token;
}
```

RECURSIVE PREDICTIVE PARSING (CONT.)

$A \rightarrow aBb \ \ | \ \ bAB$

```
proc A {
   case of the current token
        {
        'a':   - match the current token with a, and move to the next token;
               - call 'B';
               - match the current token with b, and move to the next token;
        'b':  - match the current token with b, and move to the next token;
               - call 'A';
              - call 'B';
        }
     }
```

RECURSIVE PREDICTIVE PARSING (CONT.)

- When to apply ε-productions.

  $A \rightarrow aA \mid bB \mid \varepsilon$

- If all other productions fail, we should apply an ε-production. For example, if the current token is not a or b, we may apply the ε-production.

- Most correct choice: We should apply an ε-production for a non-terminal A when the current token is in the follow set of A (which terminals can follow A in the sentential forms).

Recursive Predictive Parsing (Example)

$A \rightarrow aBe \mid cBd \mid C$

$B \rightarrow bB \mid \varepsilon$

$C \rightarrow f$

proc C { match the current token with f,
and move to the next token; }

proc A {
case of the current token {
a: - match the current token with a,
and move to the next token;
- call B;
- match the current token with e,
token with b,
and move to the next token;
c: - match the current token with c,
and move to the next token;
- call B;
- match the current token with d,
and move to the next token;
f: - call C
}
}

first set of C

proc B {
case of the current token {
b:- match the current
and move to the next token;
- call B
e,d: do nothing
}
}

follow set of B

TOP-DOWN, PREDICTIVE PARSING: LL(1)

- **L**: Left-to-right scan of the tokens
- **L**: Leftmost derivation.
- **(1)**: One token of lookahead

- Construct a leftmost derivation for the sequence of tokens.

- When expanding a nonterminal, we predict the production to use by looking at the next token of the input.

TOP-DOWN, PREDICTIVE PARSING: LL(1)

a grammar ➔ eliminate left recursion ➔ left factor ➔ a grammar suitable for predictive parsing (a LL(1) grammar) no %100 guarantee.

- 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.

$A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$

input: ... a ....

current token

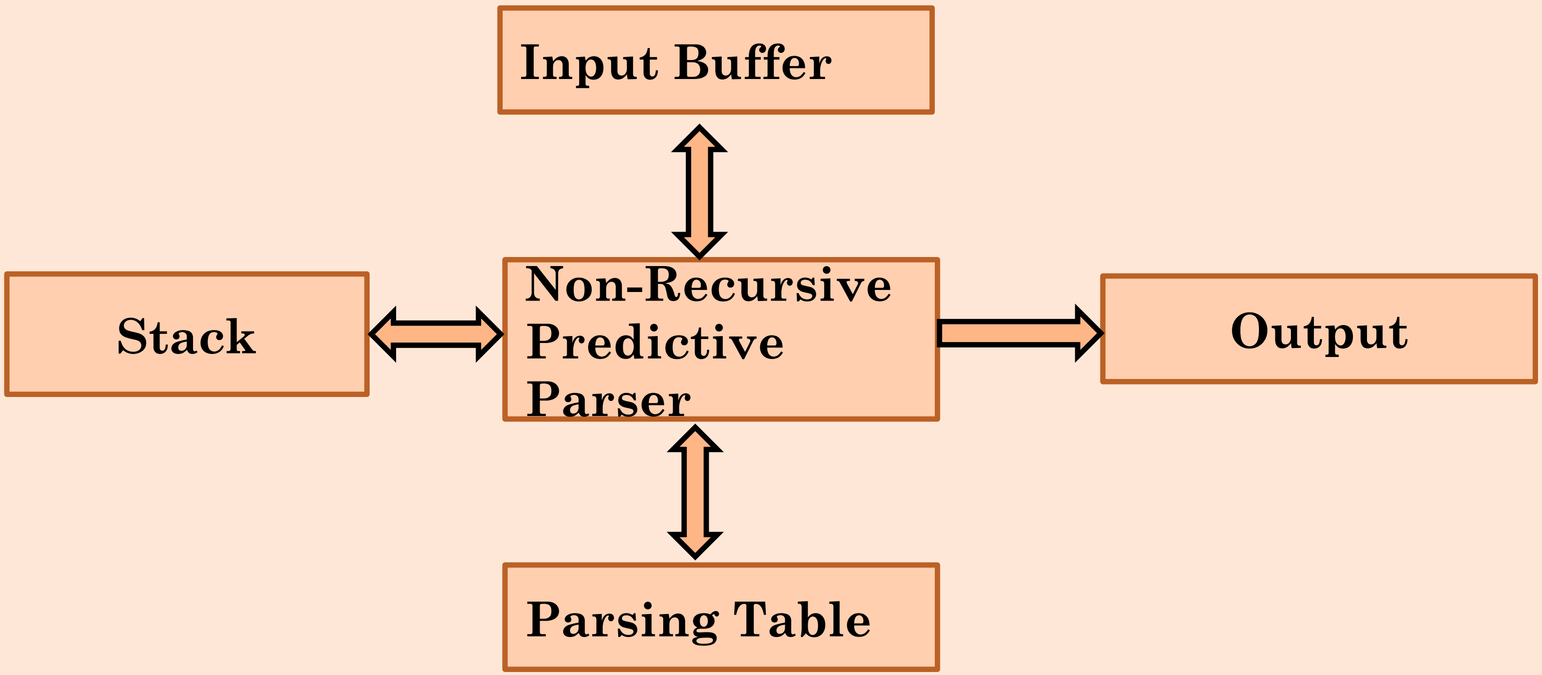
TOP-DOWN, PREDICTIVE PARSING: LL(1)

stmt → `if` ...... |
`while` ...... |
`begin` ...... |
`for` .....

- When we are trying to write the non-terminal *stmt*, if the current token is `if` we have to choose first production rule
- When we are trying to write the non-terminal *stmt*, we can uniquely choose the production rule by just looking the current token.
- We eliminate the left recursion in the grammar, and left factor it. But it may not be suitable for predictive parsing (not LL(1) grammar).

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

- 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

- Contains the 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.
- $S ← 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 row is a non-terminal symbol
- Each column is a terminal symbol or the special symbol $
- 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 \$ ➔ parser halts (successful completion)

2. If X and a are the same terminal symbol (different from \$)

   ➔ parser pops X from the stack, and moves the next symbol in the input buffer.

3. If X is a non-terminal

   ➔ parser looks at the parsing table entry M[X,a]. If M[X,a] holds a production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$, it pops X from the stack and pushes $Y_k, Y_{k-1}, \ldots, Y_1$ into the stack. The parser also outputs the production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$ to represent a step of the derivation.

4. none of the above ➔ 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.

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.

- **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 $S \stackrel{*}{\Rightarrow} \alpha A a \beta$

Compute FIRST for Any String X

- We want to tell if a particular nonterminal **A** derives a string starting with a particular terminal **t**.
- Intuitively, FIRST(**A**) is the set of terminals that can be at the start of a string produced by **A**.
- If we can compute FIRST sets for all non terminals in a grammar, we can efficiently construct the LL(1) parsing table.

COMPUTE FIRST FOR ANY STRING X

- Initially, for all non-terminals **A**, set

  FIRST(**A**) = { **t** | **A** → **t** β for some β }

  Consider the grammar :

  S→aC/bB

  B→b

  C→c

  FIRST(S) ={a,b}; FIRST (B) ={b} and FIRST(C) ={c}

- For each nonterminal **A**, for each production **A** → **B**β, set

  FIRST(**A**) = FIRST(**A**) ∪ FIRST(**B**)

Consider the grammar :

S→aC/bB/C

B→b

C→c

FIRST(S) ={a,b,c};

FIRST (B) ={b}

FIRST(C) ={c}

Consider the grammar:

S→Ab

A→a

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

FIRST COMPUTATION WITH EPSILON

- For all NT **A** where **A** → **ε** is a production, add **ε** to FIRST(A).

  For eg. S→a | **ε** **FIRST(S) →{a, ε}**

- For each production **A** → **α**, where **α** is a string of NT whose FIRST sets contain **ε**, set

  FIRST(**A**) = FIRST(**A**) ∪ { **ε** }.

  For eg. S→AB | c **;** **A→a | ε ; B→ b | ε**

  FIRST(S) →{a, b,c, ε} ; **FIRST(A) →{a, ε}** ; **FIRST(B) →{b, ε}** ;

- For each production **A** → **α**t**β**, where **α** is a string of NT whose FIRST sets contain **ε**, set

  FIRST(**A**) = FIRST(**A**) ∪ { **t** }

  For eg. S→ABcD **;** **A→a | ε ; B→ b | ε ; D→d**

  FIRST(S) →{a,b, c} ; **FIRST(A) →{a, ε}** ; **FIRST(B) →{b, ε}** ; **FIRST(D) →{d}**

- For each production **A** → **α**B**β**, where **α** is string of NT whose FIRST sets contain **ε**, set

  FIRST(**A**) = FIRST(**A**) ∪ (FIRST(**B**) - { **ε** }).

  For eg. S→ABDc | c **;** **A→a | ε ; B→ b | ε ; D→d**

  FIRST(S) →{a,b,c,d } ; **FIRST(A) →{a, ε}** ; **FIRST(B) →{b, ε}** ; **FIRST(D) →{d}**

FOLLOW SET

- The **FOLLOW set** represents the set of terminals that might come after a given nonterminal
- Formally:

  FOLLOW(**A**) = { **t** | **S** $\Rightarrow^*$ α**A**tβ for some $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ }

  where **S** is the start symbol of the grammar.
- Informally, every nonterminal that can ever come after **A** in a derivation.

COMPUTE FOLLOW FOR ANY STRING X

RULE 1: If S is the start symbol ➔ \$ is in FOLLOW(S)

RULE 2: if $A \rightarrow \alpha B \beta$ is a production rule
➔ everything in FIRST(β) is FOLLOW(B) except ε

RULE 3(i) If ($A \rightarrow \alpha B$ is a production rule) or
RULE 3(ii) ($A \rightarrow \alpha B \beta$ is a production rule and ε is in FIRST(β))
➔ everything in FOLLOW(A) is in FOLLOW(B).

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

FIRST AND FOLLOW SET EXAMPLE

Consider the grammar:

$S \rightarrow A\ a$

$A \rightarrow B\ D$

$B \rightarrow b \mid \varepsilon$

$D \rightarrow d \mid \varepsilon$

FIRST(S) = {b, d, a}
FIRST(A) = { b, d, ε }
FIRST(B) = { b, ε }
FIRST(D) = { d, ε }

FOLLOW(S) = { $ } (Rule 1)
FOLLOW(A) = { a } (Rule 2)
FOLLOW(B) = { d, a } (Rule 2; Rule 3(ii))
FOLLOW(D) = { a } Rule 3

First and Follow Set Example

Consider the grammar

C → P F class id X Y

P → public | ε

F → final | ε

X → extends id | ε

Y → implements I | ε

I → id J

J → , I | ε

FIRST(C) = {public, final, class}
FIRST(P) = { public, ε}
FIRST(F) = { final, ε}
FIRST(X) = { extends, ε }
FIRST(Y) = { implements, ε }
FIRST(I) = { id}
FIRST(J) = { ',' , ε }

FOLLOW(C)={$} (Rule 1)
FOLLOW(P)={final, class} (Rule 2; Rule 3 (ii))
FOLLOW(F) ={class} (Rule 2)
FOLLOW(X)={implements,$}(Rule 2; Rule 3(ii))
FOLLOW(Y)={$} (Rule 3(i))
FOLLOW(I)={$} (Rule 3(i))
FOLLOW(J)={$} (Rule 3(i))

LL(1) PARSING

Consider the grammar:

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

Remove Immediate Left Recursion:
(Ref: Slide no. 29)

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

FIRST EXAMPLE

Consider the grammar:

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

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

FIRST(TE′) = {(,id}
FIRST(+TE′) = {+}
FIRST(ε) = {ε}
FIRST(FT′) = {(,id}
FIRST(*FT′) = {*}
FIRST((E)) = {(}
FIRST(id) = {id}

FOLLOW EXAMPLE

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

1. If S is the start symbol ➔ $ is in FOLLOW(S)

2(i) If $A \rightarrow \alpha B\beta$ is a production rule
➔ everything in FIRST(β) is FOLLOW(B) except ε

3(i) If ($A \rightarrow \alpha B$ is a production rule) or
3(ii) ($A \rightarrow \alpha B\beta$ is a production rule and ε is in FIRST(β))

➔ everything in FOLLOW(A) is in FOLLOW(B).

Consider the following grammar:

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

E➔TE' {(Rule 1: $ in FOLLOW(**E**);
(Rule 2: A➔ αBβ : α is ε; B is **T** and β is E');
(Rule3(i): A➔ αB : α is T; B is **E'**);
Rule 3 (ii): A➔ αBβ : α is ε; B is **T** and E' is β; FIRST of β has ε)}

E➔+TE' | ε {Rule 2: A➔ αBβ : α is +; B is **T** and β is E');
(Rule3(i): A➔ αB: α is +T; B is **E'**;
(Rule3(ii): A➔ αBβ: α is +; B is **T;** β is E';FIRST of β has ε)}

T➔FT' {Rule 2: A➔ αBβ : α is ε; B is **F** and β is T');
(Rule3(i): A➔ αB : α is F; B is **T'**);
(Rule3(ii): A➔ αBβ : α is ε ; B is **F** and β is T' FIRST of β has ε)}

T'➔*FT' | ε {Rule 2: A➔ αBβ : α is *; B is **F** and β is T');
(Rule3(i): A➔ αB : α is *; B is F; β is T');
Rule3(ii): A➔ αBβ : α is *; B is F; β is T'; FIRST of β has ε)}

F → (E) | id
{(Rule 2: A➔ αBβ : α is '('; B is E and ')' is β)}

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

CONSTRUCTING LL(1) PARSING TABLE -- ALGORITHM

- for each production rule $A \rightarrow \alpha$ of a grammar G
  - for each terminal a in FIRST(α)
    ➔ add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α)
    ➔ for each terminal a in FOLLOW(A) add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α) and \$ in 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

$E \to TE'$	FIRST(TE'id}	➔ $E \to TE'$ into M[E,(] and M[E,id]
$E' \to +TE'$	FIRST(+TE')={+}	➔ $E' \to +TE'$ into M[E',+]
$E' \to \varepsilon$	FIRST(ε)={ε} but since ε in FIRST(ε) and FOLLOW(E')={$,)}	➔ none ➔ $E' \to \varepsilon$ into M[E' and M[E',)]
$T \to FT'$	FIRST(FT')={(,id}	➔ $T \to FT'$ into M[T,(] and M[T,id]
$T' \to *FT'$	FIRST(*FT')={*}	➔ $T' \to *F'$ into M[T',*]
$T' \to \varepsilon$	FIRST(ε)={ε} but since ε in FIRST(ε) and FOLLOW(T')={$,),+}	➔ none ➔ $T' \to \varepsilon$ into M[T',$], M[T' and M[T',+]
$F \to (E)$	FIRST((E))={(}	➔ $F \to (E)$ into M[F,(]
$F \to id$	FIRST(id)={id}	➔ $F \to id$ into M[F,id]

LL(1) Parser – Example 1

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

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

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

FIRST (E') has ε, so add E' → ε in FOLLOW (E')
FIRST (T') has ε, so add T' → ε in FOLLOW (T')

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

LL(1) PARSER – EXAMPLE 1

Stack	Input	Output
$E	id+id$	E→TE'
$E'T	id+id$	T→FT'
$E'T'F	id+id$	F→id
$E'T'id	id+id$	
$E'T'	+id$	T'→ε
$E'	+id$	E'→+TE'
$E'T+	+id$	
$E'T	id$	T→FT'
$E'T'F	id$	F→id
$E'T'id	id$	
$ET'	$	T'→ε
$E'	$	E'→ε
$	$	**Accept**

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

LL(1) Parser – Example 2

$S \rightarrow aBa$

$B \rightarrow bB \mid \varepsilon$

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

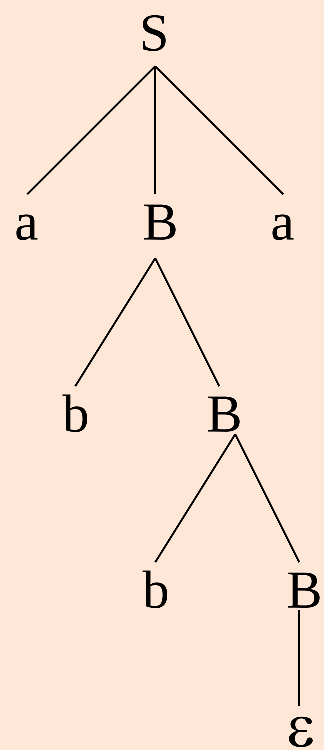
LL(1) Parsing Table

Stack	Input	Output
$S	abba$	S→aBa
$aBa	abba$	
$aB	bba$	B→bB
$aBb	bba$	
$aB	ba$	B→bB
$aBb	ba$	
$aB	a$	B→ε
$a	a$	
$	$	**Accept, Successful Completion**

LL(1) PARSER – EXAMPLE2 (CONT.)

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

Derivation(left-most): $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



parse tree

A GRAMMAR WHICH IS NOT LL(1)

$S \rightarrow i\ C\ t\ S\ E \quad | \quad a$
$E \rightarrow e\ S \quad | \quad \varepsilon$
$C \rightarrow b$

FOLLOW(S) = { \$,e }
FOLLOW(E) = { \$,e }
FOLLOW(C) = { t }

FIRST(iCtSE) = {i}
FIRST(a) = {a}
FIRST(eS) = {e}
FIRST(ε) = {ε}
FIRST(b) = {b}

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

two production rules for M[E,e]

Problem ➔ ambiguity

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

- What do we have to do it 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.
  - $A \to A\alpha \mid \beta$
    - ➔ any terminal that appears in FIRST(β) also appears FIRST($A\alpha$) because $A\alpha \Rightarrow \beta\alpha$.
    - ➔ If β is ε, any terminal that appears in FIRST(α) also appears in FIRST($A\alpha$) and FOLLOW(A).
- A grammar is not left factored, it cannot be a LL(1) grammar
  - $A \to \alpha\beta_1 \mid \alpha\beta_2$
    - ➔ any terminal that appears in FIRST($\alpha\beta_1$) also appears in 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.
  2. At most one of α and β can derive to ε.
  3. If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).
- In other word we can say that a grammar *G* is LL(1) iff for any productions

  $\mathbf{A} \rightarrow \boldsymbol{\omega}_1$ and $\mathbf{A} \rightarrow \boldsymbol{\omega}_2$, the sets

FIRST($\boldsymbol{\omega}_1$ FOLLOW($\mathbf{A}$)) and FIRST($\boldsymbol{\omega}_2$ FOLLOW($\mathbf{A}$)) are disjoint.

- This condition is equivalent to saying that there are no conflicts in the table.