B.SC. CSIT
6Th Semester
Compiler Design and ConstructionCSC-365

Unit 2.2.1
Syntax Analysis
Top-Down Parsing (4 hrs)

Instructor

Tekendra Nath Yogi

Tekendranath@gmail.com

Contents

- Introduction to Syntax Analysis (Parsing)
- Role of Parser
- Context Free Grammar:
  - Notational conventions for CFG
  - Derivation
  - Parse Tree
  - Ambiguous grammar
  - Left Recursion
  - Left Factoring
- Basic Parsing technique - Top-down Parsing
  - Recursive decent parsing
  - Recursive predictive parsing
  - Non-Recursive predictive parsing
  - LL grammar

Syntax Analysis

- Every programming language has precise rules that prescribe the syntactic structure of well-formed programs.
  - For example, a C program is made up of functions, a function out of declarations and statements, a statement out of expressions, and so on.

- To translate program, compiler need to verify that the source code written for program is syntactically valid.

- The validity of the syntax is checked by the compiler component syntax analyzer, also called the parser.

Role of Parser

- The parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language.

- For well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

- For ill-formed programs, the parser report syntax errors in an intelligible fashion and try to recover from commonly occurring errors to continue processing the remainder of the program.

Role of Parser

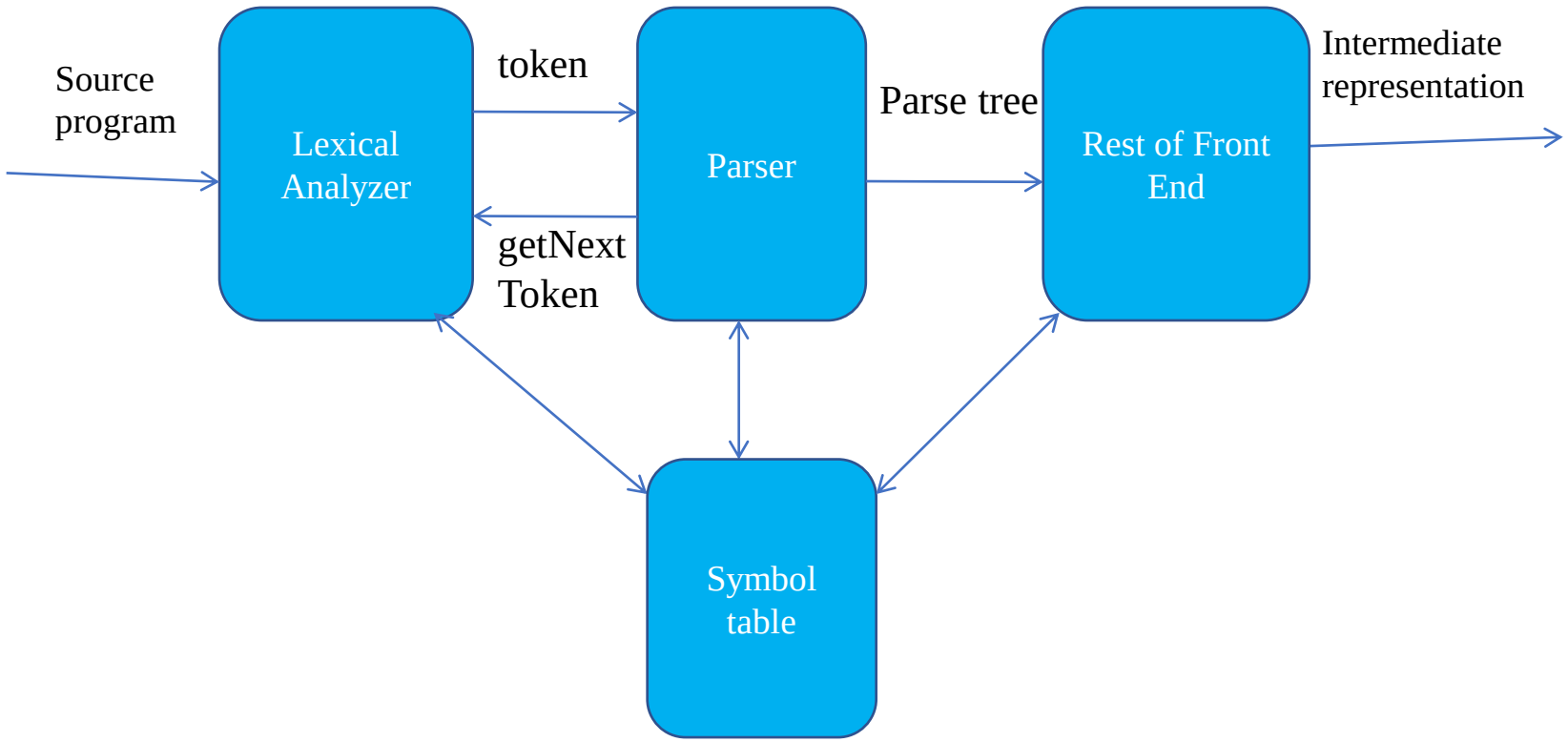
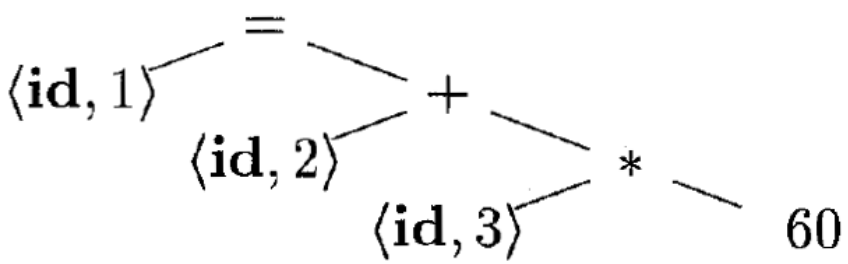


Figure: Position of parser in compiler

Role of Parser

- For example: Let the token stream <id, 1> <=> <id, 2> <+> <id, 3> <*> <60> . For the given token stream output of the parser shown in Figure below:



Types of Parsers

- Three types of parsers for grammars:
  - Universal Parsers
    - Can parse any kind of grammars.
    - Inefficient for compiler construction
    - E.g. Cocke-Younger-Kasami algorithm and Earley's algorithm.
  - Top Down Parsers
    - Builds parse tree from the top to the bottom, starting from the root.
    - Easy to build.
    - Works on subset of grammars e.g. LL grammars.
  - Bottom Up Parsers
    - Builds parse tree from bottom to up, starting from the leaves.
    - Difficult to build.
    - Works on subset of grammars e.g. LR grammars.
- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).

Context Free Grammar (CFG)

- The syntax of a programming language construct is specified by a context-free grammar (CFG). CFG offer the following significant benefits for both language designers and compiler writers.
  - A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.
  - From grammars, we can construct automatically an efficient parser that determines the syntactic structure of a source program. As a side benefit, the parser-construction process can reveal syntactic ambiguities and trouble spots that might have slipped through the initial design phase of a language.
  - The structure imparted to a language by a properly designed grammar is useful for translating source programs into correct object code and for detecting errors.
  - A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks. These new constructs can be integrated more easily into an implementation that follows the grammatical structure of the language.

Context Free Grammar (CFG)

- CFG is a 4 tuple (V, T, P, S), where
  - **V** - finite set of Variables or Non-terminals that are used to define the grammar denoting the combination of terminal or no-terminals or both.
  - **T** - Terminals, the basic symbols of the sentences. Terminals are indivisible units.
  - **P** - Production rule that defines the combination of terminals or non-terminals or both for particular non-terminal.
    - $\mathbf{A \rightarrow \alpha}$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string).
  - **S** - It is the special non-terminal symbol called start symbol.

Context Free Grammar (CFG)

- **For example:**
  - The CFG for language of all strings with equal no of 0's followed by equal no of 1's is represented by: G =({S}, {0, 1}, P, S), where
  - P represents the set of following two productions.

    S→ Є

    S→ 0S1

Context Free Grammar (CFG) – Notational Conventions

- Terminals:
  - Lowercase letters, symbols (like operators), parenthesis, digits, bold string are used for denoting terminals.
  - For example, a, b,….0, 1,…+, -, *, **if, else, etc.**
- Non-terminals:
  - Uppercase letters, italicized strings are used for denoting non-terminals.
  - For example, A, B, …., *stmt*, *etc.*
- String :
  - Lowercase Greek letters are used to denote the string of terminal, non-terminal or combination of both.
  - For example, α, β, γ, δ
- Production of the form A → a | b is read as "A produces a or b".
- For Example, *stmt* → **if** E *stmt* **;**| **if** E *stmt* **else** *stmt* **;**.

Context Free Grammar (CFG) -- Derivation

- Purpose of the grammar is to define the language and the verification of the string of terminals (Sentence) defined by the grammar is done using the production rule.
- The process of obtaining the particular sentence by expansion of the non-terminals is called derivation.
  - In general a derivation step is $\alpha A\beta \Rightarrow \alpha\gamma\beta$ if there is a production rule $A \rightarrow \gamma$ in our grammar where α and β are arbitrary strings of terminal and non-terminal symbols.
  - $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)
- The notations denotes:

  $\Rightarrow$: derives in one step

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

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

Context Free Grammar (CFG) -- Derivation

- **Example:** Consider the following CFG

  $E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid - E$

  $E \rightarrow (E)$

  $E \rightarrow id$

  - $E \Rightarrow E+E \Rightarrow id + E \Rightarrow id + id$
    - E+E derives from E
    - we can replace E by E+E
    - to able to do this, we have to have a production rule $E \rightarrow E+E$ in our grammar.

- A sequence of replacements of non-terminal symbols is called a **derivation** of id + id from E.

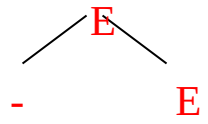
Context Free Grammar (CFG) -- Derivation

- At each derivation step, we can choose any of the non-terminal in the sentential form of CFG for the replacement.
- If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.
  - $E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(id+E) \underset{lm}{\Rightarrow} -(id+id)$
- If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.
  - $E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+id) \underset{rm}{\Rightarrow} -(id+id)$
- We will see that the top-down parsers try to find the left-most derivation of the given source program.
- We will see that the bottom-up parsers try to find the right-most derivation of the given source program in the reverse order.

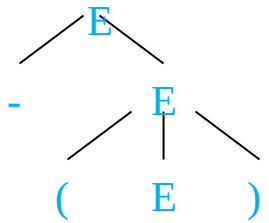
Context Free Grammar (CFG) -- Parse Tree

- The pictorial representation of the derivation. Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- For example: The parse tree for the derivation E ⇒ -E ⇒ -(E) ⇒ -(E+E) ⇒ -(id + E) ⇒ -(id + id)is as shown below:

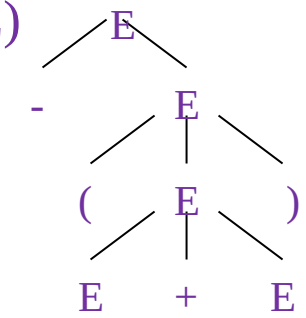
E ⇒ -E



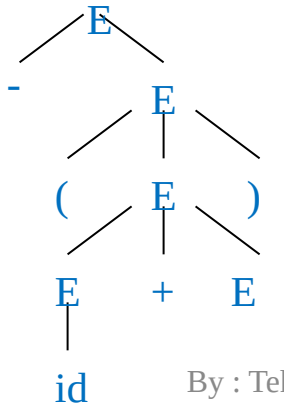
⇒ -(E)



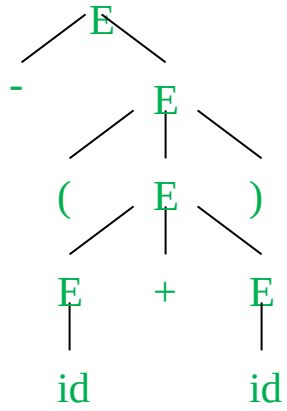
⇒ -(E+E)



⇒ -(id + E)



⇒ -(id + id)

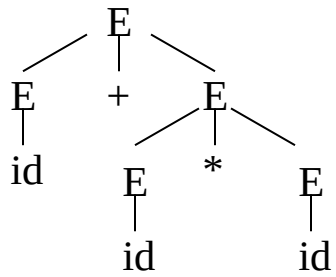


Context Free Grammar (CFG) -- Ambiguity

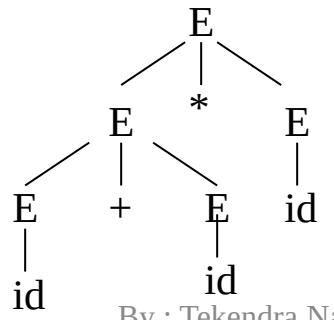
- A grammar produces more than one parse tree for a sentence is called as an **ambiguous** grammar.

  Or more than one leftmost derivation Or more than one rightmost derivation.

- **Example: For string id + id * id**
  - Leftmost derivation1: E ⇒ E+E ⇒ id+E ⇒ id+E*E ⇒ id+id*E ⇒ id+id*id



  - Leftmost derivation2: E ⇒ E*E ⇒ E+E*E ⇒ id+E*E ⇒ id+id*E ⇒ id+id*id



Context Free Grammar (CFG) -- Ambiguity

- For the most parsers, the grammar must be unambiguous.
- unambiguous grammar

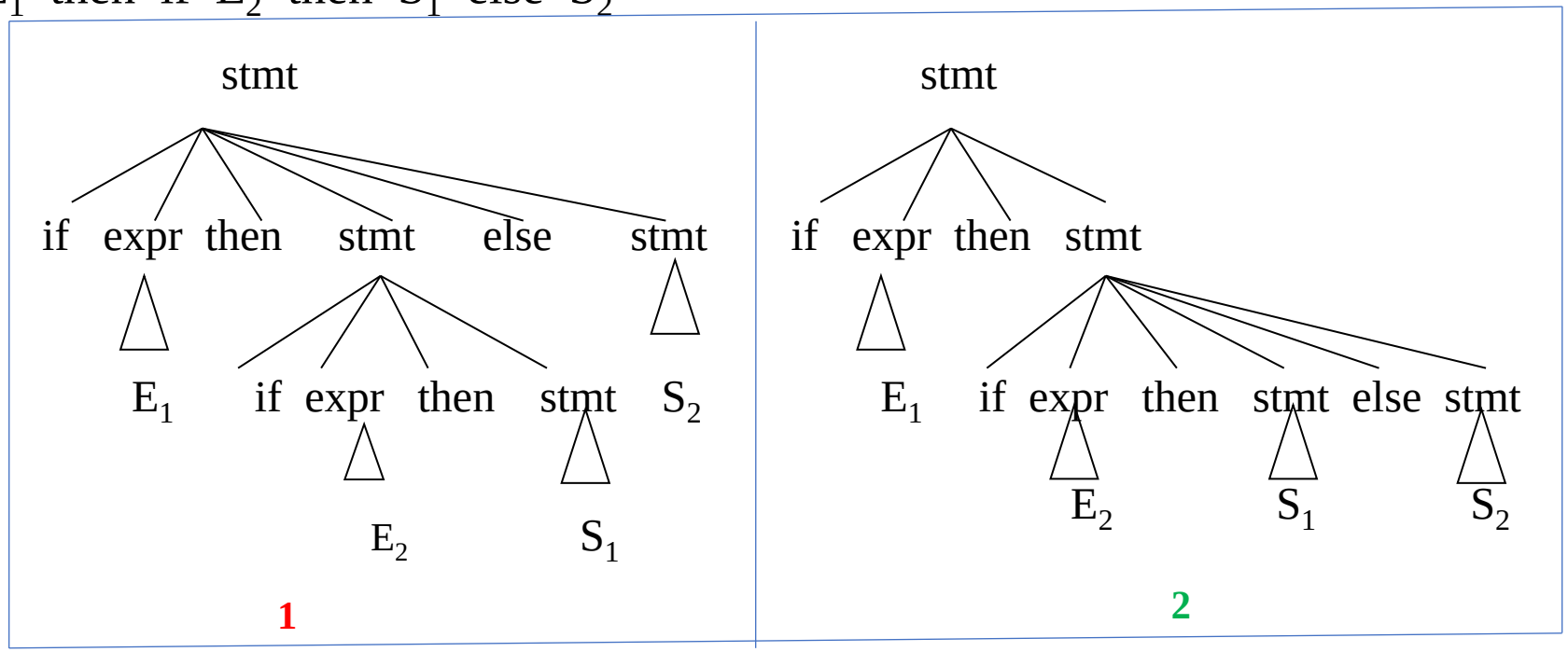
  ➔ unique selection of the parse tree for a sentence

- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
- An unambiguous grammar should be written to eliminate the ambiguity.
- We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice.

Context Free Grammar (CFG) -- Ambiguity

- **For example:**
  - *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



Context Free Grammar (CFG) -- Ambiguity

- 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

Context Free Grammar (CFG) -- Ambiguity

- **Ambiguity – Operator Precedence:**
  - Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the precedence and associativity rules.
  - For example:

$E \rightarrow E+E \mid E*E \mid E\hat{}E \mid id \mid (E)$

$\Downarrow$ disambiguate the grammar

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow G\hat{}F \mid G$

$G \rightarrow id \mid (E)$

- Note the precedence and associativity:
  - ^ (right to left)
  - * (left to right)
  - +(left to right)

Left Recursion

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

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

- 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 (General left recursion).

Immediate Left Recursion

- Immediate left recursion present in the grammar can be eliminated by using the following rule:
- Rule:

$A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\,A'$

$A' \rightarrow \alpha\,A' \mid \varepsilon$ an equivalent grammar

- In general,

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

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\,A' \mid \ldots \mid \beta_n\,A'$

$A' \rightarrow \alpha_1\,A' \mid \ldots \mid \alpha_m\,A' \mid \varepsilon$ an equivalent grammar

Immediate Left-Recursion-Example

- Example: Eliminate the left recursion present in the following grammar

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

$\Downarrow$ eliminate immediate left recursion

$E \rightarrow T\ E'$

$E' \rightarrow +T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

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

$F \rightarrow id \mid (E)$

Left-Recursion - Problem

- A grammar cannot be immediately left-recursive, but it still can be left-recursive.
- By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.
- For example:

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$

This grammar is not immediately left-recursive, but it is still left-recursive.

How?

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

$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

- So, we have to eliminate all left-recursions from our grammar

Eliminate Left-Recursion -- Algorithm

- Arrange non-terminals in some order: $A_1 \ldots A_n$
- 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
  - $A_i \rightarrow \alpha_1 \gamma \mid \ldots \mid \alpha_k \gamma$
  - where $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$

}

- eliminate immediate left-recursions among A_i productions

}

Eliminate Left-Recursion – Example1

Example1: Eliminate the left recursion present in the following grammar

$S \rightarrow Aa \mid b$

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

- Let the order of non-terminals: S, A

for S:

- we do not enter the inner loop and no immediate left recursion in S.

for A:

- Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$

  So, we will have $A \rightarrow Ac \mid Aad \mid bd \mid f$

- Eliminate the immediate left-recursion in A

  $A \rightarrow bdA' \mid fA'$

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

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid fA'$

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

Eliminate Left-Recursion – Example2

Example 2: Eliminate the left recursion present in the following grammar

$S \rightarrow Aa \mid b$

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

-Let the order of non-terminals: A, S

for A:

- we do not enter the inner loop.
- Eliminate the immediate left-recursion in A

$A \rightarrow SdA' \mid fA'$

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

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

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

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

So, the resulting equivalent grammar which is not left-recursive is:

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

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

$A \rightarrow SdA' \mid fA'$

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

Left-Factoring

- A predictive parser (a top-down parser without backtracking) insists that the grammar must be left-factored.
- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
- On seeing input if it is not clear for the parser which production to use then perform left factoring:
  - Consider following grammar:

    stmt $\rightarrow$ if expr then stmt else stmt

    | if expr then stmt

  - when we see this grammar, we cannot decide which production rule to choose to re-write *stmt* in the derivation. So, left factor the given grammar.

Left-Factoring

- In general,

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ where α is non-empty and the first symbols

  of β_1 and β_2 (if they have one)are different.

- when processing α we cannot know whether expand

  A to $\alpha\beta_1$ or

  A to $\alpha\beta_2$

- But, if we re-write the grammar as follows

  $A \rightarrow \alpha A'$

  $A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

Left-Factoring

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

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

  - Find the longest, non-empty prefix common to two or more of its alternatives and replace all of A-productions by:

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

- For example 1:

$A \rightarrow abB \mid aB \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow abB \mid aB \mid cdA'$

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

$\Downarrow$

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

$A'' \rightarrow bB \mid B$

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

Top Down Parsing

- A Top-down parser tries to create a parse tree from the root towards the leaf scanning input from left to right
- It can be also viewed as finding a leftmost derivation for an input string
- Example:
  - Input: id + id * id
  - Grammar:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

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

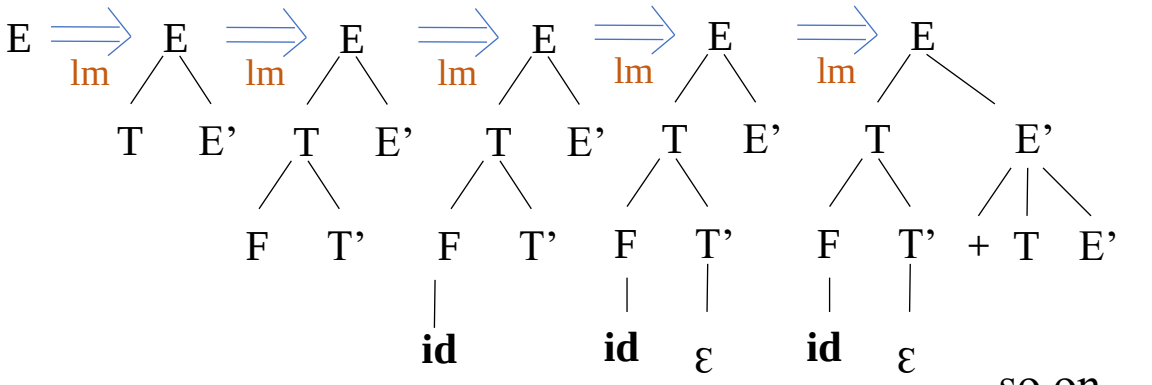


Figure: Top-down parse for id + id * id

Top Down Parsing

- Two types of top-down parser:
  - Recursive-Descent Parsing
    - It is a general parsing technique, but not widely used.
    - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
    - Not efficient
  - Predictive Parsing
    - no backtracking
    - efficient
    - needs a special form of grammars (LL(1) grammars).
    - **Non-Recursive (Table Driven) Predictive Parser** is also known as LL(1) parser.
    - **Recursive Predictive Parsing** is a special form of Recursive Descent parsing without backtracking.

Recursive-Descent Parsing (uses Backtracking)

- A recursive-descent parsing program consists of a set of procedures, one for each non-terminal.
- Execution begins with the procedure for start symbol
- A typical procedure for a non-terminal

```
void A( )
{
        1. choose an A-production, A → X1X2..Xk
        2. for (i=1 to k)
        {
        3.      if (Xi is a non-terminal)
        4.              call procedure Xi( );
        5.      else if (Xi equals the current input symbol a)
        6.              advance the input to the next symbol;
        7.      else /* an error has occurred*/
        }
}
```

Recursive-Descent Parsing (uses Backtracking)

- General recursive descent parsing may require backtracking:
  - If one failed, reset the input pointer to where it was when we first reached line (1).
  - Return to line 1 and try another A-production.
  - If there are no more A-productions to try then declare that an input error has been found.
  - If its procedure body scans the entire input string then halts and announces success .

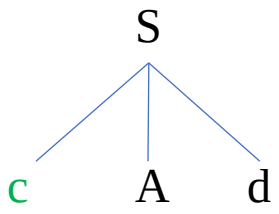
Recursive-Descent Parsing (uses Backtracking)

- **Example:** Construct a parse tree by using recursive descendent parsing for the input string w= cad, by considering the following grammar.

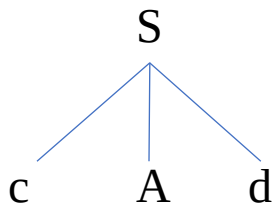
$$S \rightarrow cAd$$

$$A \rightarrow ab \mid a$$

Solution:

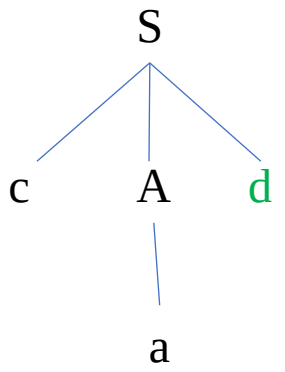


input= cad
Matched
so continue..



input= cad
Matched
so continue..

input= cad
Not Matched
So backtrack



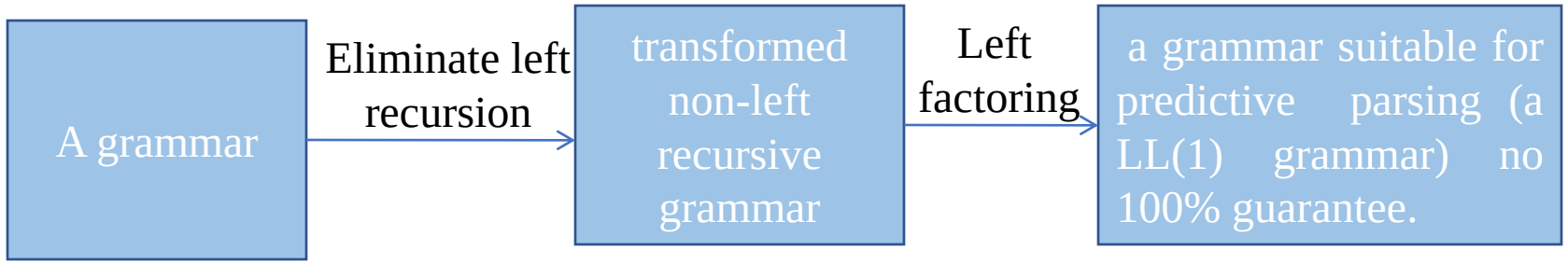
input= cad
Matched
so continue..

input= cad
Matched
so continue..

Successful parsing!

Predictive Parser (doesn't use backtracking)

- A predictive parser tries to predict which production produces the least chances of a backtracking and infinite looping.
- For this, 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). That is,



- 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.

Predictive Parser (doesn't use backtracking)

- For example: consider the following grammar

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

- When the predictive parser are trying to write the non-terminal *stmt*, the predictive parser can uniquely choose the production rule by just looking the current token. For example, if the current token is if then the predictive parser have to choose first production rule.

- Two types of predictive parsers:
  - Recursive predictive parsers
  - Non-recursive predictive parsers

Recursive Predictive Parsing

- Each non-terminal corresponds to a procedure.
- For example: 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

- For example: A → 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

- 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

- For example:

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

$B \rightarrow bB \mid \varepsilon$

$C \rightarrow f$

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

follow set of B

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.

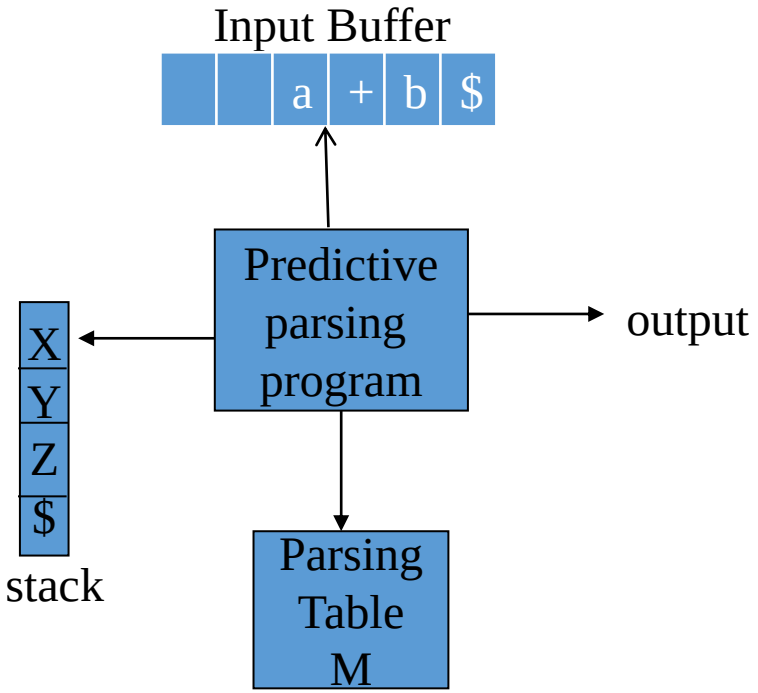


Figure: Model of a table-driven predictive parser

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

1. Input buffer

- Contains string to be parsed. Assume that its end is marked with a special symbol $.

2. Output

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

3. 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.
- when the stack is emptied (i.e. only $ left in the stack), the parsing is completed.

4. 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 -- Algorithm

- Input : a string w.
- Output: if w is in L(G), a leftmost derivation of w; otherwise error
- Method:

1. Set input pointer (ip) to the first symbol of input stream
2. Set the stack to $S where S is the start symbol of the grammar
3. repeat
   -Let X be the top stack symbol and a be the symbol pointed by ip
   -If X is a terminal or $ then
      -if X = = a then pop X from the stack and advance ip
      -else error()
   -else /* X is a non-terminal */
      -if M[X, a] = $X \rightarrow Y_1, Y_2, \ldots, \ldots, Y_k$ then
         -pop X from stack
         -push $Y_k, Y_{k-1}, \ldots, \ldots, Y_1$ onto stack (with Y_1 on top)
         -output the production $X \rightarrow Y_1, Y_2, \ldots, Y_k$
      -else error()
4. until X = $ /* stack is empty */

LL(1) Parser – Example 1

- **Example:** Construct a parse tree by using non-recursive predictive parsing for the input string w= abba$, by considering the following grammar.

  $S \rightarrow aBa$

  $B \rightarrow bB \mid \varepsilon$

- Let us consider the following LL(1) parsing table

Input Symbol / Non-Terminal	**a**	**b**	**$**
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

- **LL(1) parser action sequence:**

stack	**input**	**output**
$S	abba$	$S \rightarrow aBa$
$aBa	abba$	-
$aB	bba$	$B \rightarrow bB$
$aBb	bba$	-
$aB	ba$	$B \rightarrow bB$
$aBb	ba$	-
$aB	a$	$B \rightarrow \varepsilon$
$a	a$	-
$	$	**accept, successful Completion!**

Outputs:

$S \rightarrow aBa \;\; B \rightarrow bB \;\; B \rightarrow bB \;\; B \rightarrow \varepsilon$

Derivation(left-most):

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

Parse tree:

corresponding to the above derivation is as shown in figure below:

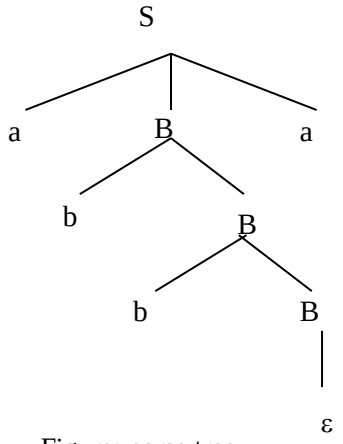


Figure: parse tree

Constructing LL(1) Parsing Tables

- To construct the LL(1) parsing table:
  - Eliminate left recursion from grammar
  - Left factor the grammar
  - Compute FIRST and FOLLOW functions

Constructing LL(1) Parsing Tables

FIRST and FOLLOW

- Two functions are used in the construction of LL(1) parsing tables:
- **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 $S \overset{*}{\Rightarrow} \alpha A$

Constructing LL(1) Parsing Tables

Compute FIRST for Any String X

- If X is a terminal symbol

  ➔ FIRST(X)={X}

- If X is a non-terminal symbol and $X \rightarrow \varepsilon$ is a production rule

  ➔ ε is in FIRST(X).

- If X is a non-terminal symbol and $X \rightarrow Y_1 Y_2 .. Y_n$ is a production rule

  ➔ if a terminal **a** in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1then **a** is in FIRST(X).

  ➔if ε is in all FIRST(Y_j) for j=1,...,n then ε is in FIRST(X).

Constructing LL(1) Parsing Tables

Compute FIRST for Any String X : Example

- Consider the following grammar

  $E \rightarrow TE'$

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

  $T \rightarrow FT'$

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

  $F \rightarrow (E) \mid id$

- Now, computing FIRST:

  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}

  FIRST(+) = {+}

  FIRST(*) = {*}

Constructing LL(1) Parsing Tables

Compute FOLLOW (for non-terminals)

- If S is the start symbol

  ➔ $ is in FOLLOW(S)

- if $A \rightarrow \alpha B \beta$ is a production rule

  ➔ everything in FIRST(β) is in FOLLOW(B) except ε

- 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)

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

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

Constructing LL(1) Parsing Tables

Compute FOLLOW : Example

Example: Consider the following grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Now, computing FOLLOW():

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

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

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

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

Constructing LL(1) Parsing Tables -- Algorithm

- Input: LL(1) Grammar G
- Output: Parsing Table M
- Method:
  - 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 Tables -- Example

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

LL(1) Parser – Example-2

- Consider the following grammar LL(1) grammar to parse the input sting **w = id + id$.**

  $E \rightarrow TE'$

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

  $T \rightarrow FT'$

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

  $F \rightarrow (E) \mid id$

- The LL(1) parsing table for the above given LL(1) grammar is:

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

LL(1) Parser – Example-2

- **LL(1) parser action sequence:**

stack	**input**	**output**
$E	id+id$	$E \rightarrow TE'$
$E'T	id+id$	$T \rightarrow FT'$
$E'T'F	id+id$	$F \rightarrow id$
$ E'T'id	id+id$	
$ E'T'	+id$	$T' \rightarrow \varepsilon$
$ E'	+id$	$E' \rightarrow +TE'$
$ E'T+	+id$	
$ E'T	id$	$T \rightarrow FT'$
$ E'T'F	id$	$F \rightarrow id$
$ E'T'id	id$	
$ E'T'	$	$T' \rightarrow \varepsilon$
$ E'	$	$E' \rightarrow \varepsilon$
$	$	**accept**

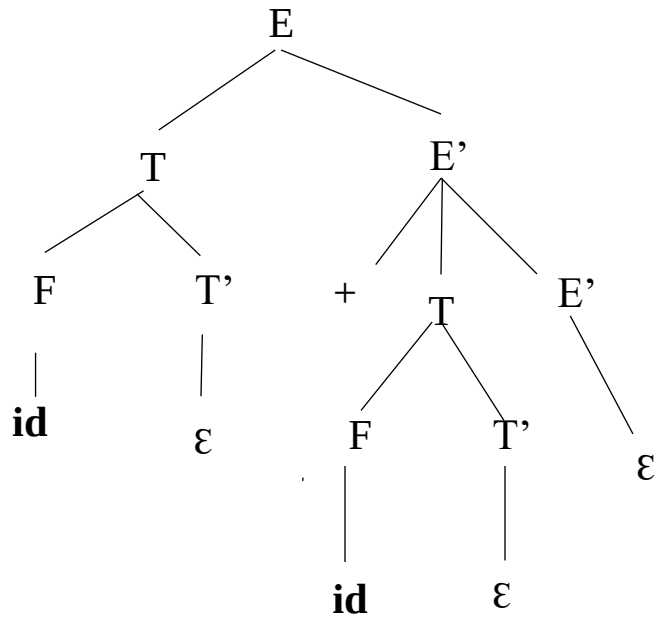
LL(1) Parser – Example-2

- **Output:** $E \rightarrow TE'$ $T \rightarrow FT'$ $F \rightarrow id$ $T' \rightarrow \varepsilon$

  $E' \rightarrow +TE'$ $T \rightarrow FT'$ $F \rightarrow id$ $T' \rightarrow \varepsilon$ $E' \rightarrow \varepsilon$

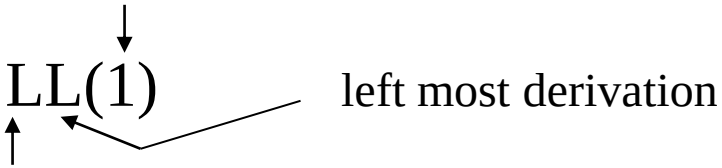
- **Derivation:** $E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow idT'E' \Rightarrow idE' \Rightarrow id+TE' \Rightarrow id+FT'E' \Rightarrow id+idTE' \Rightarrow id+idE' \Rightarrow id+id$

- **Parse tree:**



LL(1) Grammar

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



- 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.

A Grammar which is not LL(1)

- **For Example:**

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

$E \rightarrow e\ S \mid \varepsilon$

$C \rightarrow b$

FIRST(iCtSE) = {i}

FIRST(a) = {a}

FIRST(eS) = {e}

FIRST(ε) = {ε}

FIRST(b) = {b}

FOLLOW(S) = { $,e }

FOLLOW(E) = { $,e }

FOLLOW(C) = { t }

Problem ➔ ambiguity

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

two production rules for M[E, e]

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.
  - $A \rightarrow A\alpha \mid \beta$
    - ➔ 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
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
    - ➔ 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).

Homework

- Define a Context free grammar. What are the component of CFG? Explain.
- What are the problem with top down parsers? Explain the LR parsing algorithm.
- Consider the following grammar for set of all string with equal number of a's and b's.

  $S \rightarrow aSbS|bSaS|\ \varepsilon$

  - Show that this grammar is ambiguous by constructing two different leftmost derivations for sentence abab
  - Construct the corresponding rightmost derivations for abab.
  - Construct the corresponding parse trees for abab.
- Consider the following grammar

  $S \rightarrow (S)|a$

  $L \rightarrow LS|S$

  - Eliminate left recursion
  - Compute FIRST and FOLLOW for the symbol in the grammar.
- Show that the following grammar is not in a LL(1) grammar.

  $S \rightarrow cAd$

  $A \rightarrow Ab|a$

Homework

- Translate the arithmetic expression a*-(b + c) into syntax tree. Explain the ambiguous grammar.
- Explain in detail about the recursive decent parsing with example.
- Given the following grammar:

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

  Construct the parsing table for this grammar for non-recursive predictive parser.

- Find the FIRST and FOLLOW of all the non terminals in the following grammar

  $A \rightarrow TEE$
  $A \rightarrow +TE \mid \varepsilon$
  $T \rightarrow XY$
  $Y \rightarrow *XY \mid \varepsilon$
  $X \rightarrow (A) \mid a$

Homework

- The following grammar generates the grammar of the language consisting of all strings of even length.

  $S \rightarrow AS \mid \varepsilon$

  $A \rightarrow aa \mid bb \mid ab \mid ba \mid bb$

Give leftmost and rightmost derivations for the following strings:

- aabbba
- baabab
- aaabbb

Thank You !