

Compiler design

Department of Computer Science
FACULTY OF ENGINEERING AND TECHNOLOGY
Debre Markos University
Year IV Semester I

by...

Birku L
B.Sc. CS, M.Sc. SE
The Academic year of 2015 E.C

Chapter 3 Outline

❖ Role of parsing and Parsing tree?

- Context free Grammar
- Derivation
- Ambiguity

❖ Top-Down Parsing

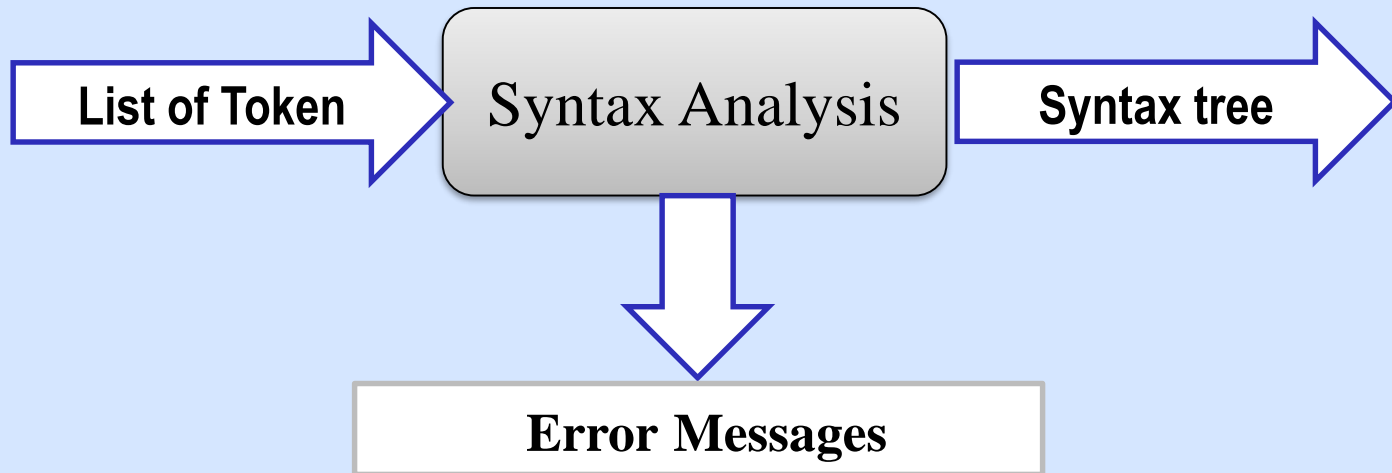
- Recursive parsing
- Non Recursive parsing

❖ Bottom-Up Parsing

- S-R parsing
- Operator-Precedence Parsing
- L- R Parsing, LALR parsing
- SLR and CLR Parsing

Syntax Analysis

Syntax (Hierarchical) analysis or parsing This phase takes the **list of tokens** produced by the lexical analysis and arranges these in a tree-structure (called the **syntax tree**) that reflects the **grammatical** structure of the program. Also known as the **heart of compiler**.



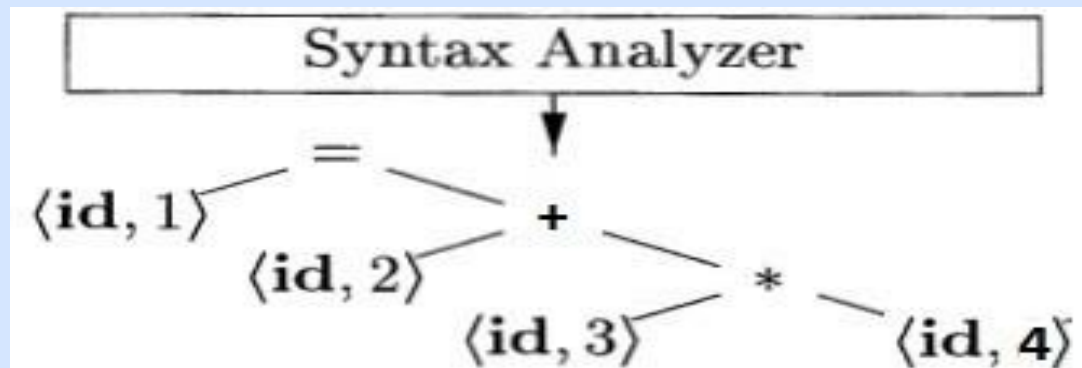
In a syntax tree:

Each **interior node** represents an **operation**

The **children of the node** represent **arguments/operands**.

Syntax Analysis

Example: Syntax Tree for Assignment Statement $D = A + B * C$

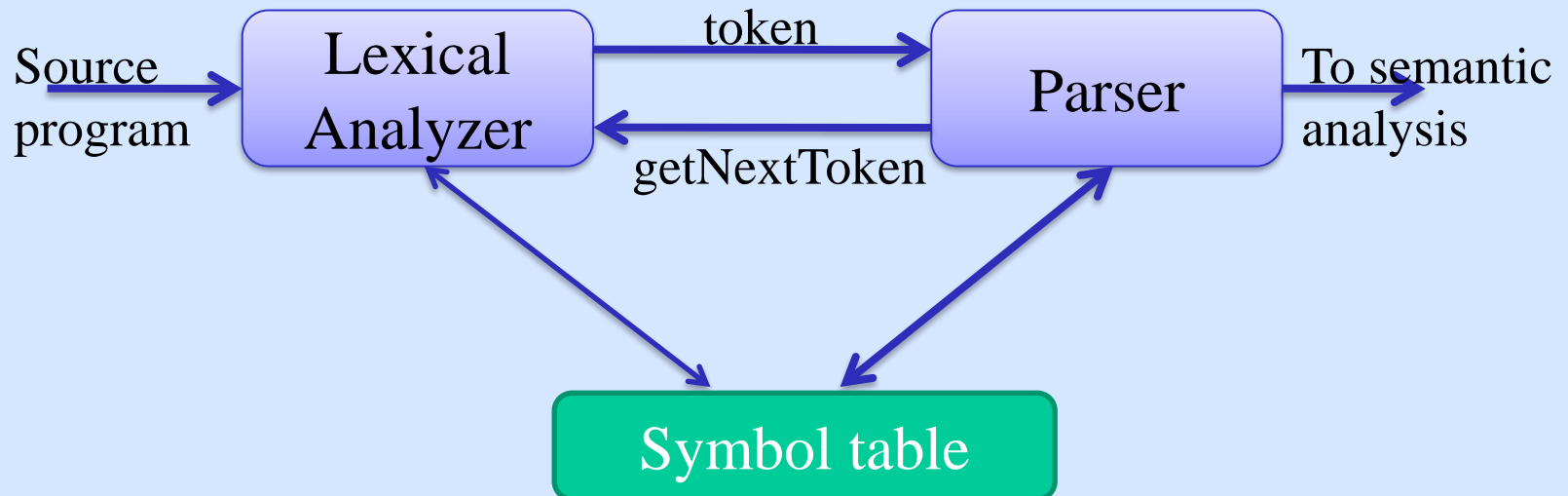


- ❖ The tree above have an interior node labeled ‘*’ with left child as (id , 3) and a right child (id , 4),
- ❖ Labeled ‘+’ with left child as (id , 2) and the multiplication of (id , 3) and a (id , 4).
- ❖ Labeled ‘=’ assign the value in to (id , 1)
- ❖ So =,+,* are interior node also (id , 1) , (id , 2) , (id , 3) are the children of the node

Role of Syntax Analysis(parsing)

❖ Main task

1. To read the sequence of Token
2. To generate them into Syntax(parse) tree using grammar
3. To produce a parse tree to the next phase(Semantic Analysis)
4. To Report errors if those tokens do not properly encode a structure.



Context-free grammar (or CFG)

- ❖ A context-free grammar (or CFG) is a formalism for defining languages..
- ❖ CFGs are good for describing the overall syntactic structure of programs.
- ❖ Can define the context-free languages, a strict superset of the regular languages, i.e. More powerful than regular expressions

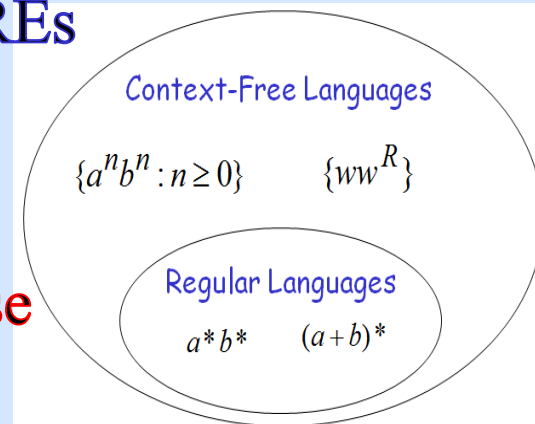
Regular expressions:

- ❖ Normally used to classify identifiers, numbers, keywords ...
- ❖ Simpler and more concise for tokens than a grammar
- ❖ More efficient scanners can be built from REs

CFGs are best explained by example...

CFGs are used to impose structure

- ❖ Brackets: (), begin ... end, if ... then ... else
- ❖ Expressions, declarations ...



Context-free grammar (or CFG)

- ❖ Many programming language constructs have an inherently recursive structure that can be defined by context-free grammars.
- ❖ For example, we might have a conditional statement defined by a rule such as
 - S_1 and S_2 are statements and E is an expression. then “if E then S_1 else S_2 IS a statement”.
- ❖ This form of conditional statement cannot be specified using the notation of regular expressions;
- ❖ On the other hand use the syntactic variable
 - stmt to denote the class of statements and
 - expr the class of expressions. we can express using the grammar production
stmt -- If expr then stmt else stmt
- ❖ for this kinds of language we are using CFG

Context-free grammar (or CFG)

- ❖ many textbooks use different **symbols** and terms to describe CFG's
- ❖ Definition Formally, a CFG G is a 4-tuple $G = (V, T, P, S)$
 - V = **variables** or **nonterminal** a finite set
 - T = **alphabet** or **terminals** a finite set
 - P = **productions** a finite set
 - S = **start variable** $S \in V$
- ❖ Productions' form, where $A \in V, a \in (V \cup T)^*$: $A \rightarrow a$
 - left-hand side: **non-terminal**
 - right-hand side: **terminals** and/or **non-terminals**
 - rules **explain** how to rewrite **non-terminals** (beginning with start symbol) into terminals

Context-free grammar (or CFG)

- ❖ context free grammar (grammar for short) has four tuples those are terminals, nonterminal, a start symbol, and productions.
- 1. Terminals are the basic symbols from which strings are formed. The word "token" is a synonym for "terminal" when we are talking about grammars for programming languages. each of the keywords **if**, **then**. and **else** is a terminal.
- 2. Nonterminal are syntactic variables that denote sets of strings. **stmt** and **expr** are nonterminal. The nonterminal define sets of strings that help define the language generated by the grammar
- 3. Start Symbol In a grammar. one nonterminal is distinguished as the start symbol.
- 4. The productions of a grammar specify the manner in which the terminals and nonterminal can be combined to form strings.
- ❖ Each production consists of a nonterminal. followed by an arrow (sometimes the symbol **= is** used in place of the **arrow**), followed by a string of nonterminal and terminals.

Arithmetic Expressions

- ❖ Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, division and Module.
- ❖ Here is one possible

Expr \rightarrow | **Expr Op Expr**

Expr \rightarrow | (**Expr**)

Expr \rightarrow | **id**

Op \rightarrow | **+**

Op \rightarrow | **-**

Op \rightarrow | *****

Op \rightarrow | **/**

Op \rightarrow | **%**

The nonterminal symbols are

V = {**Expr**, **OP**}

The terminal symbols are

T = {**(,),id,+, -, *, /, %**}

How many Production are

P = **3** for **Expr** and **5** for **Op**

Total 8 productions

S = Start Symbol **is Expr**

Example of CFG

- ❖ Suppose we want to describe only Addition and multiplication expressions and other CFG examples.
- ❖ Here is one possible

Expr \rightarrow | **Expr** + **Expr**

Expr \rightarrow | **Expr** * **Expr**

Expr \rightarrow | **id**

V = {**Expr**}

T = {**id**, +, *} 


P = 3

S = **Expr**

S \rightarrow | **aS** | **Sa** | **a**

V = {**S**}

T = {**a**}

P = 3 

S = **S**

S \rightarrow | **aSbS** | **bSaS** | **e**

V = {**S**}

T = {**a**, **b**, **e**} 

P = 3

S = **S**

Notational Conventions

- ❖ To avoid always having to state that "these are the terminals," "these are the nonterminal", and so on. we shall employ the following notational conventions with regard to grammars
1. These symbols are **terminals**:
 - a) Lower-case letters early in the alphabet such as *a*, *b*, *c*.
 - b) Operator symbols such as +, -, *, / etc.
 - c) Punctuation symbols such as parentheses, comma.
 - d) The digits 0, 1,, 9.
 - e) Boldface strings such as **id** or **if**.
 2. These symbols are **nonterminal**:
 - a) Upper-case letters early in the alphabet such as **A**, **B**, **C**.
 - b) The letter **S**, which. when it appears, is usually the **start** symbol,
 - c) Lower-case italic names such as **expr** or **Stmt**.
 3. Upper-case letters late in the alphabet. such as **X**, **Y**, **Z**. represent grammar symbols. that is, either nonterminal or terminals.

Notational Conventions

4. Lower-case letters late in the alphabet, chiefly $u, v \dots z$, represent strings of terminals.
 5. Lower-case Greek letters, α, β, γ it for example, represent strings or grammar symbols. Thus, a generic production could be written as $A \rightarrow \alpha$, indicating that there is a single nonterminal A on the left of the arrow (the left side of the production) and a string of grammar symbols α to the right of the arrow (the right side of the production).
 6. if $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_k$ are n productions with A on the left (we call them A -productions), we may write $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$. We call $\alpha_1, \alpha_2, \dots, \alpha_k$ the alternatives for A .
 7. Unless otherwise stated. the left side of the first production is the start symbol.
- ❖ Using these shorthand's, we could write the grammar of Example concisely as
- $$\text{Expr} \rightarrow \text{Expr Expr} \mid (\text{Expr}) \mid \text{id}$$
- $$A \rightarrow + \mid - \mid * \mid / \mid \%$$

Derivation

- ❖ Productions are treated as **rewriting** rules to generate a string this process is called **Derivation**.
- ❖ Show that a **sentence** is in the **grammar** (**valid program**)
 - Start with the **start symbol**
 - **Repeatedly** replace **one** of the **non-terminals** by a **right-hand side** of a **production**
 - **Stop** when the **sentence** contains **terminals** only
- ❖ At **each** step, we choose a **non-terminal** to replace.
 - *This **choice** can lead to different derivations.*
- ❖ **There are two categories of Derivation**
 - 1. A left-most derivation**
 - 2. A right-most derivation**

Left-most derivation

Definition. A **left-most derivation** of a sentential form is one in which rules transforming the **left-most** nonterminal are always applied

Example

| | | |
|---|---------------|---------------------------|
| $\text{Expr} \rightarrow \text{Expr} + \text{Expr}$ | \rightarrow | $V = \{\text{Expr}\}$ |
| $\text{Expr} \rightarrow \text{Expr} * \text{Expr}$ | | $T = \{\text{id}, +, *\}$ |
| $\text{Expr} \rightarrow \text{id}$ | | $P = 3$ |
| | | $S = \text{Expr}$ |

String $\text{id} + \text{id} * \text{id} \quad \Rightarrow \quad 2 + 3 * 4$

$\text{Expr} \Rightarrow \text{Expr} + \text{Expr}$

$\text{Expr} \Rightarrow \text{Id} + \text{Expr}$

$\text{Expr} \Rightarrow \text{Id} + \text{Expr} * \text{Expr}$

$\text{Expr} \Rightarrow \text{Id} + \text{Id} * \text{Expr}$

$\text{Expr} \Rightarrow \text{Id} + \text{Id} * \text{Id}$

$\text{Expr} \Rightarrow \text{Expr} * \text{Expr}$

$\text{Expr} \Rightarrow \text{Expr} + \text{Expr} * \text{Expr}$

$\text{Expr} \Rightarrow \text{Id} + \text{Expr} * \text{Expr}$

$\text{Expr} \Rightarrow \text{Id} + \text{Id} * \text{Expr}$

$\text{Expr} \Rightarrow \text{Id} + \text{Id} * \text{Id}$

Right-most derivation

Definition. A **right-most derivation** of a sentential form is one in which rules transforming the right-most nonterminal are always applied

Example $\text{Expr} \rightarrow \text{Expr} + \text{Expr}$
 $\text{Expr} \rightarrow \text{Expr} * \text{Expr}$
 $\text{Expr} \rightarrow \text{id}$

$V = \{\text{Expr}\}$
 $T = \{\text{id}, +, *\}$
 $P = 3$
 $S = \text{Expr}$

String $\text{id} + \text{id} * \text{id} \Rightarrow 2 + 3 * 4$

$\text{Expr} \Rightarrow \text{Expr} + \text{Expr}$

$\text{Expr} \Rightarrow \text{Expr} + \text{Expr} * \text{Expr}$

$\text{Expr} \Rightarrow \text{Expr} + \text{Expr} * \text{Id}$

$\text{Expr} \Rightarrow \text{Expr} + \text{Id} * \text{Id}$

$\text{Expr} \Rightarrow \text{Id} + \text{Id} * \text{Id}$

$\text{Expr} \Rightarrow \text{Expr} * \text{Expr}$

$\text{Expr} \Rightarrow \text{Expr} * \text{Id}$

$\text{Expr} \Rightarrow \text{Expr} + \text{Expr} * \text{Id}$

$\text{Expr} \Rightarrow \text{Expr} + \text{Id} * \text{Id}$

$\text{Expr} \Rightarrow \text{Id} + \text{Id} * \text{Id}$

Cont...

Derivate string $-(id+id)$ from G1

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$ (LMD) or

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$ (RMD) or

Parser Tree and derivation

Parsing is the process of checking that a **string** is in the **CFG** for your programming language. It is usually coupled with creating an abstract syntax tree.

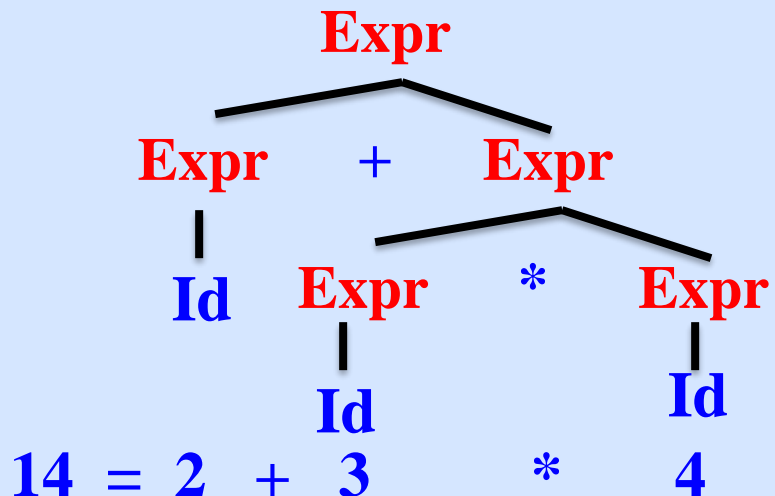
Expr \Rightarrow **Expr** + **Expr**

Expr \Rightarrow **Id** + **Expr**

Expr \Rightarrow **Id** + **Expr** * **Expr**

Expr \Rightarrow **Id** + **Id** * **Expr**

Expr \Rightarrow **Id** + **Id** * **Id**



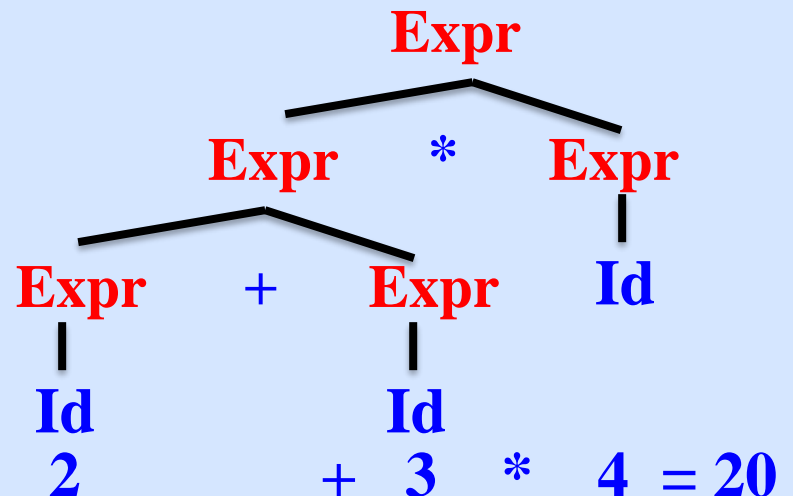
Expr \Rightarrow **Expr** * **Expr**

Expr \Rightarrow **Expr** + **Expr** * **Expr**

Expr \Rightarrow **Id** + **Expr** * **Expr**

Expr \Rightarrow **Id** + **Id** * **Expr**

Expr \Rightarrow **Id** + **Id** * **Id**



Ambiguity

- ❖ If a grammar has more than **one derivation** for a **single sentential form**, then it is **ambiguous**.
 - ❖ A CFG is **said** to be **ambiguous** if there is at **least one** string with **two or more parse trees**.
 - ❖ **Note** that **ambiguity** is a property of **purely grammars** its called (CFG), not **languages**.
 - ❖ **There** is no **algorithm** for converting **ambiguous** grammar into an **unambiguous** one.
-
- ❖ We say that a grammar is an Ambiguous, if there is
 - **Two leftmost derivations**
 - **Two rightmost derivations**
 - **Two parse trees**

Resolving ambiguity

Ambiguity may be **eliminated** by Applying Disambiguated rule

1. **Left factoring (rearranging the grammar),**
2. **Left Associative and**
3. **Precedence**

1. **Left factoring (rearranging the grammar),**

- ❖ **Sometimes**, we can transform a grammar to have this property:
- ❖ For each **non-terminal** A find the **longest prefix α** common to **two or more** of its **alternatives**.

if $\alpha \neq \epsilon$ then replace all of the A productions

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

$$A \rightarrow aAB \mid aA \mid a???$$

With

$$A \rightarrow aA'$$

$$A \rightarrow \alpha A'$$

$$A' \rightarrow AB \mid A \mid \text{epislon}$$

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

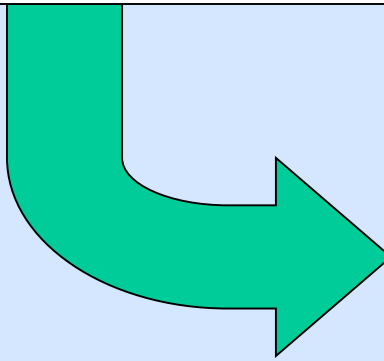
where A' is fresh

Repeat until no two alternatives for a single **non-terminal** have a **common prefix**.

Left factoring

| | |
|-----------------------------------|---|
| $\langle \text{expr} \rangle ::=$ | $\langle \text{term} \rangle + \langle \text{expr} \rangle$ |
| | $\langle \text{term} \rangle - \langle \text{expr} \rangle$ |
| | $\langle \text{term} \rangle$ |
| $\langle \text{term} \rangle ::=$ | $\langle \text{factor} \rangle * \langle \text{term} \rangle$ |
| | $\langle \text{factor} \rangle / \langle \text{term} \rangle$ |
| | $\langle \text{factor} \rangle$ |

Example Two **non-terminals** must be **left-factored**:

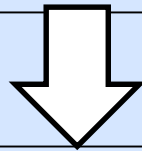


| | |
|------------------------------------|--|
| $\langle \text{expr} \rangle ::=$ | $\langle \text{term} \rangle \langle \text{expr}' \rangle$ |
| $\langle \text{expr}' \rangle ::=$ | $+ \langle \text{expr} \rangle$ |
| | $- \langle \text{expr} \rangle$ |
| | ϵ |
| $\langle \text{term} \rangle ::=$ | $\langle \text{factor} \rangle \langle \text{term}' \rangle$ |
| $\langle \text{term}' \rangle ::=$ | $* \langle \text{term} \rangle$ |
| | $/ \langle \text{term} \rangle$ |
| | ϵ |

Left factoring

1. Left factor (rearranging the grammar),

| | | |
|---------------------|--|--|
| <stmt> | | if <expr> then <stmt> |
| | | if <expr> then <stmt> else <stmt> |
| | | ... |



| | | |
|--------------------------|--|--|
| <stmt> | | <matched> |
| | | <unmatched> |
| <matched> | | if <expr> then <matched> else <matched> |
| | | Other |
| <unmatched> | | if <expr> then <stmt> |
| | | if <expr> then <matched> else <unmatched> |

This generates the **same language** as the ambiguous grammar, but applies the **common sense** rule:

— *match each else with the closest unmatched then*

Resolving ambiguity

2. Left Associative:- the grammar grow on left side

❖ applied left recursive which means left most derivation evaluate first.

❖ Left recursive:- means the left most symbol in the RHS is equal to LHS.

❖ Left recursive:- means the left most symbol in the RHS is equal to LHS.

❖ E.g.

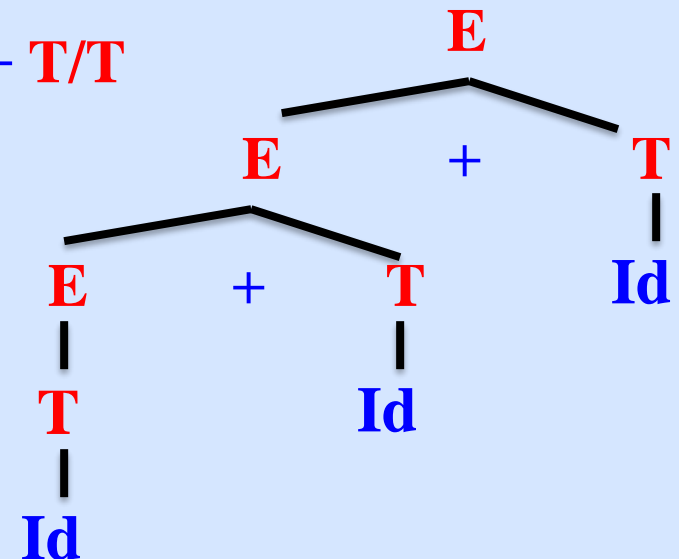
$E \rightarrow E + E$

$E \rightarrow id$

String $id + id + id$

$E \Rightarrow E + T/T$

$T \Rightarrow Id$



Resolving ambiguity

2. **Precedence** concerned about the priority of operators

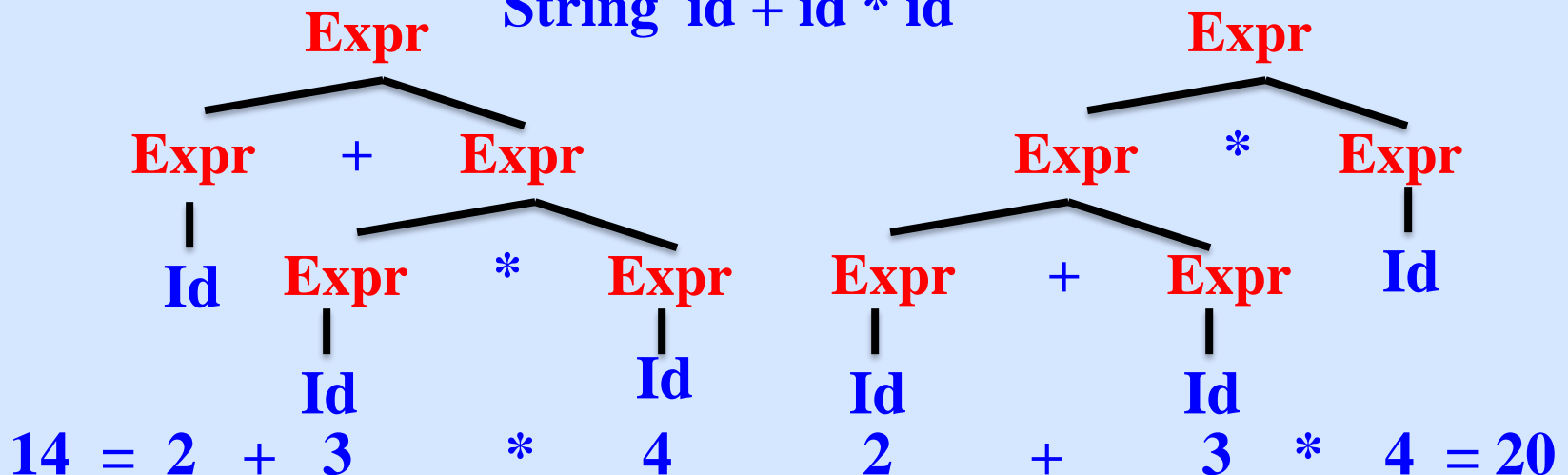
- ❖ **The least precedence** get **closest to** the **start symbol**
- ❖ **The highest precedence** get **farthest** from **start symbol**
- ❖ **Example** $\text{id} + \text{id} * \text{id}$

$\text{Expr} \rightarrow | \text{Expr} + \text{Expr}$

$\text{Expr} \rightarrow | \text{Expr} * \text{Expr}$

$\text{Expr} \rightarrow | \text{id}$

String $\text{id} + \text{id} * \text{id}$



Resolving ambiguity

2. **Precedence** concerned about the priority of operators

Example $\text{id} + \text{id} * \text{id}$

$E \rightarrow E + E$

$E \rightarrow E * E$

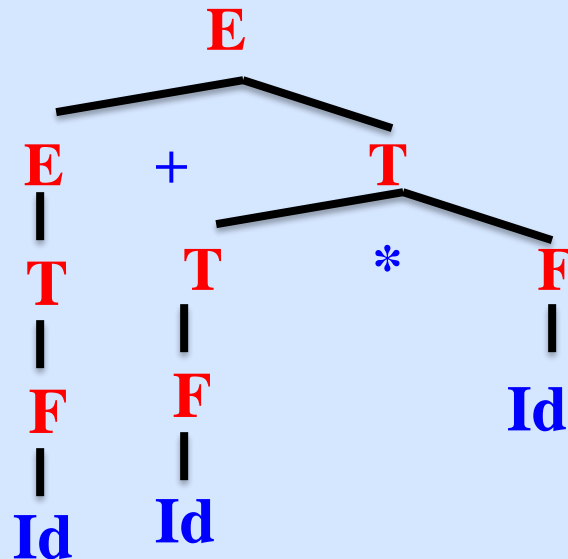
$E \rightarrow \text{id}$

$E \rightarrow E + T/T$

$T \rightarrow T * F/F$

$F \rightarrow \text{id}$

String $\text{id} + \text{id} * \text{id}$



$$14 = 2 + (3 * 4)$$

Resolving ambiguity

2. **Precedence** concerned about the priority of operators

Example $\text{id} + \text{id} * \text{id}$

$E \rightarrow | E \text{ or } E$

$E \rightarrow | E \text{ and } E$

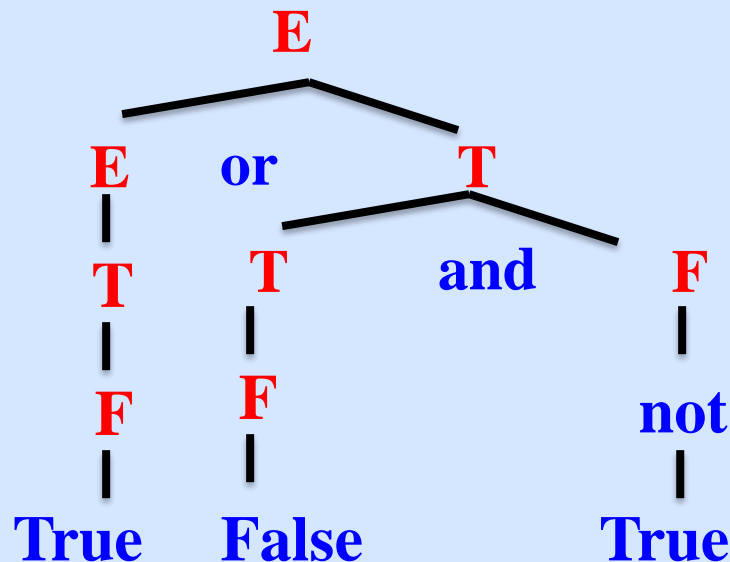
$E \rightarrow | \text{not } E$

$E \rightarrow | E \text{ or } T/T$

$T \rightarrow | T \text{ and } F/F$

$F \rightarrow | \text{not } F/\text{true}/\text{false}$

String $\text{id or id and (not id)}$



True = **T or (F and (not T))**

Resolving ambiguity

2. **Precedence** concerned about the priority of operators

Example $\text{id} + \text{id} * \text{id}$

$E \rightarrow | E \text{ or } E$

$E \rightarrow | E \text{ and } E$

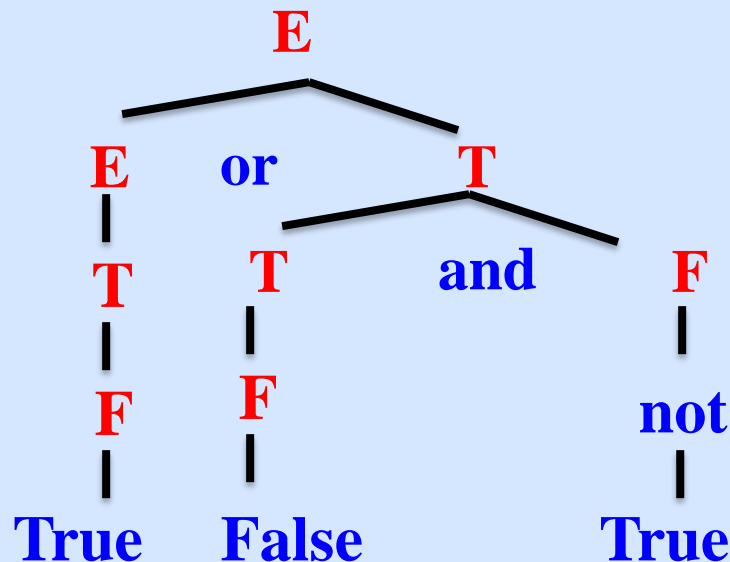
$E \rightarrow | \text{not } E$

$E \rightarrow | E \text{ or } T/T$

$T \rightarrow | T \text{ and } F/F$

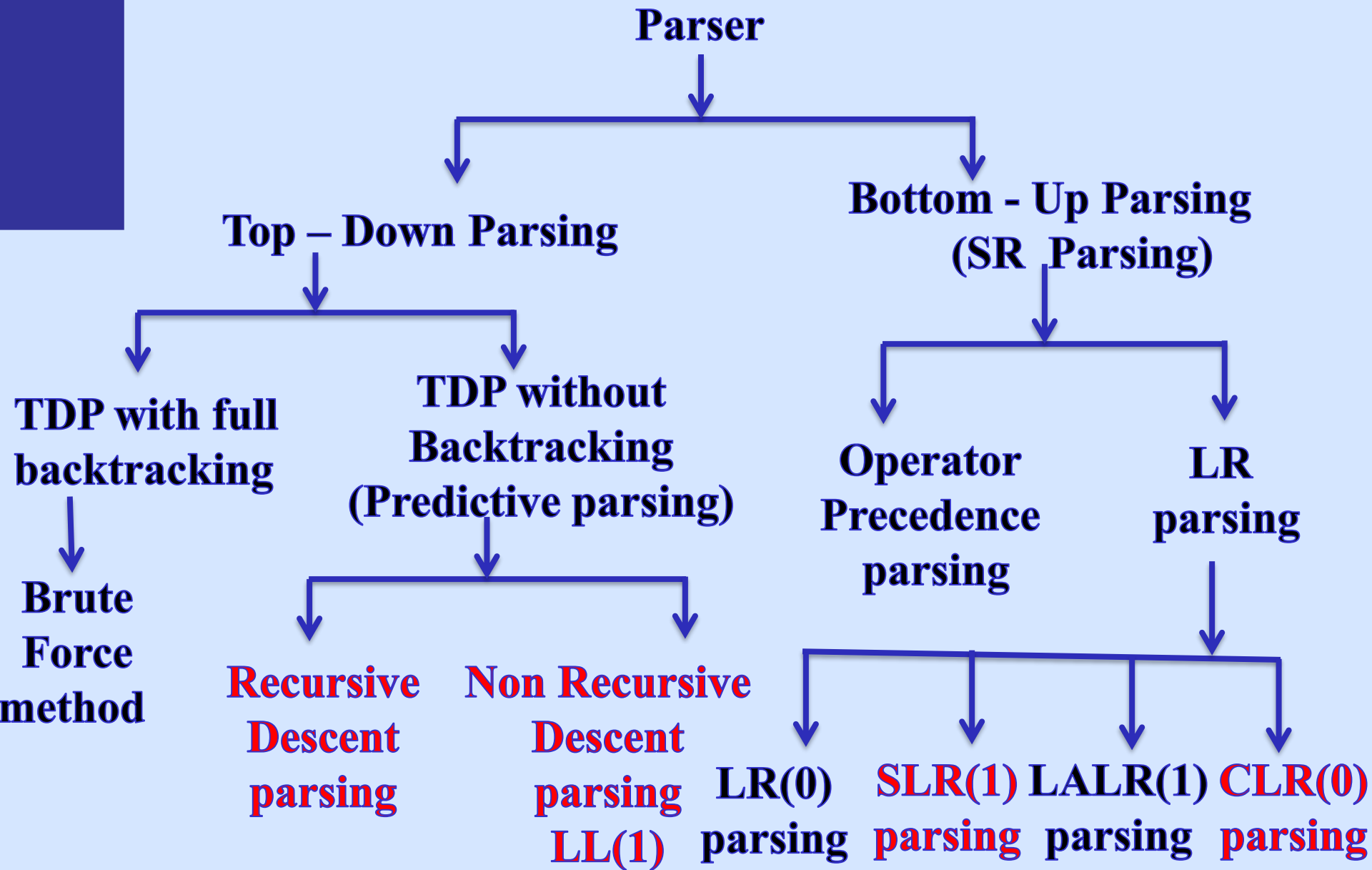
$F \rightarrow | \text{not } F/\text{true}/\text{false}$

String id or id and id



True = T or (F and (not T))

Parser



Parser

- ❖ A **parse** tree may be viewed as a **graphical representation** for a **derivation** that filters out the choice **regarding replacement** order.
- ❖ Mainly there are **two categories** of parse tree
 1. **Top-down parser**
 2. **Bottom-Up parser**

Top-down parser:

- starts at the **root** of **derivation** tree and fills in
- picks a **production** and tries to **match** the input
- some grammars are **backtrack-free** (**predictive**)

Bottom-up parser:

- starts at the **leaves** and fills in
- Up to a **state** valid for **legal first tokens**
- uses a **stack** to store both **state** and **sentential** forms

Top-Down Parser

- ❖ A **Top-down parser** tries to create a **parse tree** from the **root towards** the leafs **scanning** input from **left to right**
- ❖ It can be also **viewed** as finding a **leftmost derivation** for an input **string**

❖ **Example:** $\text{id} + \text{id} * \text{id}$

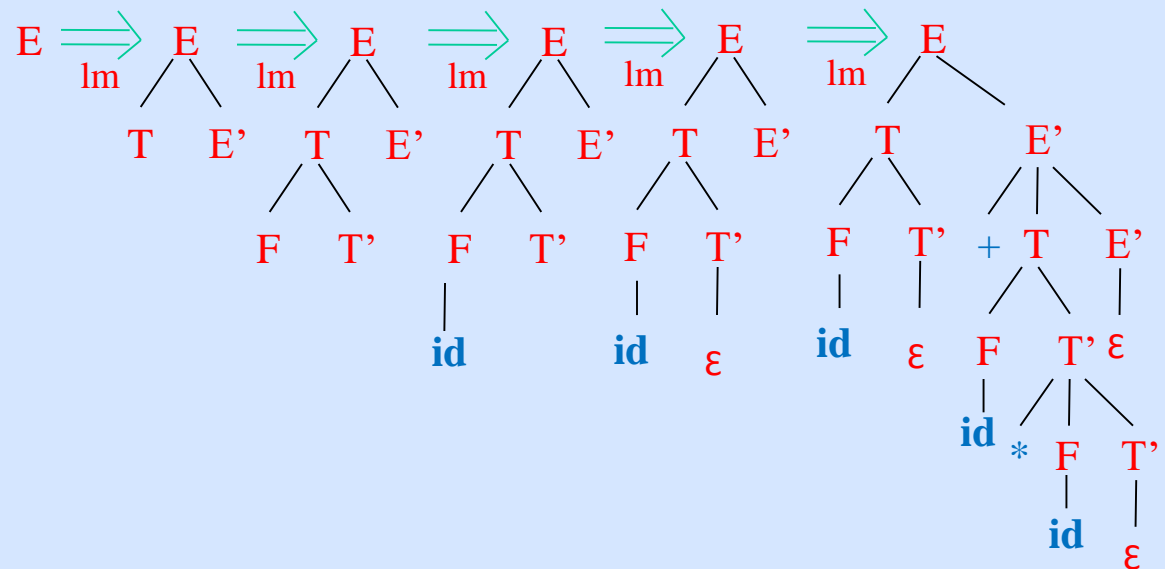
$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

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



Bottom-up parser

- ❖ **Constructs parse tree** for an input string **beginning** at the **leaves** (the **bottom**) and working **towards** the **root** (the top)
- ❖ Start from **leaves** to **root** (start Symbol)
- ❖ **Example: id * id**

E -> **E + T | T**

T -> **T * F | F**

F -> **| id**

id*id

F * id

|
id

T * id

|
id

T * **F**

|
id

|
id

T
├── **T** * **F**
└── **F**

|
id

|
id

|
id

|
id

|
id

|
id

|
id

E

|
id

|
id

|
id

|
id

|
id

|
id

|
id

Top-down parser

1. TDP with full backtracking

- ❖ **Based** on the **information** the parser **currently** has about the input, a **decision** is made **to go** with one **particular production**.
- ❖ If this choice **leads** to a **dead end**, the **parser** would have to **backtrack** to that **decision** point, moving **backwards** through the **input**, and
- ❖ start again **making** a **different** choice and so on **until** it **either** found the **production** that was the **appropriate one** or **ran out of choices**
- For example, **consider** this **simple grammar**:

S \rightarrow **bab** | **bA**

A \rightarrow **d** | **cA**

A \rightarrow **cAd**

A \rightarrow **ab|a???**

String cad???

Top-down parser

- ❖ Let's follow **parsing** the input **bcd**.
In the **trace** below,
- ❖ the **column** on the **left** will be the **expansion** thus far,
- ❖ the **middle** is the **remaining input**, and
- ❖ the **right** is the action **attempted** at
- ❖ **each** step:

| | | |
|-----|-----|---------------------------------------|
| S | bcd | Try S \rightarrow bab |
| bab | bcd | match b |
| ab | cd | dead-end, backtrack |
| S | bcd | Try S \rightarrow bA |
| bA | bcd | match b |
| A | cd | Try A \rightarrow d |
| d | cd | dead-end, backtrack |
| A | cd | Try A \rightarrow cA |
| cA | cd | match c |
| A | d | Try A \rightarrow d |
| d | d | match d |
| | | Success! |

Top-down parser

2. TDP without Backtracking (Predictive parsing)

- ❖ A **predictive parser** is **characterized** by its **ability** to choose the **production** to **apply** solely on the **basis** of the **next input symbol** and the current **nonterminal** being **processed**.
- ❖ To **enable** this, the **grammar** must take a **particular** form.
- ❖ We call such a **grammar LL(1)**.
 - The **first "L"** means we scan the input from **left** to **right**; the
 - **second "L"** means we create a **leftmost derivation**; and
 - the **1** means one input **symbol** of **look ahead**.
- ❖ Informally, an LL(1) has no **left recursive**
- ❖ There are two predictive parser
 1. Recursive Parsing
 2. Non Recursive parsing

Recursive Descent parsing

1. A **recursive descent parser** consists of **several small functions**, one for each **nonterminal** in the **grammar**.
 - ❖ As we **parse** a **sentence**, we call the **functions** that **correspond** to the **left side nonterminal** of the **productions** we are applying. If these productions are **recursive**, we end up **calling the functions recursively**.
 - ❖ **Execution** begins with the **procedure** for **start symbol**
 - ❖ A **typical** procedure for a **non-terminal**

❖ Algorithm of Recursive Descent Parsing



```
void A() {  
    choose an A-production, A->X1X2..Xk  
    for (i=1 to k) {  
        if (Xi is a nonterminal  
            call procedure Xi();  
        else if (Xi equals the current input symbol a)  
            advance the input to the next symbol;  
        else /* an error has occurred */  
    } }
```

Recursive Descent parsing

E => **IdE'**

E' => **+ IdE'/e**

Id + Id

E' ()

```
{  
    If (L== '+')  
    {  
        match('+');  
        match(Id);  
        E'();  
    }  
    Else  
        return  
}
```

E ()

```
{  
    If (L== Id)  
    {  
        match(Id);  
        E'();  
    }  
}
```

match (char t)

```
{  
    If (l ==t);  
    {  
        l = getchar( );  
    }  
    Else  
        print("Error")  
}
```

main ()

```
{  
    l = getchar( );  
    E();  
    If (l == '$')  
    {  
        l = getchar( );  
        print("parsing successfully")  
    }  
}
```

Recursive Descent parsing

Example

E \Rightarrow **TE'**

E' \Rightarrow **+ TE' / e**

T \Rightarrow **FT'**

T' \Rightarrow *** FT' / e**

F \Rightarrow **Id / (E)**

Id + Id * Id

A

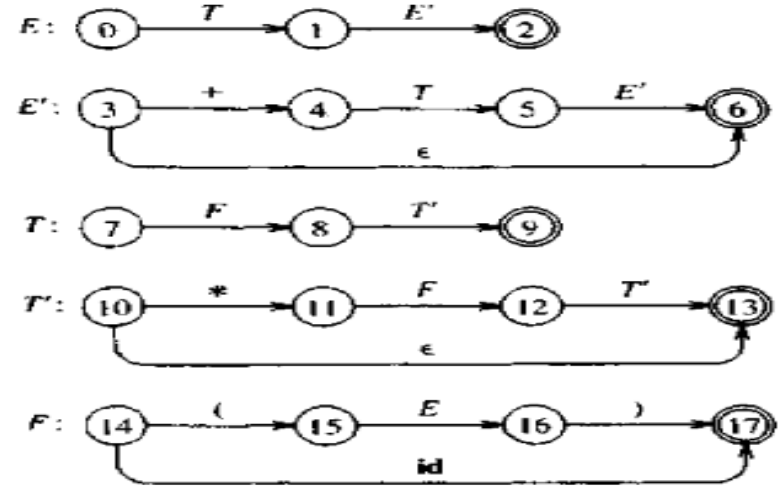


Fig. 4.10. Transition diagrams for grammar (4.11).

B

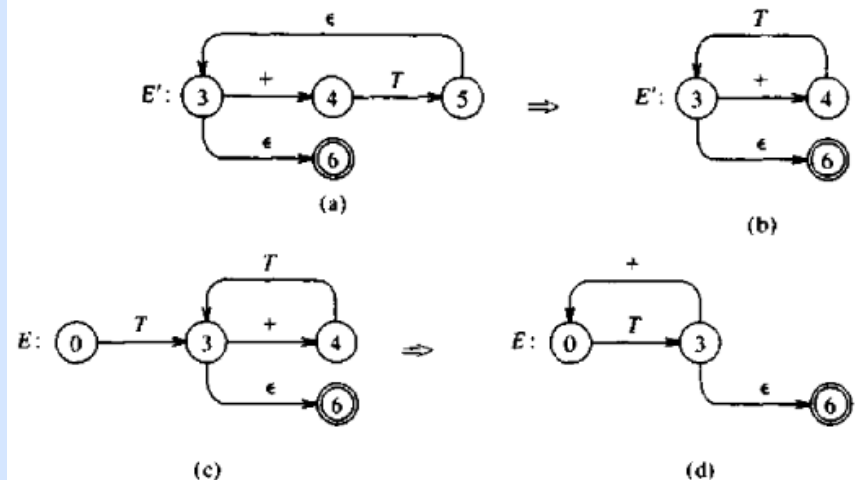


Fig. 4.11. Simplified transition diagrams.

C

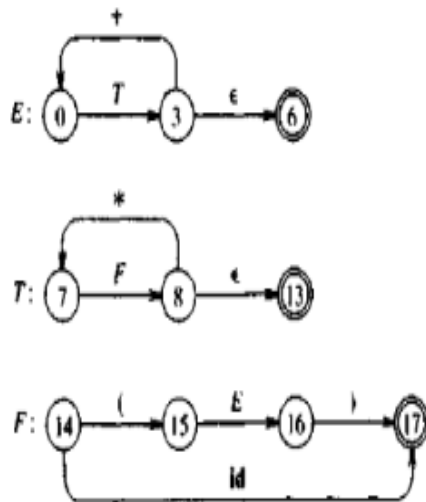


Fig. 4.12. Simplified transition diagrams for arithmetic expressions.

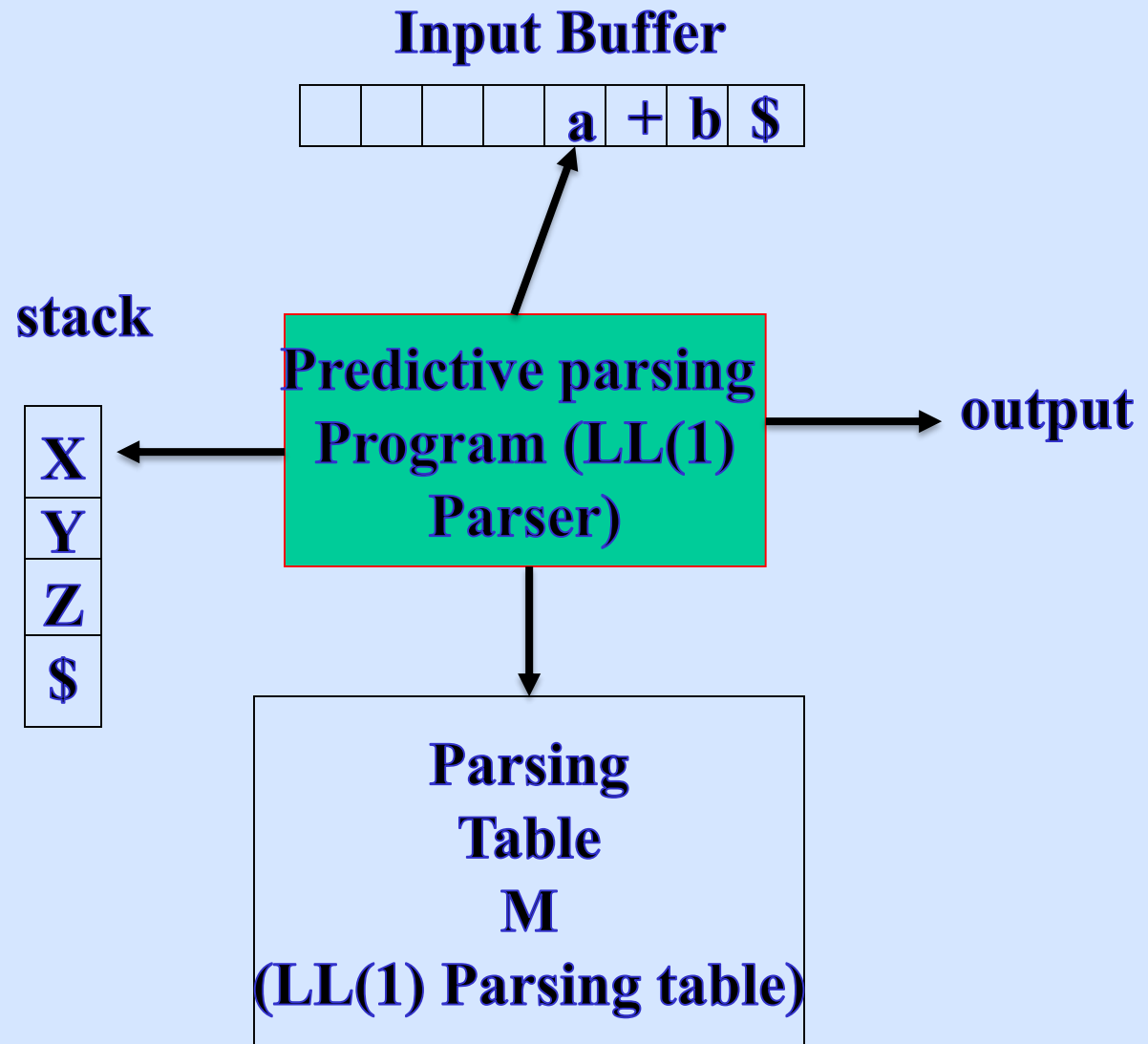
Non Recursive Descent parsing

- ❖ It is **possible** to build a **non recursive** predictive **parser** by **maintaining** a **stack explicitly**, rather than **implicitly** via **recursive calls**.
- ❖ The **key** problem during **predictive parsing** is that of **determining** the **production** to be **applied** for a **nonterminal**.
- ❖ A table-driven predictive parser has an input buffer. a stack. A parsing table. and an output stream.
- ❖ The input buffer contains the string to be **parsed**. followed by **\$**. a **symbol** used as a **right end marker** to **indicate** the **end** of the **input string**.
- ❖ The stack contains a **sequence** of **grammar** symbols with **\$** on the **bottom**, **indicating** the **bottom** of the **stack**.
- ❖ **Initially**. The **stack contains** the **start symbol** of the grammar on **top of \$**.
- ❖ The parsing table is a two dimensional array **M[A, a]**. where **A** is a **nonterminal**, and **a** a **terminal** or the **symbol \$**.

Non Recursive Descent parsing

- ❖ The parser is controlled by a program that behaves as follows. The program considers X , the symbol on (Top of the stack, and a , the current input symbol. These two symbols determine the action of the parser.
- ❖ There are three possibilities.
 1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
 2. If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
 3. If X is a nonterminal, the program consults entry $M[X, a]$ of the parsing table M . This entry will be either an X -production of the grammar or an error entry. If, for example, $M[X, a] = \{X \rightarrow UVW\}$, the parser replaces X on top of the stack by WVU (with U on top).

Non Recursive Descent parsing



Non Recursive Descent parsing Algorithm

Set **ip** point to the **first symbol** of **w\$**;

Set **X** to the top **stack symbol** and **a** the symbol pointed to by **ip**;

While (**X** \neq **\$**) { /* stack is not empty */

if (**X is a**) **pop** the **stack** and **advance ip**;

else if (**X** is a **terminal** or **\$**) **error()**;

else if (**M[X,a]** is an **error entry**) **error()**;

else if (**M[X,a] = X \rightarrow Y₁Y₂..Y_k**) {

output the **production X \rightarrow Y₁Y₂..Y_k**;

pop the **stack**;

push Y_k,...,Y₂,Y₁ on to the **stack** with **Y₁** on **top**;

 }

set X to the **top stack symbol**;

}

LL(1) Grammars

- **Predictive parsers** are those **recursive descent parsers** needing **no backtracking**
- **Grammars** for which we can create **predictive parsers** are called **LL(1)**
 - **The first L** means scanning input **from left to right**
 - **The second L** means **leftmost derivation**
 - **And 1** stands for **using** one **input symbol** for **look-ahead**
- A grammar **G** is **LL(1)** if and only if **whenever** **$A \rightarrow \alpha | \beta$** are **two distinct productions** of **G**, the following conditions hold:
 - For **no terminal** **a** do **α** and **β** both **derive strings beginning** with **a**
 - At **most one** of **α** or **β** can derive **empty string**
 - If **$\alpha \Rightarrow \epsilon$** then **$\beta$ does not** derive any string **beginning** with a **terminal** in **Follow(A)**.

LL(1) Grammars

- To represent the given grammar in LL(1) follow the following steps:-
 - Elimination of left recursion
 - Elimination of left factoring
 - Find first and follow of the given symbol
 - Construct parser table
 - Check whether the given string is accepted or not
- **Elimination of Left recursion**

$A \rightarrow A\alpha \mid \beta$ where β doesn't start with A then $A \rightarrow \beta A' \Rightarrow$

$A' \rightarrow \alpha A' \mid \epsilon$

Example

- $A \rightarrow Aba|Aa|a$ we can see it in two ways: the first one $\alpha_1 = ba$ $\alpha_2 = a$ and $\beta = a$, then first one is $A \rightarrow Aba|a$ and second one is $A \rightarrow Aa|a$. After that we can apply the above techniques of remove the redundant production. $A \rightarrow Aba|a \Leftrightarrow A \rightarrow \beta A'$ and $A' \rightarrow aA' | \epsilon$. Similarly do for the next one
- $A \rightarrow Ac|Aad|bc|c???$
- So/n $A \rightarrow bcA'|cA' \Leftrightarrow$ for the first with β_1 and β_2 and $A' \rightarrow cA'|adA' / \epsilon$

left factoring

- As we have seen in the pervious class to when one production is ambiguity we must eliminate it by left factoring. The production will have common element

Example1

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

\Downarrow

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

$$A' \rightarrow bB \mid B$$

\Downarrow

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

$$A' \rightarrow bB \mid B$$

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

Example2

$$A \rightarrow \underline{a}d \mid \underline{a} \mid \underline{a}b \mid \underline{a}bc \mid b$$

\Downarrow

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

$$A' \rightarrow d \mid \varepsilon \mid \underline{b} \mid \underline{b}c$$

\Downarrow

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

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

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

First and Follow

- There are Two type Of Function in NRDP

1.First()

2.Follow()

- First() is set of **terminals** that **begins strings** derived from
- If $\alpha \Rightarrow \epsilon$ then is also in **First(ϵ)**
- In **predictive parsing** when we have $A \rightarrow \alpha | \beta$, if **First(α)** and **First(β)** are **disjoint sets** then we can **select appropriate A-production** by looking at the **next input**
- Follow(A), for any **nonterminal A**, is set of **terminals** a that can **appear immediately after A** in some **sentential form**
 - If we have $S \Rightarrow \alpha A a \beta$ for some α and β then a is in **Follow(A)**
- If A can be the **rightmost symbol** in some **sentential form**, then **\$** is in **Follow(A)**

Computing First

- To compute **First(X)** for all grammar symbols **X**, apply following rules until no more terminals or ϵ can be added to any **First set**:
 - If **X** is a terminal then **First(X) = {X}**.
 - If **X** is a non-terminal and **X → Y₁Y₂...Y_k** is a production for some **k ≥ 1**, then place a in **First(X)** if for some i a is in **First(Y_i)** and ϵ is in all of **First(Y₁), ..., First(Y_{i-1})** that is **Y₁...Y_{i-1} ⇒ ε**. if ϵ is in **First(Y_j)** for j=1,...,k then add ϵ to **First(X)**.
 - If **X → ε** is a production then add ϵ to **First(X)**
- Example!**
 - S → aA/A** first of S is {a, b}
 - A → b** first of A is {b}

Computing Follow

- To compute **First(A)** for all **non-terminals A**, apply following rules until nothing can be added to any follow set:
 1. Place **\$** in **Follow(S)** where **S** is the start symbol
 2. If there is a production **A \rightarrow α B β** then **everything** in **First(β)** except **ϵ** is in **Follow(B)**.
 3. If there is a production **A \rightarrow B** or a production **A \rightarrow α B β** where **First(β)** contains **ϵ** , then **everything** in **Follow(A)** is in **Follow(B)**
- **Example!**
 - **S \rightarrow aAc/A** follow of S is **{ \$ }**
 - **A \rightarrow b** follow of A is **{ c, \$ }**

Examples

➤ To find the first of the given grammar remember the above rules.

1. $S \rightarrow abc|def|ghi$ the $\text{first}(S) = \{a, d, g\}$

2. $S \rightarrow ABC|def|ghi$ the $\text{first}(S) = \text{first}(A) = \{a, b, c, d, g\}$

$A \rightarrow a|b|c$

$B \rightarrow b$

$D \rightarrow d$

3. $S \rightarrow ABC$

$A \rightarrow a|b| \epsilon$

$B \rightarrow c|d| \epsilon$

$C \rightarrow e|f| \epsilon$

The $\text{first}(S) = \text{First}(A) = \{a, b, \epsilon\}$ but we didn't write epsilon before the remaining symbol is present instead of writing epsilon directly goes to B The $\text{first}(S) = \text{first}(A)$ after reached to epsilon $\text{first}(B)$ then again goto $\text{first}(C)$ when you reached at epsilon finally ,
 $\text{First}(S) = \{a, b, c, d, e, f, \}$

Examples

➤ To find the follow of G: remember the above rules.

1. The follow of starting symbol is $\{\$ \}$

2. $S \rightarrow ACD$ the $\text{follow}(A) = \{a, b\}$ and $\text{follow}(D) = \text{follow}(S) = \{\$ \}$
 $C \rightarrow a/b$

2. $S \rightarrow aSbS | bSaS$ the $\text{follow}(S) = \{\$, b, a\}$

$A \rightarrow a|b|c$

$B \rightarrow b$

$D \rightarrow d$

4. $S \rightarrow ABC$ $\text{follow}(A) = \text{first}(B)$ since ϵ goes to

$A \rightarrow DEF$ to the next = $\text{first}(C)$ again ϵ

$B \rightarrow \epsilon$ then go to the starting symbol

$C \rightarrow \epsilon$ i.e. $\text{follow}(A) = \text{follow}(S) = \{\$ \}$

$D \rightarrow \epsilon$

First and Follow function

| | First() | Follow() |
|-------------------------------------|-----------|------------|
| S \Rightarrow ABCDE | {a, b, c} | { \$ } |
| A \Rightarrow a/ε | {a, ε} | {b, c} |
| B \Rightarrow b/ε | {b, ε} | {c} |
| C \Rightarrow c | {c} | {d, e, \$} |
| D \Rightarrow d/ε | {d, ε} | {e, \$} |
| E \Rightarrow e/ε | {e, ε} | { \$ } |

| | First() | Follow() |
|-------------------------------------|---------------|----------|
| S \Rightarrow Bb/Cd | {a, b, c, d } | { \$ } |
| B \Rightarrow aB/ε | {a, ε} | {b} |
| C \Rightarrow cC/ε | {c, ε} | {d} |

Construction of predictive parsing table

- For each **production** $A \rightarrow \alpha$ in **grammar** do the **following**:
 1. For **each terminal** a in $\text{First}(\alpha)$ add $A \rightarrow$ in $M[A, a]$
 2. If ϵ is in $\text{First}(\alpha)$, then for each **terminal** b in $\text{Follow}(A)$ add $A \rightarrow \epsilon$ to $M[A, b]$.
If ϵ is in $\text{First}(\alpha)$ and $\$$ is in $\text{Follow}(A)$, add $A \rightarrow \epsilon$ to $M[A, \$]$ as well
 3. If after performing the **above**, there is **no production** in $M[A, a]$ then set $M[A, a]$ to **error**

Example

- For the **production given below apply LL(1) TDP with sequence of steps that we have seen before**
- **Consider the grammar**
 - ✓ **Elastic left recursion**
 - ✓ **Elastic left factoring**
 - ✓ **Calculate First and follow**
 - ✓ **Construct parser table**
 - ✓ **Check weather the string is acceptable or not**

List the following tuples of production

V={ }

T={ }

No P=

S={ }

Construction of predictive parsing table

| | First() | Follow() |
|---|------------------|---------------|
| E \Rightarrow TE' | {Id, (} | {\$,)} |
| E' \Rightarrow + TE' / ϵ | {+, ϵ } | {\$,)} |
| T \Rightarrow FT' | {Id, (} | {+, \$,)} |
| T' \Rightarrow * FT' / ϵ | {*, ϵ } | {+, \$,)} |
| F \Rightarrow Id/(E) | {Id, (} | {*, +, \$,)} |

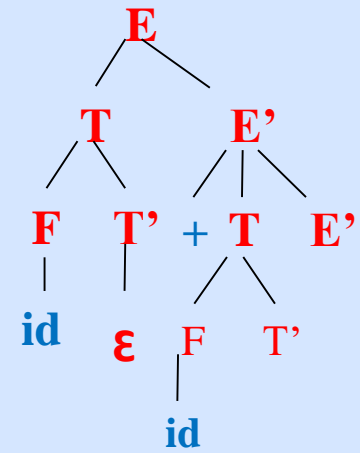
Construction of predictive parsing table

| | Id | + | * | (|) | \$ |
|----|---------------------|---------------------------|-----------------------|---------------------|---------------------------|---------------------------|
| E | $E \Rightarrow TE'$ | | | $E \Rightarrow TE'$ | | |
| E' | | $E' \Rightarrow + TE'$ | | | $E' \Rightarrow \epsilon$ | $E' \Rightarrow \epsilon$ |
| T | $T \Rightarrow FT'$ | | | $T \Rightarrow FT'$ | | |
| T' | | $T' \Rightarrow \epsilon$ | $T' \Rightarrow *FT'$ | | $T' \Rightarrow \epsilon$ | $T' \Rightarrow \epsilon$ |
| F | $F \Rightarrow Id$ | | | $F \Rightarrow (E)$ | | |

Construction of predictive parsing table

❖ Example $\text{Id} + \text{Id} * \text{Id}\$$

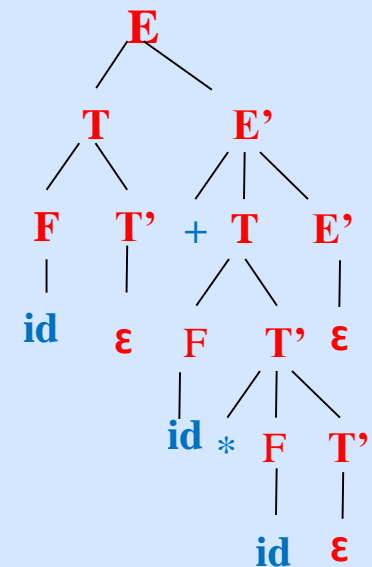
| Stack | Input | Output |
|--------------------|---------------------------------------|---------------------------|
| $\$E$ | $\text{Id} + \text{Id} * \text{Id}\$$ | |
| $\$E'T$ | $\text{Id} + \text{Id} * \text{Id}\$$ | $E \Rightarrow TE'$ |
| $\$E'T'F$ | $\text{Id} + \text{Id} * \text{Id}\$$ | $T \Rightarrow FT'$ |
| $\$E'T' \text{Id}$ | $\text{Id} + \text{Id} * \text{Id}\$$ | $F \Rightarrow \text{Id}$ |
| $\$E'T'$ | $+ \text{Id} * \text{Id}\$$ | |
| $\$E'$ | $+ \text{Id} * \text{Id}\$$ | $T' \Rightarrow \epsilon$ |
| $\$E'T +$ | $+ \text{Id} * \text{Id}\$$ | $E' \Rightarrow + TE'$ |
| $\$E'T$ | $\text{Id} * \text{Id}\$$ | |
| $\$E'T'F$ | $\text{Id} * \text{Id}\$$ | $T \Rightarrow FT'$ |
| $\$E'T' \text{Id}$ | $\text{Id} * \text{Id}\$$ | $F \Rightarrow \text{Id}$ |
| $\$E'T'$ | $* \text{Id}\$$ | |



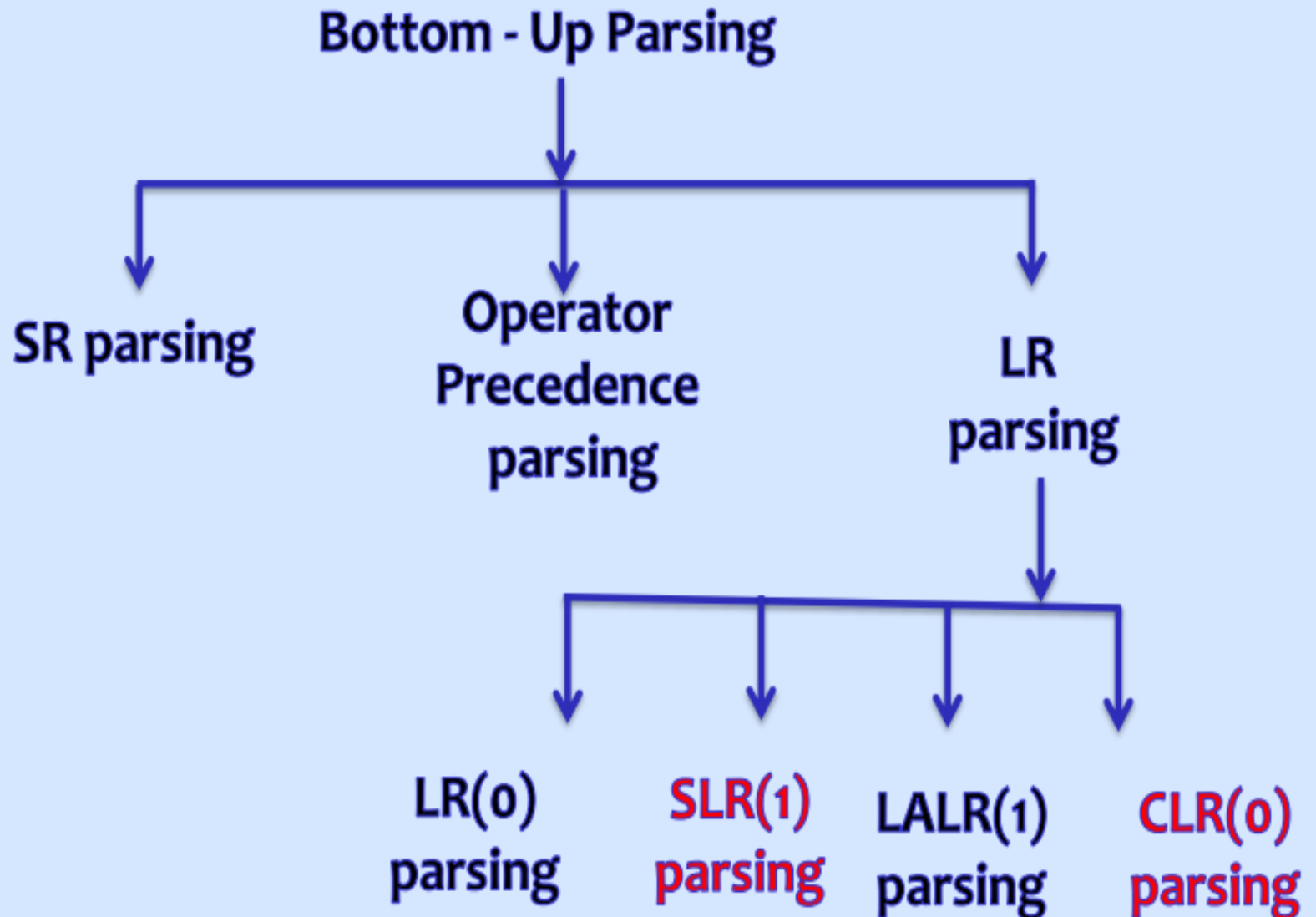
Construction of predictive parsing table

❖ Example $\text{Id} + \text{Id} * \text{Id}\$$

| Stack | Input | Output |
|--------------------|-----------------|---------------------------|
| $\$E'T'F *$ | $* \text{Id}\$$ | $T' \Rightarrow *FT'$ |
| $\$E'T'F$ | $\text{Id}\$$ | |
| $\$E'T' \text{Id}$ | $\text{Id}\$$ | $F \Rightarrow \text{Id}$ |
| $\$E'T'$ | $\$$ | |
| $\$E'$ | $\$$ | $T' \Rightarrow \epsilon$ |
| $\$$ | $\$$ | $E' \Rightarrow \epsilon$ |



Bottom Up parsing



Shift-reduce parser

- The **general** idea is to **shift** some **symbols** of input to the **stack** until a **reduction** can be **applied**
- At each **reduction step**, a specific **substring** matching the **body** of a **production** is **replaced** by the **nonterminal** at the **head** of the **production**
- The **key decisions** during **bottom-up** parsing are about **when to reduce** and about **what production to apply**
- A **reduction** is a **reverse** of a **step** in a **derivation**
- The **goal** of a **bottom-up** parser is to **construct** a **derivation** in **reverse**: that means **reverse of RMD**

– $E \Rightarrow T$

$\text{id} * \text{id}$

E

– $T \Rightarrow T * F$

$F * \text{Id}$

$T * F$

– $F \Rightarrow \text{id}$

$T * F$

$F * \text{Id}$

– $\Rightarrow \text{id} * \text{id}$

E

$\text{id} * \text{Id}$

Shift-reduce parser

Handle pruning

- A **Handle** is a **substring** that **matches** the **body** of a **production** and whose **reduction** represents one **step** along the **reverse** of a **rightmost derivation**
- **Example** $\text{id} + \text{id} * \text{id}$

| Right sentential form | Handle | Reducing production |
|-------------------------------------|-------------|---------------------------|
| $\text{id} + \text{id} * \text{id}$ | id | $E \rightarrow \text{id}$ |
| $E + \text{id} * \text{id}$ | id | $E \rightarrow \text{id}$ |
| $E + E * \text{id}$ | id | $E \rightarrow \text{id}$ |
| $E + E * E$ | $E * E$ | $E \rightarrow E * E$ |
| $E + E$ | $E + E$ | $E \rightarrow E + E$ |
| E | | |

Shift-reduce parser

- A **stack** is used to **hold grammar symbols**
- **Handle** always **appear** on **top** of the **stack**
- **Initial configuration:**

| Stack | Input |
|-------|-------|
| \$ | w\$ |

- **Acceptance configuration**

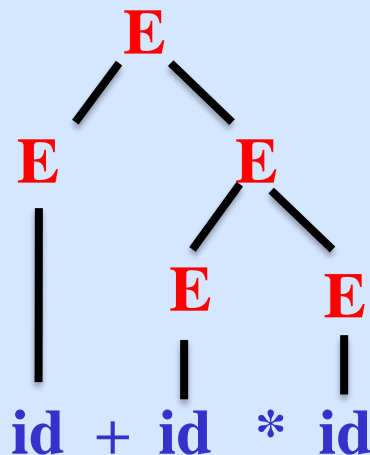
| Stack | Input |
|-------|-------|
| \$S | \$ |

Shift-reduce parser

- Basic operations:

- Shift
- Reduce
- Accept
- Error

- Example: $\text{id} + \text{id} * \text{id}$



| Stack | Input | Action |
|--------------|--------------|-------------------------------------|
| \$ | Id + id*id\$ | shift |
| \$id | +id*id\$ | reduce by $E \rightarrow \text{id}$ |
| \$E | +id*id\$ | shift |
| \$E+ | id*id\$ | shift |
| \$E + id | *id\$ | reduce by $E \rightarrow \text{id}$ |
| \$E + E | *id\$ | shift |
| \$E + E * | id\$ | shift |
| \$E + E * id | \$ | reduce by $E \rightarrow \text{id}$ |
| \$E + E * E | \$ | reduce by $E \rightarrow E * E$ |
| \$E + E | \$ | reduce by $E \rightarrow E + E$ |
| \$E | \$ | accept |

Fig. Configurations of Shift Reduce Parser on $\text{id} + \text{id} * \text{id}$

Operator Precedence parsing

- ❖ Operator Precedence parsing is mainly use to define mathematical operator.
- ❖ **Operator Grammar:** These grammars have the property that no production right side is **e** or has **two adjacent non terminals**.

- ❖ **Example**

E \Rightarrow **EAE** / **Id**

A \Rightarrow ***/+**

- ❖ The above production is not Operator grammar b/c These two **Nonterminal** are **side by side (adjacent)**

E \Rightarrow | **E * E**

E \Rightarrow | **E + E**

E \Rightarrow | **E - E**

E \Rightarrow | **E ^ E**

E \Rightarrow | **Id**

Operator Precedence parsing Algorithm

1. Set **ip** to point to the first symbol of $w\$$;
2. repeat forever
3. if **\$** is on top of the stack and **ip** points to **S** then
4. return
5. else begin.
6. let **a** be the topmost terminal symbol on the stack and let **b** be the symbol pointed to by **ip**;
7. if "**a** < **b** or **a** = **b** then begin
8. push **b** onto the stack;
9. advance **ip** to the next input symbol:
10. end;
11. else If **a** > **b** then
12. repeat
13. pop the stack
14. until the top stack terminal is related by <
15. to the terminal most recently popped
16. else error 0
17. end

| RELATION | MEANING |
|----------|--------------------------------------|
| $a < b$ | a "yields precedence to" b |
| $a = b$ | a "has the same precedence as" b |
| $a > b$ | a "takes precedence over" b |

Operator Precedence parsing

| RELATION | MEANING |
|---------------|--------------------------------------|
| $a < \cdot b$ | a "yields precedence to" b |
| $a \dot{=} b$ | a "has the same precedence as" b |
| $a \cdot > b$ | a "takes precedence over" b |

E \rightarrow | **E or E**

E \rightarrow | **E and E**

E \rightarrow | **not E**

E \rightarrow | **Id**

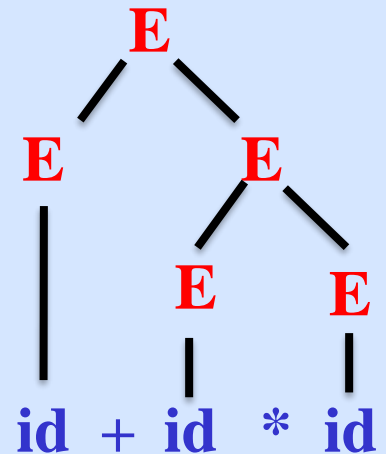
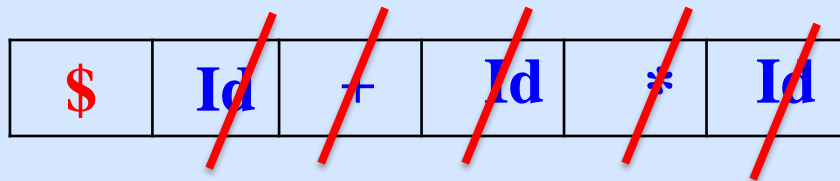
String id or id and id

Construct of Operator Precedence Relation table

$E \Rightarrow E + E$
 $E \Rightarrow E * E$
 $E \Rightarrow E * Id$

| | Id | + | * | \$ |
|----|-----------|-----------|-----------|-----------|
| Id | -- | $\cdot >$ | $\cdot >$ | $\cdot >$ |
| + | $< \cdot$ | $\cdot >$ | $< \cdot$ | $\cdot >$ |
| * | $< \cdot$ | $\cdot >$ | $\cdot >$ | $\cdot >$ |
| \$ | $< \cdot$ | $< \cdot$ | $< \cdot$ | -- |

Example
 $Id + Id * Id \$$



- ❖ Left side + and * get higher precedence
- ❖ The bottom stack is \$
- ❖ Top of stack \leq look ahead Push/Shift it
- ❖ Whenever we get or Top of stack is \geq Pop/Reduce

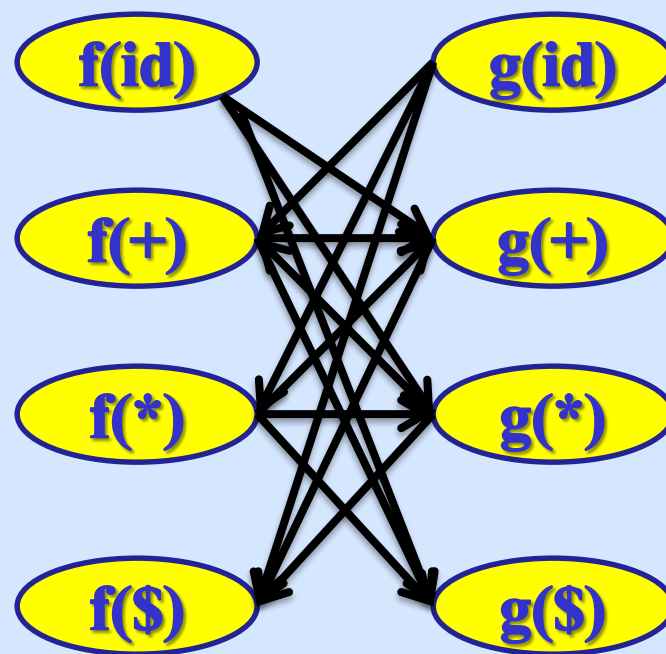
Construct of Operator Precedence Function table

- ❖ Disadvantage of operator precedence function table is size
- ❖ If we have n operator our relation table become n^2
- ❖ So to reduce the size of Relational table we are going to construct operator function table
- ❖ To construct function table we have to construct a graph with two functions $f(i)$ and $g(j)$

i

| | Id | + | * | \$ |
|----|----|----|----|----|
| Id | -- | •> | •> | •> |
| + | <• | •> | <• | •> |
| * | <• | •> | •> | •> |
| \$ | <• | <• | <• | -- |

j



Construct of Operator Precedence Function table

❖ W/h one is a longest path each function

$f(id) \longrightarrow g(*) \longrightarrow f(+) \longrightarrow g(+) \longrightarrow f(\$) = 4$

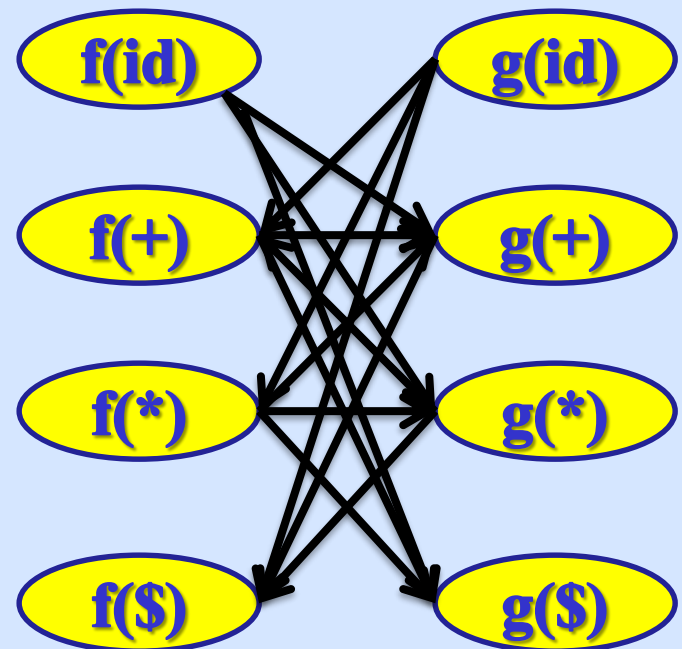
$g(id) \longrightarrow f(*) \longrightarrow g(*) \longrightarrow f(+) \longrightarrow g(+) \longrightarrow f(\$) = 5$

$f(i)$

| | Id | + | * | \$ |
|----|-----------|-----------|-----------|-----------|
| Id | -- | $\cdot >$ | $\cdot >$ | $\cdot >$ |
| + | $< \cdot$ | $\cdot >$ | $< \cdot$ | $\cdot >$ |
| * | $< \cdot$ | $\cdot >$ | $\cdot >$ | $\cdot >$ |
| \$ | $< \cdot$ | $< \cdot$ | $< \cdot$ | -- |

$g(j)$

| | Id | + | * | \$ |
|-------|----|---|---|----|
| $f()$ | 4 | 2 | 4 | 0 |
| $g()$ | 5 | 1 | 3 | 0 |



❖ there is no cycle

Construct of Operator Precedence Relation table

$E \rightarrow | E - E$

$E \rightarrow | E / E$

$E \rightarrow | Id$

String id or id and id

$E \rightarrow | E \text{ or } E$

$E \rightarrow | E \text{ and } E$

$E \rightarrow | Id$

String id or id and id

Construct of Operator Precedence Function table

From Relation table

| | + | - | * | / | ↑ | id | (|) | \$ |
|----|---|---|---|---|---|----|---|---|----|
| + | · | · | · | · | · | · | · | · | · |
| - | · | · | · | · | · | · | · | · | · |
| * | · | · | · | · | · | · | · | · | · |
| / | · | · | · | · | · | · | · | · | · |
| ↑ | · | · | · | · | · | · | · | · | · |
| id | · | · | · | · | · | · | · | · | · |
| (| · | · | · | · | · | · | · | · | · |
|) | · | · | · | · | · | · | · | · | · |
| \$ | · | · | · | · | · | · | · | · | · |

Operator-precedence relations.

❖ The Functional table is

| | + | - | * | / | ↑ | (|) | id | \$ |
|---|---|---|---|---|---|---|---|----|----|
| f | 2 | 2 | 4 | 4 | 4 | 0 | 6 | 6 | 0 |
| g | 1 | 1 | 3 | 3 | 5 | 5 | 0 | 5 | 0 |

Reading assignment

- **What is LR parsing**
- **Types of LR parsing and how it works?**
- **LR(0) parsing**
- **SLR(1) parsing**
- **LALR(1) parsing**
- **CLR(0) parsing**

Questions

