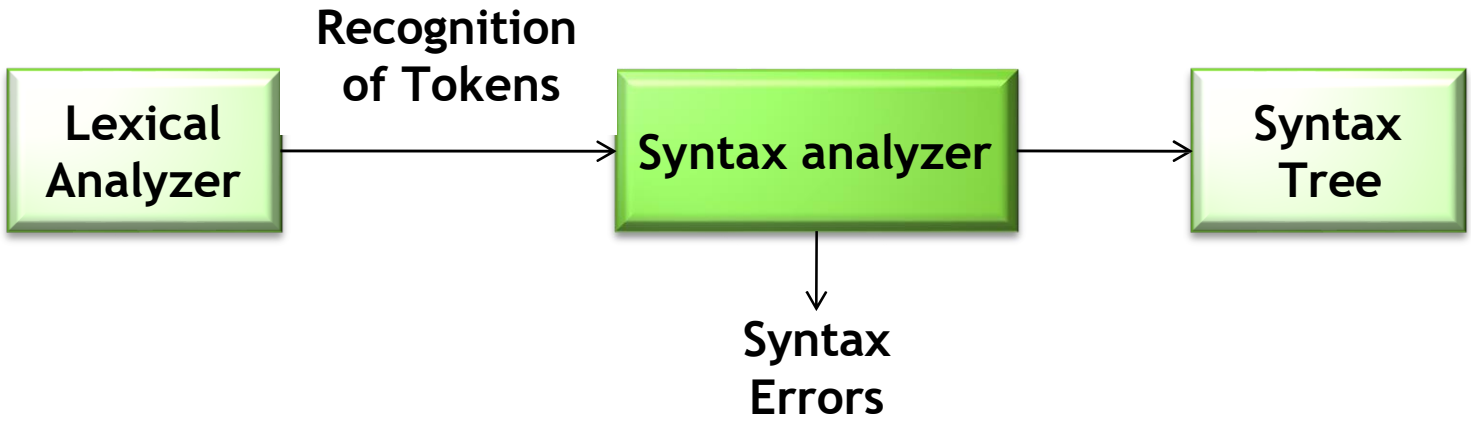
Unit - III

SYNTAX ANALYSIS

Unit - III
Agenda

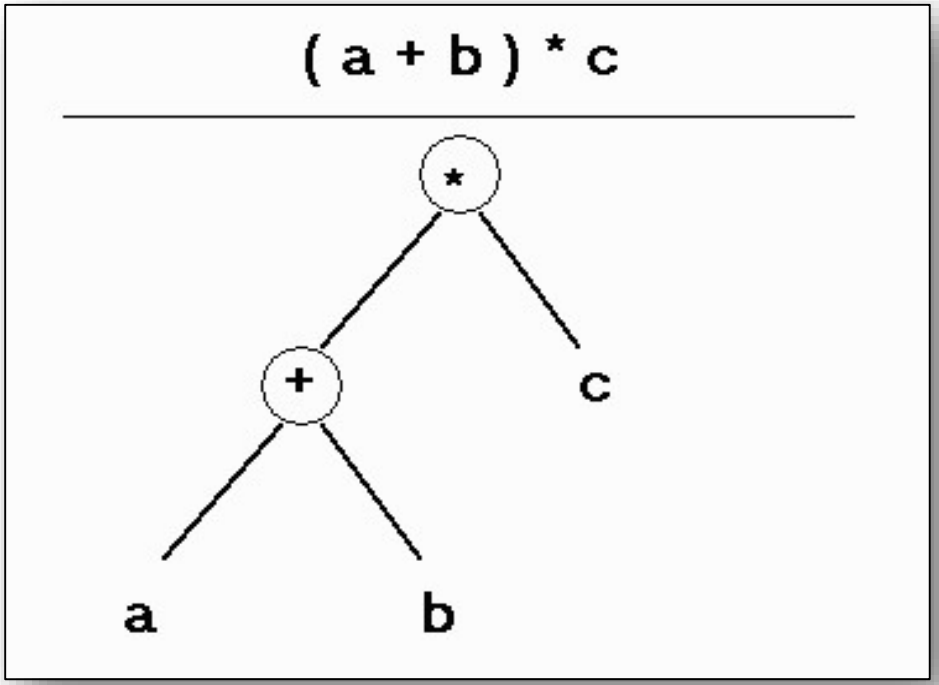
1. The need & role of the parser
2. Context Free Grammars
3. Top Down Parsing
4. Recursive descent Parser, LL(1) Parser
5. SR, LR, SLR parsing, LALR parser
6. Error Handling & Recovery
7. YAAC - Design of Syntax Analyzer

Introduction



- Lexical analysis → generate tokens
- Syntax analysis (parsing) → produce syntax tree
- The leaves of this tree are the tokens found by the lexical analysis, and if the leaves are read from left to right, the sequence is the same as in the input text.
  - **In addition to finding the structure of the input text, the syntax analysis must also reject invalid texts by reporting syntax errors.**

Syntax tee Example

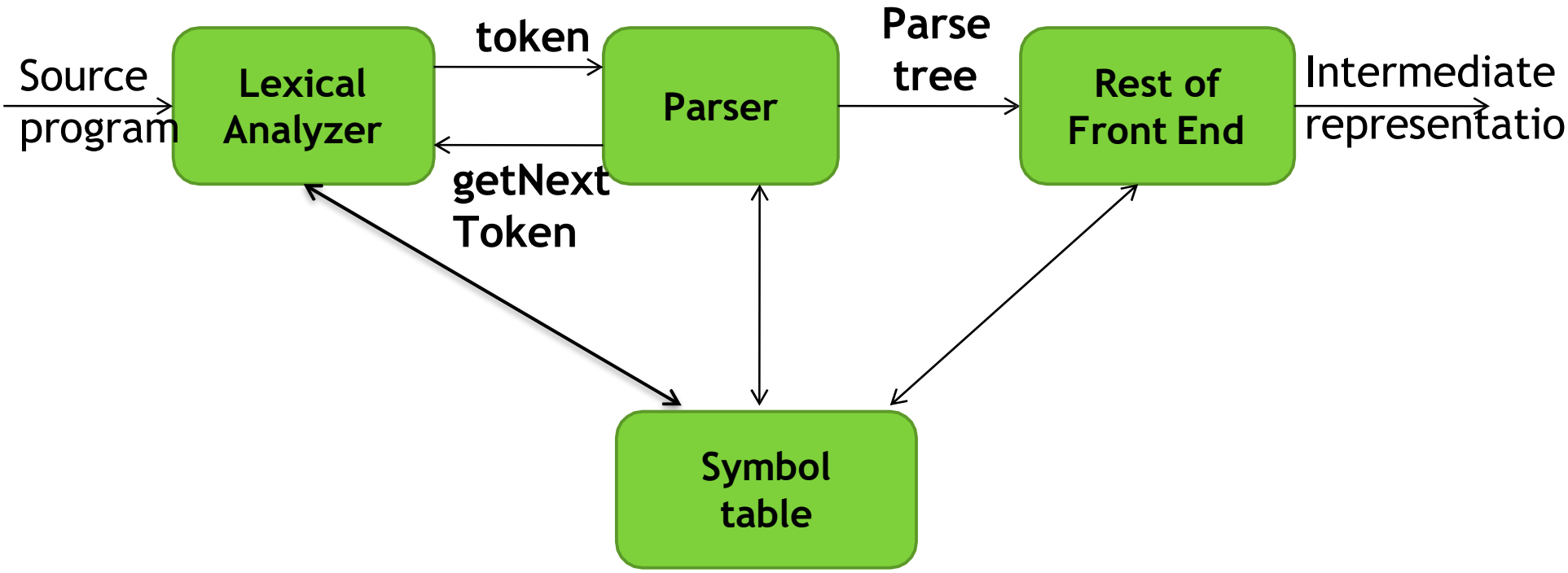


Contd…

- A notation suitable for human understanding is transformed into a machine-like low-level notation suitable for efficient execution. This process is called **parser generation**.

- The notation we use for human manipulation is **context-free grammars**

The role of parser



The role of parser - Contd…

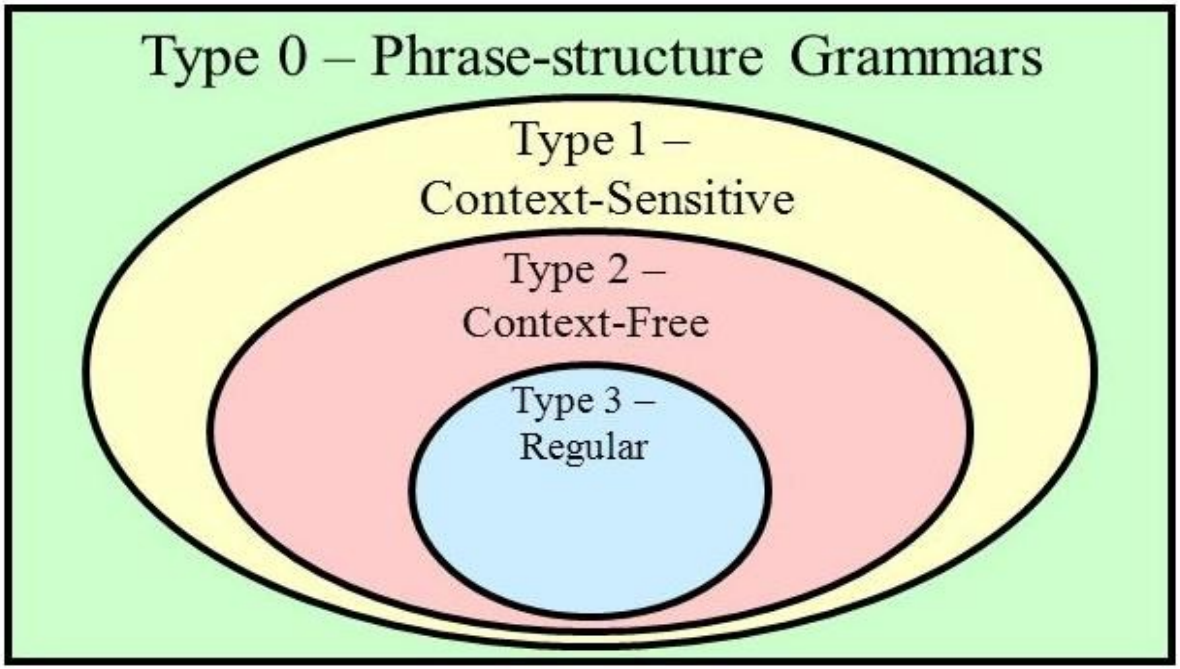
Input to Parser	• Sequence of tokens from Scanner
Output	• Parse Tree or Syntax Tree
Functions	• Verifies the structure generated by tokens based on **grammars** • Constructs Parse tree • Reports error and handles the Error
Issues	• Cannot detect errors Variable re-declaration, variable initialization before use, Data type mismatch

GRAMMARS

- Grammars are used to describe the syntax of the programming language. It specifies the structure of expressions and statements.

stmt → if (expr) stmt

expr denote Expression and stmt denote statement.



Context Free Grammars

- CFG → Type 2 Grammar
- Context Free Grammar (CFG) to specify the syntax of the language. Grammar normally describes the hierarchical structure of many programming language constructs.

Components in CFG

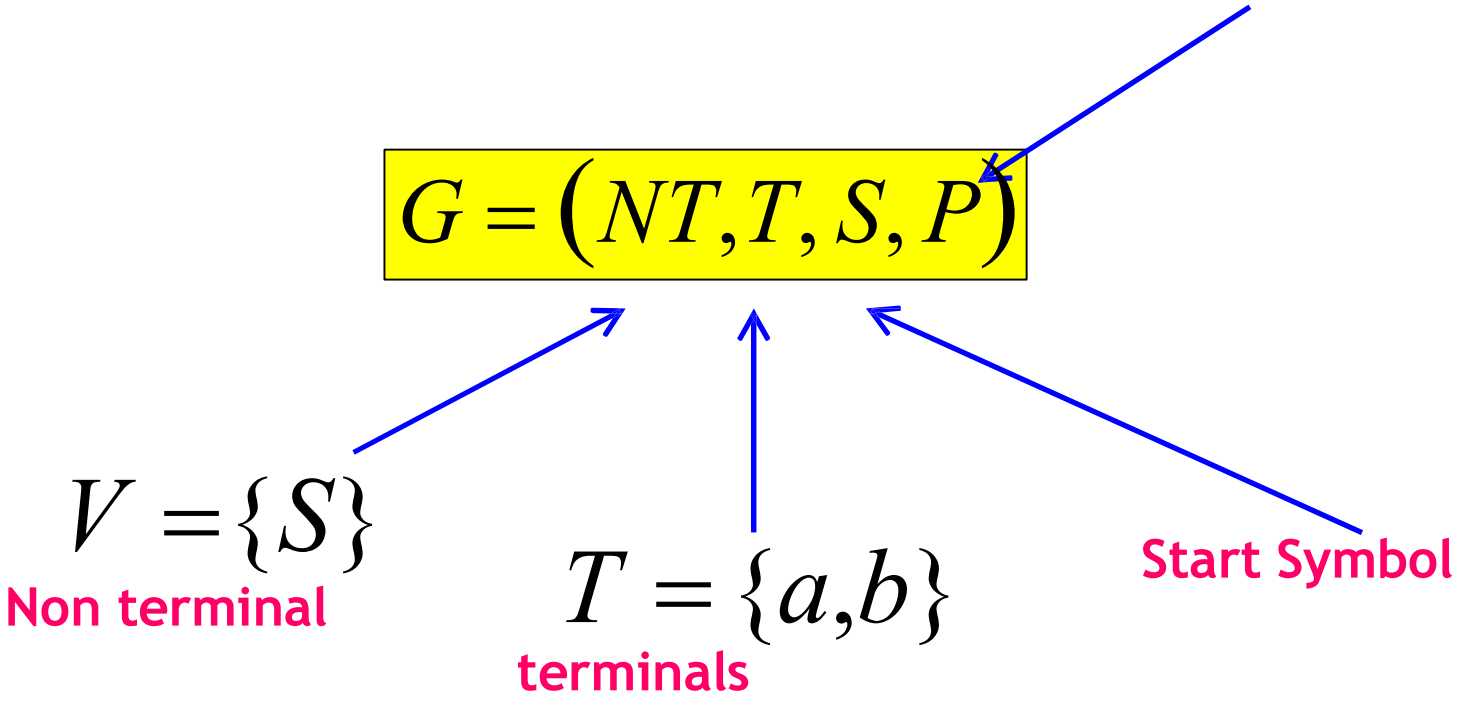
G ={Terminals, Non terminals, Start symbol, Productions}

- A set of tokens, called ***terminal*** symbols
  - Example:
    - Keywords(if, else, then, while, do)
    - operators(+,-,*,/)
    - Special symbols(,,:,;)
- A set of ***non terminals***
  - ***Example***
    - ***Stmt***
    - ***Expr***
- A ***start*** symbol.
  - One non terminal is selected as start symbol or designated symbol
- A set of ***productions***
  - Each production consists of a non terminal, called the ***left side*** of the production, an ***arrow***, and a sequence of tokens and/or non terminals, called the ***right side*** of the production.

Context Free Grammars

productions

$$P = \{S \to aSb,\ \ S \to \lambda\}$$



Context Free Grammar Example

Grammar:

Sequence of
terminals and non terminals

$$S \rightarrow aSb$$

$$S \rightarrow \in$$

Start Symbol

The right side
may be $\in$

Grammar: $S \rightarrow aSb$

$S \rightarrow \in$

Derivation of string : ab

$S \Rightarrow aSb \Rightarrow ab$

$S \rightarrow aSb$ $\qquad$ $S \rightarrow \in$

Grammar: $S \rightarrow aSb$

$S \rightarrow \in$

Derivation of string : $aabb$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$



$S \rightarrow aSb$ $\qquad$ $S \rightarrow \in$

Grammar:

$$S \to aSb$$

$$S \to \lambda$$

Other derivations:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb$$

$$\Rightarrow aaaaSbbbb \Rightarrow aaaabbbb$$

Hence, Language of the grammar is

$$L = \{a^n b^n \ : n \geq 0\}$$

Derivation, Reduction, Parse tree

- There are two basic requirements for a grammar that defines a language in a syntax analysis phase. They are
  - **To generate a valid string**
  - **To recognize a valid string**

- **Derivation:** is the process that generates a valid string with the help of the grammar.
- **Reduction**: is a process that recognizes a valid string
- **Parse tree:** pictorial representation of a derivation

Derivations (Start symbol S to Sentence ω)

- Derivation is a rewriting rule in which the non terminal on the left side is replaced by string on the right side of production.

- If there is a production, $A \rightarrow \gamma$, then replacement of A by γ eventually leads to **derivation**

- The derivation is defined by
  - **αAβ => αγβ**
  - α and β are arbitary strings

- Derivation in one step =>
- Derivation in zero or more steps $=>^*$
- Derivation in one or more steps $=>^+$

Example

- Consider the grammar for arithmetic expression as
  - **E → E + E | E * E | (E) | - E | id**
- Derive the string :
  - **-(id + id)**

Solution steps:

1. E => - E
2. => - (E)
3. => - (E + **E)**
4. => - (id + **E)**
5. => - (id + **id)**

- Here, **E** is the **start symbol**
- **- (id + id)** is the sentence (contains only **terminals**)
- Strings appearing in the derivation such as,
  E , - E , -(E), - (E + E), - (id + E), - (id + id) are called sentential forms

Types of Derivation

- It is of two types
  - **Left most derivation** (left most non terminal is replaced)
  - **Right most derivation** (right most non terminal is replaced)

Left most Reduction

E => - E
=> - (E)
=> - (E + E)
=> - (id + E)
=> - (id + id)

Right most Reduction

E => - E
=> - (E)
=> - (E + E)
=> - (E + id)
=> - (id + id)

Hence - (id + id) is the valid sentence of the grammar

Strings appeared in leftmost derivation – **left sentential forms**
Strings appeared in rightmost derivation – **right sentential forms**

Reduction (Sentence ω to start symbol S)

- If there is a production, $A \rightarrow \gamma$, then replacement of γ by A eventually leads to **reduction**
  - (**replacing right hand side by left hand side)**

  - Reduction in one step $\rightarrow$
  - Reduction in zero or more steps $\rightarrow^*$
  - Reduction in one or more steps $\rightarrow^+$

Types of Reduction

- To recognize whether the string is valid, the following steps are followed:

Left most Reduction

E => - (id + id)
=> - (E + id)
=> - (E + E)
=> - (E)
=> - E
=> E

Right most Reduction

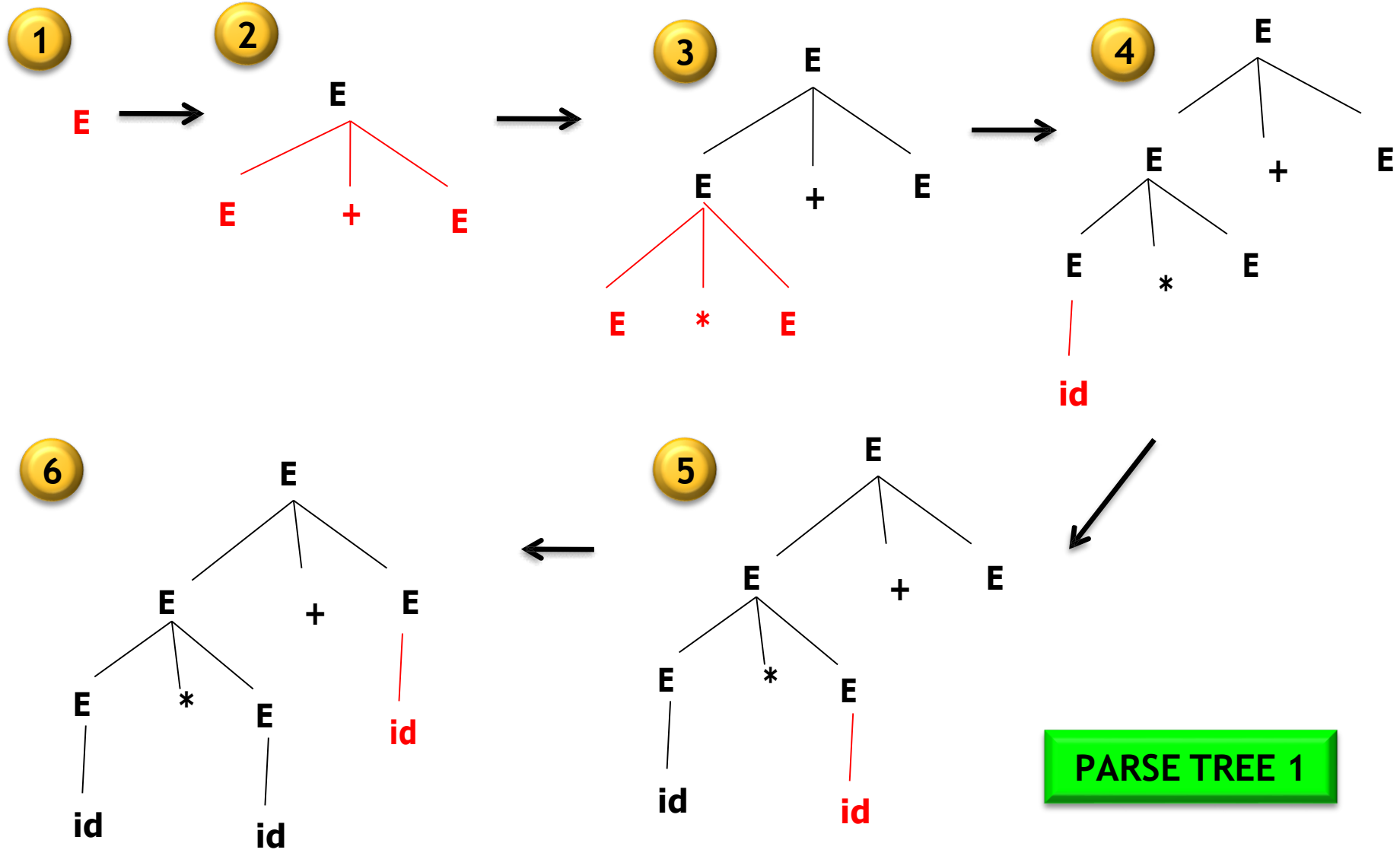
E => - (id + id)
=> - (id + E)
=> - (E + E)
=> - (E)
=> - E
=> E

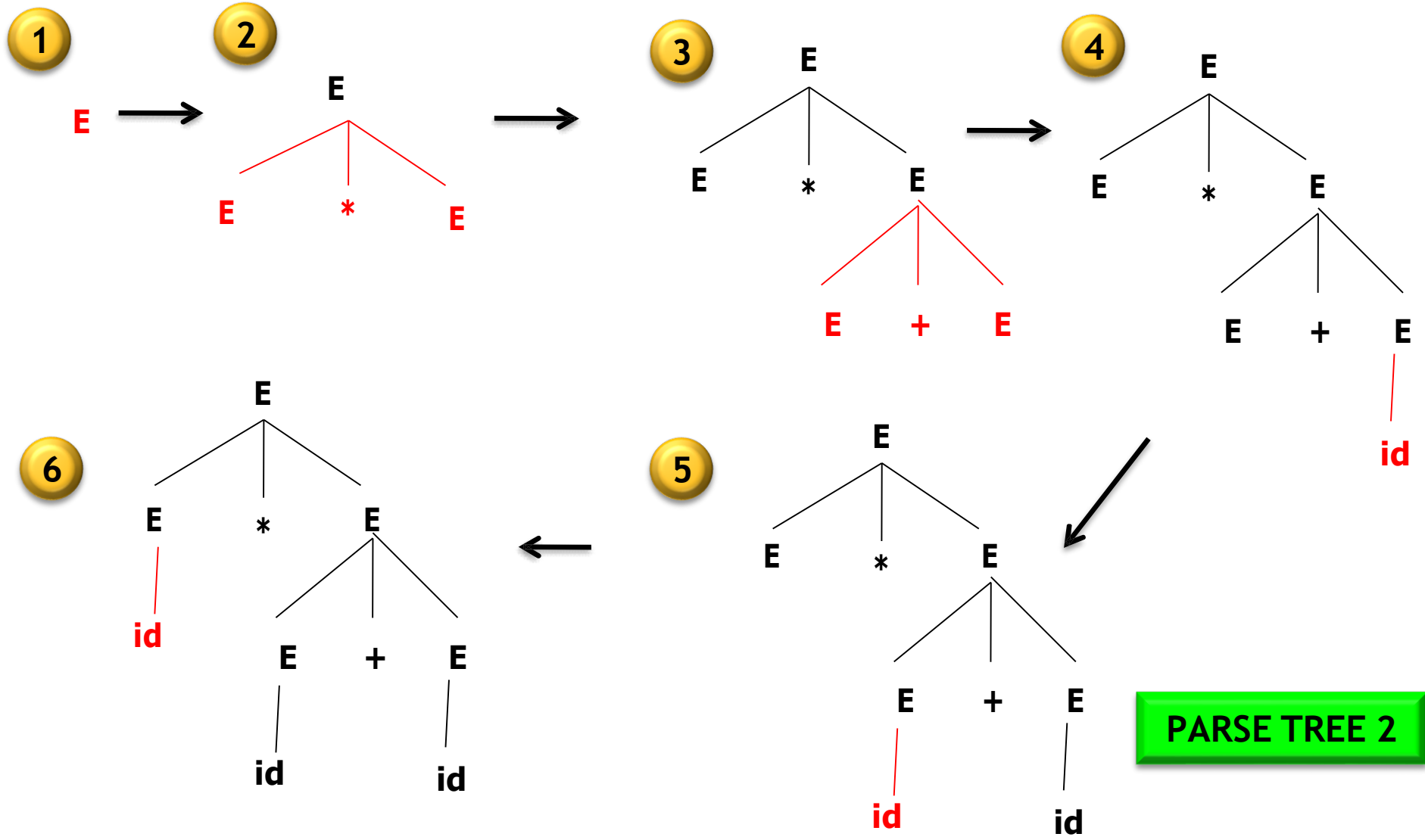
Hence - (id + id) is the valid sentence of the grammar

Parse Tree

- Graphical representation of derivation or reduction is called as **parse tree**.
  - It shows the hierarchical tree structure of sentences implied by Grammar.
    - **Interior node** – Non terminal
    - **Children** – Terminal or non terminal
- In-order traversal of the leaves is the original input.
- The sentential form of parse tree are called **yield or frontier** of the tree

- Grammar $E \rightarrow E+E \mid E*E \mid (E) \mid id$
- String $id * id + id$




1
2
E
E
E
*
E
3
E
E
*
E
E
+
E
4
E
E
*
E
E
+
E
id
5
E
E
*
E
E
+
E
id
id
6
E
E
*
E
id
E
+
E
id
id
PARSE TREE 2

Context-Free Grammar Vs Regular Expressions

- Grammars are more powerful notations than regular expressions.
  - Regular Expression → Grammar **(Possible)**
  - Grammar → Regular Expression **(Not Possible)**

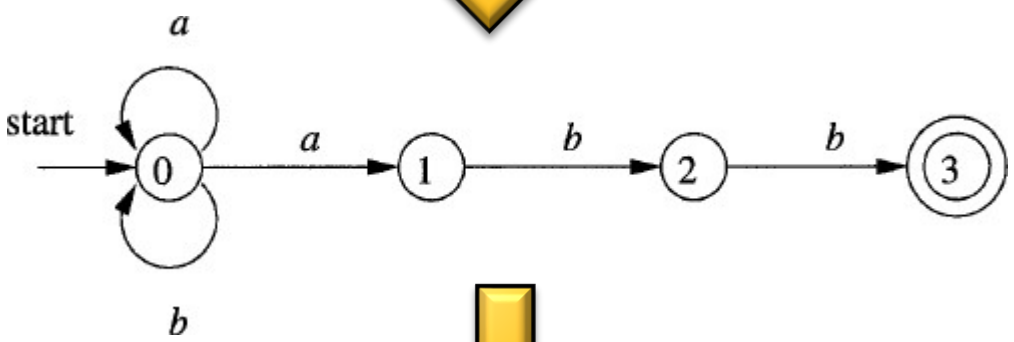


- Regular expression -> NFA -> Grammar

1. For each state i of the NFA, create a nonterminal A_i.
2. If state i has a transition to state j on input a, add the production $A_i \rightarrow aA_j$. If state i goes to state j on input ϵ, add the production $A_i \rightarrow A_j$.
3. If i is an accepting state, add $A_i \rightarrow \epsilon$.
4. If i is the start state, make A_i be the start symbol of the grammar.

Example Problem

(a|b)*abb



$$
\begin{aligned}
A_0 &\rightarrow aA_0 \mid bA_0 \mid aA_1 \\
A_1 &\rightarrow bA_2 \\
A_2 &\rightarrow bA_3 \\
A_3 &\rightarrow \epsilon
\end{aligned}
$$

Question???

If grammars are much powerful than regular expressions, why not using them in lexical analysis too?

- ✓ Lexical rules are quite simple and do not need notation as powerful as grammars
- ✓ Regular expressions are more concise and easier to understand for tokens
- ✓ More efficient lexical analyzers can be generated from regular expressions than from grammars

Problems with grammar

1. Ambiguity
2. Left Recursion
3. Left Factoring

These can create problems for the parser in the phase of syntax analysis

1. Ambiguity

Produces more than one leftmost or rightmost derivations for the same sentence

- A grammar that produces more than one parse tree for some sentence is said to be **ambiguous grammar.**
- For example, two parse trees exist for **id*id+id**
  - **Hence only Parse tree 2 is accepted**

How to eliminate ambiguity?

1. Precedence and associativity rules
   - In the above example, precedence between * and +
   - By this way we can eliminate ambiguity
2. Use ***disambiguating rules*** that *throw away* undesirable parse trees – thus giving only one parse tree finally

Dangling else grammar to eliminate Ambiguity

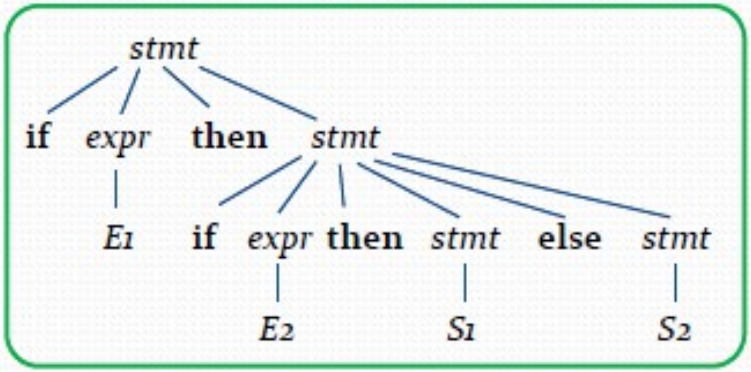


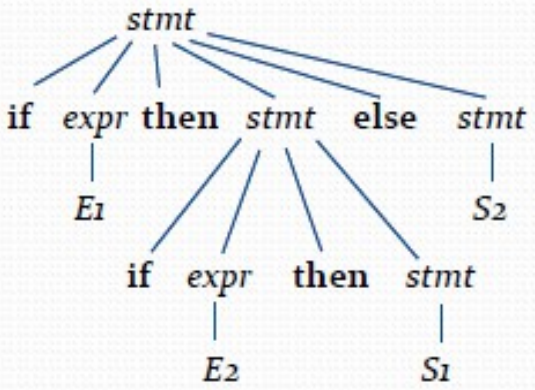
stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other**

Eliminating ambiguity:

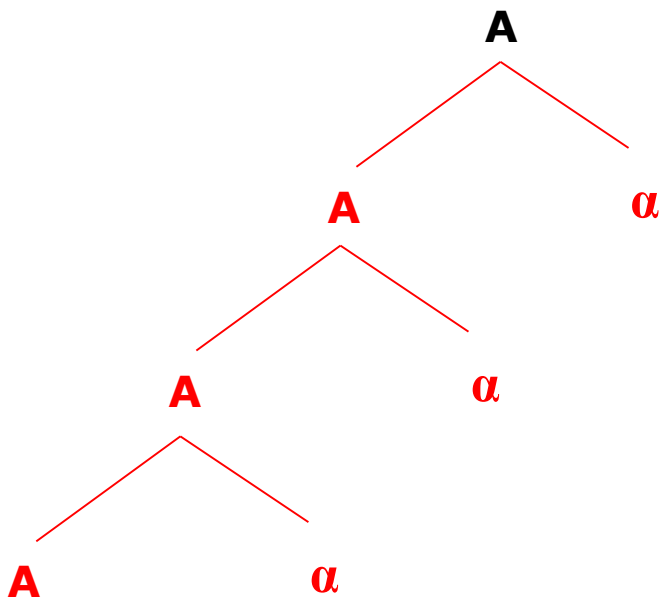
stmt → *matched_stmt*
| *unmatched_stmt*
matched_stmt →
if *expr* **then** *matched_stmt* **else** *matched_stmt*
| **other**
unmatched_stmt →
if *expr* **then** *stmt*
| **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

"**if** E_1 **then if** E_2 **then** S_2 **else** S_3"





2. Left Recursion



- If there is a production of the form **A → A α** for some string α (Leftmost non terminal is same as that of left hand side), it may cause endless recursion.

- Such grammars are called **left-recursive** and they must be transformed if we want to use a top-down parser.

Elimination of Left Recursion

1

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



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

2

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$



$$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

Example

Given Grammar

- E-> E+T | T
- T-> T*F | F
- F->(E) | id



Grammar after Eliminating Left Recursion
E->TE'
E'->+TE'|ε
T->FT'
T'->*FT'|ε
F->(E) | id

Solution

E-> E+T | T
Its of the form, A->Aα|β
Here,
A = E ; α = +T ; β = T
Replace with
A-> βA'
A'-> αA'|ε

E -> TE'
E' -> +TE'|ε

T-> T*F | F
Its of the form, A->Aα|β
Here,
A = T ; α = *F ; β = F
Replace with
A-> βA'
A'-> αA'|ε

T -> FT'
T' -> *FT'|ε

F->(E) | id

Indirect Left recursion Elimination

- How about this:

$S \rightarrow E$
$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow E-T$
$T \rightarrow \mathbf{id}$

There is direct recursion: $E \rightarrow E+T$
There is indirect recursion: $T \rightarrow E+T$, $E \rightarrow T$

Algorithm for eliminating indirect recursion
List the nonterminals in some order $A_1, A_2, \ldots, A_n$
for i=1 to n
  for j=1 to i-1
    if there is a production $A_i \rightarrow A_j\alpha$,
      replace A_j with its RHS
  eliminate any direct left recursion on A_i

Example 2

- S-> A a | b
- A-> A c | S d | ε

Solution

- S -> Aa -> Sda
- ◦ not immediate left recursive

- **After 1st step (substitute S by its RHS)**
  - **S ->A a | b**
  - **A -> A c | A a d | b d | ε**

- **After 2nd step (removal of left recursion)**
  - **S ->A a | b**
  - **A -> b d A'**
  - **A' -> c A' | a d A' | ε**

Left Factoring

- It is a grammar transformation useful for producing a grammar suitable for predictive, top-down parsing

- If there is a production of the form

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

then, grammar after left factoring is

  - $A \rightarrow \alpha A'$
  - $A' \rightarrow \beta_1 \mid \beta_2$

Example

Take the Dangling-else Problem

- S -> i E t S | i E t S e S | a
- E -> b

Solution

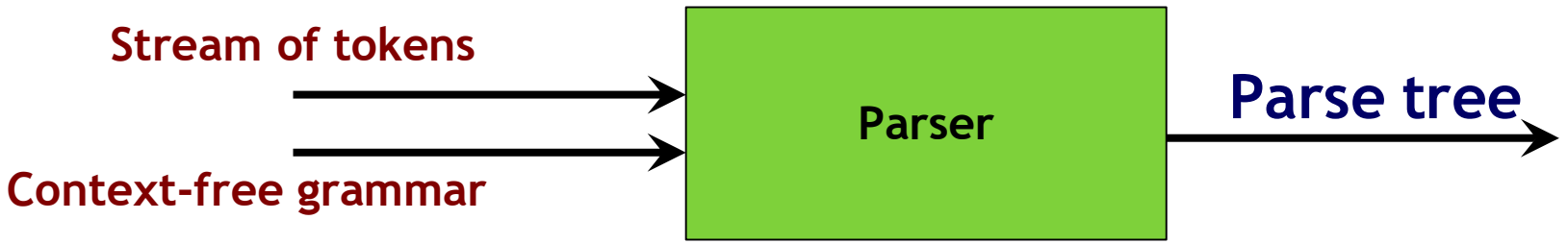
i = if ; t = then ; e = else ; E = Expression ;
S = Statement ; α = i E t S ; β_1 = ε ; β_2 = e S

Grammar after eliminating left factoring is

- S -> i E t S S' | a
- S' -> e S | ε
- E -> b

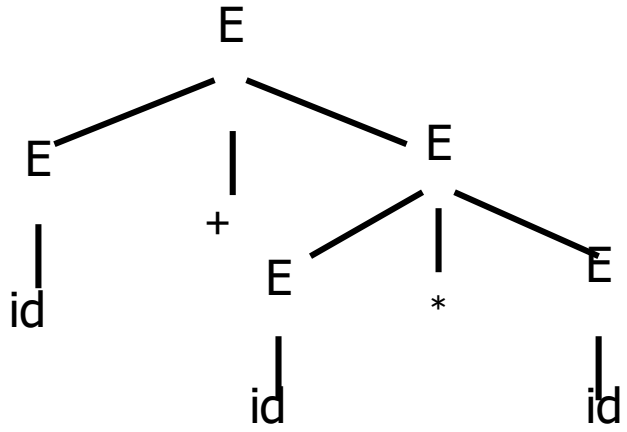
Parsing

- Parsing is a process that constructs a syntactic structure (i.e. parse tree) from the stream of tokens.
  - We already learned how to describe the syntactic structure of a language using (context-free) grammar.
  - So, a parser only needs to do this?



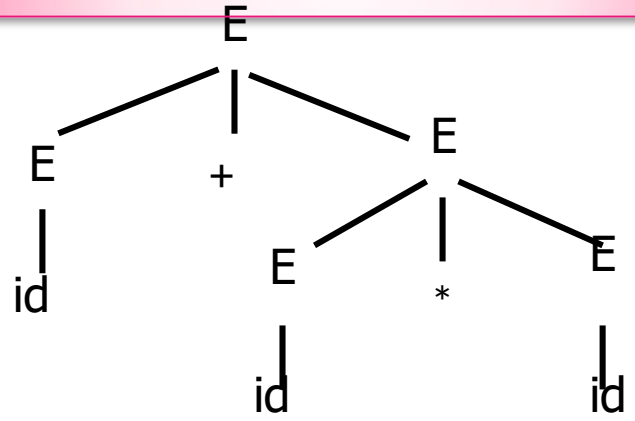
Top-Down Parsing	Bottom-Up Parsing
• A parse tree is created from root to leaves • The traversal of parse trees is a preorder traversal • Tracing leftmost derivation • Two types: ❑ Backtracking parser ❑ Predictive parser	• A parse tree is created from leaves to root • The traversal of parse trees is a reversal of postorder traversal • Tracing rightmost derivation • More powerful than top-down parsing • Three types: ❑ Shift reduce parser ❑ Operator Precedence parser ❑ LR Parser SLR – simple LR parsing LALR – Lookahead LR parsing CLR – canonical LR parsing

Parse Trees and Derivations



Top-down parsing works good for LL Grammars

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow id + E \\
&\Rightarrow id + E * E \\
&\Rightarrow id + id * E \\
&\Rightarrow id + id * id
\end{aligned}
$$



Bottom-up parsing works good for LR Grammars

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow E + E * E \\
&\Rightarrow E + E * id \\
&\Rightarrow E + id * id \\
&\Rightarrow id + id * id
\end{aligned}
$$

LL and LR

- Two important classes of parsers are called LL(k) parsers and LR(k) parsers.
- **The name LL(k) means:**
  - L - Left-to-right scanning of the input
  - L - Constructing leftmost derivation
  - k – max number of input symbols needed to select a parser action
- **The name LR(k) means:**
  - L - Left-to-right scanning of the input
  - R - Constructing rightmost derivation in reverse
  - k – max number of input symbols needed to select a parser action
- So, a LL(1) parser never needs to **"look ahead"** more than one input token

Errors in different Phases

- Lexical analysis → when they do not form token
- Syntactical analysis → syntax errors
- Semantic analysis → type mismatch

- Necessity of Error Handler in a Parser
  - It should report the presence of errors clearly and accurately
  - It should recover from each error quickly
  - It should not significantly slow down the processing of the programs

Error Recovery strategies

- Parser uses the following strategies to recover from syntactic errors
- **Panic mode**
  - Discard input until a token in a set of designated synchronizing tokens is found
- **Phrase-level recovery**
  - Perform local correction on the input to repair the error
    - Replace comma by semicolon
    - Delete a extra semicolon
    - Insert a missing semicolon
- **Error productions**
  - Augment(extend) grammar with productions for erroneous constructs
- **Global correction**
  - Choose a minimal sequence of changes to obtain a global least-cost correction

Top-Down Parsing

- The parse tree is created top to bottom.

- Two types
  - **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
    - Backtracking parsers operate in ***exponential time***
  - **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.
    - Predictive parsers operate in ***linear time***

Recursive-Descent Parsing (uses Backtracking)

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

Given Grammar Inp $\begin{array}{lcl} S & \rightarrow & c\ A\ d \\ A & \rightarrow & a\ b \mid a \end{array}$

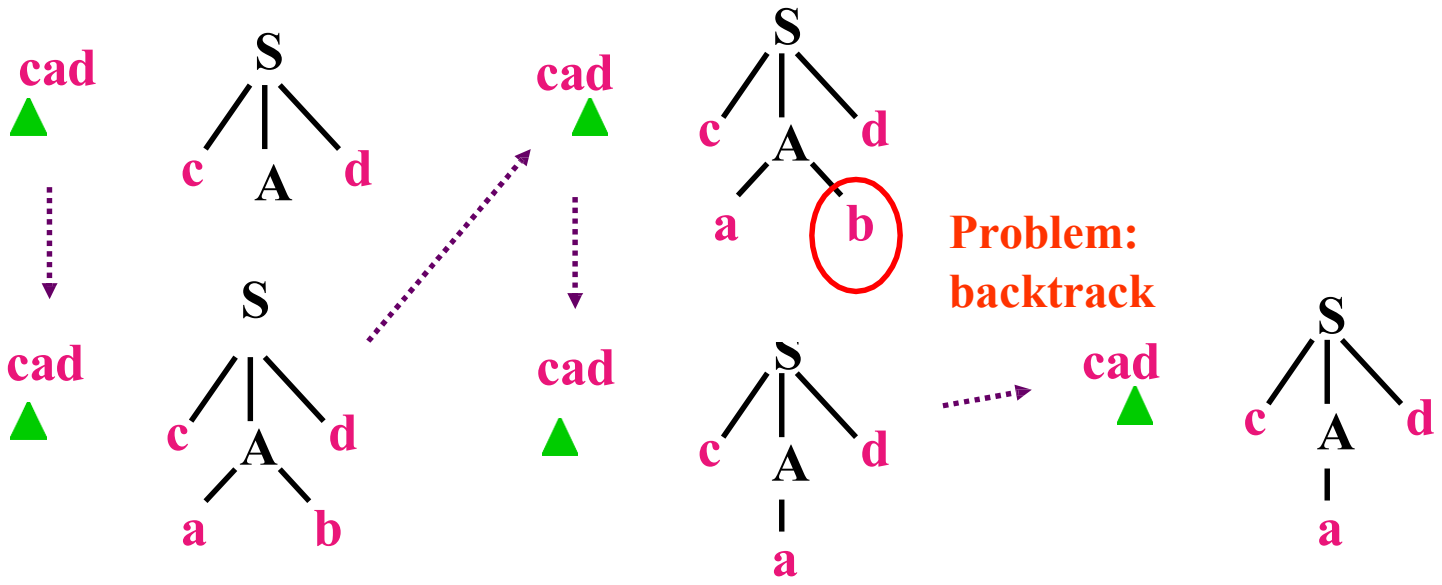
$$w = cad$$

Contd…

- Example:

$$S \rightarrow c\,A\,d$$
$$A \rightarrow ab \mid a$$

Input String: cad



Steps in top-down parsing

Algorithm for Recursive-Descent Parsing

- Al

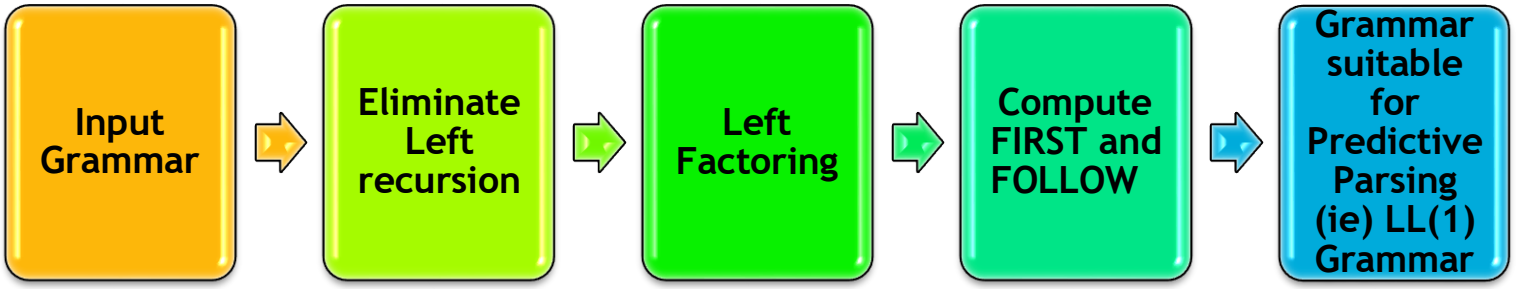
```
void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

- A recursive-descent parsing program consists of a set of procedures, one for each non terminal.
- Execution begins with the procedure for the start symbol,
- Recursive-descent may require backtracking; that is, it may require repeated scans over the input (backtracking is not very efficient)

Predictive parsing (No Backtracking)

Steps

1. Eliminate left recursion from grammar
2. Left factor the grammar
3. Compute *FIRST* and *FOLLOW*
4. Two variants:
   - Recursive (recursive calls)
   - Non-recursive (table-driven)



LL(1) Parser

INPUT BUFFER

- The Input string to be parsed.
- End of the String is marked with a special symbol $.

OUTPUT

- 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.

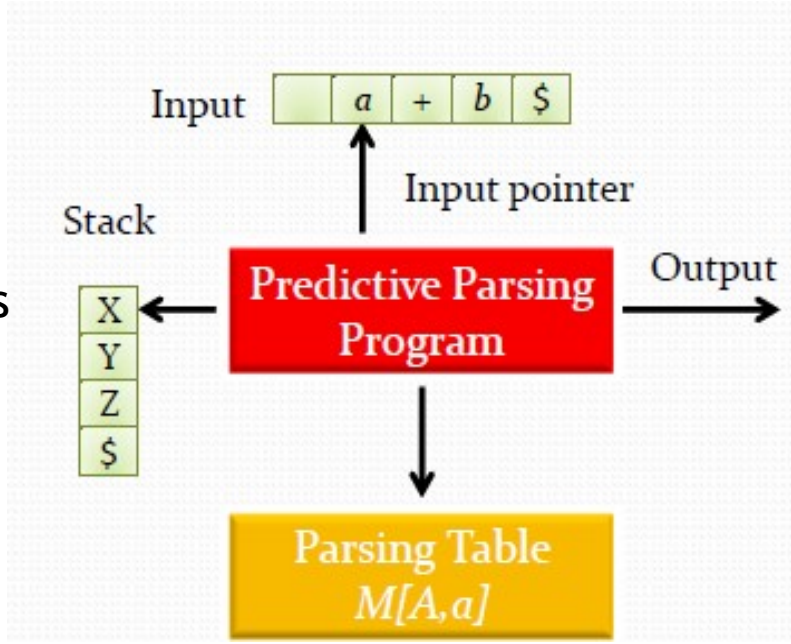
- **4 possible parser actions**.
  1. If **X = a = $,** the parser halts (successful completion)
  2. If **X = a ≠ $,** the parser pops X from the stack, and moves the next symbol in the input buffer.

  3. If **X is a non-terminal** , the parser looks at the parsing table entry M[X,a].
     i. 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.
     ii. If M[X,a] = Error, then parser calls error recovery routine.

Non-Recursive Predictive Parsing

ALGORITHM

set ip to point to the first symbol of w$
repeat
Let X be the top stack symbol and 'a' is the s
by ip;
if *X is a terminal or $*
if *X=a then*
pop *X from the stack and advance ip*
else *error()*
else /* *X is nonterminal* */
if $M[X,a] = X \rightarrow Y_1\ Y_2 \ldots\ Y_k$ **then**
pop **X from the stack**
push $Y_k, \ldots, Y_2, Y_1$ *onto the stack, with* Y_1 *on top*
Output the production $X \rightarrow Y_1\ Y_2 \ldots\ Y_k$
else *error()*
until *X* = *$* /* *the stack is empty* *?



Rules to compute FIRST(X)

Rules

(i) If **x = terminal**, then FIRST(x) = {x} itself

(ii) If **X = non terminal**, then FIRST(X) is a set derived from X.

(iii) If there is a production **x → ε**, then FIRST(x) = {ε}

(iv) If there is a production **X →Y**, (X,Y are non terminals), then

FIRST(X) = FIRST(Y)

Rules to compute Follow(X)

Rules

(i) Place $ in Follow(S), where S is the start symbol and $ is the input right end marker.

(ii) If there is a production $S \rightarrow \alpha B \beta$, then everything in First(β) except ε is placed in FOLLOW(B)

(iii) If there is a production $A \rightarrow \alpha B$, or the production $A \rightarrow \alpha B \beta$, where First($\beta$) contains ε, then everything in Follow(A) is in

Follow(B)

Construction of Predictive parsing table

- Add **A->a** to M[A, t] for each t in First(a).
- If First(a) contains empty string
  - Add A->a to M[A, t] for each t in Follow(A)
  - if $ is in Follow(A), add A->a to M[A, $]
- Make each undefined entry of M error.

Example Problem

- Construct predictive parsing table for the following grammar and check whether the strings **(id+id)** and **id+** are accepted or not.

Given Grammar:

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$

Solution

Step 1:

- Elimination of Left recursion

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Step 2:

- There is no left factoring now

Contd…

Step 3:

Computation of FIRST

- First(E) = First(T)

  [by rule 4, with production $E \rightarrow TE'$ First(T) is in First(E)]

- First(T) = First(F)

  [by rule 4, with production $T \rightarrow FT'$ First(F) is in First(T)]

- First(F) = {(, id}

  [by rule 1, with production $F \rightarrow (E) \mid id$
  where {(,id) are terminals]

Hence, **FIRST(E) = FIRST(T) = FIRST(F) = {(, id}**

Step 3 – Contd…

- FIRST(E') = {+, ε}

  [by rule 1,2,3, with production E' → +TE' | ε]

- FIRST(T') = {*, ε}

  [by rule 1,2,3, with production T' → *FT' | ε]

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(ε) = {ε}
FIRST((E)) = {(}
FIRST(id) = {id}

Contd…

Step 4:

Computation of FOLLOW

- Follow(E) = {), $}

  [by rule 1, $ is added because E → start symbol
  By rule 2, with production F → (E), the parenthesis) is added]

- Follow(E') = {), $}

  [By rule 3, with production E → TE', everything in Follow(E) is in Follow(E')]

- Follow(T) = {+,), $}

  [By rule 2, with production E → TE', everything in First(E') except ε is included. Hence, Follow(T) ={+}
  By rule 3, with production E → TE', First(β) = First(E') ➔contains ε hence, everything in Follow(E) is in Follow(T)]

E → T E'
α B β

Step 4 - Contd…

$T \rightarrow F\ T'$

$\alpha\ B\ \beta$

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

  [By rule 3, with production $T \rightarrow FT'$, First(β) = First(T') ➔contains ε hence, everything in Follow(T) is in Follow(T')]

- Follow(F= {+,*,), $}

  [By rule 2, with production $T \rightarrow FT'$, everything in First(T') except ε is included. Hence, Follow(F) ={*}
  By rule 3, with production $T \rightarrow FT'$, First(β) = First(E') ➔contains ε hence, everything in Follow(T) is in Follow(F)]

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

Generated Parser Table For String id + id * id

Non Terminal	INPUT SYMBOLS					
	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)		

Parsing table

Parse the input **id*id** using the parse table and a stack

Step	Stack	Input	Next Action
1	$E	**id*id$**	E→TE'
2	$E'T	**id*id$**	T→FT'
3	$E'T'F	**id*id$**	F→**id**
4	$E'T'**id**	**id*id$**	match **id**
5	$E'T'	***id$**	T'→*FT'
6	$T'F*	***id$**	match *
7	$T'F	**id$**	F→**id**
8	$T'**id**	**id$**	match **id**
9	$T'	**$**	T'→ε
10	$	**$**	accept

BOTTOM UP PARSING

- Builds parse tree from leaves to root
- Constructs rightmost derivation
  - LR(k) grammars

- **Three types:**
  - **Shift reduce parser**
  - **Operator Precedence parser**
  - **LR Parser**
    - **SLR** - simple LR parsing
    - **LALR** - Lookahead LR parsing
    - **CLR** - canonical LR parsing

Shift-Reduce Parsing

- Shift reduce parsing is a type of bottom up parsing that constructs parse tree from the leaves to the root.

- A shift-reduce parser tries to reduce the given input string into the starting symbol.

a string → reduced to the starting symbol

Shift-reduce Parsing: Example

Grammar: $S \rightarrow \mathbf{a}A\mathbf{B}\mathbf{e}$

$A \rightarrow A\mathbf{bc} \mid \mathbf{b}$

$B \rightarrow \mathbf{d}$

Input:	**abbcde**	(using $A \rightarrow \mathbf{b}$)
$\Rightarrow$	**a***A***bcde**	(using $A \rightarrow A\mathbf{bc}$)
$\Rightarrow$	**a***A***de**	(using $B \rightarrow \mathbf{d}$)
$\Rightarrow$	**a***AB***e**	(using $S \rightarrow \mathbf{a}AB\mathbf{e}$)
$\Rightarrow$	S	

Stack implementation of shift Reduce parsing

- Stack is used to hold Grammar.
- Initial stack just contains only the end-marker $.
- The end of the input string is marked by the end-marker **$**.

Step 1:

- Initial configuration for shift reduce parser is

STACK	INPUT
$	w$

Step 2:

The parser shifts zero or more input symbols until a handle β is on top of the stack

Step 3:

The parser reduces β to the left side of the production

Step 4:

The parser repeat the steps 2 & 3, until we are getting the following configuration

STACK	INPUT
$S	$

Actions in the Shift-Reduce Parser

- There are four possible actions of a shift-parser action:

1. **Shift :** The next input symbol is shifted onto the top of the stack.
2. **Reduce:** Replace the handle on the top of the stack by the non-terminal.
3. **Accept:** Successful completion of parsing.
4. **Error:** Parser discovers a syntax error, and calls an error recovery routine.

Shift-Reduce Parsing: An Example

E	$\longrightarrow$	E+T
E	$\longrightarrow$	T
T	$\longrightarrow$	id

Stack	Input Stream	Action
\$	id + id \$	**shift**
\$ id	+ id \$	**reduce by** $T \longrightarrow$ id
\$ T	+ id \$	reduce by $E \longrightarrow T$
\$ E	+ id \$	shift
\$ E +	id \$	shift
\$ E + id	\$	reduce by $T \longrightarrow$ id
\$ E + T	\$	reduce by $E \longrightarrow E$+T
\$ E	\$	**ACCEPT**

Shift-Reduce Parsing

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

Stack	Input	Action
\$	$\mathbf{id}_1$ * $\mathbf{id}_2$ \$	shift
\$$\mathbf{id}_1$	* $\mathbf{id}_2$ \$	reduce by $F \rightarrow \mathbf{id}$
\$F	* $\mathbf{id}_2$ \$	reduce by $T \rightarrow F$
\$T	* $\mathbf{id}_2$ \$	shift
\$T *	$\mathbf{id}_2$ \$	shift
\$T * $\mathbf{id}_2$	\$	reduce by $F \rightarrow \mathbf{id}$
\$T * F	\$	reduce by $T \rightarrow T * F$
\$T	\$	reduce by $E \rightarrow T$
\$E	\$	accept

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Notation for a Shift-Reduce Execution

Time ↓

STACK	INPUT	ACTION
$	(id+id) *id$	
$ (	id+id) *id$	Shift
$ (id	+id) *id$	Shift
$ (F	+id) *id$	Reduce by F → id
$ (T	+id) *id$	Reduce by T → F
$ (E	+id) *id$	Reduce by E → T
$ (E+	) *id$	Shift
$ (E+id	) *id$	Shift
$ (E+F	) *id$	Reduce by F → id
$ (E+T	) *id$	Reduce by T → F
$ (E	) *id$	Reduce by E → E + T
$ (E)	*id$	Shift
$F	*id$	Reduce by F → (E)
$T	*id$	Reduce by T → F
$T*	id$	Shift
$T*id	$	Shift
$T*F	$	Reduce by F → id
$T	$	Reduce by T → T * F
$E	$	Reduce by E → T
$E	$	Accept

A Stack Implementation of A Shift-Reduce Parser

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce by F → id
$F	+id*id$	reduce by T → F
$T	+id*id$	reduce by E → T
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by F → id
$E+F	*id$	reduce by T → F
$E+T	*id$	shift
$E+T*	id$	shift
$E+T*id	$	reduce by F → id
$E+T*F	$	reduce by T → T*F
$E+T	$	reduce by E → E+T
$E	$	accept

Shift-Reduce Parsing -- Example

$S \rightarrow aABb$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

input string: aaabb

aaAbb

aAbb

aABb

S

⇓ reduction

$S \underset{rm}{\Rightarrow} aABb \underset{rm}{\Rightarrow} aAbb \underset{rm}{\Rightarrow} aaAbb \underset{rm}{\Rightarrow} aaabb$

Right Sentential Forms

Bottom Up Parsing

- "Shift-Reduce" Parsing
- Reduce a string to the start symbol of the grammar.
- At every step a particular sub-string is matched ie. right hand side (RHS) is substituted by the non-terminal in the left hand side (LHS) of the production.

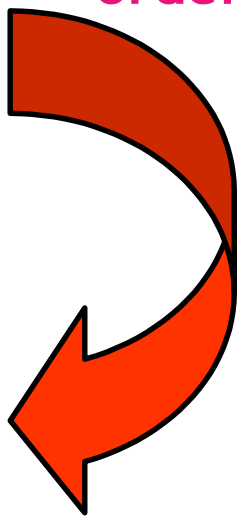
Consider:

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

abbcde
aAbcde
aAde
aABe
S

Reverse order



Rightmost Derivation:

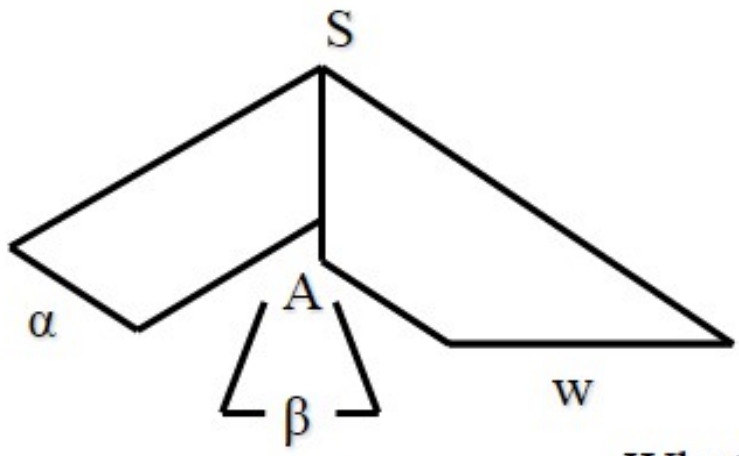
$S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$

Handler

- A **"HANDLE"** of a string is a substring that matches the RHS of a production and whose reduction to the non-terminal (on the LHS of the production) represents one step along the reverse of a rightmost derivation toward reducing to the start symbol.

Formally, if $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha Aw \underset{rm}{\Rightarrow} \alpha\beta w$, as in Fig. 4.27, then production $A \to \beta$ in the position following α is a *handle* of $\alpha\beta w$.

- Consider Right Sentential Form: αβw and Rule A → β



What Does
α Signify?

Input Processed
Still on Parsing Stack

What Does
β Represent?

Candidate Handle to
be Reduced

What Does
w Contain?

Input yet to be
Consumed

Example

- Consider the grammar
- E -> E + E | E * E | id

Right sentential form	Handle	Production Rule
id1 + id2 * id3	id1	E -> id
E + id2 * id3	id2	E -> id
E + E * id3	id3	E -> id
E + E * E	E * E	E -> E * E
E + E	E + E	E -> E +E
E		

Handle Pruning

- A rightmost derivation in reverse can be obtained by "handle pruning".That is,we start with a string of terminals w that we wish to parse.If ω is a sentence of grammar at hand,then ω = γ,where γ_n is the nth right-sentential form of some as yet unknown rightmost derivation

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = \omega$$

Input string

Viable prefixes

- The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called **viable prefixes**.

Viable prefixes

- At any point in time
  - Content of the stack is a prefix of a right-sentencial form
  - This prefix is called a *viable prefix*
- Check again!
  - Below = all the right-sentencial form of a rightmost derivation
  - S $\Rightarrow$ aABe
    $\Rightarrow$ aAde
    $\Rightarrow$ aAbcde
    $\Rightarrow$ abbcde

$
$a
$ab
$aA
$aAb
$aAbc
$aA
$aAd
$aAB
$aABe
$S

Conflicts During Shift-Reduce Parsing

- **Conflicts Type**
  - shift-reduce conflict
  - reduce-reduce conflict

- *Shift-reduce* and *reduce-reduce* conflicts are caused by
  - The limitations of the LR parsing method (even when the grammar is unambiguous)
  - Ambiguity of the grammar

Shift-Reduce conflict

- What action to take in case both shift and reduce actions are valid?

Reduce-Reduce conflict

- Which rule to use for reductions if reduction is possible by more than one rule?

Shift-Reduce Conflict

Consider the Grammar

$E \rightarrow E + E \mid E * E \qquad E \rightarrow \mathbf{id}$

Input String : id+id*id

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce $E \rightarrow$ id
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce $E \rightarrow$ id
$E+E	*id$	reduce $E \rightarrow E + E$
$E+E*		shift
$E+E*id	id$	reduce $E \rightarrow$ id
$E+E*E	$	reduce $E \rightarrow E * E$
$E+E	$	reduce $E \rightarrow E + E$
$E	$	accept
	$	

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce $E \rightarrow$ id
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce $E \rightarrow$ id
$E+E	*id$	shift
$E+E*		shift
$E+E*id	id$	reduce $E \rightarrow$ id
$E+E*E	$	reduce $E \rightarrow E * E$
$E+E	$	reduce $E \rightarrow E + E$
$E	$	accept
	$	

Consider the grammar E → E+E | E*E | id
and input id+id*id

stack	input	action
E+E	*id	reduce by E →E+E
E	*id	shift
E*	id	shift
E*id		reduce by E →id
E*E		reduce byE →E*EE

stack	input	action
E+E	*id	shift
E+E*	id	shift
E+E*id		reduce by E →id
E+E*E		reduce byE →E*E
E+E		reduce byE →E+EE

Reduce reduce conflict

Consider the grammar M → R+R | R+c | R

R → c

and input c+c

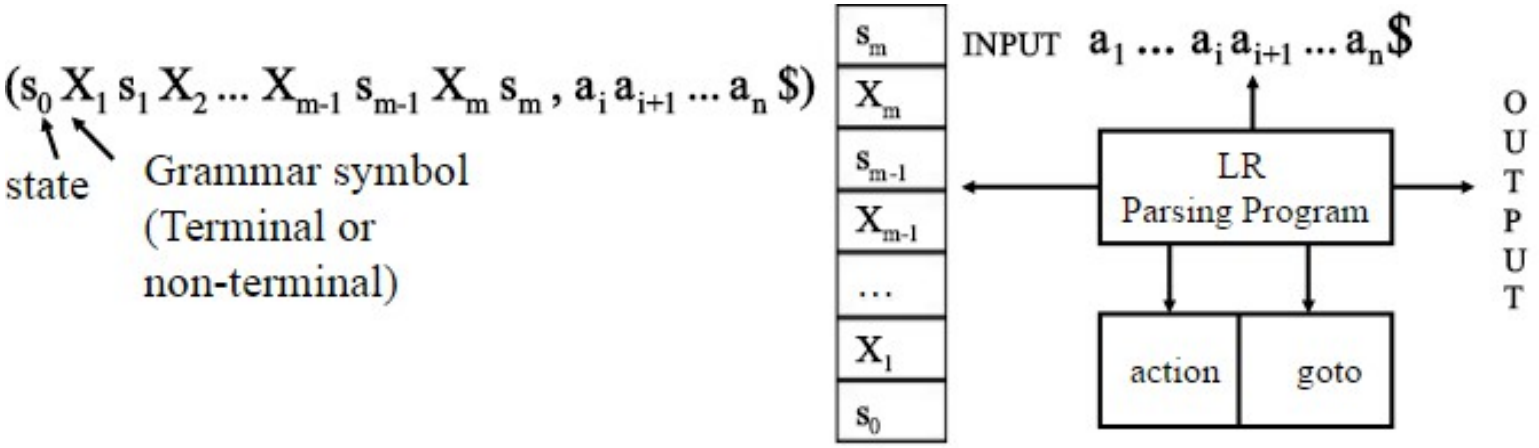
Stack	input	action
	c+c	shift
c	+c	reduce by R→c
R	+c	shift
R+	c	shift
R+c		reduce by R→c
R+R		reduce by →R+R

Stack	input	action
	c+c	**shift**
c	**+c**	**reduce by R→c**
R	**+c**	**shift**
R+	**c**	**shift**
R+c	**reduce by M→R+c**	

Three classes of LR Parsers

- Simple LR (SLR) or LR(0)
  - Easiest but Limited in Grammar Applicability
  - Grammar Leads to S/R and R/R Conflicts
- Canonical LR
  - Powerful but Expensive
  - LR(k) – Usually LR(1)
- Lookahead LR (LALR) – In Between Two
- Two Fold Focus:
  - Parser Table Construction – Item and Item Sets
  - Examination of LR Parsing Algorithm

LR Parser Structure



- action[s_m , a_i] is Parsing Table with Four Options
  1. Shift S onto Stack
  2. Reduce by Rule
  3. Accept ($,$)
  4. Report an Error
- goto[s_m , a_i] determines next state for action
- Question: What does following Represent?

$$X_1 X_2 \ldots X_{m-1} X_m a_i a_{i+1} \ldots a_n$$

What is Parsing Table?

- Combination of State, Action, and Goto
  - Shift s5 means shift input symbol and state 5
  - Reduce r2 means reduce using rule 2
  - goto state/NT indicates the next state

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Actions against Configuration

Configuration: $(s_0 X_1 s_1 X_2 \dots X_{m-1} s_{m-1} X_m s_m , a_i a_{i+1} \dots a_n \$)$

- action$[s_m , a_i] =$
  1. Shift s in Parsing Table – Move $a_i s_{m+1}$ to Stack
     $(s_0 X_1 s_1 X_2 \dots X_{m-1} s_{m-1} X_m s_m a_i s_{m+1} , a_{i+1} \dots a_n \$)$
  2. Reduce $A \rightarrow \beta$ means
     - Remove $2\times|\beta|$ symbols from stack and Push A along with state s = goto$[s_{m-1} , A]$ onto stack
     - Uses Prior State after popping to determine goto
  3. Accept – Parsing Complete
  4. Error – Call recovery Routine

YACC: Yet Another Compiler-Compiler

- Yacc generates **C code for syntax analyzer**.
- Yacc uses **grammar rules** that allow it to analyze tokens from LEX and create a syntax tree.
  - Yacc provides a general tool for describing the input to a computer program.
  - Yacc is written in **portable C**.
- The class of specifications accepted is a very general one: **LALR grammars** with disambiguating rules.

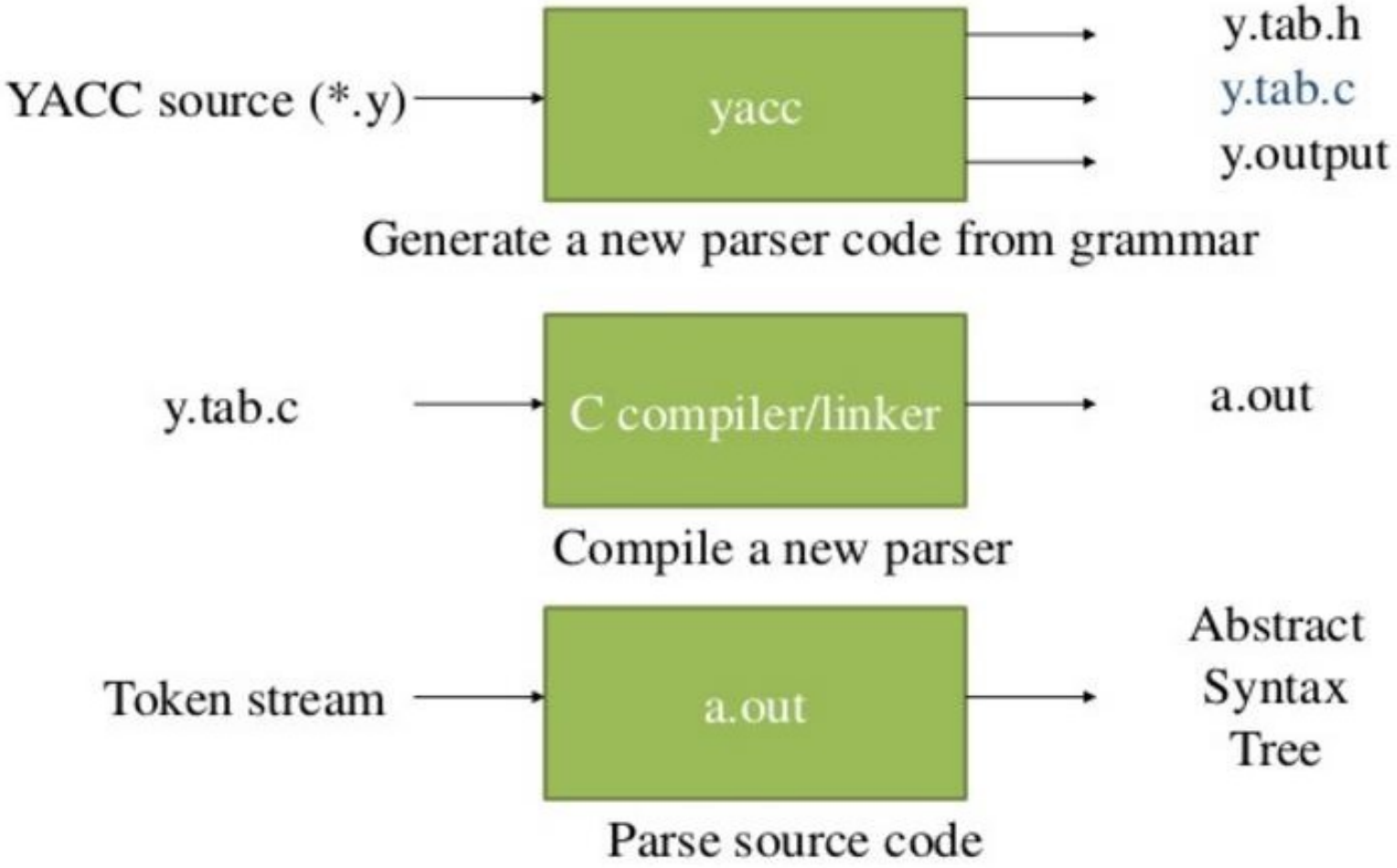
- In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

Yacc contd…….

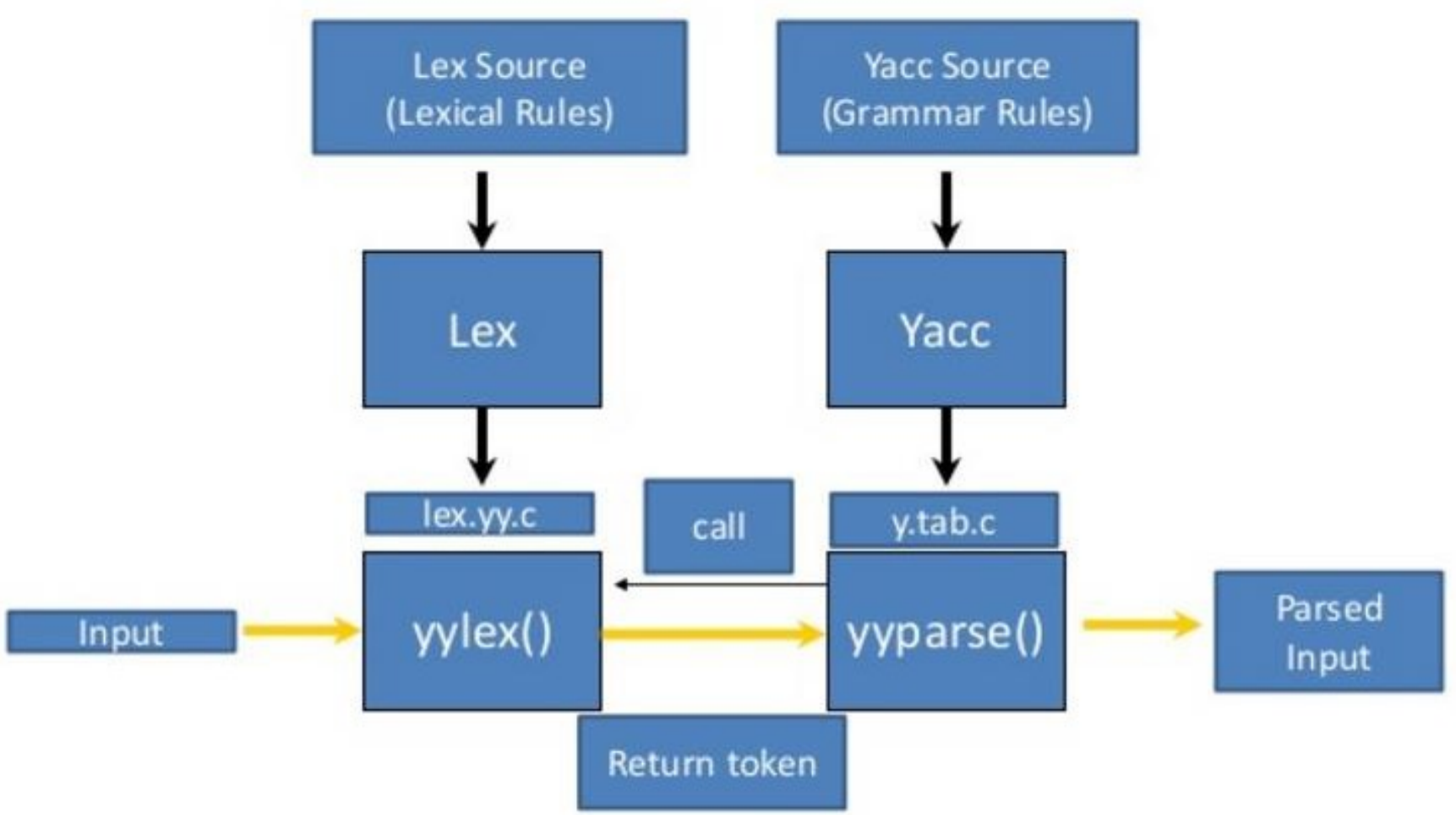
- Yacc is a computer program for the Unix operating system.
- It is a **Look Ahead Left-to-Right (LALR) parser** generator, generating a parser.
- Yacc was originally developed in the early **1970s by Stephen C. Johnson** at AT&T Corporation and written in the B programming language, but soon rewritten in C.

- Yacc has also been rewritten for other languages, including OCaml, Ratfor, ML, Ada, Pascal, Java, Python, Ruby, Go and Com mon Lisp.

How YAAC works?



LEX with YAAC



LALR PARSER GENERATOR-yacc

- YAAC tool → Generate automatically an LALR parser.
- **Input of yacc is divided into three sections:**

  .....definitions.....

  %%

  .....rules.....

  %%

  ....subroutines.....

- **def section** → consists of token declarations & C code bracketed by % { and % }.
- **rules section** → consists of grammar.
- **subroutines section** → consists of subroutines.

Example- small calculator

- The def section for the yacc input file:

  %token INTEGER

  - This def declares an INTEGER token.

    #define INTEGER 258

- When we run yacc, it generates a parser file y.tab.c & also creates an include file y.tab.h

  #

Contd…

- Lex includes this file & utilizes the definitions for token values.
- To obtain tokens, yacc calls yylex.
- Function yylex has a return type of int & returns the token value.
- Attributes associated with the token are returned by lex in variable yylval.

File	Content
calc.lex	Specifies the lex command specification file that defines the lexical analysis rules.
calc.yacc	Specifies the yacc command grammar file that defines the parsing rules, and calls the yylex subroutine created by the lex command to provide input.

Compiling the desk calculator program

1. Process the **yacc grammar file using the -d** optional flag (d → define the tokens):

   yacc -d calc.yacc

2. Use the **ls** command to verify that the following files were created:

- **y.tab.c** → C language source file, that the yacc command created for the parser
- **y.tab.h** → Header file containing define statements for the tokens used by the parser

3. Process the lex specification file:

   lex calc.lex

4. Use the ls command to verify that the following file was created:

   lex.yy.c → C language source file that the lex command created for the lexical analyzer

Compiling the example program

5. Compile and link the two C language source files:

   cc y.tab.c lex.yy.c

6. Use the ls command to verify that the following files were created:

y.tab.o →Object file for the y.tab.c source file

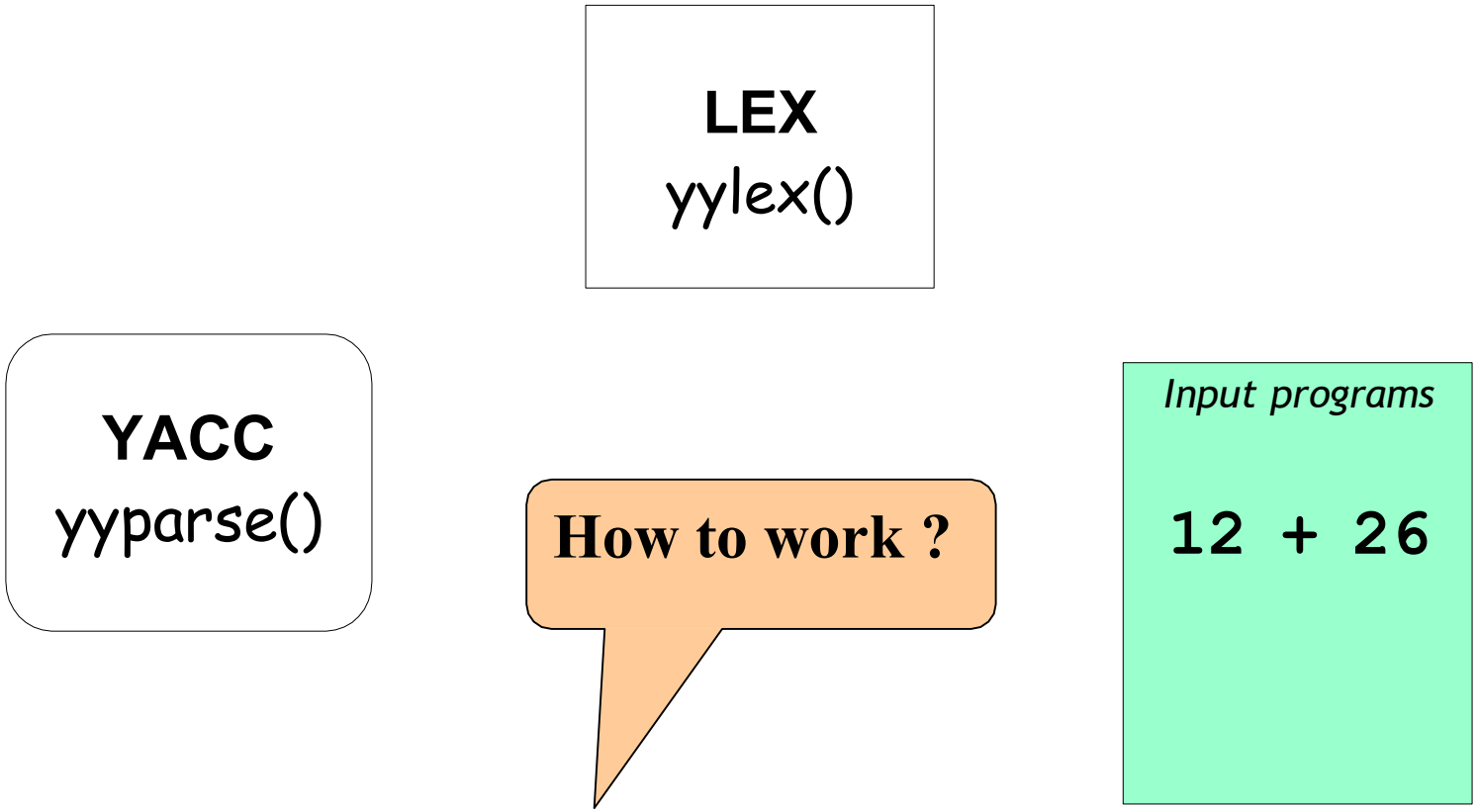
lex.yy.o → Object file for the lex.yy.c source file

a.Out → Executable program file

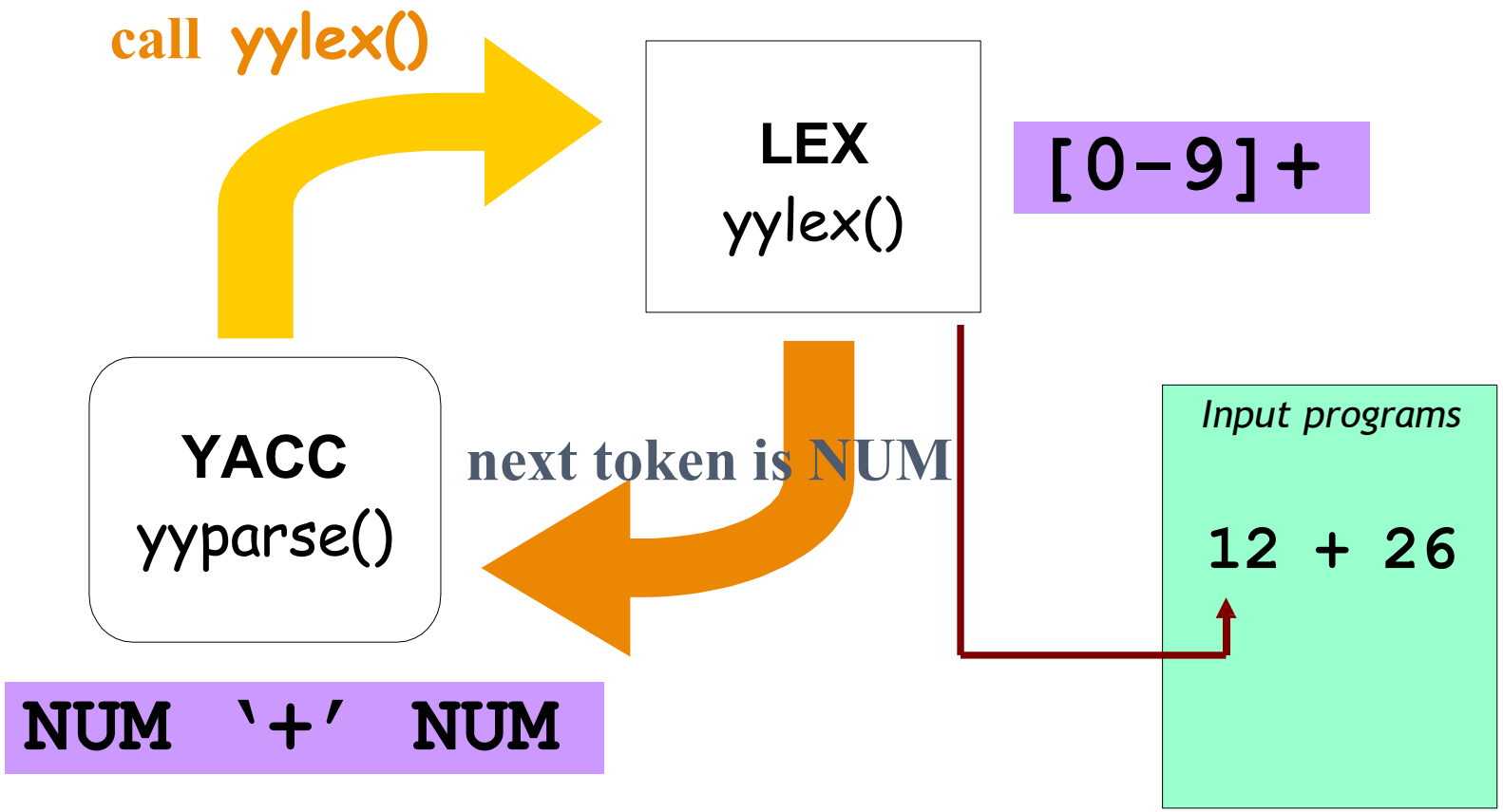
To run the program directly from the a.out file, type:

$ a.out

LEX and YACC: a team



LEX and YACC: a team



Completed