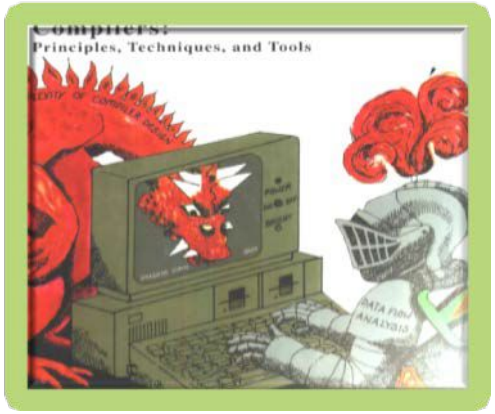
Table of Contents

MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

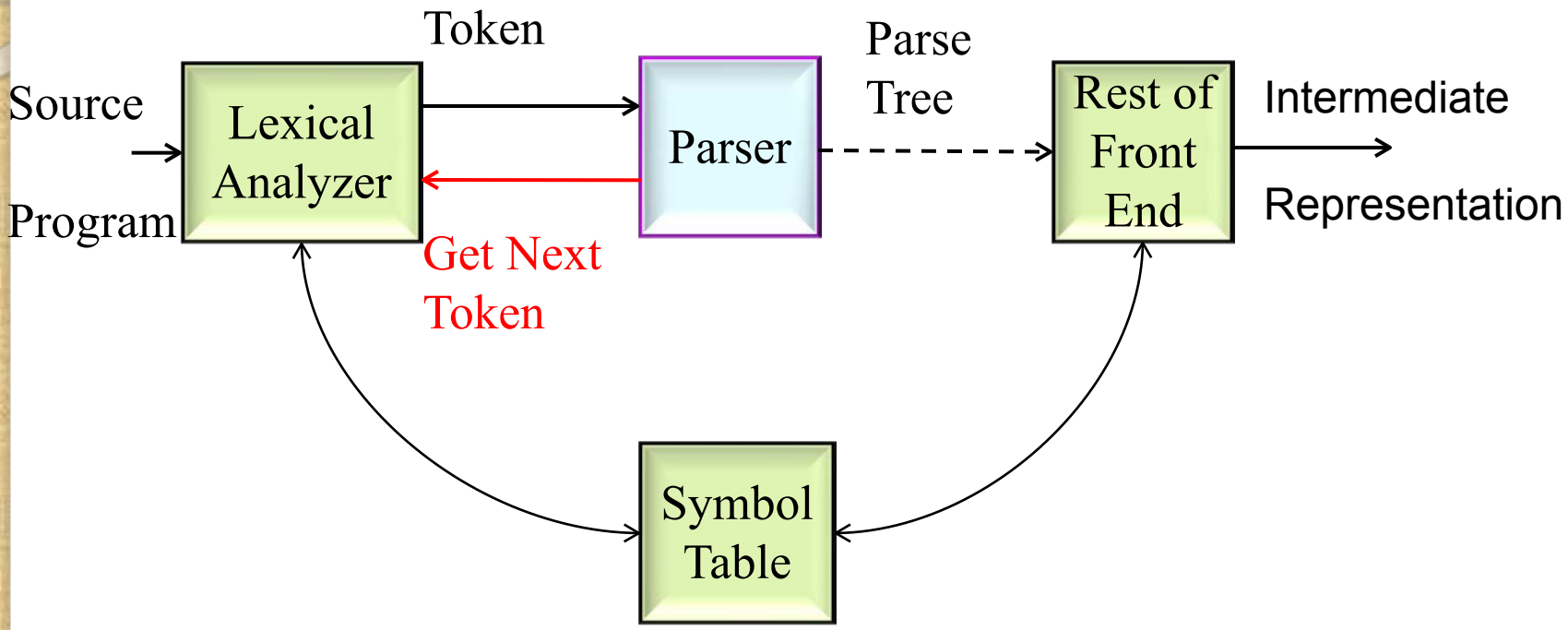
Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

System Software
Compiler Design



School of Computer Engineering and Technology

Role of Parser / Syntax Analysis



Role of Parser / Syntax Analysis

- Checks whether the token stream meets the Grammatical Specification of the Language and generates the Syntax Tree.
- A grammar of a programming language is typically described by a Context Free Grammar, which also defines the structure of the parse tree.
- A syntax error is produced by the compiler when the program does not meet the grammatical specification.

Definition of Context-Free Grammars

A context-free grammar ***G*** = (***T***, ***N***, ***S***, ***P***) consists of:

1. ***T***, a set of *terminals* (scanner tokens).
2. ***N***, a set of *nonterminals* (syntactic variables generated by productions).
3. ***S***, a designated *start* nonterminal.
4. ***P***, a set of *productions*. Each production has the form, $A ::= \alpha$, where A is a nonterminal and α is a *sentential form* , i.e., a string of zero or more grammar symbols (terminals/nonterminals).

Context-Free Grammars

- A context-free grammar defines the syntax of a programming language
- The syntax defines the syntactic categories for language constructs
  - Statements
  - Expressions
  - Declarations
- Categories are subdivided into more detailed categories
  - A Statement is a
    - **For-statement**
    - **If-statement**
    - **Assignment**

<*statement*> ::= <*for–statement*> | <*if–statement*> | <*assignment*>

<*for–statement*> ::= **for (** <*expression*> **;** <*expression*> **;** <*expression*> **)**

<*statement*>

<*assignment*> ::= <*identifier*> **:=** <*expression*>

Syntax Analysis

Syntax Analysis Problem Statement: To find a derivation sequence in a grammar ***G*** for the input token stream (or say that none exists).

Derivation

Given the following grammar:

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

Is the string -(**id** + **id**) a sentence in this grammar?

Yes because there is the following derivation:

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathbf{id} + \mathbf{id})$

Where $\Rightarrow$ reads “derives in one step”.

Parse trees

A parse tree is a graphical representation
of a derivation sequence of a sentential form.

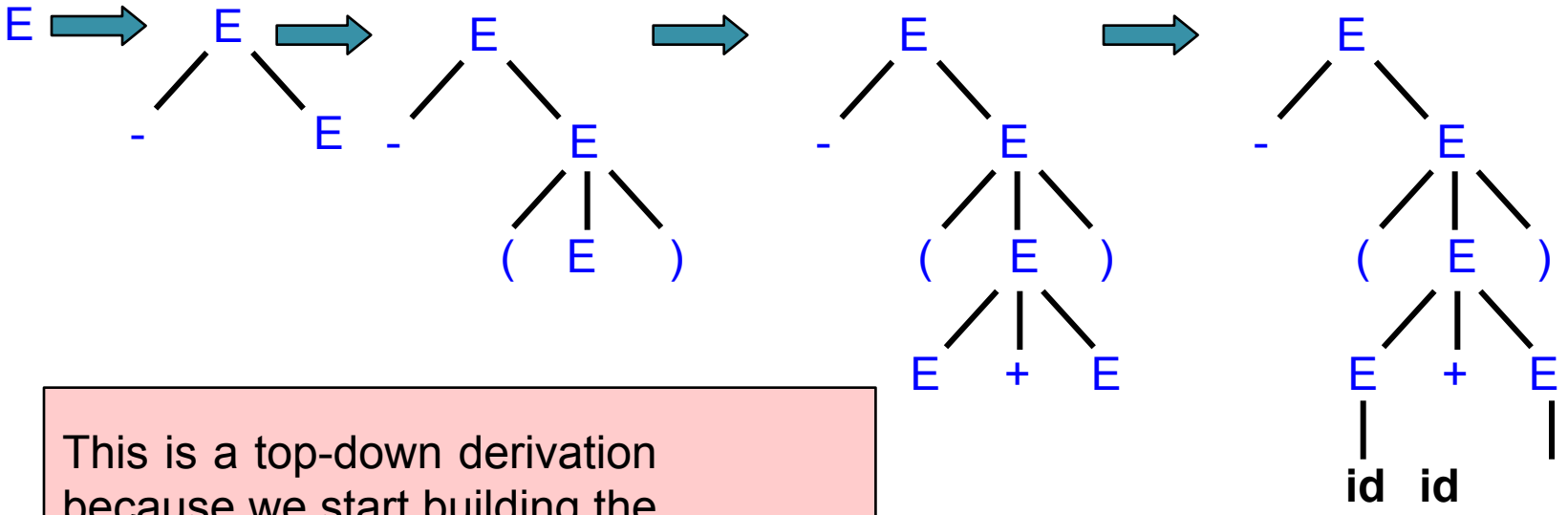
Tree nodes represent symbols of the grammar (nonterminals or terminals) and tree edges represent derivation steps.

Derivation

E → E + E | E * E | (E) | - E | **id**

Lets examine this derivation:

E ⇒ -E ⇒ -(E) ⇒ -(E + E) ⇒ -(**id** + **id**)



This is a top-down derivation because we start building the parse tree at the top

parse tree

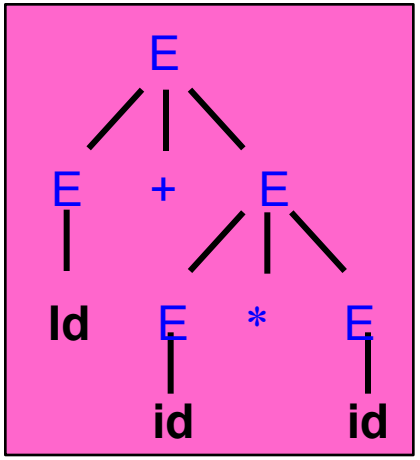
Another Derivation Example

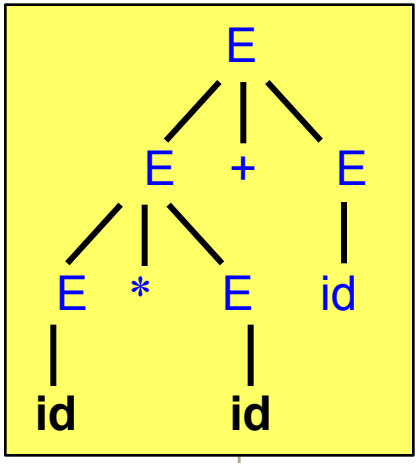
$E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \mathbf{id}$

Find a derivation for the expression:

id + id * id

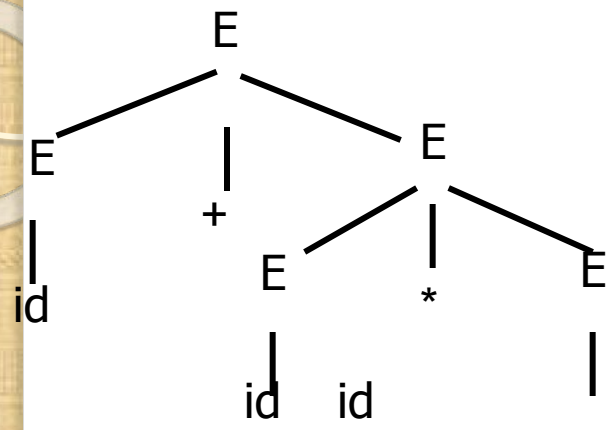
According to the grammar, both are correct.



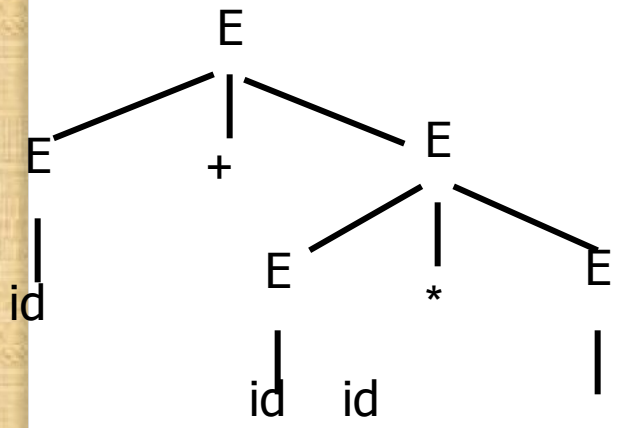


A grammar that produces more than one parse tree for any input sentence is said to be an ambiguous grammar.

Parse Trees and Derivations



Top-down parsing



Bottom-up parsing

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

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

Top–Down 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: Try different structures and backtrack if it does not matched the input

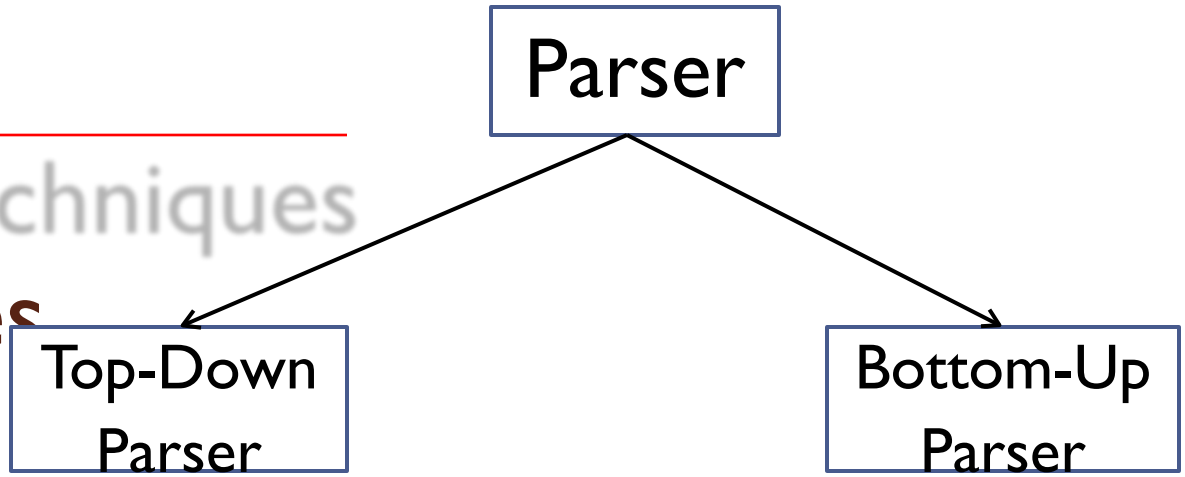
Predictive: Guess the structure of the parse tree from the next input

Bottom–Up Parsing

- 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

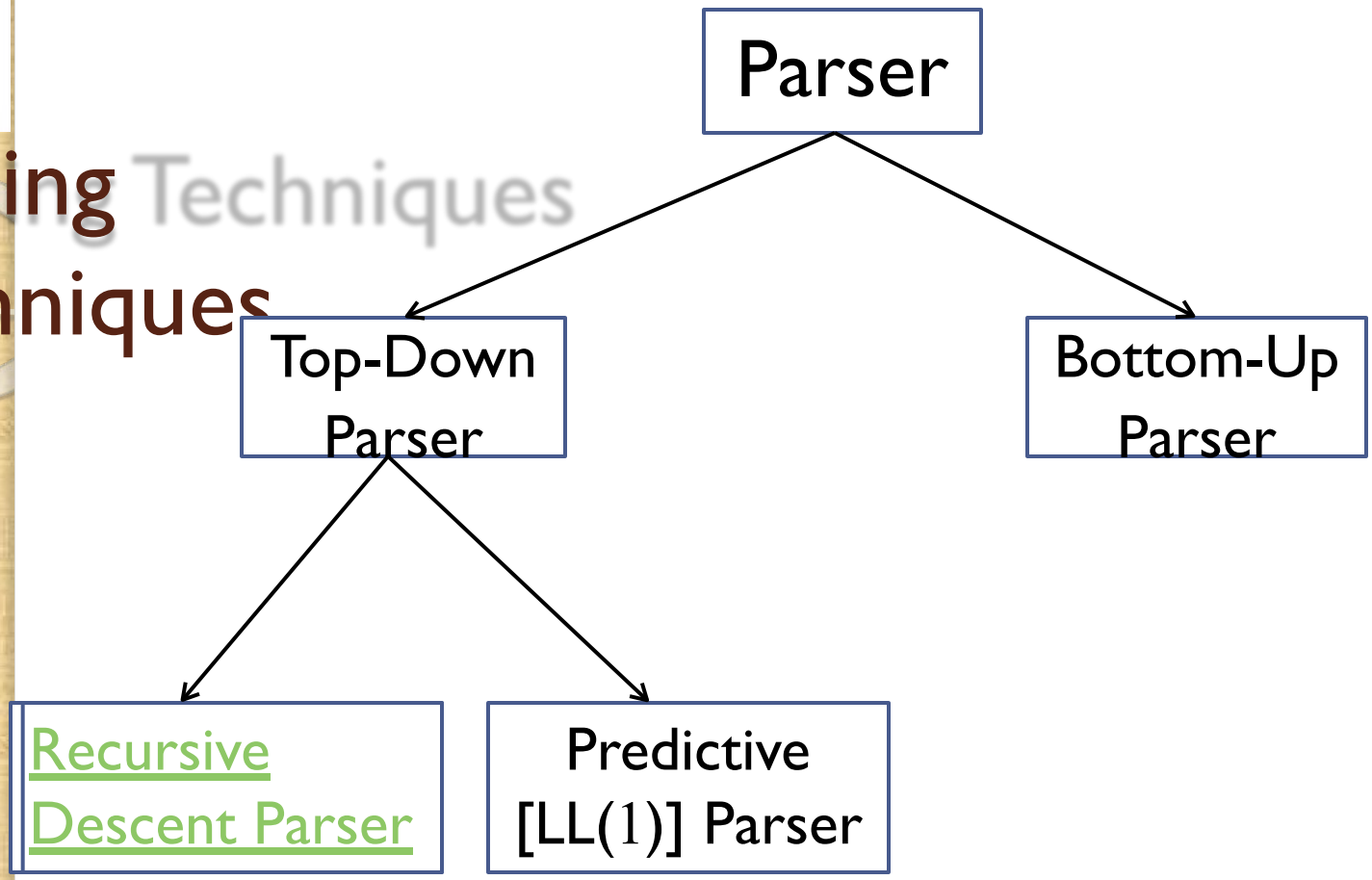
Parsing Techniques

Imp



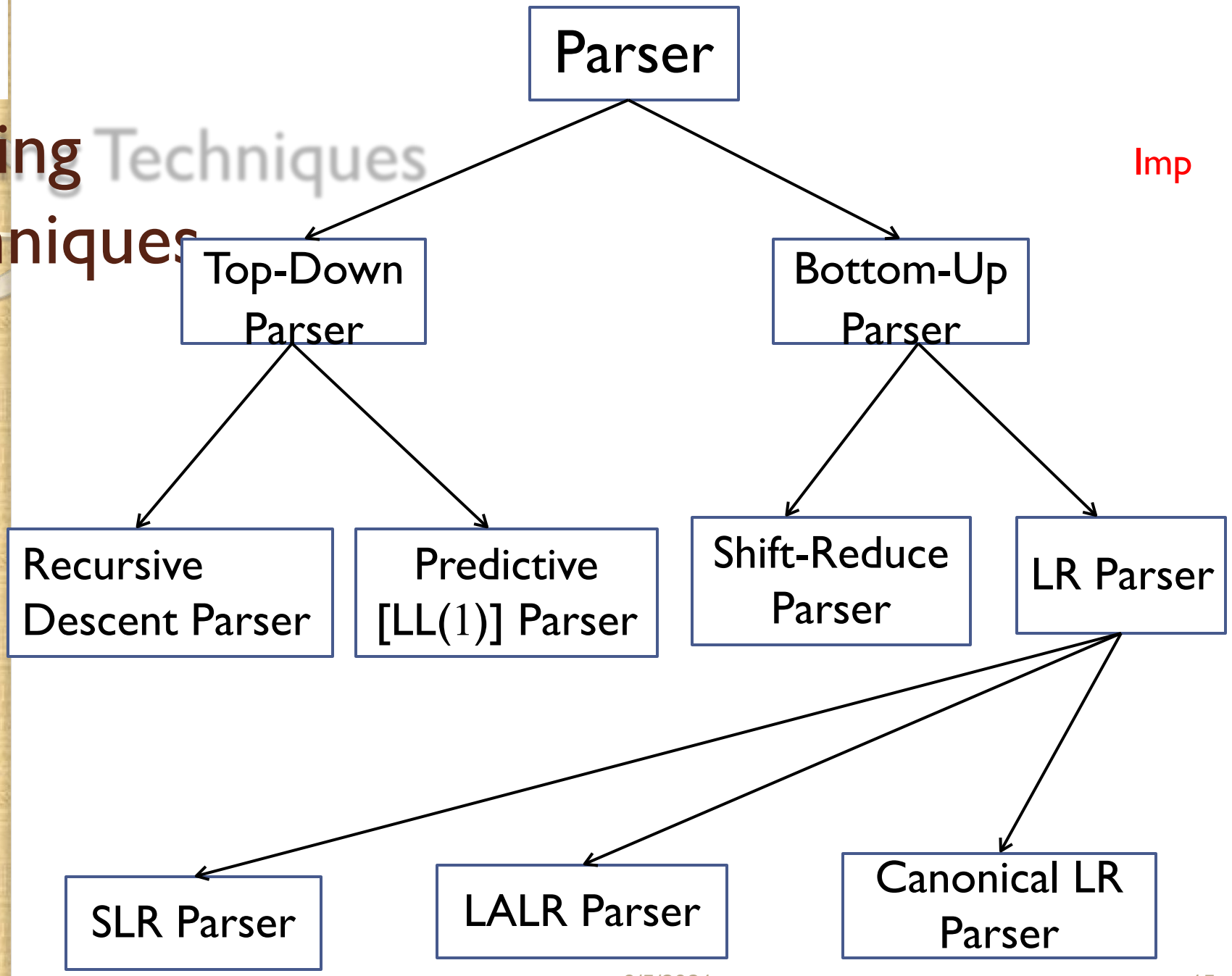
Parsing Techniques

Imp



Parsing Techniques

Imp

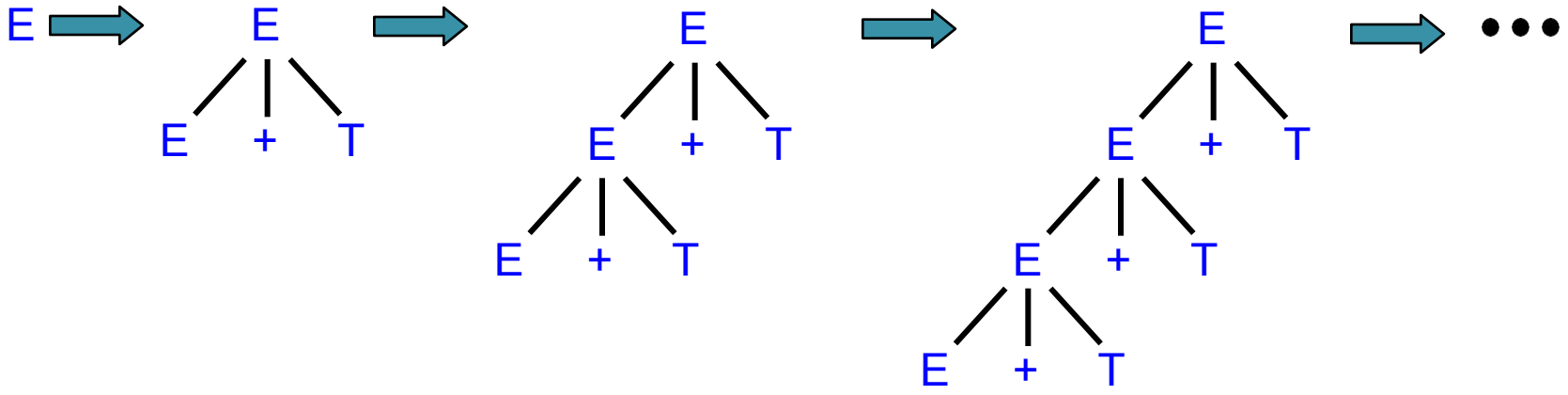


Left Recursion

Consider the grammar:

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

A top-down parser might loop forever when parsing an expression using this grammar



Left Recursion

Consider the grammar:

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

A grammar that has at least one production of the form $A \Rightarrow A\alpha$ is a left recursive grammar.

Top-down parsers do not work with left-recursive grammars.

Left-recursion can often be eliminated by rewriting the grammar.

Elimination of Left Recursion

- A grammar is **left recursive** if it has a NT A such that there is a derivation A + Aα for some string α.
- Top down parsing methods cannot handle left recursive grammars, so a transformation that eliminates left recursion is needed.

E.g. A□A α | β

It has left recursion. To eliminate it we rewrite it as

A□βA'

A' □α A' | ϵ

Contd…

- The technique to eliminate left recursion is:

A□Aα1| Aα1|…| β1| β2|…| βn

no βi begins with A.

So we replace the A-productions by

A□$\beta_1 A' | \beta_2 A' | \ldots | \beta_n A'$

A' □$\alpha_1 A' | \alpha_2 A' | \ldots. | \alpha_m A' | \epsilon$

e.g. consider the G as

S □Aa | b

A □Ac| Sd | ϵ

Left Recursion

This left-recursive grammar:

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

Can be re-written to eliminate the immediate left recursion:

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

Left Factoring

The following grammar:

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

Cannot be parsed by a predictive parser that looks one element ahead.

But the grammar can be re-written:

stmt → **if** expr **then** stmt stmt'
stmt' → **else** stmt | ε

Where ε is the empty string.

Rewriting a grammar to eliminate multiple productions starting with the same token is called left factoring.

How to do left factoring :algorithm

Algorithm :Left factoring a grammar

Input: Grammar G

Output: Eq. left factored grammar

for each NT A find the longest prefix α common to two or more its alternatives.

if $\alpha \neq \epsilon$ i.e. there is a common prefix,replace all the A- productions

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots | \alpha\beta_n | \gamma$

Where γ : all alternatives that do not begin with α

by $A \rightarrow \alpha A' | \gamma$

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

Here A' is a new NT.

Recursive descent parsing

- A parser that uses a set of recursive procedures to recognize its input is called a **recursive descent parsing.**
- It is an attempt to find a leftmost derivation for an input string
- It can be viewed as an attempt to construct a parse tree for the i/p starting from the root and creating the nodes of the parse tree in **preorder.**
- **Predictive parsing** is special case of RDP where no backtracking is required.
- Recursive descent parsers will **look ahead one character** and **advance** the input stream reading pointer when proper matches occur.

- Consider the G

S→cAd

A→ab | a

i/p is cad

S →cAd

S→c**A**d→c**ab**d

S→c**A**d→cad

A left recursive grammar can cause a RDP, even with backtracking to go into an infinite loop.

- The procedures for the arithmetic expression grammar:

- $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) | id$

input is:id+id*id$

Recursive procedures to recognize arithmetic expressions:

```
Procedure E( ):
    begin
        T( )
        E'( )
    End;
procedure E'( ):
    If input_symbol = '+' then
      begin
        ADVANCE( )
        T( )
        E'( )
      end;
```

Contd…

```
procedure T( ) :
        begin
              F( )
              T'( )
        End;

procedure T'( )
        If input_symbol = '*' then
        begin
         ADVANCE( )
          F( )
          T'( )
        End;
```

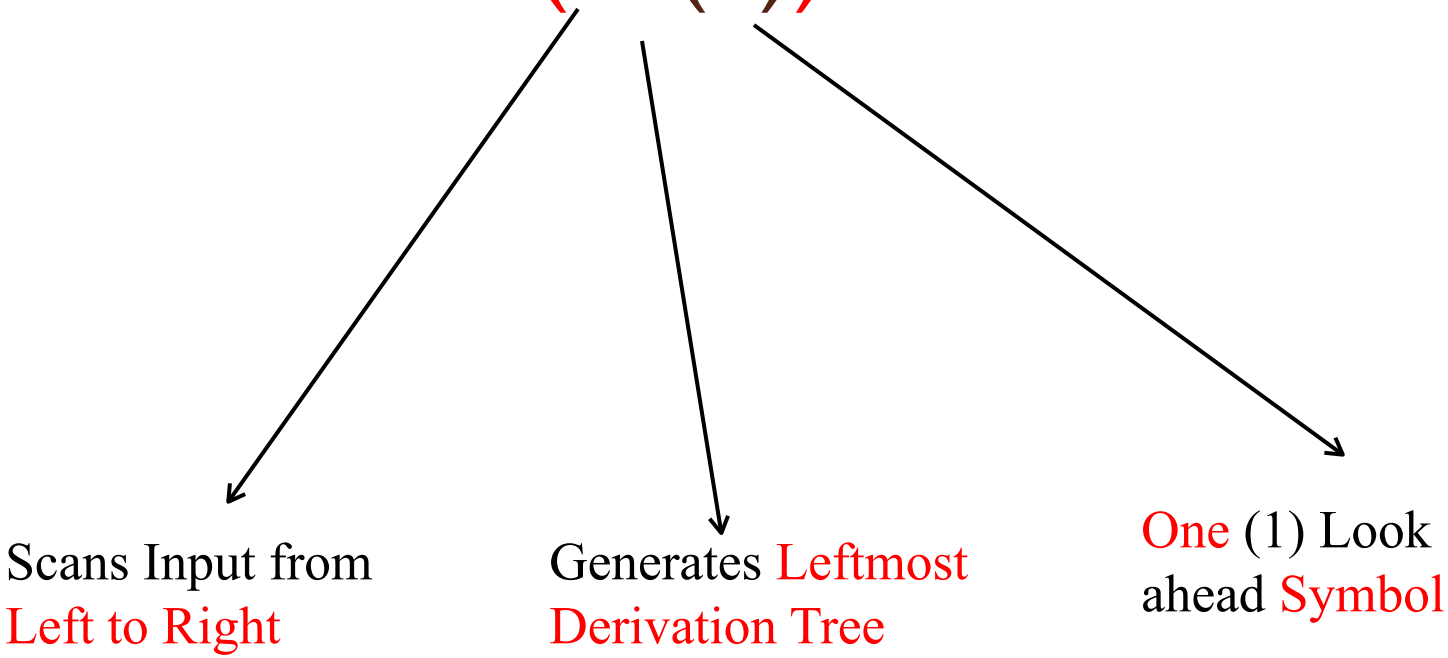
Contd…

```
procedure F( ):
    If input_symbol = 'id'
    then ADVANCE( )
   Else if input Symbol=' ( ' then
    begin
        ADVANCE( )
        E( )
        If input_symbol = ' ) '
        then ADVANCE( )
        else ERROR( )
   end
   else
  ERROR( )
```

NOTE: ADVANCE( ) moves the input pointer to the next input symbol

Predictive Parser (LL(1))

- Here (LL(1))



we can have (LL(k)) parsers also

- No Backtracking Parser:
- Writing Special Grammar – Eliminating Left Recursion & Left Factoring.
- Must know – Current Input Symbol (a) & Non terminal (A) to be expanded.

  A -> α1/ α2 / α3 ……/ αn

- Prediction of Match

- LL(1) Parser uses explicit STACK rather than Recursive calls.
- Stack Implementation:

Stack	W (Input String)
$ Start Symbol	Input String $
- - - - - - -	- - - - - - -
- - - - - - -	- - - - - - -
$	$ accept

A Predictive Parser

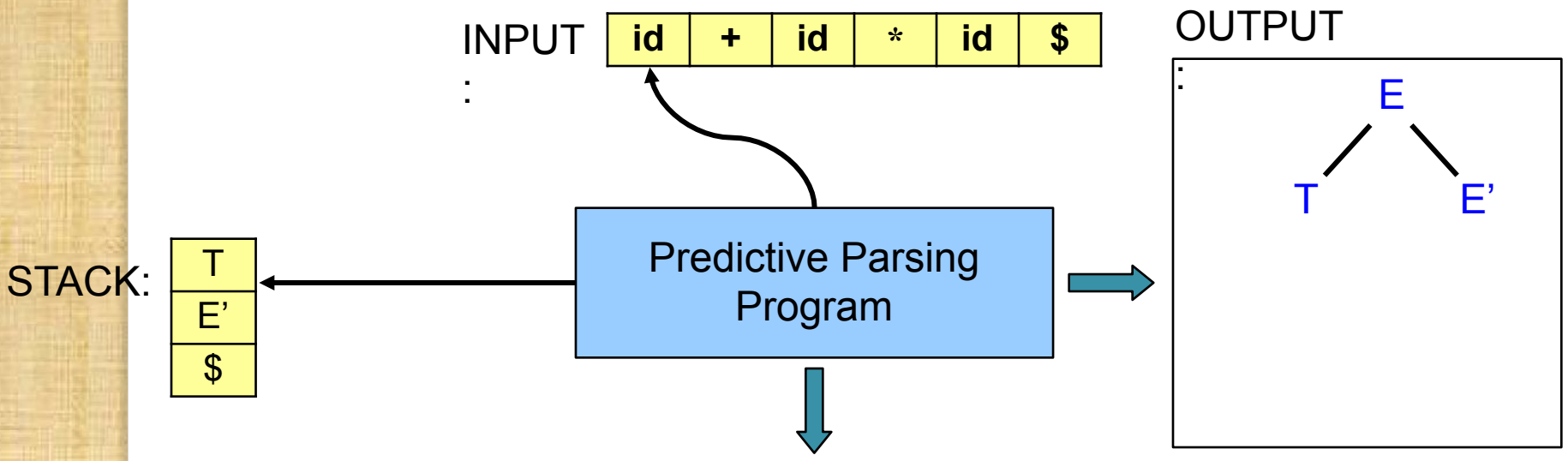
Grammar:

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

Parsing Table:

NON-TERMINAL	INPUT SYMBOL					
	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)		

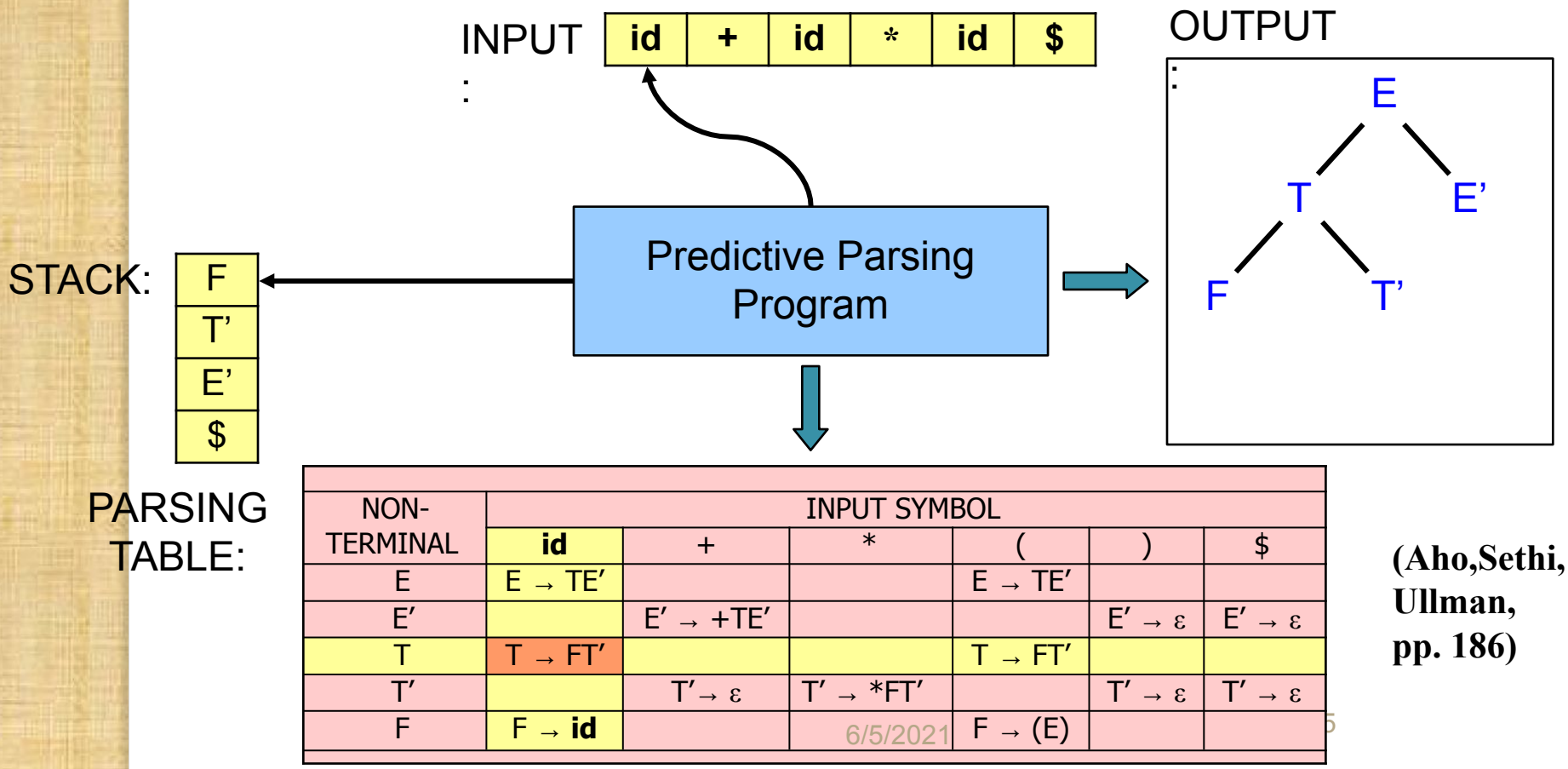
A Predictive Parser



PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	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)		

A Predictive Parser

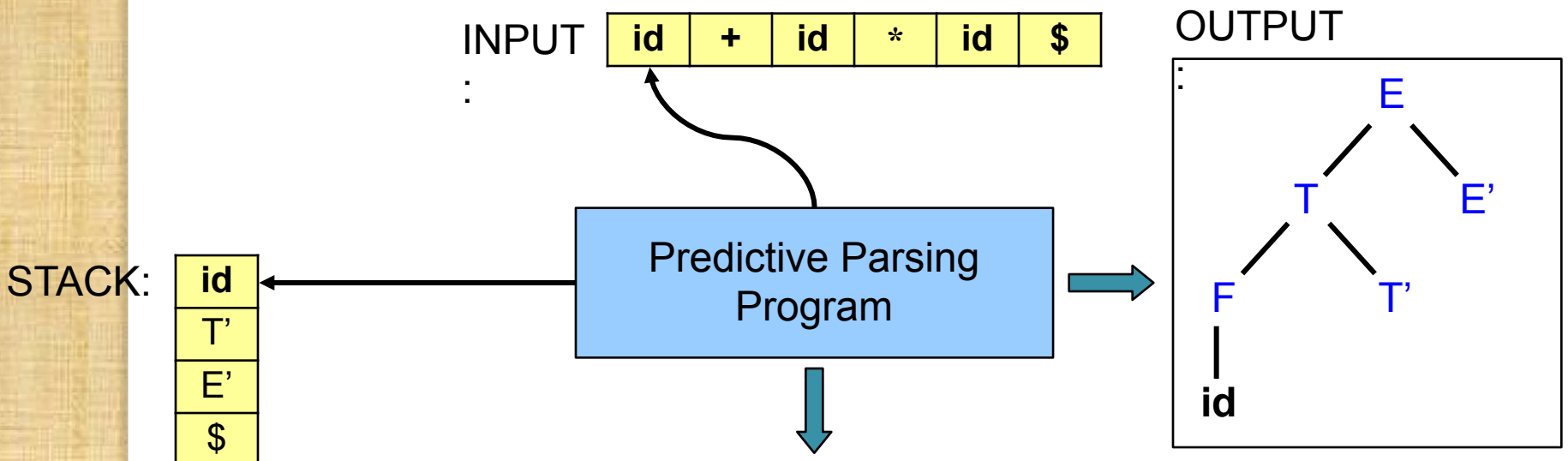


PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	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)		

(Aho,Sethi, Ullman, pp. 186)

A Predictive Parser



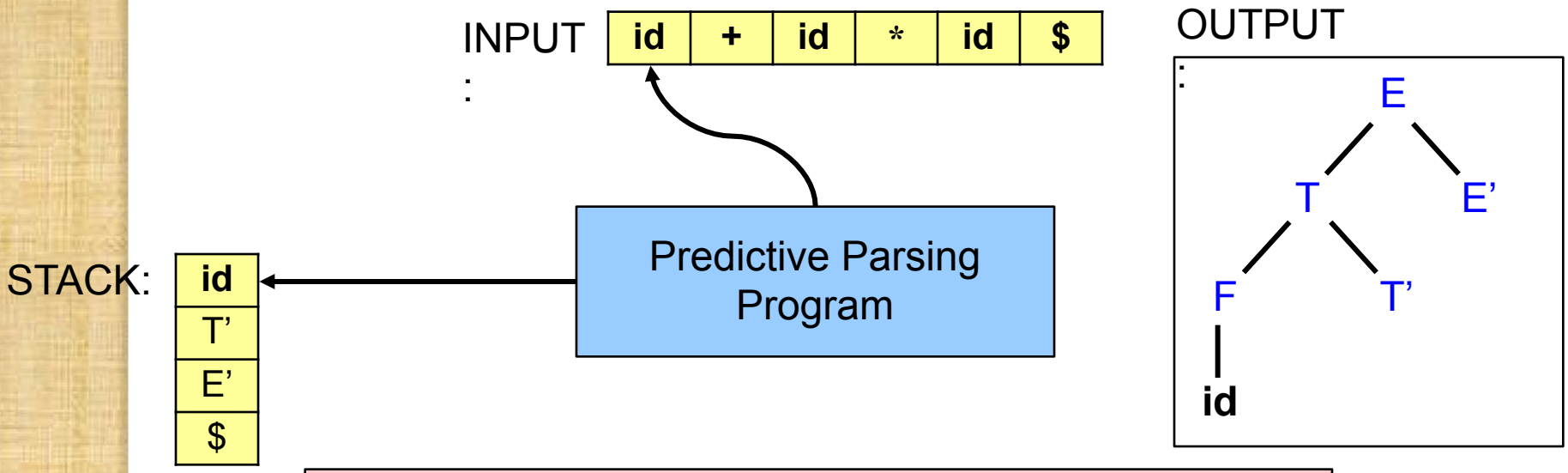
PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	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)		

(Aho,Sethi, Ullman, pp. 188)

A Predictive Parser

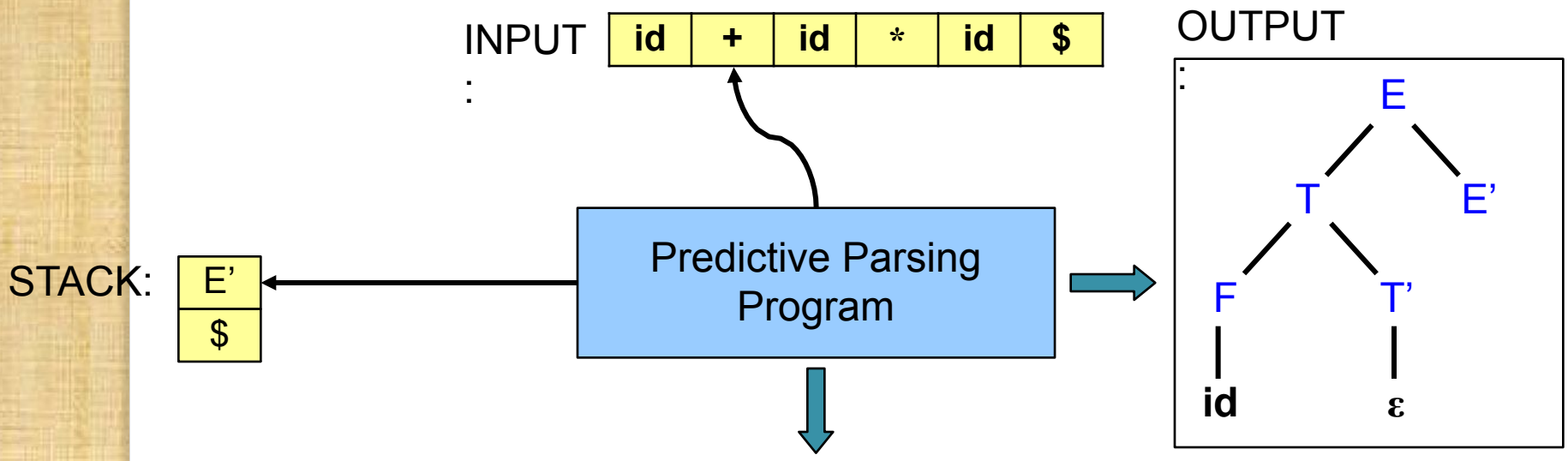
Action when Top(Stack) = input ≠ $: Pop stack, advance input.



PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	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)		

A Predictive Parser



PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	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)		

A Predictive Parser

The predictive parser proceeds in this fashion emiting the following productions:

$E' \rightarrow +TE'$

$T \rightarrow FT'$

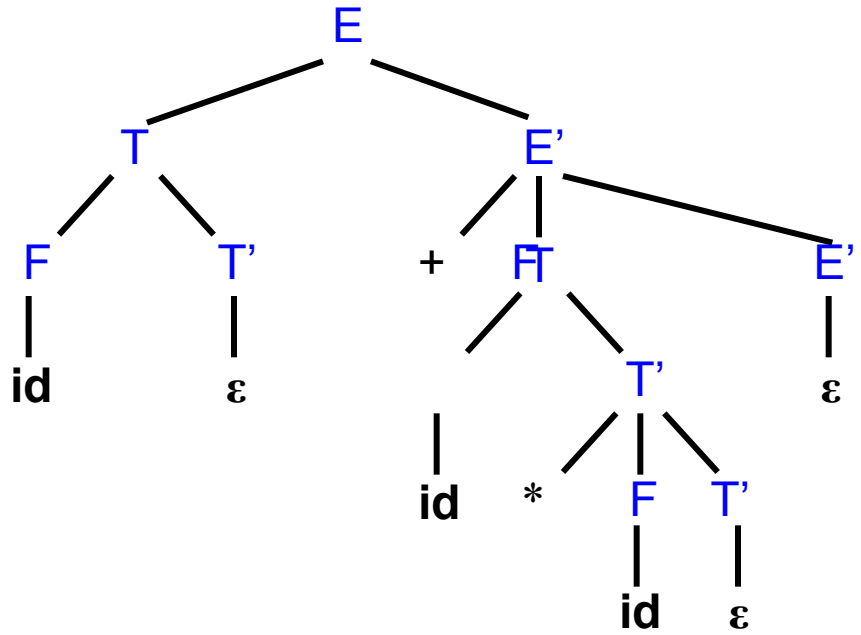
$F \rightarrow \mathbf{id}$

$T' \rightarrow * FT'$

$F \rightarrow \mathbf{id}$

$T' \rightarrow \varepsilon$

$E' \rightarrow \varepsilon$



When Top(Stack) = input = $ the parser halts and accepts the input string.

LL(*k*) Parser

This parser parses **from left to right**, and does a **leftmost-derivation**. It looks up **1 symbol ahead** to choose its next action. Therefore, it is known as a **LL(1)** parser.

An **LL(k)** parser looks **k symbols ahead** to decide its action.

The Parsing Table

Given this grammar:

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

How is this parsing table built?

PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	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)		

FIRST and FOLLOW

We need to build a FIRST set and a FOLLOW set for each symbol in the grammar.

The elements of FIRST and FOLLOW are terminal symbols.

FIRST(α) is the set of terminal symbols that can begin any string derived from α.

FOLLOW(α) is the set of terminal symbols that can follow α:

$t \in \text{FOLLOW}(\alpha) \leftrightarrow \exists$ derivation containing αt

Rules to Create FIRST

GRAMMAR:

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

SETS:

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

FIRST rules:

1. If X is a terminal, FIRST(X) = {X}
2. If $X \rightarrow \varepsilon$, then $\varepsilon \in FIRST(X)$
3. If $X \rightarrow Y_1 Y_2 \bullet\bullet\bullet Y_k$ and $Y_1 \bullet\bullet\bullet Y_{i-1} \overset{*}{\Rightarrow} \varepsilon$ and $a \in FIRST(Y_i)$ then $a \in FIRST(X)$

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

es to Create FOLLOW

GRAMMAR:

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

SETS:

FOLLOW(E) = {~~$~~} {), $}
FOLLOW(E') = {), $}
FOLLOW(T) = {), $}

FOLLOW rules:

1. If S is the start symbol, then $ ∈ FOLLOW(S)
2. If A → αBβ,
   and a ∈ FIRST(β)
   and a ≠ ε
   then a ∈ FOLLOW(B)
3. If A → αB
   and a ∈ FOLLOW(A)
   then a ∈ FOLLOW(B)

3a. If A → αBβ
   and $\beta \Rightarrow^* \varepsilon$
   and a ∈ FOLLOW(A)
   then a ∈ FOLLOW(B)

A and B are non-terminals,
α and β are strings of grammar symbols

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

ules to Create FOLLOW

GRAMMAR:

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

SETS:

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

FOLLOW rules:

1. If S is the start symbol, then $ ∈ FOLLOW(S)
2. If A → αBβ,
   and a ∈ FIRST(β)
   and a ≠ ε
   then a ∈ FOLLOW(B)
3. If A → αB
   and a ∈ FOLLOW(A)
   then a ∈ FOLLOW(B)

3a. If A → αBβ
   and $\beta \Rightarrow^* \varepsilon$
   and a ∈ FOLLOW(A)
   then a ∈ FOLLOW(B)

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

les to Create FOLLOW

GRAMMAR:

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

SETS:

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

FOLLOW rules:

1. If S is the start symbol, then $ ∈ FOLLOW(S)
2. If A → αBβ,
   and a ∈ FIRST(β)
   and a ≠ ε
   then a ∈ FOLLOW(B)

3a. If A → αBβ
   and $\beta \Rightarrow^* \varepsilon$
   and a ∈ FOLLOW(A)
   then a ∈ FOLLOW(B)

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

ules to Create FOLLOW

FOLLOW

GRAMMAR:

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

SETS:

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

FOLLOW rules:

1. If S is the start symbol, then $ ∈ FOLLOW(S)
2. If $A \rightarrow \alpha B \beta$,
   and $a \in FIRST(\beta)$
   and $a \neq \varepsilon$
   then $a \in FOLLOW(B)$
3. If $A \rightarrow \alpha B$
   and $a \in FOLLOW(A)$
   then $a \in FOLLOW(B)$

3a. If $A \rightarrow \alpha B \beta$
and $\beta \Rightarrow^* \varepsilon$
and $a \in FOLLOW(A)$
then $a \in FOLLOW(B)$

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

ules to Create FOLLOW

GRAMMAR:

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

SETS:

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

FOLLOW rules:

1. If S is the start symbol, then $ ∈ FOLLOW(S)
2. If A → αBβ,
   and a ∈ FIRST(β)
   and a ≠ ε
   then a ∈ FOLLOW(B)
3. If A → αB
   and a ∈ FOLLOW(A)
   then a ∈ FOLLOW(B)

3a. If A → αBβ
   and $\beta \Rightarrow^* \varepsilon$
   and a ∈ FOLLOW(A)
   then a ∈ FOLLOW(B)

(Aho,Sethi,Ullman, pp. 189)

GRAMMAR:

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

FIRST SETS:

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

FOLLOW SETS:

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

1. If A → α:
   if a ∈ FIRST(α), add A → α to M[A, a]

PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	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)		

GRAMMAR:

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

FIRST SETS:

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

FOLLOW SETS:

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

ules t arsin

1. If A → α:
   if a ∈ FIRST(α), add A → α to M[A, a]

PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	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)		

GRAMMAR:

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

FIRST SETS:

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

FOLLOW SETS:

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

1. If A → α:
   if a ∈ FIRST(α), add A → α to M[A, a]

PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	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)		

GRAMMAR:

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

FIRST SETS:

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

FOLLOW SETS:

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

ules t arsin

1. If A → α:
   if a ∈ FIRST(α), add A → α to M[A, a]

PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	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)		

GRAMMAR:

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

FIRST SETS:

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

FOLLOW SETS:

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

ules t arsin

1. If A → α:
   if a ∈ FIRST(α), add A → α to M[A, a]

PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	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)		

GRAMMAR:

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

FIRST SETS:

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

FOLLOW SETS:

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

1. If A → α:
   if a ∈ FIRST(α), add A → α to M[A, a]
2. If A → α:
   if ε ∈ FIRST(α), add A → α to M[A, b]
   for each terminal b ∈ FOLLOW(A),

PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	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)		

GRAMMAR:

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

FIRST SETS:

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

FOLLOW SETS:

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

1. If A → α:
   if a ∈ FIRST(α), add A → α to M[A, a]
2. If A → α:
   if ε ∈ FIRST(α), add A → α to M[A, b]
   for each terminal b ∈ FOLLOW(A),

PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	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)		

GRAMMAR:

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

FIRST SETS:

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

FOLLOW SETS:

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

1. If A → α:
   if a ∈ FIRST(α), add A → α to M[A, a]
2. If A → α:
   if ε ∈ FIRST(α), add A → α to M[A, b]
   for each terminal b ∈ FOLLOW(A),
3. If A → α:
   if ε ∈ FIRST(α), and $ ∈ FOLLOW(A),
   add A → α to M[A, $]

PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E → TE′			E → TE′		
E′		E′ → +TE′			E′ → ε	E′ → ε
T	T → FT′			T → FT′		
T′		T′→ ε	T′ → *FT′		T′ → ε	T′ → ε
F	F → **id**			F → (E)		

LL(1) Parsing Algorithm: (Table Based)

- Given

An LL(1) grammar, a parsing algorithm that uses the LL(1) parsing table

- Note: - Assuming that '$' indicates the bottom of the stack and the end of the input string –

1. Push the start symbol onto the top of the parsing stack.
2. "While" the top of the stack ≠$ and the next input token ≠$ do
3. If the top of the parsing stack is terminal a and the next input token = a

then (match and pop)

pop the parsing stack

advance the input

elseif the top of the parsing stack is Non Terminal A and the next input is Terminal a and parsing table entry M[A, a] contains production A->X1,X2,……Xn

then (generate)

pop the parsing stack

for(i=n;i<=1;i++)

push Xi on top of the stack

else error

if (the top of the parsing stack =$)

and the next input token =$

then accept

else error. Back

Bottom Up Parser

- Bottom-up parsers are basically those generates a parse tree starting from Leaves (Bottom) and creating nodes up to Root of parse tree.
- The reduction steps trace a rightmost derivation on reverse.
- More Powerful than Top down Parsers.
- Uses explicit Stack.

Bottom Up Parser

- Stack Implementation:

Stack	W (Input String)
$	Input String $
- - - - - - -	- - - - - - -
- - - - - - - -	- - - - - - - -
$ Start Symbol	$ accept

- Different types of Bottom up parser:

  -Shift Reduce Parser:

  - Operator Precedence Parser:

  - LR Parsers:

    - Simple LR (SLR)parser
    - LALR parser
    - Canonical LR (CLR)parser

- Actions:
  - Shift:
  - Reduce:
  - Accept:
  - Error:

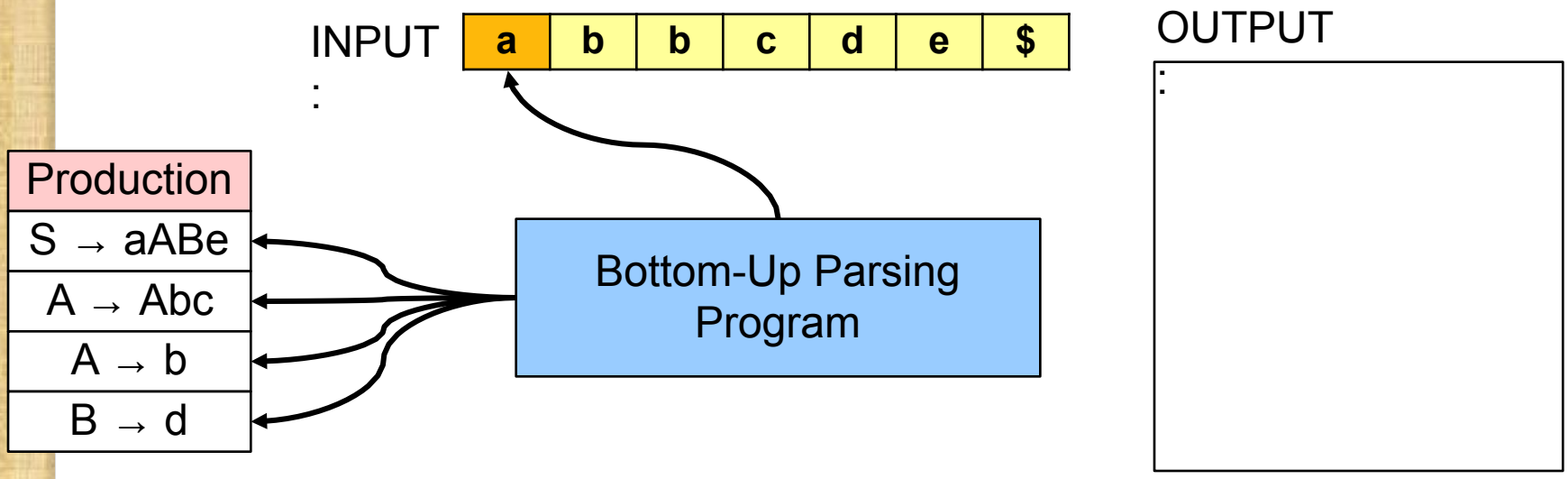
Bottom-Up Parser

Consider the Grammar:

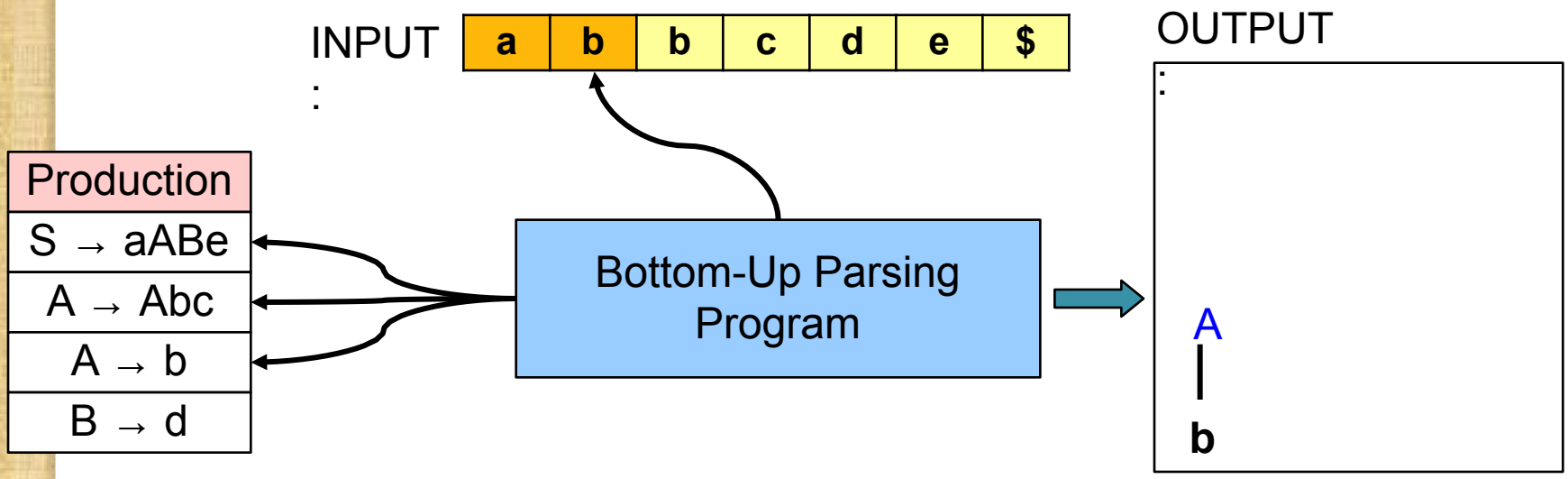
$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

We want to parse the input string abbcde.

Bottom-Up Parser Example

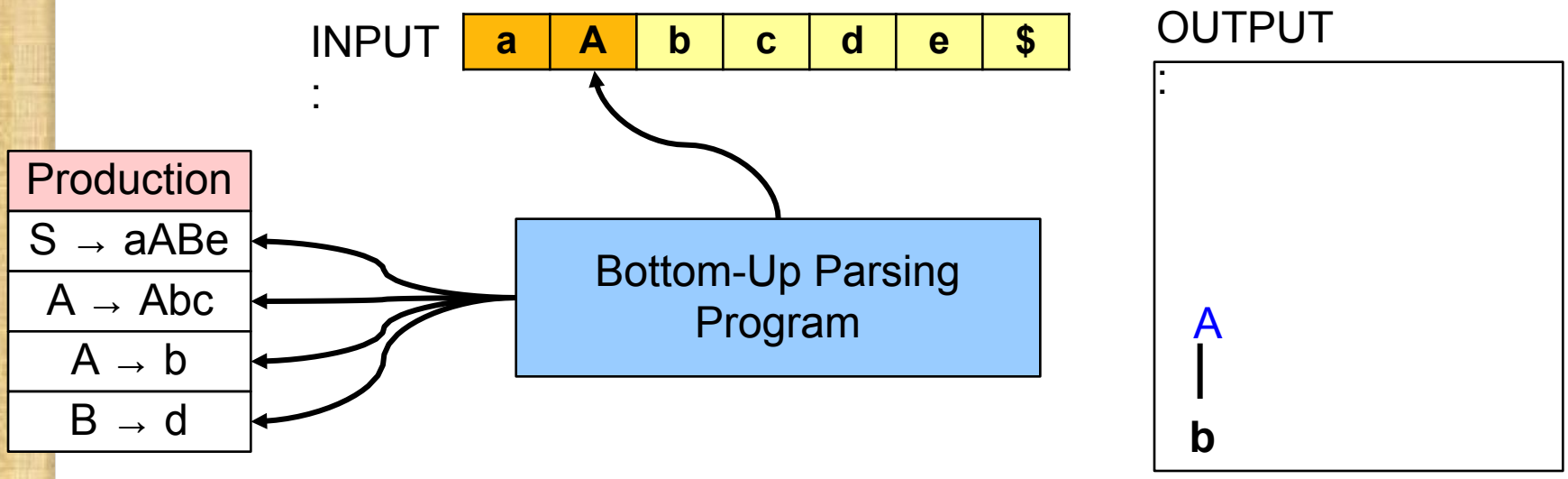


Bottom-Up Parser Example



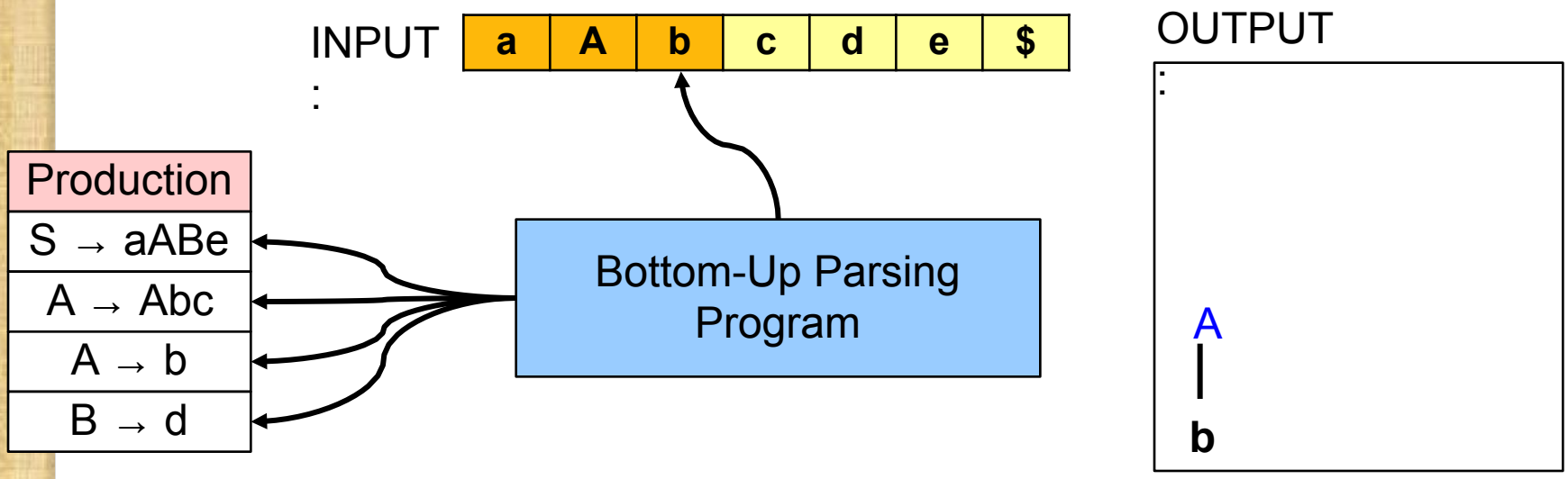
(Aho,Sethi,Ullman, pp. 195)

Bottom-Up Parser Example



(Aho,Sethi,Ullman, pp. 195)

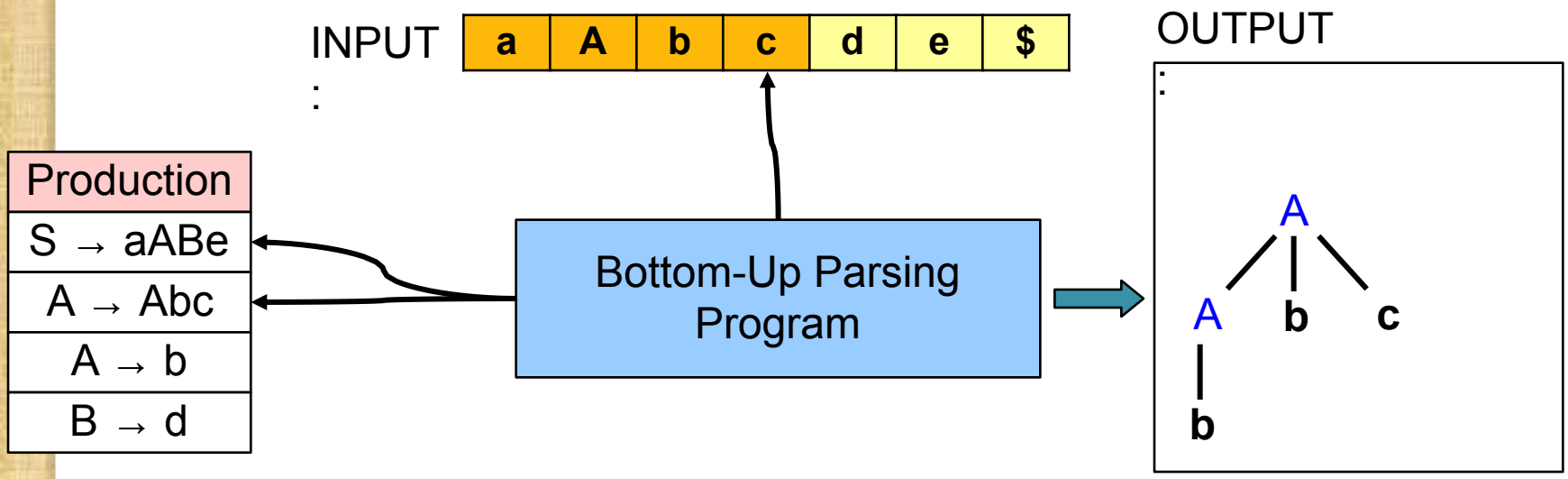
Bottom-Up Parser Example



We are not reducing here in this example.

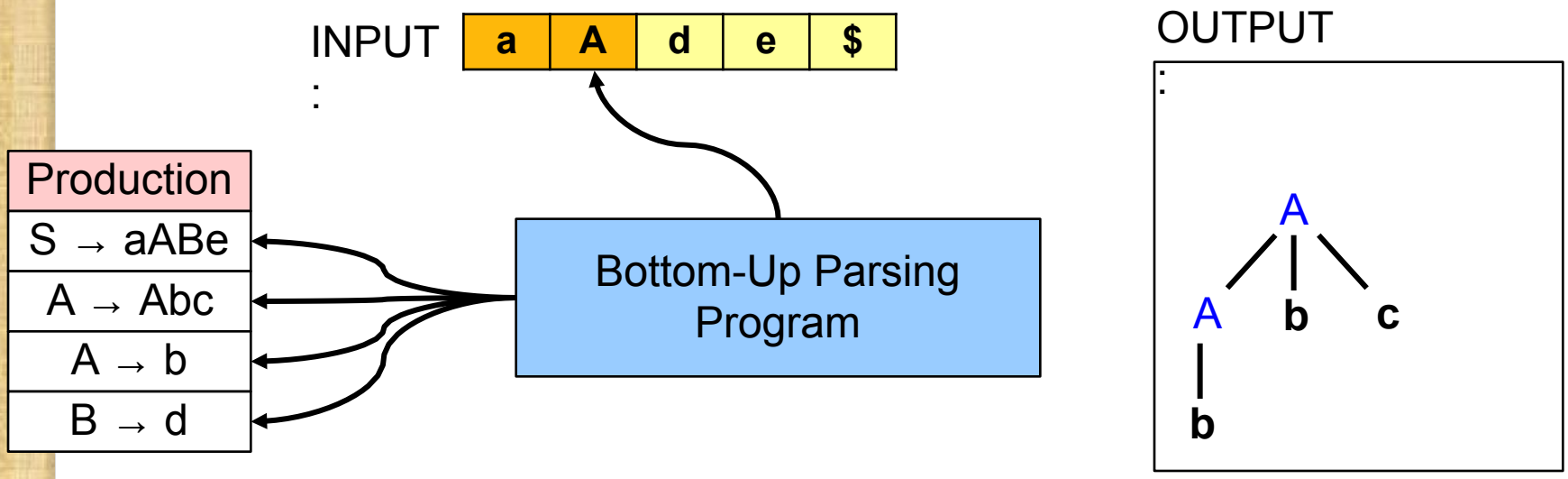
A parser would reduce, get stuck and then backtrack!

Bottom-Up Parser Example



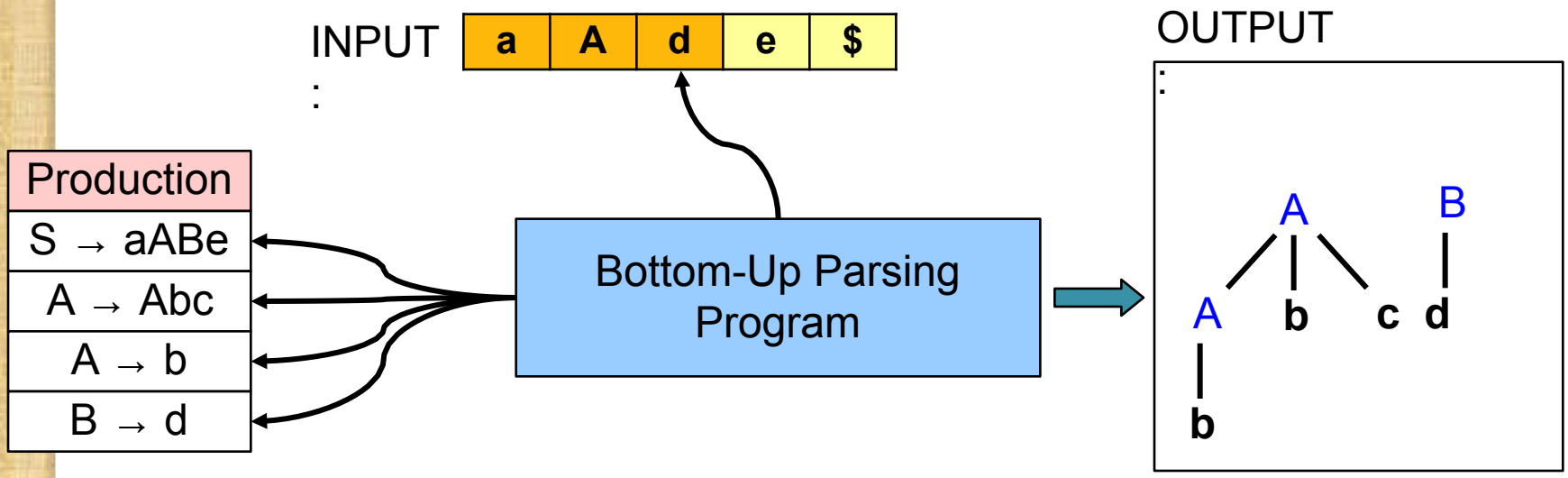
(Aho,Sethi,Ullman, pp. 195)

Bottom-Up Parser Example



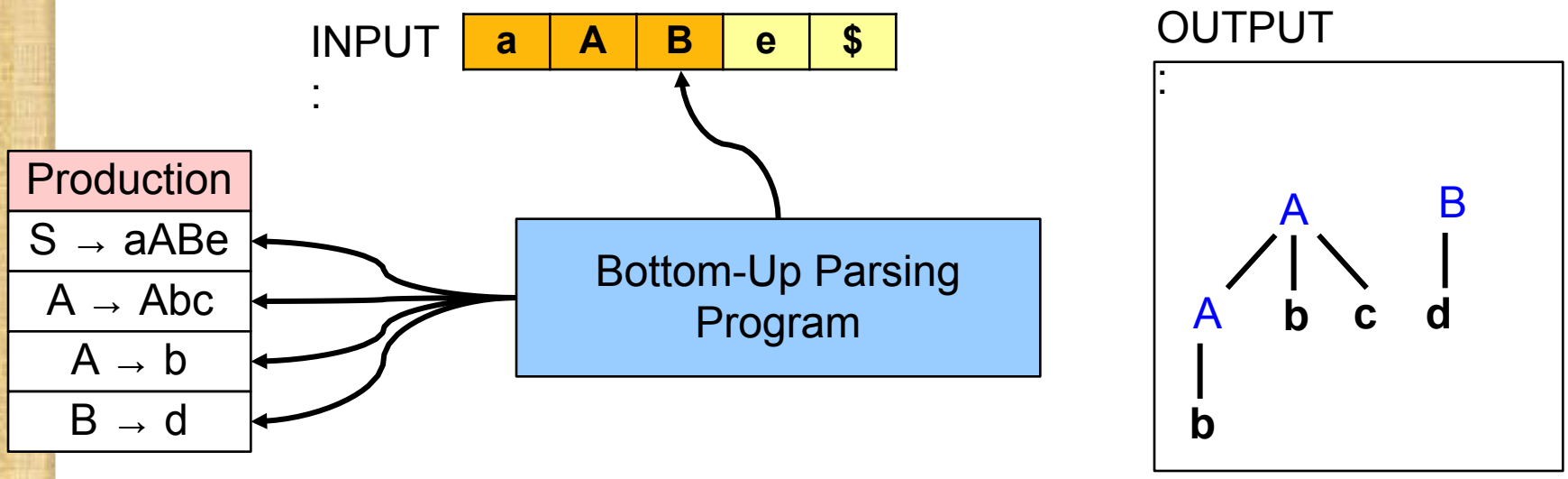
(Aho,Sethi,Ullman, pp. 195)

Bottom-Up Parser Example



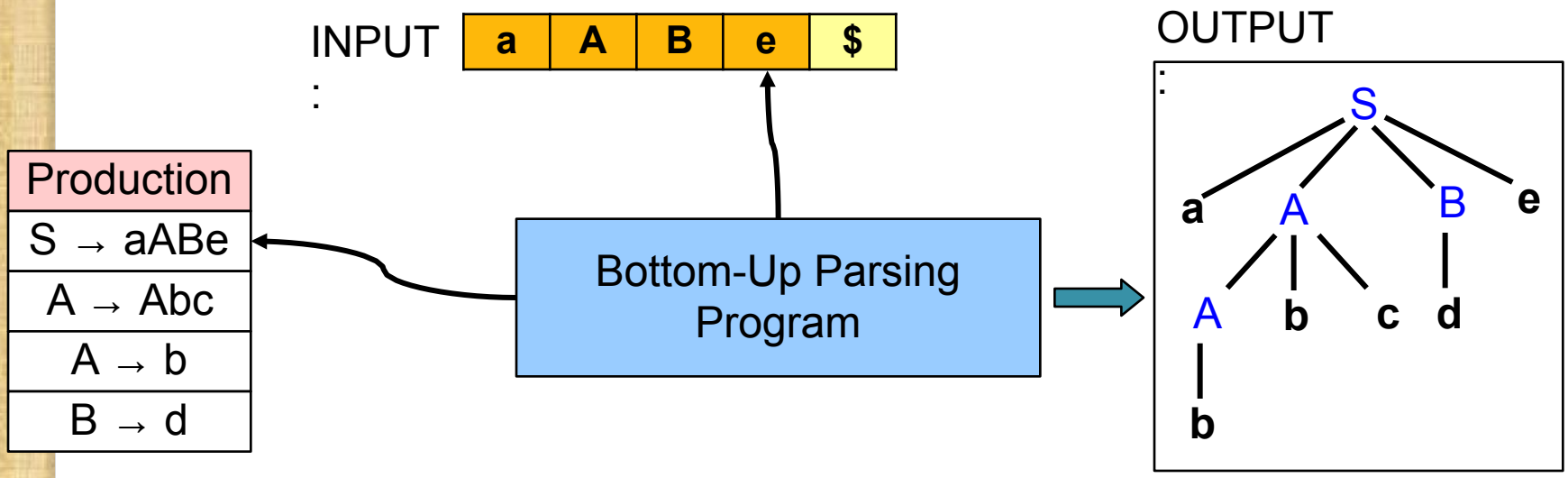
(Aho,Sethi,Ullman, pp. 195)

Bottom-Up Parser Example



(Aho,Sethi,Ullman, pp. 195)

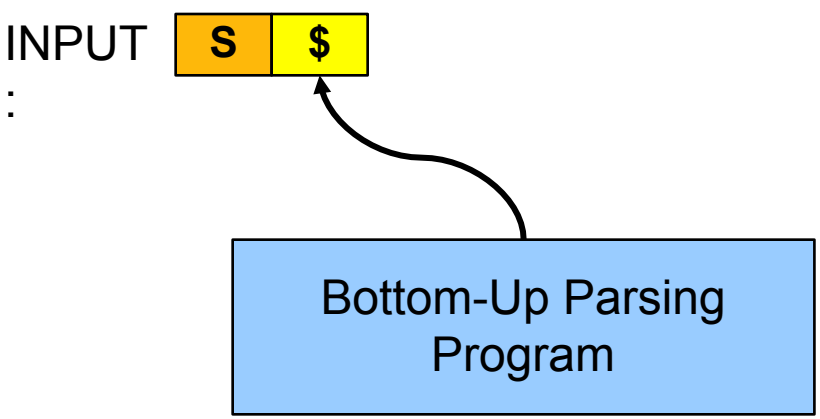
Bottom-Up Parser Example

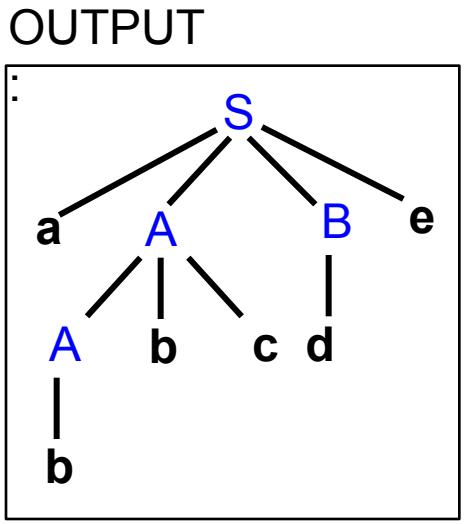


(Aho,Sethi,Ullman, pp. 195)

Bottom-Up Parser Example

Production
S → aABe
A → Abc
A → b
B → d





This parser is known as an LR Parser because it scans the input from Left to right, and it constructs a Rightmost derivation in reverse order.

(Aho,Sethi,Ullman, pp. 195)

Handle pruning

- **Principles of Bottom Up Parsing – Handles**
- The **leftmost simple phrase** of a sentential form is called the *handle*.

The basic steps of a bottom-up parser are

- to identify a *substring* within a *rightmost sentential form* which matches the *RHS of a rule*.
- when this substring is replaced by the LHS of the matching rule, it must produce the previous rightmost- sentential form.

Such a substring is called a *handle* .

- A *handle* of a right sentential form γ, is
  - a production rule $A \rightarrow \beta$, and
  - an occurrence of a sub-string β in γ

such that when the occurrence of β is replaced by A in γ, we get the previous right sentential form in a rightmost derivation of γ.

Handle pruning

$$S \overset{*rm}{\Rightarrow} \alpha A w \overset{rm}{\Rightarrow} \alpha \beta w$$

then the rule A□β and the occurrence β is the handle in βw.

Grammar is
E->E+E
E→E*E
E→(E)
E→id

Derive string " id+id*id" using rightmost derivation.

Note: String at the right of handle contains only terminal symbols.

Handle pruning

The process of discovering a handle & reducing it to the appropriate left-hand side is called ***handle pruning*.**

Handle pruning forms the basis for a bottom-up parsing method.

Reduction made by a shift reduce parser

Right sentential Form	Handle	Reducing Production
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		

Contd…

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T/F \mid F$$
$$F \rightarrow P * * F \mid P$$
$$P \rightarrow -P \mid B$$
$$B \rightarrow (E) \mid \text{id}$$

Input string is: − id ** id/id

Contd..

Stack	Input	Parser move
$	– id ** id/id$	shift –
$ –	id** id /id $	shift id
$ – id	** id / id $	reduce by B□id
$ – B	** id / id $	reduce by P□B
$ – P	** id / id $	reduce by P□ – P
$ P	** id / id $	shift **
…	…	….
$E	$	accept

Contd…

- Shift- reduce parsers require the following data structures

1. a buffer for holding the input string to be parsed
2. a data structure for detecting handles (stack)
3. a data structure for storing and accessing the LHS and RHS of rules.

Contd…

- **Stack implementation of shift-reduce parsing**

In handle pruning 2 problems are to be solved

1. Locate the substring to be reduced in a right sentential form
2. Determine what prod to choose in case there is more that one prod with that substring on the right side

- The parser operates by shifting zero or more i/p symbols onto the stack until a handle β is on top.

-Then reduce β to the left side of the appropriate prod.

- Repeat until an error is detected or stack contains the start symbol and i/p is empty.

- **Show moves by parser for string "id1+id2*id3" using arithmetic expression G.**

Contd…

- Basic actions of the shift-reduce parser are:

Shift: Moving a single token from the input buffer onto the stack till a handle appears on the stack.

Reduce: When a handle appears on the stack, it is popped and replaced by the left hand side of the corresponding production.

Accept: When the stack contains only the start symbol and input buffer is empty, the parser halts announcing a *successful* parse.

Error: When the parser can neither shift nor reduce nor accept. Halts announcing an error.

Contd…

- Conflicts in a Shift-Reduce Parser

Following conflicting situations may get into shift-reduce grammar

1. **Shift - reduce conflict**

   A handle **β** occurs on **TOS**; the next token a is such that **βaγ** happens to be another handle.

   the parser has two options

   - Reduce the handle using **A□ β**
   - Ignore the handle β ; shift **a** and continue parsing and eventually reduce using **B□ βaγ**

2. **Reduce- reduce conflict**

   the stack contents are αβγ and both βγ and γ are handles with A□ βγ and B□ γ as the corresponding rules.

Then parser has two reduce possibilities:

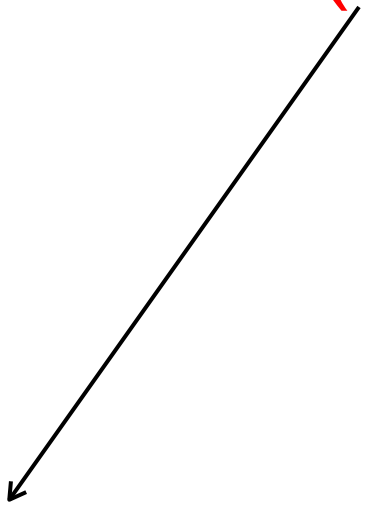
- Choose shift(or reduce) in a shift reduce conflict
- Prefer one reduce (over others) in a reduce-reduce conflict

LR Parsers

- Used for Large Class of ‘G’/ CFG.
- Called LR(K) Parsing.

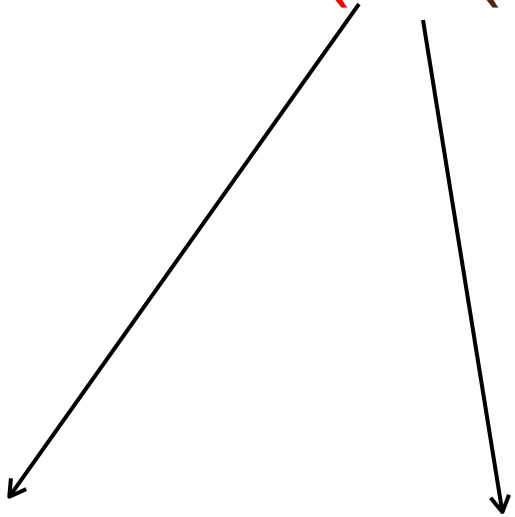
LR Parsers

- Here (LR(K))



Scans Input from
Left to Right

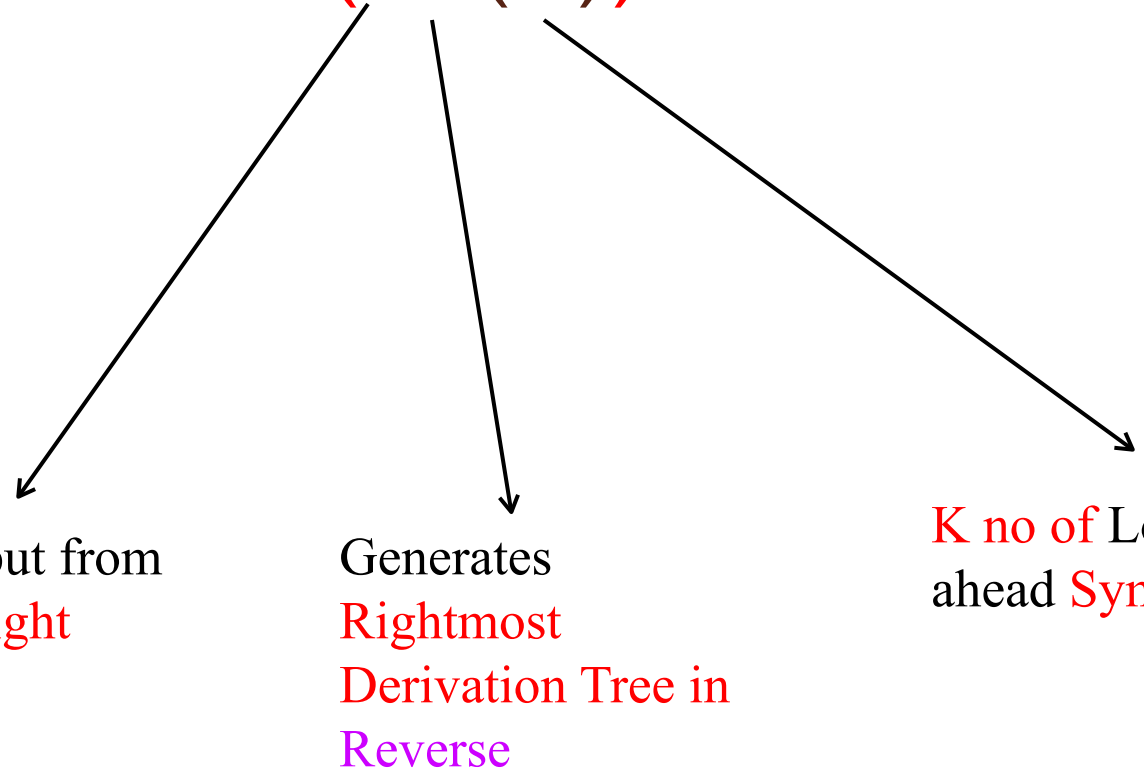
- Here (LR(K))



Scans Input from
Left to Right

Generates
Rightmost
Derivation Tree in
Reverse

- Here (LR(K))


Scans Input from
Left to Right
Generates
Rightmost
Derivation Tree in
Reverse
K no of Look
ahead Symbols

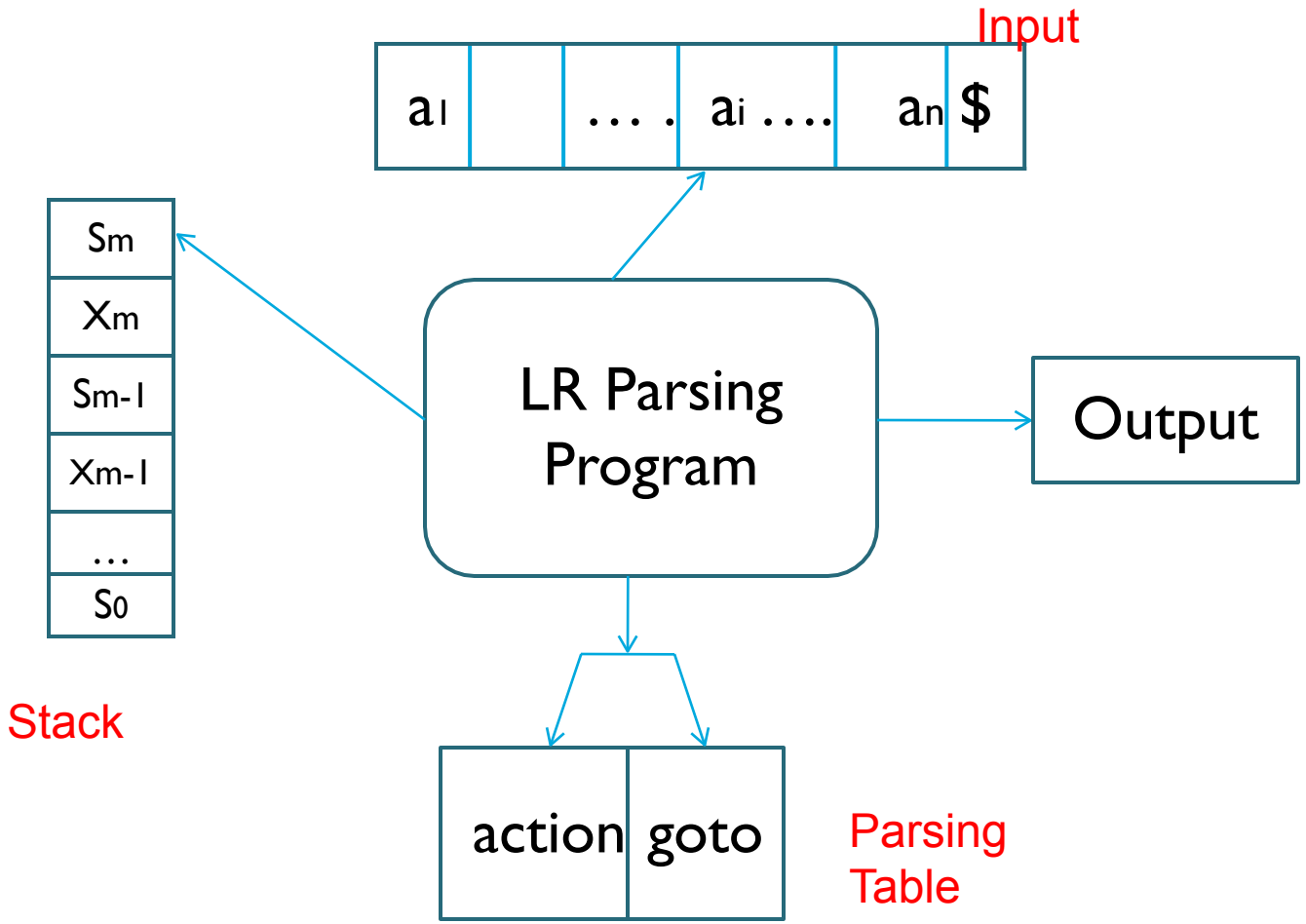
Properties of LR Parsers

- Can be constructed for which it is possible to write CFG.
- Most general Non-backtracking S-R parsing method.
- Proper Superset of CFG that can be parsed by Predictive Parsers.
- Can detect Syntactic Errors as soon as while Scanning the i/p.
- Major drawback is – Too much work to construct an LR parser by hand.

Automatic Parser Generator -YACC

Block Schematic of LR Parser:

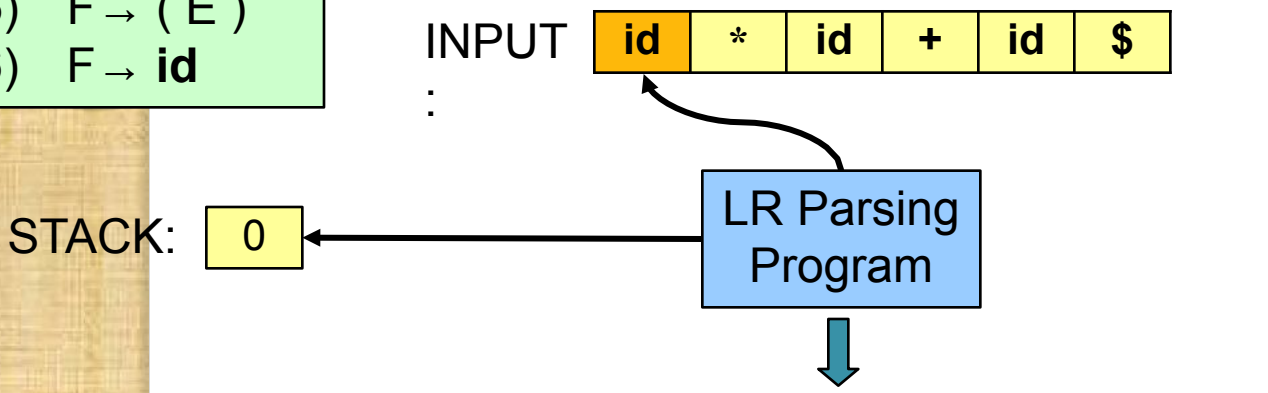
- A table driven Parser has an I/p Buffer, a Stack, a Parsing Table and O/p stream along with Driver Program.



GRAMMAR:

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

LR Parser Example



OUTPUT
:

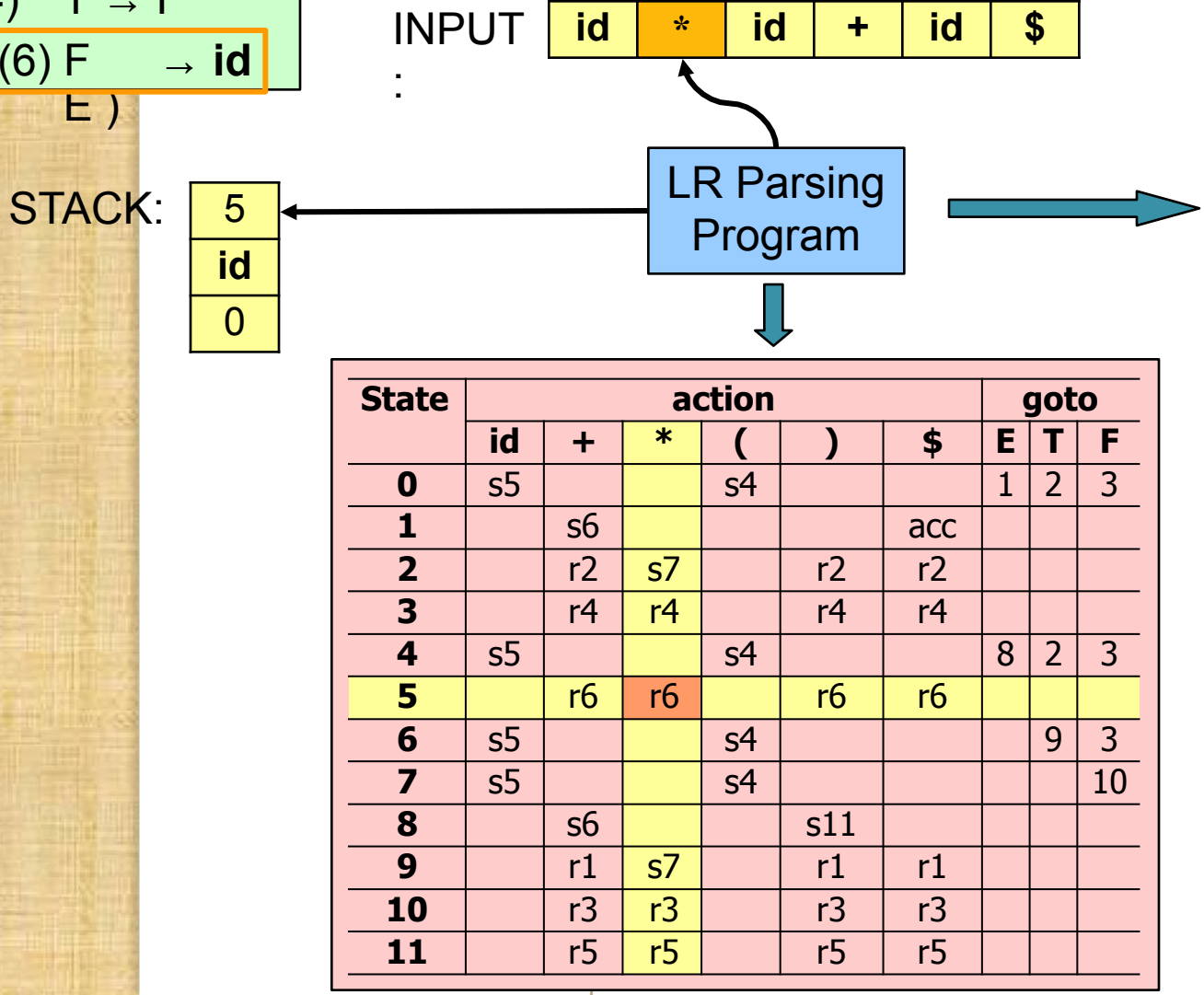
State	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

(Aho,Sethi,Ullman, pp. 220)

LR Parser Example

GRAMMAR:

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



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			

(Aho,Sethi,Ullman, pp. 220)

GRAMMAR:

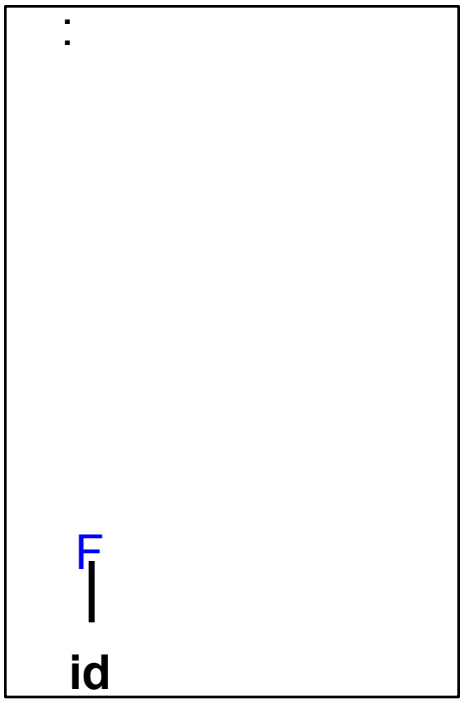
(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(6) F → id

R Parser Example

INPUT :

id	*	id	+	id	$

OUTPUT :



LR Parsing Program

STACK: 0

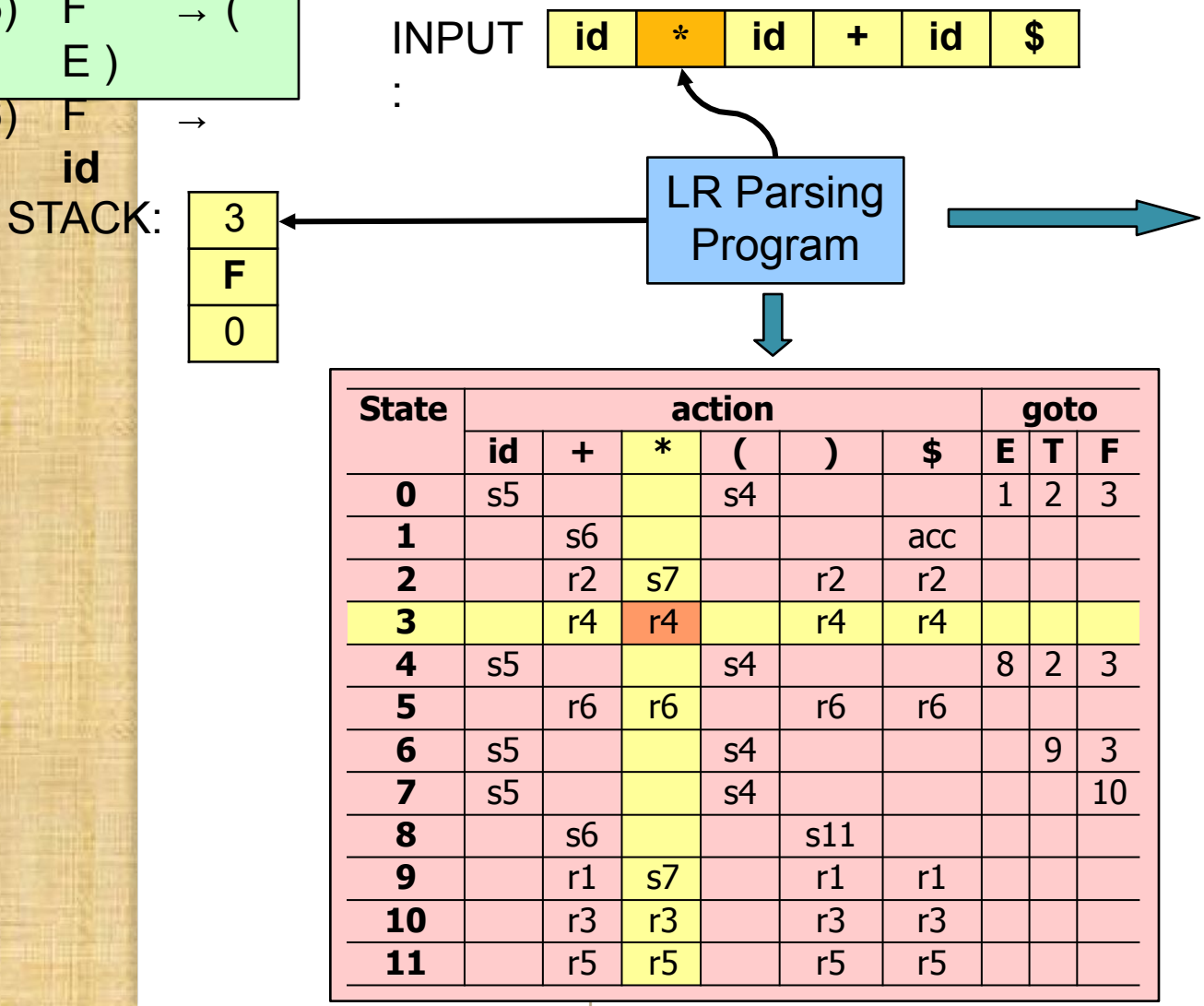
State id	action						goto	
	+	*	(	)	$	E	T	F
0				s5	s4	1	2	3
1		s6				acc		
2		r2			r2	r2		
3		s7			r4	r4		
4	s5	r4		s4			8	2
5		r6			r6	r6		3
6	s5	r6		s4				9
7	s5			s4				
8		s6			s11			3
9		r1			r1	r1		10
10		s7			r3	r3		
11		r5	r5		r5	r5		

(Aho,Sethi,Ullman, pp. 220)

GRAMMAR:

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

LR Parser Example



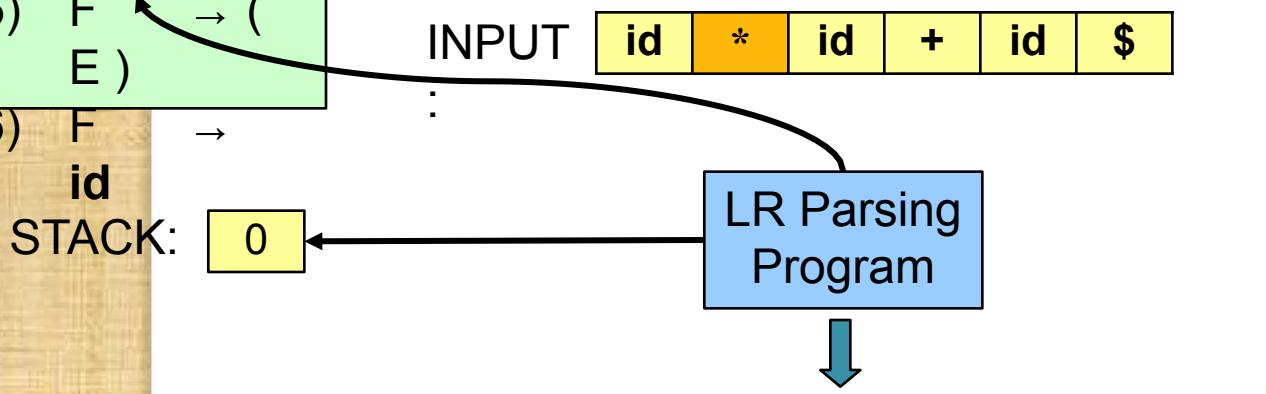
State	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

(Aho,Sethi,Ullman, pp. 220)

LR Parser Example

GRAMMAR:

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



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

OUTPUT

:

T

F

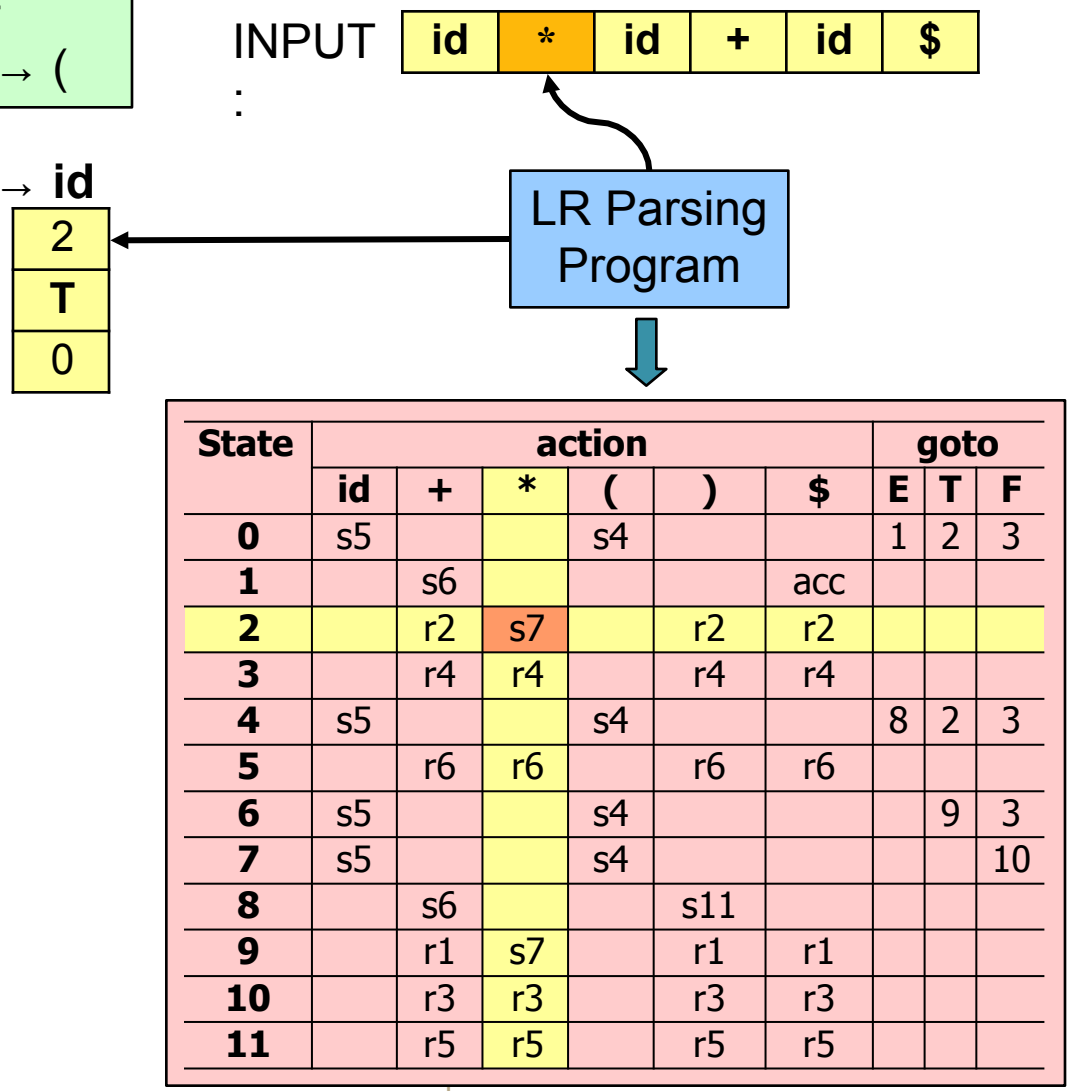
id

(Aho,Sethi,Ullman, pp. 220)

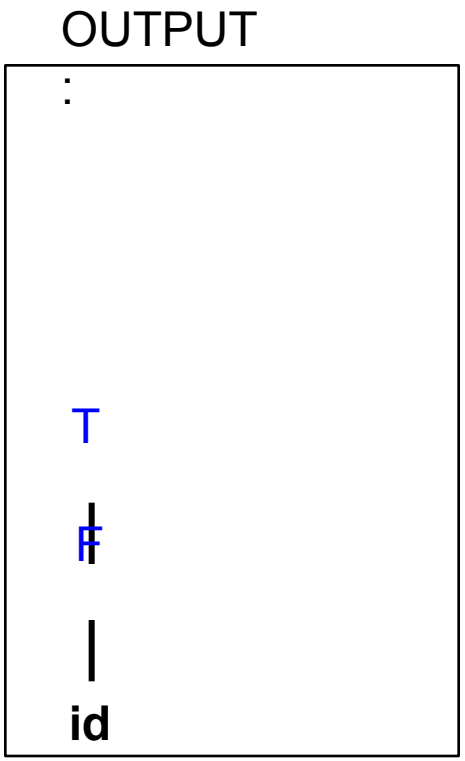
R Parser Example

GRAMMAR:

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



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			

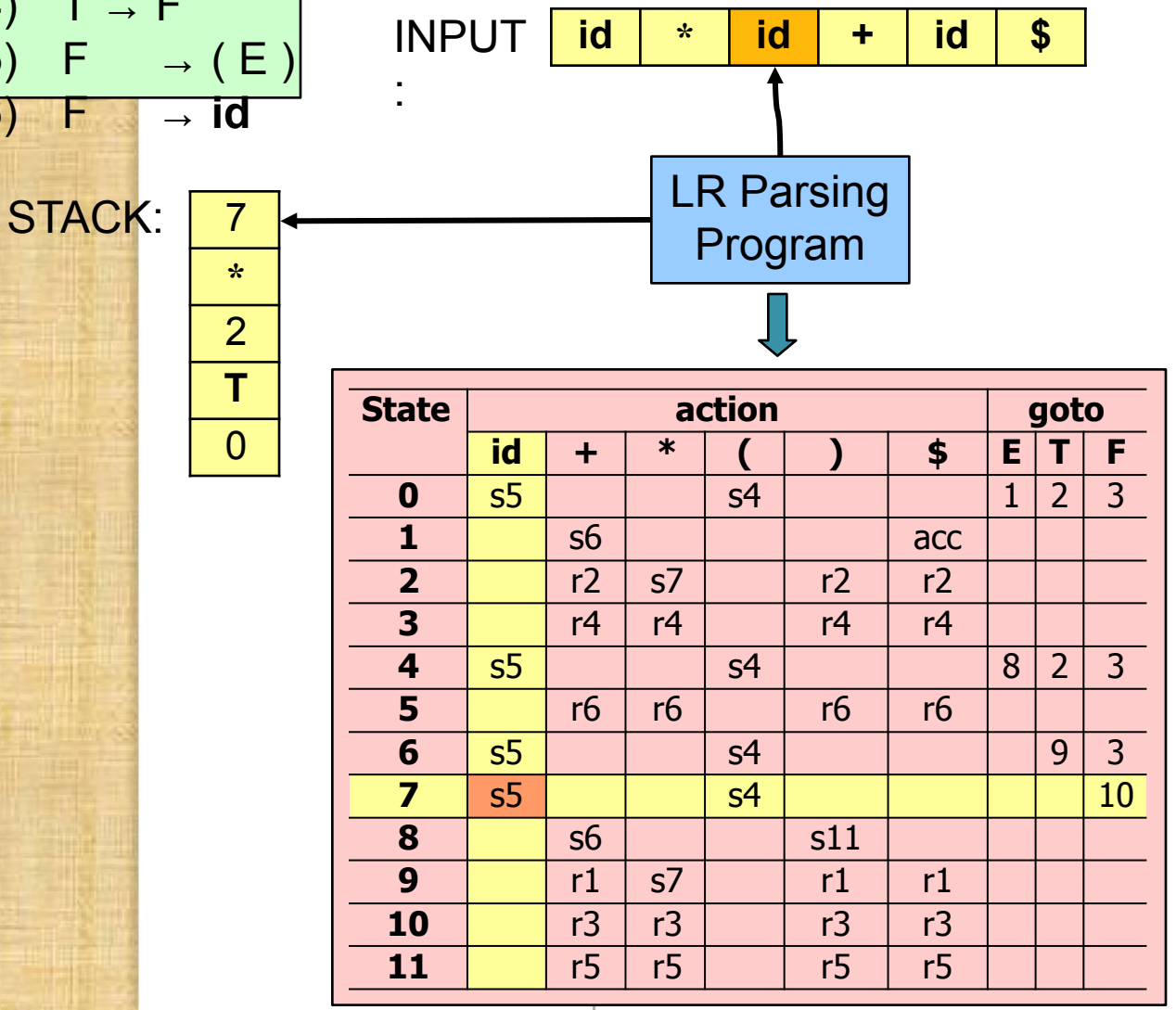


(Aho,Sethi,Ullman, pp. 220)

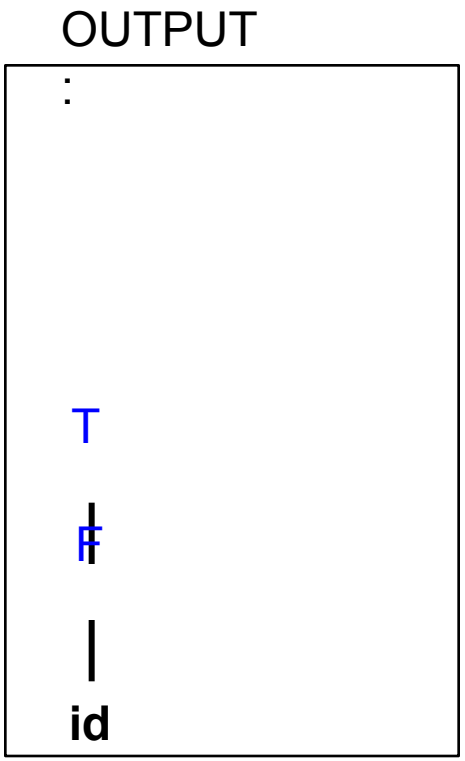
GRAMMAR:

(1) E → E + T
(2) E' → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → **id**

LR Parser Example



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			



(Aho,Sethi,Ullman, pp. 220)

GRAMMAR:

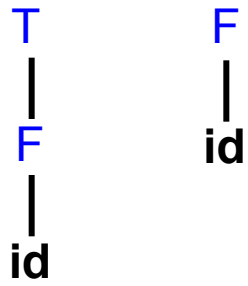
(1) E → E + T
(2) E' → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

LR Parser Example

INPUT :

id	*	id	+	id	$

OUTPUT :



STACK:

5
id
7
*
2
T
0

LR Parsing Program

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			

(Aho,Sethi,Ullman, pp. 220)

GRAMMAR:

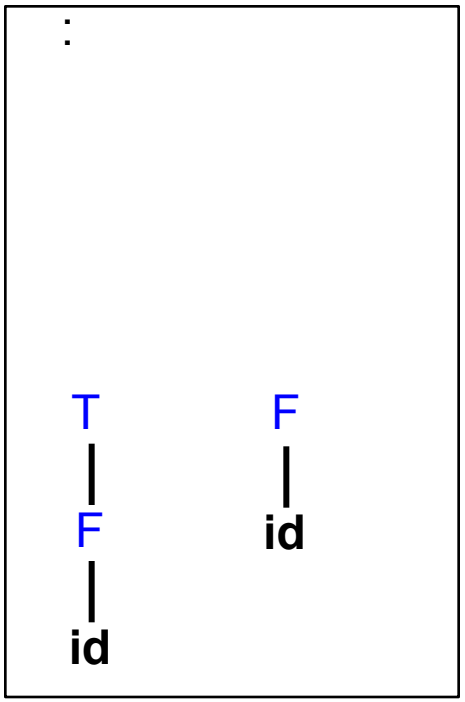
(1) E → E + T
(2) E' → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

LR Parser Example

INPUT :

id	*	id	+	id	$

OUTPUT :



STACK:

7
*
2
T
0

LR Parsing Program

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

(Aho,Sethi,Ullman, pp. 220)

GRAMMAR:

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

LR Parser Example

INPUT :

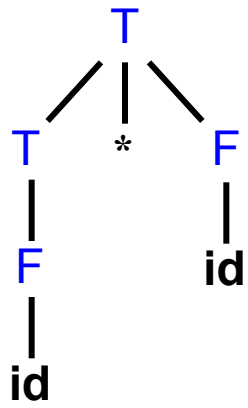
id	*	id	+	id	$

STACK:

10
F
7
*
2
T
0

LR Parsing Program

OUTPUT :



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			

(Aho,Sethi,Ullman, pp. 220)

LR Parser Example

GRAMMAR:

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

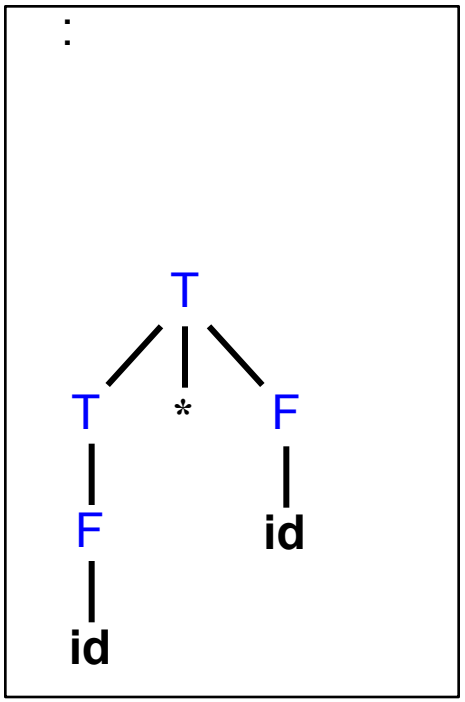
INPUT: id | * | id | + | id | $

STACK: 0

LR Parsing Program

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

OUTPUT:



(Aho,Sethi,Ullman, pp. 220)

GRAMMAR:

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

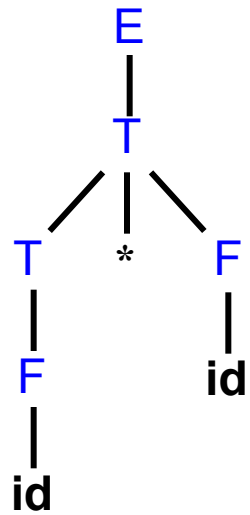
LR Parser Example

INPUT : | id | * | id | + | id | $ |

STACK: | 2 | T | 0 |

LR Parsing Program

OUTPUT :



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			

(Aho,Sethi,Ullman, pp. 220)

LR Parser Example

GRAMMAR:

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

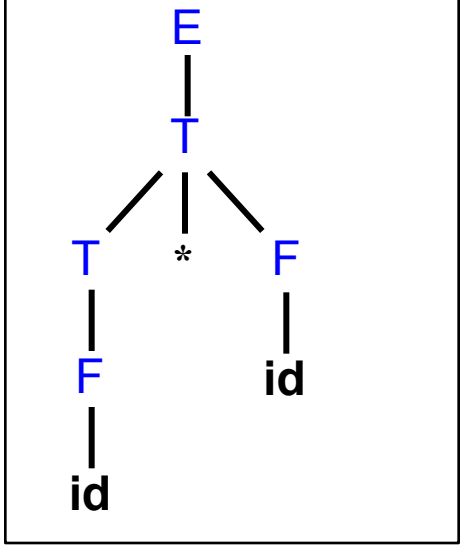
INPUT : | id | * | id | + | id | $ |

STACK: 0

LR Parsing Program

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

OUTPUT :



(Aho,Sethi,Ullman, pp. 220)

GRAMMAR:

(1) E → E + T
(2) E' → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → **id**

LR Parser Example

INPUT :

id	*	id	+	id	$

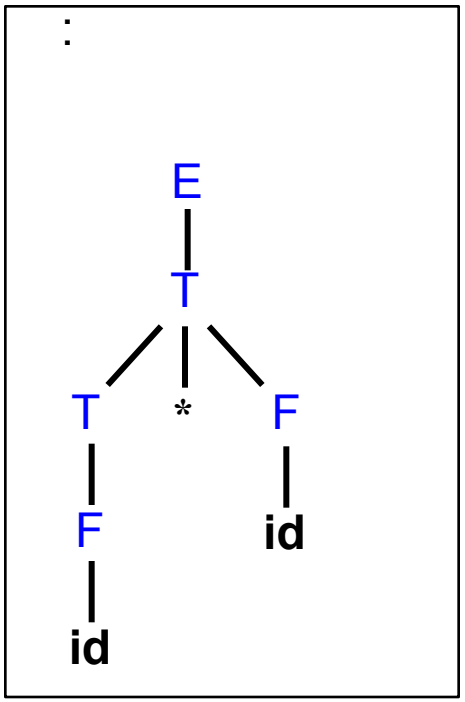
LR Parsing Program

STACK:

1
E
0

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			

OUTPUT :



(Aho,Sethi,Ullman, pp. 220)

R Parser Example

GRAMMAR:

(1) E → E + T
(2) E' → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

INPUT :

id	*	id	+	id	$

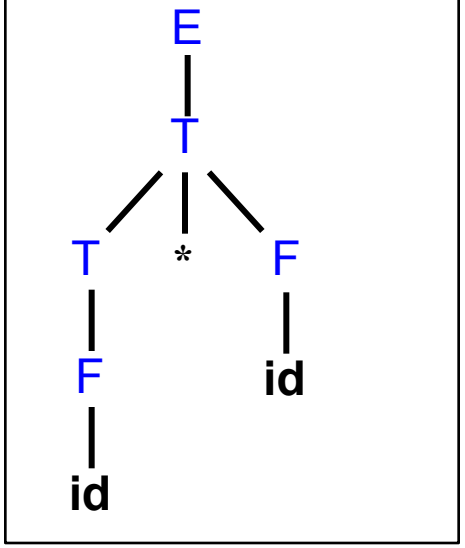
LR Parsing Program

STACK:

6
+
1
E
0

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			

OUTPUT :



(Aho,Sethi,Ullman, pp. 220)

R Parser Example

GRAMMAR:

(1) E → E + T
(2) E' → T
(3) T → T * F
(4) T → F
(6) F → (E)
id

INPUT :

id	*	id	+	id	$

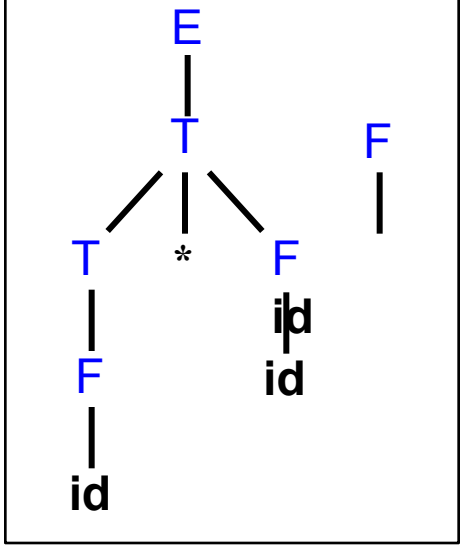
LR Parsing Program

STACK:

5
id
6
+
1
E
0

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			

OUTPUT :



(Aho,Sethi,Ullman, pp. 220)

GRAMMAR:

(1) E → E + T
(2) E' → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

LR Parser Example

INPUT :

id	*	id	+	id	$

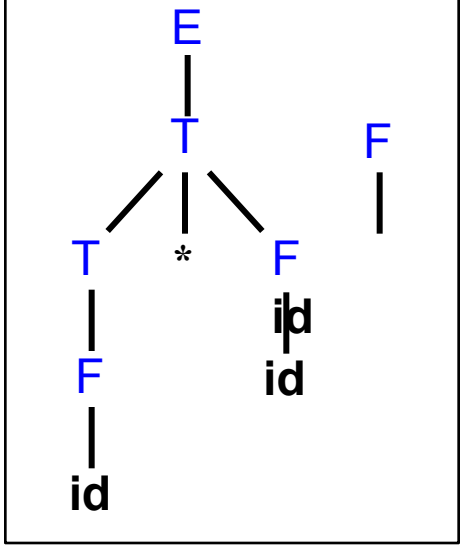
LR Parsing Program

STACK:

6
+
1
E
0

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

OUTPUT :



(Aho,Sethi,Ullman, pp. 220)

LR Parser Example

GRAMMAR:

(1) E → E + T
(2) E' → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

INPUT :

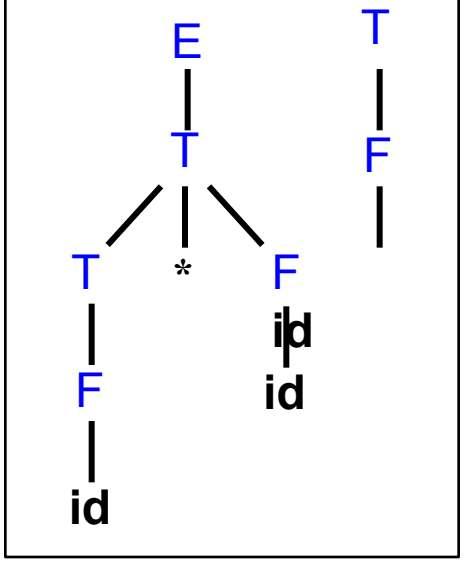
id	*	id	+	id	$

STACK:

3
F
6
+
1
E
0

LR Parsing Program

OUTPUT :



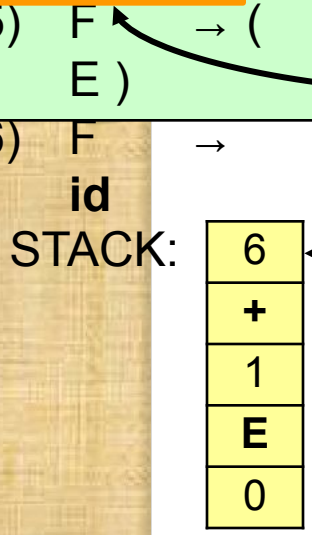
State	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

(Aho,Sethi,Ullman, pp. 220)

R Parser Example

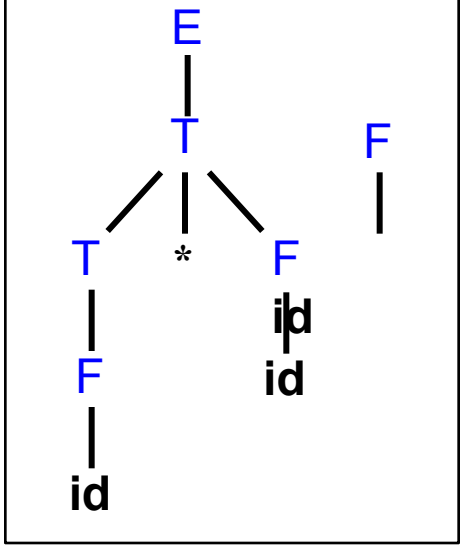
GRAMMAR:

(1) E → E + T
(2) E' → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id



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

OUTPUT
:



(Aho,Sethi,Ullman, pp. 220)

GRAMMAR:

(1) E → E +
(2) E' → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → **id**

LR Parser Example

INPUT :

id	*	id	+	id	$

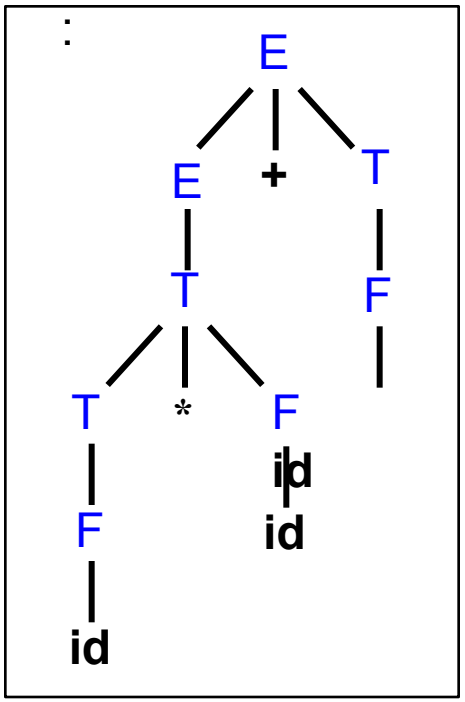
LR Parsing Program

STACK:

9
T
6
+
1
E
0

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			

OUTPUT :



(Aho,Sethi,Ullman, pp. 220)

GRAMMAR:

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

LR Parser Example

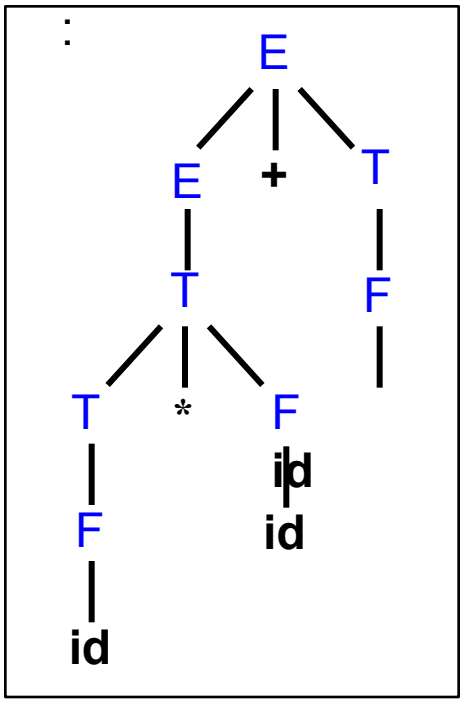
INPUT: | id | * | id | + | id | $ |

STACK: 0

LR Parsing Program

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

OUTPUT:



(Aho,Sethi,Ullman, pp. 220)

R Parser Example

GRAMMAR:

(1) E → E + T
(2) E' → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

INPUT :

id	*	id	+	id	$

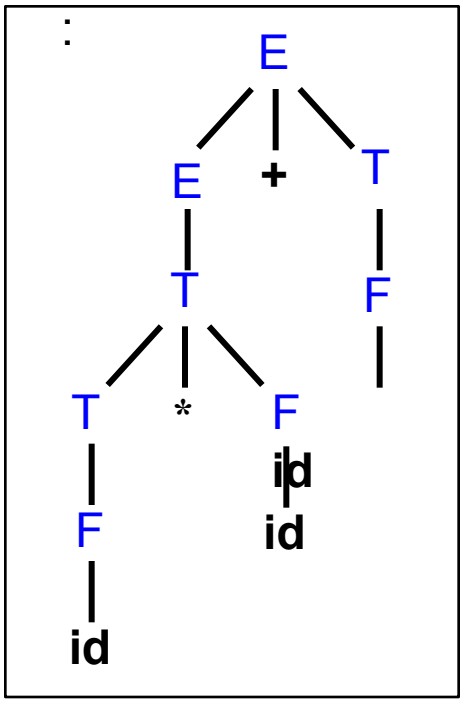
STACK:

1
E
0

LR Parsing Program

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			

OUTPUT :



(Aho,Sethi,Ullman, pp. 220)

Constructing Parsing Tables

All LR parsers use the same parsing program that we demonstrated in the previous slides.

What differentiates the LR parsers are the action and the goto tables:

Simple LR (SLR): succeds for the fewest grammars, but is the easiest to implement. (See AhoSethiUllman pp. 221-230).

Canonical LR: succeds for the most grammars, but is the hardest to implement. It splits states when necessary to prevent wrong reductions that would get the parser stuck. (See AhoSethiUllman pp. 230-236).

Lookahead LR (LALR): succeds for most common syntatic constructions used in programming languages, but produces LR tables much smaller than canonical LR.

(See AhoSethiUllman pp. 236-247).

(Aho,Sethi,Ullman, pp. 221)

Closure()

- I set of items of G
- Closure(I)

  Initially every item of I is included in Closure(I)

- Repeat

  If A->α.Bβ in closure(I) and B-> γ is a production, add B->.γ (If it is not already present) to closure, Until no new items can be added to closure (I)

goto()

- Goto(I,X), I set of items, X grammar symbol
  - Goto(I,X) := closure({A->αX.β| A->α.Xβ ∈ I}) For valid items I for viable prefix γ ,

then goto(I,X) = valid items for viable prefix γX

- Kernel Items:Which includes the initial item S'□.S, and all items whose dots are not at the Left End.
- Nonkernel Items:Which have their dots at the Left End.

Set of Items Construction

```
procedure items(G');
begin
C := closure({S'->.S});
repeat
for each set of items I in C and each
grammar symbol X such that goto(I,X) is not
empty and not in C do
add goto(I,X) to C
until no more items can be added to C
end
```

- E’ -> E
- E -> E + T | T
- T -> T * F | F
- F -> (E) | id

I0 =

I1 =

I2 =

I3 =

I4 =

I5 =

I6 =

I7 =

I8 =

I9 =

I10 =

I11 =

LR(0) Items

I0 = E' -> .E
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

LR(0) Items

I0 = E' -> .E
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I1 = E' -> E.
E -> E. + T

LR(0) Items

I0 = E' -> .E
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I1 = E' -> E.
E -> E. + T

I2 = E□T.
T □T.* F

LR(0) Items

I0 = E' -> .E
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I1 = E' -> E.
E -> E. + T

I2 = E ->T.
T -> T. * F

I3 = T -> F.

LR(0) Items

I0 = E' -> .E
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I1 = E' -> E.
E -> E. + T

I2 = E -> T.
T -> T. * F

I3 = T -> F.

I4 = F -> (.E)
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

LR(0) Items

I0 = E' -> .E
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I1 = E' -> E.
E -> E. + T

I2 = E -> T.
T -> T. * F

I3 = T -> F.

I4 = F -> (.E)
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I5 = F -> id.

LR(0) Items

I0 = E' -> .E
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I1 = E' -> E.
E -> E. + T

I2 = E -> T.
T -> T. * F

I3 = T -> F.

I4 = F -> (.E)
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I5 = F -> id.

I6 =E -> E +. T
T -> .T * F
T -> .F
F -> .(E)
F -> id

LR(0) Items

I0 = E' -> .E
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I1 = E' -> E.
E -> E. + T

I2 = E -> T.
T ->T. * F

I3 = T -> F.

I4 = F -> (.E)
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I5 = F -> id.

I6 =E -> E +. T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I7=T -> T *. F
F -> .(E)
F -> id

LR(0) Items

I0 = E' -> .E
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I1 = E' -> E.
E -> E. + T

I2 = E ->
T. T -> T.
* F

I3 = T -> F.

I4 = F -> (.E)
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I5 = F -> id.

I6 =E -> E +. T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I7 =T -> T *. F
F -> .(E)
F -> id

I8 =F -> (E.)
E-> E.+T

LR(0) Items

I0 = E' -> .E
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I1 = E' -> E.
E -> E. + T

I2 = E ->
T. T -> T.
* F

I3 = T -> F.

I4 = F -> (.E)
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I5 = F -> id.

I6 =E -> E +. T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I7 =T -> T *. F
F -> .(E)
F -> id

I8 =F -> (E.)
E-> E.+T

I9 =E -> E
+T. T -> T. * F

LR(0) Items

I0 = E' -> .E
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I1 = E' -> E.
E -> E. + T

I2 = E ->
T. T -> T.
* F

I3 = T -> F.

I4 = F -> (.E)
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I5 = F -> id.

I6 =E -> E +. T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I7 =T -> T *. F
F -> .(E)
F -> id

I8 =F -> (E.)
E-> E.+T

I9 =E -> E
+T. T -> T. * F

I10 = T -> T*F.

LR(0) Items

I0 = E' -> .E
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I1 = E' -> E.
E -> E. + T

I2 = E ->
T. T -> T.
* F

I3 = T -> F.

I4 = F -> (.E)
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I5 = F -> id.

I6 =E -> E +. T
T -> .T * F
T -> .F
F -> .(E)
F -> id

I7 =T -> T *. F
F -> .(E)
F -> id

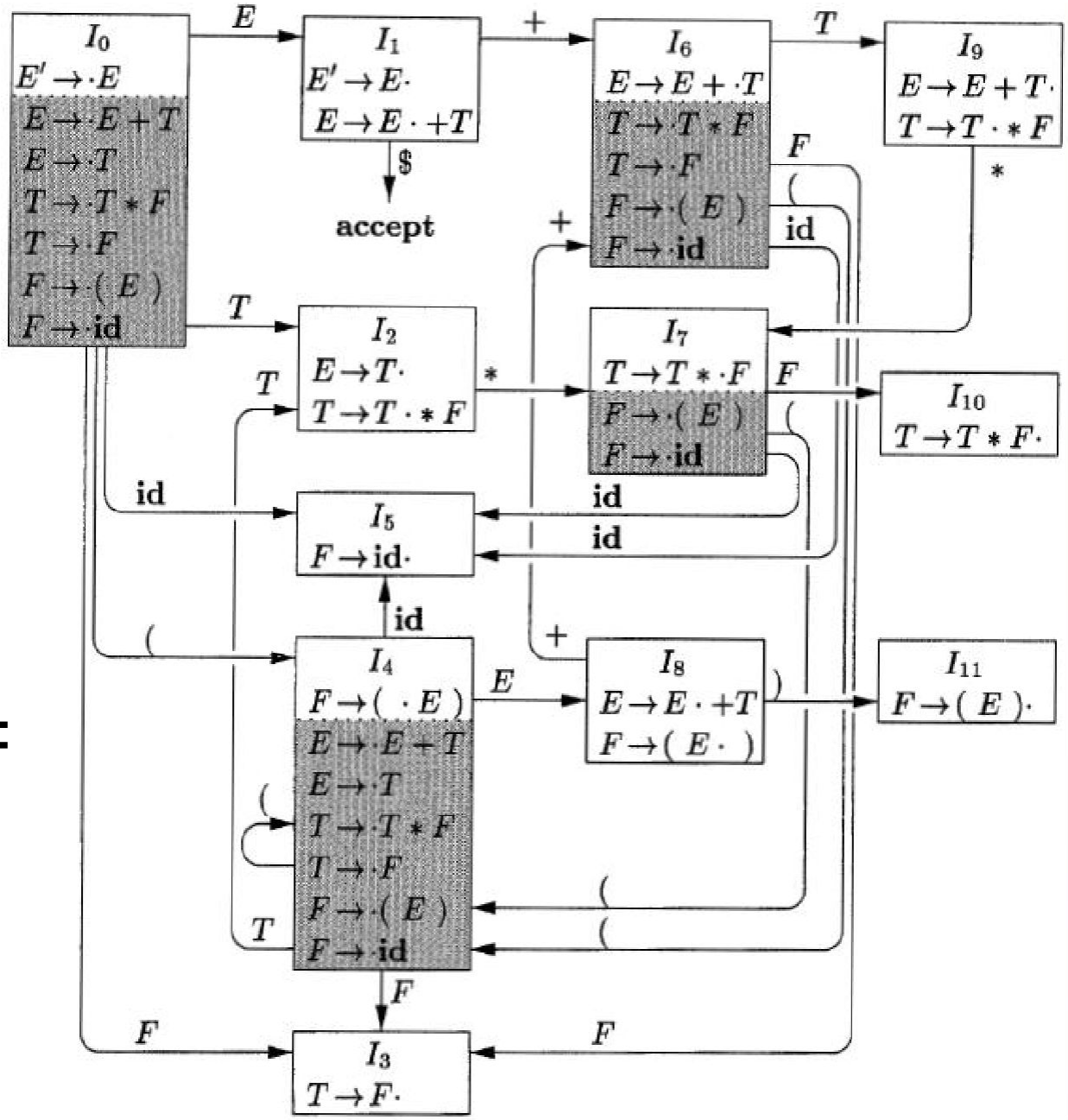
I8 =F -> (E.)
E-> E.+T

I9 =E -> E
+T. T -> T. * F

I10 = T -> T*F.

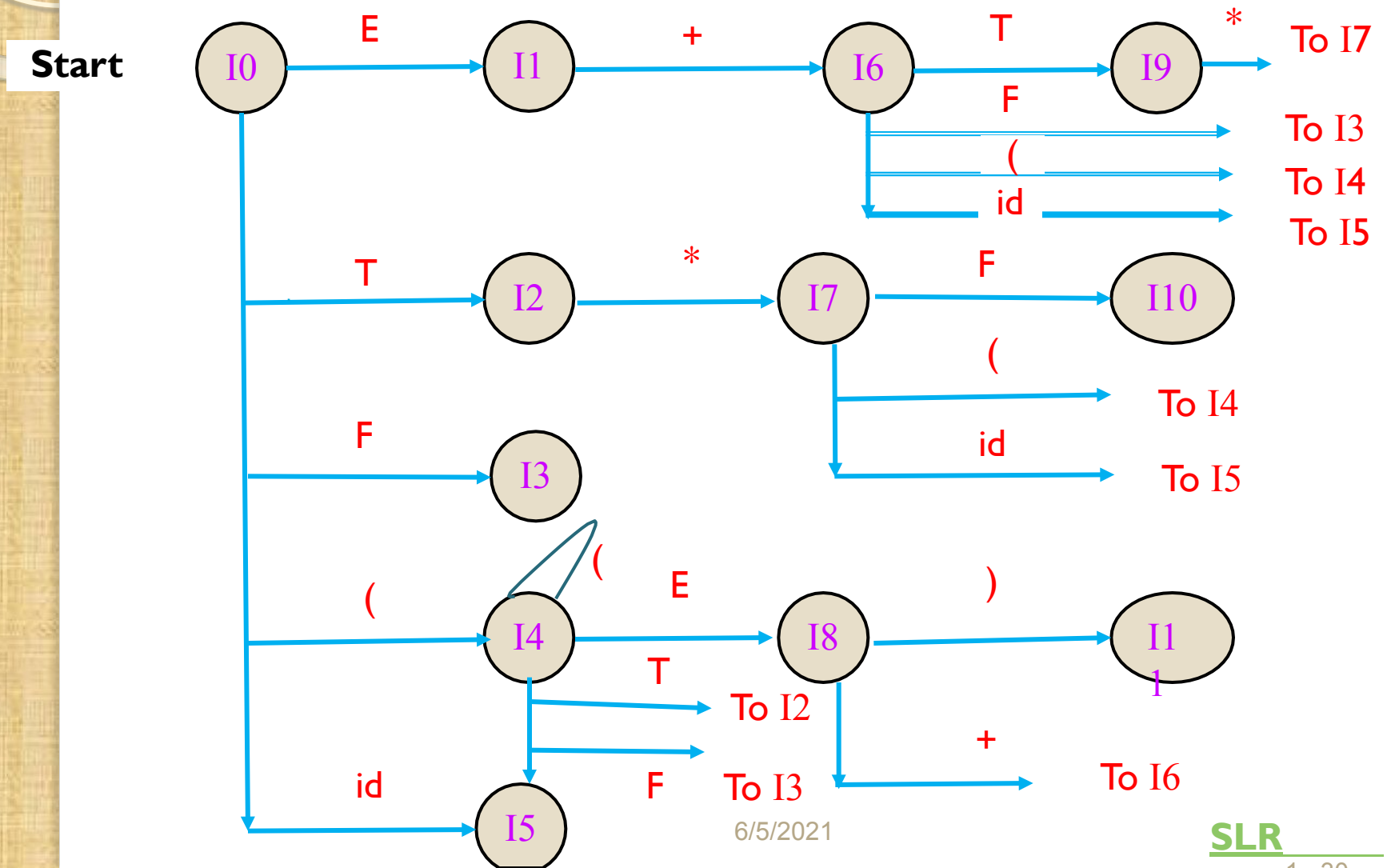
I11 =F -> (E).

- E’ -> E
- E -> E + T | T
- T -> T * F | F
- F -> (E) | id



DFA…

Each State is Final State



- Input: An Augmented Grammar G'
- Output: The SLR Parsing Table function action and goto for G'.
- Method:

1. Construct C={I0,I1…..In}, the collection of sets of LR(0) items for G'.
2. State **i** is constructed from **Ii** .The parsing actions for the state **i** are determined as follows:

   a] If [A->α.aβ] is in **Ii** and goto(**Ii**, a) = Ij, then **set action[i, a] to "shift j".** Here **a** must be terminal.

b] If [A->α.] is in **Ii,** then **set action[i, a] to "reduce A->α "** for all **'a'** in FOLLOW(A) ; here A may not be S'.

c] If [S'->S.] is in **Ii,** then **set action[i, $] to "Accept" .**

3. The goto transitions for state i are constructed for all nonterminals A using rule : If **goto(Ii, A) = Ij, then goto[i, A] =j.**
4. All entries not defined by rules (2) and (3) are made " error".
5. The initial state of the parser is the one constructed from the set of items [S'->.S]

Continue…

b] If [A->α.] is in **Ii,** then **set action[i, a] to "reduce A->α "** for all a in FOLLOW(A) ; here A may not be S'.

c] If [S'->S.] is in **Ii,** then **set action[i, $] to "Accept".**

3.The goto transitions for state i are constructed for all nonterminals A using rule :

If go

4.Al
are

5.T
he initial state of the parser is the one constructed from the set of items containing [S'->.S]

If any conflicting actions are generated by the above rules (a,b,c), we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

Constructing Canonical LR Parsing Table

- The extra information is incorporated into the state by redefining items which includes a terminal symbol as a Second Component. (*is lookahead of the item*)
- [A->α.β, **a**] where A->αβ is prod.

  a is terminal or $

Second Component.

- We call such an object as an LR(1) item.

Closure(I)

- Begin
  - Repeat

for each item [A->α.Bβ, a] in (I)

each production B-> γ is in G',

and each terminal b in FIRST(βa)

such that [B->.γ, b] is not in I (If it is

not already present) do add [B->.γ, b] to I;

Until no new items can be added to closure (I)

return I

goto(I, X)

- begin

  Let J be the set of items [A->αX.β, a] such that [A->α.Xβ, a is in I;

return closure(J)

end;

Procedure items(G');

begin

C = {closure({[S'□.S, $]})};

repeat

for each set of items I in C and each grammar symbol X,

such that goto(I, X) is not empty and not in C do

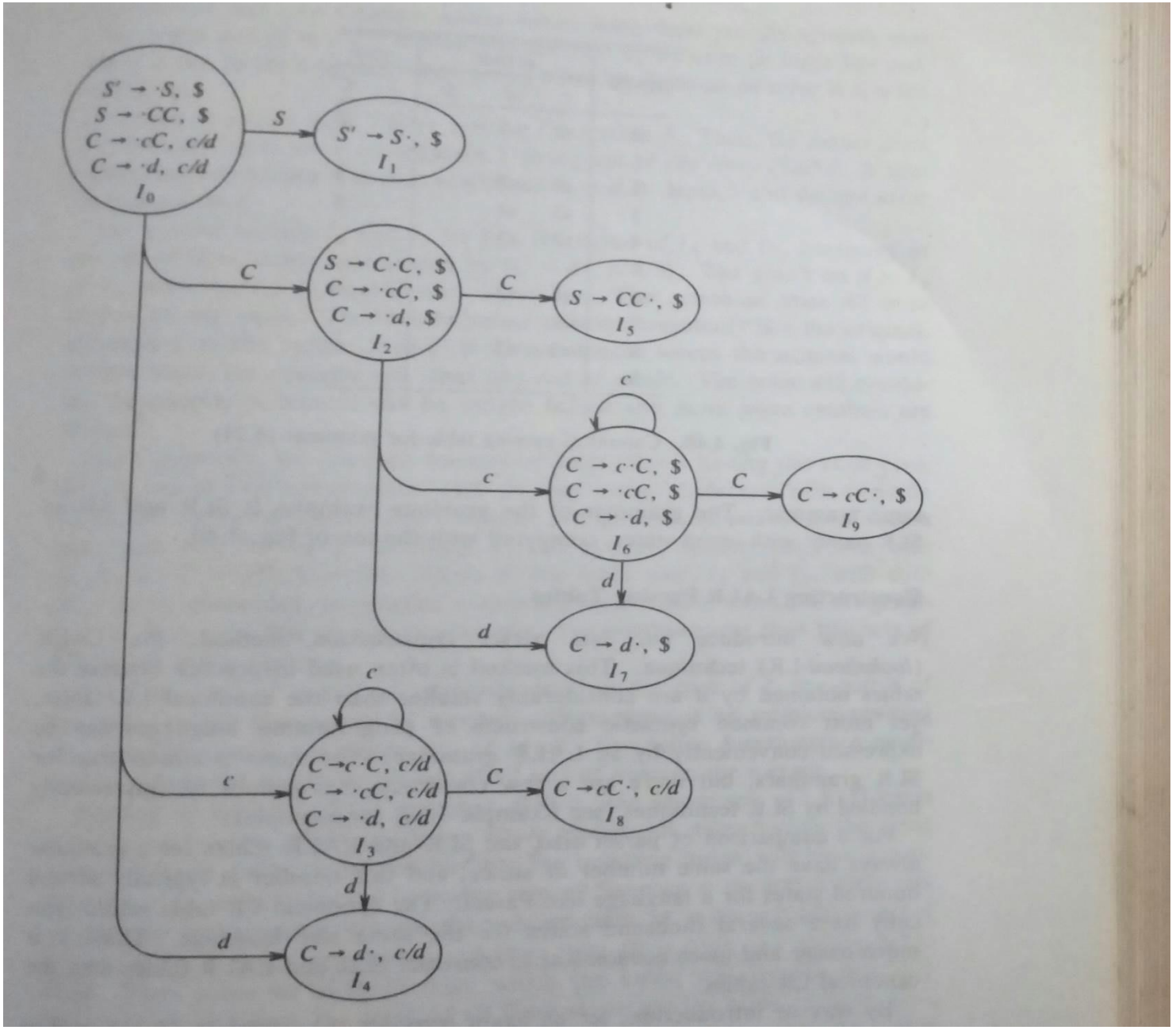
add goto(I, X) to C.

Until more items can be added to C

end.

MIT-WPU

Example....



- Input: An Augmented Grammar G'
- Output: The canonical LR Parsing Table function action and goto for G'.
- Method:

1. Construct C={I0,I1…..In}, the collection of sets of LR(1) items for G'.
2. State **i** is constructed from **Ii** .The parsing actions for the state **i** are determined as follows:

   a] If [A->α.aβ, b] is in **Ii** and goto(**Ii**, a) = Ij, then **set action[i, a] to "shift j".** Here **a** must be terminal.

b] If [A->α., a] is in **Ii,** then **set action[i, a] to "reduce A->α " ,** here A may not be S'.

c] If [S'->S., $] is in **Ii,** then **set action[i, $] to "Accept" .**

3. The goto transitions for state i are constructed for all nonterminals A using rule : If **goto(Ii, A) = Ij, then goto[i, A] =j.**
4. All entries not defined by rules (2) and (3) are made " error".
5. The initial state of the parser is the one constructed from the set of items containing [S'->.S, $].

Continue…

c] If [S'->S. , $] is in Ii, then set action[i, a] to "Accept" .

If any conflicting actions are generated by the above rules (a,b,c), we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.

State	Action			Go to	
	c	d	$	S	C
0	S3	S4		1	2
1			acc		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

Constructing Lookahead-LR (LALR) Parsing Table

- The table obtained are smaller than Canonical .
- Also It can handle some constructs that cannot be handled by SLR Grammar.
- We look for sets of *LR(1) items having the same core, that is, set of first components and we may merge these sets with common cores into one set of items.*

- Input: An Augmented Grammar G'
- Output: The LALR Parsing Table functions action and goto for G'.
- Method:

1. Construct C={I0,I1…..In}, the collection of sets of LR(1) items.
2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.
3. Let C' = { J1, J2,… Jm } be the resulting sets of LR(1) items. The parsing actions for state i are constructed from Ji in the same manner as in algorithm(CLR).

If there is a parsing action conflict., the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

Continue…

4. The goto table is constructed as follows.

If J is union of one or more sets of LR(1) items , that is $J = I1 \cup I2 \cup \ldots \cup Ik$, then the cores of goto(I1, X) , goto(I2, X) …. goto(Ik, X) are the same, since I1, I2, …. Ik all have the same core. Let K be the union of all sets of items having the same core as goto(I1, X).

Then goto(J, X) = K.

If there are no parsing action conflicts, then the given grammar is said to be an LALR(1) grammar

MIT-WPU

LALR parsing table for grammar

State	Action			Go to	
	c	d	$	S	C
0	S36	S47		1	2
1			acc		
2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

- Compare

- SLR Vs CLR Vs LALR

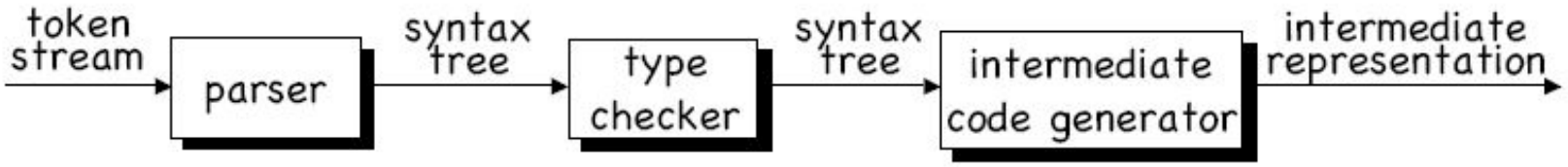
Semantic Analysis

- Here Compiler tries to discover the meaning of a program by analyzing its Parse Tree or Abstract Syntax Tree.
- Checks whether the SP is according to Syntactic and Semantic Conventions of Source Lang or not.
- Also known as Context Sensitive Analysis.
- Answer depends on Value not Syntax.

Attribute Grammar

- Attributes on Symbols
- Attribute Evaluation Rules
- Indexing of Grammar Symbols

Type Checking



- TYPE CHECKING is the main activity in semantic analysis.
- Goal: calculate and ensure consistency of the type of every expression in a program
- If there are type errors, we need to notify the user.
- Otherwise, we need the type information to generate code that is correct.

MIT-WPU
Final Year(B.Tech)

System Software and Compiler Design

Unit III

- **Introduction to compilers:** passes, phases, symbol table.
- **Lexical Analyzer:** Role of LEX Analyzer, Specification of tokens, Recognition of tokens, input buffering.
- **Syntax Analysis:** RDP, Predictive parser, SLR, LR (1), LALR parsers, using ambiguous grammar, Error detection and recovery.
- **LEX and YACC:** Specification and generation using LEX tool, Lexical errors. Automatic construction of parsers using YACC

Introduction to Compilers

- A compiler is a program that can read a program in one language – the *source* language – and translate it into an equivalent program in another language – the *target* language.

Passes

- A pass is a complete traversal of the source program, or a complete traversal of some internal representation of the source program.
- Sometimes a single “pass” corresponds to several phases that are interleaved in time.
- What and how many passes a compiler does over the source program is an important design decision.

Passes

- In an implementation of compiler ,portions of one or more phases are combined into a module called a **pass.**
- A **pass** in compiler design is the group of several phases of compiler to perform analysis or synthesis of source program.
- Two types of pass:-
  1:-one pass
  2:-two pass
- In one pass structure:

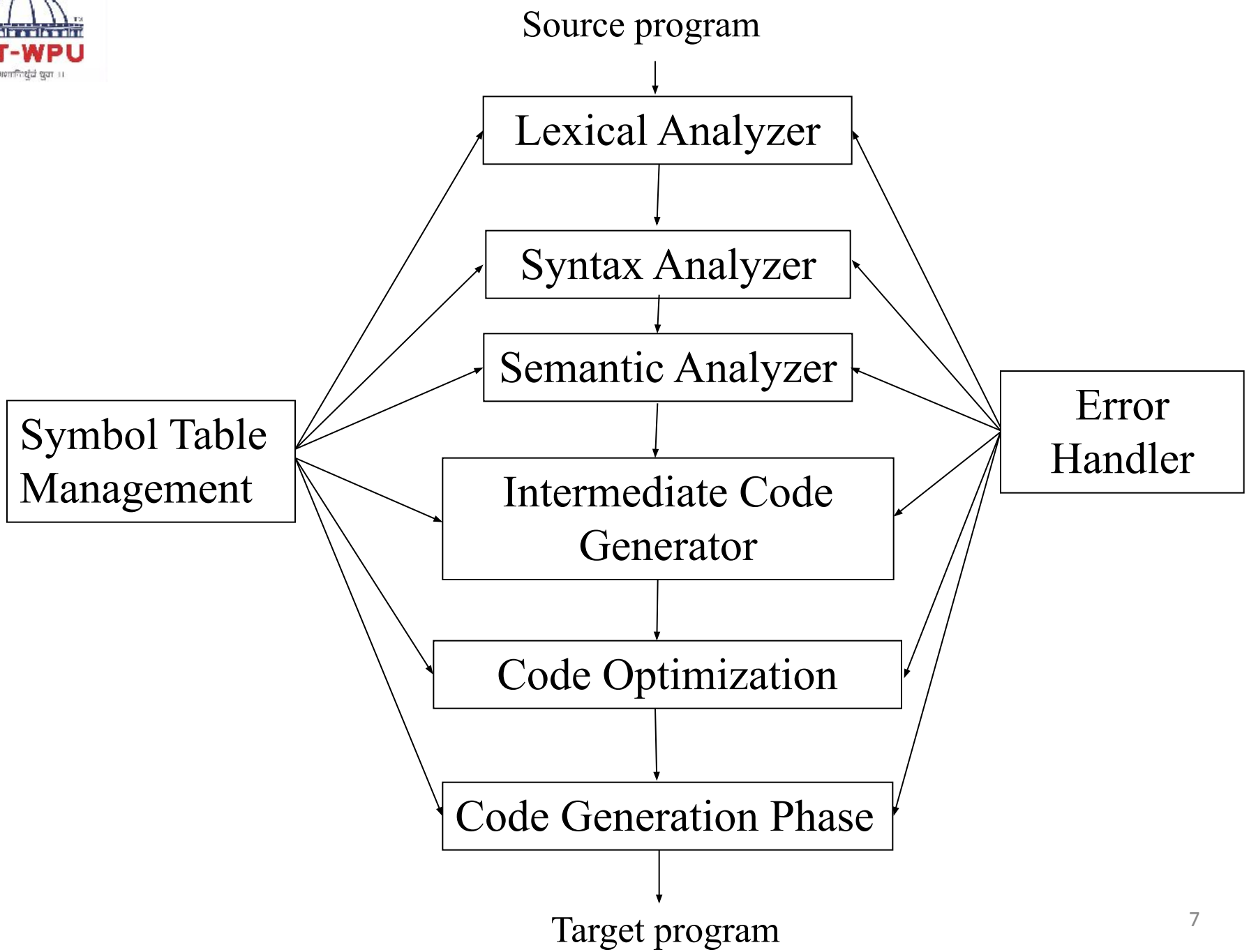
  both analysis and synthesis of source program is done in the flow from beginning to end of program.
- In two pass structure:

  analysis of source program is done in first pass

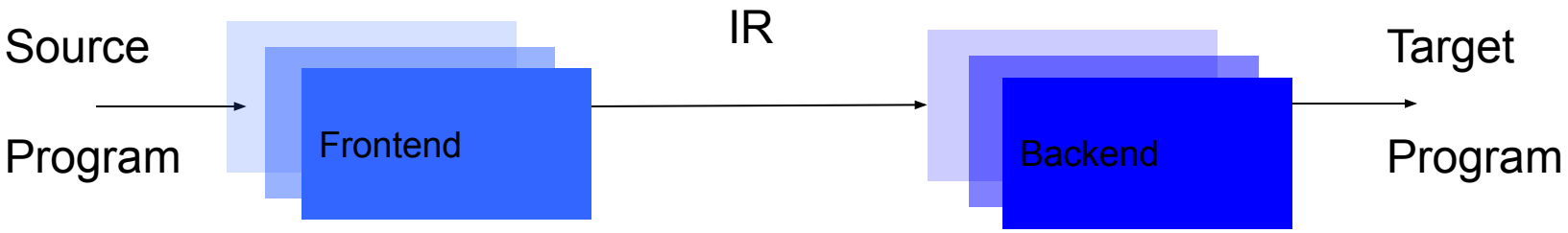
  synthesis of source program is done in second pass

Phases of a Compiler

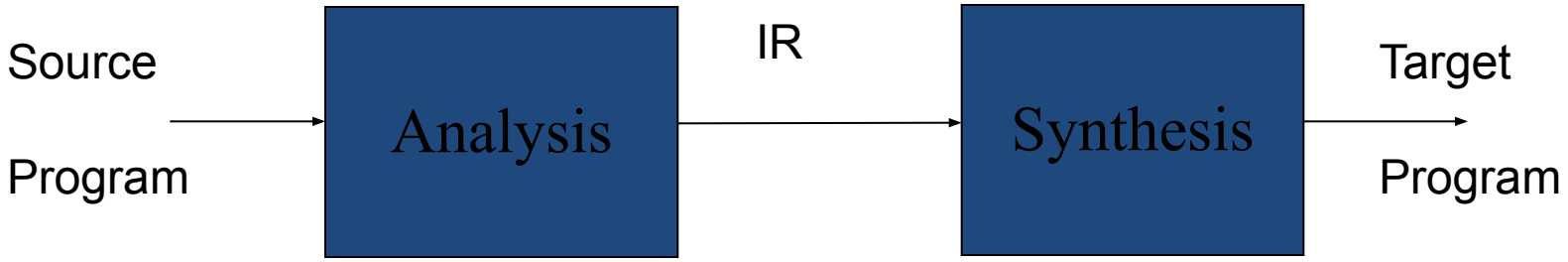
- Lexical analysis
- Syntax analysis
- Semantic analysis
- Intermediate (machine-independent) code generation
- code optimization
- Target (machine-dependent) code generation


Source program
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Code Optimization
Code Generation Phase
Target program
Symbol Table Management
Error Handler

Front End and Back End Model of Compiler



- Analysis and Synthesis Phase of Compiler



Symbol table

- **Symbol table**
- It is an important data structure created and maintained by compilers.
- It is used by compiler to keep track of scope/binding information about names.
- These names are used in the source program to identify various program elements like variables names, function names, objects, classes, interfaces, etc.
- Symbol table is used by both the analysis and the synthesis parts of a compiler.

Symbol table

- A symbol table purposes are :

1. To store the names of all entities in a structured form at one place.
2. To verify if a variable has been declared.
3. To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
4. To determine the scope of a name (scope resolution).
5. A symbol table is simply a table which can be either linear or a hash table.

- It maintains an entry for each name in the following format:

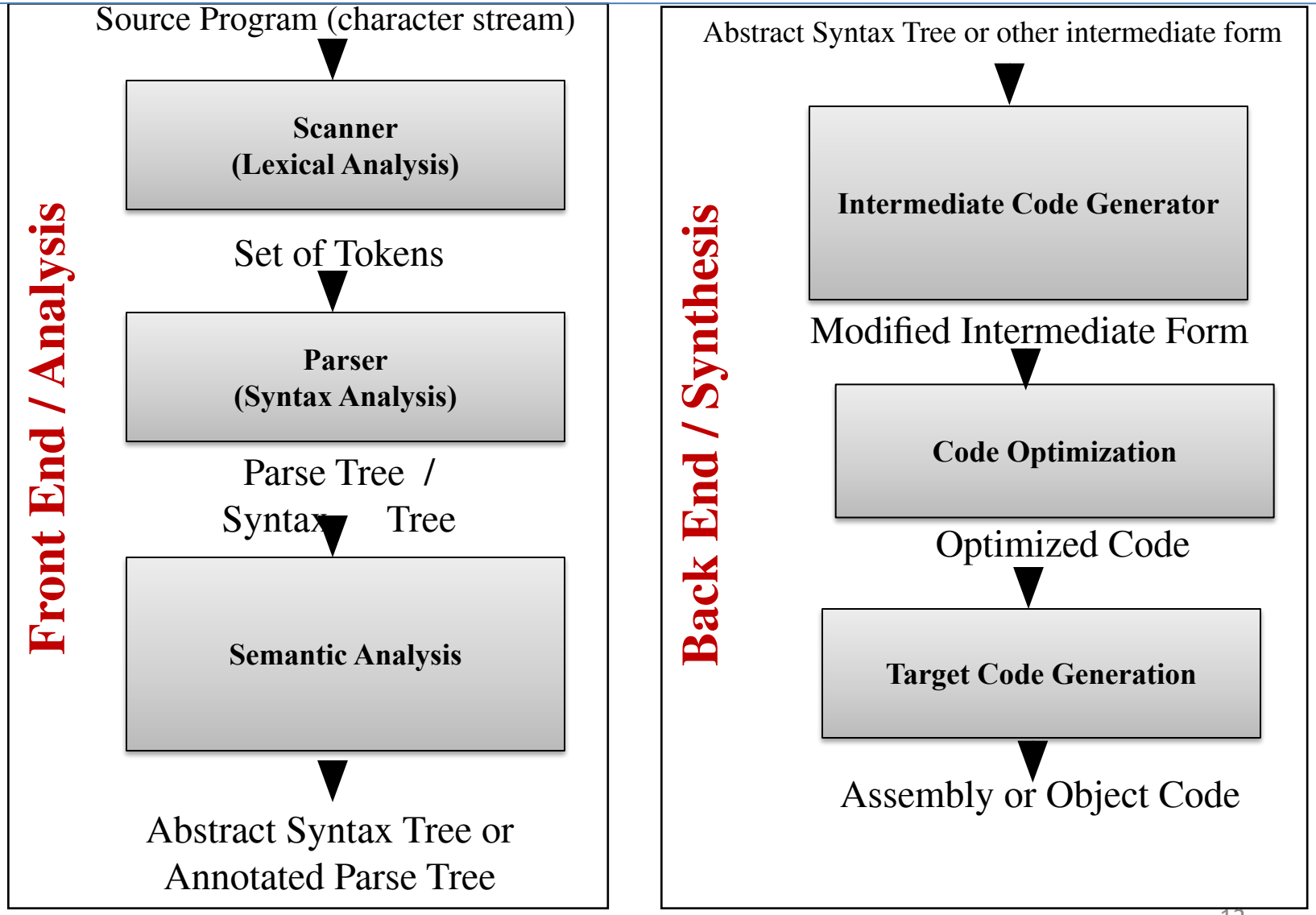
  <symbol name,type,attribute>

Symbol tables

- Data structures used for symbol table:

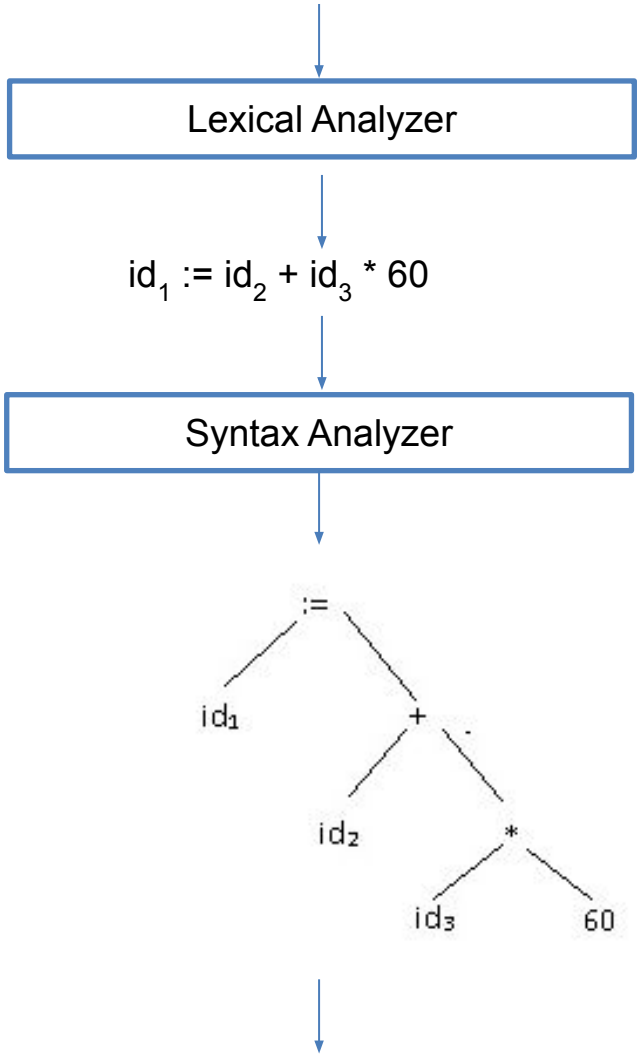
1. List
2. Linked list
3. Binary trees
4. Hash tables

Compiler Front End – Back End / Analysis –Synthesis Phase



Assignment Statement Translation

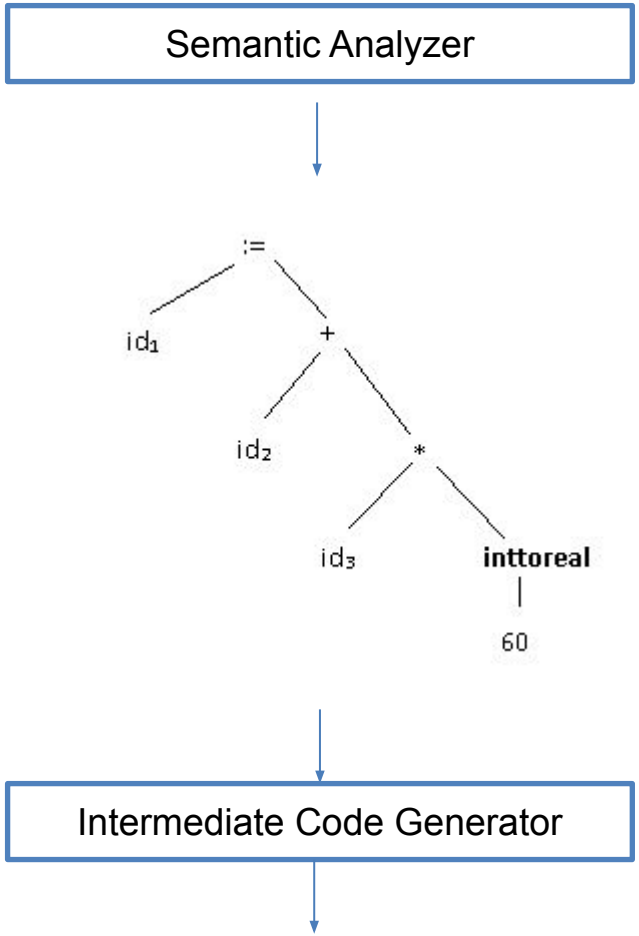
position := initial + rate * 60



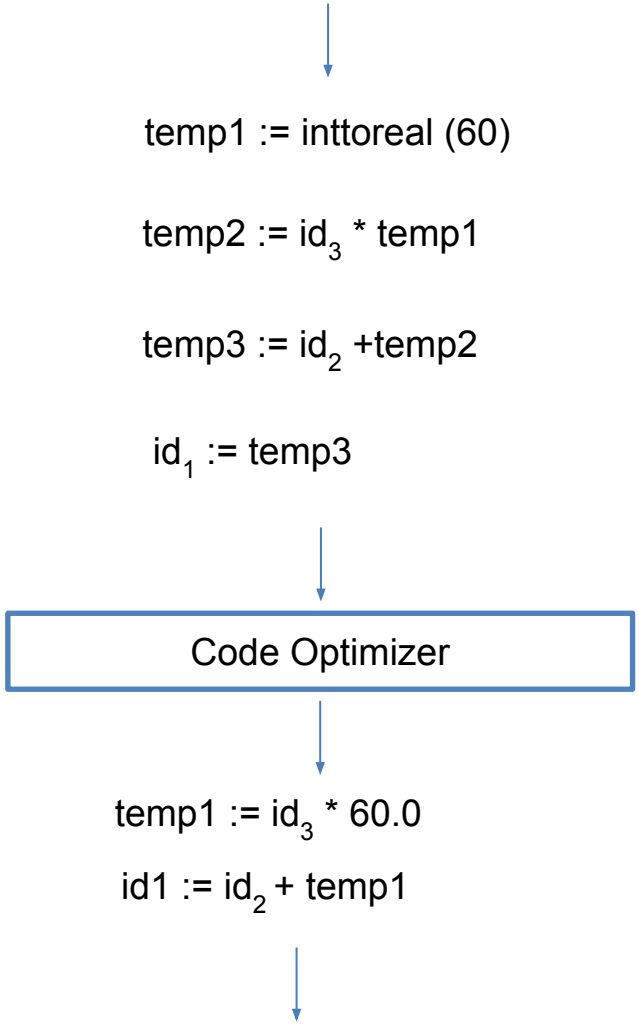
Symbol Table

1	position	…
2	initial	…
3	rate	…
4		

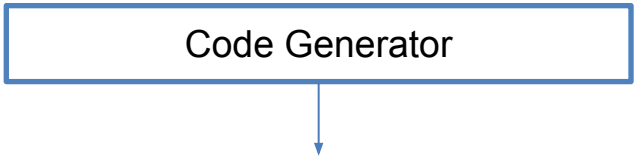
Assignment Statement Translation



Assignment Statement Translation



Assignment Statement Translation



MOVF id_3, R2

MULF #60.0, R2

MOVF id_2, R1

ADDF R2, R1

MOVF R1, id_1

Regular Expression

- Rules

1. Є is a RE that denotes {Є }.

Regular Expression

- Rules

1. Є is a RE that denotes {Є }.
2. If ‘a’ is a symbol in ∑ RE is {a}

Regular Expression

- Rules

1. Є is a RE that denotes {Є }.
2. If ‘a’ is a symbol in ∑ RE is {a}
3. Suppose r & s are RE s denoting the languages L(r) & L(s) then

a.(r) | (s) is a RE denoting L(r) U L(s)

b. (r)(s) is a RE denoting L(r) . L(s)

c. $(r)^*$ is a RE denoting $(L(r))^*$

d. . (r) is a RE denoting L(r)

Contd…

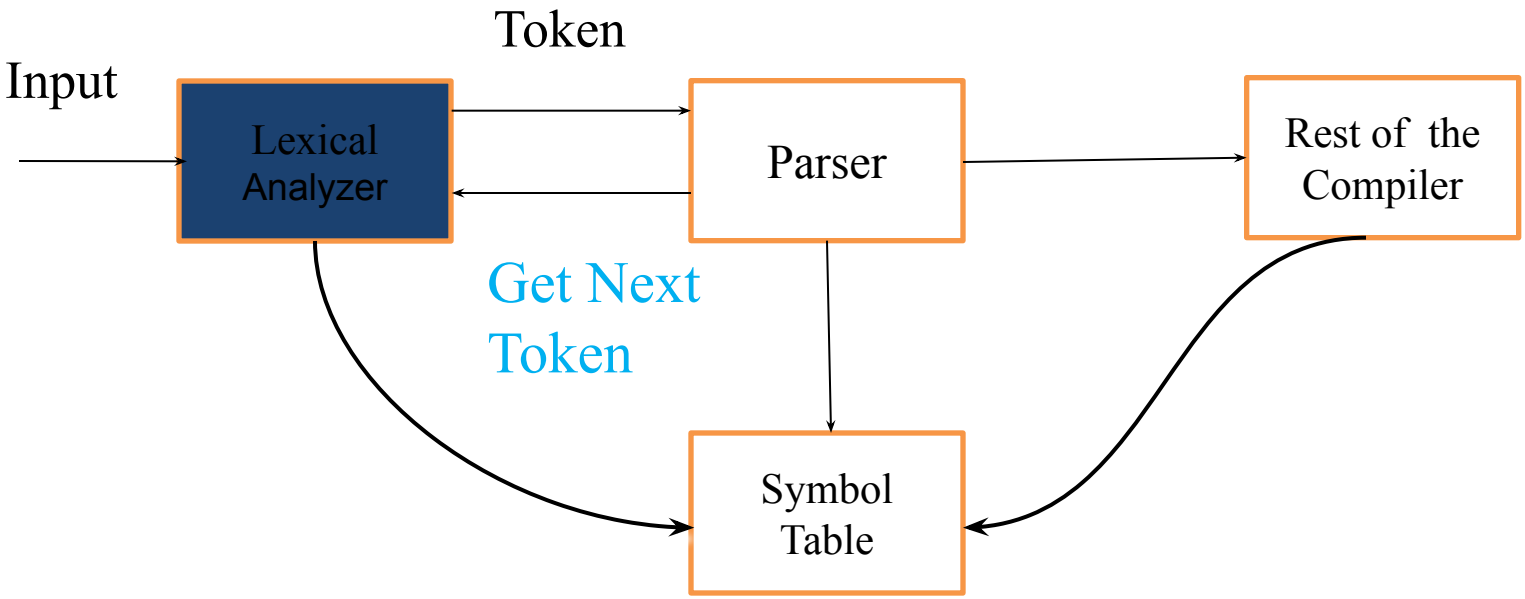
Axioms

- $r|s = s|r$
- $r|(s|t) = (r|s)|t$
- $(rs)t = r(st)$
- $r|s|t = rs|rt$
- $(s|t)r = sr|tr$
- $\epsilon r = r$
- $r\epsilon = r$
- $r^* = (r|\epsilon)^*$
- $r^{**} = r^*$

Description

- | is commutative
- | is associaltive
- Concat is associative
- Concat is distributes over
- same as above
- ϵ is the identity element For concatenation
- Relation between * and ϵ
- * is idempotent

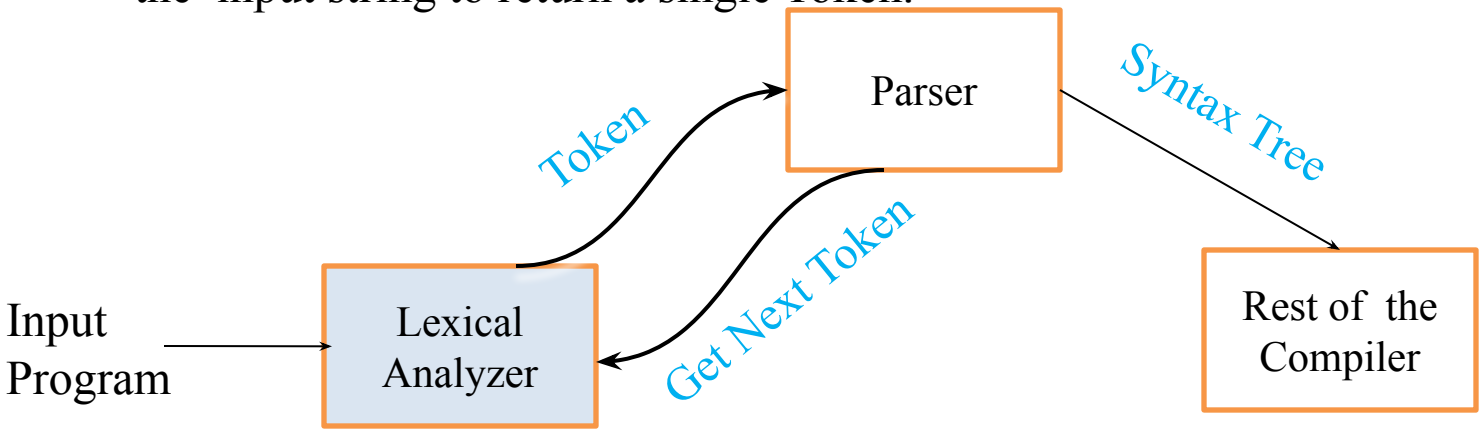
Lexical Analyzer



Lexical Analyzer (cont…)

- Where does the Lexical Analyzer fits into the rest of Compiler?

  - The front end of most compilers is Parser Driven.
  - When the parser needs the next Token, it involves the Lexical Analyzer.
  - Instead of analyzing the entire input string, the lexical analyzer sees enough of the input string to return a single Token.



Lexical Analyzer acts as a **Sub-routine.**

Contd…

- Issues in lexical analysis

1. Simple design
2. Compiler efficiency is improved
3. Compiler portability is enhanced

Lexical Analyzer (cont…)

- **Terms used in Lexical Analyzer:**

  - LEXEME-Smallest Logical Unit (Word) of Program.
    e.g.{ I, sum, buffer, for,10,+ …}

  - TOKEN –Set of Similar Lexemes.
    e.g.
    Identifier - { I, sum, buffer …}
    Keyword – { for, ….}
    Number – { 0, 23, ….}

  - PATTERN- as good as Regular Expression
    e.g. DIGIT [0-9]

Example

Token	Sample Lexemes	Informal Description of Pattern
const	const	const
if	if	if
relation	<,<=,=,<>,>,>=	< or <= or = or <> or > or >=
id	pi,count,D2	Letter followed by letters and digits
num	3.14,0,6.02	Any numerical constant
literal	“core dumped”	Any characters bet “ and “ except “

Lexical Analyzer (cont…)

- **Lexemes not passed to the parser:**

  - White Spaces (WS) – Tab, Blanks, New Lines
  - Comments

  These too have to be detected and Ignored.

Lexical Analyzer (cont…)

- **Tasks of a Lexical Analyzer:**

1. Scans the input program,identifies valid words of the language.
2. Removes extra white spaces,blanks,tabs,new lines ,comments etc
3. Expands user defined macros

   (done at the compile time by lexical analyser)

   e.g. #define Max 5

   #include<stdio.h>
4. Report presence of foreign words
5. May perform case conversion
6. It generates tokens and pass to syntax analysis phase.
7. Lexical Analyzer is implemented as Finite automata.

Lexical Analyzer (cont…)

- **Basic Tasks of a Lexical Analyzer:**
  - Recognizing Basic Elements.
  - Removal of White Spaces and Comments.
  - Recognizing Constants and Literals.
  - Recognizing Keywords and Identifiers.

< token, token value>

Pointer to Symbol Table Entry

Ex: < id, .>

< no, 9>

Lexical Analyzer (cont…)

- **Token:**
- Token stream: Each significant lexical chunk of the program is represented by a token
  - Operators & Punctuation: {}[]!+-=*;: …
  - Keywords: if while return goto
  - Identifiers: id & actual name
  - Constants: kind & value; int, floating-point character, string, …

Lexical Analyzer (cont…)

Example:

position = initial + rate * 60

Tokenized to:

position : The identifier

= : The Assignment Operator

initial : The identifier

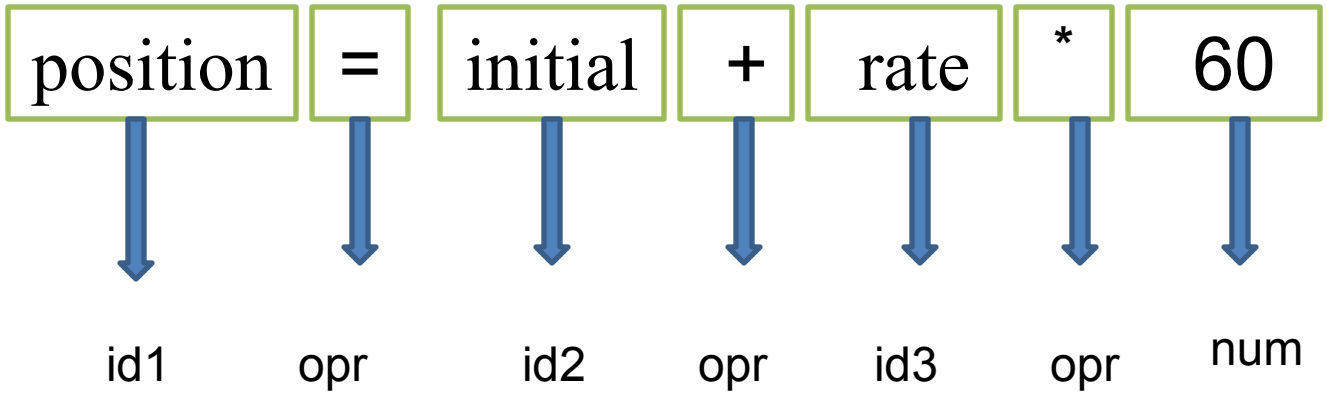
+ : The Plus Operator

rate : The identifier

* : The Multiplication Operator

60 : The Number/ Constant

Lexical Analyzer (cont…)



id1 = id2 + id3 * 60

Design of Lexical Analyzer

- Every action is implemented by Transition Diagram

- TG for Identifiers, Keywords, Operators…
- Regular Expression.
- Finite Automata.

Two Approaches

1. Hand Code : This is only of historical interest now.
   (possibly more efficient)

2. Use Generator : To generate the lexical analyzer from a format description.
   - The generation process is faster.
   - Less prone to Errors.

Contd…

- Lexical analyzer generator consists of two parts:

1. Specification of tokens – done through RE
2. Specification of actions

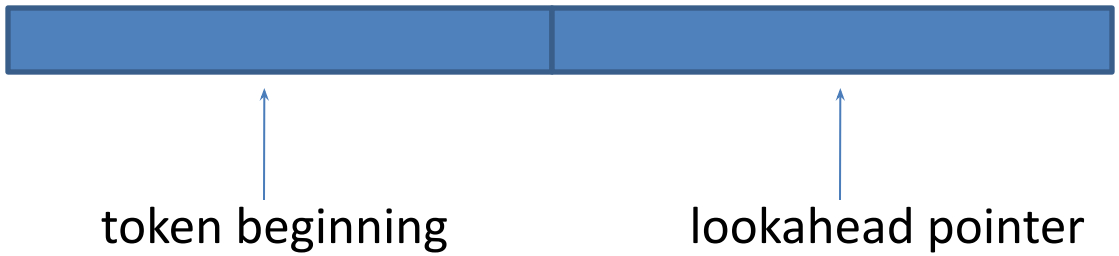
The lexical analyzer generator

- Processes RE s & forms a graph DFA
- Copies the action routines without any change
- Adds a driver routine

these 3 things put together constitutes the lexical analyzer.

Input buffering

- The lexical analyzer scans the characters of the source program one at a time to discover tokens.
- many characters beyond the next token may have to be examined before the next token itself can be determined.
- For this and other reasons, it is desirable for the lexical analyzer to read its input from an input buffer
- Figure shows a buffer divided into two halves of, say 100 characters each.



Contd…

- E.g. DECLARE(arg1,arg2,…,argn)

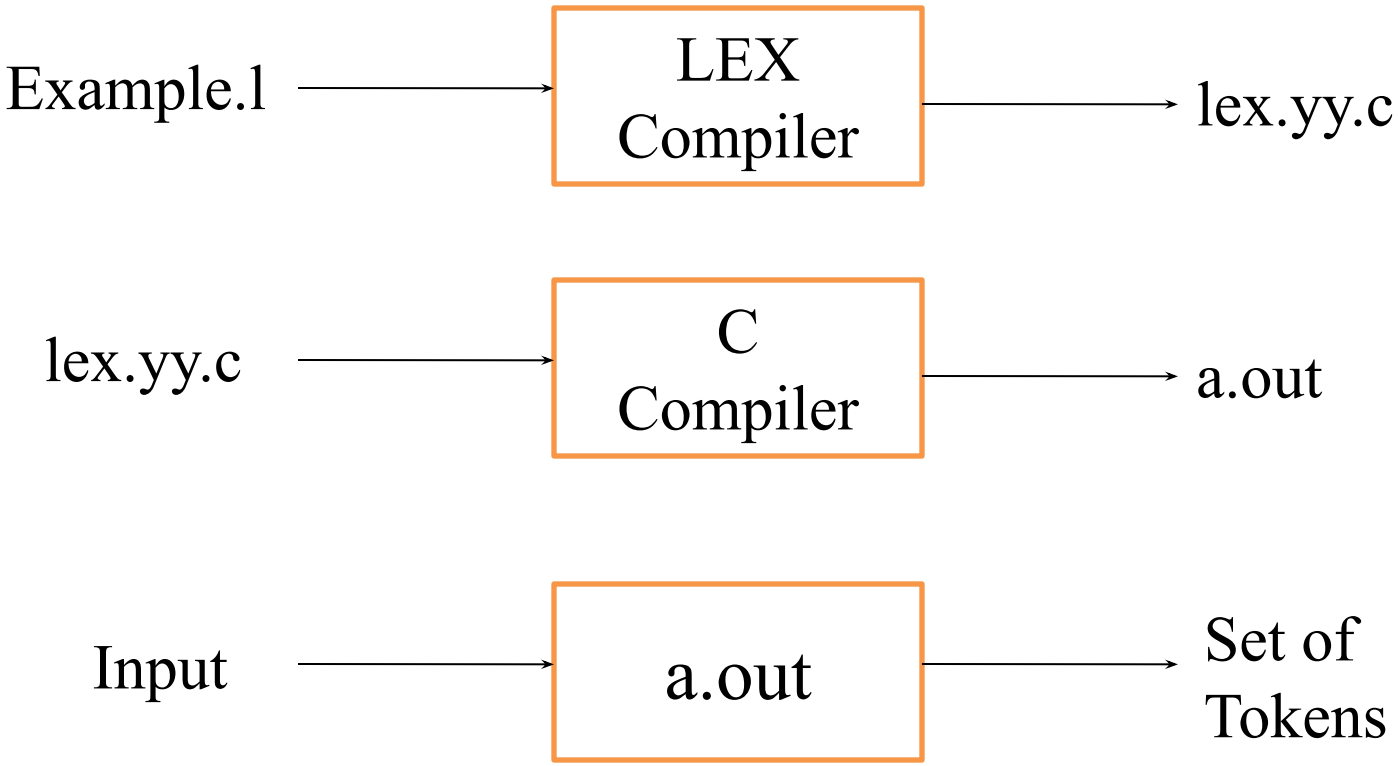
Without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis.

If the look ahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file.

Lexical Errors

- Matched but ambiguous:
  - left to the other phases(e.g., parser)
  - e.g., fi (a == f(x)) … : fi => identifier ?? misspelling of “if”
- d=2r ,no symbol can start with 2(digit)
- Unmatched:
  - Panic mode recovery: delete successive characters from the remaining input until a well-formed token is found
  - Repair input (single error):
    - deleting an extraneous character
    - inserting a missing character
    - replacing with a correct character
    - transposing two adjacent character

LEX


Example.l
LEX Compiler
lex.yy.c
lex.yy.c
C Compiler
a.out
Input
a.out
Set of Tokens

LEX Specification

- Declaration Section : Variable, Manifest Constant, Regular Definition.

%{

%}

- Translation Rules Section

P1 ……. { action1}
P2 ……. { action2}
P3 ……. { action3}
P4 ……. { action4}

%%

%%

COMPULSARY SECTION

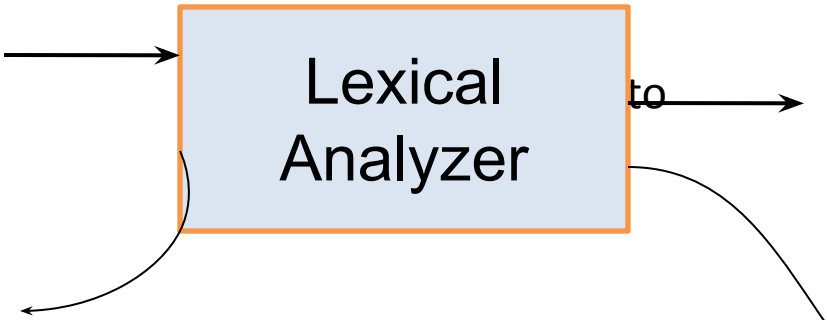
- Subroutine Section/Auxiliary Procedures

LEX (cont_...)

Implementing the Interactions

Uses getchar() In C to read a Character

Caller



Pushes back ch using

ungetc(ch,stdin)

global variable to

attribute value (yylval)

LEX (cont_...)

- Commands Used:

```
# lex filename.l
# cc lex.yy.c –ll
# ./a.out
```

Sample program

```
%{
%{
      #include<stdio.h>
      int wcnt=0, lcnt=0,
char_cnt=0;
%}
    charac      [^\n\t]
    eol          \n
   word          “    "
%%
     {eol}       {lcnt++; wcnt++;}
     {word}      {wcnt++;}
    {charac}    {char_cnt++;}
%%
```

Definition section

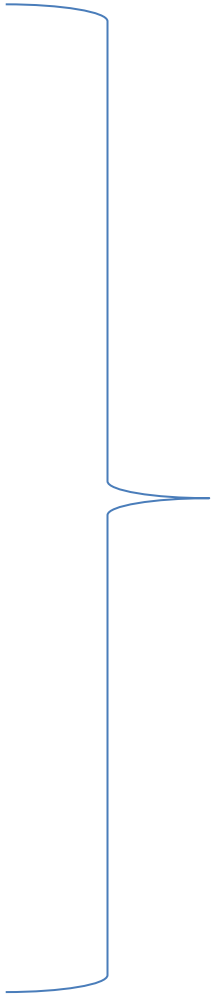
Regular definition section

Declaration section

Rules section

Contd….

```
main()
main()
{
        yyin=fopen("sample.txt","r");
                yylex();
      printf("\n\nNumber of lines: %d",lcnt);
      printf("\nNumber of words: %d",wcnt);
      printf("\nNo.
ofcharacters:%d",char_cnt);
}
int yywrap()
{
return 1;
}
```



Auxiliary procedure section

Contd…

yylex()

- Entry point
- Call yylex() to start or resume scanning
- If a lex action does a return to pass a value to the calling program, the next call to yylex() will continue from the point where it left off
- All code in the rules section is copied into yylex()

yywrap()

- When EOF is found it calls routine yywrap() to find out what to do next.
- Returns 0 –scanner continues scanning
- Returns 1 – the scanner returns zero token to report the EOF

Contd…

yytext

- Whenever a lexer matches a token the text of the token is stored in the null terminated string yytext
- When flex finds a match, yytext points to the first character of the match in the input buffer
- The value of yytext will be overwritten the next time yylex() is called.
- The value of yytext is only valid from within the matched rule's action

Regular Expression

.	Matches any single character except new line character
*	Matches 0 or more copies of the preceding expression
[]	char class which matches any char within the bracket
^	Matches the beginning of the line as 1st char of RE
$	Matches the end of line as the last char of a RE
{}	Indicates how many times the previous pattern is allowed to match when containing one or two nos.
\	Used to escape metacharacters & as part of the usual c escape sequences e.g. “\n”
+	Matches one or more occurrence of the preceding RE matches zero or one occurrence of the preceding RE
?	e.g. -?[0-9]+

Contd…

|	Matches either the preceding RE or the following RE e.g. is|am|are
"…"	Interprets everything within the quotation marks literally
/	Matches the preceding RE but only if the followed by the following RE
()	Groups series of RE together into a new RE

Lex Program

```
%{
    /* definitions of manifest constants
        LT, LE, EQ, NE, GT, GE,
        IF, THEN, ELSE, ID, Number, RELOP */
%}

/* regular definitions */
delim                [ \t\n]
ws                   {delim}+
letter               [A-Za-z]
digit                [0-9]
Id                   {letter} ({letter}|{digit})*
number               {digit}+(\. {digit}+)?(E+-]?{digit}+)?
```

Lex Program (cont…)

```
%%
{ws}          {/* no action and no return */}
If            {return (IF);}
then          {return (THEN);}
else          {return (ELSE);}
{id}          {yylval = (int) installID(); return (ID);}
{number}      {yylval = (int) installNum(); return (NUMBER);}
"<"           {yyval = LT; return (RELOP);}
"<="          {yyval = LE; return (RELOP);}
"="           {yyval = EQ; return (RELOP);}
"<>"          {yyval = NE; return (RELOP);}
">"           {yyval = GT; return (RELOP);}
">="          {yyval = GE; return (RELOP);}
%%
```

Lex Program (cont…)

int installID() {/* function to install the lexeme, whose first character is pointed to by yytext, and whose length is yyleng, into the symbol table and return a pointer thereto */

int installNum {/* similar to installID, but puts numerical constants into a separate table */

Scanner: Lexical Analysis

– What kind of **ERRORS** can be reported by **LA**?

- Issues an Appropriate Error Message
- Errors:

1. The Entire Lexeme is read and then truncated to the Specified Length.
2. Error of the Second Type-
   a. Skip Illegal Character.
   b. Pass the Character to the parser which has better knowledge of the context in which Error has occurred.
3. Wait till end of File and issue Error Message.

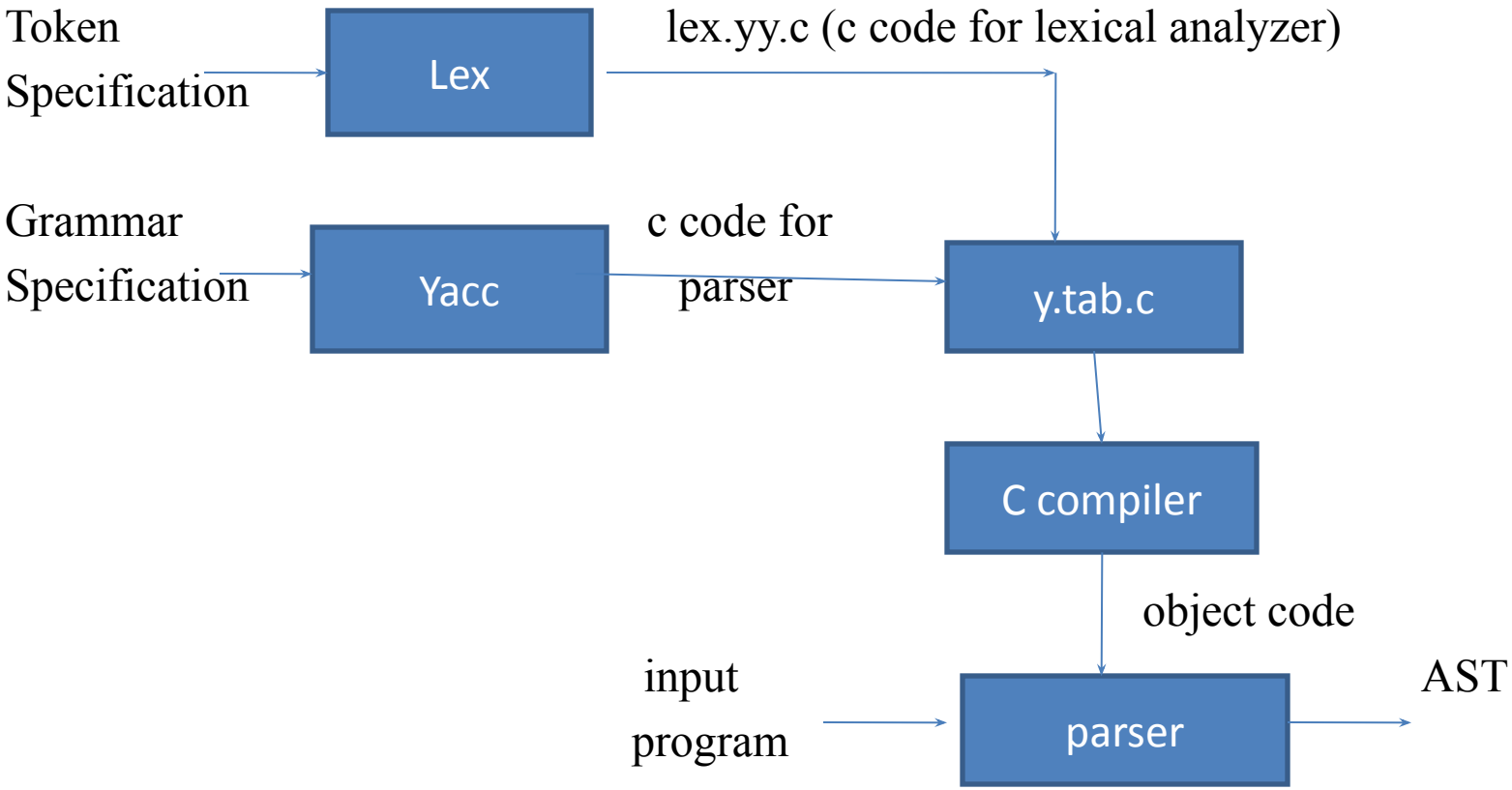
Like Misspelling of Keywords.

Yacc Introduction

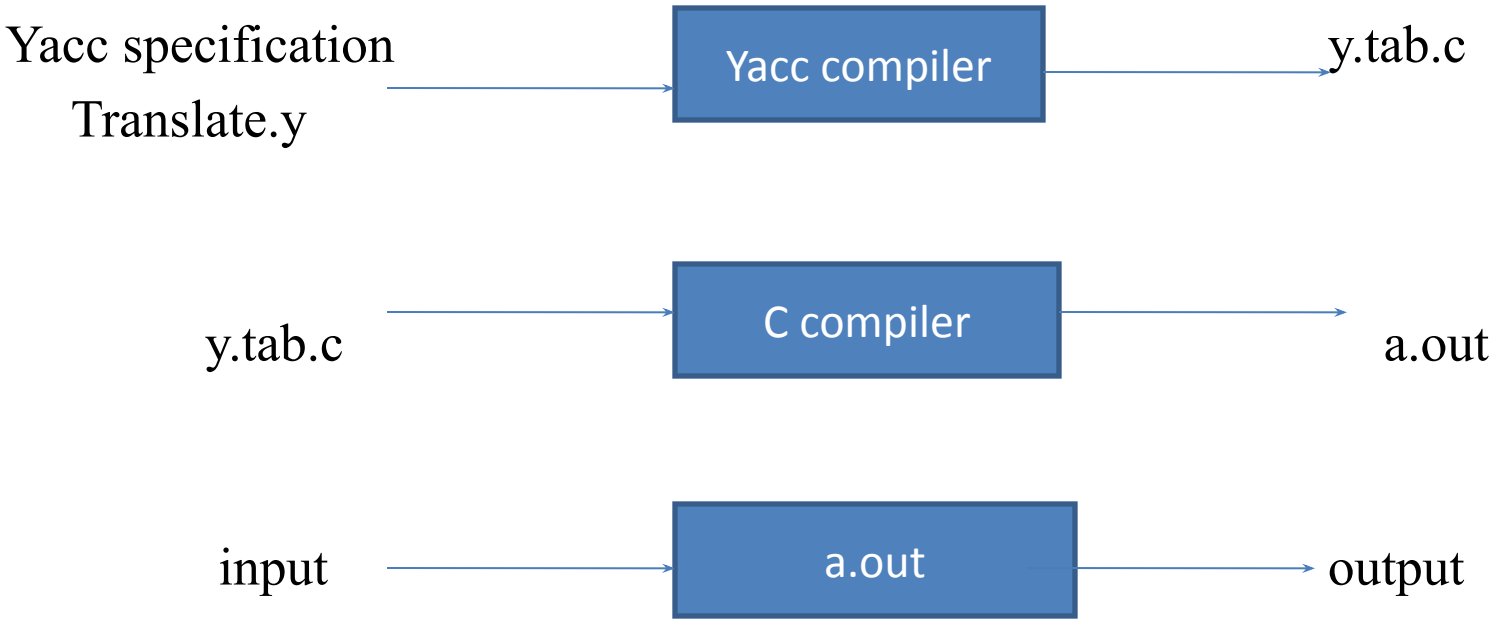
Yacc

- Parser invokes scanner for tokens.
- Parser analyze the syntactic structure according to grammars.
- parser executes the semantic routines**.**
- **Yacc – yet another compiler compiler.**
- An **LALR(1)** parser generator.
- Yacc generates
  - **Tables** – according to the grammar rules.
  - **Driver routines** – in C programming language.
  - **y.output** – a report file.

Yacc and Lex schema



Yacc


Yacc specification
Translate.y
Yacc compiler
y.tab.c
y.tab.c
C compiler
a.out
input
a.out
output

Contd…

A Yacc source program has three parts:

declaration
%%
translation rules
%%
supporting C routines

Contd…

The declaration part

There are two parts

First part – ordinary C declarations delimited by

%{ %}

Second part – declarations of grammar tokens

e.g. %token Name,Number

Contd…

Translation rules part:

Each rule consists of a grammar production and the associated semantic actions

<leftside> □ <alt1>|<alt2>|…|<alt n>

Would be written in Yacc as

<leftside> □ <alt1> { semantic action 1}

| <alt2> { semantic action 2}

…

|<alt n> { semantic action n}

A yacc semantic action is a **sequence of C statements.**

Contd…

Third section is

Auxiliary procedures Like

main( )

{

yyparse( );

yyerror( );

}

yyparse( )

- Entry point to a yacc generated parser is yyparse( )
- When your program calls yyparse( ),parser attempts to parse i/p stream
- Parser returns a value of zero if the parse succeeds

Contd…

yyerror( )

- Whenever a yacc parser detects a syntax error, it calls yyerror () to report error.

e.g.

```
yyerror(const char *msg)
        {
            printf("%d:%s at%s\n"yylineno,msg,yytext);
           }
```

Contd…

- In a semantic action the symbol

  $$ -- attributed value associated with NT on left

  $i -- value associated with ith grammar symbol (T or NT) on right

- Semantic action is performed whenever we reduce by the associated production
- Semantic action computes a value for $$ in terms of $i's.

e.g.

E□E+T | T

Associated semantic action is:

```
E : E '+' T        {$$ = $1+ $3;}
  | T              {$$ = $1;}
  ;
```

The symbol '$ ' is used as a signal to Yacc in this context.

To return a value , the action normally sets the pseudo variable "$$" to some value.

Contd…

E□E+T | T

Associated semantic action is:

```
E : E '+' T        {$$ = $1+ $3;}
  | T              {$$ = $1;}
  ;
```

The symbol '$ ' is used as a signal to Yacc in this context.

To return a value , the action normally sets the pseudo variable "$$" to some value.

To obtain the values returned by previous actions and the lexical analyzer ,the action use the pseudo variables $1,$2,…. Which refer to the values returned by the components of the right side of a rule, reading from left to right.

Contd…

E□E+T | T

Associated semantic action is:

```
E : E '+' T        {$$ = $1+ $3;}
  | T              {$$ = $1;}
  ;
```

NT 'T' in the first prod is the 3rd grammar symbol on the right while '+' is 2nd.

In general

{$$ = $1;}

is the default semantic action.

Line: E '\n' { printf("%d\n",$1); }

Supporting C routines part:

the lex yylex( ) produces pairs consisting of a **token** & its associated **attribute value.**

The **attribute value** associated with a token is communicated to the parser through a yacc defined variable **yylval**

contd

- **Symbol values and actions**

Every symbol in a yacc parser has a value

	Symbol	**Values**
1.	Number	particular number
2.	Literal text string	pointer to a copy of the string
3.	Variable	ptr to a symbol table entry describing the var

In real parsers, data types are

int,double for **numeric symbols**

char * for **strings**

pointers to structures for **high level symbols**

For all thesse,yacc can creat a

C union typedef --- YYSTYPE

It contains all these datatypes.

Contd…

```
  %token  Name Num
%%
Stmt : Name ‘=‘ E
        | E                    {  printf(“=%d\n”,$1); }
        ;
     E : E ‘+’ Num           { $$ = $1 + $3; }
        | E ‘+’ Num           {$$ = $1 -  $3; }
        |  Num                 { $$ = $1 ;}
        ;
```

Contd…

The lexer

```
%{
    #include “y.tab.h”
   extern int yylval;
%}
%%
[0-9]+       { yylval = atoi (yytext);
                  return num ;           }
[a-z]+        {yylval = yytext;
                  return name;            }
%%
```

Contd…

- Whenever the lexer returns a token to the parser, if the token has an **associated value**, the lexer must store the value in **yylval** before returning.
- **yylval** is a **union** and puts the definition in **y.tab.h**

Compiling and running a parser

- Yacc takes a grammar & creates y.tab.c in a C language parser & y.tab.h,the include file with the token number definitions.
- Lex creates lexyy.c, the C language lexer.
- Now we need only compile them together with yacc & lex libraries.

Contd…

Compiling and running parser:

>**byacc –d parsetry.y**

it makes y.tab.c & y.tab.h

>**flex parsetry.l**

it makes lexyy.c

- Then “open project” menu
- Name the project file
- Add item in Alt-p
- Add lexyy.c
- Add y.tab.c
- And run the project

compile and link C files

Contd…

Precedence

- It controls which operators to execute first in an expression
- According to mathematical & programming tradition multiplication & division takes precedence over addition & subtraction.

e.g. **a+b*c means a+(b*c)**

d/e – f means (d/e)-f

- In any expression grammar ,operators are grouped into levels of precedence from lowest to highest.
- The total number of levels depend on the language.
- The C language has total **fifteen levels.**

Associativity

- It controls the grouping of operators at the same precedence level.
- Operators may group **to the left**

e.g. **a-b-c in C means (a-b)-c**

Or **to the right**

a=b=c in C means a=(b=c)

Contd…

- There are two ways to specify precedence & associativity in a grammar, **implicitly & explicitly**
- **Implicitly** – rewrite the grammar using separate NTs for each precedence level

e.g. assume usual precedence & left assciativity for everything

```
exp      : exp '+' mulexp
         | exp '-' mulexp
         | mulexp
         ;
mulexp:mulexp '*' primary
          | mulexp '/' primary
          | primary
          ;
primary : '(' exp ')'
          | '-' primary
          | num;
```

Contd…

Explicitly

- Add these lines to the definition section

%left ‘+’ ’-’

%left ‘*’ ‘/’

%nonassoc UMINUS

- Each of these declarations defines a level of precedence
- ‘+’ ‘-’ are left associative & at the lowest precedence level.
- ‘*’ ‘/’ are left associative & at the highest precedence level.

- If **shift/reduce conflict** encounters then parser consults the table of precedence
- If all of the rules involved in conflict include a token which appears in a precedence declaration, then it uses precedence to resolve the conflict.

Contd…

Precedence:

- The **precedence & associativities** are attached to tokens in the declaration section.
- This is done by a series of lines beginning with yacc keyword**:%left, %right or %nonassoc** followed by list of token.
- All tokens on the same line are assumed to have the same precedence level & associativity.
- The lines are listed in order of increasing precedence.
- e.g.

%left '+' '-'

%left '*' '/'

+ & - are left associative and have lower precedence than * & / ,are also left associative.

Contd…

```
%right '='
%left'+' '-'
%left'*' '/'
%%
exp : exp '=' exp
      | exp '+' exp
      | exp '-' exp
      | exp '*' exp
      | exp '/' exp
      |name
      ;
```

This grammar can be used to structure i/p a=b=c*d-e-f*g

As : a=(b=((c*d)-e)-(f*g)))

Contd…

- Keyword %prec,changes the precedence level associated with a particular grammar rule

e.g.

```
exp : exp '=' exp
    | exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | exp '/' exp
    | '-' exp %prec '*'
    | name
    ;
```

Contd…

```
Stmt_list : stmt' \n'
          | stmt_list stmt '\n'
          ;
     Stmt : Name '=' exp          {vbltable[$1]=$3;}
          | exp                    {printf("%g\n",$1);}
          ;
      exp : exp'+'exp              {$$ =$1 + $3;}
          | exp '-'exp              { $$ = $1-$3;}
          | Num
          | Name                   {$$= vabltable[$1];}
          ;
%%
main( )
{ yyparse( );}
```

Contd…

Lex file

```
%{
#include”y_tab.h”
#include<math.h>
Extern double vbltable[26];
%}
%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)
                {       yylval.dval=atof(yytext);
                        return Num;
                 }
[\t]        ;
[a-z]       {yylval.vblno=yytext[0]-’a’;
              return Name;
“$”         {return 0;}
%%
```

Contd…

Token numbers

- Tokens are identified by small integers
- Token number of a literal token is the number value in ASCII
- Symbolic token usually have values assigned by yacc
- User can assign token numbers

e.g.

```
%token  UP 50 DOWN 60
```

- Two tokens cannot have same nos.
- Lexer needs to know the token numbers in order to return the appropriate values to the parser.

Contd…

y.tab.h

Header file y.tab.h includes a copy of definitions so that it can be used in lexer.

```
#define Name 257
#define Number 256
#define UMINUS 259
```

Token Values

- Each symbol in a yacc parser can have an associated value
- Token value is always stored in the variable yylval

e.g.

```
[0-9]+    {  yylval=atoi(yytext);
             return Number;}
```

Contd…

%union declaration

```
%union{
        double dval;
         int vblno;
       }
```

- The contents of the declaration are copied verbatim to the output file as the contents of a C union declaration defining the type YYSTYPE as a C typedef.
- The generated header file y.tab.h includes a copy of the definition so that you can use it in lexer

Ex.where y.tab.h is generated from grammar

Contd…

```
#define Name 257
#define Number 256
#define UMINUS 259
typedef union{
                    double dval;
                    int vblno;
                  }YYSTYPE;
extern YYSTYPE yylval;
%token <vblno>Name
%toekn <dval> Number
%type <dval>expression
```

generated file also declares the variable yylval & defines the token number for the symbolic tokens in the G

Note:%type sets the type for NTS.

Contd…

- How to handle variables with single letter names?
- How to handle multiple expressions?
- How to use floating point values?

```
%{
        double vbltable[26];
%}
%union{
        double dval;
        int vblno;
        }
%token <vblno> Name
%token <dval> Num
%left '-' '+'
%left '*' '/'
% nonassoc UMINUS
%type<dval> exp
%%
```