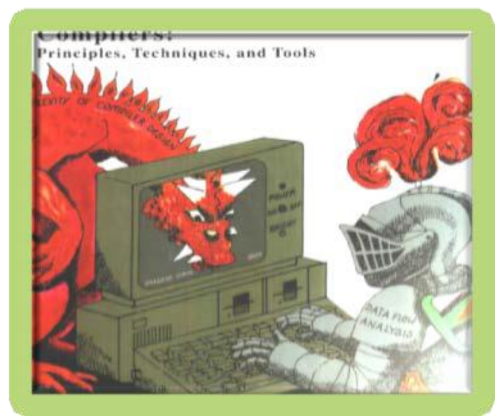
MIT-WPU

॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

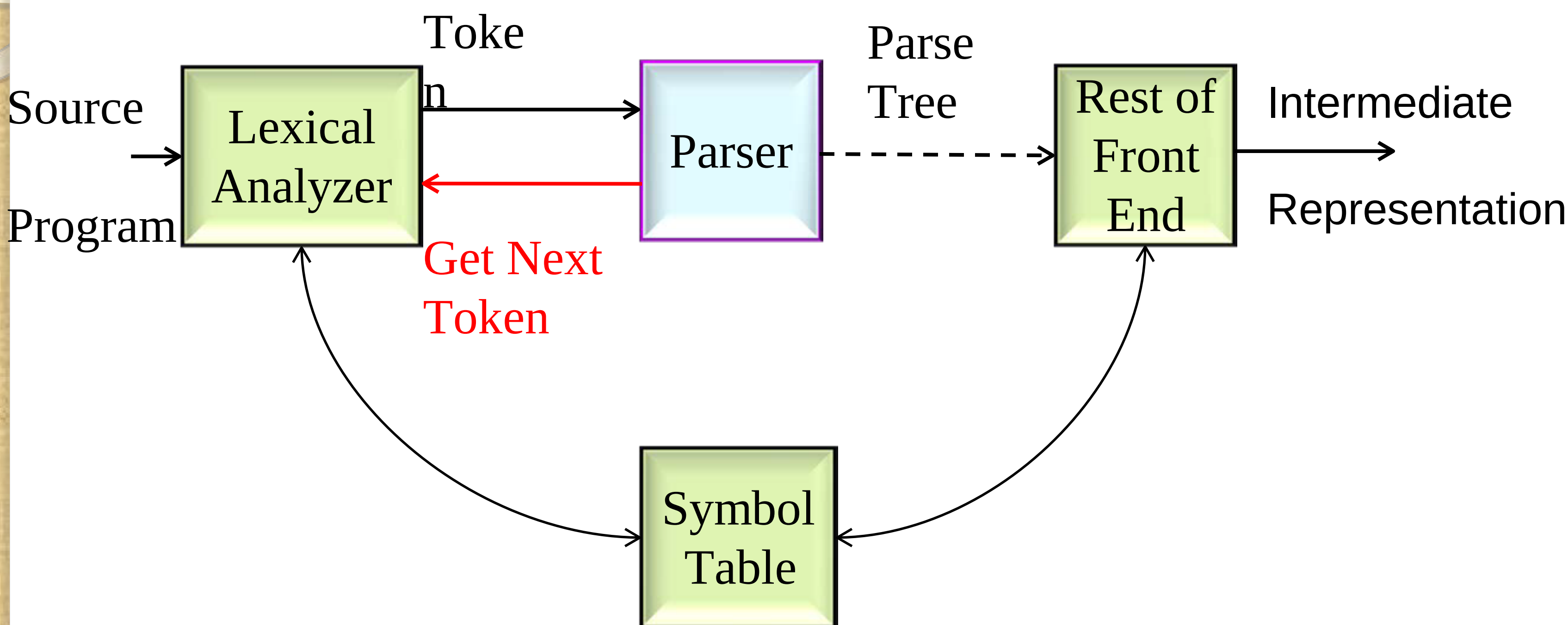
Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

System Software
Compiler Design



School of Computer Engineering and Technology

Role of Parser / Syntax Analysis
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

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*::= α , 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

<u>Syntax Analysis Problem Statement:</u> 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 → E + E | E * E | (E) | - E | **id**

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

Yes because there is the following

derivation: E ⇒ -E ⇒ -(E) ⇒ -(E + E) ⇒ -

(**id** + **id**)

Where ⇒ reads "derives in one step".

Parse trees

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

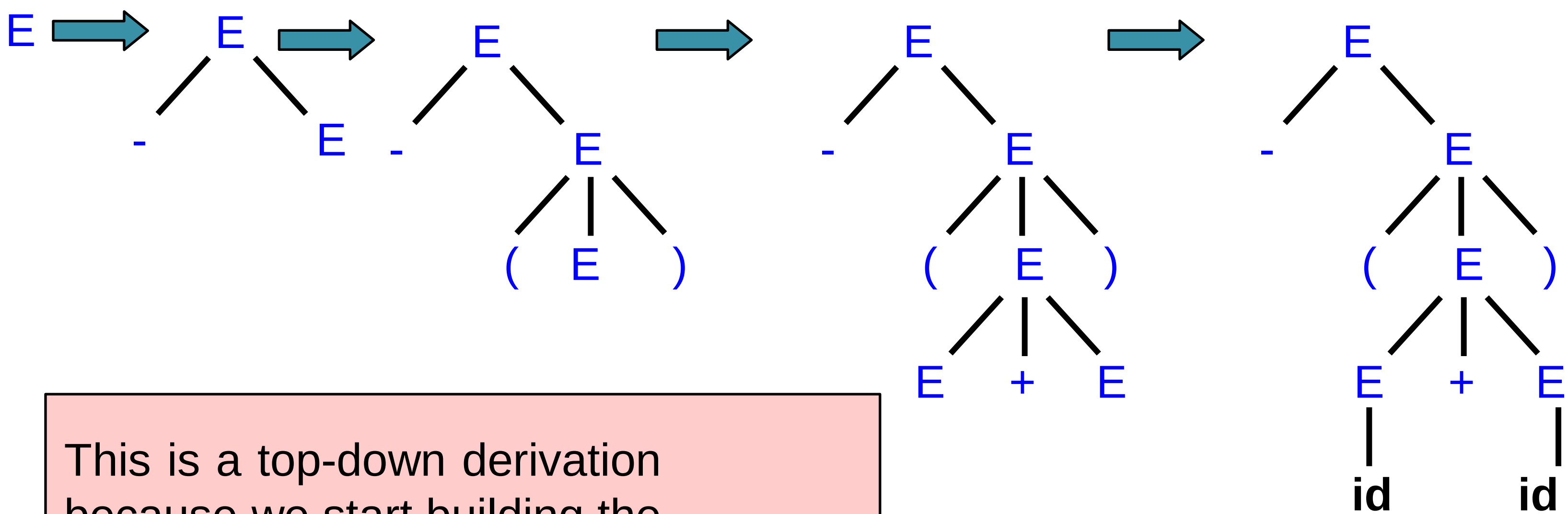
Tree nodes represent symbols of the grammar (nonterminals or terminals) and tree edges represent derivation steps.

Derivation

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

Lets examine this derivation:

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



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

parse tree

Another Derivation Example

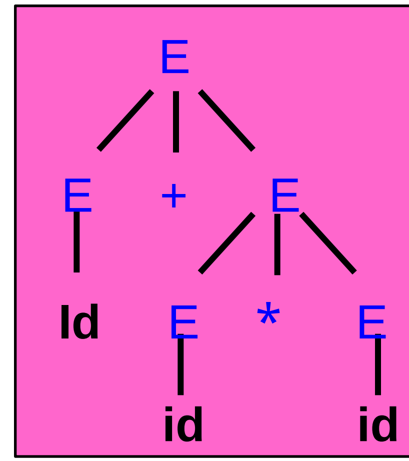
$E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \textbf{id}$

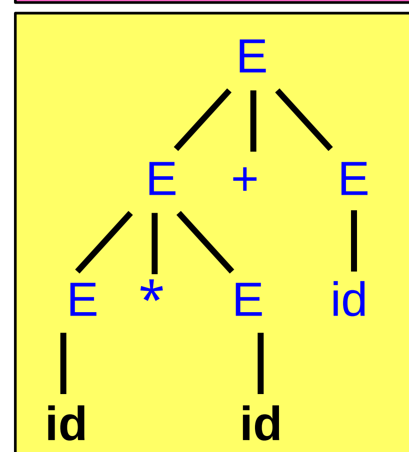
Find a derivation for the expression:

id + id * id

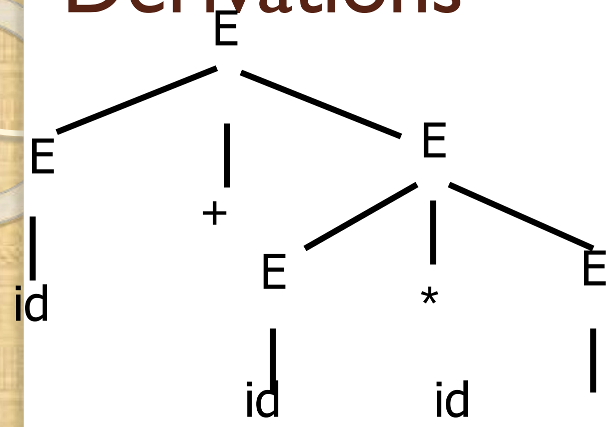
According to the grammar, both are correct.

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

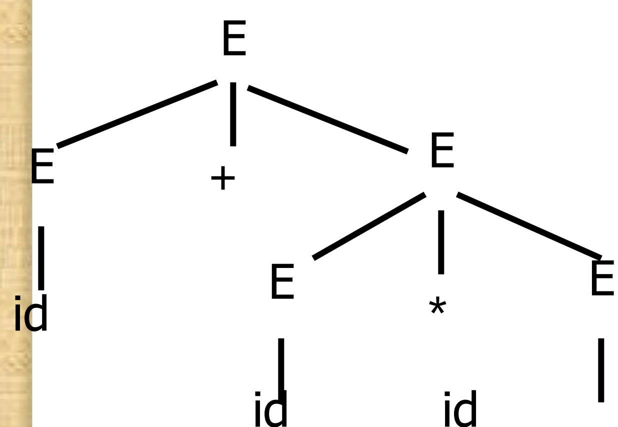




Parse Trees and Derivations



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

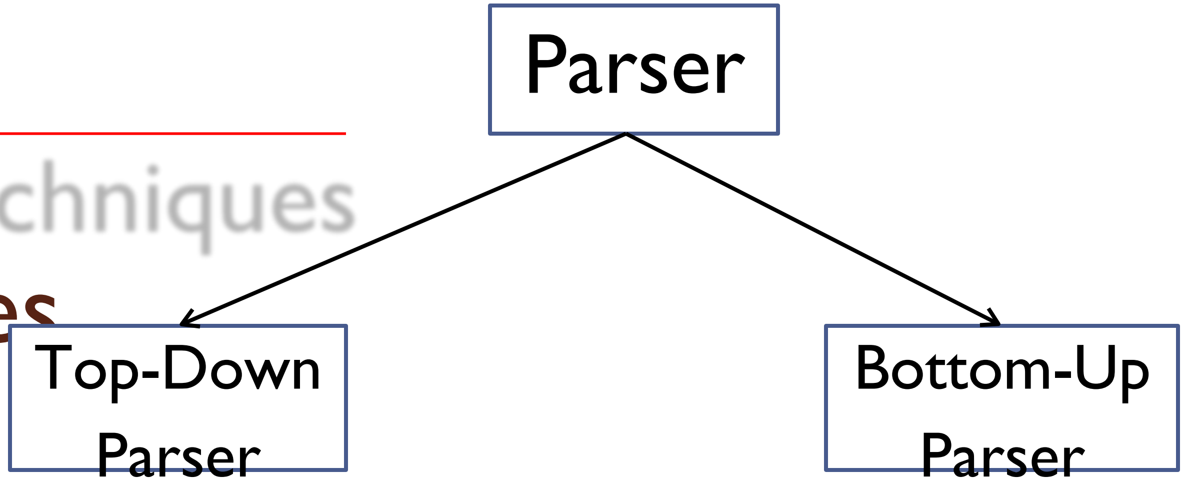
Predictive: Guess the structure of the parse tree from the next input

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

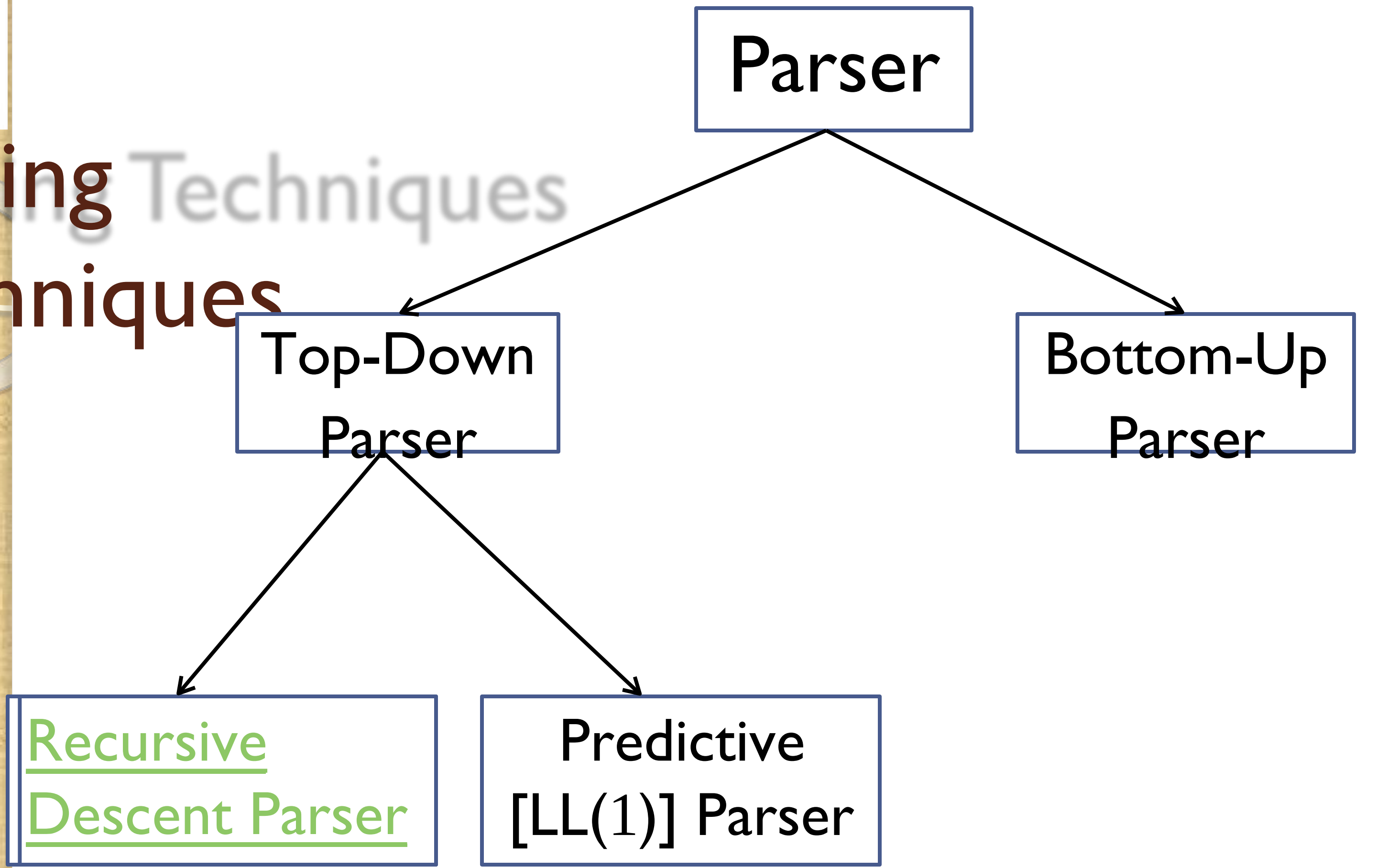
Imp



Parsing Techniques

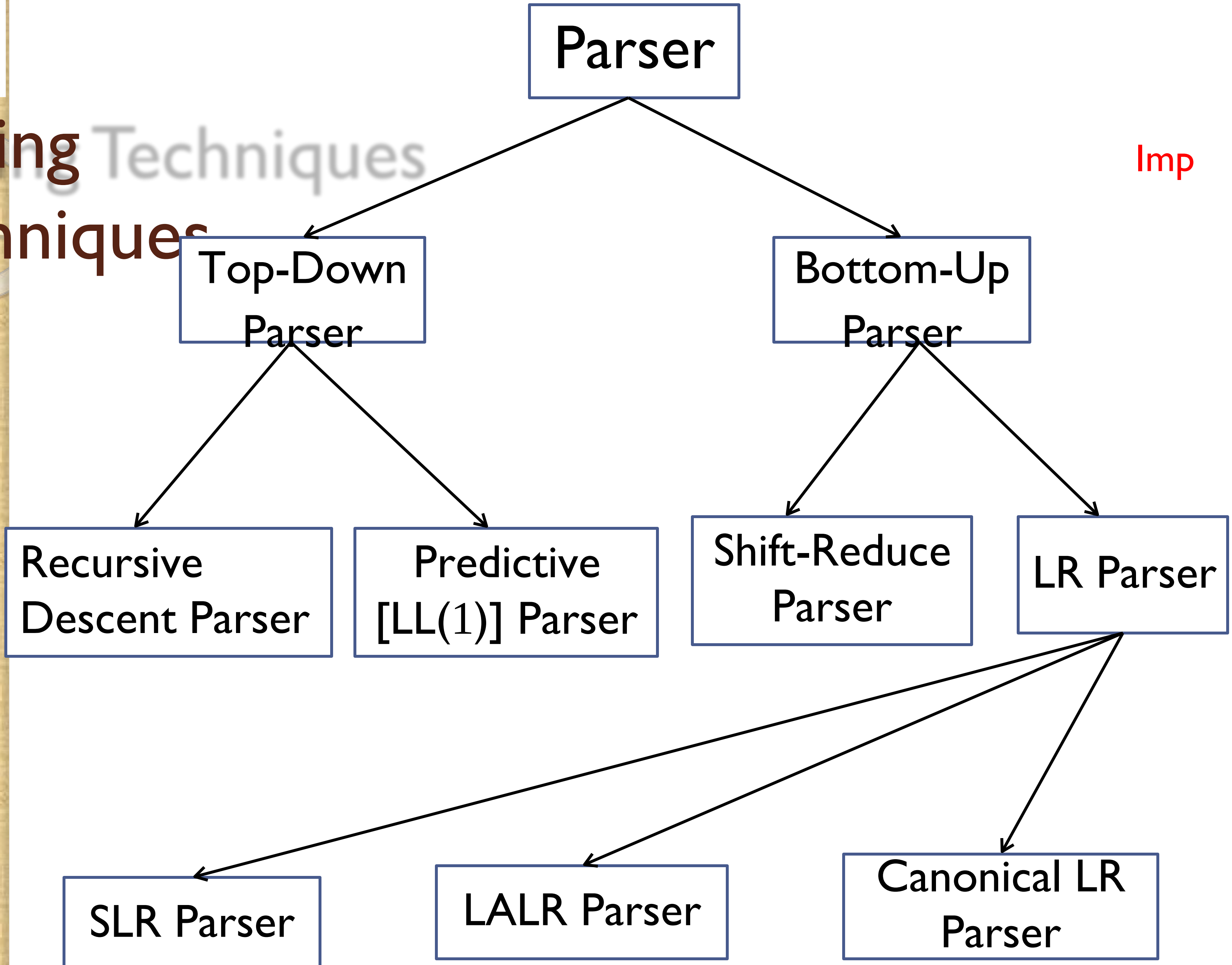
Parsing Techniques

Imp



Parsing Techniques

Imp

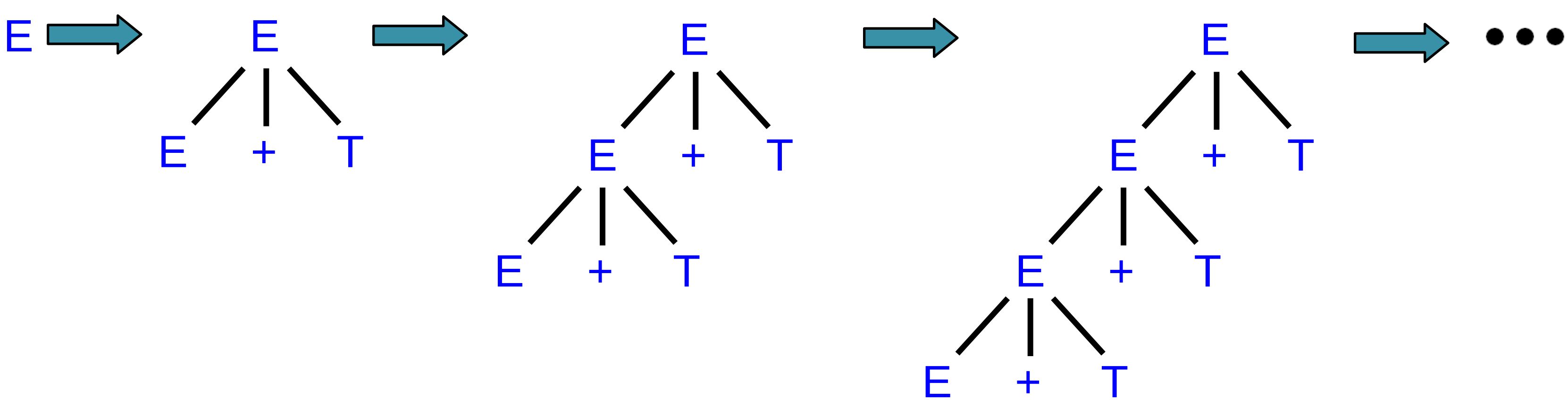


Left Recursion

Consider the grammar:

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

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



Left Recursion

Consider the grammar:

E → E + T | T
T → T * F |
F F → (E) |
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 $A + \longrightarrow A\alpha$ for some string α.
- Top down parsing methods cannot handle left recursive grammars, so a transformation that eliminates left recursion is needed.

E.g. $A \rightarrow A\ \alpha \mid \beta$

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

$A \rightarrow \beta A'$

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

Contd…

- The technique to eliminate left recursion is:

$A \rightarrow A\alpha 1 \mid A\alpha 1 \mid \ldots \mid \beta 1 \mid \beta 2 \mid \ldots \mid \beta n$

no βi begins with A.

So we replace the A-productions

by $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ \ldots \mid \ \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots . \mid \alpha_m A' \mid \epsilon$

e.g. consider the G as

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \epsilon$

Left Recursion

This left-recursive grammar:

E → E + T | T
T → T * F | F
F → (E) |id

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

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

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

Where γ : all alternatives that do not begin with

α by $A \rightarrow \alpha A' \mid \gamma$

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

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))

Scans Input from Left to Right

Generates Leftmost Derivation Tree

One (1) Look ahead Symbol

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.

  $A \rightarrow \alpha 1 / \alpha 2 / \alpha 3 \ldots\ldots / \alpha n$

- 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

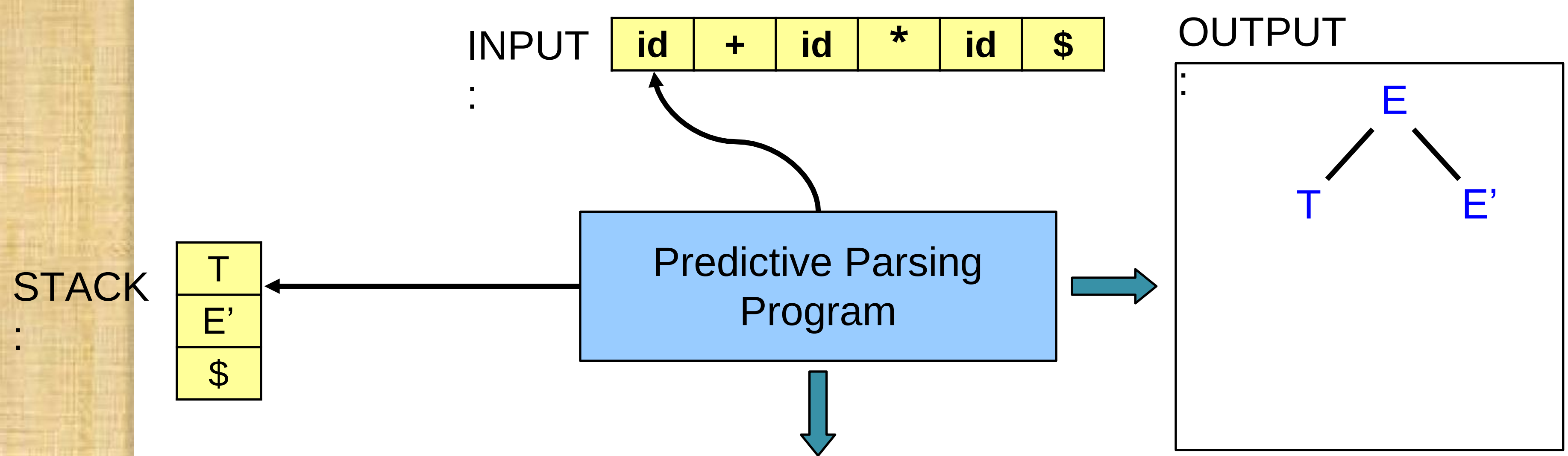
Grammar:

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)		

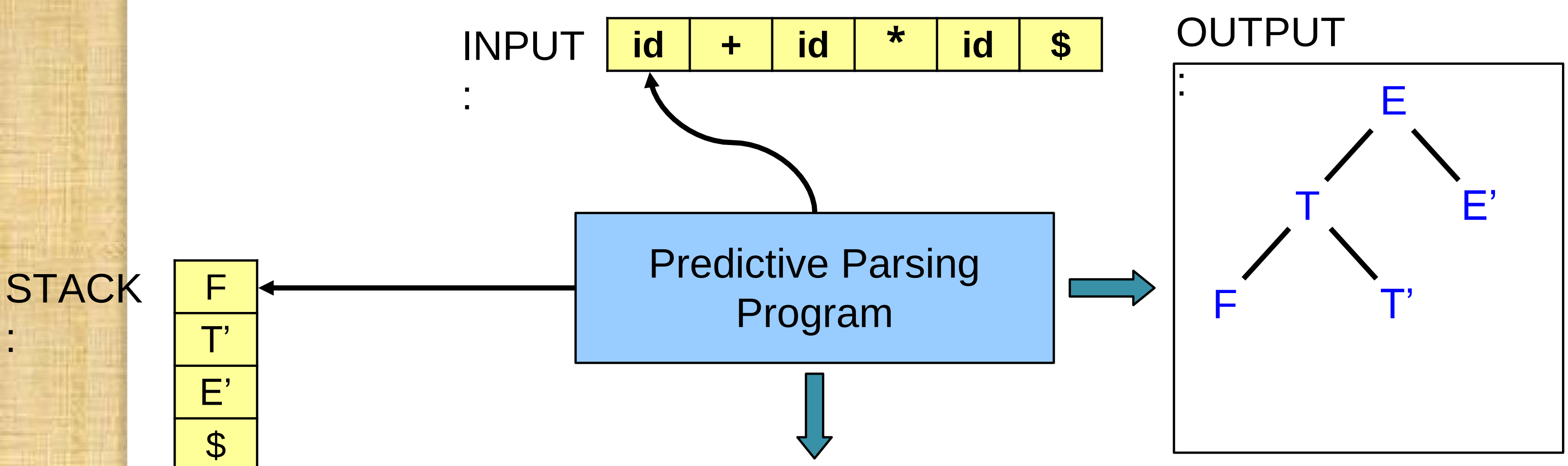
A Predictive Parser



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

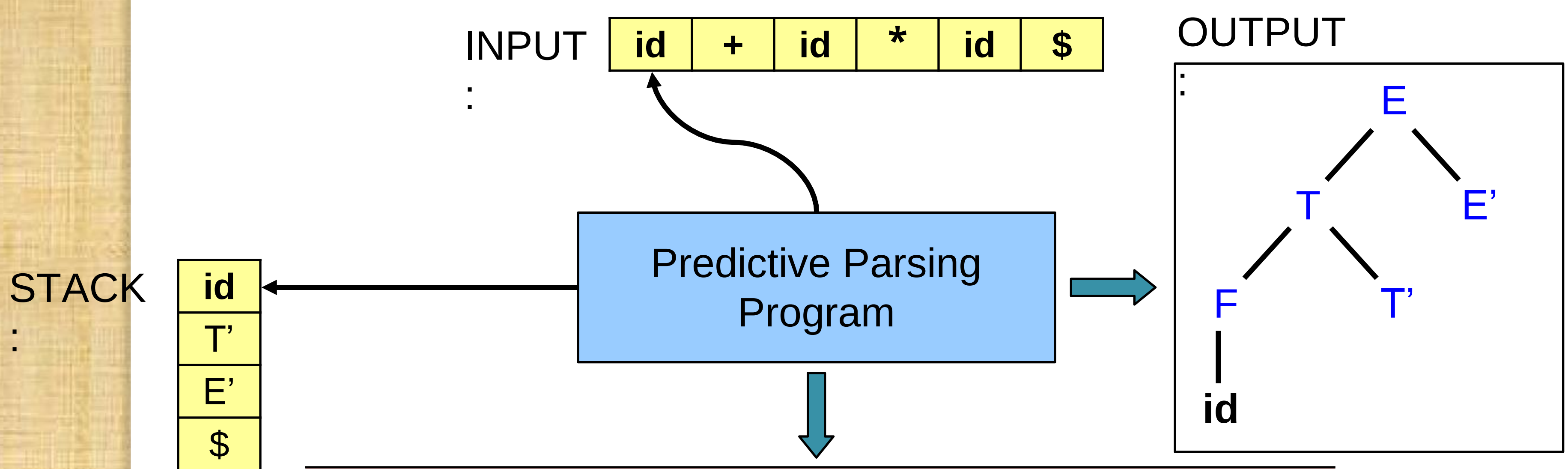


PARSING TABLE:

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 $\neq$ \$: Pop stack, advance input.

INPUT :

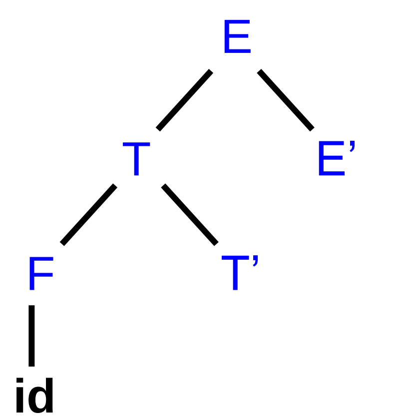
id	**+**	**id**	*****	**id**	**$**

Predictive Parsing Program

STACK :

id
T'
E'
$

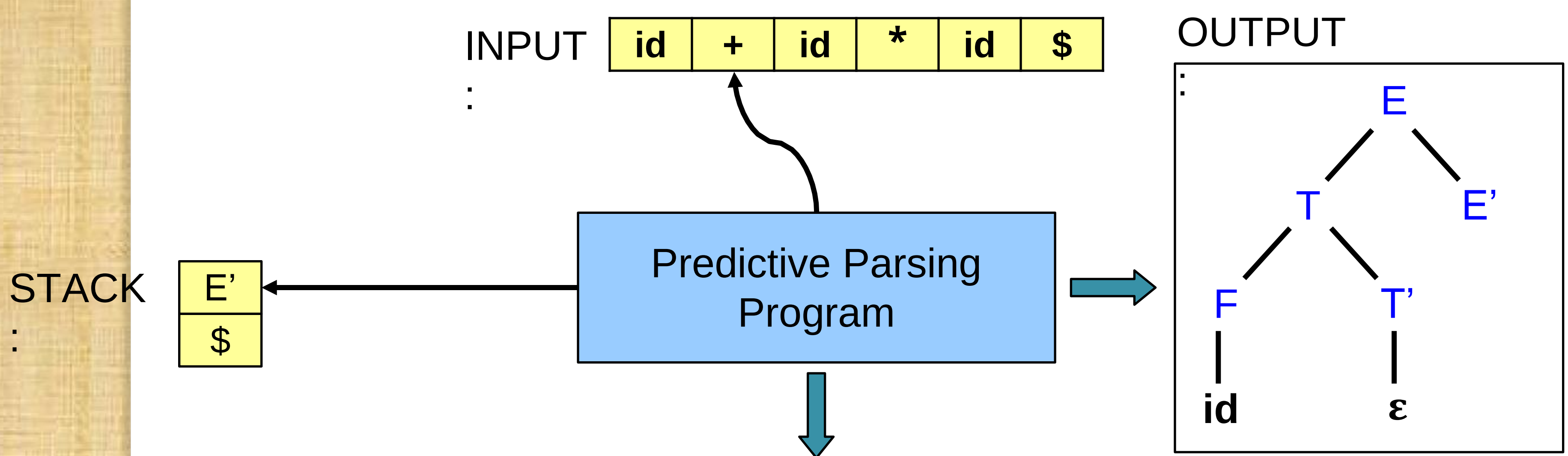
OUTPUT :



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' → +TE'

T → FT'

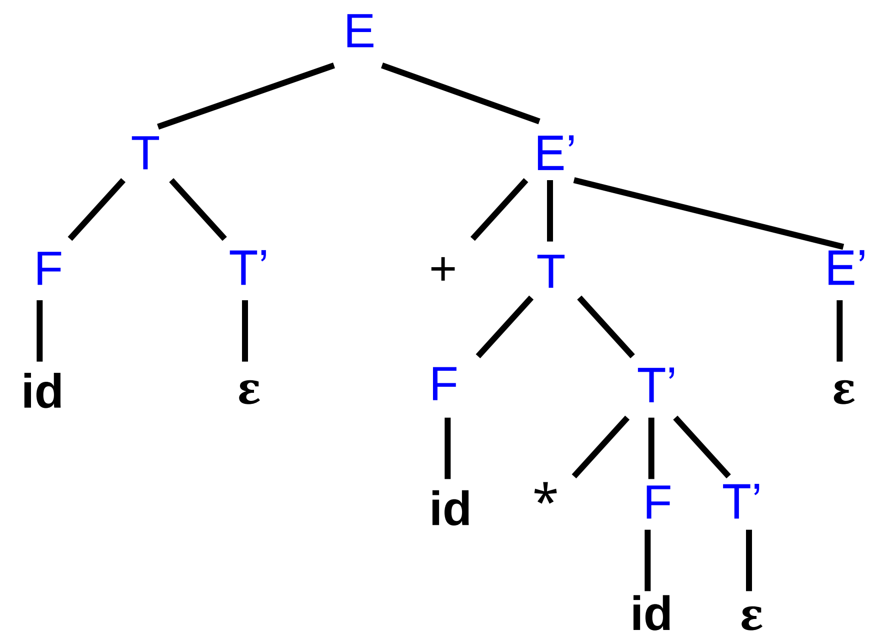
F → **id**

T' → * FT'

F → **id**

T' → **ε**

E' → **ε**



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:

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

Rules to Create FIRST

GRAMMAR:

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

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 $X \rightarrow Y_1Y_2 \cdots Y_k$ and $Y_1 \cdots Y_{i-1} \Rightarrow^{*} \varepsilon$ 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

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}

Rules 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 rules:

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

3a. If $A \rightarrow \alpha B \beta$
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

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)

Rules to Create FOLLOW

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

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)

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 → 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) = {**+**, *****, **)**, **$**}

...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**

ules t arsin

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) = {**+**, *****, **)**, **$**}

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**

...ules t... ...arsin...

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) = {**+**, *****, **)**, **$**}

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:

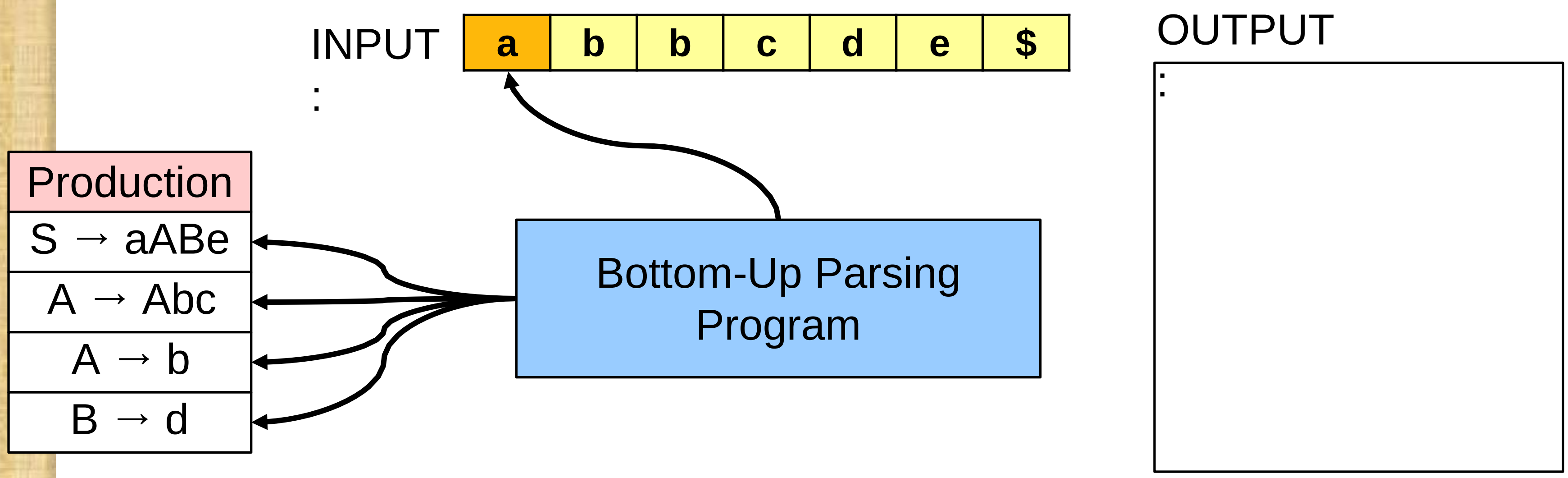
Bottom-Up Parser

Consider the Grammar:

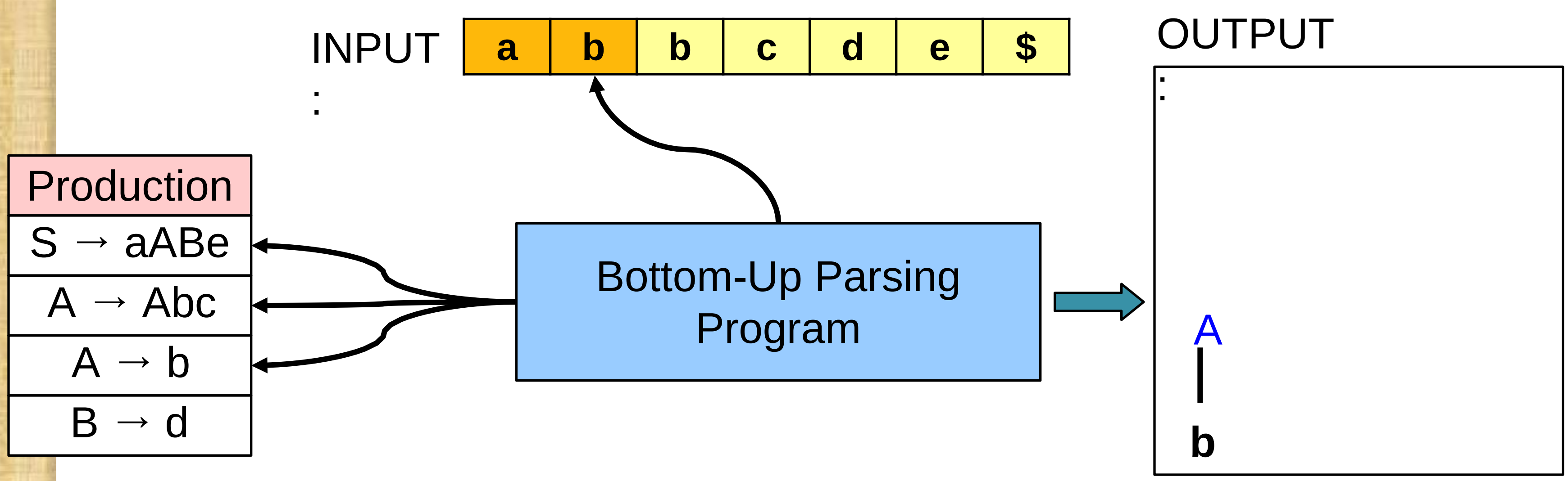
$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

We want to parse the input string abbcde.

Bottom-Up Parser Example

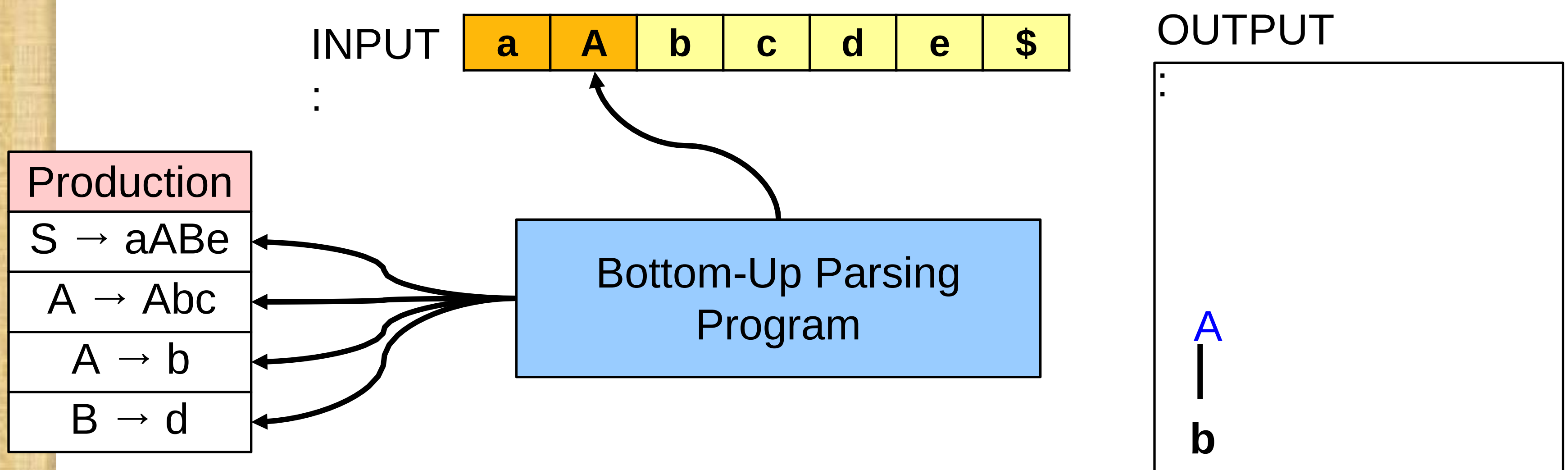


Bottom-Up Parser Example



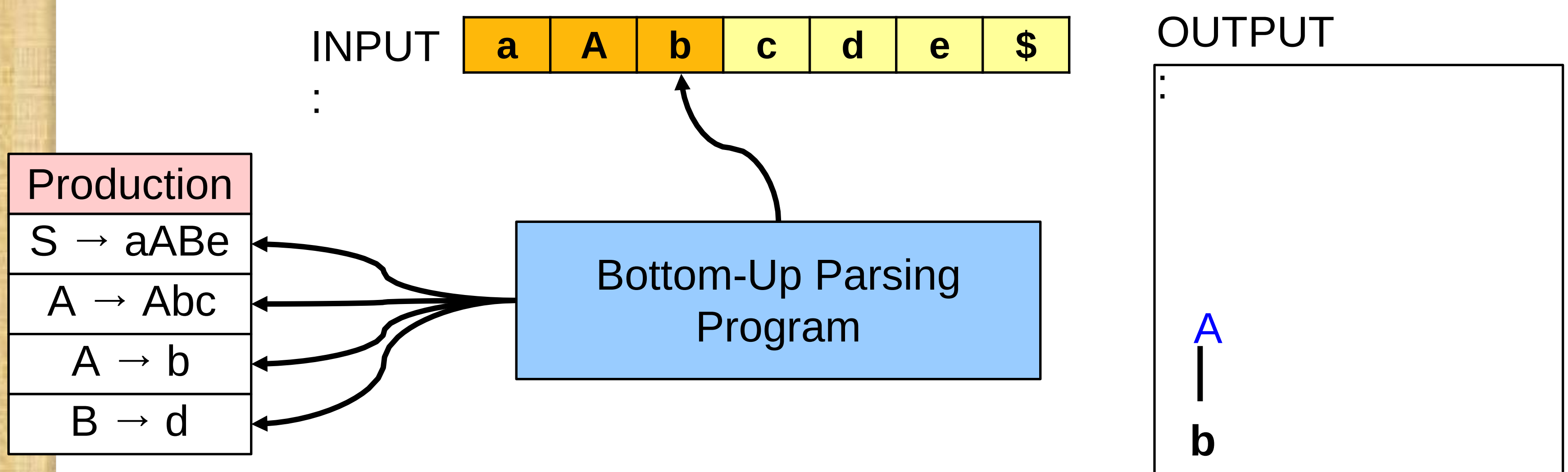
(Aho,Sethi,Ullman, pp. 195)

Bottom-Up Parser Example



(Aho,Sethi,Ullman, pp. 195)

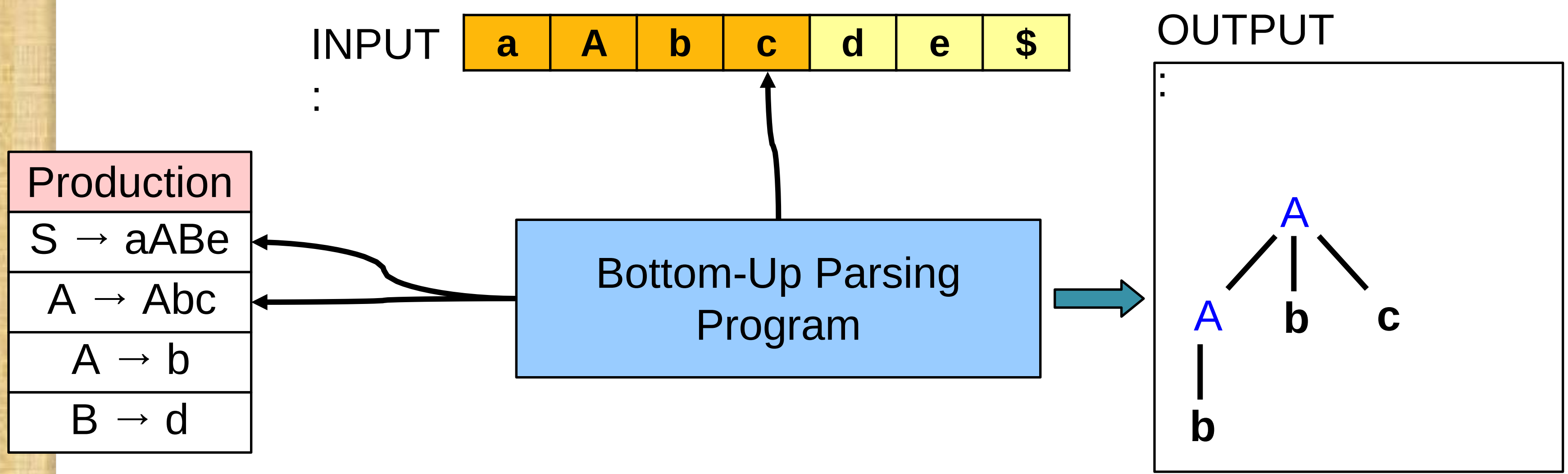
Bottom-Up Parser Example



We are not reducing here in this example.

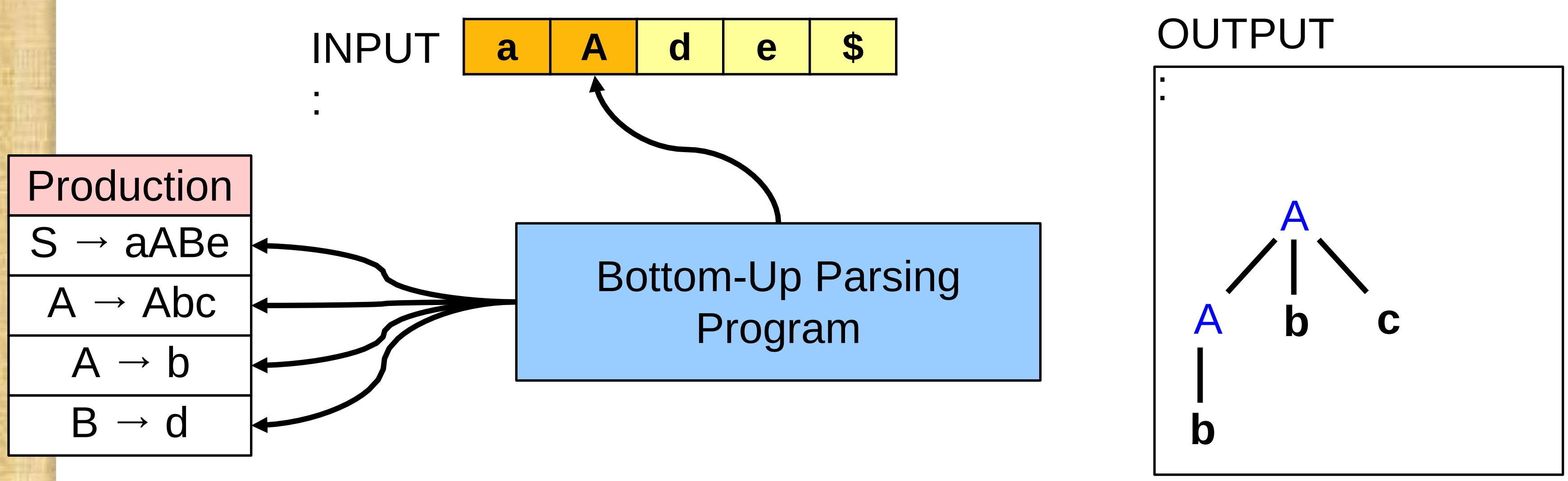
A parser would reduce, get stuck and then backtrack!

Bottom-Up Parser Example



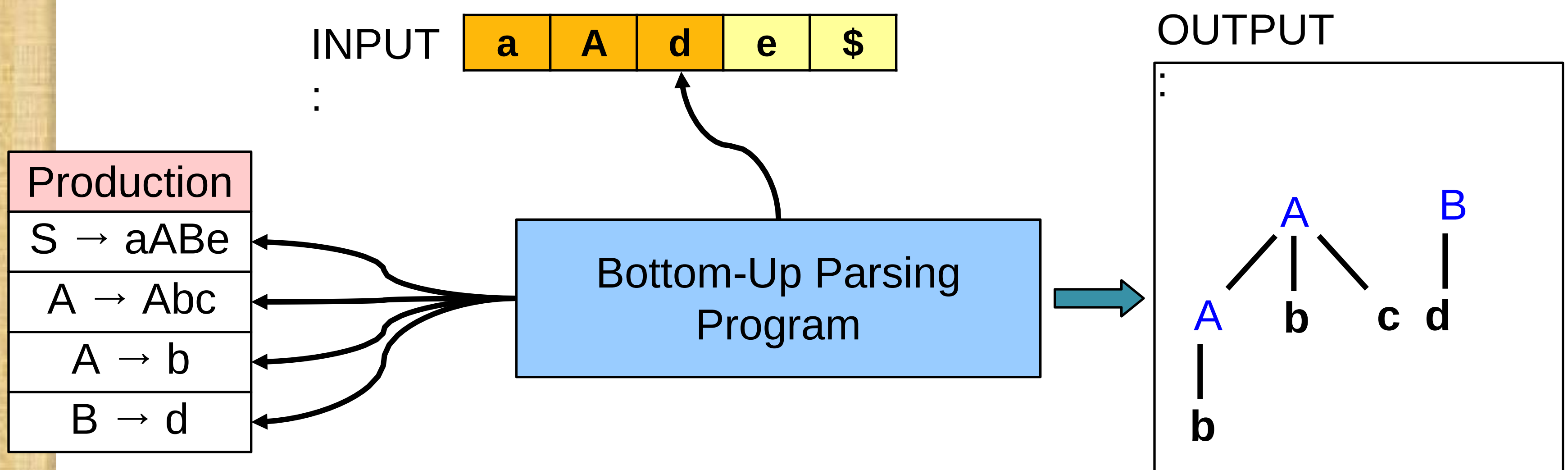
(Aho,Sethi,Ullman, pp. 195)

Bottom-Up Parser Example



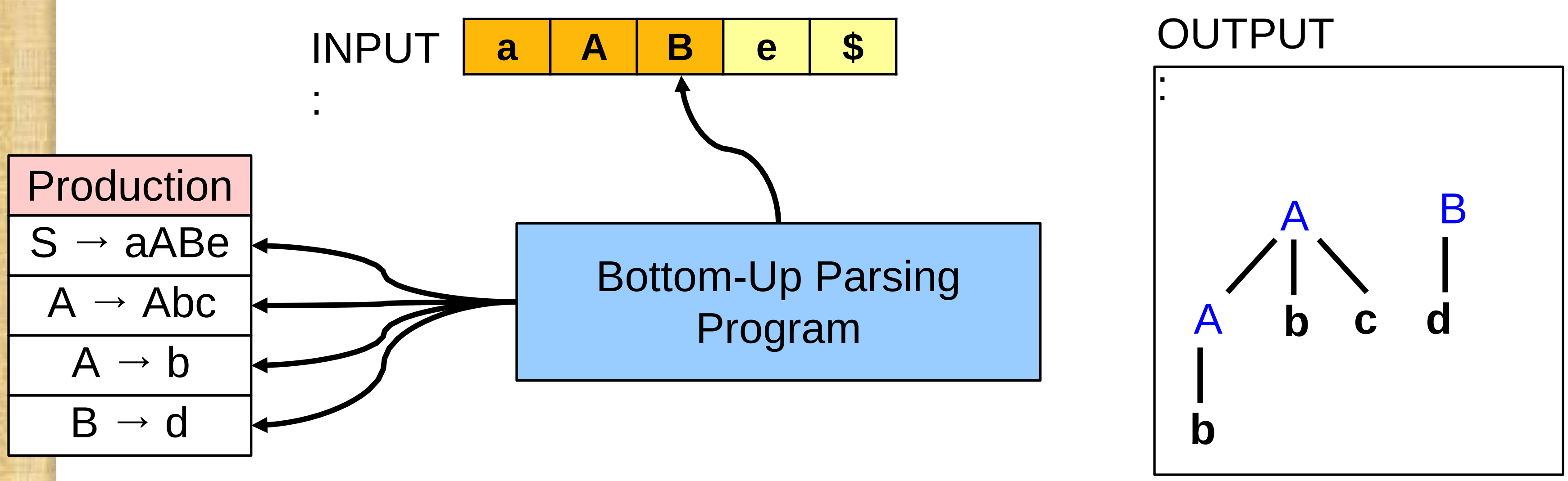
(Aho,Sethi,Ullman, pp. 195)

Bottom-Up Parser Example



(Aho,Sethi,Ullman, pp. 195)

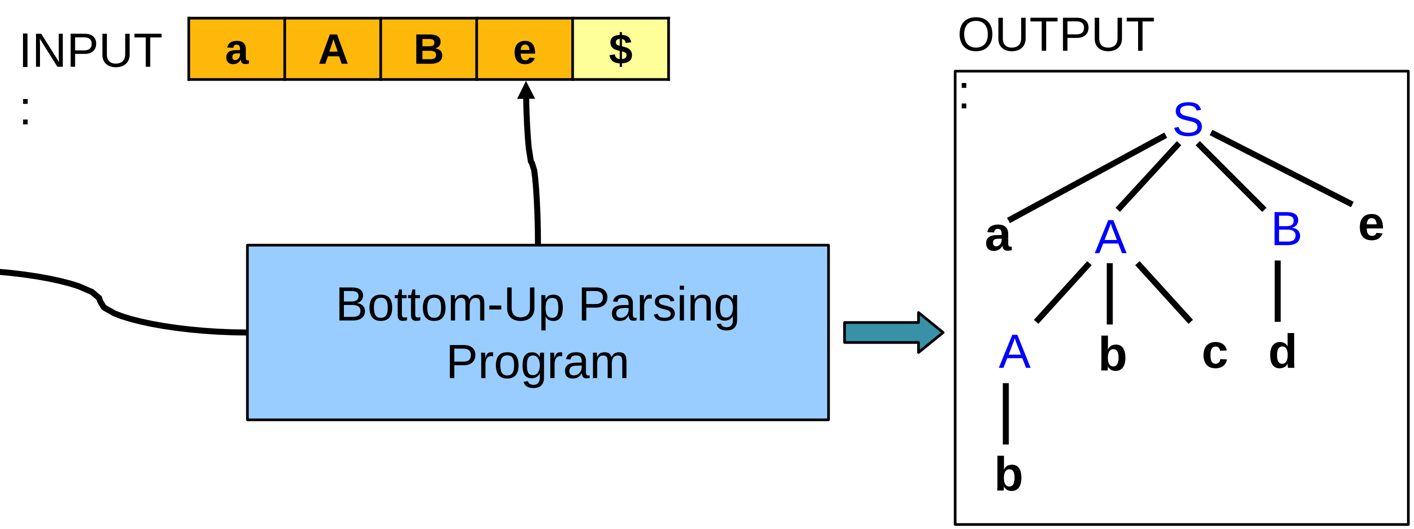
Bottom-Up Parser Example



(Aho,Sethi,Ullman, pp. 195)

Bottom-Up Parser Example

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



(Aho,Sethi,Ullman, pp. 195)

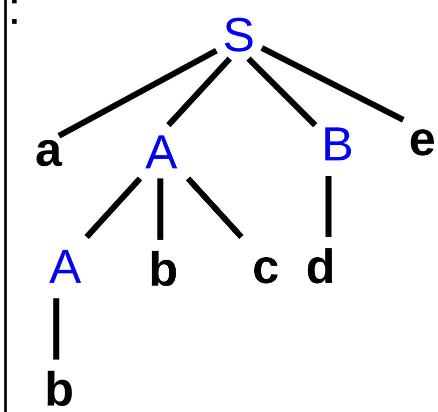
Bottom-Up Parser Example

INPUT:

S	$

Bottom-Up Parsing Program

OUTPUT:



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

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

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

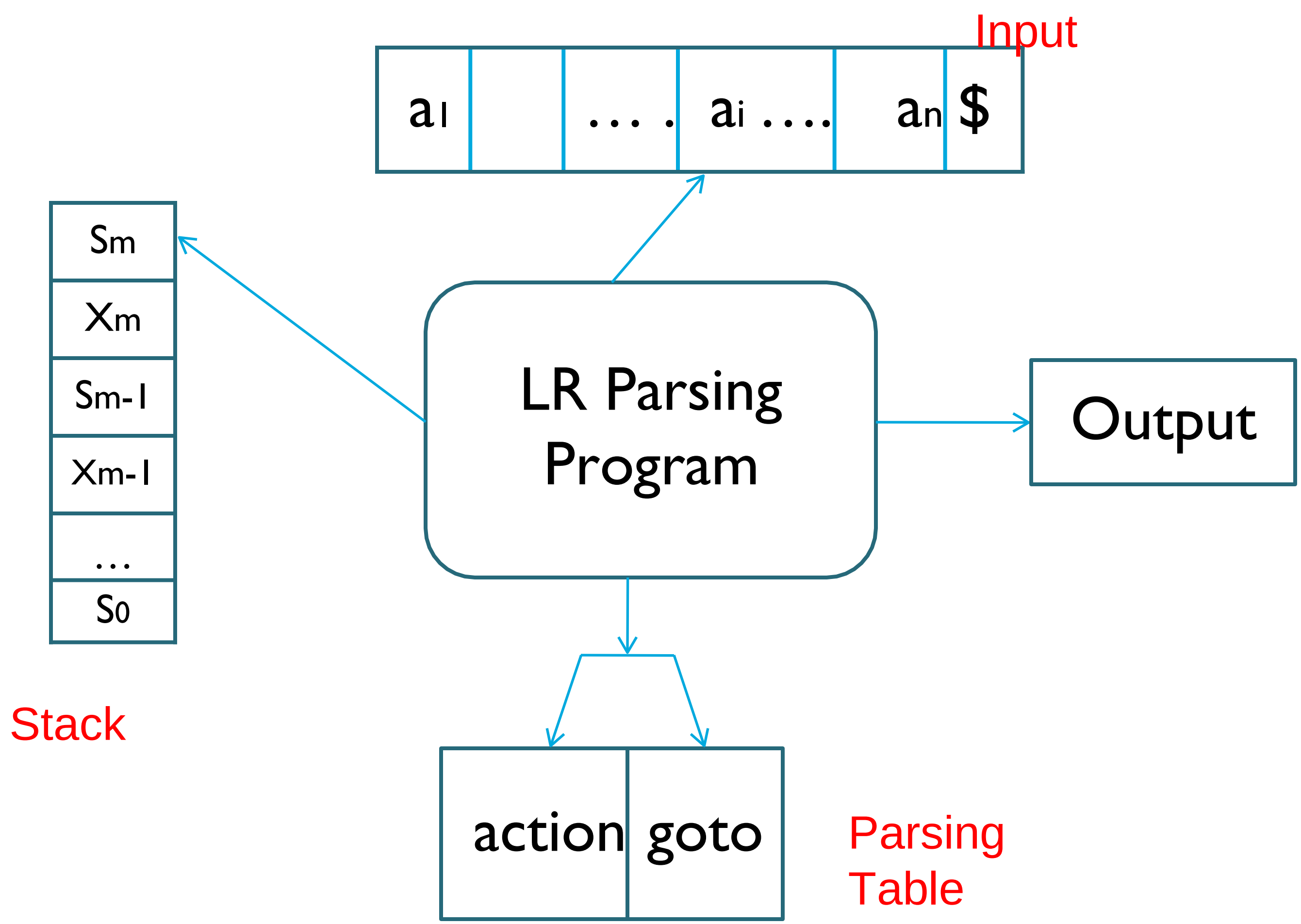
Properties of LR Parsers

- 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:

INPUT:

id	*	id	+	id	$

STACK:

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
(6) F → id
E)

LR Parser Example

INPUT:

id	*****	**id**	**+**	**id**	**$**

STACK:

5
id
0

LR Parsing Program

OUTPUT:

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
(6) F → id
E)

R Parser Example

INPUT:

id	*	id	+	id	$

STACK: 0

LR Parsing Program

OUTPUT:

F

id

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

(Aho,Sethi,Ullman, pp. 220)

GRAMMAR:

(1) E → E + T
(2) E → T
$\boxed{\text{(4) T} \rightarrow \text{F}}$
(5) F → (E)
(6) F → **id**

LR Parser Example

INPUT:

id	*****	**id**	**+**	**id**	**$**

STACK:

3
F
0

LR Parsing Program

OUTPUT:

T
|
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)

LR Parser Example

GRAMMAR:

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

INPUT:

id	*	id	+	id	$

STACK:

0

LR Parsing Program

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

OUTPUT:

T
|
F
|
id

(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**

R Parser Example

INPUT:

id	*	id	+	id	$

STACK:

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			

OUTPUT:

T
|
F
|
id

(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:

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			

OUTPUT:

T
|
F
|
id

(Aho,Sethi,Ullman, pp. 220)

GRAMMAR:

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

LR Parser Example

INPUT:

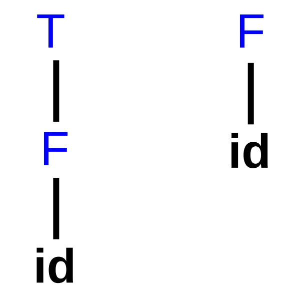
id	*****	**id**	**+**	**id**	**$**

STACK:

5
id
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) E' → T
(3) T → T * F
(4) T → F
(5) F → (...)
(6) F → **id**

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

STACK: 7, *, 2, T, 0

LR Parsing Program

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

OUTPUT:

T
|
F
|
id

F
|
id

(Aho,Sethi,Ullman, pp. 220)

GRAMMAR:

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

LR Parser Example

INPUT:

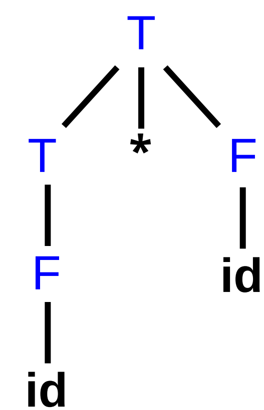
id	*	id	+	id	$

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:

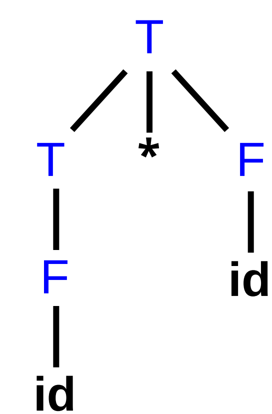
(1) E → E + T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → **id**

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

STACK: | 0 |

LR Parsing Program

OUTPUT:



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

(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	$

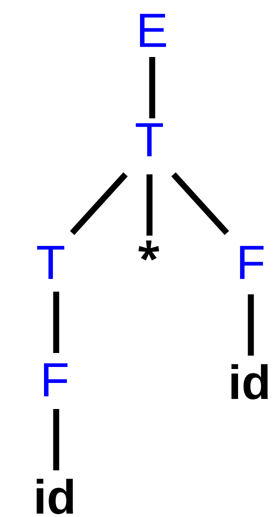
STACK:

2
T
0

LR Parsing Program

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

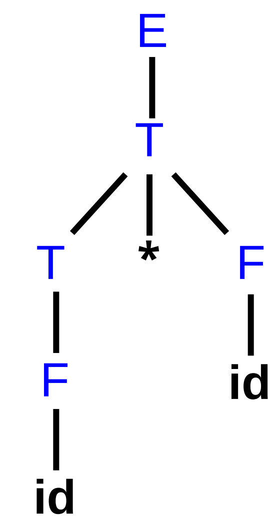
INPUT: | **id** | ***** | **id** | **+** | **id** | **$** |

STACK: | 0 |

LR Parsing Program

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

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:

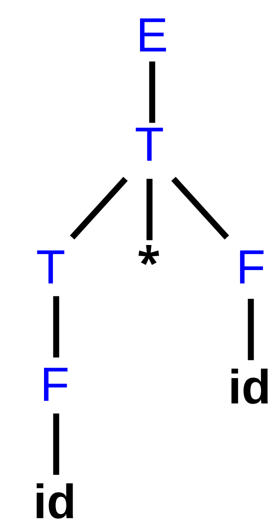
id	*****	**id**	**+**	**id**	**$**

LR Parsing Program

STACK:

1
E
0

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)

GRAMMAR:

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

R Parser Example

INPUT:

id	*	id	+	id	$

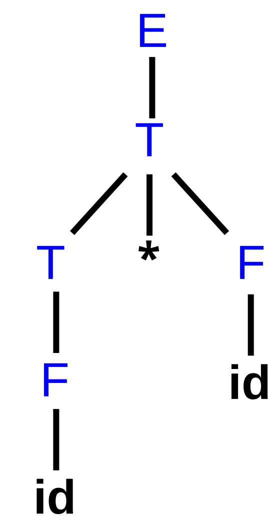
STACK:

6
+
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)

GRAMMAR:

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

R Parser Example

INPUT :

id	*	id	+	id	$

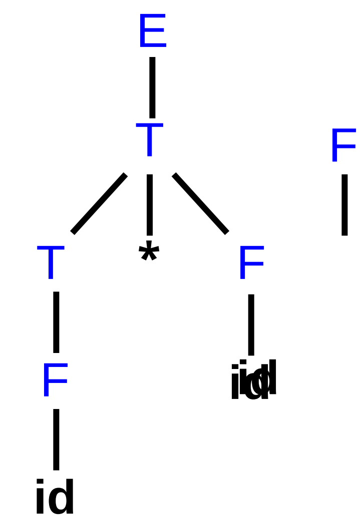
STACK :

5
id
6
+
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)

GRAMMAR:

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

LR Parser Example

INPUT:

id	*	id	+	id	$

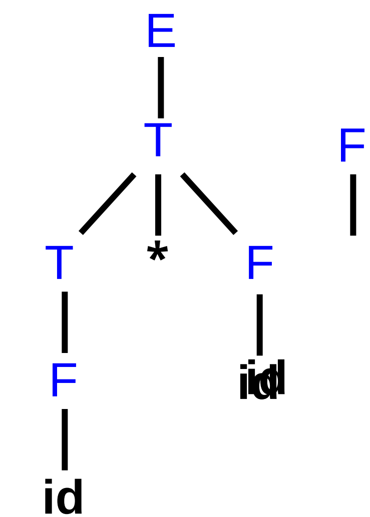
STACK:

6
+
1
E
0

LR Parsing Program

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

OUTPUT:



(Aho,Sethi,Ullman, pp. 220)

GRAMMAR:

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

LR Parser Example

INPUT:

id	*	id	+	id	$

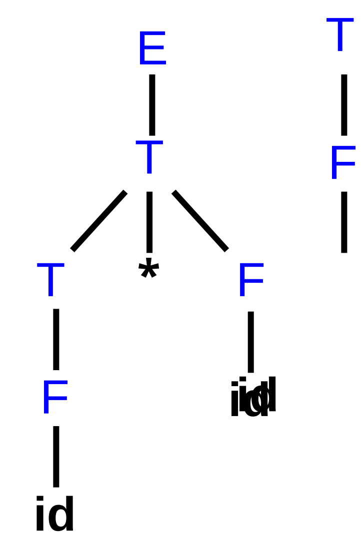
STACK:

3
F
6
+
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)

GRAMMAR:

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

LR Parser Example

INPUT:

id	*	id	+	id	$

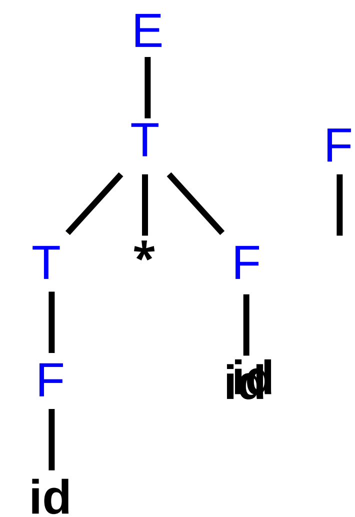
STACK:

6
+
1
E
0

LR Parsing Program

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

OUTPUT:



(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	$

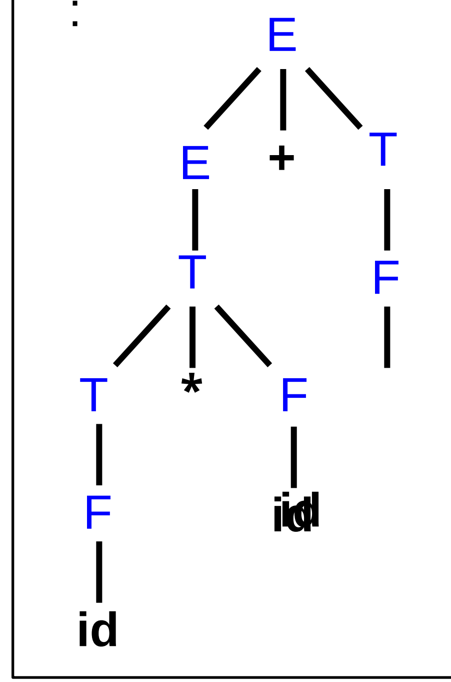
STACK:

9
T
6
+
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)

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	$

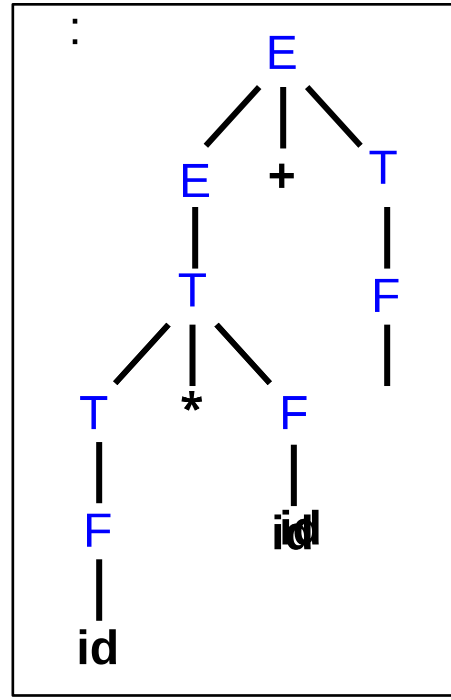
STACK:

0

LR Parsing Program

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

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**

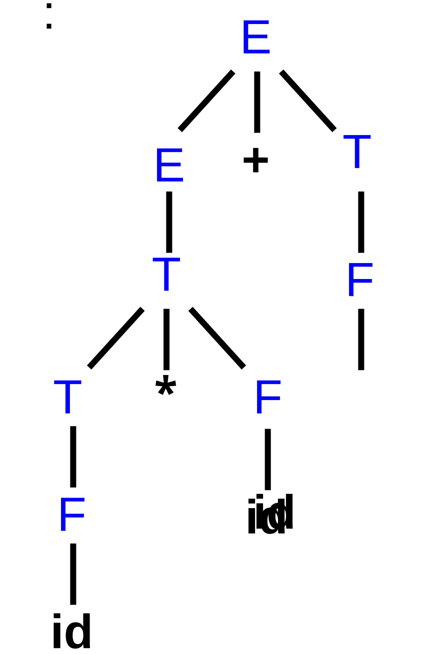
INPUT: | **id** | ***** | **id** | **+** | **id** | **$** |

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 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

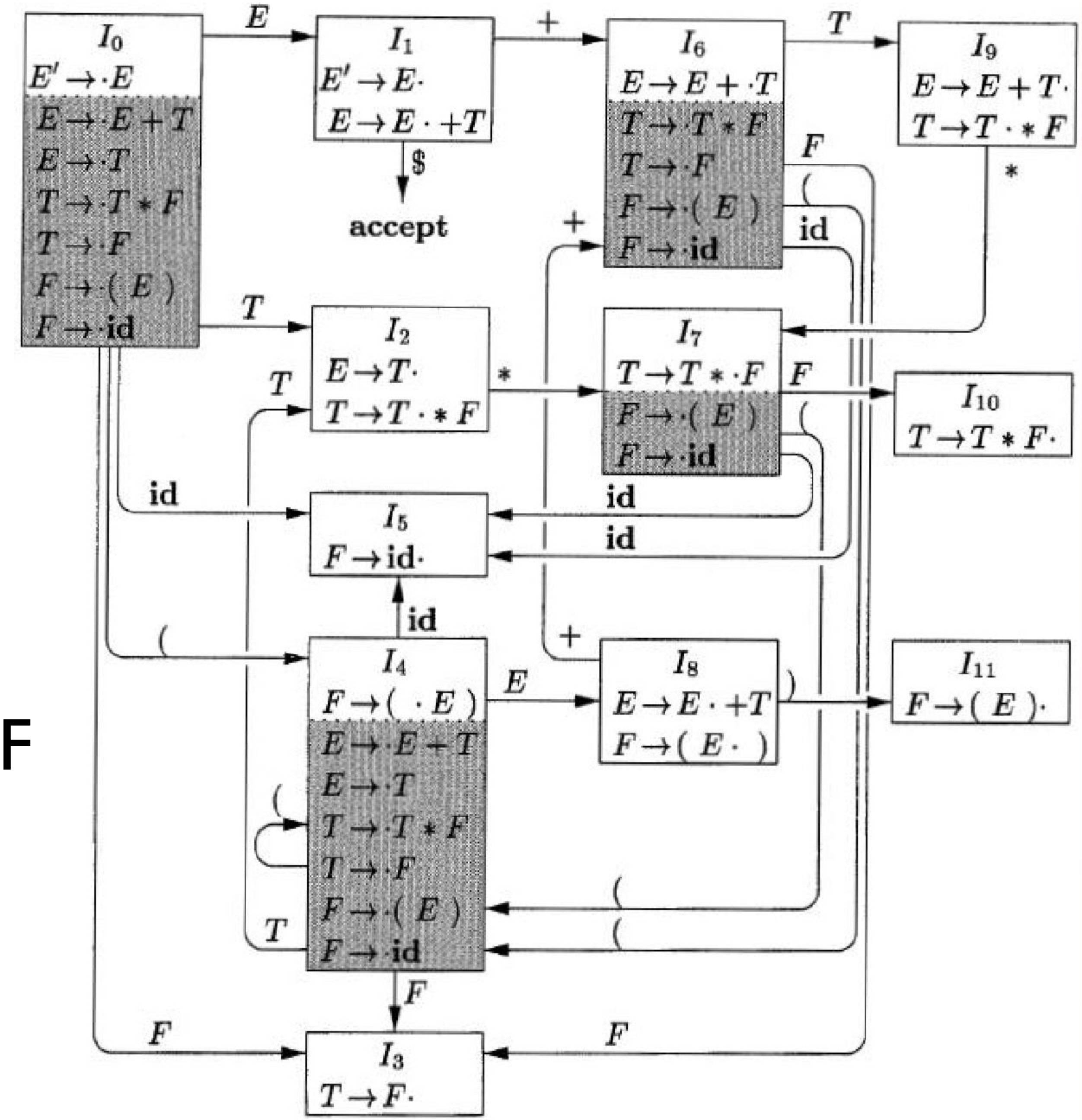
I8 =F -> (E.)
E-> E.+T

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

I10 = T -> T*F.

I11 =F -> (E).

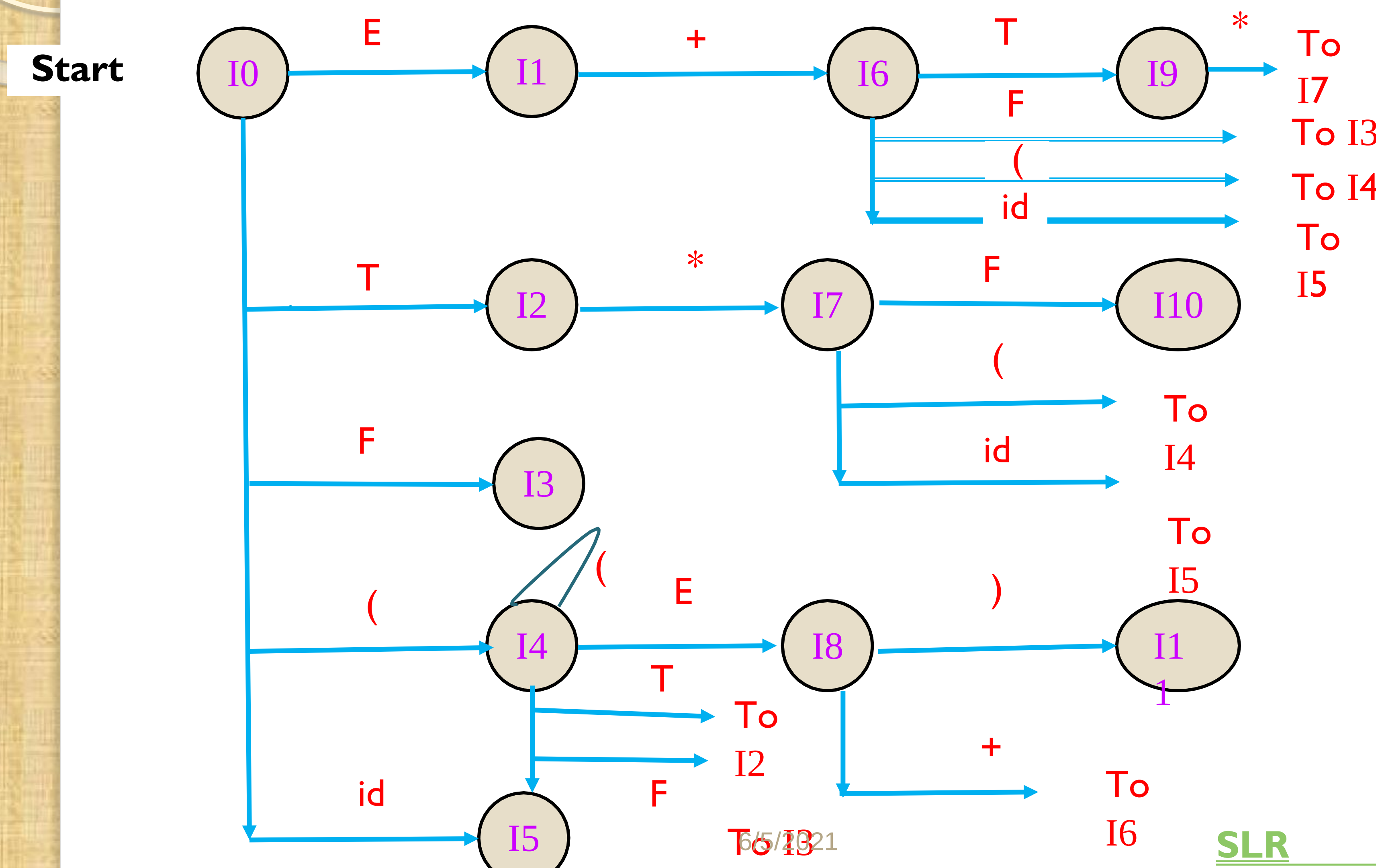
- E' -> E
- E -> E + T | T
- T -> T * F | F
- F -> (E) | id



DFA . . .

. . .

Each State is Final State



SLR

SLR Parsing Table

- 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.

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 **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]

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.The 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.

Canonical LR Parsing Table

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

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,

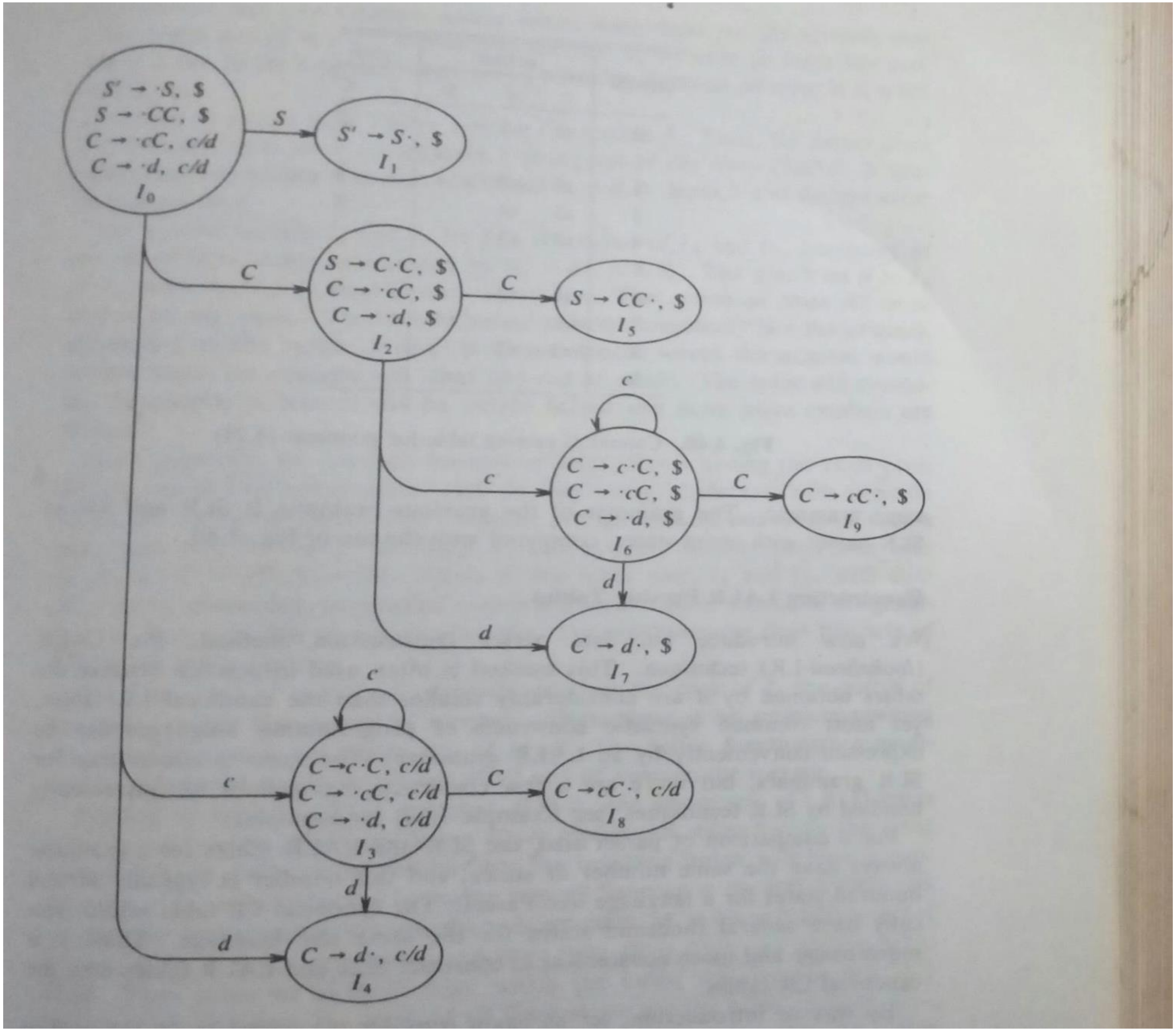
such that goto(I, X) is not empty and not in C

do add goto(I, X) to C.

Until more items can be added to C

end.

Example….


S' → ·S, $
S → ·CC, $
C → ·cC, c/d
C → ·d, c/d
I0
S
S' → S·, $
I1
C
S → C·C, $
C → ·cC, $
C → ·d, $
I2
C
S → CC·, $
I5
c
C → c·C, $
C → ·cC, $
C → ·d, $
I6
C
C → cC·, $
I9
d
d
C → d·, $
I7
c
c
C→c·C, c/d
C → ·cC, c/d
C → ·d, c/d
I3
C
C →cC·, c/d
I8
d
d
C → d·, c/d
I4

CLR Parsing Table

- 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.

Continue…

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, $].

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.*

LALR Parsing Table

- 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 U I2 U … U 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 CLR Vs LALR
- SLR Vs

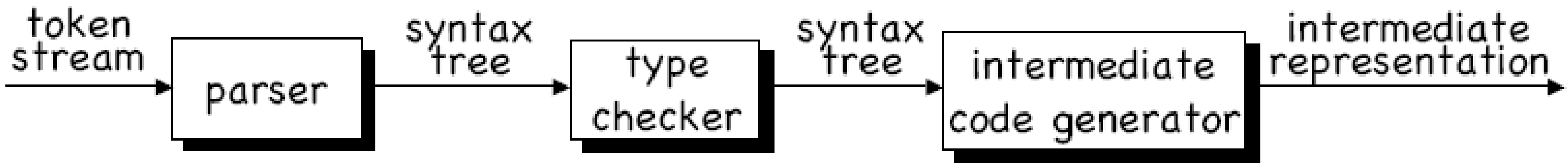
Semantic Analysis

- 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.