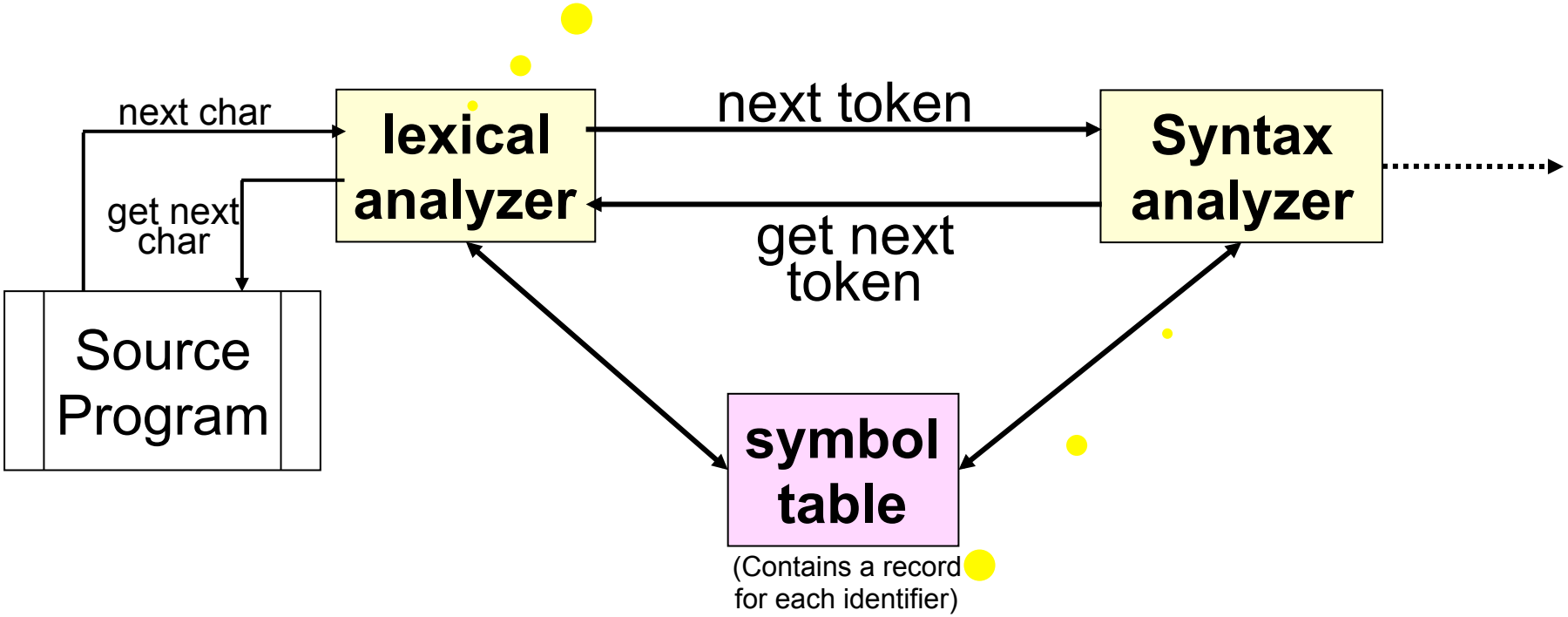
Language Processing Systems

Prof. Mohamed Hamada

Software Engineering Lab.
The University of Aizu
Japan

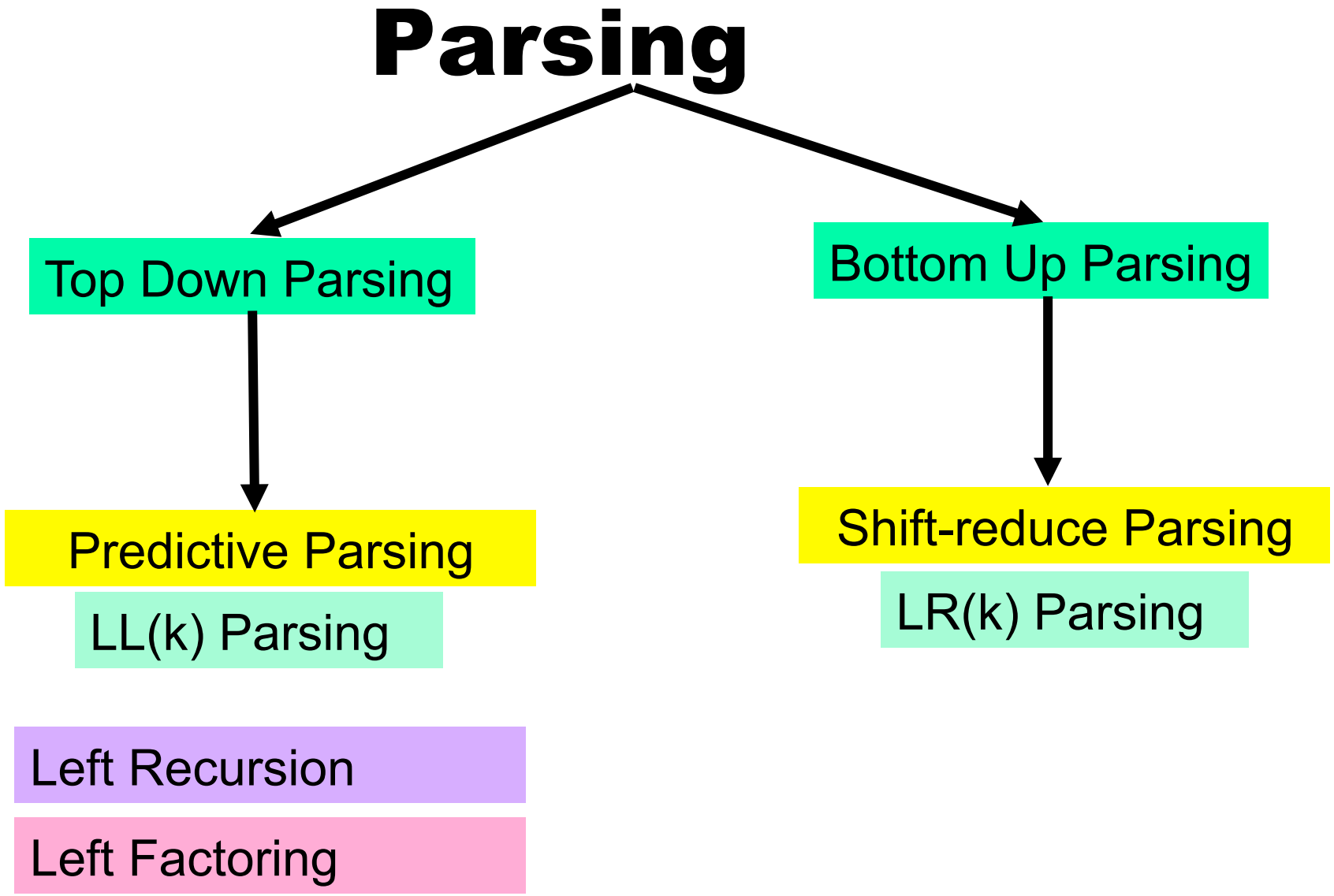
Syntax Analysis
(Parsing)

1. Uses Regular Expressions to define tokens
2. Uses Finite Automata to recognize tokens

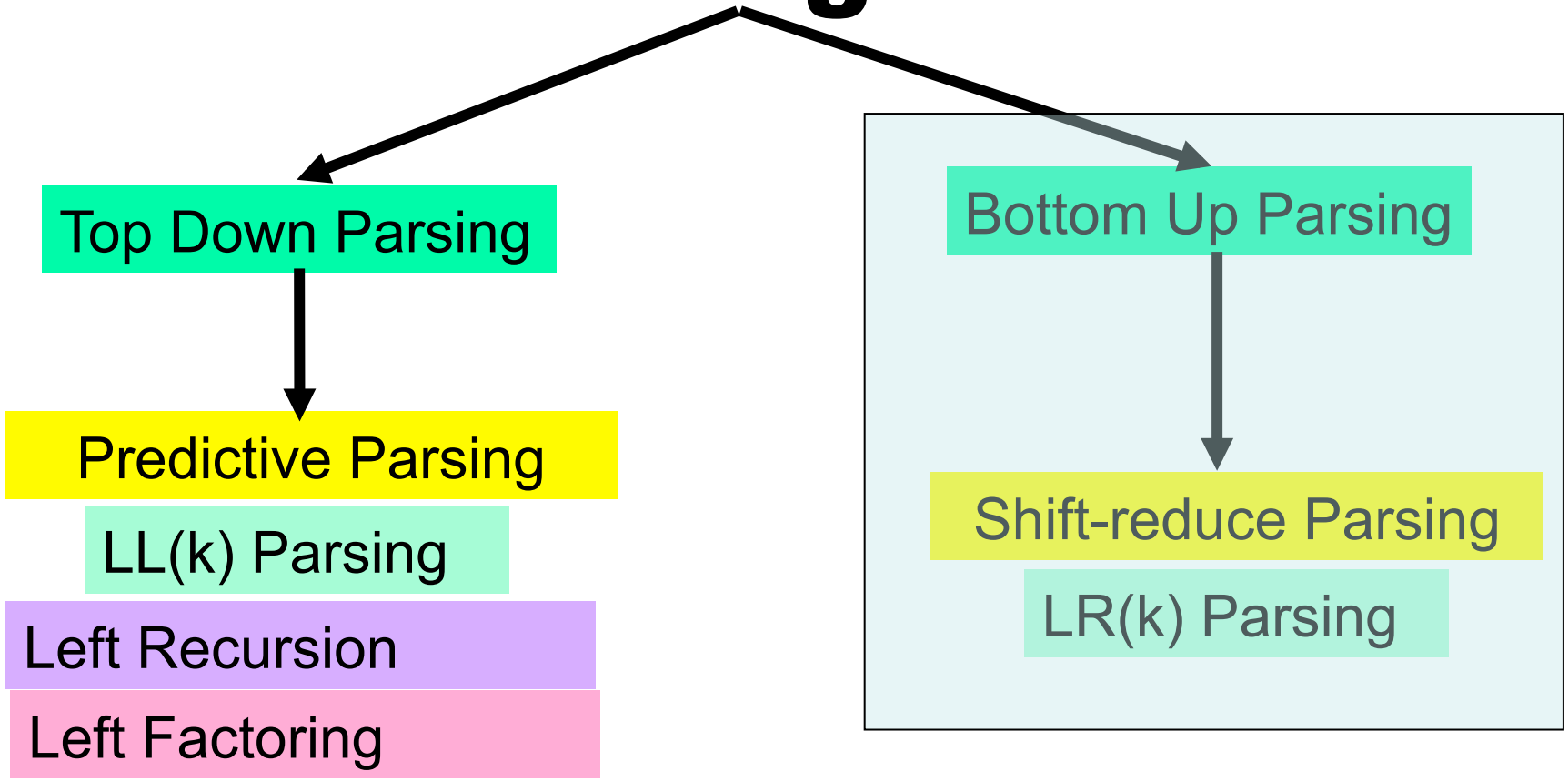


Uses Top-down parsing or Bottom-up parsing

To construct a Parse tree


Parsing
Top Down Parsing
Bottom Up Parsing
Predictive Parsing
LL(k) Parsing
Shift-reduce Parsing
LR(k) Parsing
Left Recursion
Left Factoring

Parsing



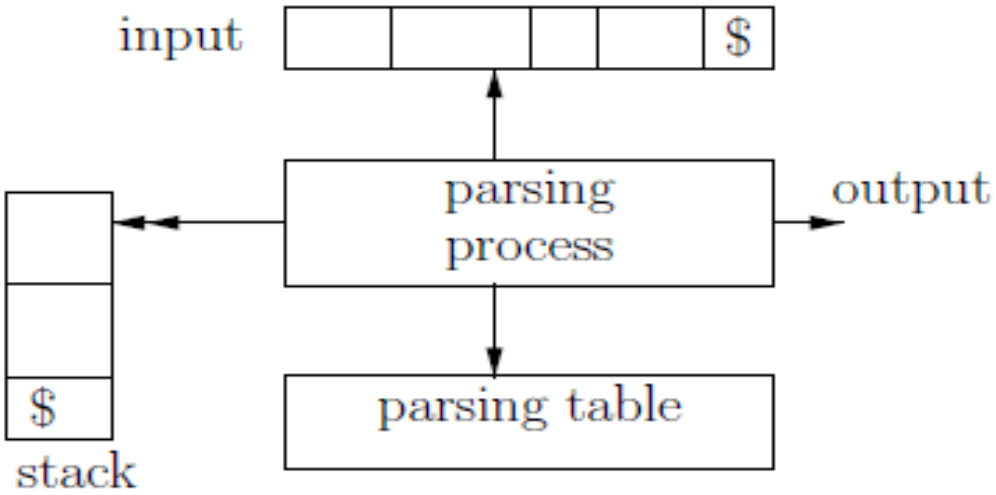
Top-down parsers: starts constructing the parse tree at the top (root) of the tree and move down towards the leaves.
Easy to implement by hand, but work with restricted grammars.
Example: predictive parsers

A Predictive Parser

How it works?

1. Construct the *parsing table* from the given grammar

2. Apply the predictive parsing algorithm to construct the parse tree



A Predictive Parser

1. Construct the *parsing table* from the given grammar

The following algorithm shows how we can construct the *parsing table*:

Input: a grammar G

Output: the corresponding parsing table M

Method: For each production A → α of the grammar do the following steps:

1. For each terminal a in FIRST(α), add A → α to M[A,a].
2. If λ in FIRST(α), add A → α to M[A,b] for each terminal b in FOLLOW(A).
3. If λ FIRST(α) and $ in FOLLOW(A), add A → α to M[A,$]

A Predictive Parser

2. Apply the predictive parsing algorithm to construct the parse tree

The following algorithm shows how we can construct the move parsing table for an input string w$ with respect to a given grammar G.

set *ip* to point to the first symbol of the input string *w*$

repeat

if Top(stack) is a terminal or $ then
if Top(stack) = Current-Input(*ip*) then
Pop(stack) and advance *ip*
else null
else
if M[X,a]= X→ $Y_1Y_2 \ldots Y_k$ then
begin
Pop(stack);
Push Y_1; Y_2;… ; Y_k onto the stack, with Y_1 on top;
Output the production X→ $Y_1Y_2 \ldots Y_k$
end
else null

until Top(stack) = $ (i.e. the stack become empty)

A Predictive Parser

2. Apply the predictive parsing algorithm to construct the parse tree

Example

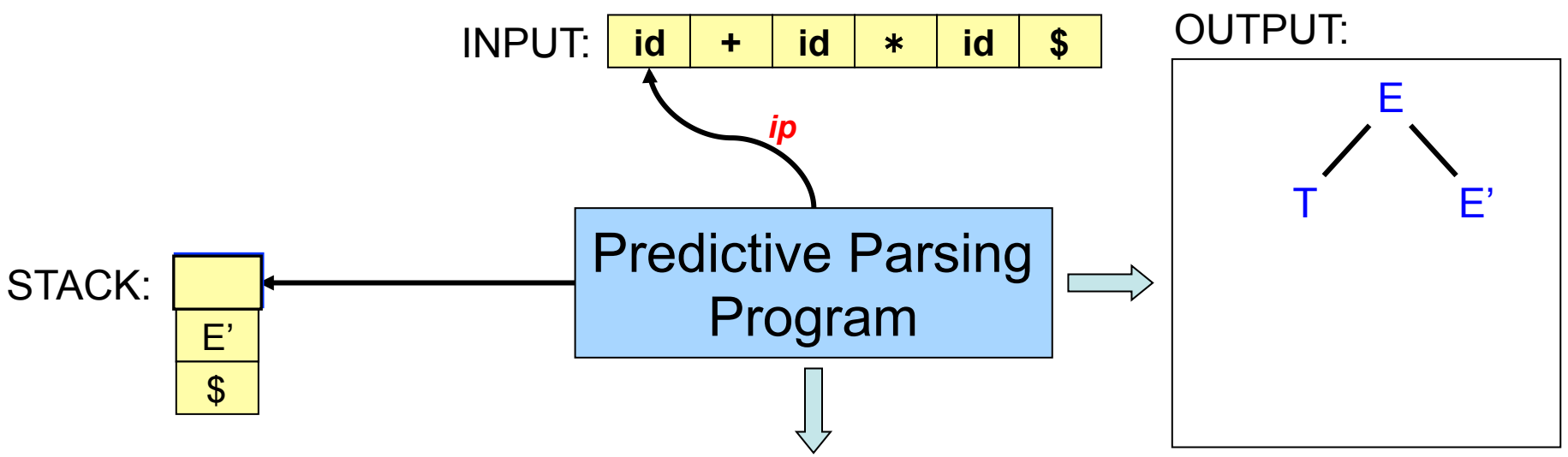
Grammar:

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

Parsing Table:

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

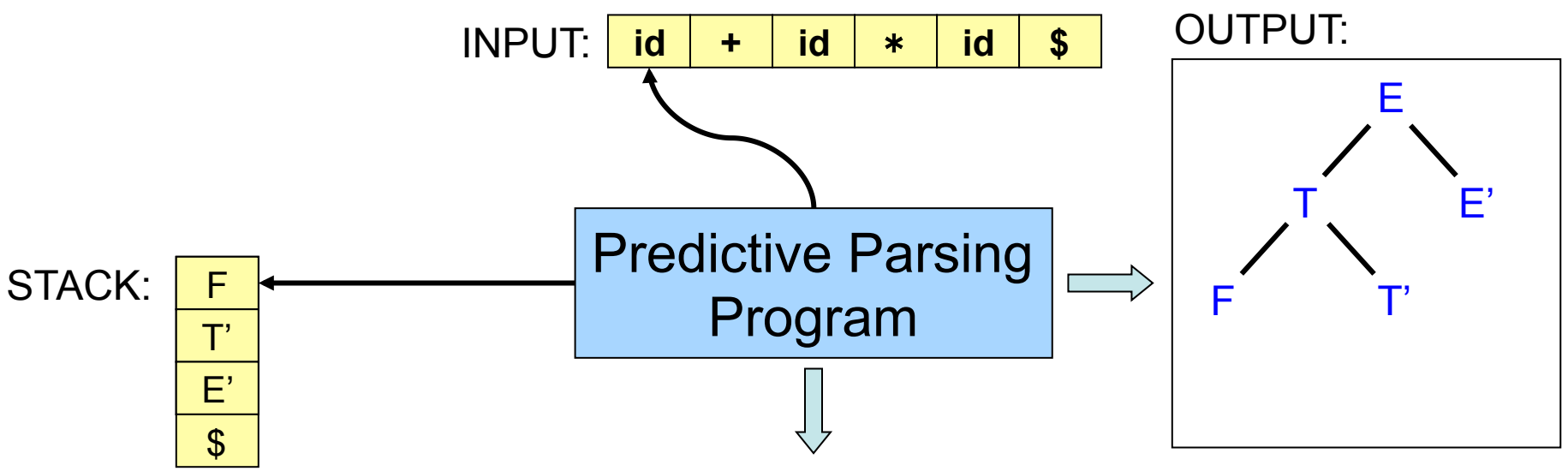
```
Set ip to point to the first symbol of the input string w$
repeat
    if Top(stack) is a terminal or $ then
        if Top(stack) = Current-Input(ip) then
            Pop(stack) and advance ip
        else else null
    else if M[X,a]= X→ Y1Y2 …Yk then
        begin
            Pop(stack);
            Push Y1; Y2;… ; Yk onto the stack, with Y1 on top;
            Output the production Y1; Y2;… ; Yk ;
        end
    else null
until Top(stack)=$ (i.e. the stack become empty)
```



PARSING TABLE:

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

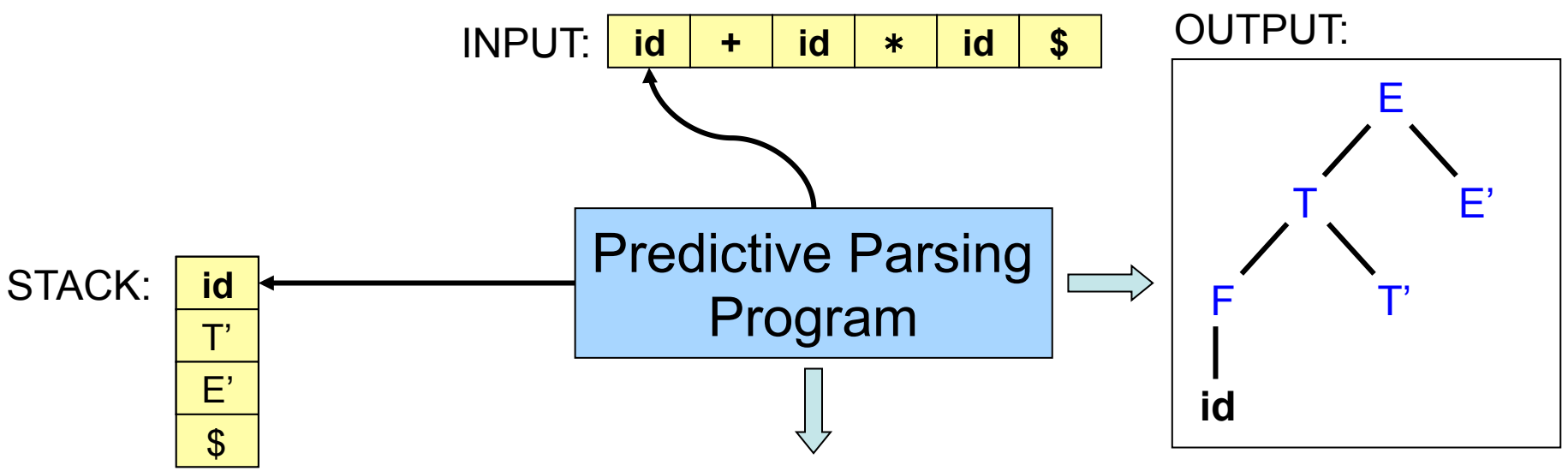
A Predictive Parser



PARSING TABLE:

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

A Predictive Parser

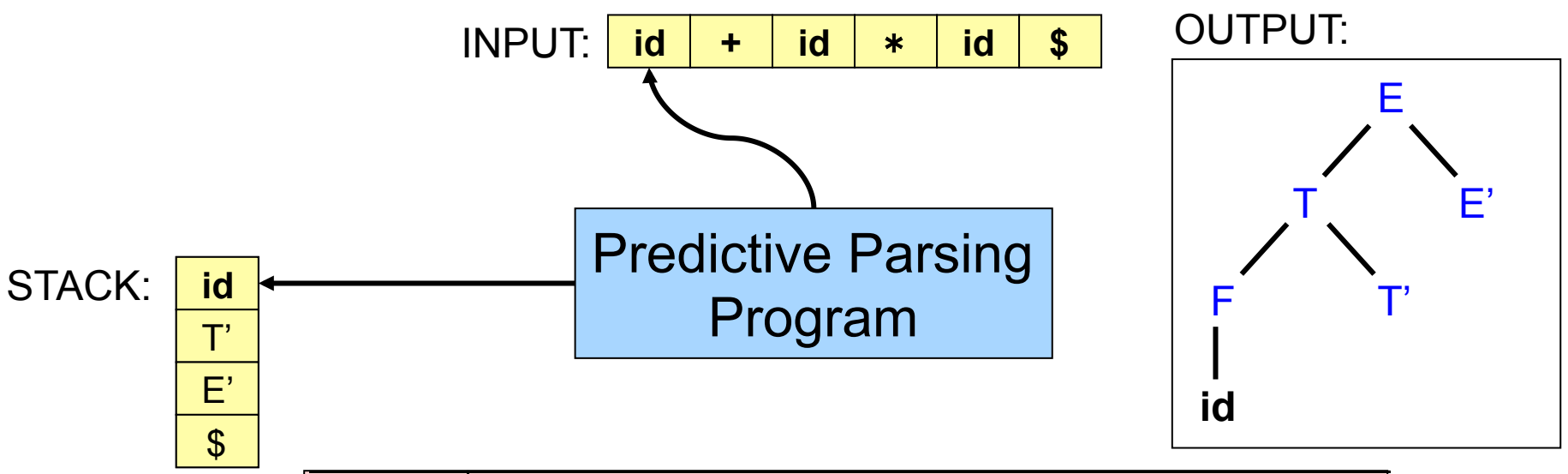


PARSING TABLE:

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

A Predictive Parser

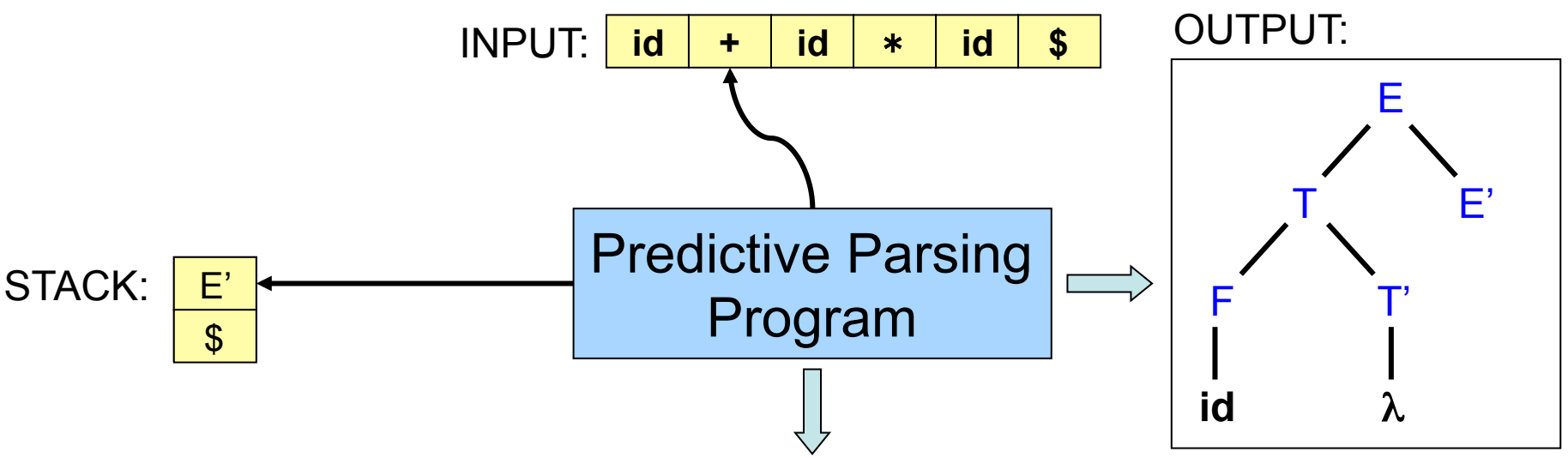
Action when Top(Stack) = input ≠ $: Pop stack, advance input.



PARSING TABLE:

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

A Predictive Parser



PARSING TABLE:

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

A Predictive Parser

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

E' → +TE'

T → FT'

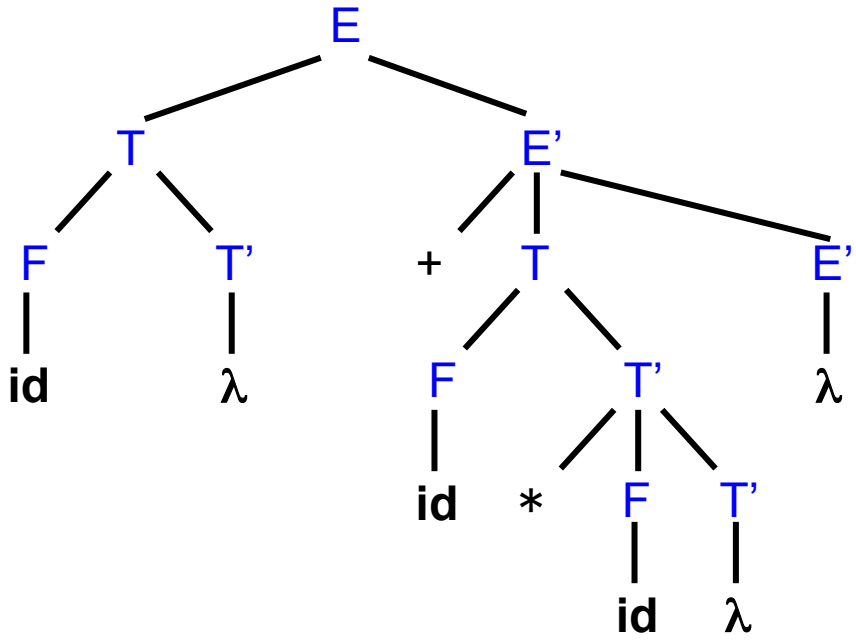
F → **id**

T' → * FT'

F → **id**

T' → λ

E' → λ



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.

LL(1) A grammar whose parsing table has no multiply-defined entries

LL(1) grammars enjoys several nice properties: for example they are not ambiguous and not left recursive.

LL(1) A grammar whose parsing table has no multiply-defined entries

Example 1 The grammar

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \lambda \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \lambda \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

Whose PARSINGTABLE:

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

Is **LL(1)** grammar

LL(1) A grammar whose parsing table has no multiply-defined entries

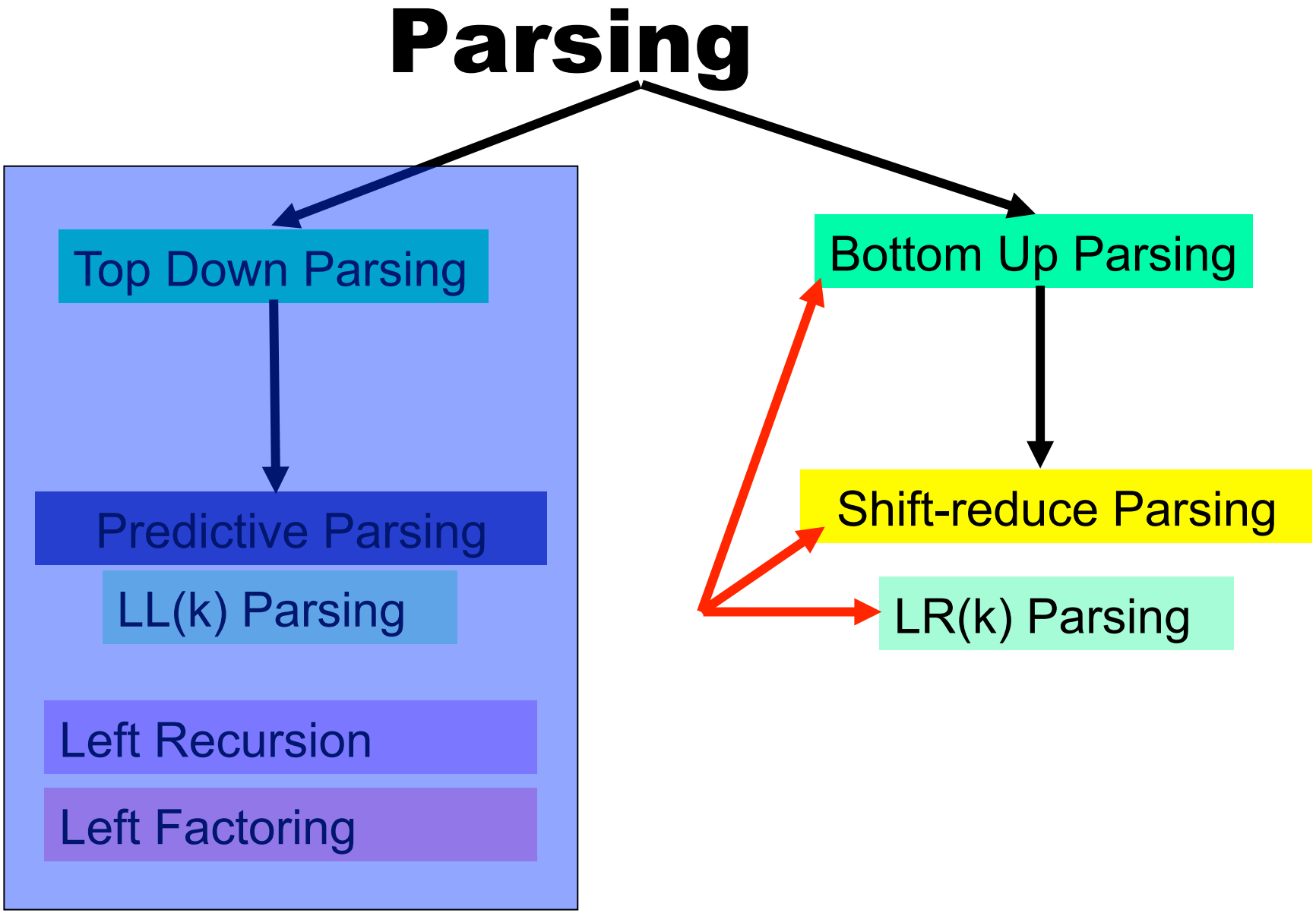
Example 2 The grammar

$S \rightarrow iEtSS` \mid a$
$S' \rightarrow eS \mid \lambda$
$E \rightarrow Fb$

Whose PARSINGTABLE:

NON-TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	$
S	S→ a			S → iEtSS′		
S′			S′ → λ S′ →eS			S′ → λ
E		E →b				

Is NOT **LL(1)** grammar


Parsing
Top Down Parsing
Predictive Parsing
LL(k) Parsing
Left Recursion
Left Factoring
Bottom Up Parsing
Shift-reduce Parsing
LR(k) Parsing

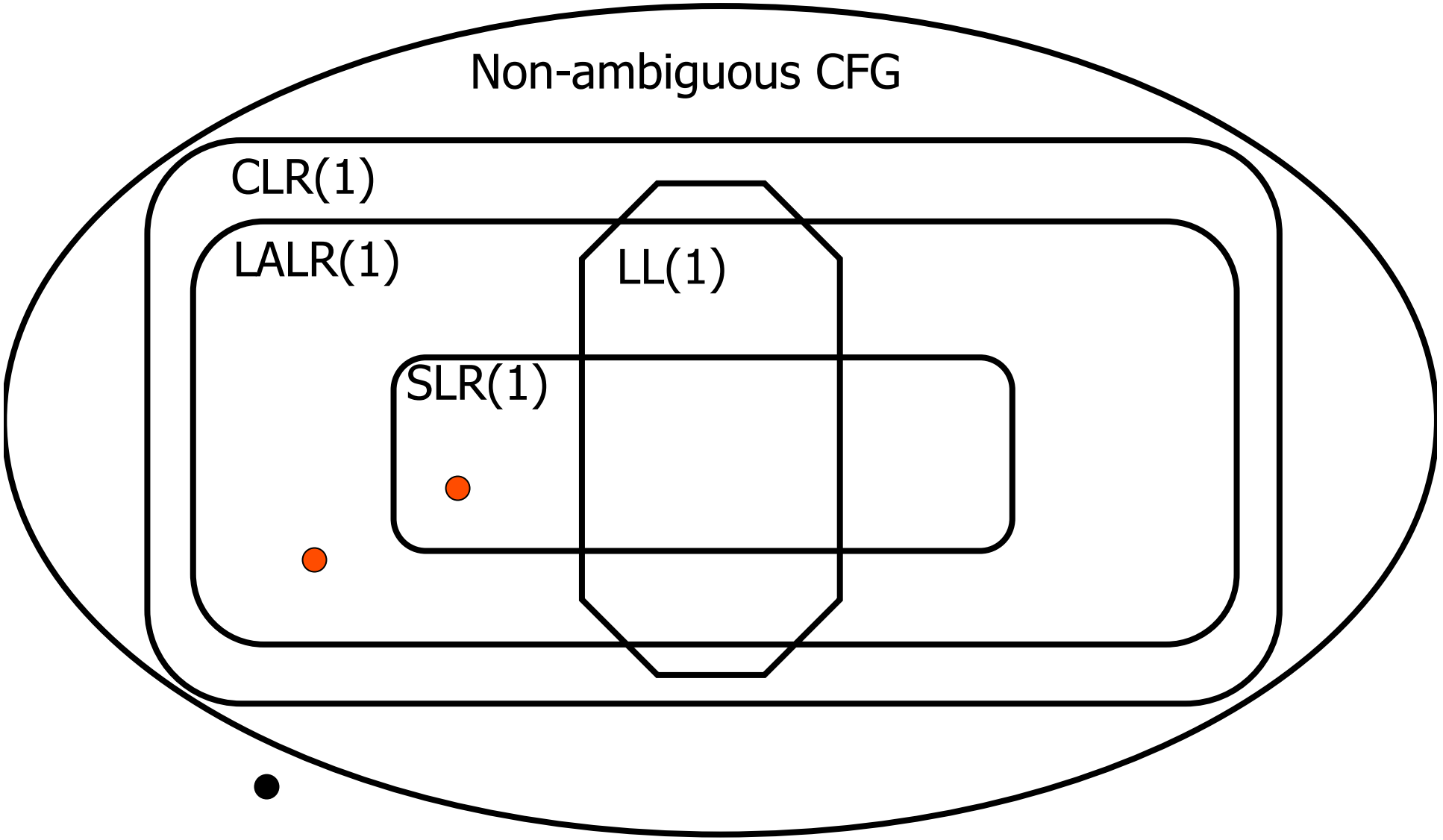
Bottom-Up Parsers

Bottom-up parsers: build the nodes on the bottom of the parse tree first. Suitable for automatic parser generation, handle a larger class of grammars.
Examples: shift-reduce parser (or LR(*k*) parsers)

Bottom-up Parsing

- No problem with left-recursion
- Widely used in practice
- LR(1), SLR(1), LALR(1)

Grammar Hierarchy



Bottom-up Parsing

- Works from tokens to start-symbol
- Repeat:
  - identify handle - reducible sequence:
    - non-terminal is not constructed but
    - all its children have been constructed
  - reduce - construct non-terminal and update stack
- Until reducing to start-symbol

Bottom-up Parsing

1 + (2) + (3)

E + (2) + (3)

E + (E) + (3)

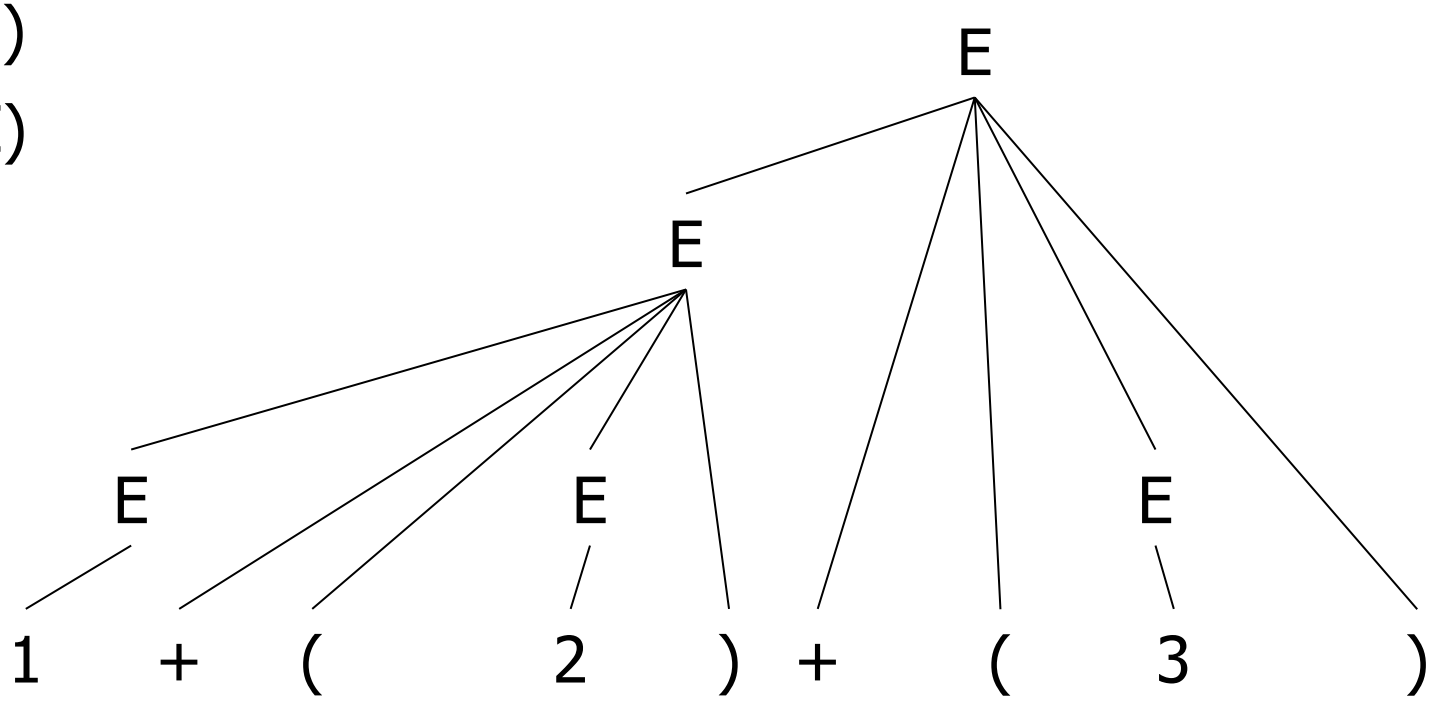
E + (3)

E + (E)

E

$E \rightarrow E + (E)$

$E \rightarrow i$ $\quad i = 0,1, 2, \ldots, 9$



Bottom-up Parsing

- Is the following grammar LL(1) ?

$$E \rightarrow E + (E)$$
$$E \rightarrow i$$

  - NO

1 + (2)

1 + (2) + (3)

  - But this is a useful grammar

Bottom-Up Parser

A bottom-up parser, or a shift-reduce parser, begins at the leaves and works up to the top of the tree.

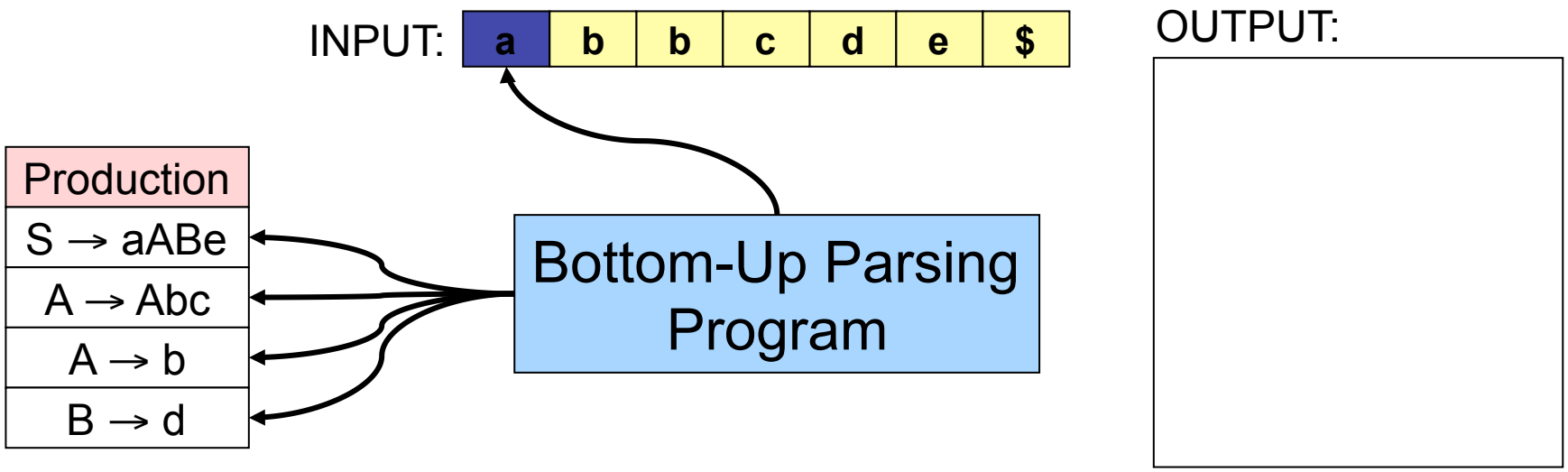
The reduction steps trace a rightmost derivation on reverse.

Consider the Grammar:

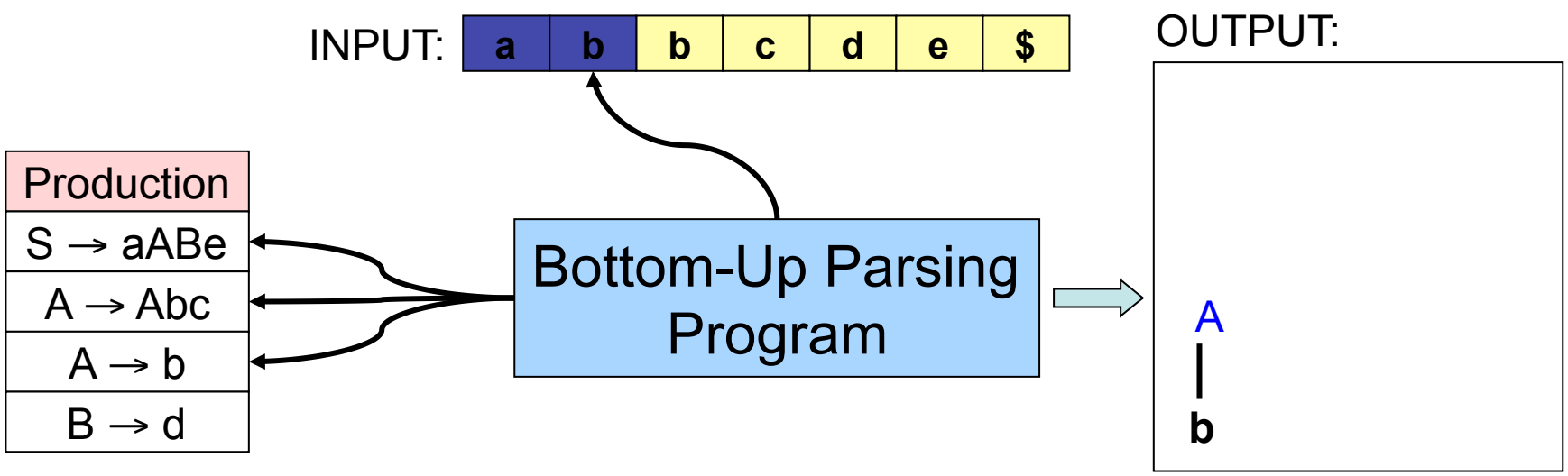
S → aABe
A → Abc | b
B → d

We want to parse the input string abbcde.

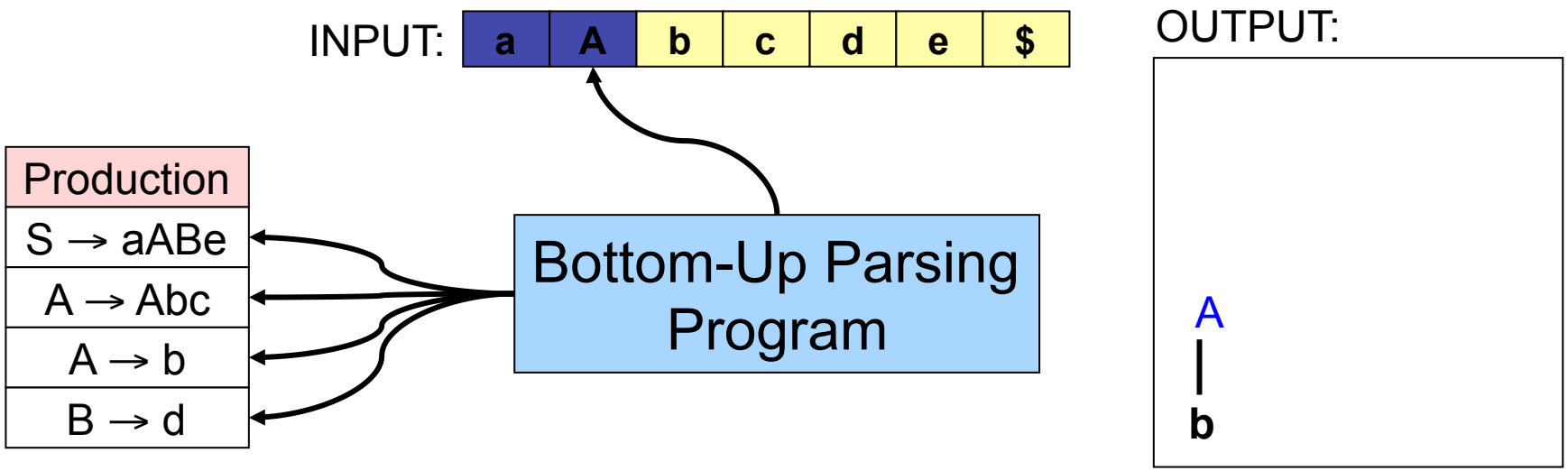
Bottom-Up Parser Example



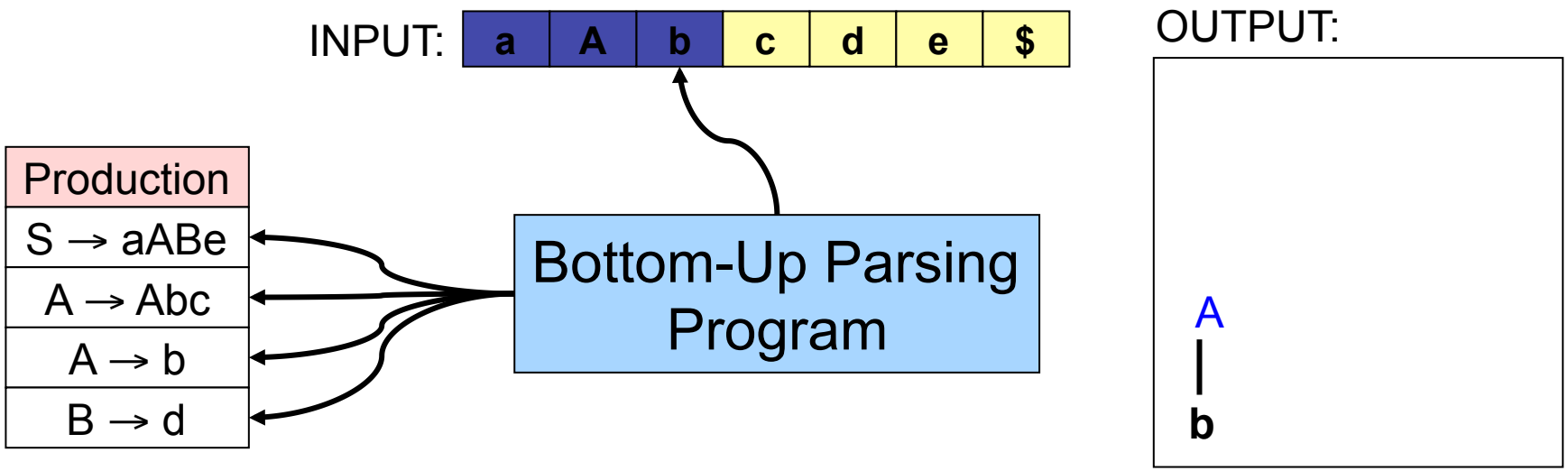
Bottom-Up Parser Example



Bottom-Up Parser Example



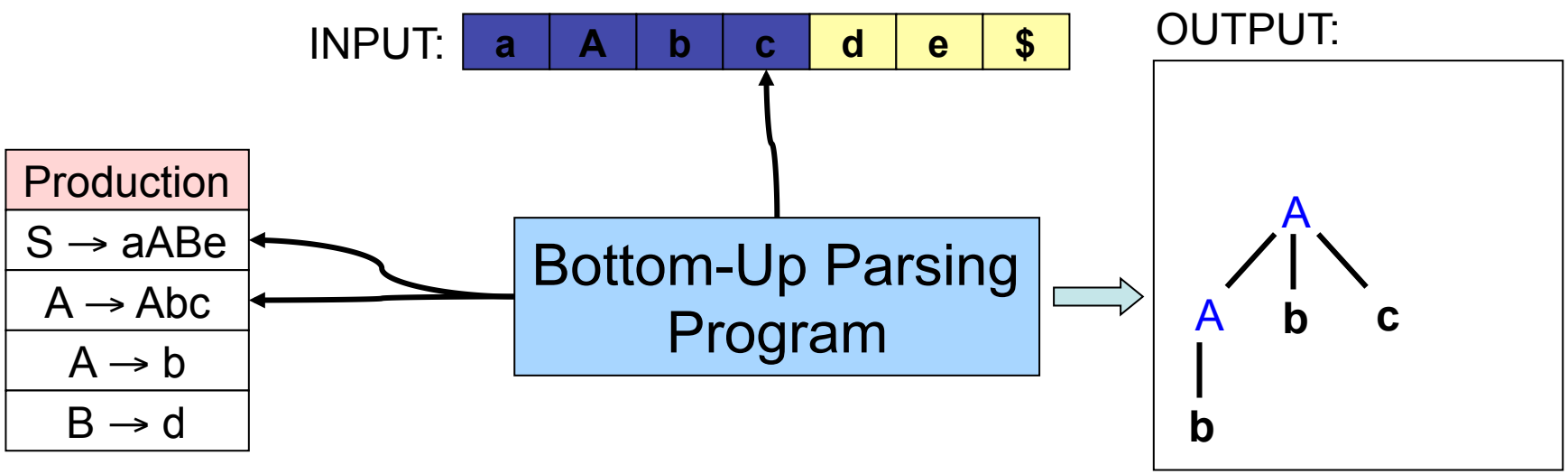
Bottom-Up Parser Example



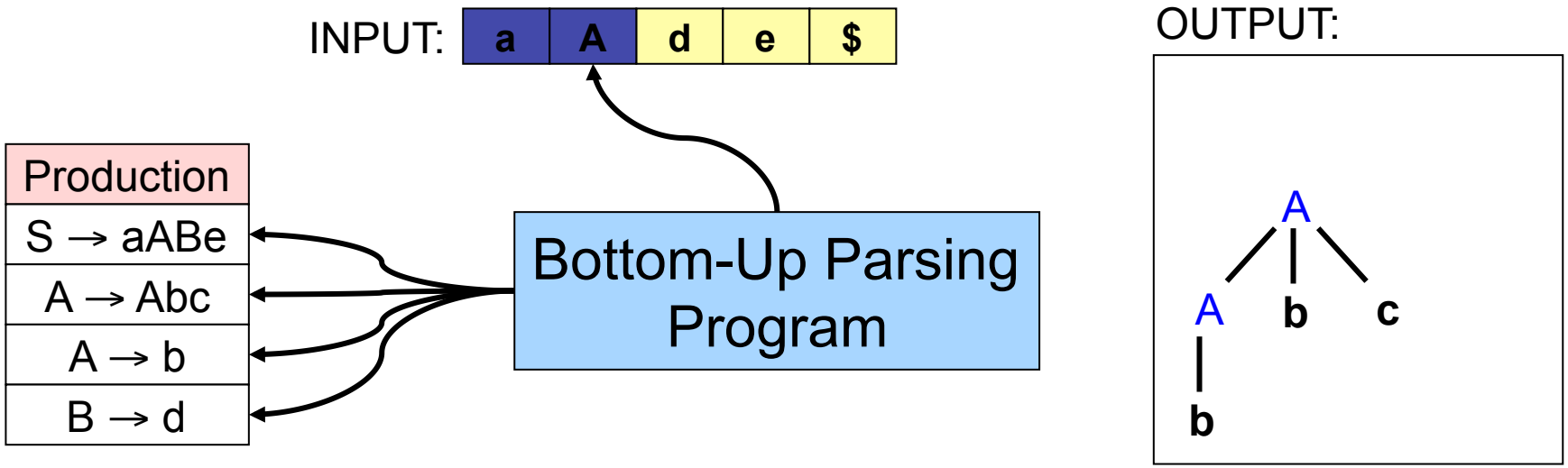
We are not reducing here in this example.

A parser would reduce, get stuck and then backtrack!

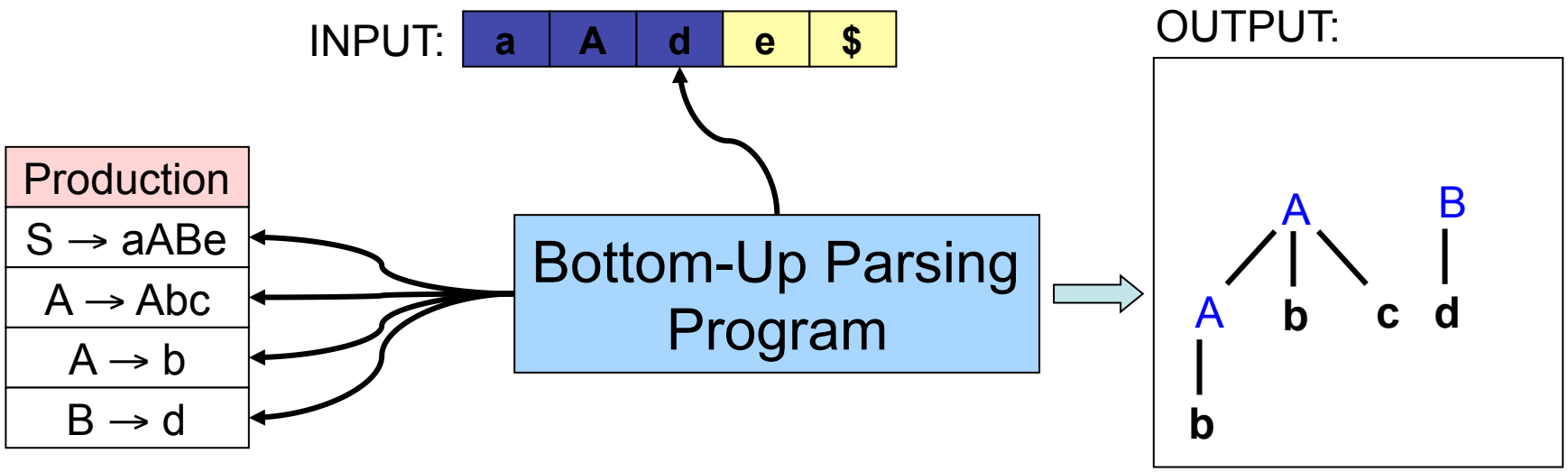
Bottom-Up Parser Example



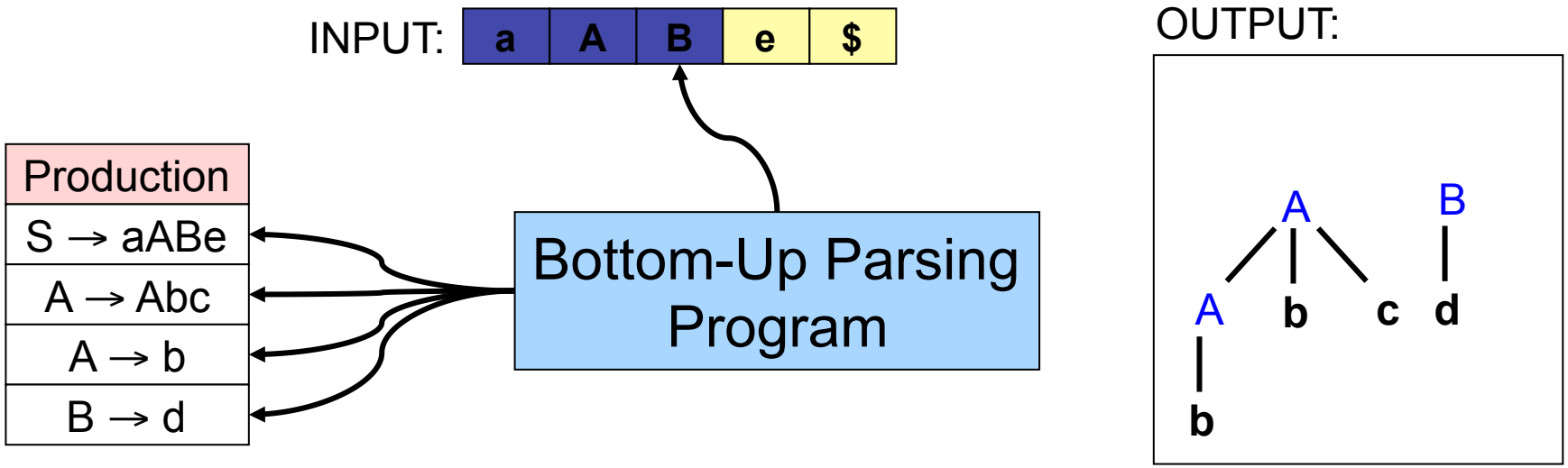
Bottom-Up Parser Example



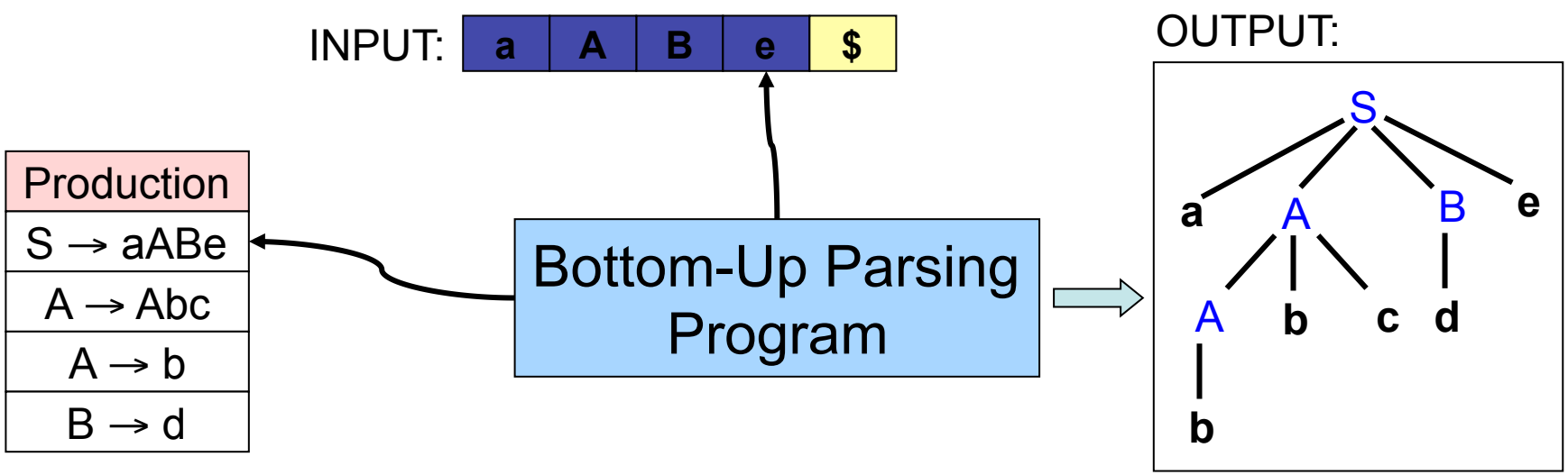
Bottom-Up Parser Example



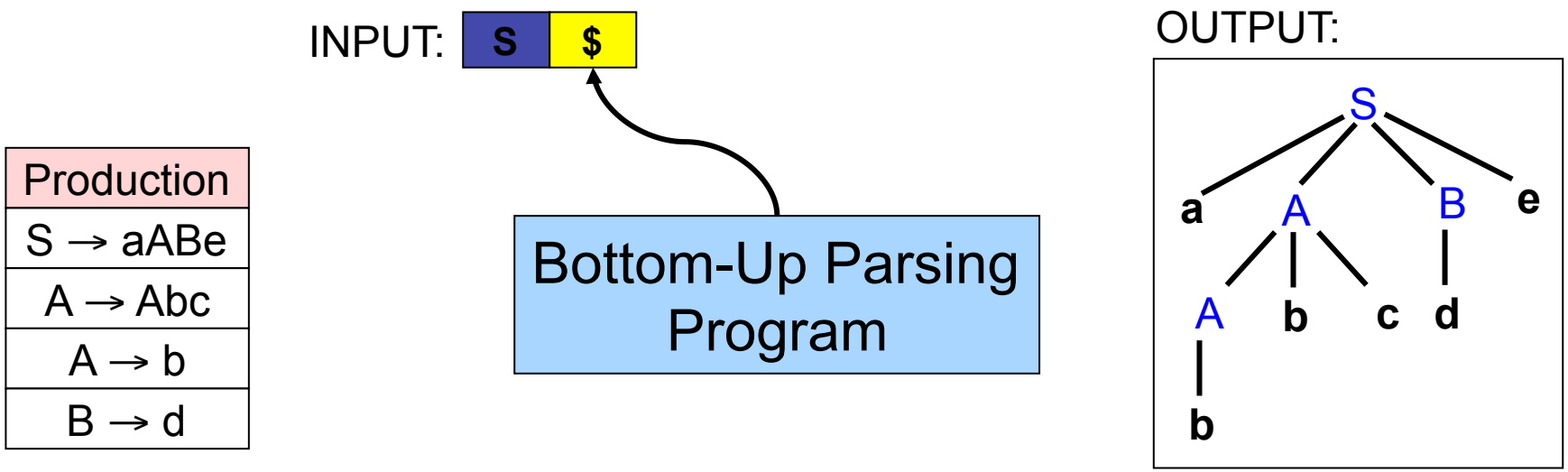
Bottom-Up Parser Example



Bottom-Up Parser Example



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.

Bottom-Up Parser Example

The scanning of productions for matching with handles in the input string, and backtracking makes the method used in the previous example very inefficient.

Can we do better?

See next lecture