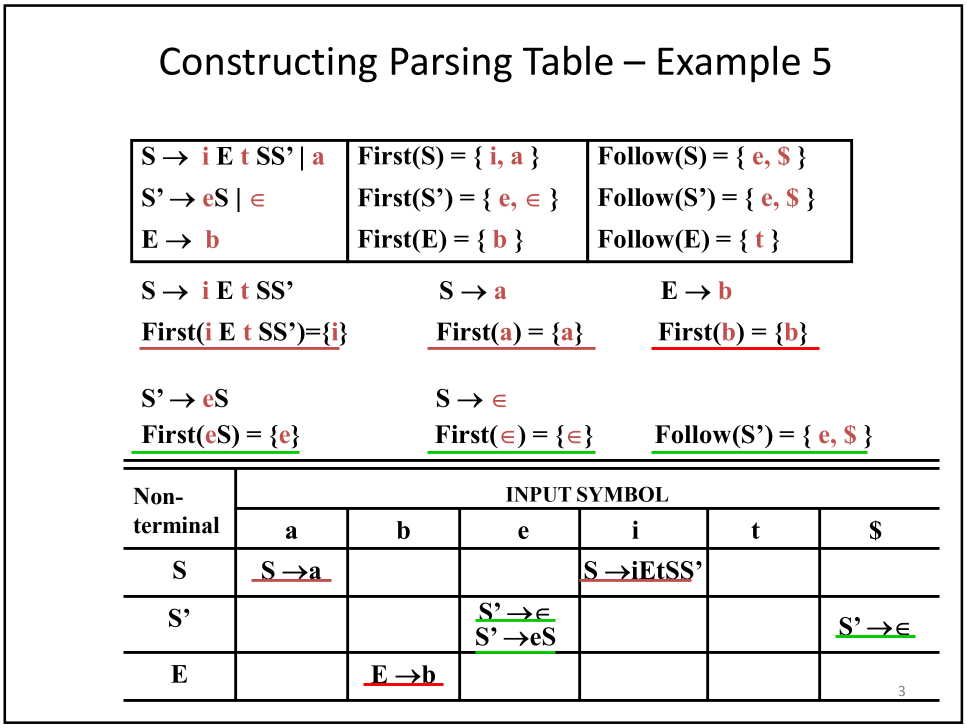
Chapter 4
Syntax Analysis

Constructing Parsing Table – Example 5

S → i E t SS' | a	First(S) = { i, a }	Follow(S) = { e, $ }
S' → eS | ∈	First(S') = { e, ∈ }	Follow(S') = { e, $ }
E → b	First(E) = { b }	Follow(E) = { t }



Constructing Parsing Table – Example 5

S → i E t SS' | a	First(S) = { i, a }	Follow(S) = { e, $ }
S' → eS | ∈	First(S') = { e, ∈ }	Follow(S') = { e, $ }
E → b	First(E) = { b }	Follow(E) = { t }

S → i E t SS'
First(i E t SS')={i}

S → a
First(a) = {a}

E → b
First(b) = {b}

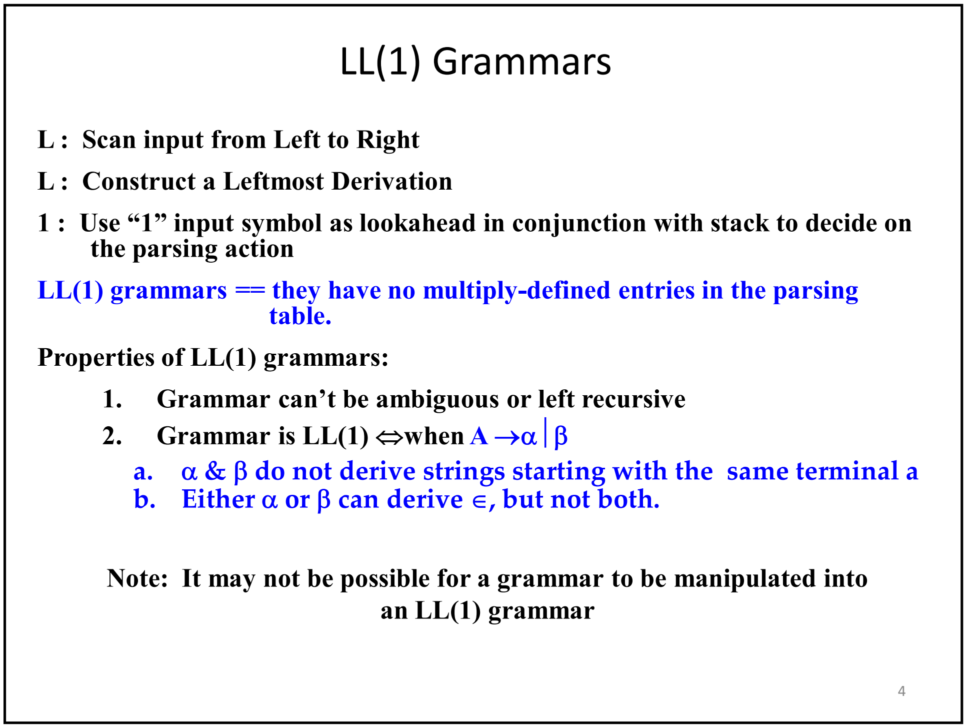
S' → eS
First(eS) = {e}

S → ∈
First(∈) = {∈}

Follow(S') = { e, $ }

Non-terminal	INPUT SYMBOL					
	a	b	e	i	t	$
S	S →a			S →iEtSS'		
S'			S' →∈ S' →eS			S' →∈
E		E →b				

3



LL(1) Grammars

L : Scan input from Left to Right

L : Construct a Leftmost Derivation

1 : Use "1" input symbol as lookahead in conjunction with stack to decide on the parsing action

LL(1) grammars == they have no multiply-defined entries in the parsing table.

Properties of LL(1) grammars:

1. Grammar can't be ambiguous or left recursive
2. Grammar is LL(1) ⇔when A →α | β
   a. α & β do not derive strings starting with the same terminal a
   b. Either α or β can derive ∈, but not both.

Note: It may not be possible for a grammar to be manipulated into an LL(1) grammar

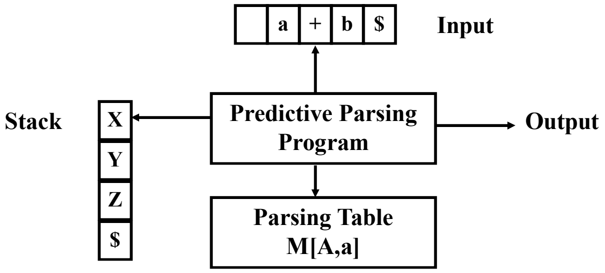
4

Is given grammar LL(1) ?

S→ AaAb | BbBa
A→ ε
B→ ε

Error Recovery

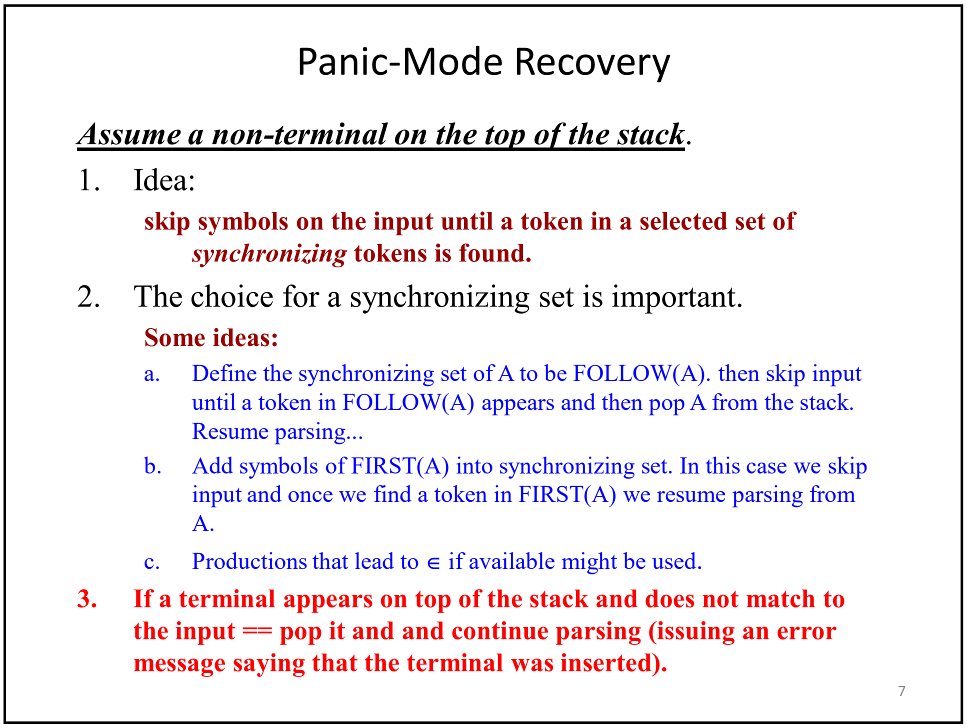
When Do Errors Occur? Recall Predictive Parser Function:



1. **If X is a terminal and it doesn't match input.**
2. **If M[X, Input] is empty – No allowable actions**

Consider two recovery techniques:

A. **Panic Mode**
B. **Phrase-level Recovery**



Panic-Mode Recovery

***Assume a non-terminal on the top of the stack*.**

1. Idea:

   skip symbols on the input until a token in a selected set of *synchronizing* tokens is found.

2. The choice for a synchronizing set is important.

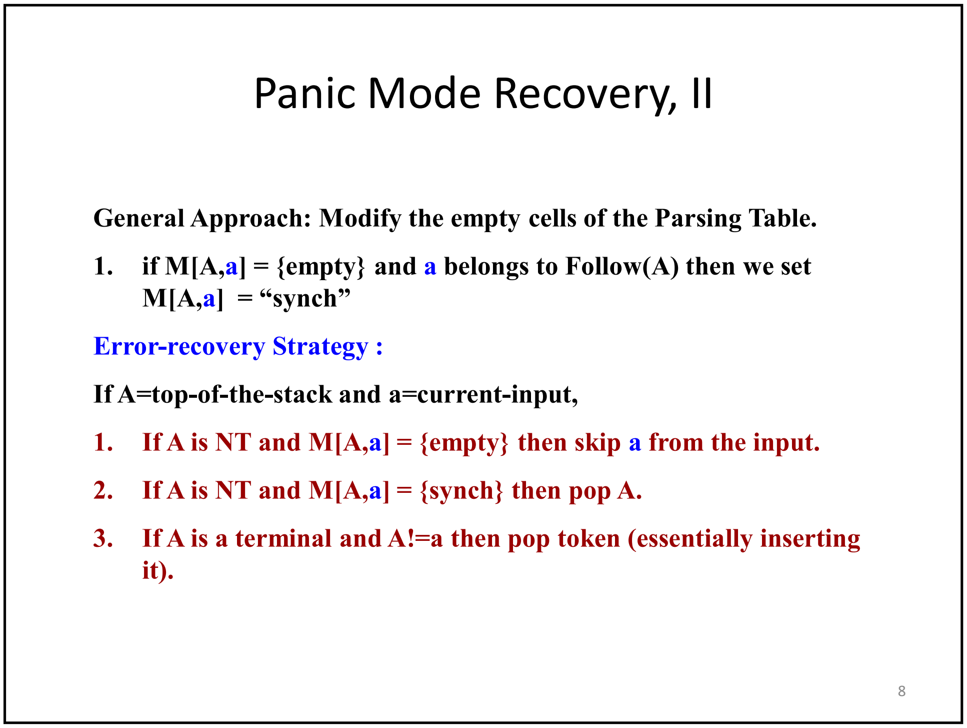
   Some ideas:

   a. Define the synchronizing set of A to be FOLLOW(A). then skip input until a token in FOLLOW(A) appears and then pop A from the stack. Resume parsing...

   b. Add symbols of FIRST(A) into synchronizing set. In this case we skip input and once we find a token in FIRST(A) we resume parsing from A.

   c. Productions that lead to ∊ if available might be used.

3. **If a terminal appears on top of the stack and does not match to the input == pop it and and continue parsing (issuing an error message saying that the terminal was inserted).**



Panic Mode Recovery, II

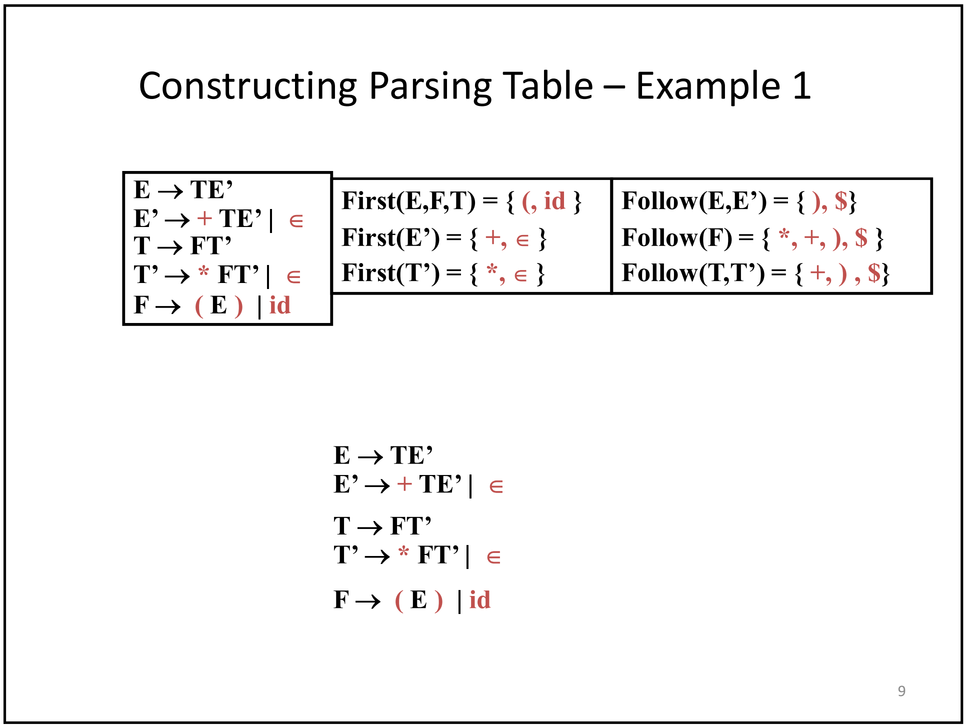
General Approach: Modify the empty cells of the Parsing Table.

1. **if M[A,a] = {empty} and a belongs to Follow(A) then we set M[A,a] = "synch"**

Error-recovery Strategy :

If A=top-of-the-stack and a=current-input,

1. **If A is NT and M[A,a] = {empty} then skip a from the input.**
2. **If A is NT and M[A,a] = {synch} then pop A.**
3. **If A is a terminal and A!=a then pop token (essentially inserting it).**



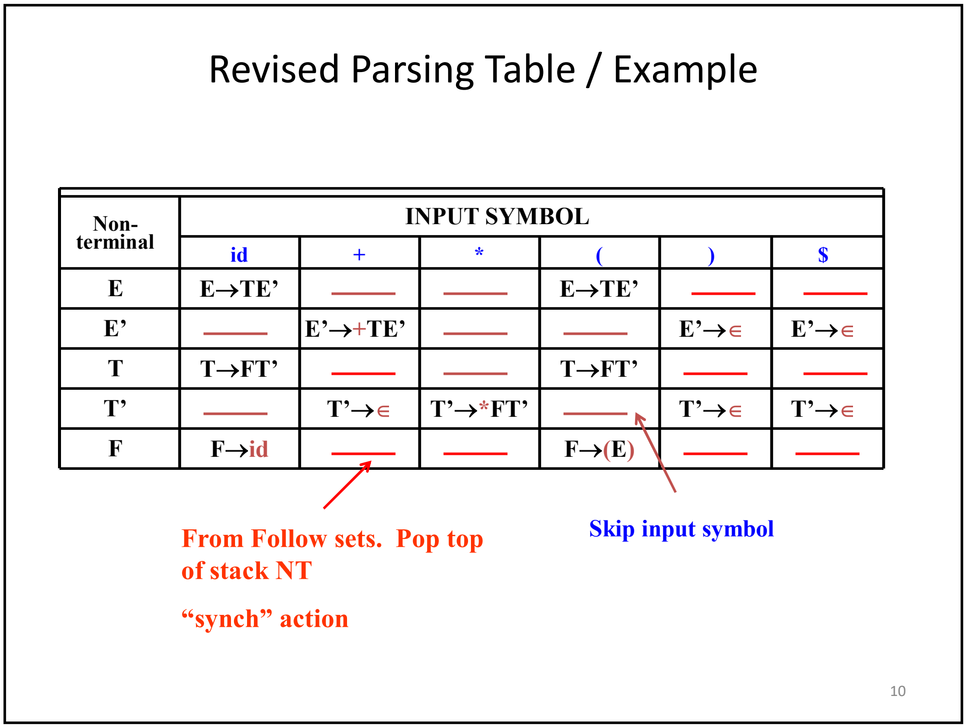
Constructing Parsing Table – Example 1

E → TE’
E’ → + TE’ | ∈
T → FT’
T’ → * FT’ | ∈
F → (E) | id
	First(E,F,T) = { (, id }
	First(E’) = { +, ∈ }
	First(T’) = { *, ∈ }
		Follow(E,E’) = {), $}
		Follow(F) = { *, +,), $ }
		Follow(T,T’) = { +,) , $}

E → TE’
E’ → + TE’ | ∈

T → FT’
T’ → * FT’ | ∈

F → (E) | id



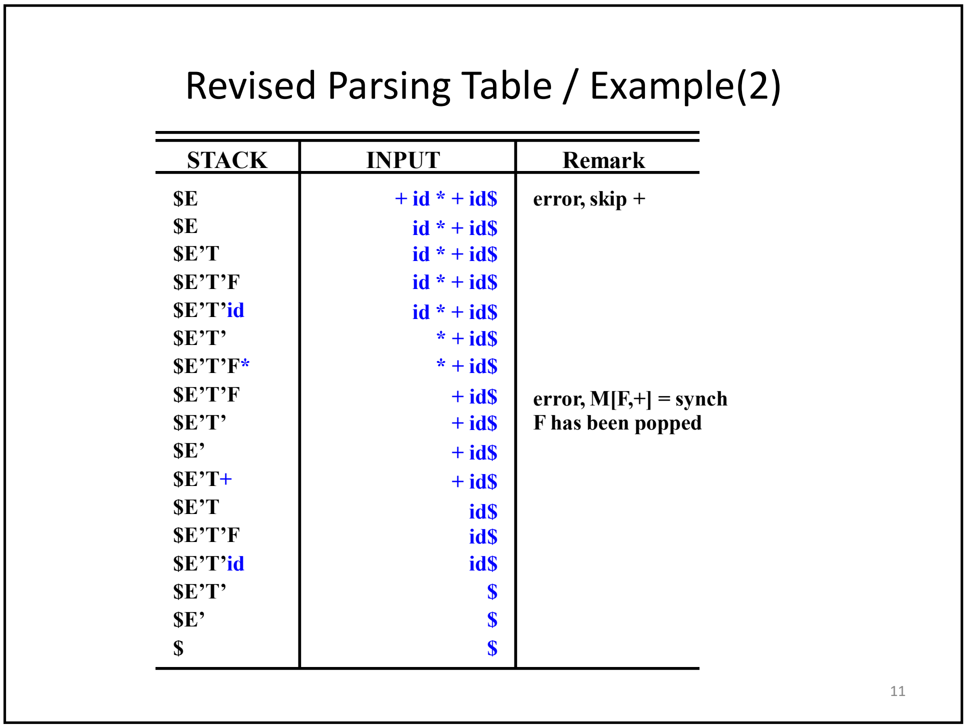
Revised Parsing Table / Example

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE’	——	——	E→TE’	——	——
E’	——	E’→+TE’	——	——	E’→∈	E’→∈
T	T→FT’	——	——	T→FT’	——	——
T’	——	T’→∈	T’→*FT’	——	T’→∈	T’→∈
F	F→id	——	——	F→(E)	——	——

From Follow sets. Pop top of stack NT

“synch” action

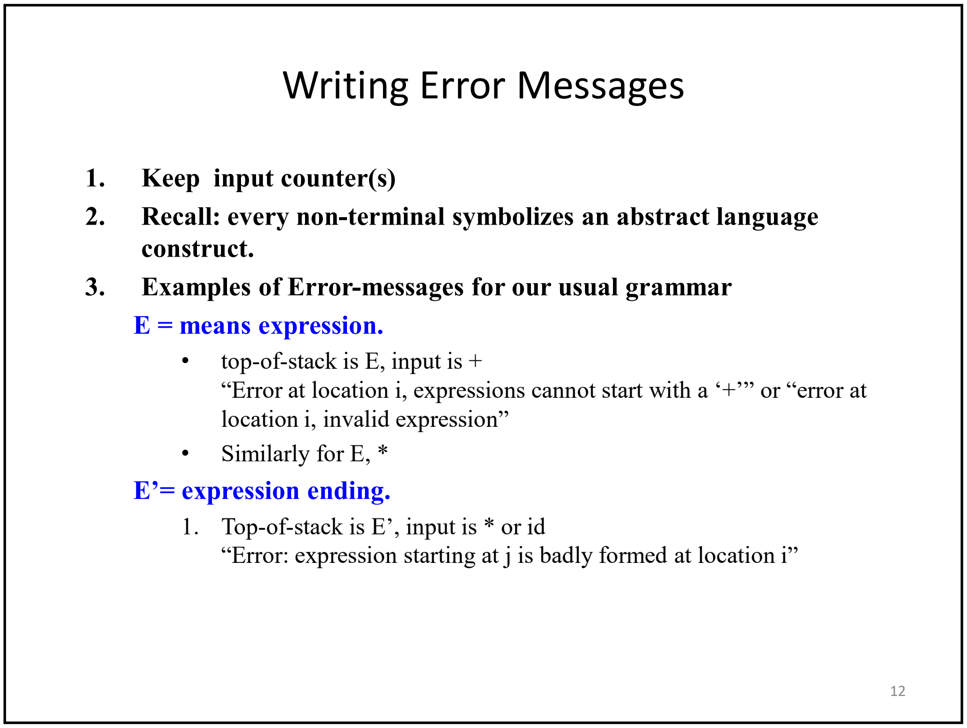
Skip input symbol



Revised Parsing Table / Example(2)

STACK	INPUT	Remark
$E	+ id * + id$	error, skip +
$E	id * + id$	
$E’T	id * + id$	
$E’T’F	id * + id$	
$E’T’id	id * + id$	
$E’T’	* + id$	
$E’T’F*	* + id$	
$E’T’F	+ id$	error, M[F,+] = synch
$E’T’	+ id$	F has been popped
$E’	+ id$	
$E’T+	+ id$	
$E’T	id$	
$E’T’F	id$	
$E’T’id	id$	
$E’T’	$	
$E’	$	
$	$	

11



Writing Error Messages

1. **Keep input counter(s)**
2. **Recall: every non-terminal symbolizes an abstract language construct.**
3. **Examples of Error-messages for our usual grammar**

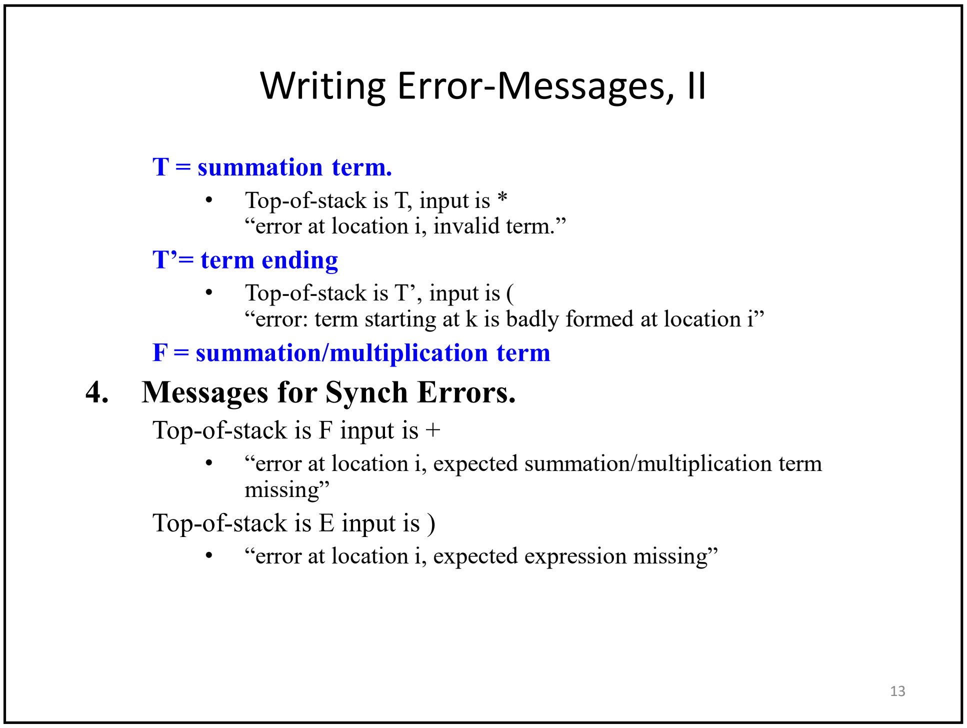
E = means expression.

- top-of-stack is E, input is +
  “Error at location i, expressions cannot start with a ‘+’” or “error at location i, invalid expression”
- Similarly for E, *

E’= expression ending.

1. Top-of-stack is E’, input is * or id
   “Error: expression starting at j is badly formed at location i”

12



Writing Error-Messages, II

T = summation term.

- Top-of-stack is T, input is *
  “error at location i, invalid term.”

T’= term ending

- Top-of-stack is T’, input is (
  “error: term starting at k is badly formed at location i”

F = summation/multiplication term

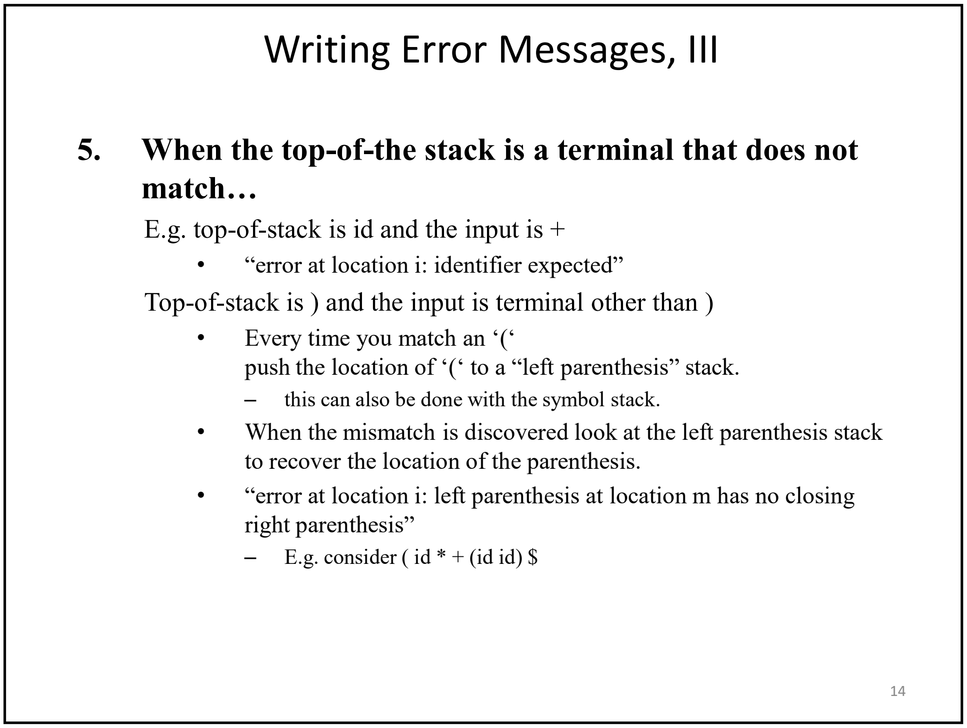
4. Messages for Synch Errors.

Top-of-stack is F input is +

- “error at location i, expected summation/multiplication term missing”

Top-of-stack is E input is)

- “error at location i, expected expression missing”



Writing Error Messages, III

5. When the top-of-the stack is a terminal that does not match…

E.g. top-of-stack is id and the input is +

- “error at location i: identifier expected”

Top-of-stack is) and the input is terminal other than)

- Every time you match an ‘(‘
  push the location of ‘(‘ to a “left parenthesis” stack.
  - this can also be done with the symbol stack.
- When the mismatch is discovered look at the left parenthesis stack to recover the location of the parenthesis.
- “error at location i: left parenthesis at location m has no closing right parenthesis”
  - E.g. consider (id * + (id id) $

Phrase-Level Recovery

1. Fill in blanks entries of parsing table with error handling routines
2. These routines
   a) Modify stack and / or input stream
   b) Issue error message
3. Problems:
   a) Modifying stack has to be done with care, so as to not create possibility of derivations that aren't in language
   b) Infinite loops must be avoided
4. Can be used in conjunction with panic mode to have more complete error handling

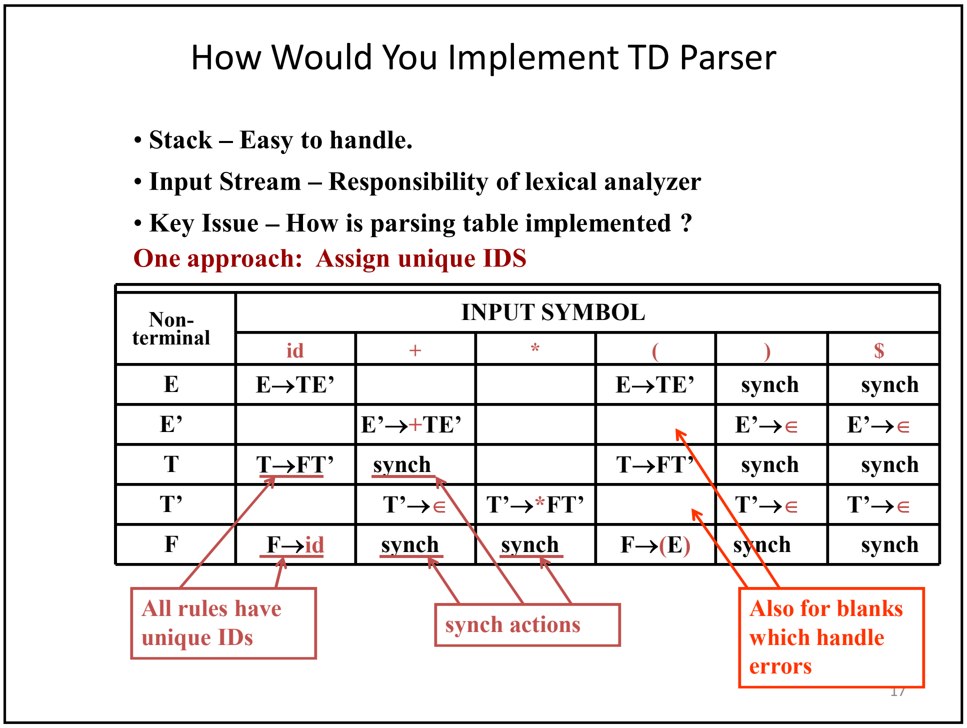
Constructing Parsing Table – Example 1

E → TE'
E' → + TE' | ∈
T → FT'
T' → * FT' | ∈
F → (E) | id
	First(E,F,T) = { (, id }
	First(E') = { +, ∈ }
	First(T') = { *, ∈ }
		Follow(E,E') = {), $}
		Follow(F) = { *, +,), $ }
		Follow(T,T') = { +,) , $}

E → TE'
E' → + TE' | ∈

T → FT'
T' → * FT' | ∈

F → (E) | id



How Would You Implement TD Parser

- Stack – Easy to handle.
- Input Stream – Responsibility of lexical analyzer
- Key Issue – How is parsing table implemented ?

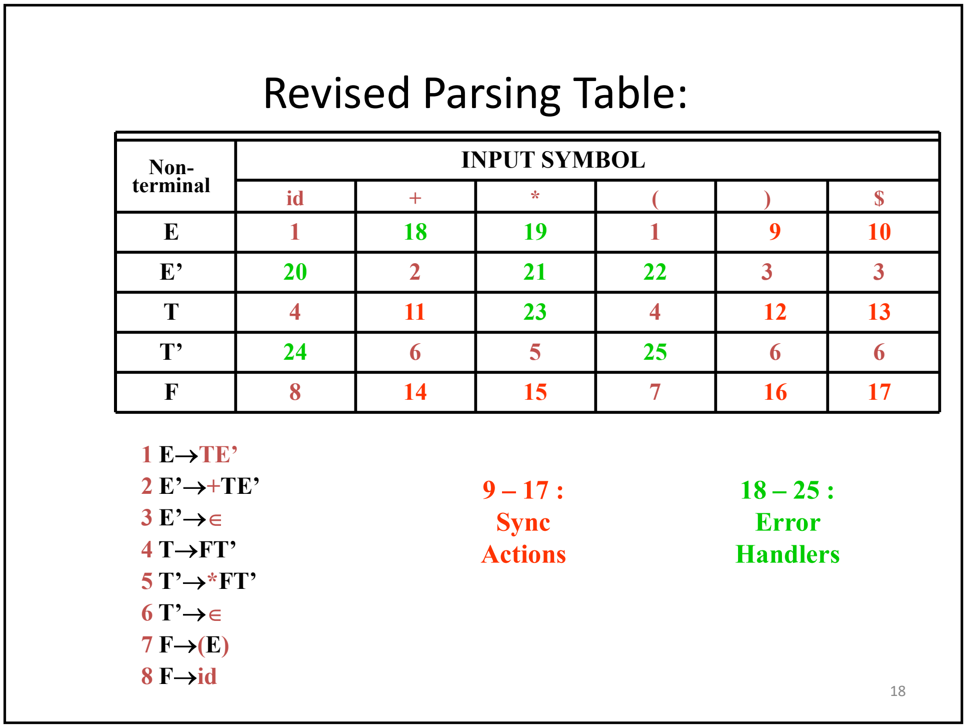
One approach: Assign unique IDS

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE’			E→TE’	synch	synch
E’		E’→+TE’			E’→∈	E’→∈
T	T→FT’	synch		T→FT’	synch	synch
T’		T’→∈	T’→*FT’		T’→∈	T’→∈
F	F→id	synch	synch	F→(E)	synch	synch

All rules have unique IDs

synch actions

Also for blanks which handle errors



Revised Parsing Table:

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	1	18	19	1	9	10
E’	20	2	21	22	3	3
T	4	11	23	4	12	13
T’	24	6	5	25	6	6
F	8	14	15	7	16	17

1 E→TE’
2 E’→+TE’
3 E’→∈
4 T→FT’
5 T’→*FT’
6 T’→∈
7 F→(E)
8 F→id

9 – 17 : Sync Actions

18 – 25 : Error Handlers