Syntax Analysis

Md Shad Akhtar
Assistant Professor
IIIT Dharwad

Top-Down Parsing

Top-Down Parsing

- Parse tree are created top to bottom
  - Begin with the start symbol to generate the input string.

- Top-down parser
  - Recursive-Descent Parsing
  - Predictive Parsing
  - Non-recursive Predictive Parsing (LL(1) parsing)

Recursive-Descent Parsing

- Tries to find the left-most derivation
- Recursively applies production rules
- If current production fails, ***backtrack***, apply another rule.
- Its simple but not widely used
- Not efficient
  - Cost of backtracking is involved which may be huge.

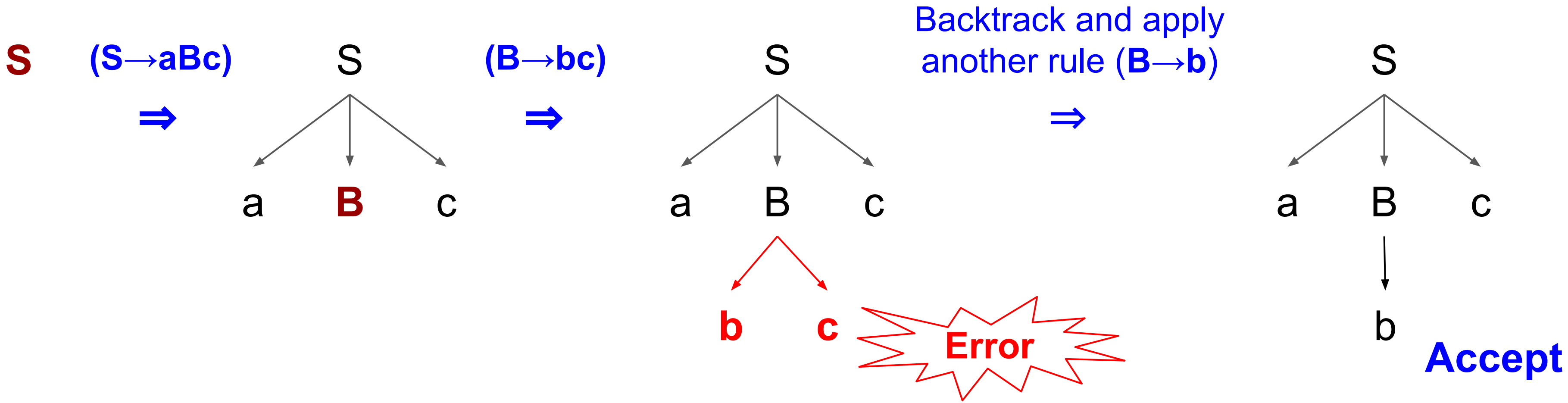
Recursive-Descent Parsing

- Let grammar G:

  $S \rightarrow aBc$

  $B \rightarrow bc \mid b$

- Input string: **a b c**



Designing a recursive-descent parser

- Write a procedure/function for each non-terminal.
- Call the associated function whenever a non-terminal is encountered during derivation.

```
function S()
{
    // match the input and/or call non-terminal functions
    // Backtrack, if it does not apply.
}
```

Designing a recursive-descent parser

- Design a recursive-descent parser for the following grammar.

$$S \rightarrow aBc$$

$$B \rightarrow bc \mid b$$

(Recursive) Predictive Parser

- A special form of recursive-descent parsing without backtracking.
- Since, no backtracking, its efficient
- But needs a special kind of grammar, i.e., LL(1) grammar
- Uniquely choose a production rule by looking at the current symbol in the input string

Production $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

Input: … a …

Current token

Predictive Parser

- **Constraints on grammar**
  a. Unambiguous
  b. No left recursion should be there
  c. Grammar should be left-factored
- Still, no 100% guarantee

Predictive Parser

- Let grammar G:

$$A \rightarrow aBe \mid cBd \mid C$$
$$B \rightarrow bB \mid \varepsilon$$
$$C \rightarrow f$$

- Left recursion?
  - No left recursion ⇒ This ensures that, given the current token, we don’t have to backtrack
- Left factored?
  - Yes ⇒ This ensures that, for a input symbol, we no longer have to make a decision

Predictive Parser

- For predictive parser, we need **lookahead symbols**
  a. Compute **FIRST()** and **FOLLOW()** for the grammar

- For a given non-terminal A and the current input symbol a
  a. IF FIRST(A) contains symbol a,
     - Apply the production associated with symbol a.
  b. ElseIF FIRST(A) contains symbol ε,
     - IF FOLLOW(A) contains symbol a,
       - Apply the production $A \rightarrow$ ε and proceed.
  c. Else
     - Error

Input string: **a b e**

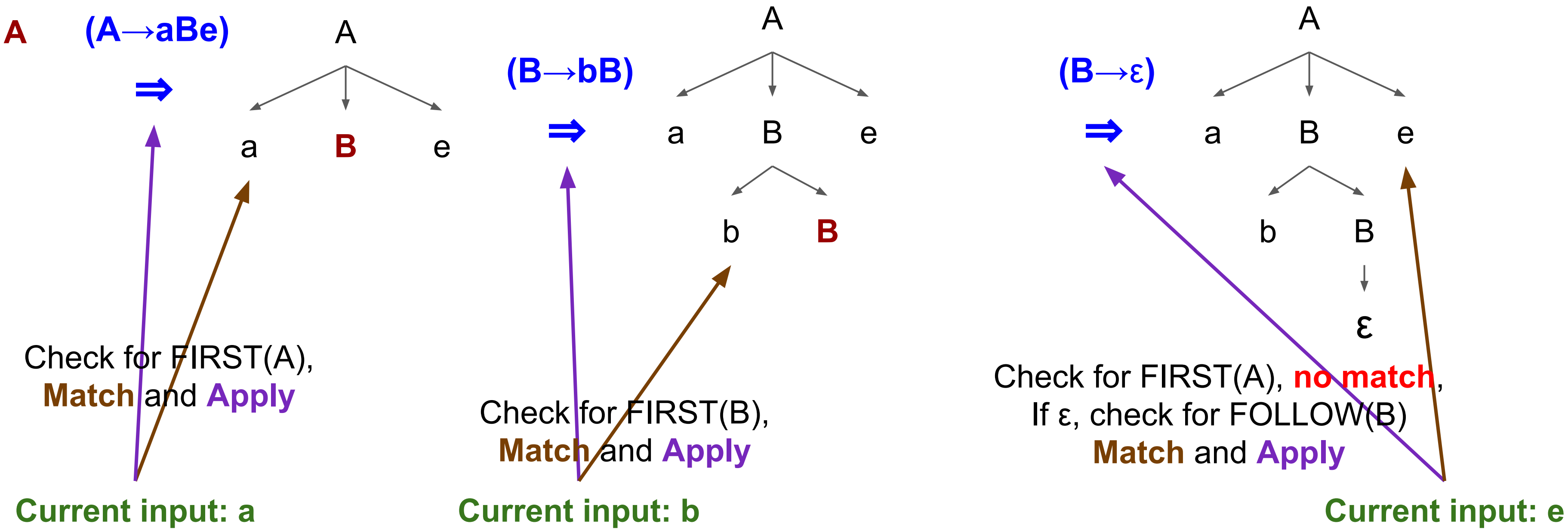
Predictive Parser

- Grammar G:

  A → aBe | cBd | C
  B → bB | ε
  C → f

  FIRST(A) = {a, c, f}
  FIRST(B) = {b, ε}
  FIRST(C) = {f}

  FOLLOW(A) = {$}
  FOLLOW(B) = {e, d}
  FOLLOW(C) = {$}



Predictive Parser

- Let grammar G:

$$S \rightarrow aBc$$
$$B \rightarrow bc \mid b$$

- Left recursion?
  - No left recursion

- Left factored?
  - No

$$S \rightarrow aBc$$
$$B \rightarrow bB'$$
$$B' \rightarrow c \mid \varepsilon$$

Predictive Parser

- Let the new grammar G':

  $S \rightarrow aBc$

  $B \rightarrow bB'$

  $B' \rightarrow c \mid \varepsilon$

- Find FIRST and FOLLOW

  a. FIRST(S) = {a}   FIRST(B) = {b}   FIRST(B') = {c, ε}

  b. FOLLOW(S) = {$}   FOLLOW(B) = {c}   FOLLOW(B') = {c}

- Input string: **a b c**

Designing a Predictive Parser

- Write a procedure/function for each non-terminal.
- Call the associated function whenever a non-terminal is encountered during derivation.
- Match lookahead with the current input and apply the rule

```
function S(lookahead, current)
{
    // match the input and/or call non-terminal functions
}
```

Designing a Predictive Parser

- Design a recursive-descent parser for the following grammar.

$$A \rightarrow aBe \mid cBd \mid C$$

$$B \rightarrow bB \mid \varepsilon$$

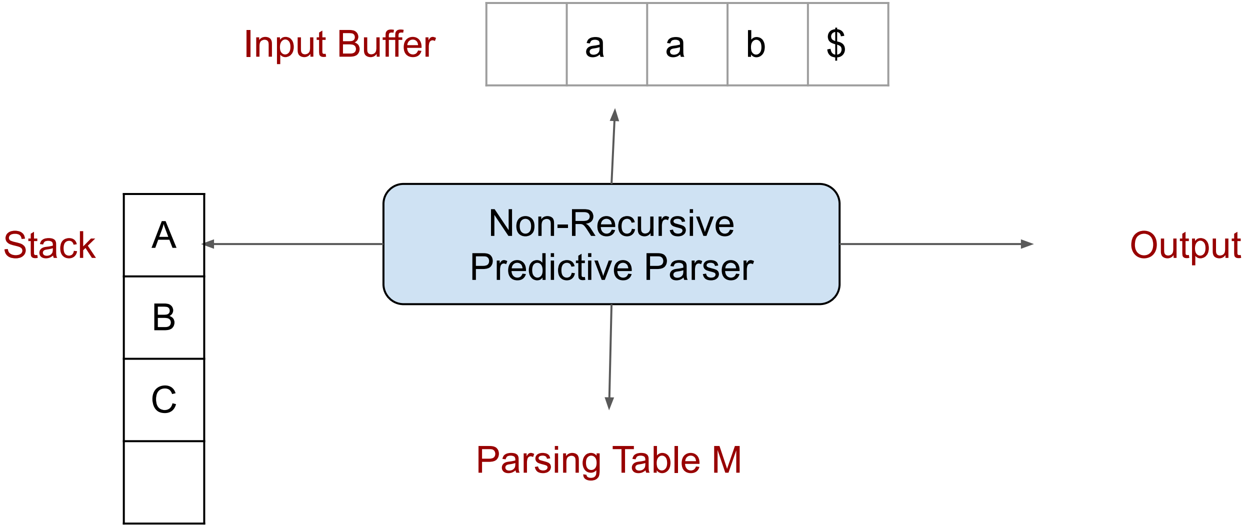
$$C \rightarrow f$$

```
proc A {
     current token {
          a:
               - match the current token with a, and move to the next token;
               - call B;
               - match the current token with e, and move to the next token;
          c:
               - match the current token with c, and move to the next token;
               - call B;
               - match the current token with d, and move to the next token;
          f:
               - call C
     }}
proc B {
     current token {
          b:
               - match the current token with b, and move to the next token;
               - call B
          e,d:
               do nothing                //FOLLOW(B)
     }}
proc C {
          f:   - match the current token with f, and move to the next token;
     }
```

LL(1) Parser

Non-Recursive Predictive or LL(1) Parser

- Top-down parser
- Table-driven parser
- LL(k), with k = 1
    - Left-to-right Left-most-derivation with k lookahead symbols



Non-Recursive Predictive or LL(1) Parser

- **Input buffer**
  - Contains the string to be parsed with end marked with a special symbol $
- **Output**
  - A production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer
- **Stack**
  - Contains the grammar symbols
  - At the bottom of the stack, there is a special end marker symbol $
  - Initially the stack contains only the symbol $ and the starting symbol S
    - $S ← Initial stack
  - Parsing completes when both input and stack becomes empty (i.e., only $ left in stack)
- **Parsing table**
  - A two-dimensional array M[A, a]
  - Each row is a non-terminal symbol
  - Each column is a terminal symbol or the special symbol $
  - Entries holds a production rule.

LL(1) Grammar

- For any grammar G, if we can build an LL(1), then the grammar is called LL(1) grammar.
  - No *left-recursive*, *non-left-factored* or *ambiguous* grammar can be LL(1)
  - Still, there are some grammar which are *non-left-recursive*, *left-factored* and *unambiguous* but not a LL(1) grammar.

- A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
  - Both α and β cannot derive strings starting with same terminals
  - At most one of α and β can derive to ε
  - If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A) and vice-versa.

Construction of LL(1) parsing table

Input: Grammar G.

Output: Parsing Table M.

1. For each production $A \rightarrow \alpha$ of the grammar,
2. do
   a. For each terminal a in FIRST(α)
      i. Add $A \rightarrow \alpha$ to M[A, a]
   b. If ε is in FIRST(α)
      i. For each terminal a in FOLLOW(A)
         1. Add $A \rightarrow \alpha$ to M[A, a]
   c. If ε is in FIRST(α) and $ is in FOLLOW(A)
      i. Add $A \rightarrow \alpha$ to M[A, a]

Construction of LL(1) parsing table

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

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

$F \rightarrow id \mid (E)$

FIRST(T') = {*, ε} $\quad$ FIRST(E) = FIRST(T) = FIRST(F) = { id, (}

FIRST(E') = {+, ε} $\quad$ FOLLOW(E) = FOLLOW(E') = {), $}

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

Non-Term	**Input Symbols**					
	id	+	*	(	)	$
E						
E'						
T						
T'						
F						

Construction of LL(1) parsing table

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

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

Production $E \rightarrow TE'$ ⇒ First(TE') = First(T) = { (, id } ⇒ Add $E \rightarrow TE'$ to M[E, id] and M[E, (]

Non-Term	Input Symbols					
	id	+	*	(	)	$
E	***E → TE'***			***E → TE'***		
E'						
T						
T'						
F						

Construction of LL(1) parsing table

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

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

Production $T \rightarrow FT'$ ⇒ First(FT') = First(F) = { (, id } ⇒ Add $T \rightarrow FT'$ to M[T, id] and M[T, (]

Non-Term	Input Symbols					
	id	+	*	(	)	$
E	*E → TE'*			*E → TE'*		
E'						
T	***T → FT'***			***T → FT'***		
T'						
F						

Construction of LL(1) parsing table

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

FIRST(T') = {*, ε}

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

FIRST(E') = {+, ε}

FOLLOW(E) = FOLLOW(E') = {), $}

FOLLOW(F) = {+, *,), $}

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

Production $F \rightarrow id$ ⇒ First(id) = { id } ⇒ Add $F \rightarrow id$ to M[F, id]

Non-Term	Input Symbols					
	id	+	*	(	)	$
E	*E → TE'*			*E → TE'*		
E'						
T	*T → FT'*			*T → FT'*		
T'						
F	***F → id***					

Construction of LL(1) parsing table

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

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

Production $F \rightarrow (E)$ ⇒ First((E)) = { (} ⇒ Add $F \rightarrow (E)$ to M[F, (]

Non-Term	Input Symbols					
	id	+	*	(	)	$
E	*E → TE'*			*E → TE'*		
E'						
T	*T → FT'*			*T → FT'*		
T'						
F	*F → id*			***F → (E)***		

Construction of LL(1) parsing table

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

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

Production $E' \rightarrow +TE'$ ⇒ First(+TE') = { + } ⇒ Add $E' \rightarrow +TE'$ to M[E', +]

Non-Term	**Input Symbols**					
	id	+	*	(	)	$
E	*E → TE'*			*E → TE'*		
E'		***E' → +TE'***				
T	*T → FT'*			*T → FT'*		
T'						
F	*F → id*			*F → (E)*		

Construction of LL(1) parsing table

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

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

Production $T' \rightarrow *FT'$ ⇒ First(*FT') = { * } ⇒ Add $T' \rightarrow *FT'$ to M[T', *]

Non-Term	**Input Symbols**					
	id	+	*	(	)	$
E	*E → TE'*			*E → TE'*		
E'		*E' → +TE'*				
T	*T → FT'*			*T → FT'*		
T'			***T' → *FT'***			
F	*F → id*			*F → (E)*		

Construction of LL(1) parsing table

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

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

Production $E' \rightarrow \varepsilon$ ⇒ Follow(E') = {), $ } ⇒ Add $E' \rightarrow \varepsilon$ to M[E',)] and M[E', $]

Non-Term	Input Symbols					
	id	+	*	(	)	$
E	*E → TE'*			*E → TE'*		
E'		*E' → +TE'*			***E' → ε***	***E' → ε***
T	*T → FT'*			*T → FT'*		
T'			*T' → *FT'*			
F	*F → id*			*F → (E)*		

Construction of LL(1) parsing table

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

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

Production T' → ε ⇒ Follow(T') = {+,), $ } ⇒ Add T' → ε to M[T', +], M[T',)] and M[T', $]

Non-Term	Input Symbols					
	id	+	*	(	)	$
E	*E → TE'*			*E → TE'*		
E'		*E' → +TE'*			*E' → ε*	*E' → ε*
T	*T → FT'*			*T → FT'*		
T'		***T' → ε***	*T' → *FT'*		***T' → ε***	***T' → ε***
F	*F → id*			*F → (E)*		

LL(1) parsing

- Once we have built a parsing table M, verify whether a given string w is part of the language or not.
- **LL(1) parsing algorithm**
  - Look at the symbol at the top of the stack (e.g., X) and the current symbol in the input string (e.g., a)
    - If X == a == \$ → Halt with success;
    - If X == a != \$ → Pop; Move to next input;
    - If X == terminal OR M[X, a] is empty → Halt with error;
    - If X in non-terminal and M[X, a] == X → $\alpha_1\alpha_2\alpha_3...\alpha_k$
      - Pop
      - Push $\alpha_k\alpha_{k-1}\alpha_{k-2}...\alpha_1$ [top of stack = α_1]
      - Output the production X → $\alpha_1\alpha_2\alpha_3...\alpha_k$

LL(1) parsing

Input: id + id * id

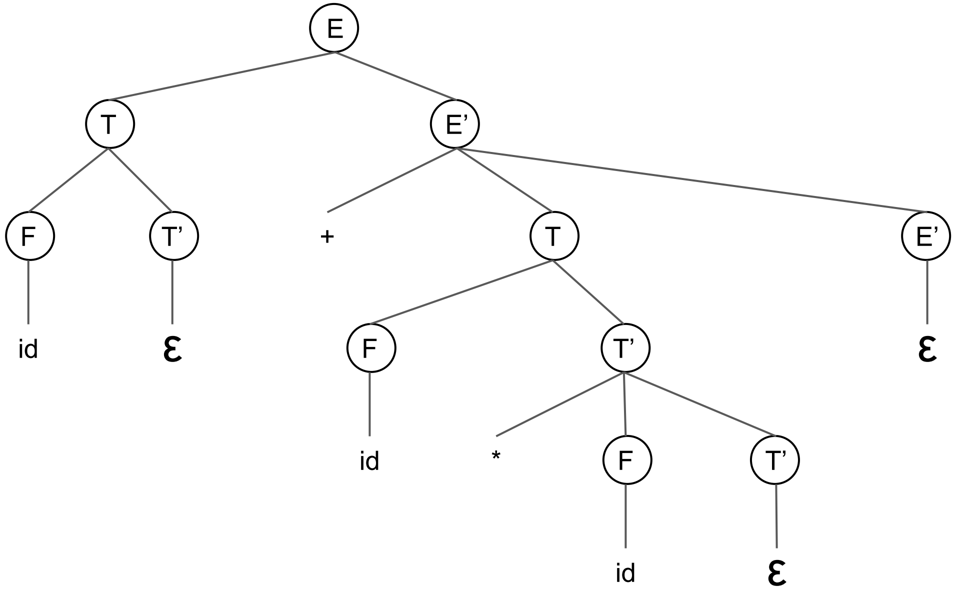
Matched	STACK	Input	Output
	E $	id + id * id $	
	T E' $	id + id * id $	Output: E → T E'
	F T' E' $	id + id * id $	Output: T → F T'
	id T' E' $	**id** + id * id $	Output: F → id
id	T' E' $	+ id * id $	Match: **id**
id	E' $	+ id * id $	Output: T' → ε
id	**+** T E' $	**+** id * id $	Output: E' → + T E'
id +	T E' $	id * id $	Match: **+**
id +	F T' E' $	id * id $	Output: T → F T'
id +	**id** T' E' $	**id** * id $	Output: F → id
id + id	T' E' $	* id $	Match: **id**
id + id	***** F T' E' $	***** id $	Output: T' → * F T'
id + id *	F T' E' $	id $	Match: *****
id + id *	**id** T' E' $	**id** $	Output: F → id
id + id * id	T' E' $	$	Match: **id**
id + id * id	E' $	$	Output: T' → ε
id + id * id	$	$	Output: E' → ε

Stack: Empty
Input: Empty
Accept and stop!!

LL(1) parsing

- Following the output sequence gives you left-most derivation for the input

$$
\begin{aligned}
E &\rightarrow T\ E' \\
&\rightarrow F\ T'\ E' \\
&\rightarrow id\ T'\ E' \\
&\rightarrow id\ \varepsilon\ E' \\
&\rightarrow id + T\ E' \\
&\rightarrow id + F\ T'\ E' \\
&\rightarrow id + id\ T'\ E' \\
&\rightarrow id + id * F\ T'\ E' \\
&\rightarrow id + id * id\ T'\ E' \\
&\rightarrow id + id * id\ \varepsilon\ E' \\
&\rightarrow id + id * id\ \varepsilon
\end{aligned}
$$



LL(1) parsing

- Construct parsing table for the following grammar:

  $S \rightarrow i\ E\ t\ S\ \mid i\ E\ t\ S\ e\ S \mid a$

  $E \rightarrow b$

[S' on e] = ??

Non-Term	**Input Symbols**					
	a	b	e	i	t	$
S	***S → a***			***S → i E t S S'***		
S'			***S' → ε*** ***S' → eS***			***S' → ε***
E		***E → b***				

Error Recovery in LL(1) Parsing

- An error may occur in the predictive parsing (LL(1) parsing), if
  - The terminal symbol on the top of stack does not match with the current input symbol
  - top of stack is a non-terminal A, the current input symbol is a, and the parsing table entry M[A, a] is empty.

Panic-mode Error Recovery in LL(1) Parsing

- Skip over the symbols on the input until a synchronization [sync] token is found.
- Synchronization tokens
  - Place all the symbols in the FOLLOW(A) into the synchronizing token set for the non-terminal A.

- If a non-terminal A can generate ε, then A → ε can be used as default choice.
- If a terminal on the top of stack cannot be matched, pop the terminal.
- If a non-terminal on the top of stack has an entry sync on a terminal a, skip the terminal.

Modified LL(1) parsing table with "sync" tokens

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

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

Non-Term	**Input Symbols**					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	**sync**	**sync**
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$	**sync**		$T \rightarrow FT'$	**sync**	**sync**
T'		$\boldsymbol{T' \rightarrow \varepsilon}$	$T' \rightarrow *FT'$		$\boldsymbol{T' \rightarrow \varepsilon}$	$\boldsymbol{T' \rightarrow \varepsilon}$
F	$F \rightarrow id$	**sync**	**sync**	$F \rightarrow (E)$	**sync**	**sync**

Panic-mode Recovery in LL(1) parsing

Input:) id * + id

STACK	Input	Remarks
E $	**) id * + id $**	**Error, M[E,)] = sync, skip)**
E $	id * + id $	id is in FIRST(E)
T E’ $	id * + id $	
F T’ E’ $	id * + id $	
id T’ E’ $	**id** * + id $	
T’ E’ $	* + id $	
* F T’ E’ $	***** + id $	
F T’ E’ $	**+ id $**	**Error, M[F, +] = sync, skip +**
F T E’ $	id $	id is in FIRST(F)
id T’ E’ $	**id** $	
T’ E’ $	$	
E’ $	$	
$	$	