Lecture 05

Syntax Analyzer (Parser)

Part 2: Top-down parsing

Hyosu Kim

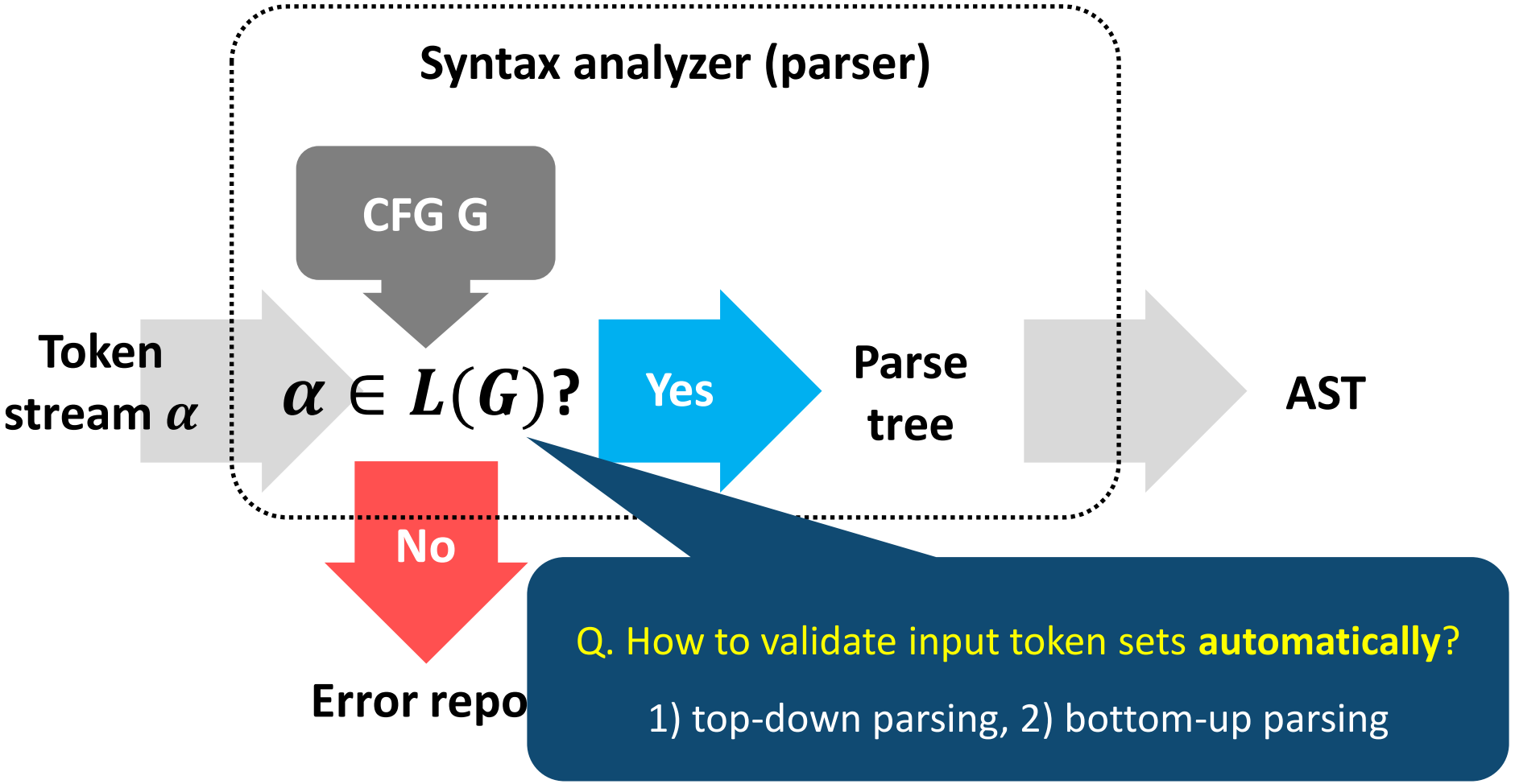
School of Computer Science and Engineering

Chung-Ang University, Seoul, Korea

https://hcslab.cau.ac.kr

hskimhello@cau.ac.kr, hskim.hello@gmail.com

Syntax analyzer



Top-down parsing

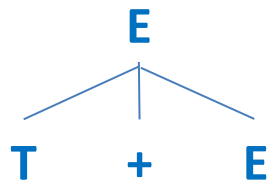
Finds a sequence of derivations from the start symbol to an input string

- It applies **a leftmost derivation** to the input string
- Its corresponding parse tree is built **from top (root) to bottom (leaf)**

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

For $id * id + id$

- E
- $\Rightarrow_{lm} T + E$



Top-down parsing

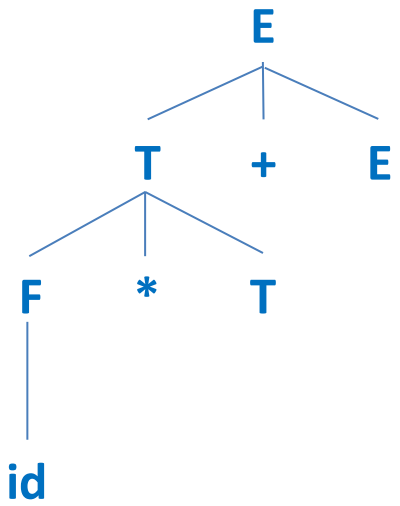
Finds a sequence of derivations from the start symbol to an input string

- It applies **a leftmost derivation** to the input string
- Its corresponding parse tree is built **from top (root) to bottom (leaf)**

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

For $id * id + id$

- E
- $\Rightarrow_{lm} T + E$
- $\Rightarrow_{lm} F * T + E$
- $\Rightarrow_{lm} id * T + E$



Top-down parsing

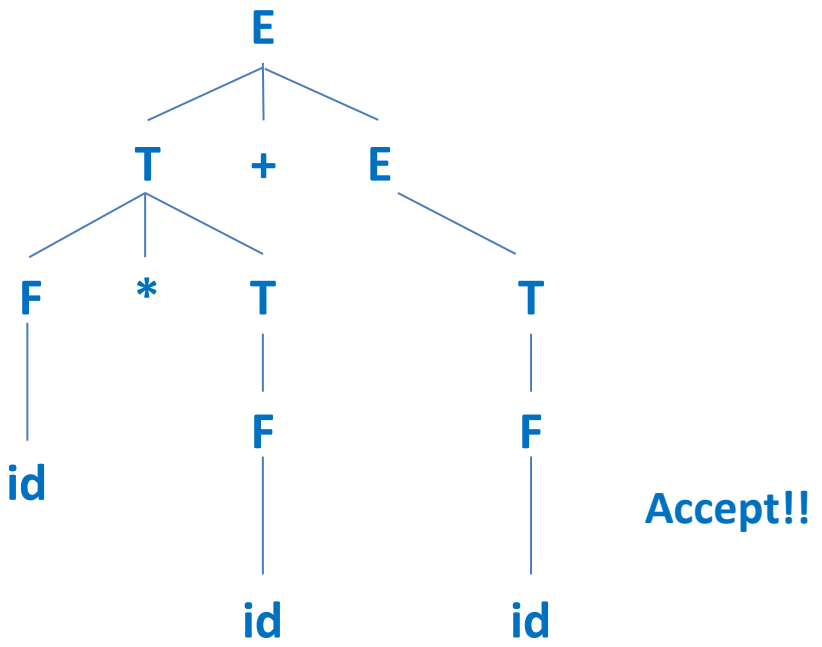
Finds a sequence of derivations from the start symbol to an input string

- It applies **a leftmost derivation** to the input string
- Its corresponding parse tree is built **from top (root) to bottom (leaf)**

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

For $id * id + id$

- E
- $\Rightarrow_{lm} T + E$
- $\Rightarrow_{lm} F * T + E$
- $\Rightarrow_{lm} id * T + E$
- $\Rightarrow_{lm} id * F + E$
- $\Rightarrow_{lm} id * id + E$
- $\ldots \Rightarrow_{lm} id * id + id$



Accept!!

Top-down parsing

Finds a sequence of derivations from the start symbol to an input string

- It applies **a leftmost derivation** to the input string
- Its corresponding parse tree is built **from top (root) to bottom (leaf)**

$$E \rightarrow T + E | T, \qquad T \rightarrow F * T | F, \qquad F \rightarrow (E) | id$$

Top-down parsing is also called LL parsing

The first **L** means **"left-to-right scan of input"**

The second **L** means **"leftmost derivation"**

- $\Rightarrow_{lm} id * F + E$
- $\Rightarrow_{lm} id * id + E$
- $\ldots \Rightarrow_{lm} id * id + id$

Top-down parsing

Finds a sequence of derivations from the start symbol to an input string

- It applies **a leftmost derivation** to the input string
- Its corresponding parse tree is built **from top (root) to bottom (leaf)**

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

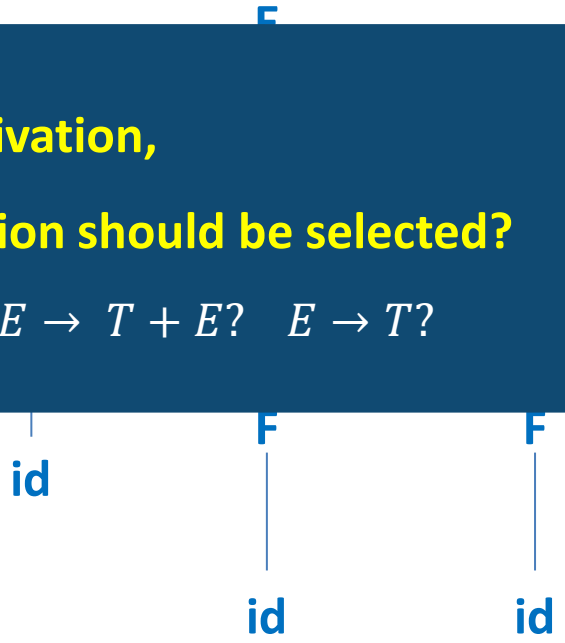
For $id * id + id$

- E
- $\Rightarrow_{lm} T + E$
- $\Rightarrow_{lm} F * T + E$
- $\Rightarrow_{lm} id * T + E$
- $\Rightarrow_{lm} id * F + E$
- $\Rightarrow_{lm} id * id + E$
- $\ldots \Rightarrow_{lm} id * id + id$

Q. At each derivation,

which production should be selected?

$E \rightarrow T + E?$ $E \rightarrow T?$



Top-down parsing #1: Recursive descent

The simplest top-down parsing technique

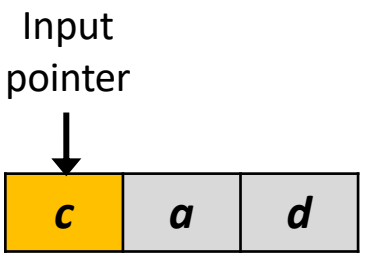
- Parsing begins from a start symbol

$S \rightarrow dAc|cAe|cAd$

$A \rightarrow a$

For an input string cad

- S



S

Top-down parsing #1: Recursive descent

The simplest top-down parsing technique

- Parsing begins from a start symbol
- It tries the rules for a leftmost non-terminal **in order**

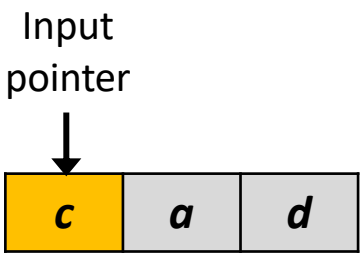
$S \rightarrow dAc|cAe|cAd$

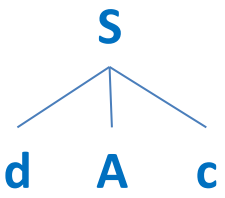
$A \rightarrow a$

For an input string cad

- S

  $\Rightarrow dAc$





Top-down parsing #1: Recursive descent

The simplest top-down parsing technique

- Parsing begins from a start symbol
- It tries the rules for a leftmost non-terminal **in order**
  - If terminals are newly derived before a leftmost non-terminal, compare them with the input
    - If it is matched: advance an input pointer
    - Otherwise: do backtracking

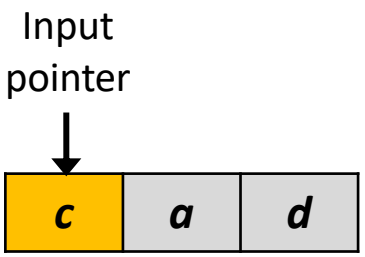
$S \rightarrow dAc|cAe|cAd$

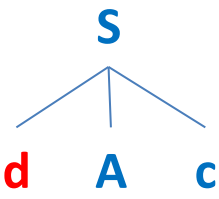
$A \rightarrow a$

For an input string cad

- S

  $\Rightarrow dAc$





Mismatch at the first terminal: $c \neq d$

Do backtrack!!

(the second rule of S is applied)

Top-down parsing #1: Recursive descent

The simplest top-down parsing technique

- Parsing begins from a start symbol
- It tries the rules for a leftmost non-terminal **in order**
  - If terminals are newly derived before a leftmost non-terminal, compare them with the input
    - If it is matched: advance an input pointer
    - Otherwise: do backtracking

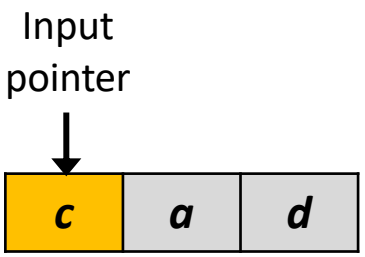
$S \rightarrow dAc|cAe|cAd$

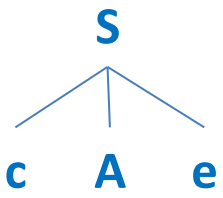
$A \rightarrow a$

For an input string cad

- S

  $\Rightarrow cAe$





The first terminal is matched!

Then, try parsing for the next non-terminal

+ advance the input pointer

Top-down parsing #1: Recursive descent

The simplest top-down parsing technique

- Parsing begins from a start symbol
- It tries the rules for a leftmost non-terminal **in order**
  - If terminals are newly derived before a leftmost non-terminal, compare them with the input
    - If it is matched: advance an input pointer
    - Otherwise: do backtracking

$S \rightarrow dAc|cAe|cAd$

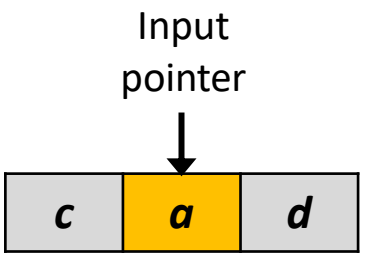
$A \rightarrow a$

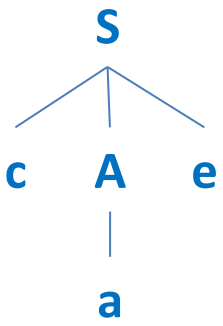
For an input string cad

- S

  $\Rightarrow cAe$

  $\Rightarrow cae$





The second terminals is matched

Advance the input pointer

Top-down parsing #1: Recursive descent

The simplest top-down parsing technique

- Parsing begins from a start symbol
- It tries the rules for a leftmost non-terminal **in order**
  - If terminals are newly derived before a leftmost non-terminal, compare them with the input
    - If it is matched: advance an input pointer
    - Otherwise: do backtracking

$S \rightarrow dAc|cAe|cAd$

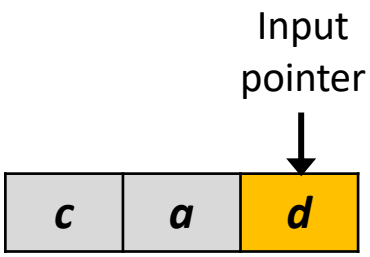
$A \rightarrow a$

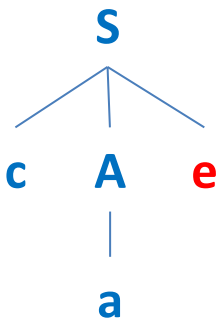
For an input string cad

- S

  $\Rightarrow cAe$

  $\Rightarrow cae$





Mismatch: $d \neq e$

Do backtrack!!

Top-down parsing #1: Recursive descent

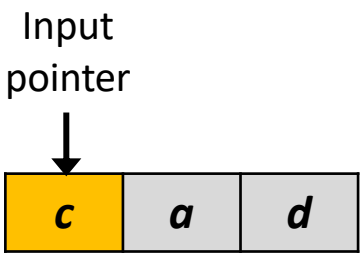
The simplest top-down parsing technique

- Parsing begins from a start symbol
- It tries the rules for a leftmost non-terminal **in order**
  - If terminals are newly derived before a leftmost non-terminal, compare them with the input
    - If it is matched: advance an input pointer
    - Otherwise: do backtracking

$S \rightarrow dAc|cAe|cAd$

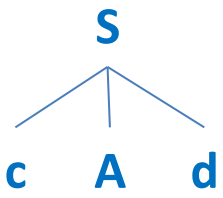
$A \rightarrow a$

For an input string cad



- S

  $\Rightarrow cAd$



Top-down parsing #1: Recursive descent

The simplest top-down parsing technique

- Parsing begins from a start symbol
- It tries the rules for a leftmost non-terminal **in order**
  - If terminals are newly derived before a leftmost non-terminal, compare them with the input
    - If it is matched: advance an input pointer
    - Otherwise: do backtracking
- Accept if all input characters are matched (reject if all productions are exhausted)

$S \rightarrow dAc|cAe|cAd$

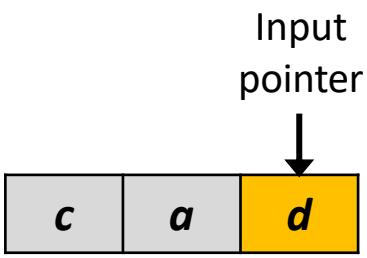
$A \rightarrow a$

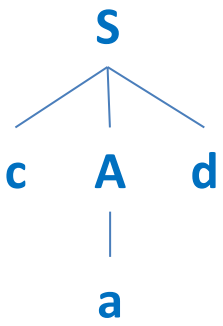
For an input string cad

- S

  $\Rightarrow cAd$

  $\Rightarrow cad$





End of input, accept!!

Top-down parsing #1: Recursive descent

The simplest top-down parsing technique

Let's try

$$E \rightarrow T + E | T,\ T \rightarrow F * T | F,\ F \rightarrow (E) | id$$

For an input string $id + id$

Summary: Recursive descent

The advantages of recursive descent parsers

- Easy to understand
- Easy to implement (by hand)

Conditions for using recursive descent parsers

- A CFG is non-ambiguous
- A CFG is no left recursive
- Left factoring is not needed

But, the recursive descent parsing is unpopular…

Why?? Because of backtracking

Top-down parsing #2: Predictive parsing

Predictive parsing is also called LL(k) parsing

- **The first L:** scanning input from left to right
- **The second L:** producing a leftmost derivation
- **k:** using k input symbols for a lookahead -> “prediction”
- In practice, LL(1) is used
  - For a leftmost nonterminal A,

    the LL(1) parser decides a production based on the next input symbol a

  - e.g., For a CFG: $S \rightarrow aA|b|A, \quad A \rightarrow (S)$

leftmost nonterminal	Next input symbol	Decision
S	a	$S \Rightarrow aA$
S	b	$S \Rightarrow b$
S	$($	$S \Rightarrow A$

Top-down parsing #2: Predictive parsing

A recursive descent parsing, needing no backtracking

Conditions for using predictive parsers

- A CFG is non-ambiguous
- A CFG is no left recursive
- **A CFG must be left factored**

How LL(1) parser works

1. For a non-left recursive and left factored CFG, construct LL(1) parsing table

$E \rightarrow TE'$, $E' \rightarrow +E|\epsilon$, $T \rightarrow FT'$, $T' \rightarrow *T|\epsilon$, $F \rightarrow (E)|id$

		The next input symbol					
		+	$*$	$($	$)$	***id***	$ (endmarker)
Leftmost non-terminal	***E***			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*T$		ϵ		ϵ
	F			(E)		id	

How LL(1) parser works

2. For a given input string, start parsing based on the LL(1) parsing table

		The next input symbol					
		+	*	(	)	**id**	$ (endmarker)
Leftmost non-terminal	**E**			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*T$		ϵ		ϵ
	F			(E)		id	

- For $(id)\$$
  - The next input symbol = (
  - $E \Rightarrow_{lm} TE' \Rightarrow_{lm} FT'E' \Rightarrow_{lm} (E)T'E'$

How LL(1) parser works

2. For a given input string, start parsing based on the LL(1) parsing table

		The next input symbol					
		+	*	(	)	***id***	$ (endmarker)
Leftmost non-terminal	***E***			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*\,T$		ϵ		ϵ
	F			(E)		id	

- For $(id)\$$
  - The next input symbol = ***id***
  - $E \Rightarrow_{lm}^{*} (E)T'E' \Rightarrow_{lm} (TE')T'E' \Rightarrow_{lm} (FT'E')T'E' \Rightarrow_{lm} (idT'E')T'E'$

How LL(1) parser works

2. For a given input string, start parsing based on the LL(1) parsing table

		The next input symbol					
		+	*	(	)	**id**	$ (endmarker)
Leftmost non-terminal	**E**			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*T$		ϵ		ϵ
	F			(E)		id	

- For $(id)\$$
  - The next input symbol =)
  - $E \Rightarrow_{lm}^{*} (idT'E')T'E' \Rightarrow_{lm} (idE')T'E' \Rightarrow_{lm} (id)T'E'$

How LL(1) parser works

2. For a given input string, start parsing based on the LL(1) parsing table

		The next input symbol					
		+	*	(	)	**id**	$ (endmarker)
Leftmost non-terminal	**E**			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*T$		ϵ		ϵ
	F			(E)		id	

- For $(id)\$$
  - The next input symbol = $
  - $E \Rightarrow_{lm}^{*} (id)T'E' \Rightarrow_{lm} (id)E' \Rightarrow_{lm} (id)$ **Accept!!**

How LL(1) parser works

1. For a given CFG, construct LL(1) parsing table

2. For a given input string, start parsing based on the LL(1) parsing table

How to construct the LL(1) table easily????

LL(1) parsing table construction

Definition: First set

- The first set of a non-terminal A: $\boldsymbol{First(A) = \{x | A \Rightarrow^* x\alpha\}}$
  - A set of terminals x that begin strings derived from A
  - α is any sequence of non-terminals and terminals
  - $\epsilon \in First(A), \ \ if\ A \Rightarrow^* \epsilon$

- The first set of x (terminal): $\boldsymbol{First(x) = \{x\}}$

- The first set of α (a sequence of non-terminals and terminals): $First(\alpha)$
  - $First(\alpha) = First(x), \ if\ \alpha = x\beta$
  - $First(\alpha) = First(A_1) \cup First(A_2) \cup \cdots \cup First(A_n) \cup First(x),$
    $if\ \alpha = A_1 A_2 \ldots A_n x\beta\ and\ \epsilon \in First(A_i)\ for\ all\ i$
  - $\epsilon \in First(\alpha), \ \ if\ \alpha = A_1 A_2 \ldots A_n\ and\ \epsilon \in First(A_i)\ \ for\ all\ i$

LL(1) parsing table construction

Examples for the first set

$E \to TE', \qquad E' \to +E|\epsilon, \qquad T \to FT', \qquad T' \to *T|\epsilon, \qquad F \to (E)|id$

- $First(F) = First\big((E)\big) \cup First(id) = First(() \cup First(id) = \{(, id\}$
- $First(T') = First(*T) \cup First(\epsilon) = First(*) \cup First(\epsilon) = \{*, \epsilon\}$
- $First(T) = First(FT') = First(F) = \{(, id\}$
- $First(E') = First(+E) \cup First(\epsilon) = First(+) \cup First(\epsilon) = \{+, \epsilon\}$
- $First(E) = First(TE') = First(T) = \{(, id\}$

LL(1) parsing table construction

Definition: Follow set

- The follow set of a non-terminal A: $\boldsymbol{Follow(A) = \{x | S \Rightarrow^* \alpha A x \beta\}}$, where S is a start symbol
  - A set of terminals x that can appear immediately to the right of A during derivations

- $\$ \in Follow(S)$

- $First(\beta) - \{\epsilon\} \subseteq Follow(A), \quad if\ there\ is\ a\ production\ B \rightarrow \alpha A \beta$

- $Follow(B) \subseteq Follow(A), \quad if\ there\ \ is\ a\ production\ B \rightarrow \alpha A \beta, where\ \epsilon \in First(\beta)$

$$or, if\ there\ is\ a\ production\ B \rightarrow \alpha A$$

LL(1) parsing table construction

Examples for the follow set

$$E \to TE', \qquad E' \to +E|\epsilon, \qquad T \to FT', \qquad T' \to * T|\epsilon, \qquad F \to (E)|id$$

$$First(F) = \{(, id\}, \qquad First(T') = \{*, \epsilon\}, \qquad First(T) = \{(, id\},$$
$$First(E') = \{+, \epsilon\}, \qquad First(E) = \{(, id\}$$

- $Follow(E) = \{\$\} \cup First()) \cup Follow(E') = \{\$,)\} \cup Follow(E) = \{\$,)\}$
- $Follow(E') = Follow(E) = \{\$,)\}$
- $Follow(T) = First(E') - \{\epsilon\} \cup Follow(E) \cup Follow(T') = \{+, \$,)\} \cup Follow(T) = \{+, \$,)\}$
- $Follow(T') = Follow(T) = \{+, \$,)\}$
- $Follow(F) = First(T') - \{\epsilon\} \cup Follow(T) = \{*, +, \$,)\}$

LL(1) parsing table construction

For each production $A \rightarrow \alpha$ in a CFG,

- For each terminal $x \in First(\alpha)$,
  - Fill the table entry $[A, x]$ as α

- For each terminal $x \in Follow(A)$,
  - Fill the table entry $[A, x]$ as α, if $\epsilon \in First(\alpha)$

LL(1) parsing table construction

For $A \rightarrow \alpha$, fill the table entry $[A, x]$ as α for each terminal $x \in First(\alpha)$

$$E \rightarrow TE', \qquad E' \rightarrow +E|\epsilon, \qquad T \rightarrow FT', \qquad T' \rightarrow * T|\epsilon, \qquad F \rightarrow (E)|id$$

$$First(TE') = First(T) = \{(, id\}, \quad First(+E) = First(+) = \{+\}$$
$$First(FT') = First(F) = \{(, id\}, \quad First(* T) = First(*) = \{*\}$$
$$First((E)) = First(() = \{(\}$$

		The next input symbol					
		+	*	(	)	id	$ (endmarker)
Leftmost non-terminal	E			TE'		TE'	
	E'	$+E$					
	T			FT'		FT'	
	T'		$* T$				
	F			(E)		id	

LL(1) parsing table construction

For $A \to \alpha$, fill the table entry $[A, x]$ as α for each terminal $x \in Follow(A)$, if $\epsilon \in First(\alpha)$

$$E \to TE', \qquad E' \to +E|\epsilon, \qquad T \to FT', \qquad T' \to *T|\epsilon, \qquad F \to (E)|id$$

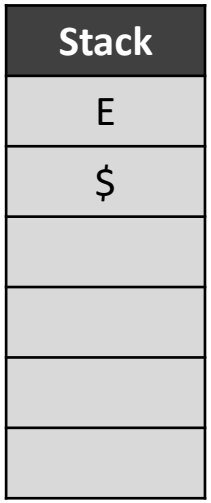
$$In\ E' \to \epsilon, First(\epsilon) = \{epsilon\}: \quad Follow(E') = Follow(E) = \{\$,)\}$$

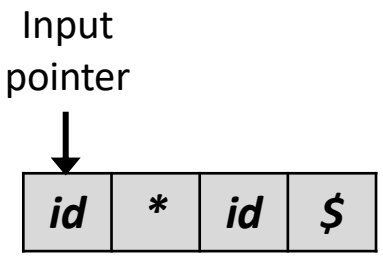
$$In\ T' \to \epsilon, First(\epsilon) = \{epsilon\}: \quad Follow(T') = Follow(T) = \{+, \$,)\}$$

		The next input symbol					
		+	*	(	)	***id***	$ (endmarker)
Leftmost non-terminal	***E***			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*T$		ϵ		ϵ
	F			(E)		id	

Implementation of LL(1) parser

Step 1: Push the endmarker symbol and start symbol to a stack



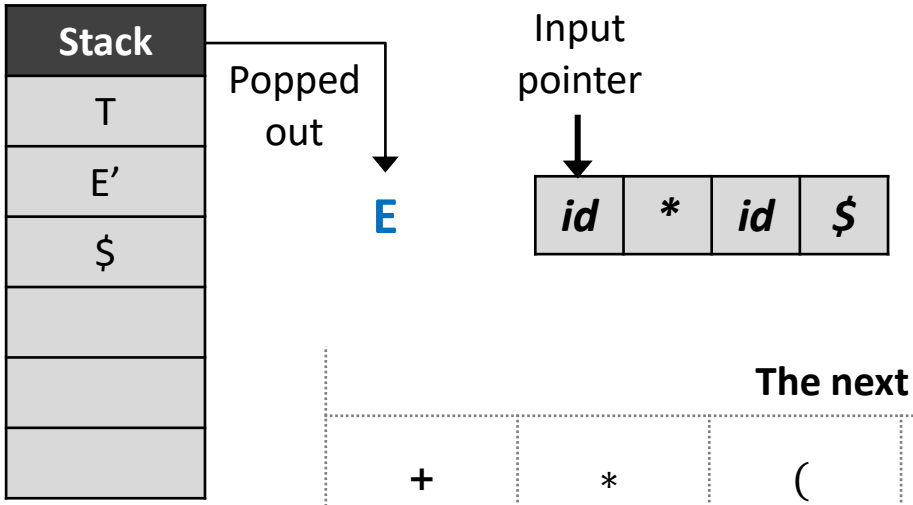


Leftmost non-terminal	The next input symbol: +	*	(	)	id	$ (endmarker)
E			TE'		TE'	
E'	$+E$			ϵ		ϵ
T			FT'		FT'	
T'	ϵ	$*T$		ϵ		ϵ
F			(E)		id	

Implementation of LL(1) parser

Step 2: Pop the first component of the stack

If it is non-terminal, do LL(1) parsing and push parsing result into the stack

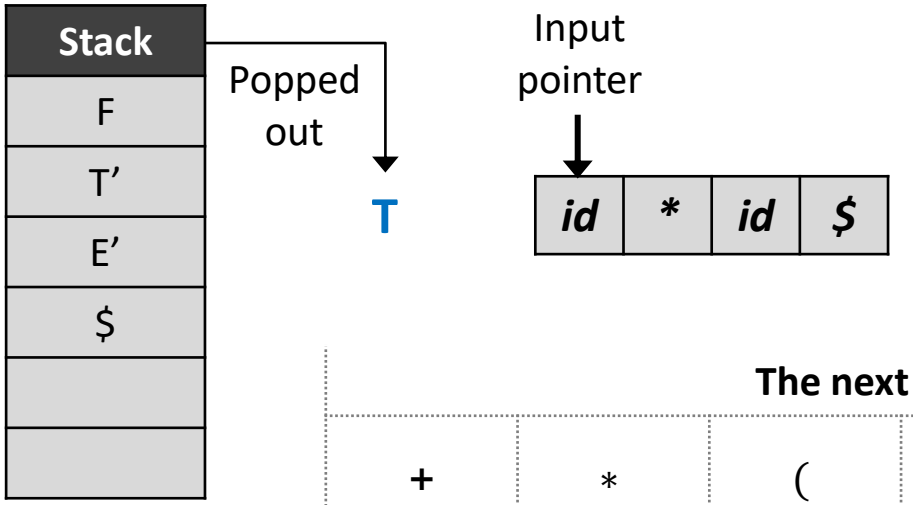


		The next input symbol					
		+	*	(	)	id	$ (endmarker)
Leftmost non-terminal	E			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*T$		ϵ		ϵ
	F			(E)		id	

Implementation of LL(1) parser

Step 2: Pop the first component of the stack

If it is non-terminal, do LL(1) parsing and push parsing result into the stack

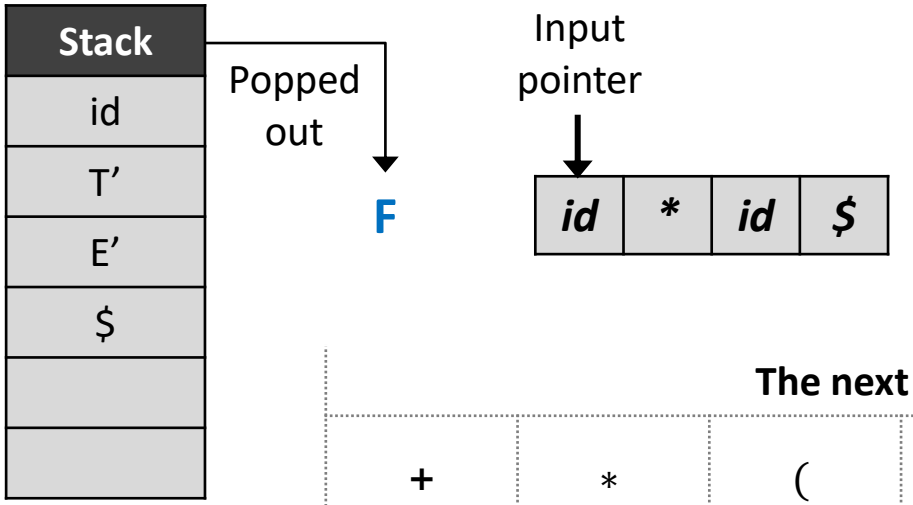


		The next input symbol					
		+	*	(	)	id	$ (endmarker)
Leftmost non-terminal	E			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*T$		ϵ		ϵ
	F			(E)		id	

Implementation of LL(1) parser

Step 2: Pop the first component of the stack

If it is non-terminal, do LL(1) parsing and push parsing result into the stack

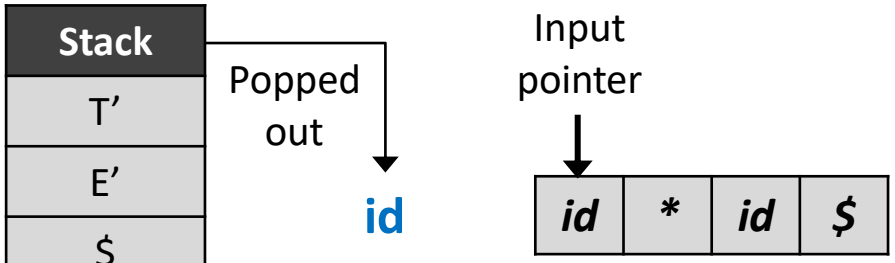


		The next input symbol					
		+	*	(	)	id	$ (endmarker)
Leftmost non-terminal	E			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*T$		ϵ		ϵ
	F			(E)		id	

Implementation of LL(1) parser

Step 2: Pop the first component of the stack

If it is terminal, compare it with the current input symbol



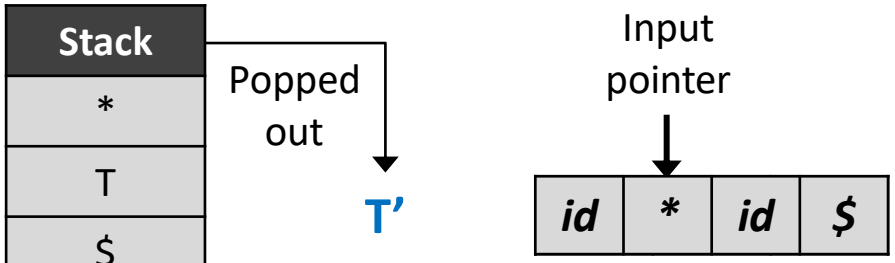
Match!!

Advance the input pointer

		The next input symbol					
		+	*	(	)	id	$ (endmarker)
Leftmost non-terminal	E			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*T$		ϵ		ϵ
	F			(E)		id	

Implementation of LL(1) parser

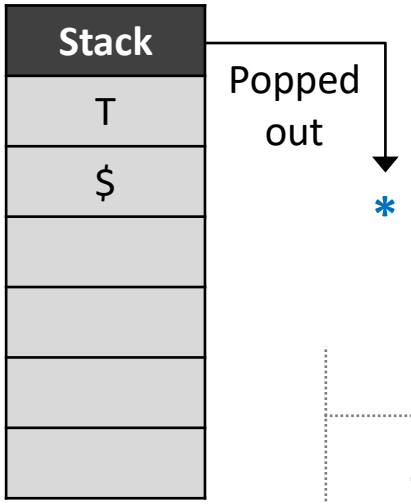
Repeat step 2 until all the input string is matched or mismatch (error) occurs

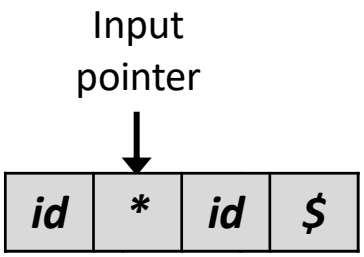


		The next input symbol					
		+	*	(	)	id	$ (endmarker)
Leftmost non-terminal	E			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*T$		ϵ		ϵ
	F			(E)		id	

Implementation of LL(1) parser

Repeat step 2 until all the input string is matched or mismatch (error) occurs





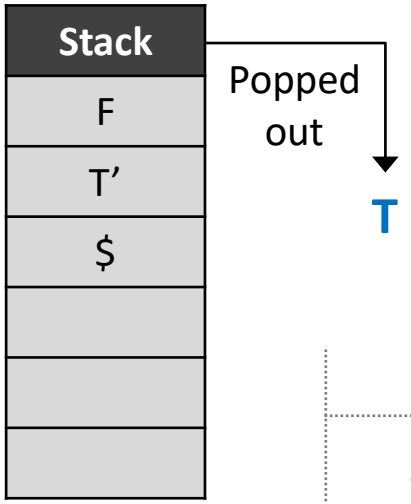
Match!!

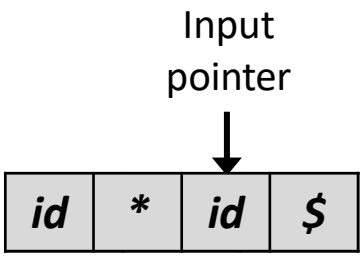
Advance the input pointer

		The next input symbol					
		+	*	(	)	id	$ (endmarker)
Leftmost non-terminal	E			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*T$		ϵ		ϵ
	F			(E)		id	

Implementation of LL(1) parser

Repeat step 2 until all the input string is matched or mismatch (error) occurs

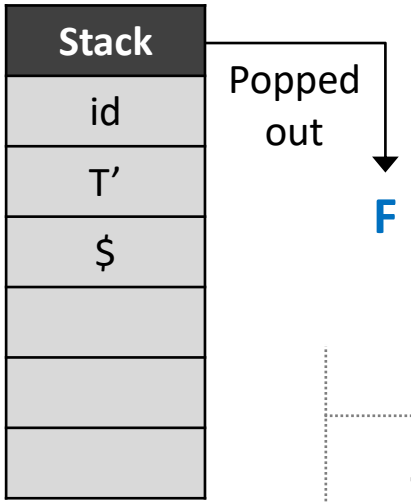




		The next input symbol					
		+	*	(	)	id	$ (endmarker)
Leftmost non-terminal	E			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*T$		ϵ		ϵ
	F			(E)		id	

Implementation of LL(1) parser

Repeat step 2 until all the input string is matched or mismatch (error) occurs



Popped out

F

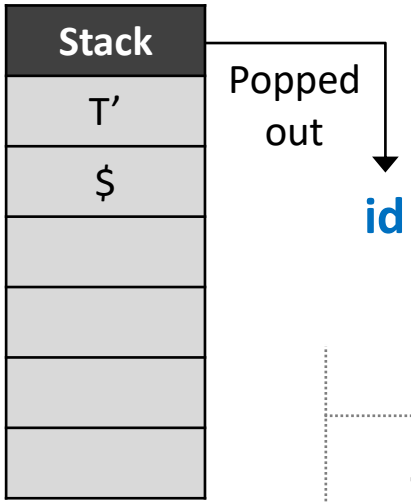
Input pointer

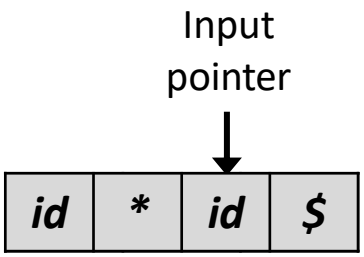
id	*	*id*	*$*

Leftmost non-terminal	The next input symbol: +	*	(	)	*id*	$ (endmarker)
E			TE'		TE'	
E'	$+E$			ϵ		ϵ
T			FT'		FT'	
T'	ϵ	$*\,T$		ϵ		ϵ
F			(E)		id	

Implementation of LL(1) parser

Repeat step 2 until all the input string is matched or mismatch (error) occurs





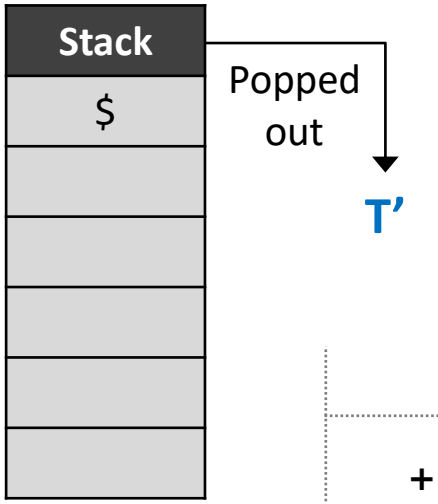
Match!!

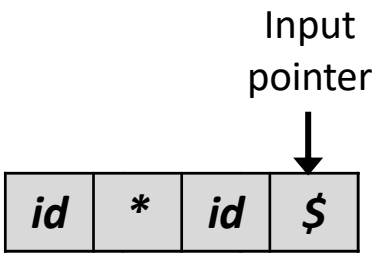
Advance the input pointer

		The next input symbol					
		+	*	(	)	id	$ (endmarker)
Leftmost non-terminal	E			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*T$		ϵ		ϵ
	F			(E)		id	

Implementation of LL(1) parser

Repeat step 2 until all the input string is matched or mismatch (error) occurs

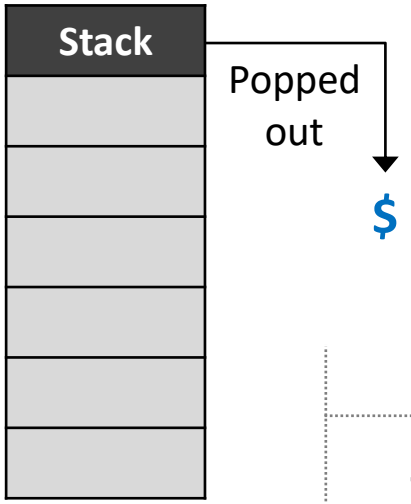


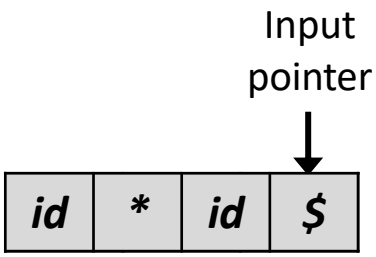


		The next input symbol					
		+	*	(	)	id	$ (endmarker)
Leftmost non-terminal	E			TE'		TE'	
	E'	$+E$			ϵ		ϵ
	T			FT'		FT'	
	T'	ϵ	$*T$		ϵ		ϵ
	F			(E)		id	

Implementation of LL(1) parser

Repeat step 2 until all the input string is matched or mismatch (error) occurs





Accept!!

Leftmost non-terminal	The next input symbol: +	*	(	)	id	$ (endmarker)
E			TE'		TE'	
E'	$+E$			ϵ		ϵ
T			FT'		FT'	
T'	ϵ	$*T$		ϵ		ϵ
F			(E)		id	

Summary: Predictive parsing

A recursive descent parsing, needing no backtracking

Conditions for using recursive descent parsers

- A CFG is non-ambiguous
- A CFG is no left recursive
- **A CFG must be left factored**

How it works

1. For a given CFG, construct LL(1) parsing table

   First set, Follow set…

2. For a given input string, start parsing based on the LL(1) parsing table

   by using stack

Syntax analyzer