Lecture 06

Syntax Analyzer (Parser)

Part 3: Bottom-up parsing

Hyosu Kim

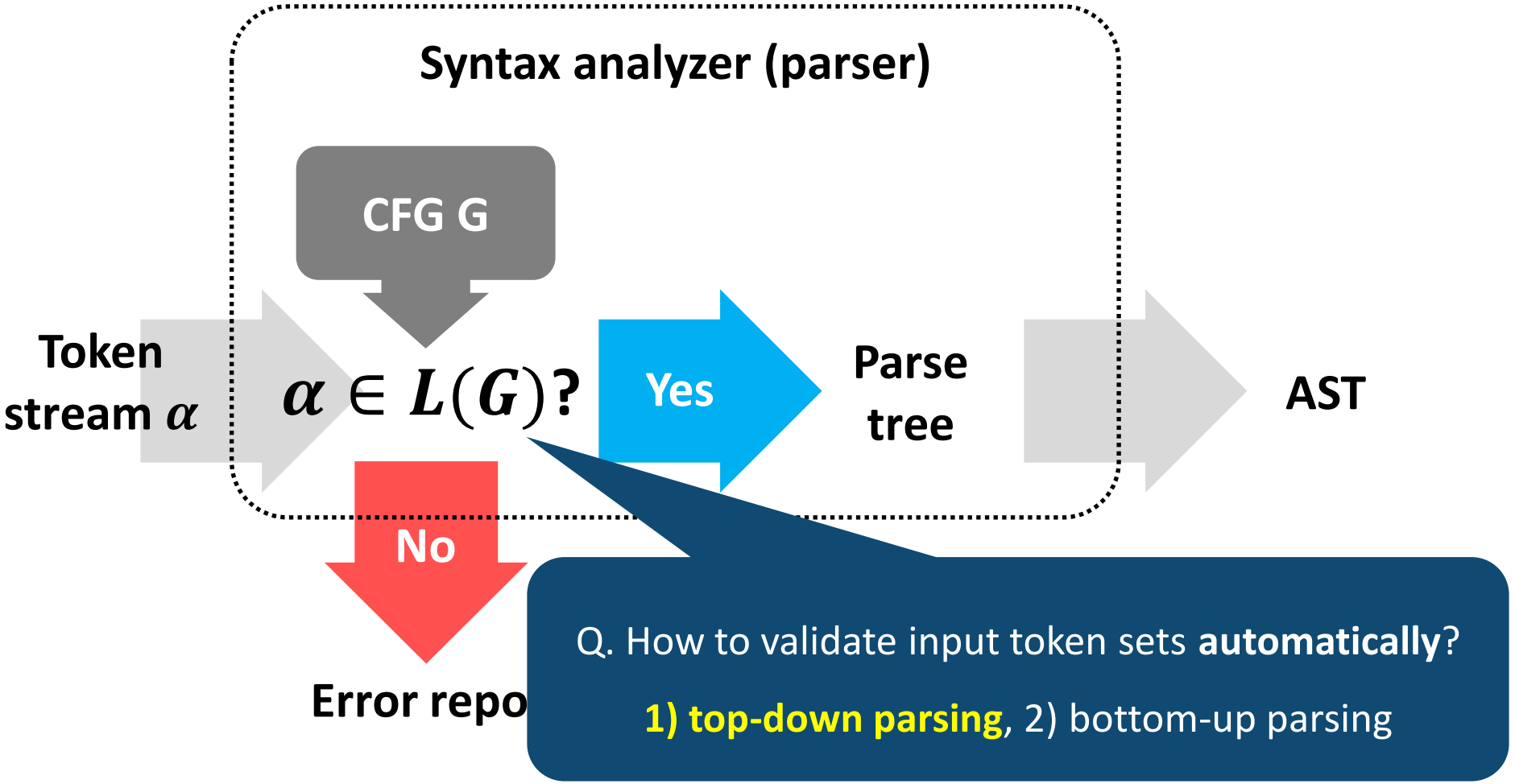
School of Computer Science and Engineering

Chung-Ang University, Seoul, Korea

https://hcslab.cau.ac.kr

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

Syntax analyzer



Reminder: 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 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

Reminder: Predictive parsing

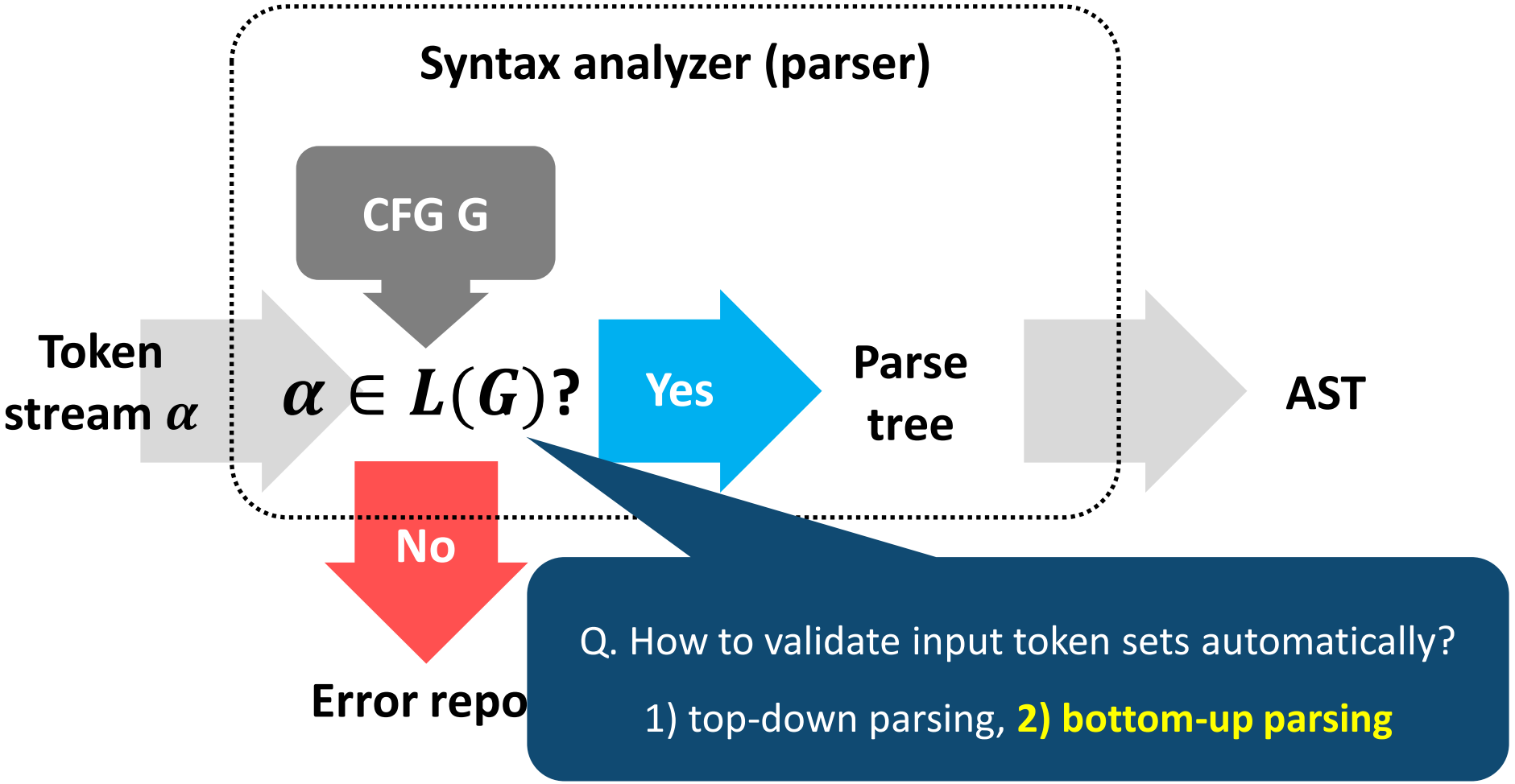
For a CFG G: $S \rightarrow \mathbf{while}(A),\ A \rightarrow BA',\ A' \rightarrow compB|\epsilon,\ B \rightarrow id|num$

(G is non-ambiguous, non-left recursive, and left-factored)

- Step 1: compute the first set and follow set for non-terminals

- Step 2: construct LL(1) parsing table

Syntax analyzer



Bottom-up parsing

Constructs a parse tree for an input string, starting from the leaves (input strings) and working up towards the root (the start symbol)

It traces **a right derivation** of the input string **in reverse: "reduction"**

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

For $id * id$

- E
- $\Rightarrow_{rm} T$
- $\Rightarrow_{rm} F * T$
- $\Rightarrow_{rm} F * F$
- $\Rightarrow_{rm} F * id$
- $\Rightarrow_{rm} id * id$

id * id

Bottom-up parsing

Constructs a parse tree for an input string, starting from the leaves (input strings) and working up towards the root (the start symbol)

It traces **a right derivation** of the input string **in reverse: "reduction"**

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

For $id * id$

- E
- $\Rightarrow_{rm} T$
- $\Rightarrow_{rm} F * T$
- $\Rightarrow_{rm} F * F$
- $\Rightarrow_{rm} \boldsymbol{F * id}$
- $\Rightarrow_{rm} id * id$

Bottom-up parsing

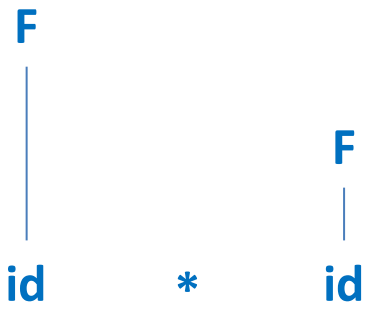
Constructs a parse tree for an input string, starting from the leaves (input strings) and working up towards the root (the start symbol)

It traces **a right derivation** of the input string **in reverse: "reduction"**

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

For $id * id$

- E
- $\Rightarrow_{rm} T$
- $\Rightarrow_{rm} F * T$
- $\Rightarrow_{rm} \boldsymbol{F * F}$
- $\Rightarrow_{rm} F * id$
- $\Rightarrow_{rm} id * id$



Bottom-up parsing

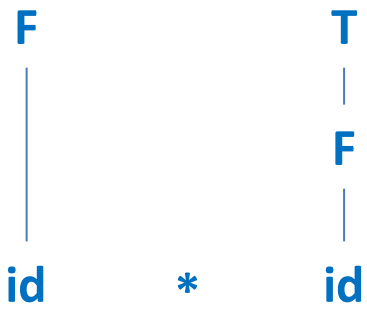
Constructs a parse tree for an input string, starting from the leaves (input strings) and working up towards the root (the start symbol)

It traces **a right derivation** of the input string **in reverse: "reduction"**

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

For $id * id$

- E
- $\Rightarrow_{rm} T$
- $\Rightarrow_{rm} F * T$
- $\Rightarrow_{rm} F * F$
- $\Rightarrow_{rm} F * id$
- $\Rightarrow_{rm} id * id$



Bottom-up parsing

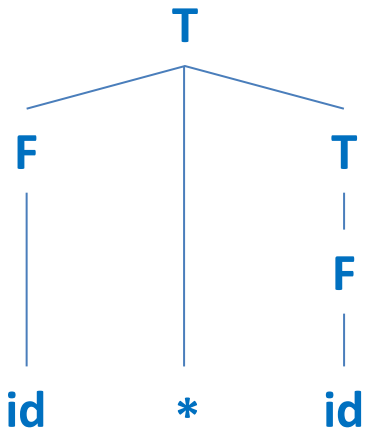
Constructs a parse tree for an input string, starting from the leaves (input strings) and working up towards the root (the start symbol)

It traces **a right derivation** of the input string **in reverse: "reduction"**

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

For $id * id$

- E
- $\Rightarrow_{rm} T$
- $\Rightarrow_{rm} F * T$
- $\Rightarrow_{rm} F * F$
- $\Rightarrow_{rm} F * id$
- $\Rightarrow_{rm} id * id$



Bottom-up parsing

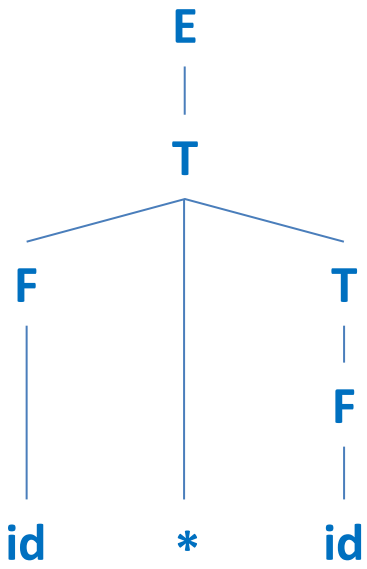
Constructs a parse tree for an input string, starting from the leaves (input strings) and working up towards the root (the start symbol)

It traces **a right derivation** of the input string **in reverse: "reduction"**

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

For $id * id$

- E
- $\Rightarrow_{rm} T$
- $\Rightarrow_{rm} F * T$
- $\Rightarrow_{rm} F * F$
- $\Rightarrow_{rm} F * id$
- $\Rightarrow_{rm} id * id$



Accept!!

Bottom-up parsing

Constructs a parse tree for an input string, starting from the leaves (input strings) and working up towards the root (the start symbol)

It traces **a right derivation** of the input string **in reverse: "reduction"**

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

Bottom-up parsing is also called LR parsing

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

The second **R** means **"rightmost derivation"**

- $\Rightarrow_{rm} F * id$
- $\Rightarrow_{rm} id * id$

Simple summary of Bottom-up parsing

Constructs a parse tree for an input string, starting from the leaves (input strings) and working up towards the root (the start symbol)

It traces **a right derivation** of the input string **in reverse: "reduction"**

- **Bottom-up parsing is more preferred than top-down parsing**

  Why? **More general!!**

  - Left factoring and left-recursive elimination are not required

    (But, grammars should be unambiguous)

Bottom-up parsing

Constructs a parse tree for an input string, starting from the leaves (input strings) and working up towards the root (the start symbol)

It traces **a right derivation** of the input string **in reverse: "reduction"**

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

For $id * id$

- E
- $\Rightarrow_{rm} T$
- $\Rightarrow_{rm} F * T$
- $\Rightarrow_{rm} F * F$
- $\Rightarrow_{rm} F * id$
- $\Rightarrow_{rm} id * id$

Q. At each reduction,

which substring should be reduced?

F can be reduced to T or id can be reduced to F

Bottom-up parsing with backtracking

$bool\ BUParsingWithBacktracking(string\ \alpha)\{$

$SStr = \{\beta | \beta\ is\ a\ substring\ of\ \alpha\ and\ \beta\ can\ be\ reduced\ by\ a\ non-terminal\}$

$(there\ is\ a\ production\ X \rightarrow \beta_i)$

$for\ each\ \beta_i \in SStr$

$replace\ \beta_i\ by\ its\ corresponding\ non-terminal\ and\ store\ the\ result\ as\ \alpha'$

$if\ (\alpha' == S)||(BUParsingWithBacktracking(\alpha') == true),\quad return\ true;$

end

$return\ false;$

$\}$

$\boldsymbol{if\ BUParsingWithBacktracking(inputString) == true,\ accept}$

$\boldsymbol{otherwise,\quad reject}$

Bottom-up parsing with backtracking

Example

$S \rightarrow dAc|cAe|cAd, \quad A \rightarrow a$

For an input string cad

- $Check\ BUParsingWithBacktracking(cad)$
  - $SStr = \{a\}\ (there\ is\ a\ production\ A \rightarrow a)$
  - $\alpha' = cAd\ (replace\ a\ in\ cad\ by\ A)$
  - $Check\ BUParsingWithBacktracking(cAd)$
    - $SStr = \{cAd\}\ (there\ is\ a\ production\ S \rightarrow cAd)$
    - $\alpha' = S\ (replace\ cAd\ in\ cAd\ by\ S)$
    - **Accept!!**

Bottom-up parsing with backtracking

Example

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

For an input string $id + id$

- $Check\ BUParsingWithBacktracking(id + id)$
  - $SStr = \{id, id\}\ (there\ is\ a\ production\ F \rightarrow id)$
  - $\alpha' = F + id\ (replace\ the\ first\ id\ in\ id + id\ by\ F)$
  - $Check\ BUParsingWithBacktracking(F + id)$
    - $SStr = \{F, id\}\ (there\ are\ productions\ T \rightarrow F\ and\ F \rightarrow id)$
    - $\alpha' = T + id\ (replace\ F\ in\ F + id\ by\ T)$
    - $Check\ BUParsingWithBacktracking(T + id)$
      - ...

Bottom-up parsing with backtracking

Advantages

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

Conditions

- A CFG is non-ambiguous
- **A CFG can be left recursive**
- **Left factoring is not needed**

But, inefficient…

Why?? Because of backtracking

Bottom-up parsing #1: Shift-reduce

Characteristics of bottom-up parsing

Let's suppose that $\alpha\beta\omega$ is the current string (sentinel form of G) during bottom-up parsing

- If the next reduction is done by $X \rightarrow \beta$, **then ω is a sequence of terminals**

Why?? Rightmost derivations (always the rightmost non-terminal is replaced)

$\boldsymbol{S \Rightarrow_{rm}^{*} \alpha X \omega \Rightarrow_{rm}}$**??**

- If ω includes any non-terminal, then $\alpha X \omega \Rightarrow_{rm} \alpha X \omega'$ (the non-terminal in ω is replaced)
- Otherwise, $\alpha X \omega \Rightarrow_{rm} \alpha\beta\omega$ (with a production $X \rightarrow \beta$)

Let's reduce an input string by examining the string from left to right

Bottom-up parsing #1: Shift-reduce

Key idea #1: Splitting a string ω into two substrings

$$\omega = \alpha \mid \beta$$

(Note: ω is a sentinel form of a CFG G)

- $\mid$: an indicator (splitter)
  - The $\mid$ is not part of the string (also, it's not the union symbol)
- α: the left substring
  - Already examined by a parser
  - A sequence of non-terminals and terminals
- β: the right substring
  - Not examined yet by a parser
  - A sequence of terminals

Bottom-up parsing #1: Shift-reduce

Key idea #2: Shifting or reducing at each step

- Initially, an input string is not examined: $|\ id * id$
- There are two actions: shift & reduce

Shift: a splitter | moves to the right

- $XaY|bcd \Rightarrow_{shift} XaYb|cd$

Reduce: the right end of the left substring is reduced

- If $Z \rightarrow Yb$ is a production of a CFG G,

  $XaYb|cd \Rightarrow_{reduce} XaZ|cd$

  - Yb is reduced as Z

Bottom-up parsing #1: Shift-reduce

Examples

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

For an input string $id * id + id$

State	Action
$\|id * id + id$	Shift
$id\| * id + id$	Reduce by $F \rightarrow id$
$F\| * id + id$	Shift
$F * \|id + id$	Shift
$F * id\| + id$	Reduce by $F \rightarrow id$
$F * F\| + id$	Reduce by $T \rightarrow F$
$F * T\| + id$	Reduce by $T \rightarrow F * T$
$T\| + id$	Shift
$T + \|id$	Shift
$T + id\|$	Reduce by $F \rightarrow id, T \rightarrow F, E \rightarrow T, E \rightarrow T + E$
$E\|$	Accept!!

Bottom-up parsing #1: Shift-reduce

The implementation of shift-reduce parsing: Use a stack!!

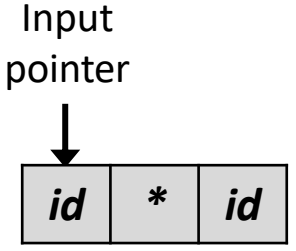
State	Action
$\|id * id$	

$$E \rightarrow T + E | T$$

$$T \rightarrow F * T | F,$$

$$F \rightarrow (E) | id$$

Input pointer



Bottom-up parsing #1: Shift-reduce

The implementation of shift-reduce parsing: Use a stack!!

- For **"shift"**
  - Push the **target** terminal of an input string into the stack
  - Advance the input pointer

State	Action
$\|id * id$	Shift
$id\| * id$	

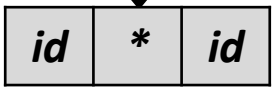
Stack
id

$$E \rightarrow T + E | T$$

$$T \rightarrow F * T | F,$$

$$F \rightarrow (E) | id$$

Input pointer

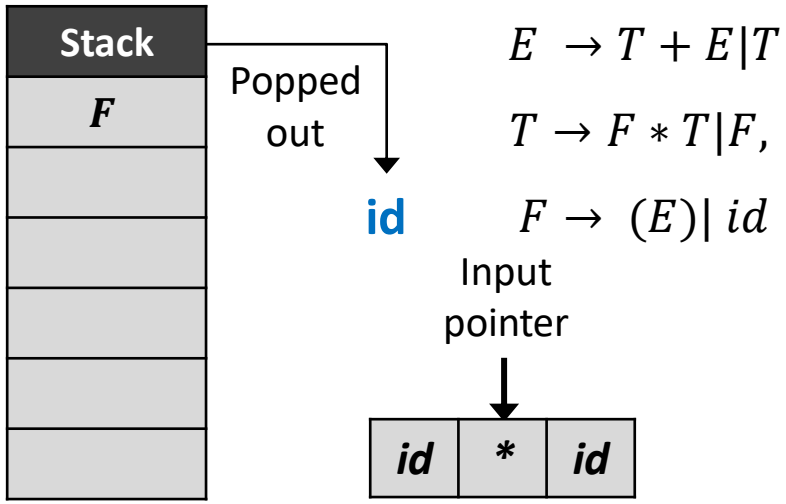


Bottom-up parsing #1: Shift-reduce

The implementation of shift-reduce parsing: Use a stack!!

- For **"shift"**
  - Push the **target** terminal of an input string into the stack
  - Advance the input pointer
- For **"reduce"**: Pop the string will be reduced from the stack and Push the result into the stack

State	Action
$\|id * id$	Shift
$id\| * id$	Reduce by $F \rightarrow id$
$F\| * id$	



Bottom-up parsing #1: Shift-reduce

The implementation of shift-reduce parsing: Use a stack!!

- For **"shift"**
  - Push the **target** terminal of an input string into the stack
  - Advance the input pointer
- For **"reduce"**: Pop the string will be reduced from the stack and Push the result into the stack

State	Action
$\|id * id$	Shift
$id \| * id$	Reduce by $F \rightarrow id$
$F \| * id$	Shift
$F * \|id$	Shift
$F * id \|$	

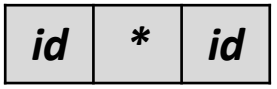
Stack
id
*****
F

$$E \rightarrow T + E \mid T$$

$$T \rightarrow F * T \mid F,$$

$$F \rightarrow (E) \mid id$$

Input pointer

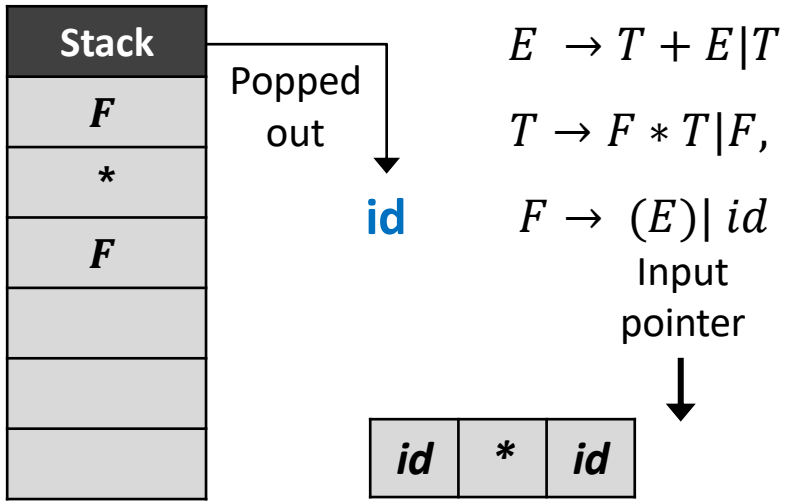


Bottom-up parsing #1: Shift-reduce

The implementation of shift-reduce parsing: Use a stack!!

- For **"shift"**
  - Push the **target** terminal of an input string into the stack
  - Advance the input pointer
- For **"reduce"**: Pop the string will be reduced from the stack and Push the result into the stack

State	Action
$\|id * id$	Shift
$id\| * id$	Reduce by $F \rightarrow id$
$F\| * id$	Shift
$F * \|id$	Shift
$F * id\|$	Reduce by $F \rightarrow id$
$F * F\|$	

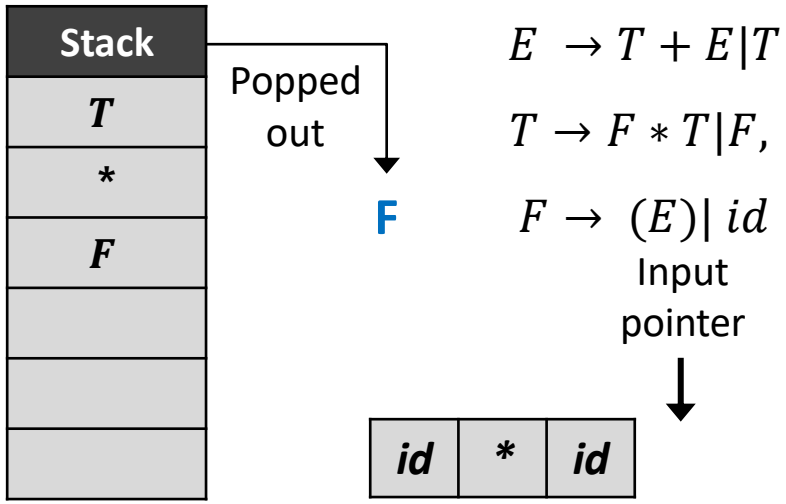


Bottom-up parsing #1: Shift-reduce

The implementation of shift-reduce parsing: Use a stack!!

- For **"shift"**
  - Push the **target** terminal of an input string into the stack
  - Advance the input pointer
- For **"reduce"**: Pop the string will be reduced from the stack and Push the result into the stack

State	Action
$\|id * id$	Shift
$id \| * id$	Reduce by $F \rightarrow id$
$F \| * id$	Shift
$F * \|id$	Shift
$F * id \|$	Reduce by $F \rightarrow id$
$F * F \|$	Reduce by $T \rightarrow F$
$F * T \|$	

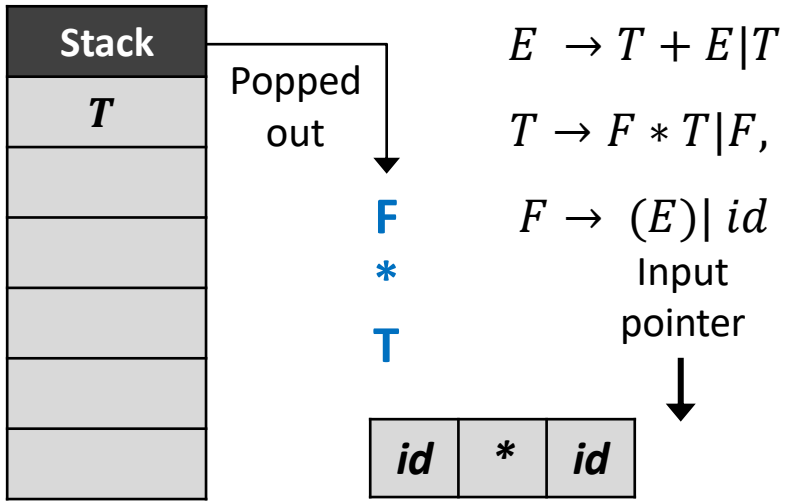


Bottom-up parsing #1: Shift-reduce

The implementation of shift-reduce parsing: Use a stack!!

- For **"shift"**
  - Push the **target** terminal of an input string into the stack
  - Advance the input pointer
- For **"reduce"**: Pop the string will be reduced from the stack and Push the result into the stack

State	Action
$\|id * id$	Shift
$id\| * id$	Reduce by $F \rightarrow id$
$F\| * id$	Shift
$F * \|id$	Shift
$F * id\|$	Reduce by $F \rightarrow id$
$F * F\|$	Reduce by $T \rightarrow F$
$F * T\|$	Reduce by $T \rightarrow F * T$
T|	

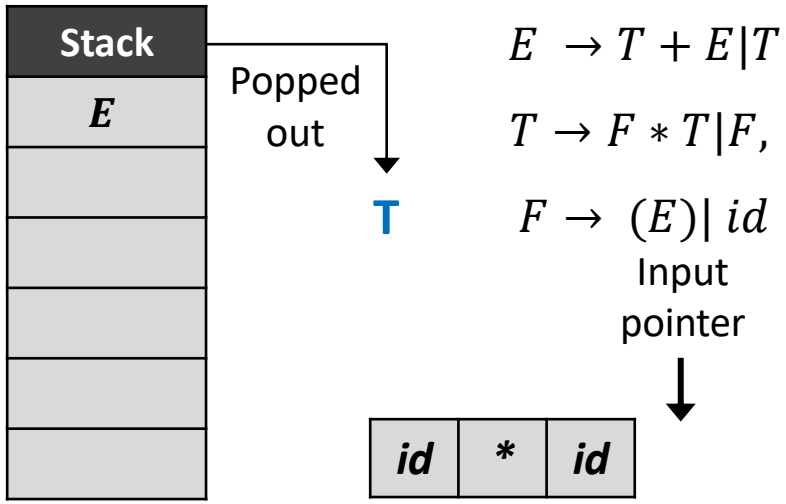


Bottom-up parsing #1: Shift-reduce

The implementation of shift-reduce parsing: Use a stack!!

- For **"shift"**
  - Push the **target** terminal of an input string into the stack
  - Advance the input pointer
- For **"reduce"**: Pop the string will be reduced from the stack and Push the result into the stack

State	Action
$\|id * id$	Shift
$id \| * id$	Reduce by $F \rightarrow id$
$F \| * id$	Shift
$F * \|id$	Shift
$F * id \|$	Reduce by $F \rightarrow id$
$F * F \|$	Reduce by $T \rightarrow F$
$F * T \|$	Reduce by $T \rightarrow F * T$
T|	Reduce by $E \rightarrow T$
E|	Accept



Simple summary of Bottom-up parsing

Constructs a parse tree for an input string, starting from the leaves (input strings) and working up towards the root (the start symbol)

It traces **a right derivation** of the input string **in reverse: "reduction"**

Shift-reduce parsing

- $\alpha \mid \beta$
- e.g., For a CFG G,
  - $E \rightarrow T + E | T,\ T \rightarrow F * T | F,\ F \rightarrow (E) | id$
  - For an input string $id * id$

State	Action
$\|id * id$	Shift
$id \| * id$	Reduce by $F \rightarrow id$
$F \| * id$	Shift
$F * \|id$	Shift
$F * id \|$	Reduce by $F \rightarrow id$
$F * F \|$	Reduce by $T \rightarrow F$
$F * T \|$	Reduce by $T \rightarrow F * T$
$T \|$	Reduce by $E \rightarrow T$
$E \|$	Accept

Simple summary of Bottom-up parsing

But, there are still two types of conflicts

1. Shift-reduce conflict

At each step, we should decide whether to shift or reduce

- e.g., If a production $A \rightarrow \alpha$ exists, what should we do with $\boldsymbol{\alpha} \mid \boldsymbol{\beta}$??

State	Action
$id \mid * id$	Shift?? Reduce by $F \rightarrow id$??

2. Reduce-reduce conflict

In reduction, we should decide which reduction to be used

- e.g., If two productions $A \rightarrow \alpha$ and $B \rightarrow \alpha$ exist, which reduction should be used for $\boldsymbol{\alpha} \mid \boldsymbol{\beta}$??

State	Action
$F * T \mid$	Reduce by $T \rightarrow F * T$?? Reduce by $E \rightarrow T$??