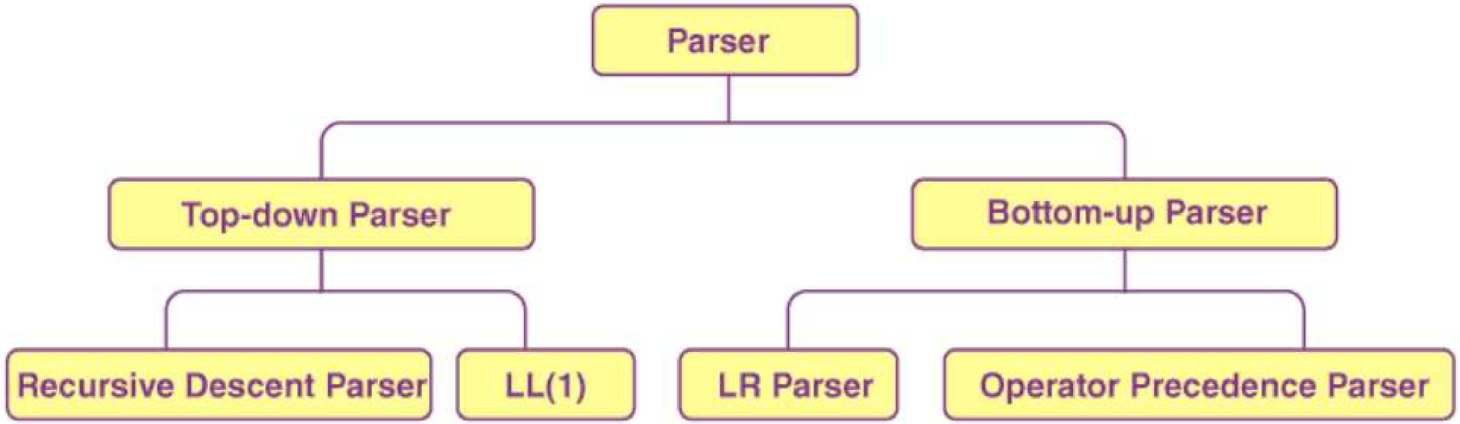
Compiler Design

BoTtOm-Up PaRsInG

InTrO..



- Bottom-UP-Parsing
  - Construct a parse tree for an input string beginning at leaves and going towards root

    OR

  - Reduce a string **w** of input to start symbol of grammar **S**

BoTtOm-Up PaRsInG

- Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
  - Preferred method in practice
- Also called LR parsing
  - **L** means that tokens are read left to right
  - **R** means that it constructs a rightmost derivation !

An Introductory Example

- LR parsers don't need left-factored grammars and can also handle left-recursive grammars
- Consider the following grammar: E → E + (E) | int
  - Why is this not LL(1)?

An **expression (E)** can be: Another expression followed by + (E), or Just an **int.**

Here, E → E + (E) is **left-recursive**, so LL(1) fails.

Consider the string:

int + (int) + (int)

An LL(1) parser would face difficulty when it sees the first int:

- Should it expand E → int or try E → E + (E)?
- With only 1 symbol lookahead, it cannot decide.

An LR parser doesn't guess.

- It shifts **int,** then reduces it to **E,**
- then when it sees + (int) ..., it recognizes the pattern E + (E) and reduces correctly.
- Finally, it parses the whole input without problems.

Contd.,

- Consider a grammar

S → aABe
A → Abc | b
B → d

Input string : abbcde

aAbcde
aAde
aABe
S

The sentential forms happen to be a right most derivation in the reverse order

Right Most Derivation →→→→→

S → a A B e
→ a A d e
→ a A b c d e
→ a b b c d e

Contd.,

$S \rightarrow aABb$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

input string: aaabb

aaAbb

aAbb $\Downarrow$ reduction

aABb

S

$$S \underset{rm}{\Rightarrow} aABb \underset{rm}{\Rightarrow} aAbb \underset{rm}{\Rightarrow} aaAbb \underset{rm}{\Rightarrow} aaabb$$

Right Sentential Forms

ThE iDeA

- LR parsing reduces a string to the start symbol by inverting (**reversing**) productions.
  - str **w** input string of terminals
  - Repeat
    - Identify **β** in **str** such that **A →β** is a production (**i.e., str = αβγ**)
    - Replace **β** by **A** in str (i.e., str w = α A γ)
    - until str = **S** (the start symbol)

      OR all possibilities are exhausted

A Bottom-up Parse in Detail (1)

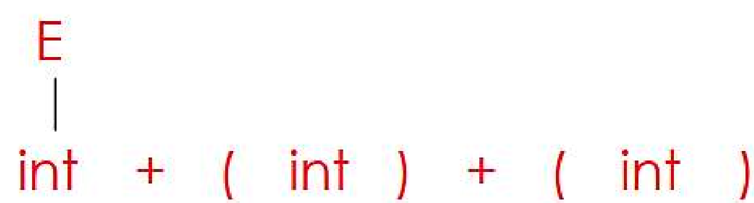
int + (int) + (int)

int + (int) + (int)

A Bottom-up Parse in Detail (2)

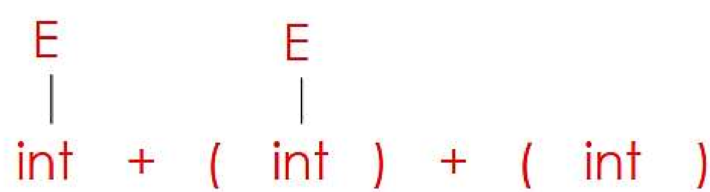
int + (int) + (int)

E + (int) + (int)



A Bottom-up Parse in Detail (3)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)



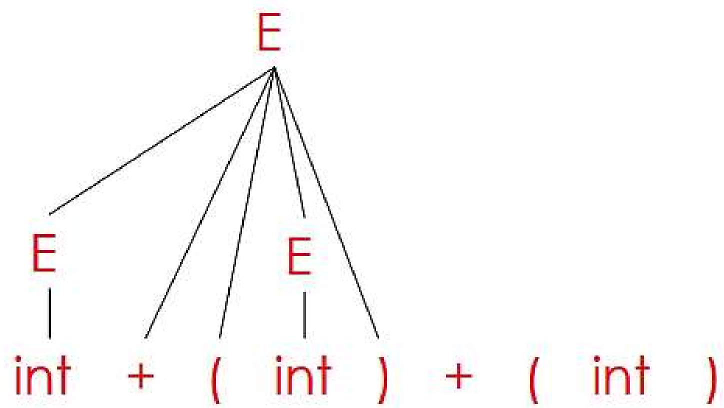
A Bottom-up Parse in Detail (4)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)



A Bottom-up Parse in Detail (5)

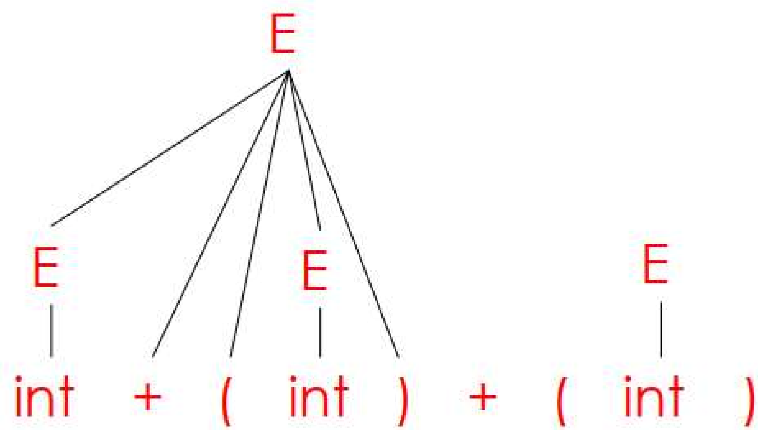
int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)

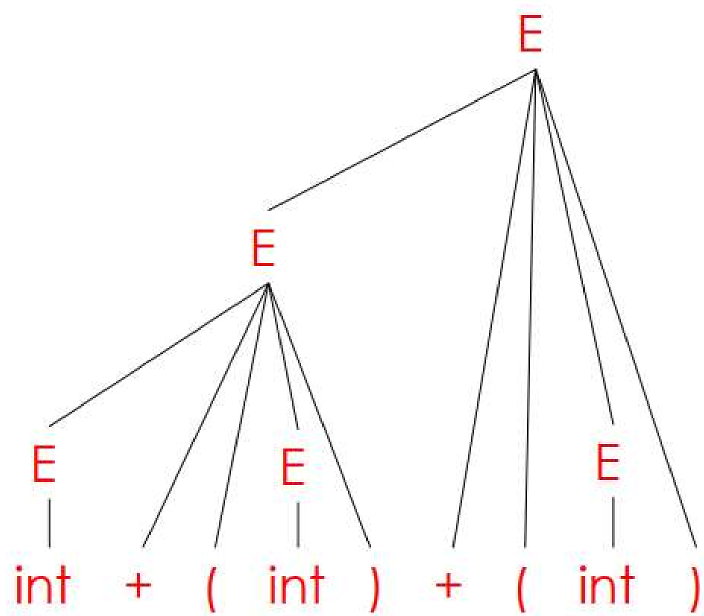
E + (E)



A Bottom-up Parse in Detail (6)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)
E

A rightmost derivation in reverse



Important Fact #1

- Important Fact #1 about bottom-up parsing:

An LR parser traces a rightmost derivation in reverse

Where Do Reductions Happen

- Important Fact #1 has an interesting consequence:
  - Let $\alpha\beta\gamma$ be a step of a bottom-up parse
  - Assume the next reduction is by using $A \rightarrow \beta$
  - Then γ is a string of terminals
- Why? Because $\alpha A\gamma \rightarrow \alpha\beta\gamma$ is a step in a rightmost derivation

Idea: Notation

- Split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)
  - Left substring has terminals and non-terminals
- The dividing point is marked by a I
  - The I is not part of the string
- Initially, all input is unexamined: Ix1, x2, . . . xn

Shift-Reduce Parsing

- Bottom-up parsing uses only two kinds of actions: **Shift and Reduce**

Shift: Move | one place to the right

- Shifts a terminal to the left string

$$E + (\,|\, \text{int}\,) \Rightarrow E + (\text{int}\,|\,)$$

Shift

←←←←←←←

In general:

$$ABC \,|\, xyz \Rightarrow ABCx \,|\, yz$$

Reduce: Apply an inverse production at the right end of the left string

- If $E \to E + (E)$ is a production, then

$$E + (\underline{E + (E)} \,|\,) \Rightarrow E + (\underline{E} \,|\,)$$

Reduce

→→→→→→→

In general, given $A \to xy$, then:

$$Cbxy \,|\, ijk \Rightarrow CbA \,|\, ijk$$

Shift-Reduce Example

$E \rightarrow E + (E) \mid int$

| int + (int) + (int)\$ *shift*

int + (int) + (int)

Shift-Reduce Example

$E \rightarrow E + (E) \mid int$

| int + (int) + (int)$ shift

int | + (int) + (int)$ reduce $E \rightarrow int$

int   +   (int   ) +   (int   )

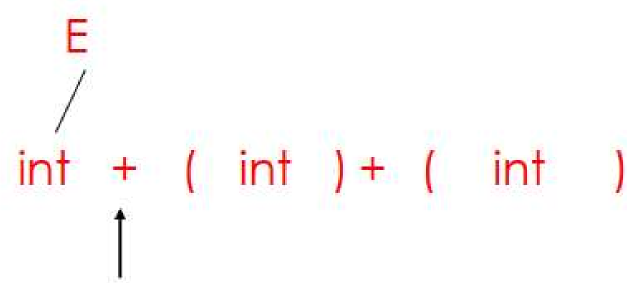
Shift-Reduce Example

$E \rightarrow E + (E) \mid int$

| int + (int) + (int)\$ *shift*

int | + (int) + (int)\$ *reduce E → int*

E | + (int) + (int)\$ *shift 3 times*



Shift-Reduce Example

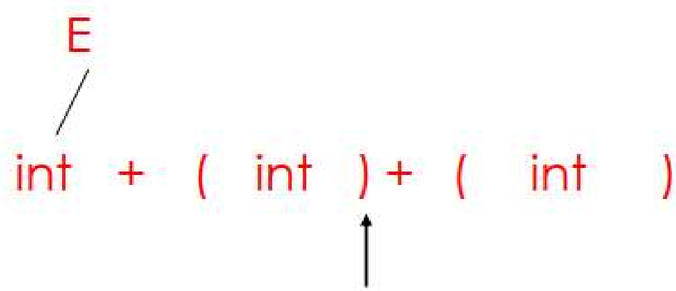
$E \rightarrow E + (E) \mid int$

| int + (int) + (int)\$ shift

int | + (int) + (int)\$ reduce $E \rightarrow int$

E | + (int) + (int)\$ shift 3 times

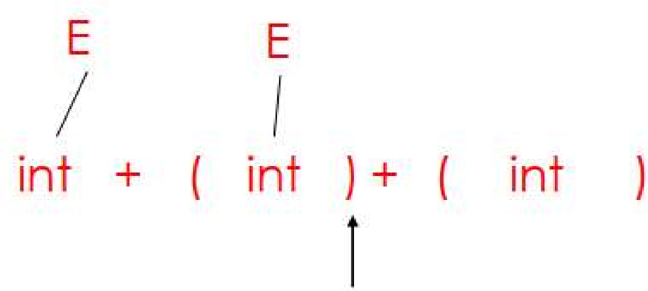
E + (int |) + (int)\$ reduce $E \rightarrow int$



Shift-Reduce Example

E → E + (E) | int

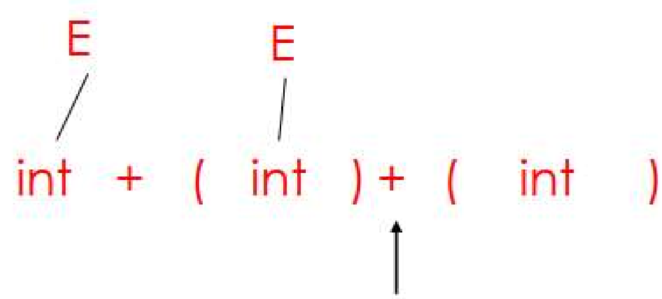
| int + (int) + (int)$	shift
int | + (int) + (int)$	reduce E → int
E | + (int) + (int)$	shift 3 times
E + (int |) + (int)$	reduce E → int
E + (E |) + (int)$	shift



Shift-Reduce Example

E → E + (E) | int

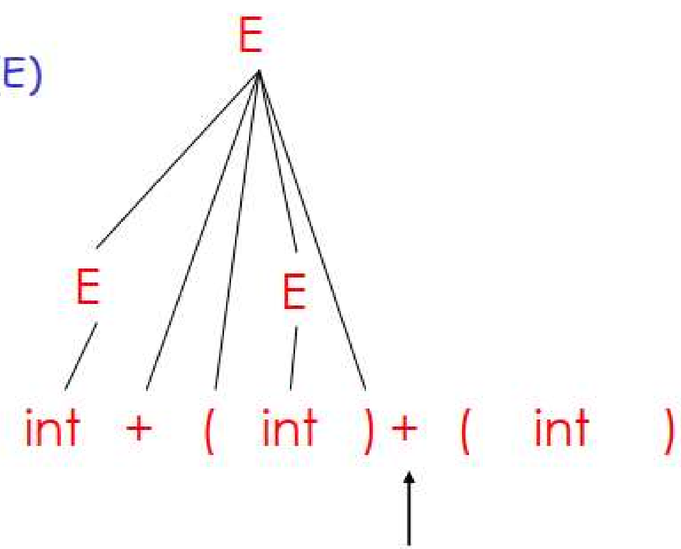
| int + (int) + (int)$	shift
int | + (int) + (int)$	reduce E → int
E | + (int) + (int)$	shift 3 times
E + (int |) + (int)$	reduce E → int
E + (E |) + (int)$	shift
E + (E) | + (int)$	reduce E → E + (E)



Shift-Reduce Example

$E \rightarrow E + (E) \mid int$

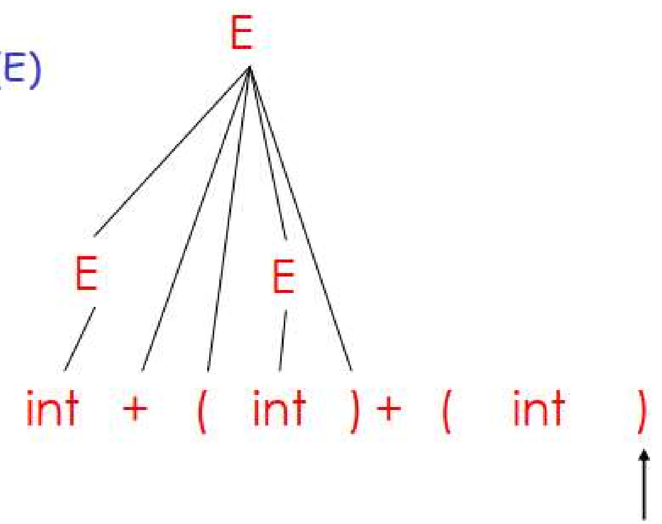
| int + (int) + (int)\$	shift
int | + (int) + (int)\$	reduce $E \rightarrow int$
E | + (int) + (int)\$	shift 3 times
E + (int |) + (int)\$	reduce $E \rightarrow int$
E + (E |) + (int)\$	shift
E + (E) | + (int)\$	reduce $E \rightarrow E + (E)$
E | + (int)\$	shift 3 times



Shift-Reduce Example

$E \to E + (E) \mid int$

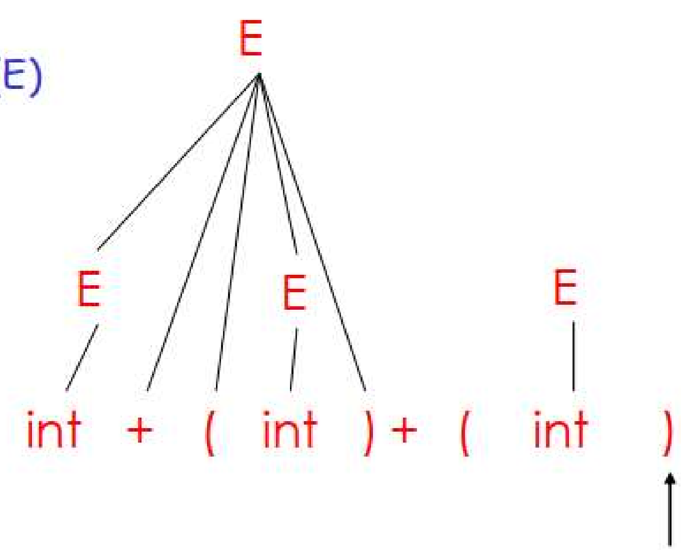
| int + (int) + (int)$ shift
int | + (int) + (int)$ reduce $E \to int$
E | + (int) + (int)$ shift 3 times
E + (int |) + (int)$ reduce $E \to int$
E + (E |) + (int)$ shift
E + (E) | + (int)$ reduce $E \to E + (E)$
E | + (int)$ shift 3 times
E + (int |)$ reduce $E \to int$



Shift-Reduce Example

E → E + (E) | int

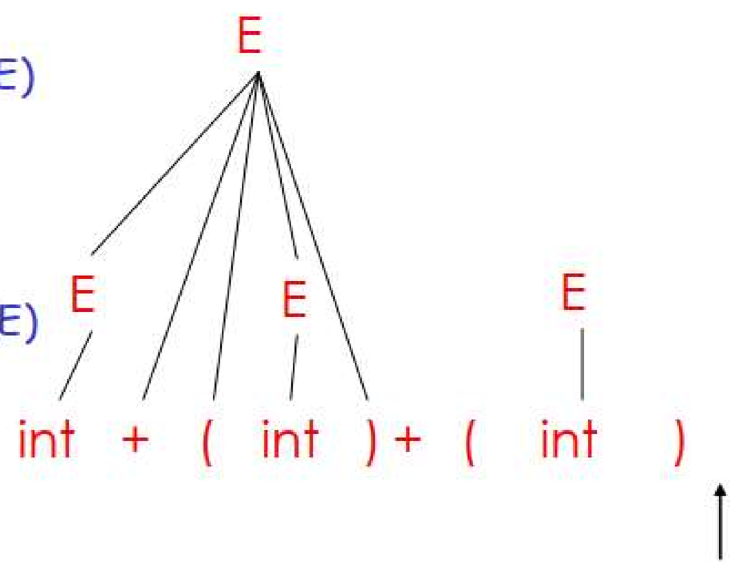
<table>
<tr><td>| int + (int) + (int)$</td><td>shift</td></tr>
<tr><td>int | + (int) + (int)$</td><td>reduce E → int</td></tr>
<tr><td>E | + (int) + (int)$</td><td>shift 3 times</td></tr>
<tr><td>E + (int |) + (int)$</td><td>reduce E → int</td></tr>
<tr><td>E + (E |) + (int)$</td><td>shift</td></tr>
<tr><td>E + (E) | + (int)$</td><td>reduce E → E + (E)</td></tr>
<tr><td>E | + (int)$</td><td>shift 3 times</td></tr>
<tr><td>E + (int |)$</td><td>reduce E → int</td></tr>
<tr><td>E + (E |)$</td><td>shift</td></tr>
</table>



Shift-Reduce Example

$E \rightarrow E + (E) \mid int$

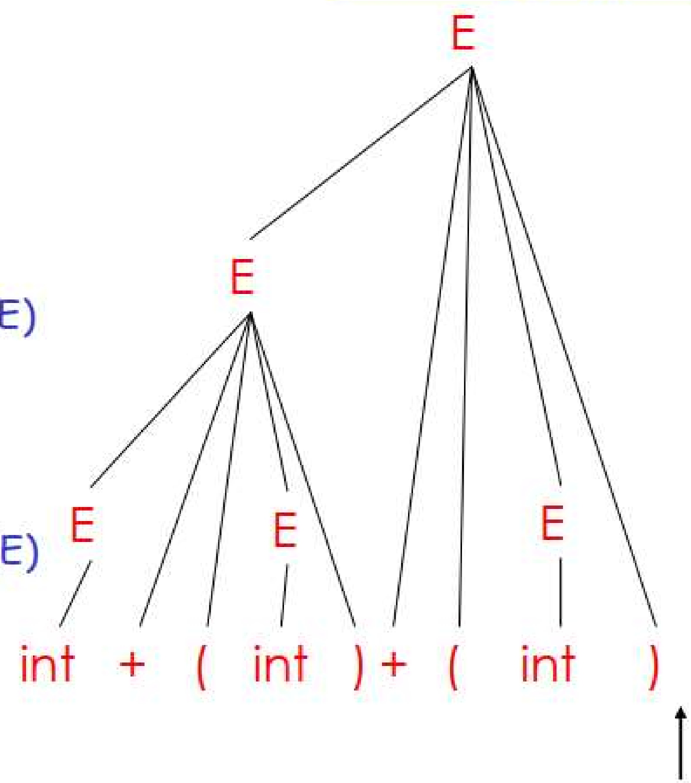
I int + (int) + (int)$	shift
int I + (int) + (int)$	reduce $E \rightarrow int$
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	reduce $E \rightarrow int$
E + (E I) + (int)$	shift
E + (E) I + (int)$	reduce $E \rightarrow E + (E)$
E I + (int)$	shift 3 times
E + (int I)$	reduce $E \rightarrow int$
E + (E I)$	shift
E + (E) I $	reduce $E \rightarrow E + (E)$



Shift-Reduce Example

$E \rightarrow E + (E) \mid int$

$\mid$ int + (int) + (int)\$	shift
int $\mid$ + (int) + (int)\$	reduce $E \rightarrow int$
E $\mid$ + (int) + (int)\$	shift 3 times
E + (int $\mid$) + (int)\$	reduce $E \rightarrow int$
E + (E $\mid$) + (int)\$	shift
E + (E) $\mid$ + (int)\$	reduce $E \rightarrow E + (E)$
E $\mid$ + (int)\$	shift 3 times
E + (int $\mid$)\$	reduce $E \rightarrow int$
E + (E $\mid$)\$	shift
E + (E) $\mid$ \$	reduce $E \rightarrow E + (E)$
E $\mid$ \$	accept



A Bottom-up Parse... Example (2)

E → T + E | T
T → int * T | int | (E)

Bottom-up parsing reduces a string to the start symbol by inverting productions:

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	

int * int + int

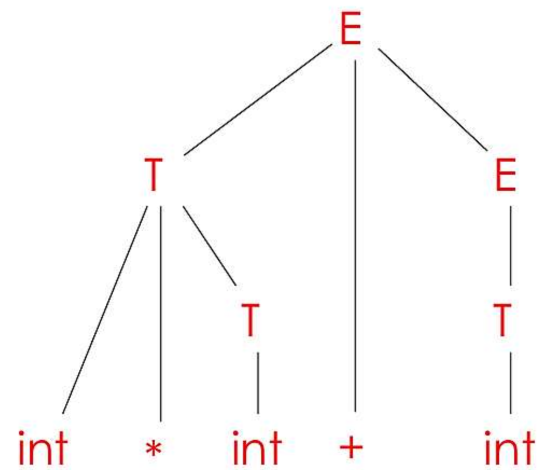
int * T + int

T + int

T + T

T + E

E



A Bottom-up Parse Ex 2 in Detail (1)

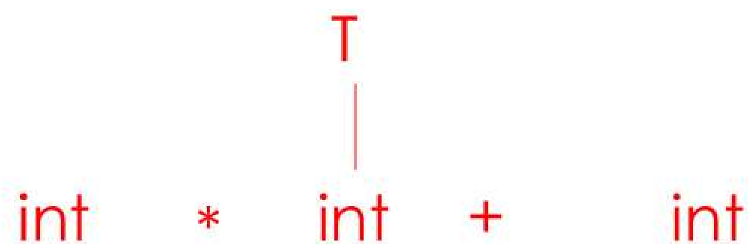
int * int + int

int * int + int

A Bottom-up Parse Ex 2 in Detail (2)

int * int + int

int * T + int

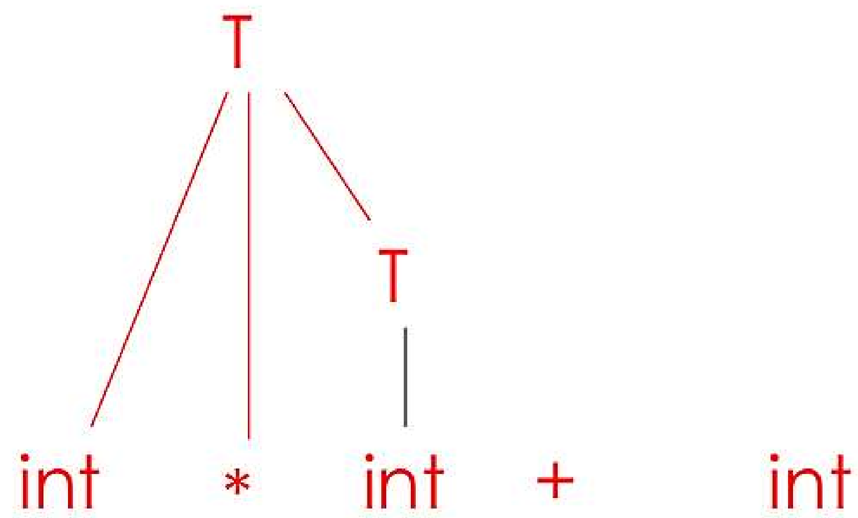


A Bottom-up Parse Ex 2 in Detail (3)

int * int + int

int * T + int

T + int



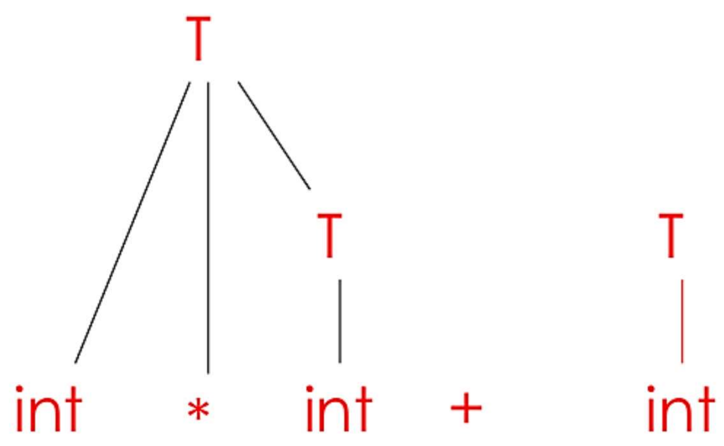
A Bottom-up Parse Ex 2 in Detail (4)

int * int + int

int * T + int

T + int

T + T



A Bottom-up Parse Ex 2 in Detail (5)

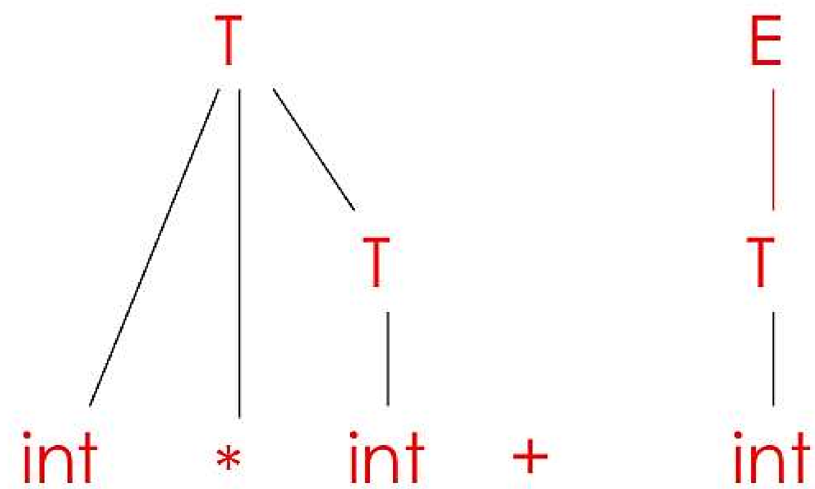
int * int + int

int * T + int

T + int

T + T

T + E



A Bottom-up Parse Ex 2 in Detail (6)

int * int + int

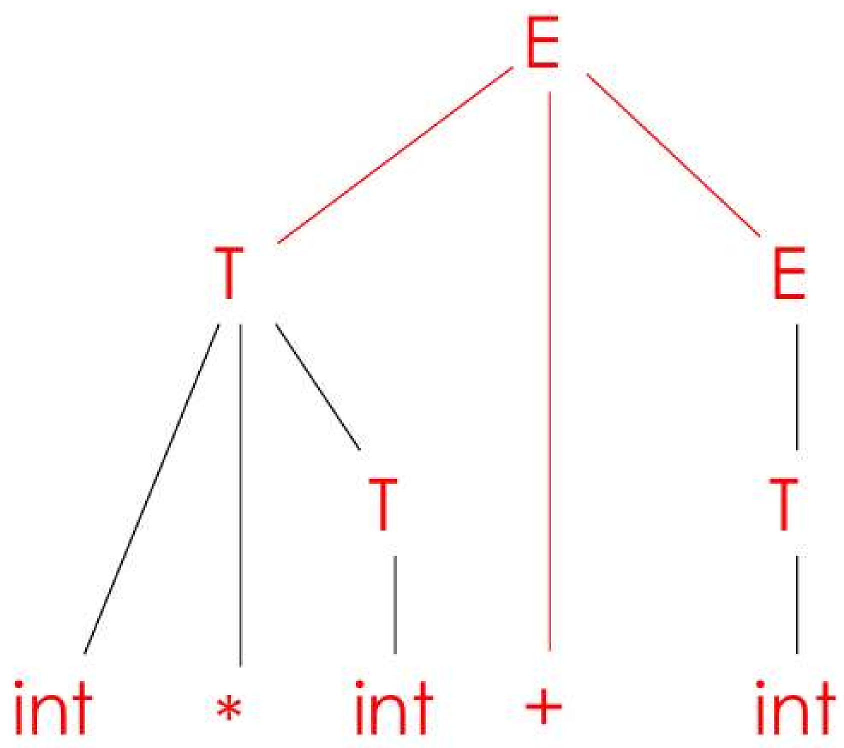
int * T + int

T + int

T + T

T + E

E



The Example with Shift-Reduce Parsing

$\mid$ int * int + int	shift
int $\mid$ * int + int	shift
int * $\mid$ int + int	shift
int * int $\mid$ + int	reduce T → int
int * T $\mid$ + int	reduce T → int * T
T $\mid$ + int	shift
T + $\mid$ int	shift
T + int $\mid$	reduce T → int
T + T $\mid$	reduce E → T
T + E $\mid$	reduce E → T + E
E $\mid$	

A Shift-Reduce Parse in Detail (1)

| int * int + int

int * int + int

↑

A Shift-Reduce Parse in Detail (2)

| int * int + int

int | * int + int

int * int + int

↑

A Shift-Reduce Parse in Detail (3)

| int * int + int

int | * int + int

int * | int + int

int * int + int

↑

A Shift-Reduce Parse in Detail (4)

| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * int + int

↑

A Shift-Reduce Parse in Detail (5)

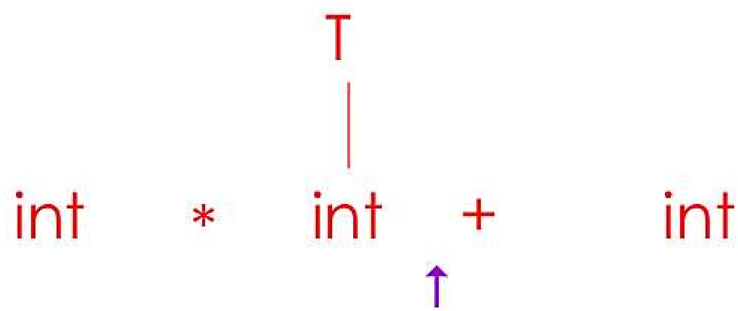
| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int



A Shift-Reduce Parse in Detail (6)

I int * int + int

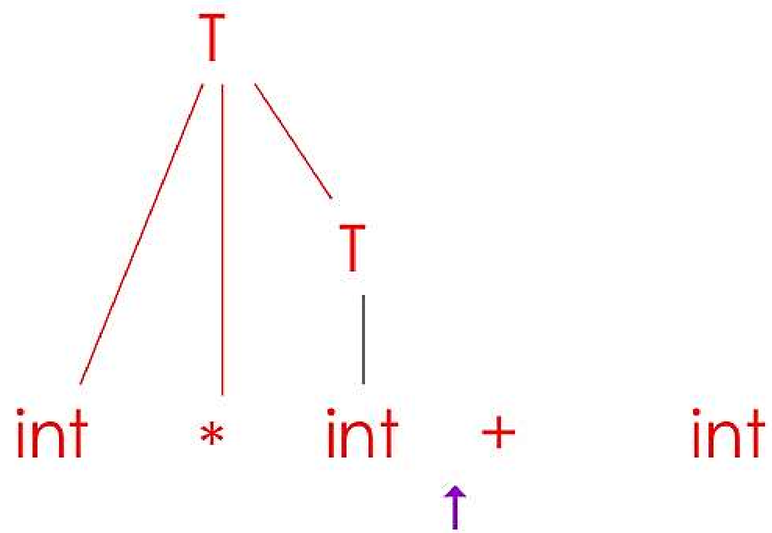
int I * int + int

int * I int + int

int * int I + int

int * T I + int

T I + int



A Shift-Reduce Parse in Detail (7)

I int * int + int

int I * int + int

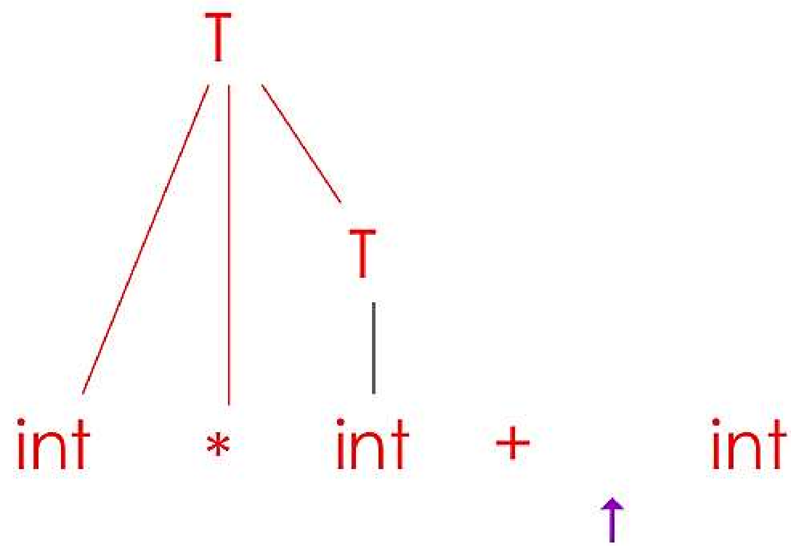
int * I int + int

int * int I + int

int * T I + int

T I + int

T + I int



A Shift-Reduce Parse in Detail (8)

I int * int + int

int I * int + int

int * I int + int

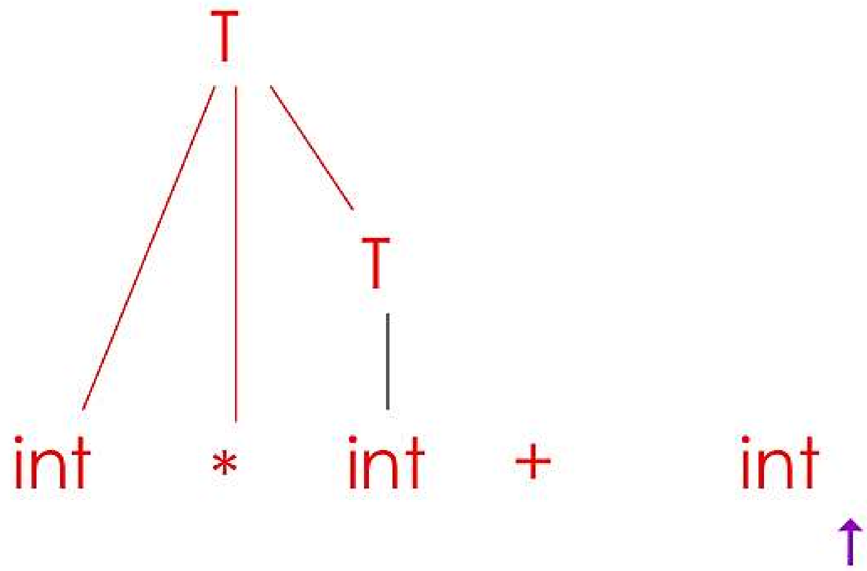
int * int I + int

int * T I + int

T I + int

T + I int

T + int I



A Shift-Reduce Parse in Detail (9)

| int * int + int

int | * int + int

int * | int + int

int * int | + int

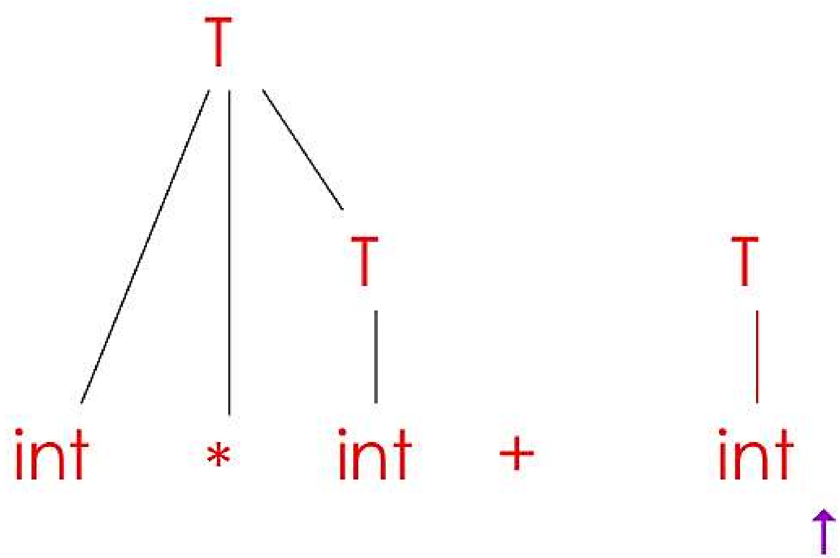
int * T | + int

T | + int

T + | int

T + int |

T + T |



A Shift-Reduce Parse in Detail (10)

I int * int + int

int I * int + int

int * I int + int

int * int I + int

int * T I + int

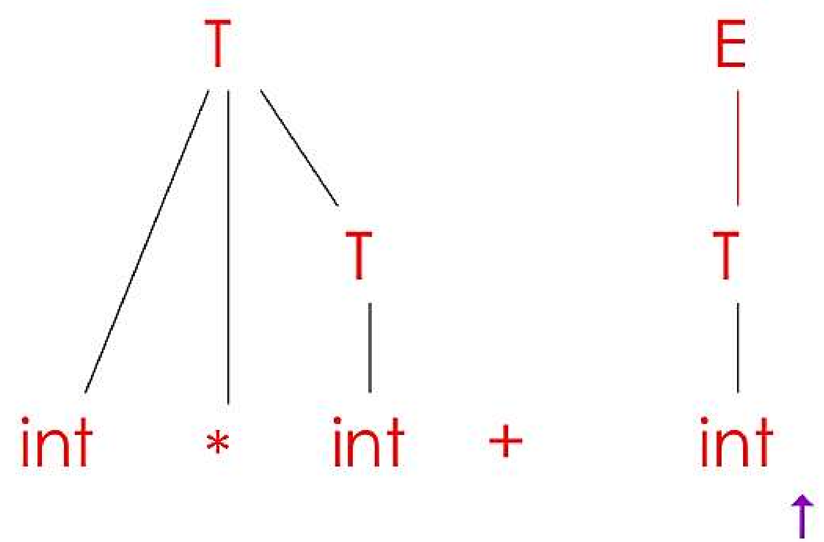
T I + int

T + I int

T + int I

T + T I

T + E I



A Shift-Reduce Parse in Detail (11)

I int * int + int

int I * int + int

int * I int + int

int * int I + int

int * T I + int

T I + int

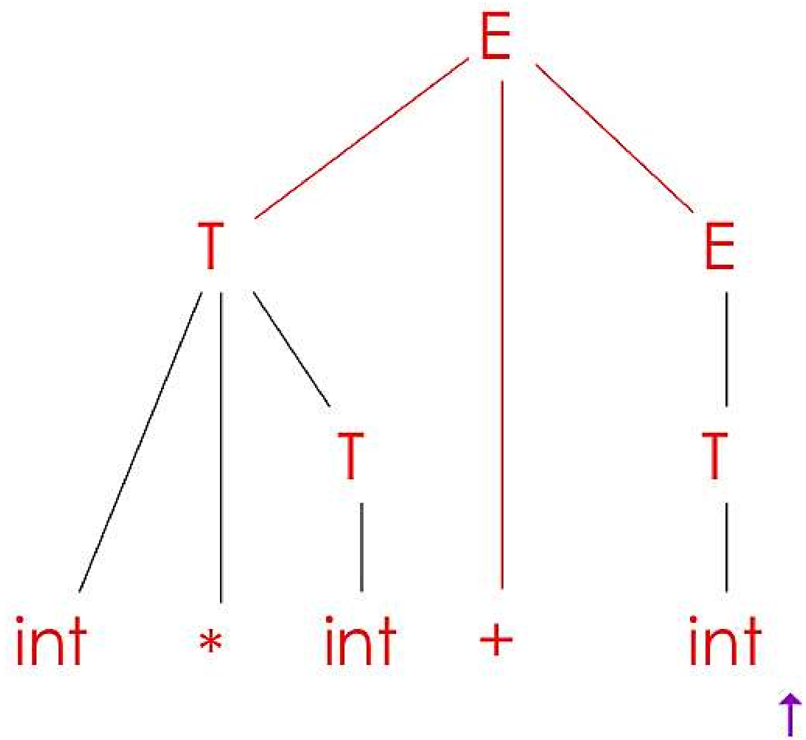
T + I int

T + int I

T + T I

T + E I

E I



A Stack Implementation of a Shift-Reduce Parser

- There are four possible actions of a shift-parser action:
  - Shift : The next input symbol is shifted onto the top of the stack.
  - Reduce: Replace the handle on the top of the stack by the non-terminal.
  - Accept: Successful completion of parsing.
  - Error: Parser discovers a syntax error, and calls an error recovery routine.

- Initial stack just contains only the end-marker $.
- The end of the input string is marked by the end-marker $