Top-Down Parsing and Intro to Bottom-Up Parsing

Lecture 4

Announcements

- PA1
  - Due today (30 Jan.) at 11:59pm
- PA2
  - Assigned today
- WA2
  - Assigned today

Predictive Parsers

- Parser can "predict" which production to use
  - By looking at the next few tokens
  - No backtracking

- Predictive parsers accept LL(k) grammars
  - L means "left-to-right" scan of input
  - L means "leftmost derivation"
  - k means "predict based on k tokens of lookahead"
  - In practice, LL(1) is used

LL(1) vs. Recursive Descent

- In recursive-descent,
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices
- In LL(1),
  - At each step, only one choice of production
  - That is
    - When a non-terminal A is leftmost in a derivation
    - The next input symbol is t
    - There is a unique production $A \rightarrow \alpha$ to use
      - Or no production to use (an error state)
- LL(1) is a recursive descent variant without backtracking

Predictive Parsing and Left Factoring

- Recall the grammar

  $E \rightarrow T + E \mid T$

  $T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

- Hard to predict because
  - For T two productions start with int
  - For E it is not clear how to predict

- We need to left-factor the grammar

Left-Factoring Example

- Recall the grammar

$$E \rightarrow T + E \mid T$$
$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

- Factor out common prefixes of productions

$$E \rightarrow T X$$
$$X \rightarrow + E \mid \varepsilon$$
$$T \rightarrow (E) \mid \text{int } Y$$
$$Y \rightarrow * T \mid \varepsilon$$

LL(1) Parsing Table Example

- Left-factored grammar

  $E \rightarrow T\ X$ $\qquad$ $X \rightarrow +\ E \mid \varepsilon$

  $T \rightarrow (\ E\ ) \mid \text{int}\ Y$ $\qquad$ $Y \rightarrow *\ T \mid \varepsilon$

- The LL(1) parsing table:

next input token

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

leftmost non-terminal

rhs of production to use

LL(1) Parsing Table Example (Cont.)

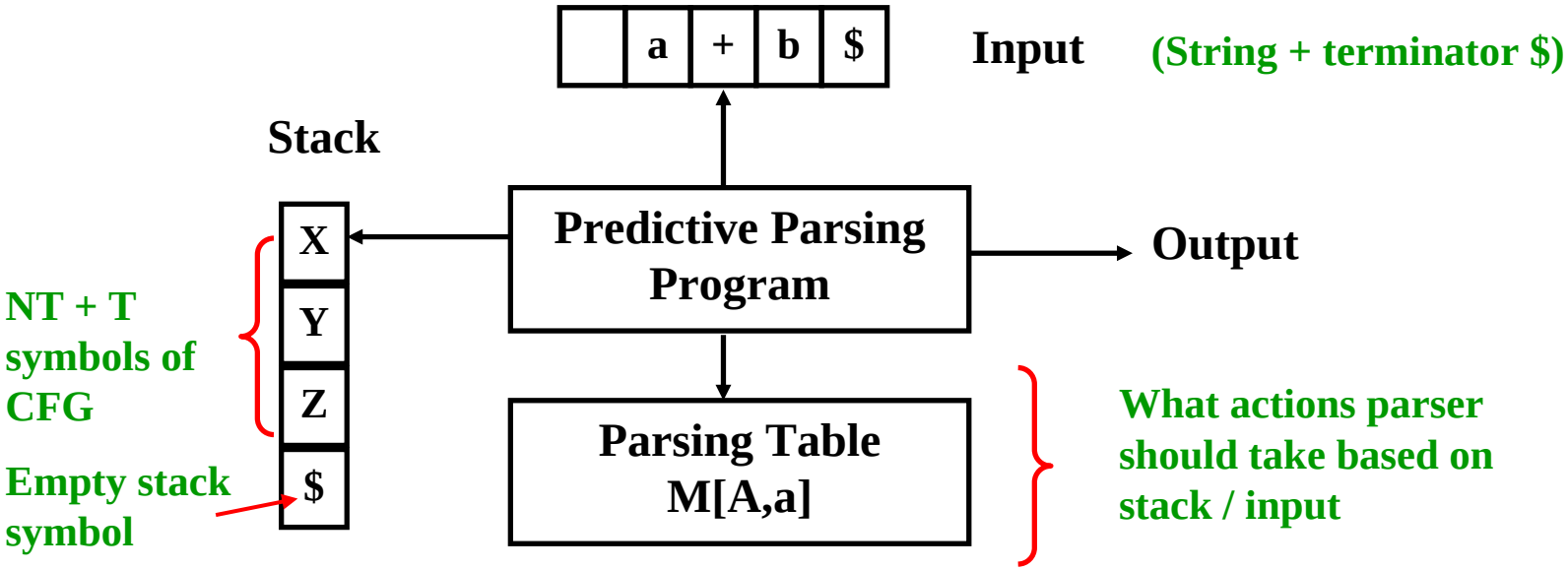
- Consider the [E, int] entry
  - "When current non-terminal is E and next input is int, use production $E \rightarrow T\ X$"
  - This can generate an int in the first position

- Consider the [Y,+] entry
  - "When current non-terminal is Y and current token is +, get rid of Y"
  - Y can be followed by + only if $Y \rightarrow \varepsilon$

LL(1) Parsing Tables. Errors

- Blank entries indicate error situations
- Consider the [E,*] entry
  - "There is no way to derive a string starting with * from non-terminal E"

LL(1) Parsing Algorithm



General parser behavior: **X : top of stack** **a : current token**

1. When X=a = $ halt, accept, success
2. When X=a ≠ $, POP X off stack, advance input, go to 1.
3. When X is a non-terminal, examine M[X, a], if it is an error, call recovery routine if M[X, a] = {UVW}, POP X, PUSH U,V,W, and DO NOT advance input

LL(1) Parsing Algorithm - pseudocode

Set *next* to point to the first symbol of w$;

repeat

let X be the top stack symbol and *a* the symbol pointed to by *next*;

if X is terminal or $ **then**

if X=a **then**

pop X from the stack and advance *next*

else *error()*

else /* X is a non-terminal */

if M[X,a] = $Y_1Y_2\ldots Y_k$ **then begin**

pop X from stack;

push $Y_1, Y_2, \ldots, Y_k$ onto stack, with Y_1 on top

output the production $X \rightarrow Y_1Y_2\ldots Y_k$

end

else *error()*

until X=$ /* stack is empty */

Input pointer

May also execute other code based on the production used

LL(1) Parsing Example

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε
$	$	ACCEPT

Constructing Parsing Tables: The Intuition

- Consider non-terminal A, production $A \to \alpha$, & token t
- $T[A,t] = \alpha$ in two cases:

- If $\alpha \to^* t\ \beta$
  - α can derive a t in the first position
  - We say that $t \in First(\alpha)$

- If $A \to \alpha$ and $\alpha \to^* \varepsilon$ and $S \to^* \beta\ A\ t\ \delta$
  - Useful if stack has A, input is t, and A cannot derive t
  - In this case only option is to get rid of A (by deriving ε)
    - Can work only if t can follow A in at least one derivation
  - We say $t \in Follow(A)$

Computing First Sets

Definition

$$First(X) = \{ t \mid X \rightarrow^* t\alpha \} \cup \{ \varepsilon \mid X \rightarrow^* \varepsilon \}$$

Algorithm sketch:

1. $First(t) = \{ t \}$
2. $\varepsilon \in First(X)$
   - if $X \rightarrow \varepsilon$
   - if $X \rightarrow A_1 \ldots A_n$ and $\varepsilon \in First(A_i)$ for $1 \leq i \leq n$
3. $First(\alpha) \subseteq First(X)$ if $X \rightarrow A_1 \ldots A_n \alpha$
   - and $\varepsilon \in First(A_i)$ for $1 \leq i \leq n$

First Sets. Example

- Recall the grammar

$E \rightarrow T\ X$ $\qquad$ $X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow (\ E\ ) \mid \text{int}\ Y$ $\qquad$ $Y \rightarrow *\ T \mid \varepsilon$

- First sets

First(( ) = { (}

First( )) = {) }

First(int) = { int }

First(+) = { + }

First(*) = { * }

First(T) = {int, (}

First(E) = {int, (}

First(X) = {+, ε }

First(Y) = {*, ε }

Computing Follow Sets

- Definition:

$$\text{Follow}(X) = \{ t \mid S \rightarrow^* \beta\, X\, t\, \delta \}$$

- Intuition
  - If $X \rightarrow A\ B$ then $\text{First}(B) \subseteq \text{Follow}(A)$ and
    $\text{Follow}(X) \subseteq \text{Follow}(B)$
    - if $B \rightarrow^* \varepsilon$ then $\text{Follow}(X) \subseteq \text{Follow}(A)$
  - If S is the start symbol then $\$ \in \text{Follow}(S)$

Computing Follow Sets (Cont.)

Algorithm sketch:

1. $ $\in$ Follow(S)
2. First(β) - {ε} $\subseteq$ Follow(X)
   - For each production $A \to \alpha$ X β
3. Follow(A) $\subseteq$ Follow(X)
   - For each production $A \to \alpha$ X β where $\varepsilon \in$ First(β)

Follow Sets. Example

- Recall the grammar

$E \rightarrow T\ X$ $\qquad$ $X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow (\ E\ ) \mid \text{int}\ Y$ $\qquad$ $Y \rightarrow *\ T \mid \varepsilon$

- Follow sets

Follow(E) = {), $}

Follow(T) = {+,) , $}

Follow(Y) = {+,) , $}

Follow(X) = {), $}

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G

- For each production $A \to \alpha$ in G do:
  - For each terminal $t \in First(\alpha)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in First(\alpha)$, for each $t \in Follow(A)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in First(\alpha)$ and $\$ \in Follow(A)$ do
    - $T[A, \$] = \alpha$

Example 1

$E \rightarrow TX$ $\quad\quad$ $X \rightarrow +E \mid \varepsilon$

$T \rightarrow (E) \mid \text{int } Y$ $\quad\quad$ $Y \rightarrow * T \mid \varepsilon$

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Example 2

S→ Sa | b
First(S)={b}
Follow(S)={$,a}

	a	b	$
S		b, Sa	

Notes on LL(1) Parsing Tables

- If any entry is multiply defined then *G* is not LL(1)
  - If *G* is ambiguous
  - If *G* is left recursive
  - If *G* is not left-factored
  - And in other cases as well

- Most programming language CFGs are not LL(1)

Notes on LL(1) Grammars

Grammar is LL(1) $\Leftrightarrow$ when for all $A \rightarrow \alpha \mid \beta$

1. First(α) $\cap$ First(β) = $\varnothing$; besides, only one of α or β can derive $\in$

2. if α derives $\in$, then Follow(A) $\cap$ First(β) = $\varnothing$

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

Error Handling

Error Handling

- Purpose of the compiler is
  - To detect non-valid programs
  - To translate the valid ones
- Many kinds of possible errors (e.g. in C)

Error kind	Example	Detected by …
Lexical	… $ …	Lexer
Syntax	… x *% …	Parser
Semantic	… int x; y = x(3); …	Type checker
Correctness	your favorite program	Tester/User

Syntax Error Handling

- Error handler should
  - Report errors accurately and clearly
  - Recover from an error quickly
  - Not slow down compilation of valid code

- Good error handling is not easy to achieve

Approaches to Syntax Error Recovery

- From simple to complex
  - Panic mode
  - Error productions
  - Automatic local or global correction

Error Recovery: Panic Mode

- Simplest, most popular method
- When an error is detected:
  - Discard tokens until one with a clear role is found
  - Continue from there
- Such tokens are called synchronizing tokens
  - Typically the statement or expression terminators

Syntax Error Recovery: Panic Mode (Cont.)

- Consider the erroneous expression

  (1 + + 2) + 3

- Panic-mode recovery:
  - Skip ahead to next integer and then continue

Syntax Error Recovery: Error Productions

- Idea: specify in the grammar known common mistakes
- Essentially promotes common errors to alternative syntax
- Example:
  - Write 5 x instead of 5 * x
  - Add the production $E \rightarrow \ldots \mid E\ E$
- Disadvantage
  - Complicates the grammar

Error Recovery: Local and Global Correction

- Idea: find a correct "nearby" program
  - Try token insertions and deletions
  - Exhaustive search

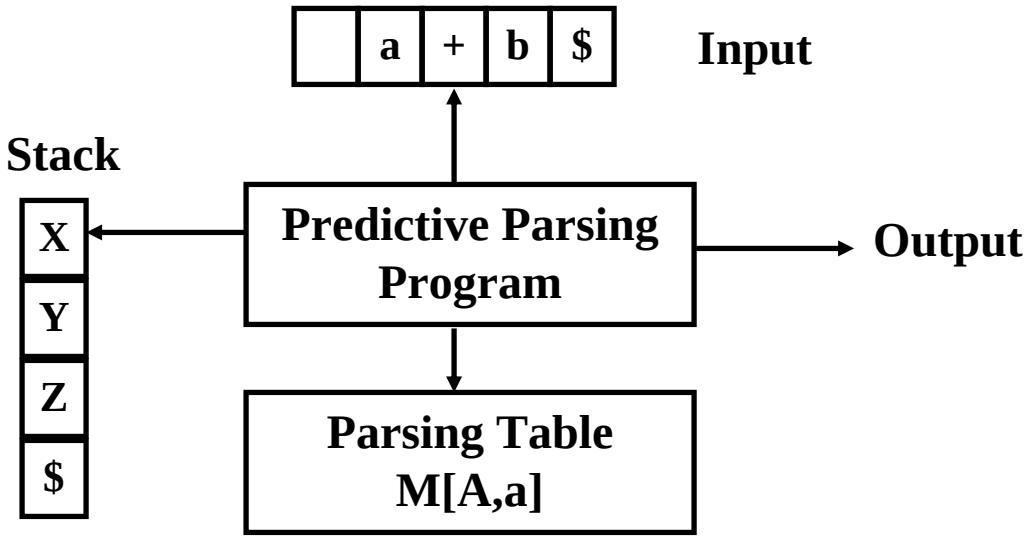
- Disadvantages:
  - Hard to implement
  - Slows down parsing of correct programs
  - "Nearby" is not necessarily "the intended" program

Syntax Error Recovery: Past and Present

- Past
  - Slow recompilation cycle (even once a day)
  - Find as many errors in one cycle as possible

- Present
  - Quick recompilation cycle
  - Users tend to correct one error/cycle
  - Complex error recovery is less compelling
  - Panic-mode seems enough

Implementing Panic Mode in LL(1)



Error situations include:

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

Panic-Mode Recovery

- Assume in a syntax error, non-terminal *A* is on the top of the stack.
- The choice for a synchronizing set is important.
  - 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...
  - add symbols of FIRST(*A*) to the synchronizing set. In this case, we skip input and once we find a token in FIRST(*A*), we resume parsing from *A*.

Panic-Mode Recovery (Cont.)

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-token,

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 *A* (This is essentially inserting *A* before a).

Parse Table / Example

	id	+	*	(	)	$
E	T E'			T E'	synch	synch
E'		+ T E'			∈	∈
T	F T'	synch		F T'	synch	synch
T'		∈	* F T'		∈	∈
F	id	synch	synch	(E)	synch	synch

Pop top of stack NT for "synch" cells

Skip current-token for empty cells

Parsing Example

	id	+	*	(	)	$
E	T E'			T E'	synch	synch
E'		+ T E'			∈	∈
T	F T'	synch		F T'	synch	synch
T'		∈	* F T'		∈	∈
F	id	synch	synch	(E)	synch	synch

STACK	INPUT	Remark
E $	+ id * + id $	error, skip +
E $	id * + id $	
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 $	

Possible Error Msg:
"Misplaced +
I am skipping it"

Parsing Example (Cont.)

	id	+	*	(	)	$
E	T E'			T E'	synch	synch
E'		+ T E'			∈	∈
T	F T'	synch		F T'	synch	synch
T'		∈	* F T'		∈	∈
F	id	synch	synch	(E)	synch	synch

STACK	INPUT	Remark
F T' E' $	+ id $	error, M[F,+] = synch, F is popped
T' E' $	+ id $	
E' $	+ id $	
+ T E' $	+ id $	
T E' $	id $	
F T' E' $	id $	
id T' E' $	id $	
T' E' $	$	
E' $	$	
$	$	

Possible Error Msg:
"Missing Term"

Introduction to Bottom-Up Parsing

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
- Bottom-up is the preferred method
- Concepts today, algorithms next time

An Introductory Example

- Bottom-up parsers don't need left-factored grammars

- Revert to the "natural" grammar for our example:

  $E \rightarrow T + E \mid T$

  $T \rightarrow \text{int} * T \mid \text{int} \mid (E)$

- Consider the string: int * int + int

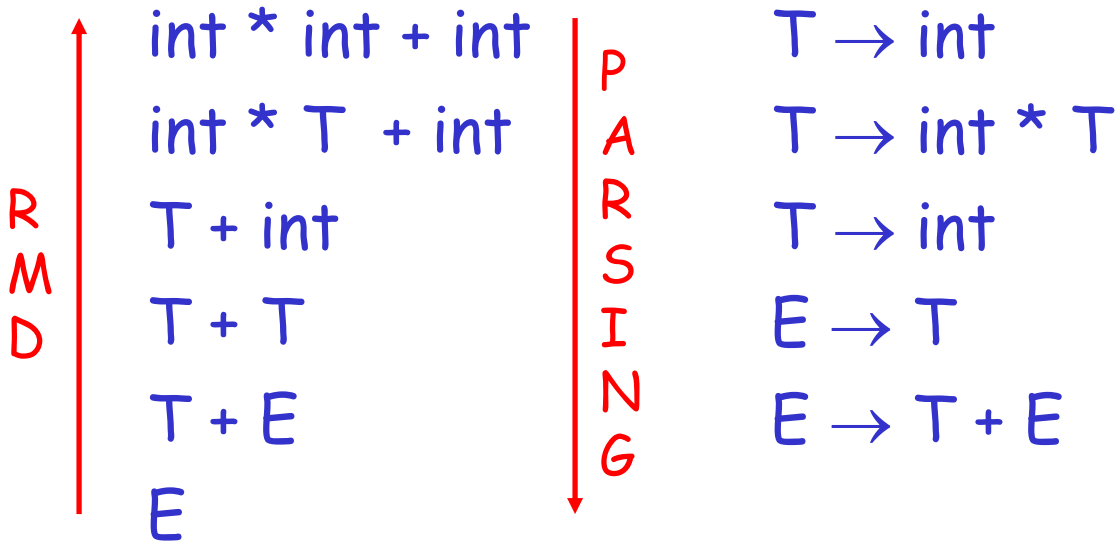
The Idea

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

int * int + int	$T \rightarrow$ int
int * T + int	$T \rightarrow$ int * T
T + int	$T \rightarrow$ int
T + T	$E \rightarrow T$
T + E	$E \rightarrow T + E$
E	

Observation

- Read the productions in reverse (from bottom to top)
- This is a rightmost derivation!



Important Fact #1

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse

A Bottom-up Parse

int * int + int

int * T + int

T + int

T + T

T + E

E

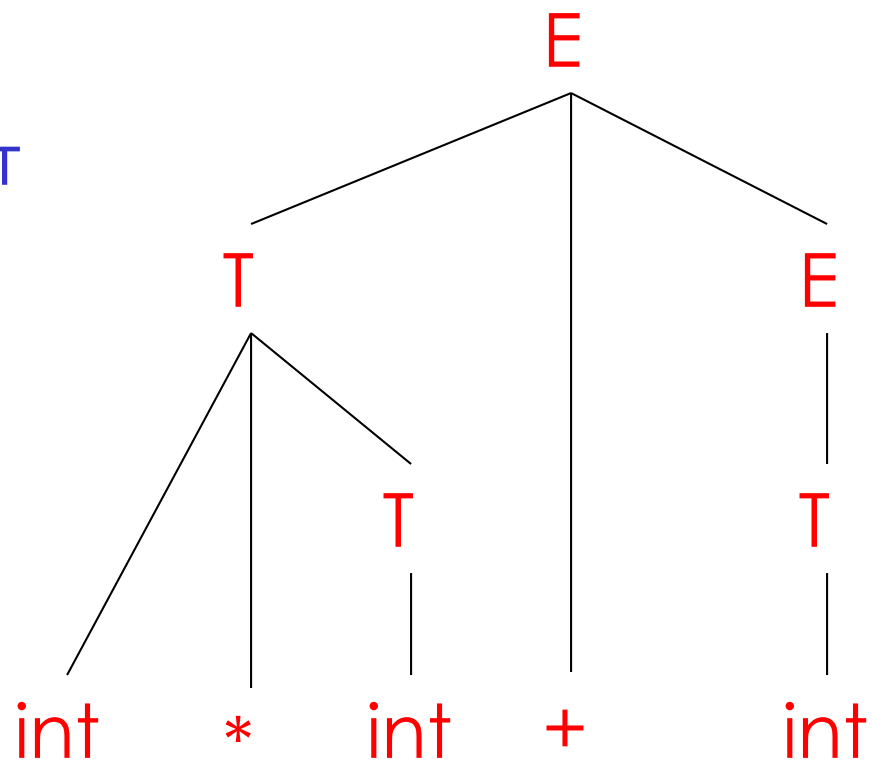
$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow int * T$

$T \rightarrow int$

$T \rightarrow (E)$



A Bottom-up Parse in Detail (1)

int * int + int

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow \text{int} * T$

$T \rightarrow \text{int}$

$T \rightarrow (E)$

int * int + int

A Bottom-up Parse in Detail (2)

int * int + int

int * T + int

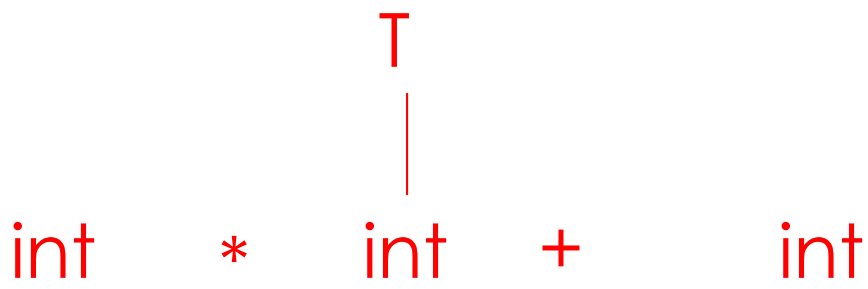
$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow int * T$

$T \rightarrow int$

$T \rightarrow (E)$



A Bottom-up Parse in Detail (3)

int * int + int

int * T + int

T + int

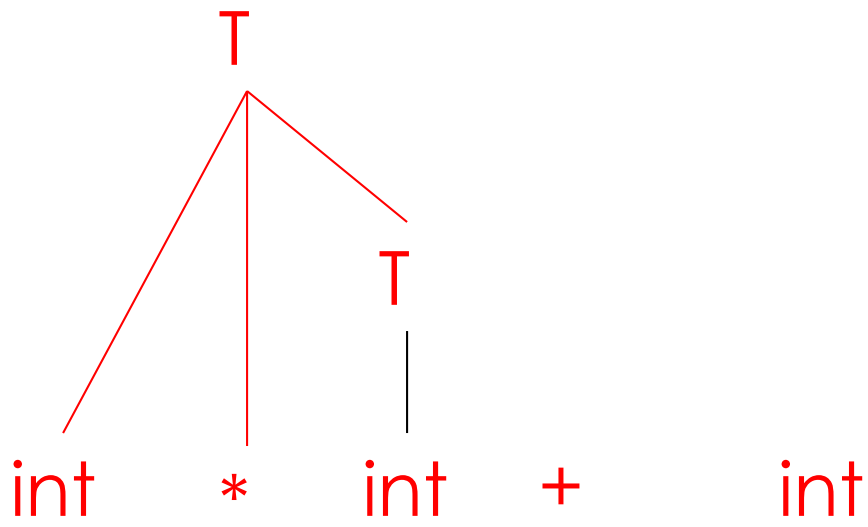
$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow int * T$

$T \rightarrow int$

$T \rightarrow (E)$



A Bottom-up Parse in Detail (4)

int * int + int

int * T + int

T + int

T + T

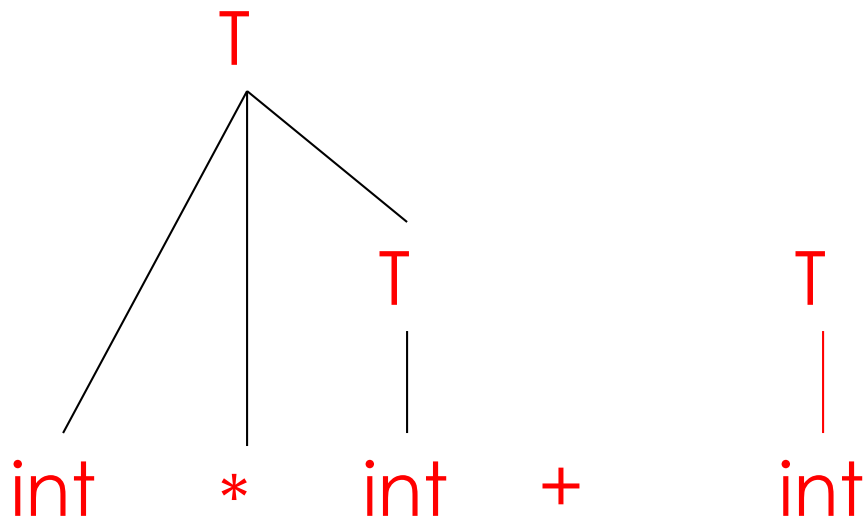
$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow int * T$

$T \rightarrow int$

$T \rightarrow (E)$



A Bottom-up Parse in Detail (5)

int * int + int

int * T + int

T + int

T + T

T + E

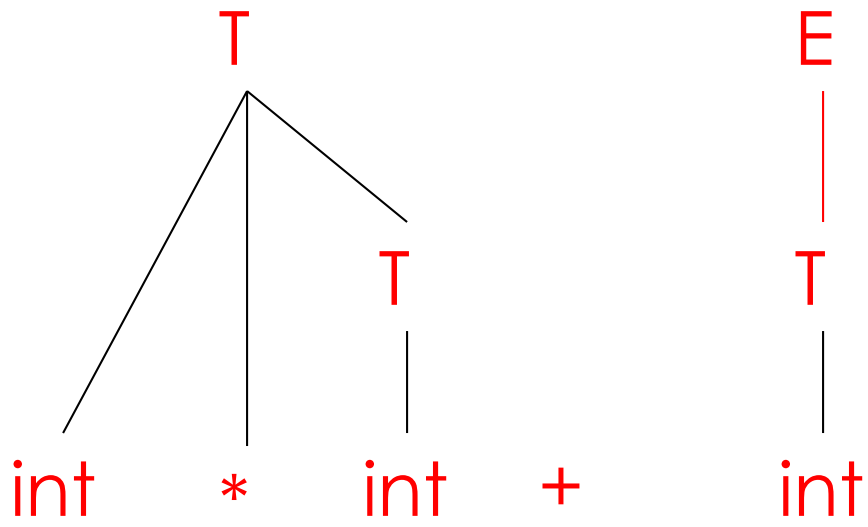
$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow int * T$

$T \rightarrow int$

$T \rightarrow (E)$



A Bottom-up Parse in Detail (6)

int * int + int

int * T + int

T + int

T + T

T + E

E

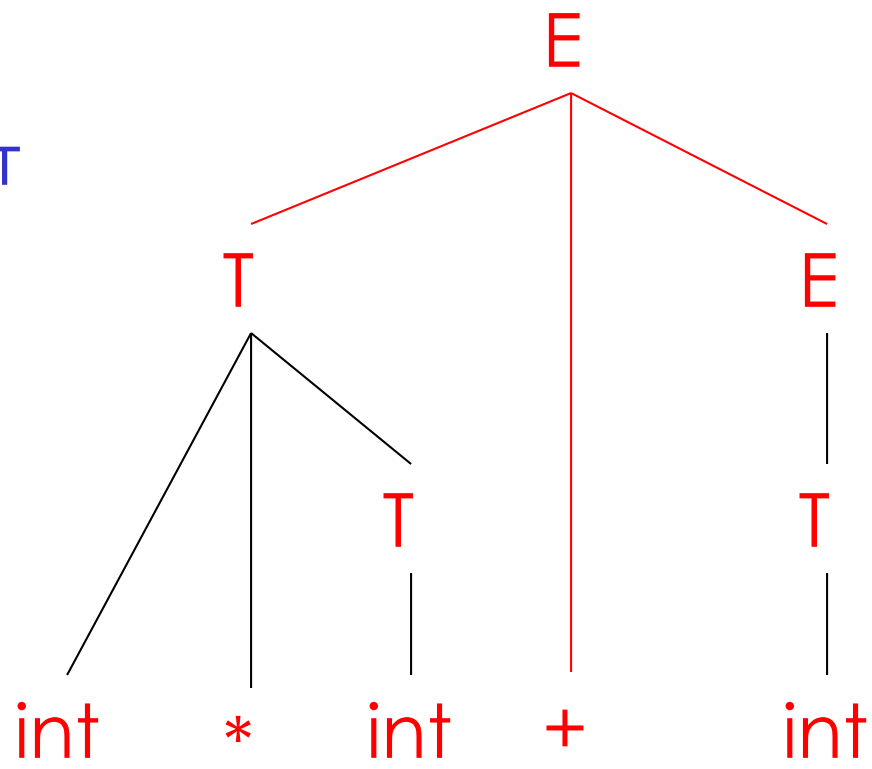
$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow int * T$

$T \rightarrow int$

$T \rightarrow (E)$



A Trivial Bottom-Up Parsing Algorithm

Let I = input string

repeat

pick a non-empty substring β of I

where $X \rightarrow \beta$ is a production

if no such β, backtrack

replace one β by X in I

until I = "S" (the start symbol) or all possibilities are exhausted

Where Do Reductions Happen?

Important Fact #1 has an interesting consequence:

- Let $\alpha\beta\omega$ be a step of a bottom-up parse
- Assume the next reduction is by $X \rightarrow \beta$
- Then ω is a string of terminals

Why? Because $\alpha X \omega \rightarrow \alpha\beta\omega$ is a step in a right-most derivation

Notation

- Idea: 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 |
  - The | is not part of the string
- Initially, all input is unexamined $|x_1 x_2 \ldots x_n$

Shift-Reduce Parsing

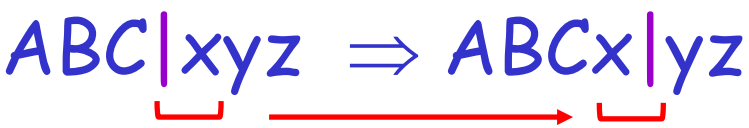
Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

Shift

- *Shift*: Move | one place to the right
  - Shifts a terminal to the left string



Reduce

- Apply an inverse production at the right end of the left string
  - If $A \rightarrow xy$ is a production, then

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

The Example with Reductions Only

int * int | + int	reduce T → int
int * T | + int	reduce T → int * T
T + int |	reduce T → int
T + T |	reduce E → T
T + E |	reduce E → T + E
E |	

The Example with Shift-Reduce Parsing

|int * int + int	shift
int | * int + int	shift
int * | int + int	shift
int * int | + int	reduce T → int
int * T | + int	reduce T → int * T
T | + int	shift
T + | int	shift
T + int |	reduce T → int
T + T |	reduce E → T
T + E |	reduce E → T + E
E |	

A Shift-Reduce Parse in Detail (1)

|int * int + int

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow \text{int} * T$

$T \rightarrow \text{int}$

$T \rightarrow (E)$

int * int + int

↑

A Shift-Reduce Parse in Detail (2)

|int * int + int

int | * int + int

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow \text{int} * T$

$T \rightarrow \text{int}$

$T \rightarrow (E)$

int * int + int

↑

A Shift-Reduce Parse in Detail (3)

|int * int + int

int | * int + int

int * | int + int

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow \text{int} * T$

$T \rightarrow \text{int}$

$T \rightarrow (E)$

int * int + int

↑

A Shift-Reduce Parse in Detail (4)

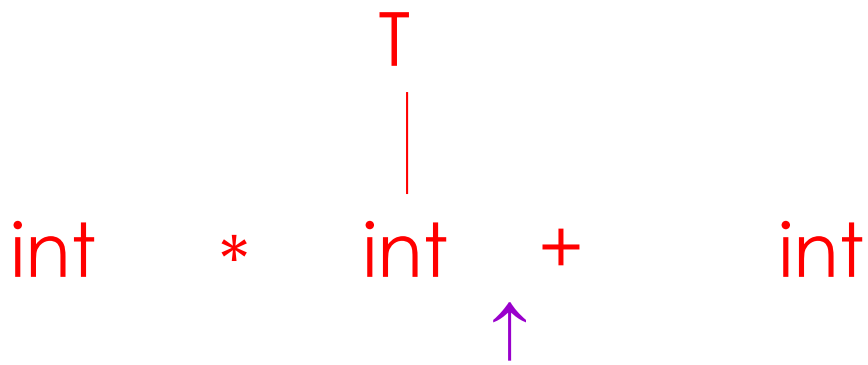
|int * int + int
int | * int + int
int * | int + int
int * int | + int

$E \rightarrow T + E$
$E \rightarrow T$
$T \rightarrow \text{int} * T$
$T \rightarrow \text{int}$
$T \rightarrow (E)$

int * int + int

A Shift-Reduce Parse in Detail (5)

|int * int + int	
int | * int + int	$E \rightarrow T + E$
int * | int + int	$E \rightarrow T$
int * int | + int	$T \rightarrow \text{int} * T$
int * T | + int	$T \rightarrow \text{int}$
	$T \rightarrow (E)$



A Shift-Reduce Parse in Detail (6)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int

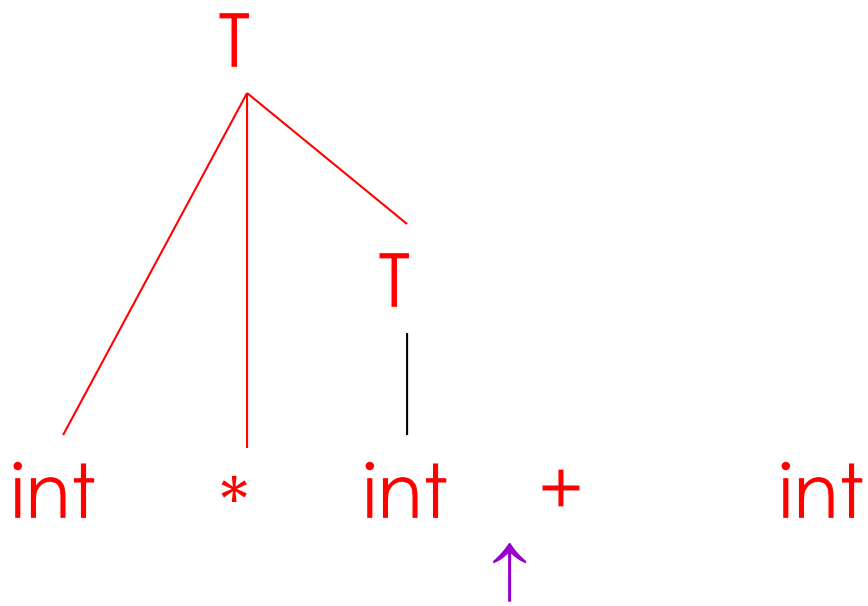
$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow int * T$

$T \rightarrow int$

$T \rightarrow (E)$



A Shift-Reduce Parse in Detail (7)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int

T + | int

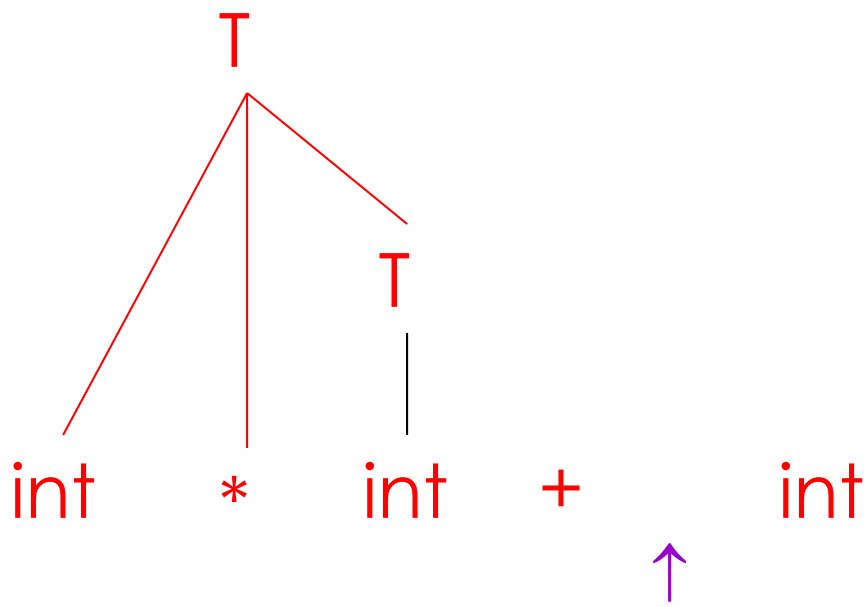
$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow int * T$

$T \rightarrow int$

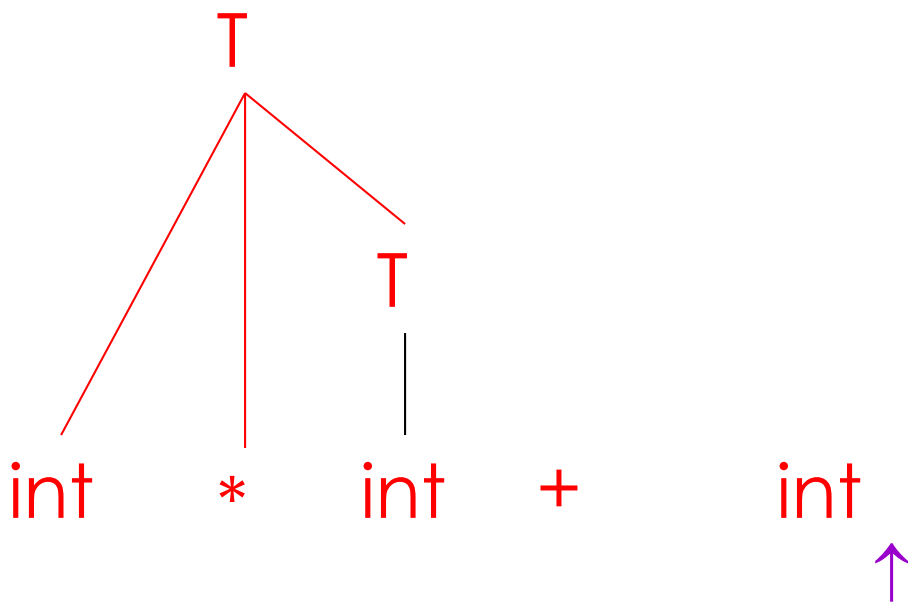
$T \rightarrow (E)$



A Shift-Reduce Parse in Detail (8)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |

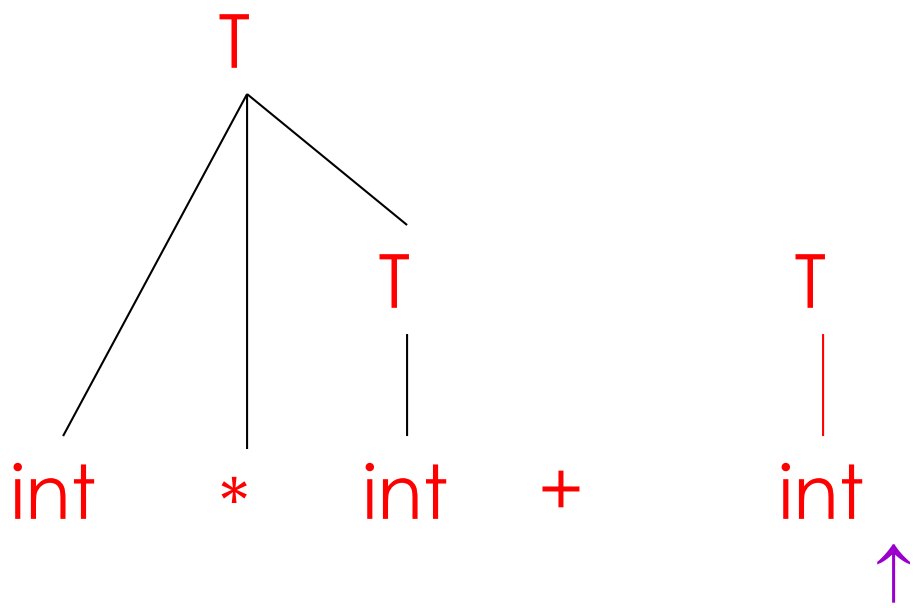
$E \rightarrow T + E$
$E \rightarrow T$
$T \rightarrow \text{int} * T$
$T \rightarrow \text{int}$
$T \rightarrow (E)$



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 |

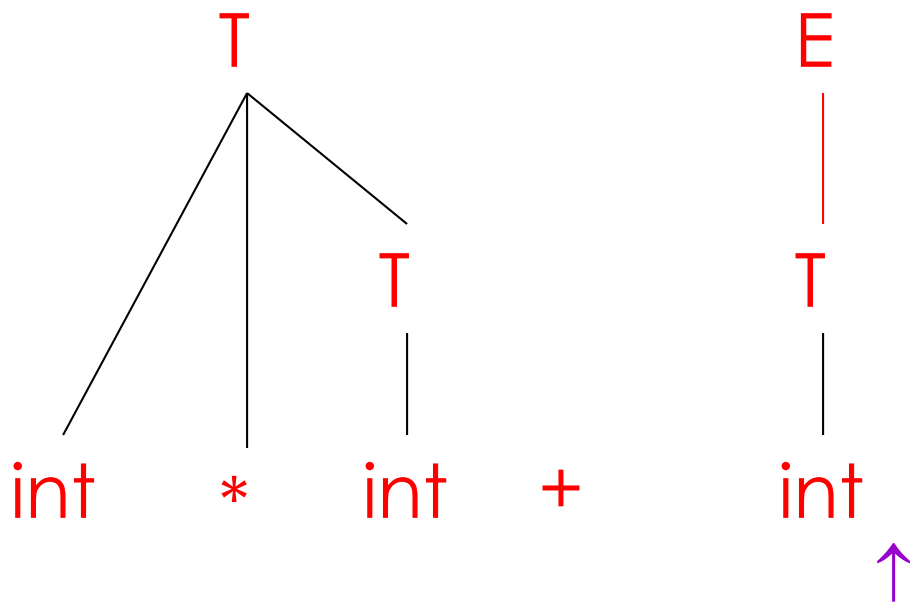
$E \rightarrow T + E$
$E \rightarrow T$
$T \rightarrow int * T$
$T \rightarrow int$
$T \rightarrow (E)$



A Shift-Reduce Parse in Detail (10)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
T + E |

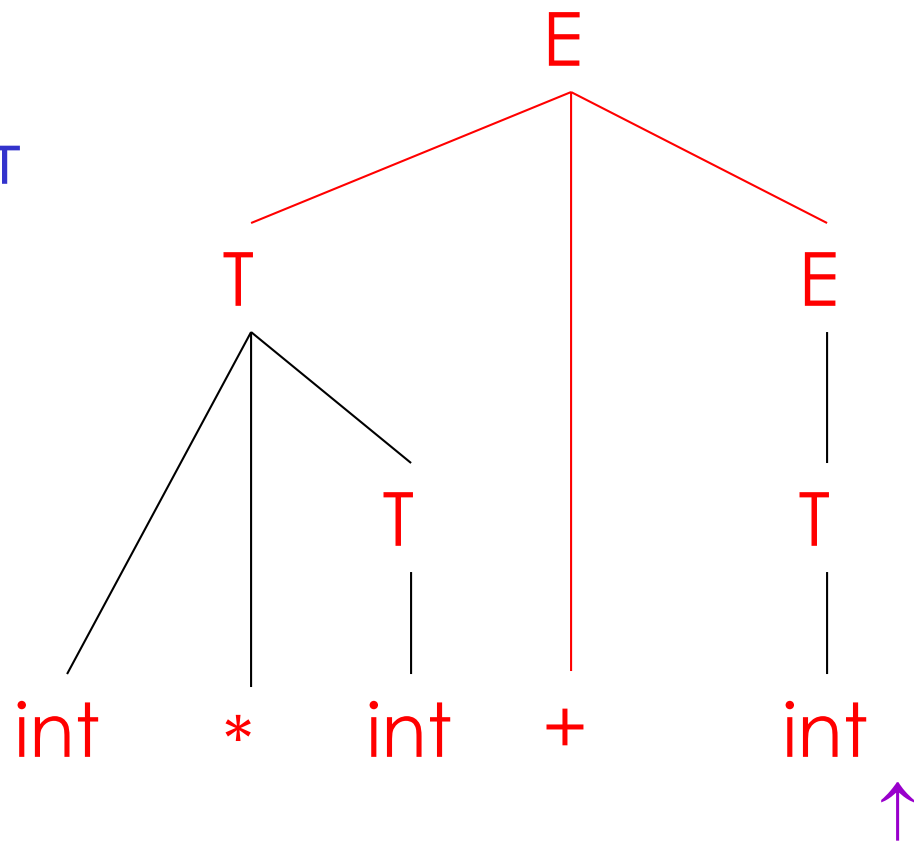
$E \rightarrow T + E$
$E \rightarrow T$
$T \rightarrow int * T$
$T \rightarrow int$
$T \rightarrow (E)$



A Shift-Reduce Parse in Detail (11)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
T + E |
E |

$E \rightarrow T + E$
$E \rightarrow T$
$T \rightarrow int * T$
$T \rightarrow int$
$T \rightarrow (E)$



The Stack

- Left string can be implemented by a stack
  - Top of the stack is the |
- Shift pushes a terminal on the stack
- Reduce pops 0 or more symbols off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

Conflicts

- In a given state, more than one action (shift or reduce) may lead to a valid parse
- If it is legal to shift or reduce, there is a *shift-reduce* conflict
- If it is legal to reduce by two different productions, there is a *reduce-reduce* conflict
- You will see such conflicts in your project!
  - More next time . . .

Question?

Choose the alternative that correctly left factors "if" statements in the given grammar

```
EXPR → if BOOL then { EXPR}
     | if BOOL then { EXPR} else { EXPR}
     | …
BOOL → true | false
```

○
```
EXPR → if true then {EXPR}
     | if false then { EXPR}
     | if true then { EXPR} else { EXPR}
     | if false then { EXPR} else { EXPR}
     | …
```

○
```
EXPR → EXPR' | EXPR' else { EXPR}
EXPR' → if BOOL then { EXPR}
      | …
BOOL → true | false
```

○
```
EXPR → if BOOL EXPR'
     | …
EXPR' → then { EXPR}
      | then { EXPR} else { EXPR}
BOOL → true | false
```

○
```
EXPR → if BOOL then { EXPR} EXPR'
     | …
EXPR' → else { EXPR} | ε
BOOL → true | false
```

Question?

Choose the next parse state given the grammar, parse table, and current state below. The initial string is:

if true then { true } else { if false then { false } } $

	if	then	else	{	}	true	false	$
E	if Bthen { E}E'				ε	B	B	ε
E'			else { E}		ε			ε
B						true	false	

	Stack	Input
Current	E' $	else { if false then { false } } $
○	$	$
○	else {E} $	else { if false then { false } } $
○	E} $	if false then { false } } $
○	else {if Bthen {E} E' } $	else { if false then { false } } $

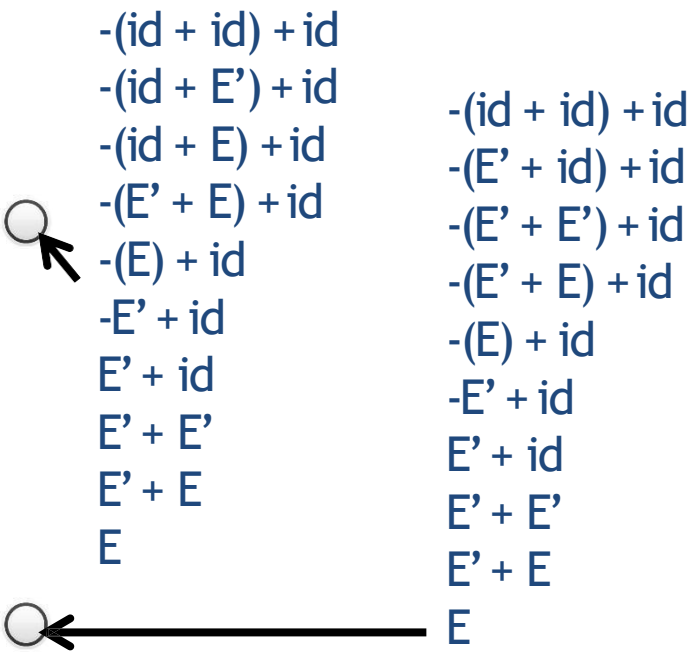
E→ if B then { E} E'
| B| ε
E' → else { E} | ε
B→ true | false

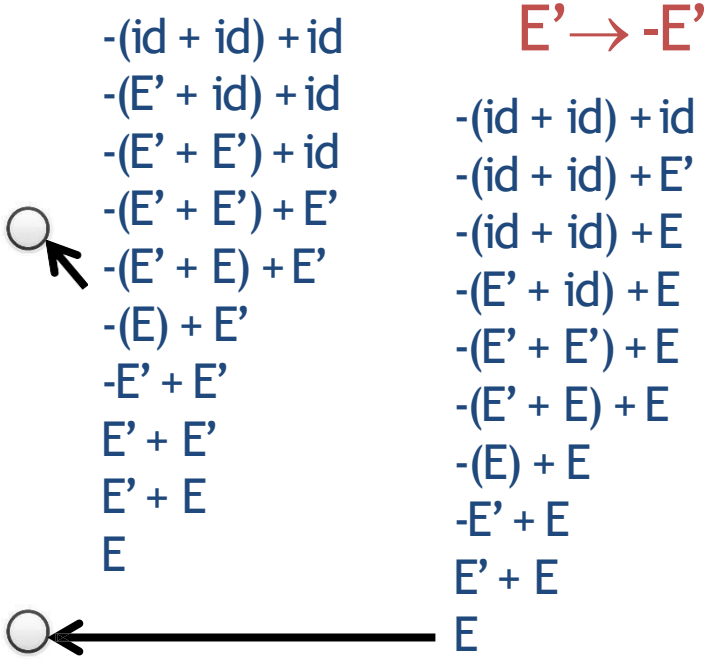
Question?

For the given grammar, what is the correct series of reductions for the string: -(id + id) + id

$E \rightarrow E' \mid E' + E$

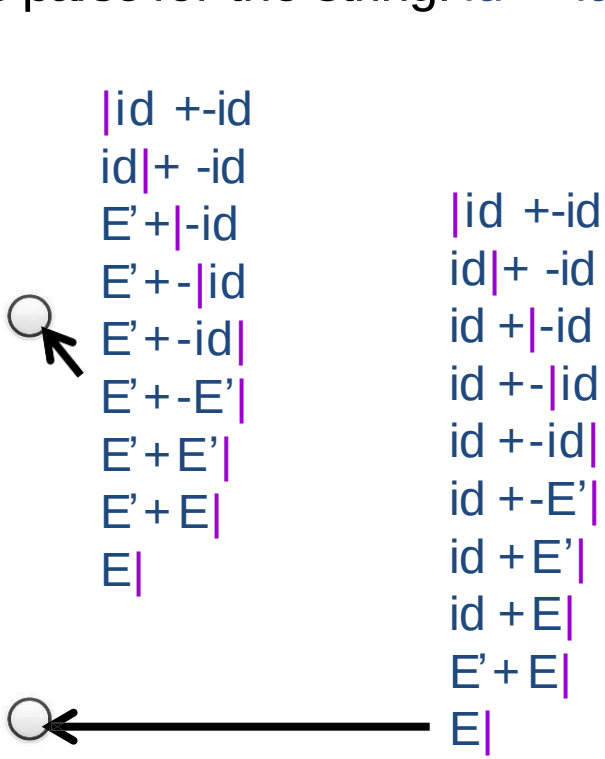
$E' \rightarrow -E' \mid id \mid (E)$

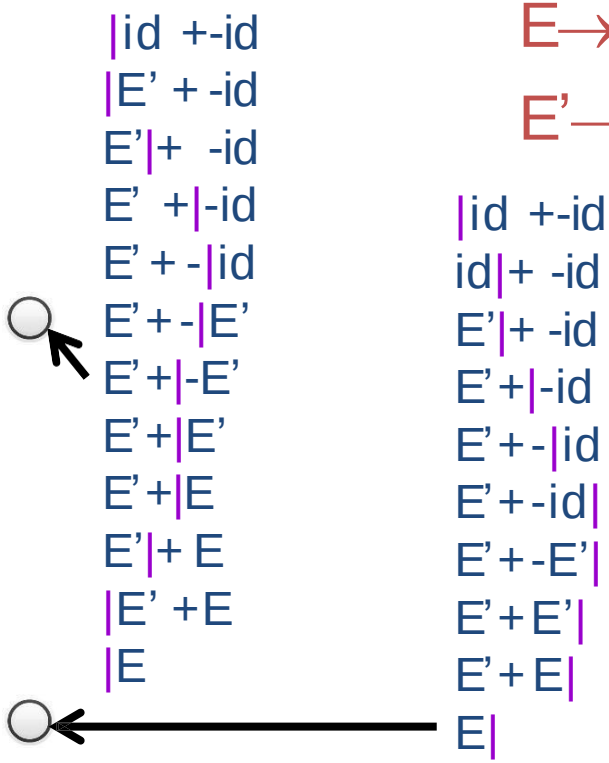




Question?

For the given grammar, what is the correct shift-reduce parse for the string: id + -id





$E \rightarrow E' \mid E' + E$

$E' \rightarrow -E' \mid id \mid (E)$

Question?

For the given grammar, find the First and Follow of Non-terminals and the Parse table

$S \rightarrow i\ E\ t\ S\ S' \mid a$ $S' \rightarrow e\ S \mid \in$ $E \rightarrow b$	First(S) = First(S') = First(E) =	Follow(S) = Follow(S') = Follow(E) =

	a	b	e	i	t	$
S						
S'						
E						

Question?

For the given grammar, find the First and Follow of Non-terminals and the Parse table

$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \in$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \in$
$F \rightarrow (\ E\ ) \mid id$

First(E,T,F) =

First(E') =

First(T') =

Follow(E) =

Follow(E') =

Follow(T) =

Follow(T') =

Follow(F) =

	id	+	*	(	)	$
E						
E'						
T						
T'						
F						

Question?

Which of the following statements are true about the given grammar?

$S \rightarrow a\ T\ U\ b \mid \varepsilon$
$T \rightarrow c\ U\ c \mid b\ U\ b \mid a\ U\ a$
$U \rightarrow S\ b \mid c\ c$

Choose all that are correct.

- ○ The follow set of S is { $, b }
- ○ The first set of U is { a, b, c }
- ○ The first set of S is {ε, a, b }
- ○ The follow set of T is { a, b, c }

Question?

Consider the following grammar:

$S \rightarrow A\ (\ S\ )\ B \mid \varepsilon$
$A \rightarrow S \mid S\ B\ x \mid \varepsilon$
$B \rightarrow S\ B \mid y$

What are the first and follow sets of S

- First: { x, y, (, ε } Follow: { y, x, (,) }
- First: { x, ε } Follow: { $, y, x, (,) }
- First: { y, (, ε } Follow: { $, y, (,)}
- First: { x, y, (, ε } Follow: { $, y, x, (,) }
- First: { x, y, (} Follow: { $, y, x, (,) }
- First: { x, (} Follow: { $, y, x }