Review of CFGs and Parsing – II
Bottom-up Parsers

Lecture 5

Outline

- Parser Overview
- Top-down Parsers (Covered largely through labs)
- Bottom-up Parsers

The Functionality of the Parser

- **Input:** sequence of tokens from lexer

- **Output:**

  - parse tree/AST of the program
  - Actions, e.g., semantic analysis, type checking
  - Intermediate representation

Comparison with Lexical Analysis

Phase	*Input*	*Output*
Lexer	String of characters	String of tokens
Parser	String of tokens	Parse tree/ AST

Bottom-up vs. Top-down

	Top-down Parsers	Bottom-up Parsers
Successful Parse	From start symbol of grammar to the string	From the string to the start symbol of the grammar
Example of grammars	LL(k) Left-to-right, Leftmost derivation first	LR(k), LALR Left-to-right, Rightmost derivation first (in reverse)
Example of parser technique	Recursive-descent	Shift-reduce
Ease of implementation	Literally recursive descent	Many grammar generators available (Yacc, Bison,…)
Issue with left-recursion	Yes	No
Issues with left-factoring	Yes	No

Review: Bottom-Up Parsing

- Bottom-up parsing is more general than "traditional" top-down parsing
  - And just as efficient
  - Doesn't have issues with left-recursion
  - Doesn't require left-factoring
  - Many well-known parser generators (Yacc, Bison,…)
  - Can handle many more grammars without backtracking than otherwise
- PEG parsers are top-down, more general than "traditional" top-down and bottom-up parsers

Review: An Introductory Example

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

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

  $E \rightarrow T + E \mid T$

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

- Consider the string: int * int + int

Review: The Idea

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	

Observation

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

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	

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

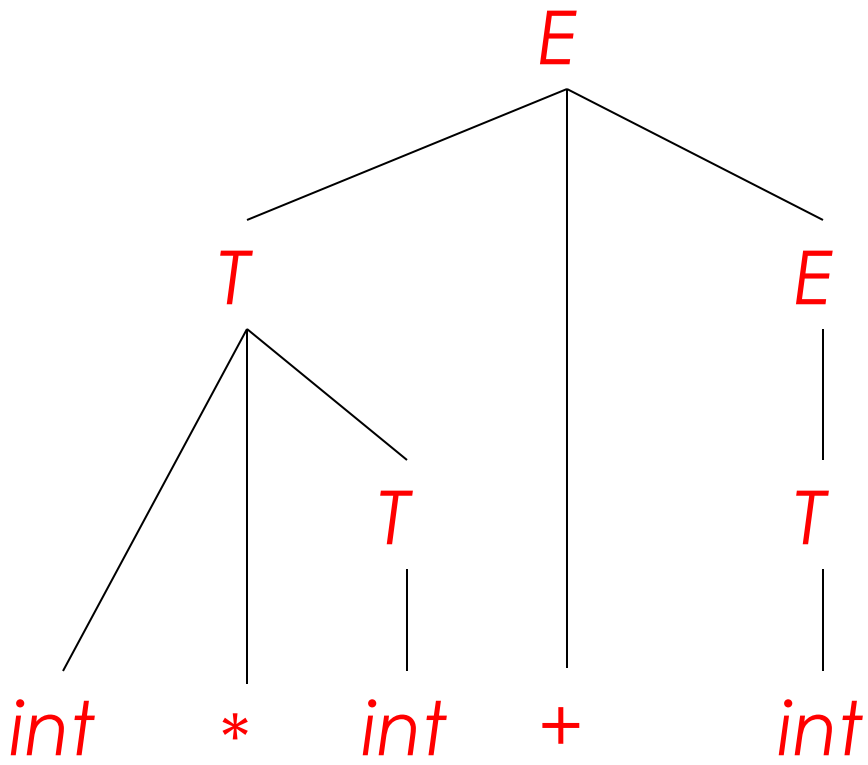
int * T + int

T + int

T + T

T + E

E



A Bottom-up Parse in Detail (1)

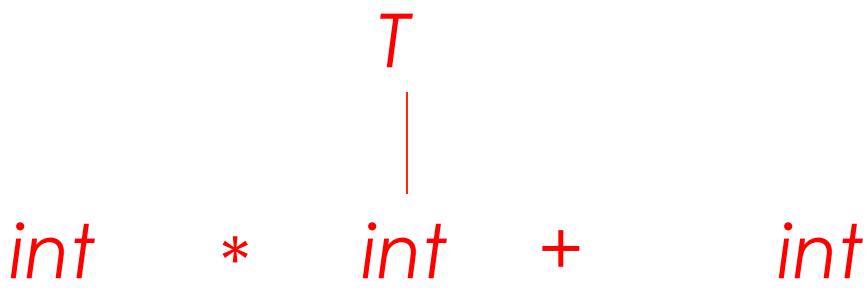
int * int + int

int * *int* + *int*

A Bottom-up Parse in Detail (2)

int * int + int

int * T + int

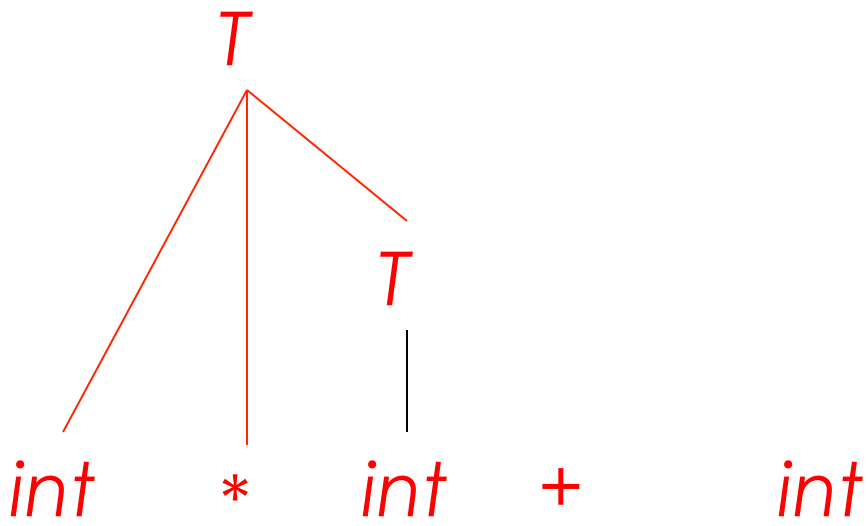


A Bottom-up Parse in Detail (3)

int * int + int

int * T + int

T + int



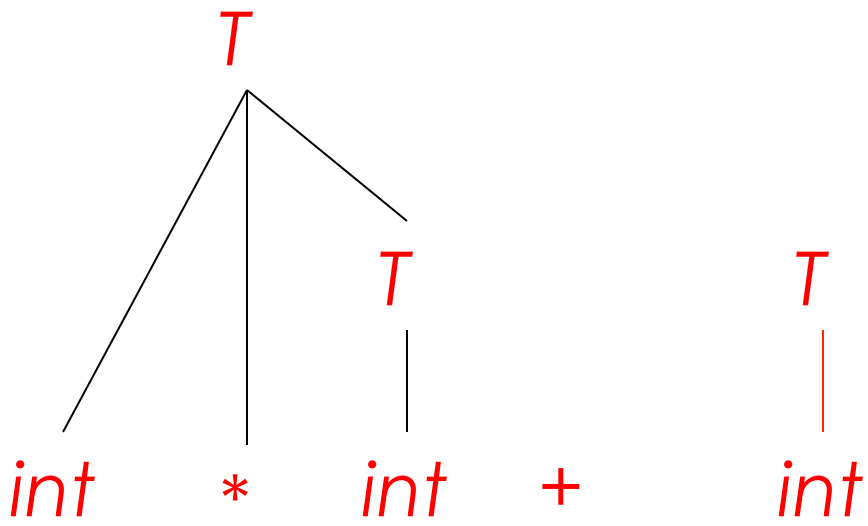
A Bottom-up Parse in Detail (4)

int * int + int

int * T + int

T + int

T + T



A Bottom-up Parse in Detail (5)

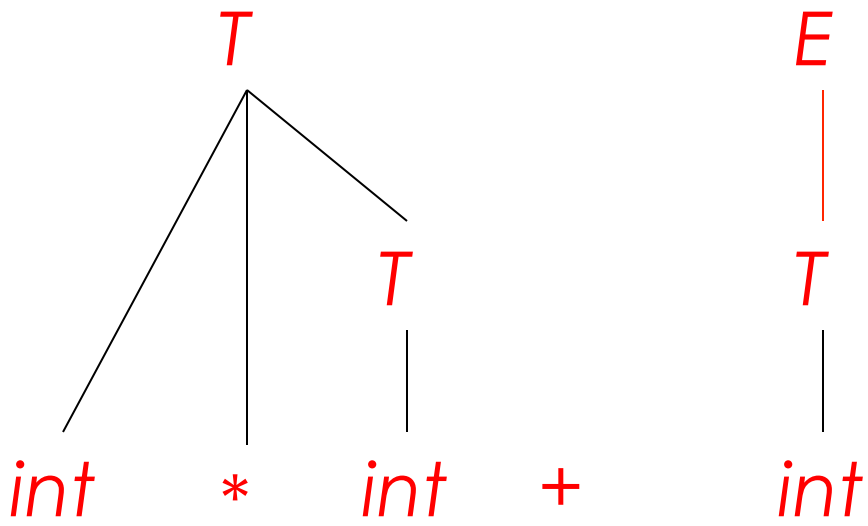
int * int + int

int * T + int

T + int

T + T

T + E



A Bottom-up Parse in Detail (6)

int * int + int

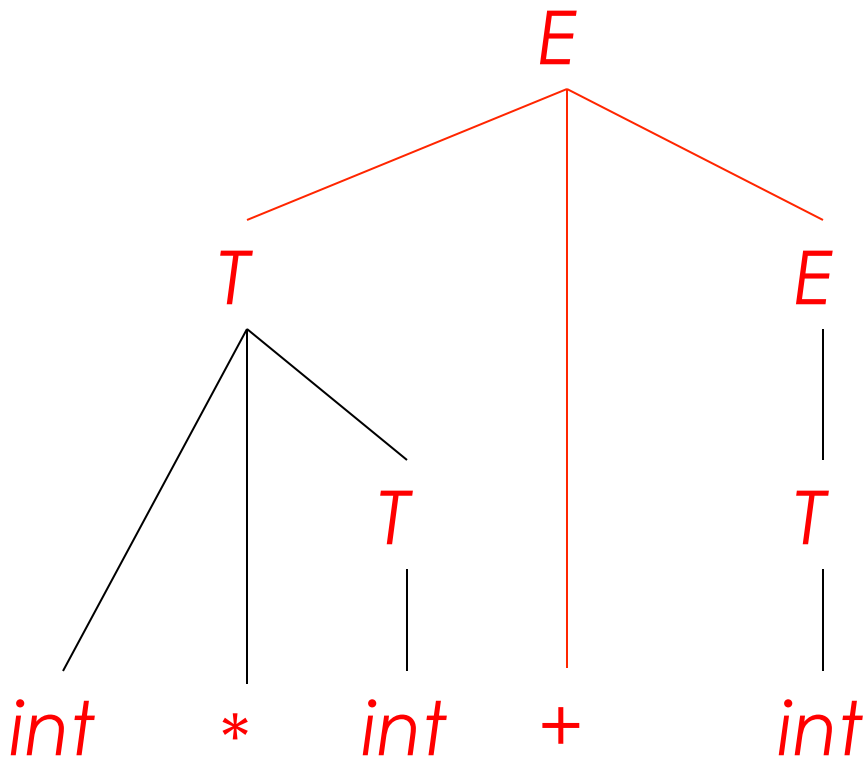
int * T + int

T + int

T + T

T + E

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

Questions

- Does this algorithm terminate?
- How fast is the algorithm?
- Does the algorithm handle all cases?
- How do we choose the substring to reduce at each step?

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

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

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

A Shift-Reduce Parse in Detail (2)

|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)

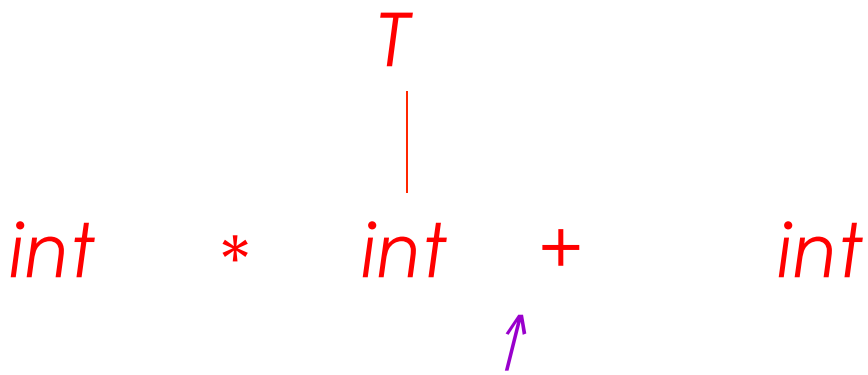
|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int



A Shift-Reduce Parse in Detail (6)

|int * int + int

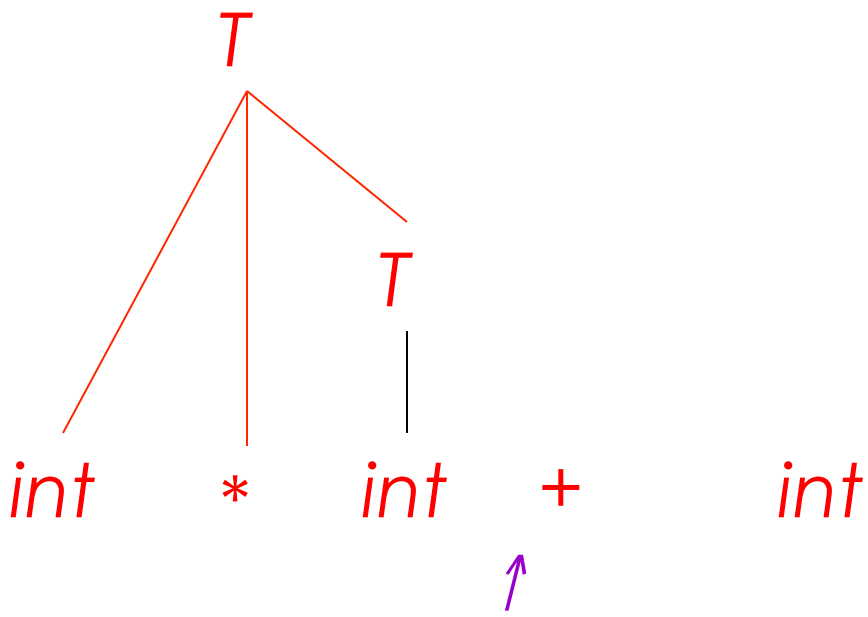
int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int



A Shift-Reduce Parse in Detail (7)

|int * int + int

int | * int + int

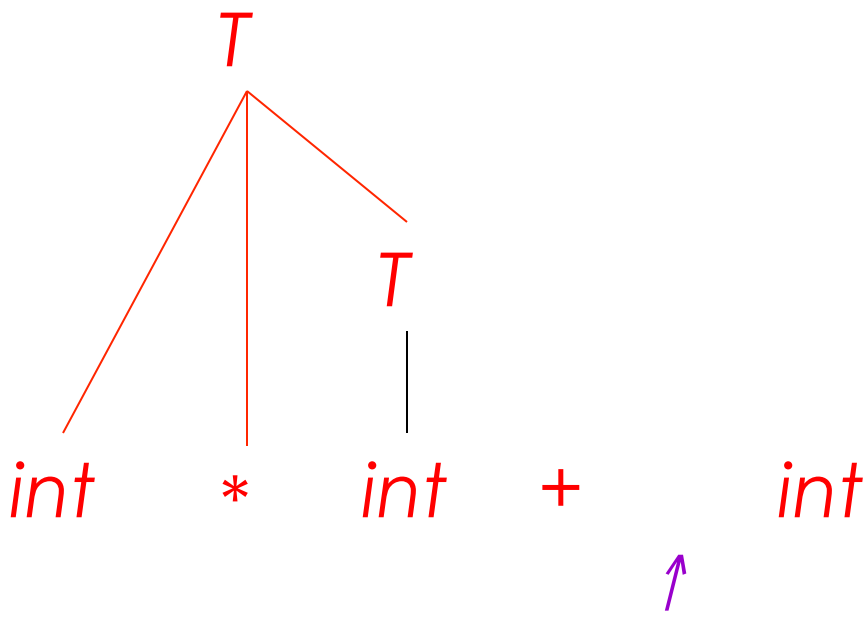
int * | int + int

int * int | + int

int * T | + int

T | + int

T + | int



A Shift-Reduce Parse in Detail (8)

|int * int + int

int | * int + int

int * | int + int

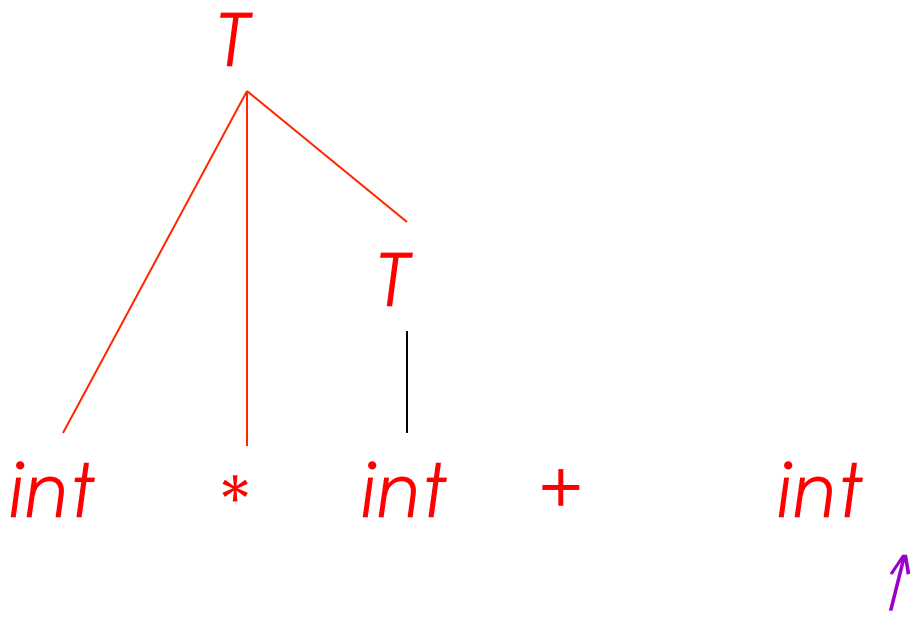
int * int | + int

int * T | + int

T | + int

T + | int

T + int |



A Shift-Reduce Parse in Detail (9)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

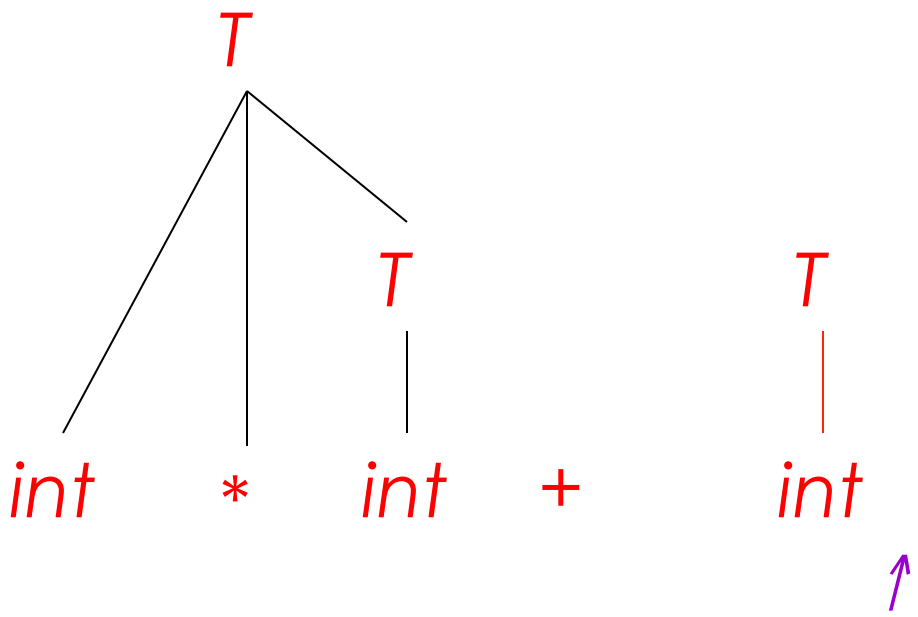
int * T | + int

T | + int

T + | int

T + int |

T + T |



A Shift-Reduce Parse in Detail (10)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

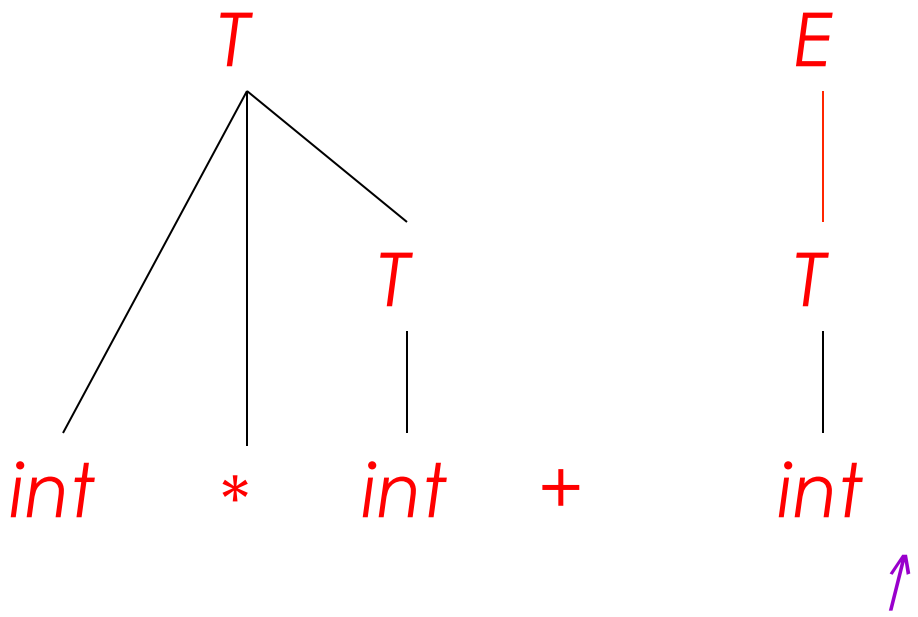
T | + int

T + | int

T + int |

T + T |

T + E |



A Shift-Reduce Parse in Detail (11)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int

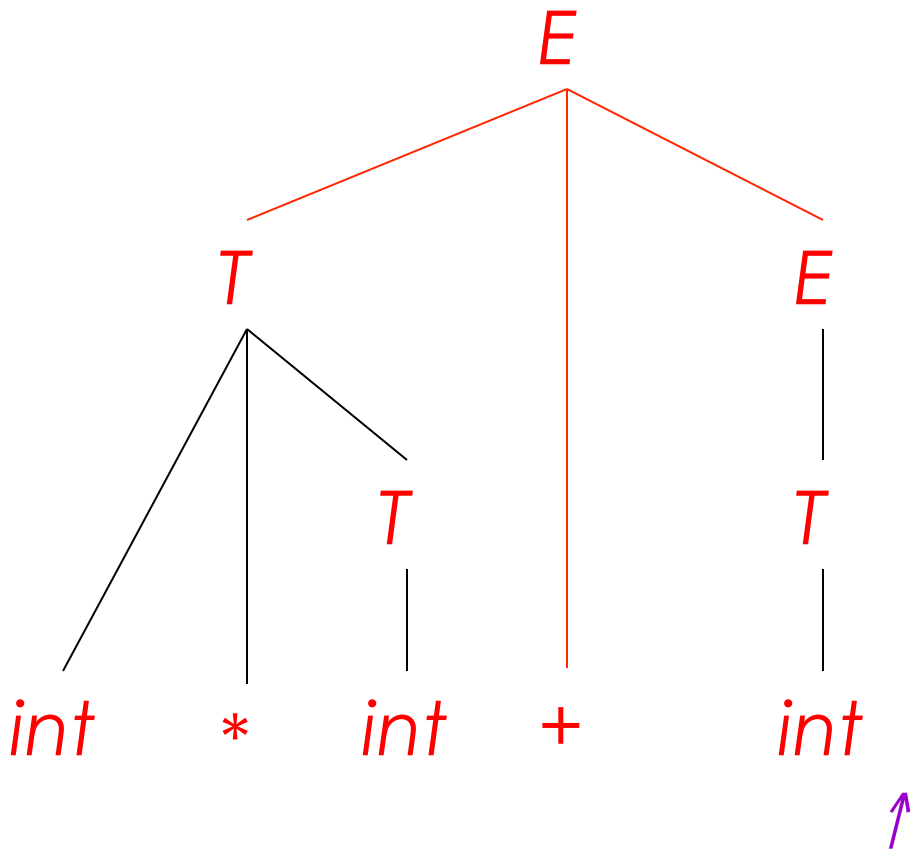
T + | int

T + int |

T + T |

T + E |

E |



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

Key Issue: To Shift or Reduce?

- How do we decide when to shift or reduce?
- Example grammar:

  E → T + E | T

  T → int * T | int | (E)

- Consider step int | * int + int
  - We could reduce by T → int giving T | * int + int
  - A fatal mistake!
    - No way to reduce to the start symbol E

Handles: Symbols replaced by Reduction

- **Intuition:** Want to reduce only if the result can still be reduced to the start symbol

- Handle: Informally, represents the RHS of a production. Let $X \rightarrow \beta$ be a production in *G*. Then β in the position after α is a *handle* of $\alpha\beta\omega$

Handles (Cont.)

- Handles formalize the intuition
  - A handle is a string that can be reduced and also allows further reductions back to the start symbol (using a particular production at a specific spot)
- We only want to reduce at handles
- Note: We have said what a handle is, not how to find handles

Important Fact #2

Important Fact #2 about bottom-up parsing:

In shift-reduce parsing, handles appear only at the top of the stack, never inside

Using the symbol already shifted into the stack (left context) and the next k lookahead symbols (right context), decide to shift or reduce