Compiler

Parsing: Bottom-Up

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 in practice
- We'll explain the intuition, you should read the book to understand the algorithm

An Introductory Example

- Bottom-up parsers don't need left-factored grammars
- Hence we can 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	

For You To Do

- **Question:** find the rightmost derivation of the string int * int + int

Observation

- Read the productions found by bottom-up parse in reverse (i.e., from bottom to top)
- This is a rightmost derivation!

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	

A Bottom-up Parse

int * int + int

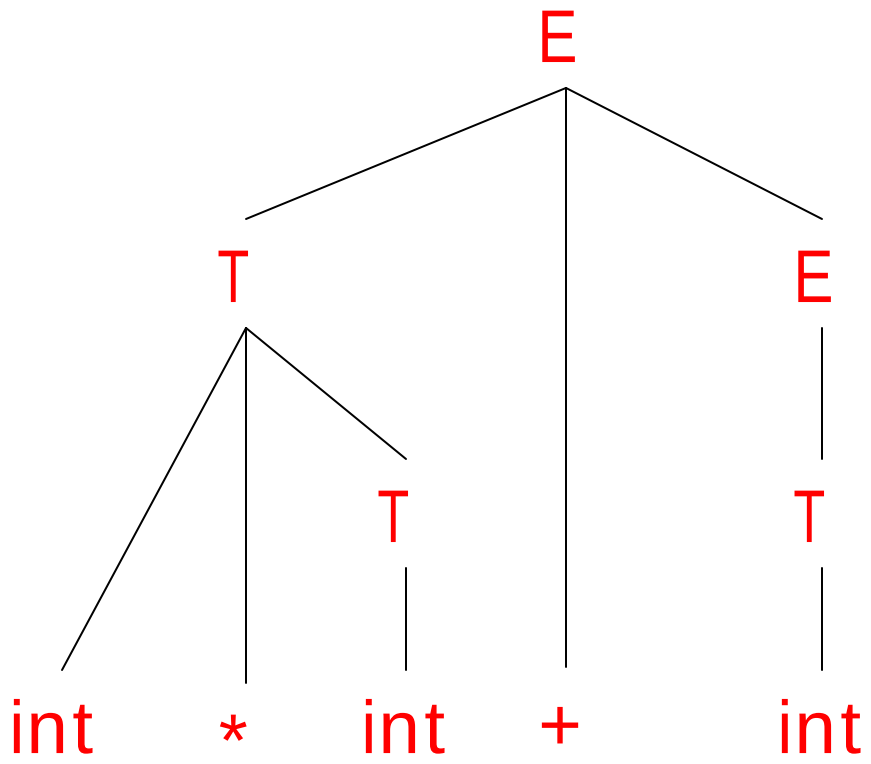
int * T + int

T + int

T + T

T + E

E



A bottom-up parser traces a rightmost derivation in reverse

A Bottom-up Parse in Detail (1)

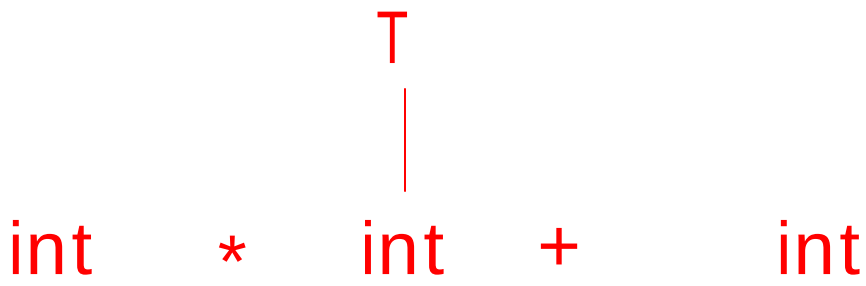
int * int + int

int * int + int

A Bottom-up Parse in Detail (2)

int * int + int

int * T + int

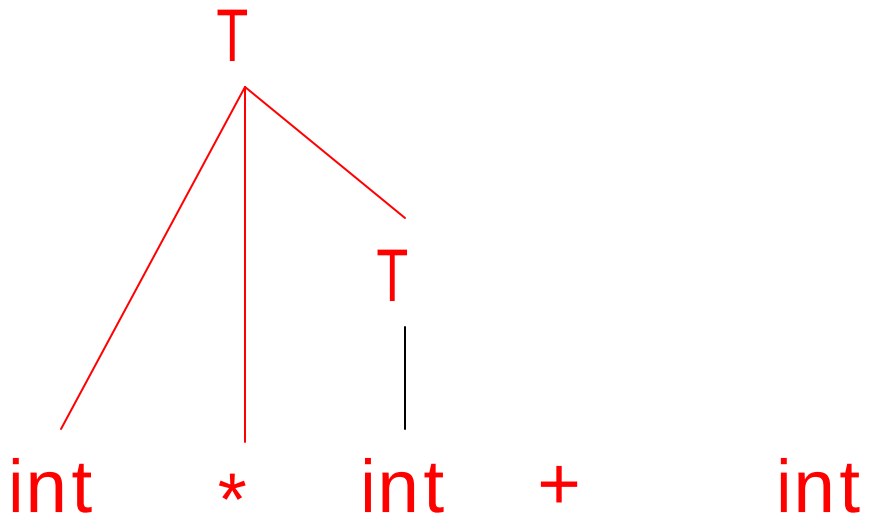


A Bottom-up Parse in Detail (3)

int * int + int

int * T + int

T + int



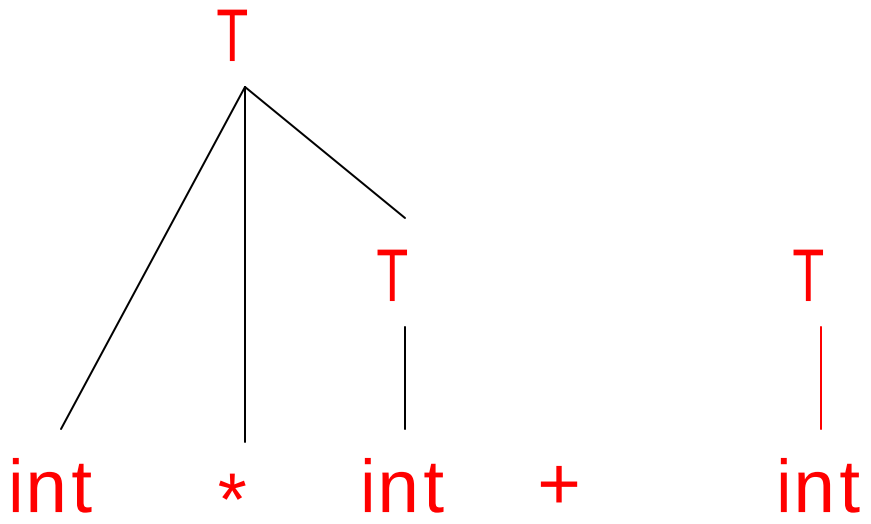
A Bottom-up Parse in Detail (4)

int * int + int

int * T + int

T + int

T + T



A Bottom-up Parse in Detail (5)

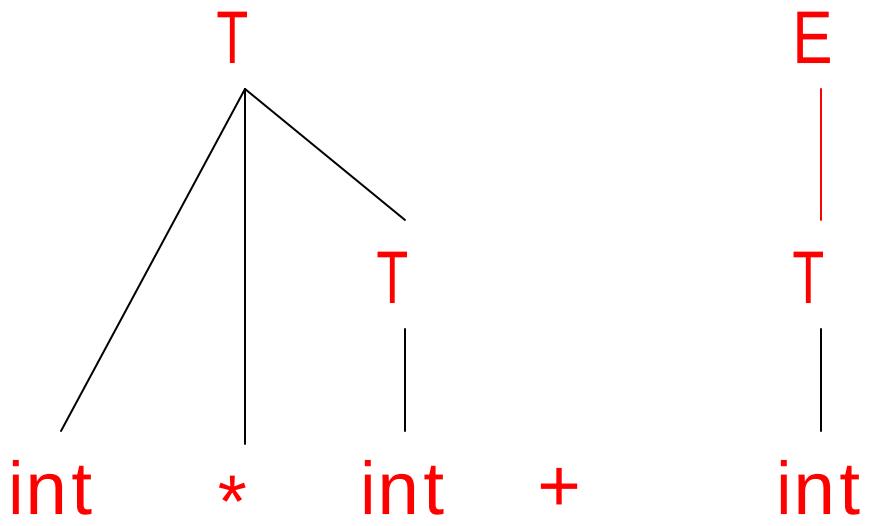
int * int + int

int * T + int

T + int

T + T

T + E



A Bottom-up Parse in Detail (6)

int * int + int

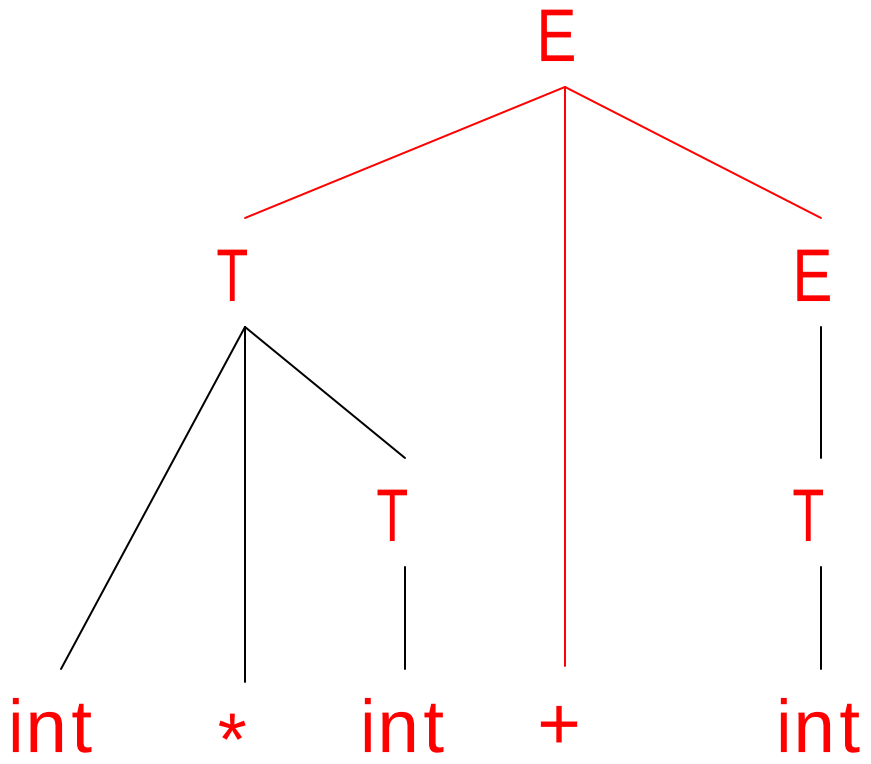
int * T + int

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

For You To Do

Do you see any problems with this algorithm?

Think about performance and completeness

Where Do Reductions Happen

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

Idea

- Split string into two substrings
  - Right substring : 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

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

  - xy is called a *handle*

The Example with Reductions

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 $\rightarrow$ int
int * T | + int	reduce T $\rightarrow$ int * T
T | + int	shift
T + | int	shift
T + int |	reduce T $\rightarrow$ int
T + T |	reduce E $\rightarrow$ T
T + E |	reduce E $\rightarrow$ 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

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

A Shift-Reduce Parse in Detail (5)

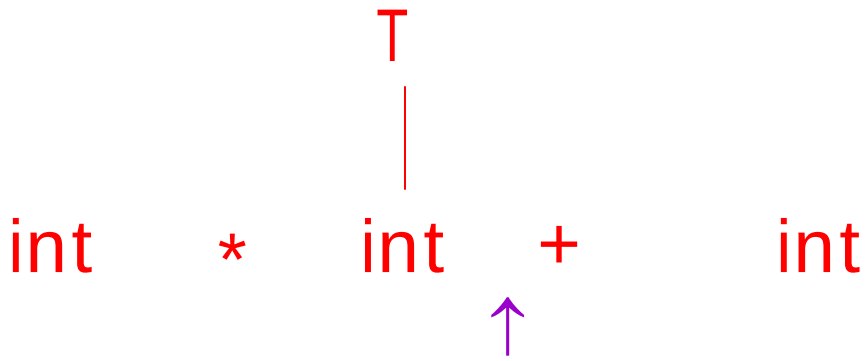
|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int



A Shift-Reduce Parse in Detail (6)

|int * int + int

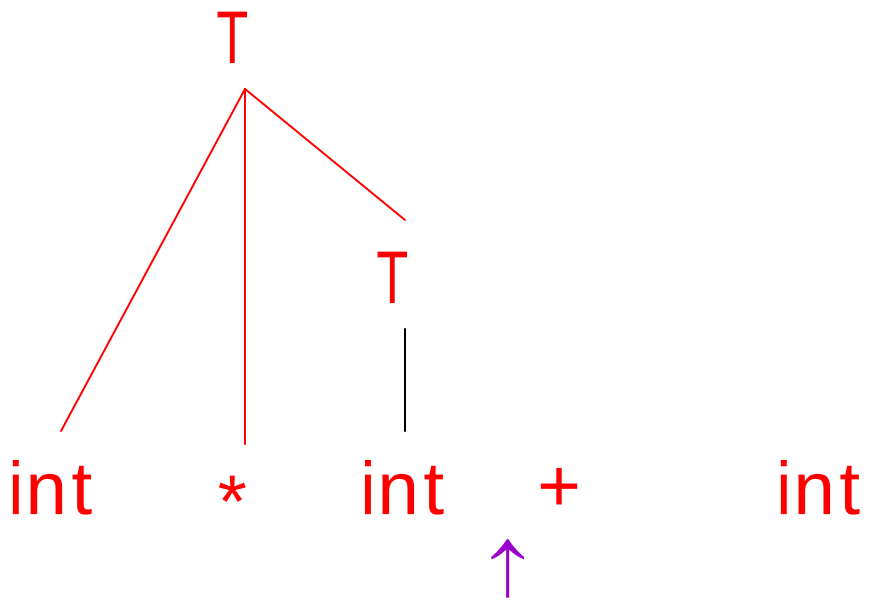
int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int



A Shift-Reduce Parse in Detail (7)

|int * int + int

int | * int + int

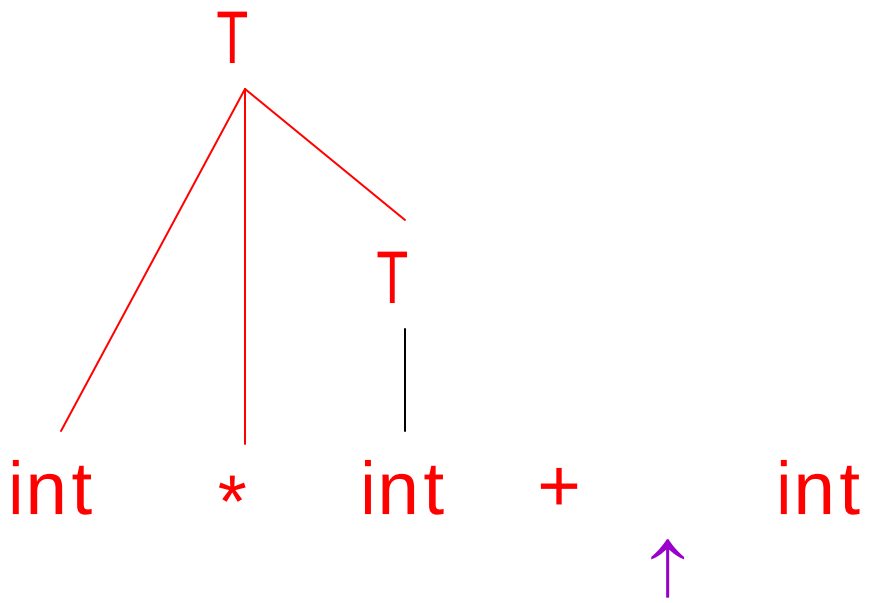
int * | int + int

int * int | + int

int * T | + int

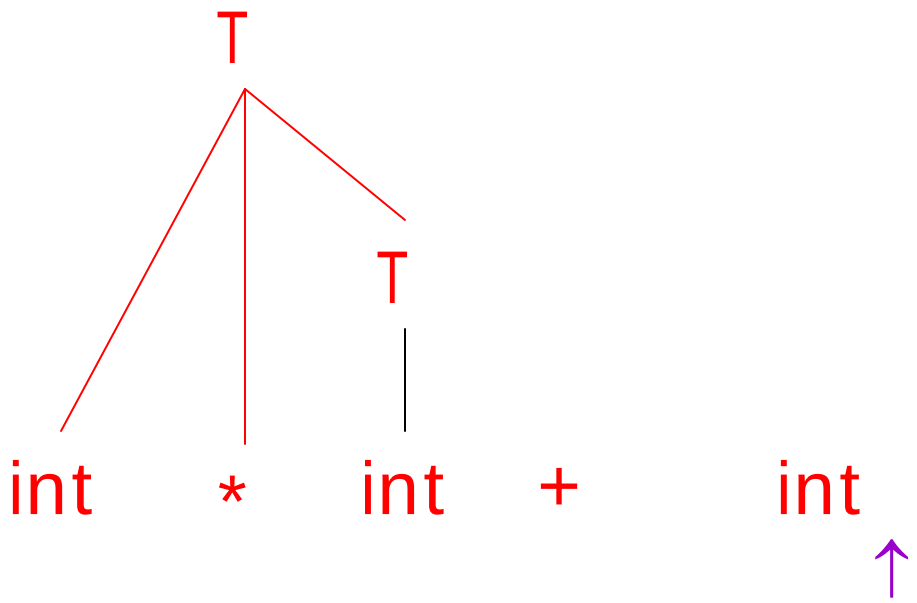
T | + int

T + | int



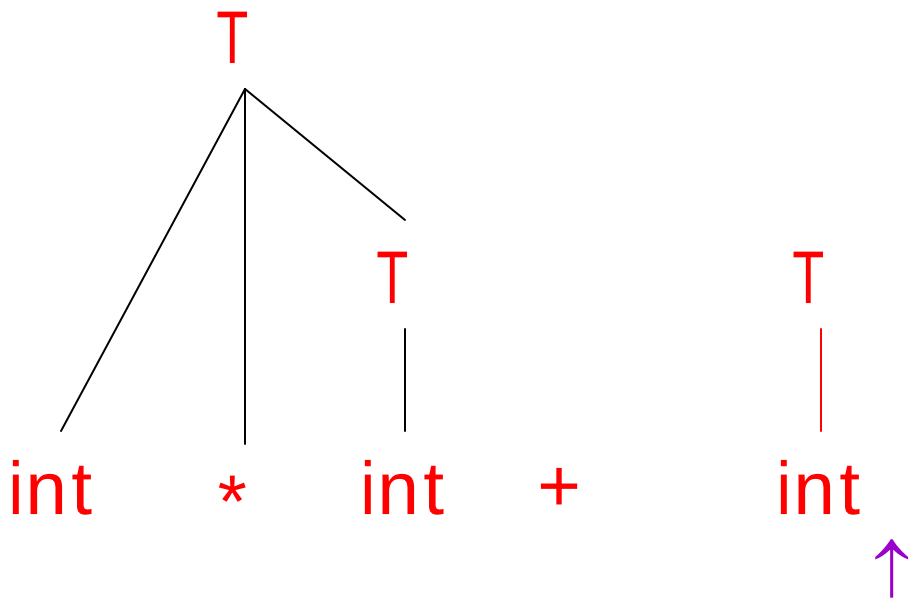
A Shift-Reduce Parse in Detail (8)

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



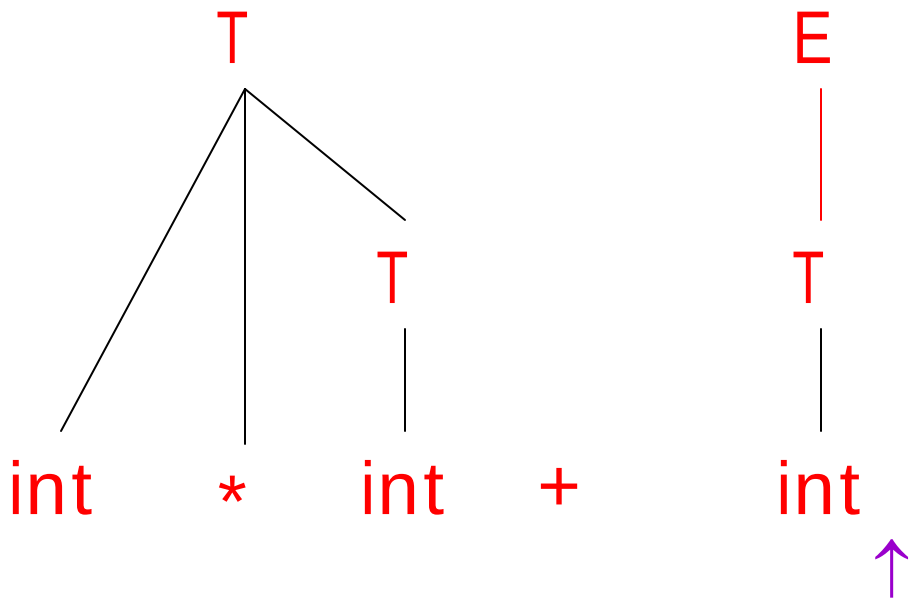
A Shift-Reduce Parse in Detail (9)

|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

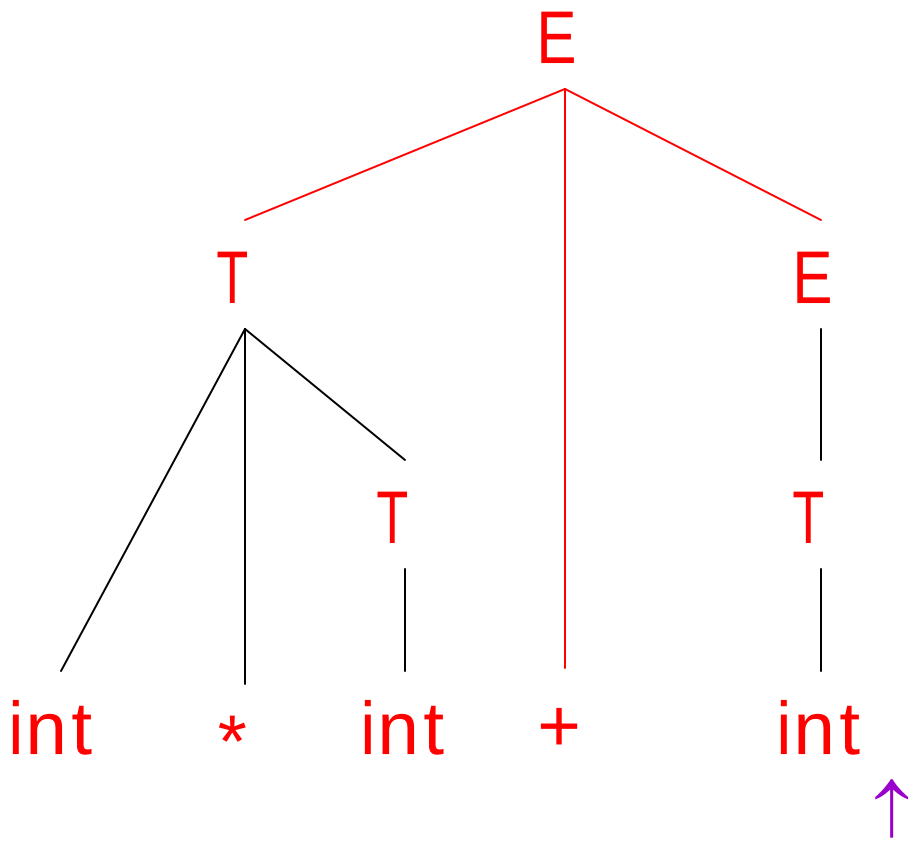
T + | int

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)

Key Issue

- How do we decide when to shift or reduce?
  - 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
- This is resolved by various bottom-up parsing algorithms

Conflicts

- Generic shift-reduce strategy:
  - If there is a handle on top of the stack, reduce
  - Otherwise, shift
- But what if there is a choice?
  - 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

Source of Conflicts

- Ambiguous grammars always cause conflicts
- But beware, so do many non-ambiguous grammars

Conflict Example

Consider our favorite ambiguous grammar:

$$
\begin{array}{lcl}
E & \rightarrow & E + E \\
 & | & E * E \\
 & | & (E) \\
 & | & \text{int}
\end{array}
$$

One Shift-Reduce Parse

|int * int + int	shift
. . .	. . .
E * E | + int	reduce E $\rightarrow$ E * E
E | + int	shift
E + | int	shift
E + int|	reduce E $\rightarrow$ int
E + E |	reduce E $\rightarrow$ E + E
E |	

Another Shift-Reduce Parse

|int * int + int	shift
. . .	. . .
E * E | + int	shift
E * E + | int	shift
E * E + int |	reduce E $\rightarrow$ int
E * E + E|	reduce E $\rightarrow$ E + E
E * E |	reduce E $\rightarrow$ E * E
E |	

Example Notes

- In the second step E * E | + int we can either shift or reduce by E → E * E
- Choice determines associativity of + and *
- As noted previously, grammar can be rewritten to enforce precedence
- Precedence declarations are an alternative

Precedence Declarations Revisited

- Precedence declarations cause shift-reduce parsers to resolve conflicts in certain ways
- Declaring "* has greater precedence than +" causes parser to reduce at E * E | + int
- More precisely, precedence declaration is used to resolve conflict between reducing a * and shifting a +

Precedence Declarations Revisited (Cont.)

- The term “precedence declaration” is a bit misleading

- These declarations do not define precedence; they define conflict resolutions
  - Not quite the same thing!