



Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
 - Just as efficient
 - Builds on ideas in top-down parsing
- Bottom-up is the preferred method in practice
- Reading: Section 4.5



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



Observation

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

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



A Bottom-up Parse

int * int + int

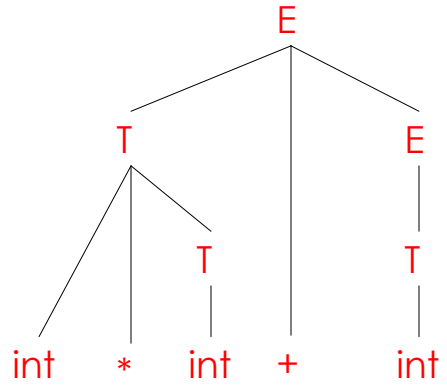
int * T + int

T + int

T + T

T + E

E



A bottom-up parser traces a rightmost derivation in reverse!



Bottom-up Parse in Detail

int * (int + int) + int



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 $|X_1X_2 \dots X_n$



Shift-Reduce Parsing

Bottom-up parsing uses 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

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



Example with Reductions Only

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

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

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

reduce $T \rightarrow \text{int}$
reduce $E \rightarrow T$
reduce $E \rightarrow T + E$



The Example with Shift-Reduce Parsing

int * int + int	shift
int * int + int	shift
int * int + int	shift
int * int + int	reduce $T \rightarrow \text{int}$
int * T + int	reduce $T \rightarrow \text{int} * T$
T + int	shift
T + int	shift
T + int	reduce $T \rightarrow \text{int}$
T + T	reduce $E \rightarrow T$
T + E	reduce $E \rightarrow T + E$
E	




Shift-Reduce Parse in Detail

int * (int + int) + int



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 (will be resolved by algorithms)

- How do we decide when to shift or reduce?
 - Consider step `int | * int + int`
 - We could reduce by $T \rightarrow \text{int}$ giving `T | * int + int`
 - A fatal mistake: No way to reduce to the start symbol E



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 $E \rightarrow \text{int}$
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 $E \rightarrow \text{int}$
E * E + E	reduce $E \rightarrow E + E$
E * E	reduce $E \rightarrow E * E$
E	



Example Notes

- In the second step $E * E | + \text{int}$ we can either shift or reduce by $E \rightarrow E * E$
- 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 $E * E \mid + int$
- More precisely, precedence declaration is used to resolve conflict between reducing a * and shifting a +



Precedence Declarations Revisited (Cont.)

- The term “precedence declaration” is misleading
- These declarations do not define precedence; they define conflict resolutions
 - Not quite the same thing!