

## Introduction to Bottom-Up Parsing

Lecture Notes by  
Profs. Alex Aiken and George Necula  
(UCB)

## Outline

---

- The strategy: *shift-reduce* parsing
- A key concept: *handles*
- *Ambiguity* and *precedence* declarations

## Predictive Parsing Summary

---

- *First* and *Follow* sets are used to construct predictive tables
  - For non-terminal  $A$  and input  $t$ , use a production  $A \rightarrow \alpha$  where  $t \in \text{First}(\alpha)$
  - For non-terminal  $A$  and input  $t$ , if  $t \in \text{Follow}(\alpha)$  and  $\varepsilon \in \text{First}(A)$ , use a production  $A \rightarrow \alpha$  where  $\varepsilon \in \text{First}(\alpha)$

## Bottom-Up Parsing

---

- Bottom-up parsing is more general than top-down parsing.
  - Don't need left-factored grammars.
  - Left recursion fine.
  - Just as efficient.
  - Builds on ideas in top-down parsing.
- Bottom-up parsing is the preferred method in practice.
  - Automatic parser generators: YACC, Bison, ...

## An Introductory Example

---

- 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:  $\text{int} * \text{int} + \text{int}$

## The Idea

---

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

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

## Observation

---

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

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

## Important Fact #1

---

Important Fact #1 about bottom-up parsing:

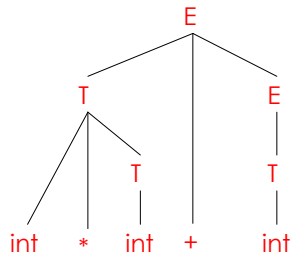
*A bottom-up parser traces a rightmost derivation in reverse.*

*LR-parser*

## A Bottom-up Parse

---

int \* int + int  
int \* T + int  
T + int  
T + T  
T + E  
E



## A Bottom-up Parse in Detail (1)

---

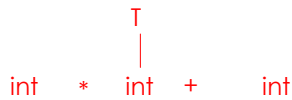
int \* int + int

int \* int + int

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

---

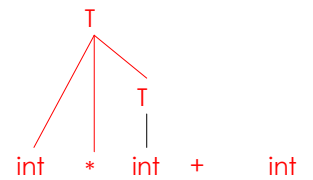
int \* int + int  
int \* T + int



## A Bottom-up Parse in Detail (3)

---

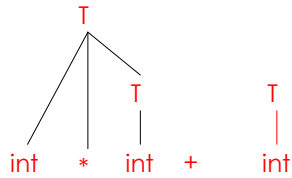
int \* int + int  
int \* T + int  
T + int



### A Bottom-up Parse in Detail (4)

---

int \* int + int  
int \* T + int  
T + int  
T + T



CS780(Prasad)

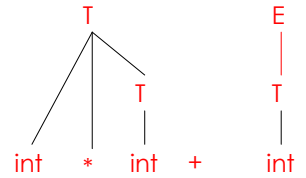
L12BUP

13

### A Bottom-up Parse in Detail (5)

---

int \* int + int  
int \* T + int  
T + int  
T + T  
T + E



CS780(Prasad)

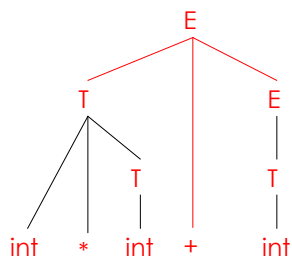
L12BUP

14

### A Bottom-up Parse in Detail (6)

---

int \* int + int  
int \* T + int  
T + int  
T + T  
T + E  
E



CS780(Prasad)

L12BUP

15

### A Trivial Bottom-Up Parsing Algorithm

---

Let I = input string  
repeat  
    pick a non-empty substring  $\beta$  of I  
        where  $X \rightarrow \beta$  is a production  
    if no such  $\beta$ , backtrack  
    replace one  $\beta$  by X in I  
until I = "S" (the start symbol) or all possibilities are exhausted

CS780(Prasad)

L12BUP

16

## Questions

---

- Does this algorithm terminate?
- How fast is the algorithm?
- Does the algorithm deal with 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  $\omega$  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 parser (hence is 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 only *two* kinds of actions:

- *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  $T \rightarrow int$   
int \* T | + int      reduce  $T \rightarrow int * T$

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

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

int \* int + int  
          |  
          T  
          ↑

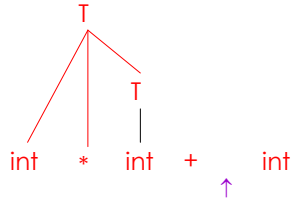
### A Shift-Reduce Parse in Detail (6)

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

int \* int + int  
          |  
          T  
          /  |  \  
int  \*  int  +  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



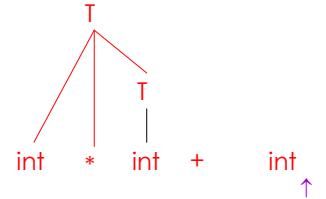
CS780(Prasad)

L12BUP

29

### 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 |



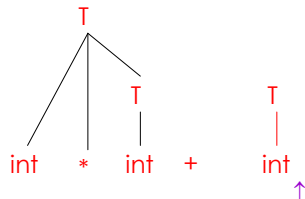
CS780(Prasad)

L12BUP

30

### 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 |



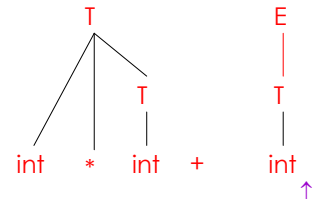
CS780(Prasad)

L12BUP

31

### 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 |



CS780(Prasad)

L12BUP

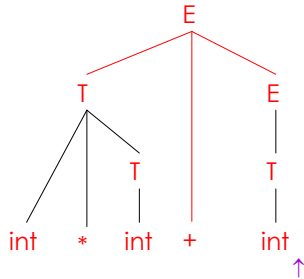
32



## 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 |
    
```



L12BUP

33

## The Stack

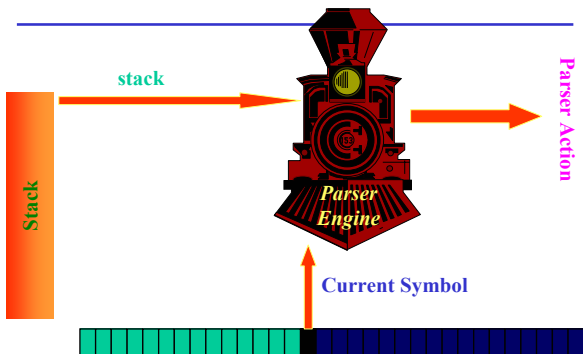
- Left string can be implemented by a **stack**
  - Top of the **stack** is the |
- **Shift** pushes a terminal on the **stack**.
- **Reduce** pops 0 or more symbols off the **stack** (production rhs) and **pushes** a non-terminal on the **stack** (production lhs).

CS780(Prasad)

L12BUP

34

## Shift-Reduce Parser



## 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:* Because there is no way to **reduce** to the start symbol `E`.

```

E → T + E | T
T → int * T | int | (E)
    
```

CS780(Prasad)

L12BUP

36

## Handles

---

- **Intuition:** Want to **reduce** only if the result can still be reduced to the start symbol.
- Assume a rightmost derivation:  
$$S \Rightarrow^* \alpha X \omega \rightarrow \alpha \beta \omega$$
- Then  $\alpha\beta$  is a **handle** of  $\alpha\beta\omega$ .

## Handles (Cont.)

---

- A **handle** is a string that can be **reduced**, and that also allows further **reductions** back to the start symbol.
- 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.*

## Why?

---

- Informal induction on # of **reduce** moves:
- True initially, **stack** is empty
- Immediately after **reducing** a handle
  - right-most non-terminal on top of the stack.
  - next handle must be to right of right-most non-terminal, because this is a right-most derivation.
  - Sequence of shift moves reaches next handle.

## Summary of Handles

---

- In shift-reduce parsing, handles always appear at the top of the stack.
- Handles are never to the left of the rightmost non-terminal.
  - Therefore, shift-reduce moves are sufficient; the  $|$  need never move left.
- Bottom-up parsing algorithms are based on recognizing handles.

## 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.

Consider our favorite ambiguous grammar:

E	→	E + E
		E * E
		(E)
		int

## One Shift-Reduce Parse

---

int * int + int	shift
...	...
E * E   + int	reduce E → E * E
E   + int	shift
E +   int	shift
E + int	reduce E → int
E + E	reduce E → 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** and **precedence** 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  $E * E | + \text{int}$ .
- More precisely, **precedence declaration** is used to resolve conflict between **reducing a  $*$**  and **shifting a  $+$**