

# Top-Down Parsing and Intro to Bottom-Up Parsing

## Lecture 7

Prof. Aiken CS 143 Lecture 7

1

## Predictive Parsers

- Like recursive-descent but 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

Prof. Aiken CS 143 Lecture 7

2

## 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  $\dagger$
    - 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

Prof. Aiken CS 143 Lecture 7

3

## 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  $\text{int}$
  - For  $E$  it is not clear how to predict
- We need to left-factor the grammar

Prof. Aiken CS 143 Lecture 7

4

## 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
 
$$E \rightarrow TX$$

$$X \rightarrow + E \mid \epsilon$$

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

$$Y \rightarrow * T \mid \epsilon$$

Prof. Aiken CS 143 Lecture 7

5

## LL(1) Parsing Table Example

- Left-factored grammar
 
$$E \rightarrow TX$$

$$X \rightarrow + E \mid \epsilon$$

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

$$Y \rightarrow * T \mid \epsilon$$
- The LL(1) parsing table: *next input token*

	int	*	+	(	)	\$
E	TX			TX		
X			+E		$\epsilon$	$\epsilon$
T	int Y			(E)		
Y		*T	$\epsilon$		$\epsilon$	$\epsilon$

*leftmost non-terminal*

*rhs of production to use*

Prof. Aiken CS 143 Lecture 7

6

### LL(1) Parsing Table Example (Cont.)

- Consider the  $[E, \text{int}]$  entry
  - “When current non-terminal is  $E$  and next input is  $\text{int}$ , use production  $E \rightarrow TX$ ”
  - This can generate an  $\text{int}$  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  $Y \rightarrow \epsilon$

Prof. Aiken CS 143 Lecture 7

7

### 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  $E$ ”

Prof. Aiken CS 143 Lecture 7

8

### Using Parsing Tables

- Method similar to recursive descent, except
  - For the leftmost non-terminal  $S$
  - We look at the next input token  $a$
  - And choose the production shown at  $[S, a]$
- A stack records frontier of parse tree
  - Non-terminals that have yet to be expanded
  - Terminals that have yet to be matched against the input
  - Top of stack = leftmost pending terminal or non-terminal
- Reject on reaching error state
- Accept on end of input & empty stack

Prof. Aiken CS 143 Lecture 7

9

### LL(1) Parsing Algorithm

```

initialize stack = <S $> and next
repeat
  case stack of
    <X, rest> : if  $T[X, *next] = Y_1 \dots Y_n$ 
                then stack  $\leftarrow <Y_1 \dots Y_n \text{ rest}>$ ;
                else error ();
    <t, rest> : if  $t == *next ++$ 
                then stack  $\leftarrow <\text{rest}>$ ;
                else error ();
until stack == <>
    
```

Prof. Aiken CS 143 Lecture 7

10

### LL(1) Parsing Algorithm

```

initialize stack = <S $> and next
repeat
  case stack of
    <X, rest> : if  $T[X, *next] = Y_1 \dots Y_n$ 
                then stack  $\leftarrow <Y_1 \dots Y_n \text{ rest}>$ ;
                else error ();
    <t, rest> : if  $t == *next ++$ 
                then stack  $\leftarrow <\text{rest}>$ ;
                else error ();
until stack == <>
    
```

*\$ marks bottom of stack*

*For non-terminal X on top of stack, lookup production*

*Pop X, push production rhs on stack. Note leftmost symbol of rhs is on top of the stack.*

*For terminal t on top of stack, check t matches next input token.*

Prof. Aiken CS 143 Lecture 7

11

### LL(1) Parsing Example

Stack	Input	Action
E \$	int * int \$	$TX$
$TX \$$	int * int \$	int Y
int Y X \$	int * int \$	terminal
Y X \$	* int \$	* T
* TX \$	* int \$	terminal
TX \$	int \$	int Y
int Y X \$	int \$	terminal
Y X \$	\$	$\epsilon$
X \$	\$	$\epsilon$
\$	\$	ACCEPT

Prof. Aiken CS 143 Lecture 7

12

## Constructing Parsing Tables: The Intuition

- Consider non-terminal  $A$ , production  $A \rightarrow \alpha$ , & token  $t$
- $T[A, t] = \alpha$  in two cases:
  - If  $\alpha \rightarrow^* t \beta$ 
    - $\alpha$  can derive a  $t$  in the first position
    - We say that  $t \in \text{First}(\alpha)$
  - If  $A \rightarrow \alpha$  and  $\alpha \rightarrow^* \epsilon$  and  $S \rightarrow^* \beta A \delta$ 
    - 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  $\epsilon$ )
      - Can work only if  $t$  can follow  $A$  in at least one derivation
    - We say  $t \in \text{Follow}(A)$

Prof. Aiken CS 143 Lecture 7

13

## Computing First Sets

Definition

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

Algorithm sketch:

- $\text{First}(t) = \{ t \}$
- $\epsilon \in \text{First}(X)$ 
  - if  $X \rightarrow \epsilon$
  - if  $X \rightarrow A_1 \dots A_n$  and  $\epsilon \in \text{First}(A_i)$  for  $1 \leq i \leq n$
- $\text{First}(\alpha) \subseteq \text{First}(X)$  if  $X \rightarrow A_1 \dots A_n \alpha$ 
  - and  $\epsilon \in \text{First}(A_i)$  for  $1 \leq i \leq n$

Prof. Aiken CS 143 Lecture 7

14

## First Sets. Example

- Recall the grammar

$$\begin{array}{ll} E \rightarrow TX & X \rightarrow +E \mid \epsilon \\ T \rightarrow (E) \mid \text{int } Y & Y \rightarrow *T \mid \epsilon \end{array}$$

- First sets

$$\begin{array}{ll} \text{First}(( ) ) = \{ ( \} & \text{First}(T) = \{ \text{int}, ( \} \\ \text{First}() ) = \{ \} & \text{First}(E) = \{ \text{int}, ( \} \\ \text{First}(\text{int}) = \{ \text{int} \} & \text{First}(X) = \{ +, \epsilon \} \\ \text{First}(+) = \{ + \} & \text{First}(Y) = \{ *, \epsilon \} \\ \text{First}(*) = \{ * \} & \end{array}$$

Prof. Aiken CS 143 Lecture 7

15

## 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  $B \rightarrow^* \epsilon$  then  $\text{Follow}(X) \subseteq \text{Follow}(A)$
- If  $S$  is the start symbol then  $\$ \in \text{Follow}(S)$

Prof. Aiken CS 143 Lecture 7

16

## Computing Follow Sets (Cont.)

Algorithm sketch:

- $\$ \in \text{Follow}(S)$
- $\text{First}(\beta) - \{ \epsilon \} \subseteq \text{Follow}(X)$ 
  - For each production  $A \rightarrow \alpha X \beta$
- $\text{Follow}(A) \subseteq \text{Follow}(X)$ 
  - For each production  $A \rightarrow \alpha X \beta$  where  $\epsilon \in \text{First}(\beta)$

Prof. Aiken CS 143 Lecture 7

17

## Follow Sets. Example

- Recall the grammar

$$\begin{array}{ll} E \rightarrow TX & X \rightarrow +E \mid \epsilon \\ T \rightarrow (E) \mid \text{int } Y & Y \rightarrow *T \mid \epsilon \end{array}$$

- Follow sets

$$\begin{array}{ll} \text{Follow}(+) = \{ \text{int}, ( \} & \text{Follow}(*) = \{ \text{int}, ( \} \\ \text{Follow}(( ) ) = \{ \text{int}, ( \} & \text{Follow}(E) = \{ \}, \$ \} \\ \text{Follow}(X) = \{ \$, ) \} & \text{Follow}(T) = \{ +, ) , \$ \} \\ \text{Follow}() ) = \{ +, ) , \$ \} & \text{Follow}(Y) = \{ +, ) , \$ \} \\ \text{Follow}(\text{int}) = \{ *, +, ) , \$ \} & \end{array}$$

Prof. Aiken CS 143 Lecture 7

18

### Constructing LL(1) Parsing Tables

- Construct a parsing table  $T$  for CFG  $G$
- For each production  $A \rightarrow \alpha$  in  $G$  do:
  - For each terminal  $t \in \text{First}(\alpha)$  do
    - $T[A, t] = \alpha$
  - If  $\epsilon \in \text{First}(\alpha)$ , for each  $t \in \text{Follow}(A)$  do
    - $T[A, t] = \alpha$
  - If  $\epsilon \in \text{First}(\alpha)$  and  $\$ \in \text{Follow}(A)$  do
    - $T[A, \$] = \alpha$

Prof. Aiken CS 143 Lecture 7

19

### 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)

Prof. Aiken CS 143 Lecture 7

20

### 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

Prof. Aiken CS 143 Lecture 7

21

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

Prof. Aiken CS 143 Lecture 7

22

### The Idea

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

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

Prof. Aiken CS 143 Lecture 7

23

### Observation

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

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

Prof. Aiken CS 143 Lecture 7

24

## Important Fact #1

Important Fact #1 about bottom-up parsing:

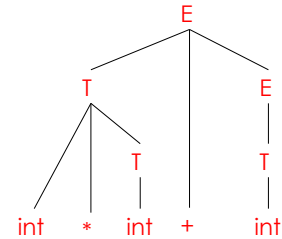
*A bottom-up parser traces a rightmost derivation in reverse*

Prof. Aiken CS 143 Lecture 7

25

## A Bottom-up Parse

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



Prof. Aiken CS 143 Lecture 7

26

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

int \* int + int

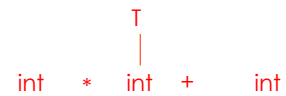
int \* int + int

Prof. Aiken CS 143 Lecture 7

27

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

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

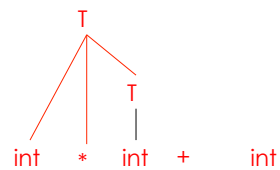


Prof. Aiken CS 143 Lecture 7

28

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

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

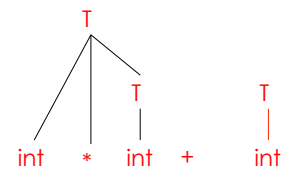


Prof. Aiken CS 143 Lecture 7

29

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

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

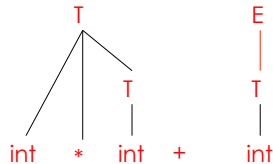


Prof. Aiken CS 143 Lecture 7

30

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

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

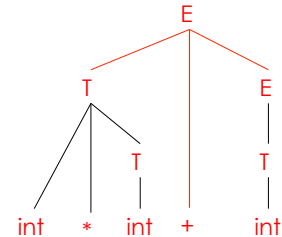


Prof. Aiken CS 143 Lecture 7

31

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

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



Prof. Aiken CS 143 Lecture 7

32

### 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

Prof. Aiken CS 143 Lecture 7

33

### 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?

Prof. Aiken CS 143 Lecture 7

34

### 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

Prof. Aiken CS 143 Lecture 7

35

### 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  $|x_1x_2 \dots x_n$

Prof. Aiken CS 143 Lecture 7

36

## Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

*Shift*

*Reduce*

Prof. Aiken CS 143 Lecture 7

37

## Shift

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

$ABC|xyz \Rightarrow ABCx|yz$

Prof. Aiken CS 143 Lecture 7

38

## Reduce

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

$Cbxy|ijk \Rightarrow CbA|ijk$

Prof. Aiken CS 143 Lecture 7

39

## 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  $	

Prof. Aiken CS 143 Lecture 7

40

## 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	

Prof. Aiken CS 143 Lecture 7

41

## A Shift-Reduce Parse in Detail (1)

| int \* int + int

int \* int + int  
↑

Prof. Aiken CS 143 Lecture 7

42

### A Shift-Reduce Parse in Detail (2)

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

int \* int + int  
↑

Prof. Aiken CS 143 Lecture 7

43

### A Shift-Reduce Parse in Detail (3)

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

int \* int + int  
↑

Prof. Aiken CS 143 Lecture 7

44

### A Shift-Reduce Parse in Detail (4)

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

int \* int + int  
↑

Prof. Aiken CS 143 Lecture 7

45

### A Shift-Reduce Parse in Detail (5)

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

T  
|  
int \* int + int  
↑

Prof. Aiken CS 143 Lecture 7

46

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

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

T  
/ | \  
int \* int + int  
↑

Prof. Aiken CS 143 Lecture 7

47

### 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

T  
/ | \  
int \* int + int  
↑

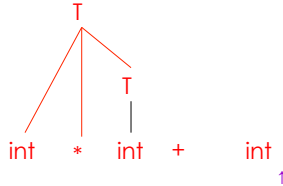
Prof. Aiken CS 143 Lecture 7

48



### 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 |
```

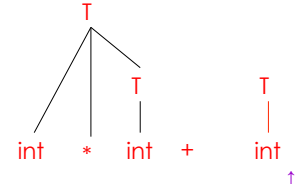


Prof. Aiken CS 143 Lecture 7

49

### 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 |
```

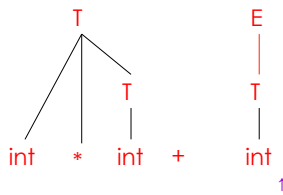


Prof. Aiken CS 143 Lecture 7

50

### 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 |
```

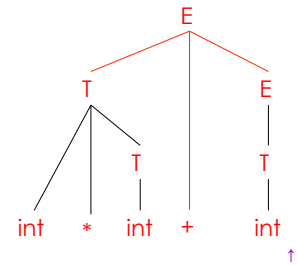


Prof. Aiken CS 143 Lecture 7

51

### 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 |
```



Prof. Aiken CS 143 Lecture 7

52

### 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)

Prof. Aiken CS 143 Lecture 7

53

### 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...

Prof. Aiken CS 143 Lecture 7

54