The more things change...

LR Parsing

Bottom-Up Parsing

#1

Outline

- No Stopping The Parsing!
- Bottom-Up Parsing
- LR Parsing
  - Shift and Reduce
  - LR(1) Parsing Algorithm
- LR(1) Parsing Tables

#2

In One Slide

- An **LR(1) parser** reads tokens from left to right and constructs a bottom-up rightmost derivation. LR(1) parsers shift terminals and reduce the input by application productions in reverse. LR(1) parsing is fast and easy, and uses a finite automaton with a stack. LR(1) works fine if the grammar is left-recursive, or not left-factored.

#3

Bottom-Up Parsing

- **Bottom-up parsing** is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
  - Preferred method in practice
- Also called **LR parsing**
  - L means that tokens are read left to right
  - R means that it constructs a rightmost derivation

#4

An Introductory Example

- LR parsers don't need left-factored grammars and can also handle left-recursive grammars
- Consider the following grammar:

  $E \rightarrow E + (E) \mid int$

  - Why is this not LL(1)? (Guess before I show you!)
- Consider the string: **int + (int) + (int)**

#5

The Idea

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

str $\leftarrow$ input string of terminals
repeat
  - Identify β in str such that $A \rightarrow \beta$ is a production (i.e., str = $\alpha\ \beta\ \gamma$)
  - Replace β by A in str (i.e., str becomes $\alpha\ A\ \gamma$)

until str = S

#6

A Bottom-up Parse in Detail (1)

int + (int) + (int)

int + (int) + (int)

#7

A Bottom-up Parse in Detail (2)

int + (int) + (int)
E + (int) + (int)

E
|
int + (int) + (int)

#8

A Bottom-up Parse in Detail (3)

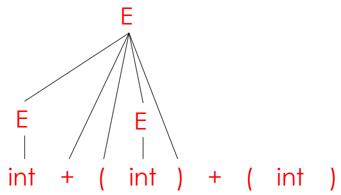
int + (int) + (int)
E + (int) + (int)
E + (E) + (int)

E   E
| |
int + (int) + (int)

#9

A Bottom-up Parse in Detail (4)

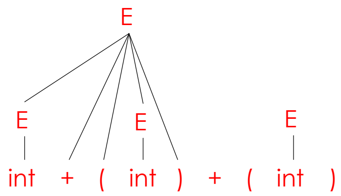
int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)



#10

A Bottom-up Parse in Detail (5)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)

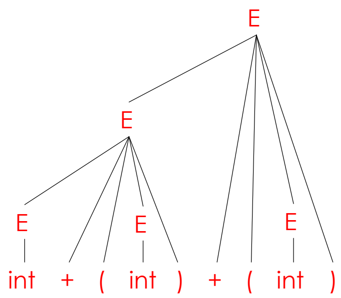


#11

A Bottom-up Parse in Detail (6)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)
E

A rightmost derivation *in reverse*



#12

Important Fact

Important Fact #1 about bottom-up parsing:

An LR parser traces a rightmost derivation in reverse.

#13

Where Do Reductions Happen

Important Fact #1 has an interesting consequence:
- Let $\alpha\beta\gamma$ be a step of a bottom-up parse
- Assume the next reduction is by $A \rightarrow \beta$
- Then γ is a string of **terminals**!

Why? Because $\alpha A\gamma \rightarrow \alpha\beta\gamma$ is a step in a right-most derivation

#14

Notation

- Idea: Split the string into two substrings
  - Right substring (a string of terminals) is as yet unexamined by parser
  - 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 new: ►$x_1 x_2 \ . \ . \ . \ x_n$

#15

Shift-Reduce Parsing

- Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

#16

Shift

Shift: Move ► one place to the right

- Shifts a terminal to the left string

E + (► int)

⇒

E + (int ►)

#17

Reduce

Reduce: Apply an inverse production at the right end of the left string

- If T → E + (E) is a production, then

E + (E + (E) ►)

⇒

E +(T ►)

Reductions can only happen here!

#18

Shift-Reduce Example

▸ int + (int) + (int)$ shift

int + (int) + (int)

↑

Shift-Reduce Example

▸ int + (int) + (int)$ shift

int ▸ + (int) + (int)$ red. E → int

int + (int) + (int)

↑

Shift-Reduce Example

▸ int + (int) + (int)$ shift

int ▸ + (int) + (int)$ red. E → int

E ▸ + (int) + (int)$ shift 3 times

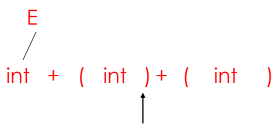
E

int + (int) + (int)

↑

Shift-Reduce Example

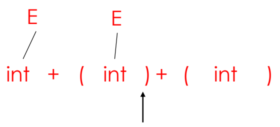
▶ int + (int) + (int)$	shift
int ▶ + (int) + (int)$	red. E → int
E ▶ + (int) + (int)$	shift 3 times
E + (int ▶) + (int)$	red. E → int



#22

Shift-Reduce Example

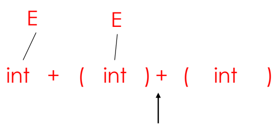
▶ int + (int) + (int)$	shift
int ▶ + (int) + (int)$	red. E → int
E ▶ + (int) + (int)$	shift 3 times
E + (int ▶) + (int)$	red. E → int
E + (E ▶) + (int)$	shift



#23

Shift-Reduce Example

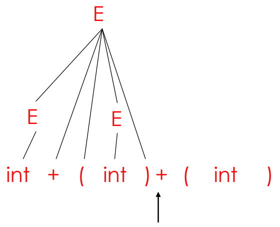
▶ int + (int) + (int)$	shift
int ▶ + (int) + (int)$	red. E → int
E ▶ + (int) + (int)$	shift 3 times
E + (int ▶) + (int)$	red. E → int
E + (E ▶) + (int)$	shift
E + (E) ▶ + (int)$	red. E → E + (E)



#24

Shift-Reduce Example

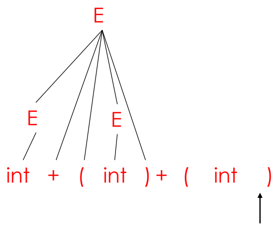
▶ int + (int) + (int)$ shift
int ▶ + (int) + (int)$ red. E → int
E ▶ + (int) + (int)$ shift 3 times
E + (int ▶) + (int)$ red. E → int
E + (E ▶) + (int)$ shift
E + (E) ▶ + (int)$ red. E → E + (E)
E ▶ + (int)$ shift 3 times



#25

Shift-Reduce Example

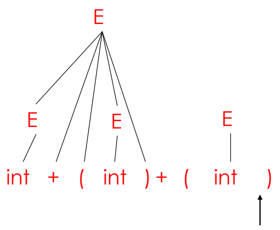
▶ int + (int) + (int)$ shift
int ▶ + (int) + (int)$ red. E → int
E ▶ + (int) + (int)$ shift 3 times
E + (int ▶) + (int)$ red. E → int
E + (E ▶) + (int)$ shift
E + (E) ▶ + (int)$ red. E → E + (E)
E ▶ + (int)$ shift 3 times
E + (int ▶)$ red. E → int



#26

Shift-Reduce Example

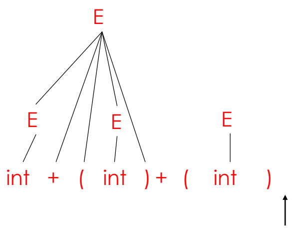
▶ int + (int) + (int)$ shift
int ▶ + (int) + (int)$ red. E → int
E ▶ + (int) + (int)$ shift 3 times
E + (int ▶) + (int)$ red. E → int
E + (E ▶) + (int)$ shift
E + (E) ▶ + (int)$ red. E → E + (E)
E ▶ + (int)$ shift 3 times
E + (int ▶)$ red. E → int
E + (E ▶)$ shift



#27

Shift-Reduce Example

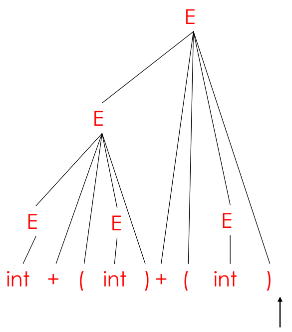
► int + (int) + (int)$ shift
int ► + (int) + (int)$ red. E → int
E ► + (int) + (int)$ shift 3 times
E + (int ►) + (int)$ red. E → int
E + (E ►) + (int)$ shift
E + (E) ► + (int)$ red. E → E + (E)
E ► + (int)$ shift 3 times
E + (int ►)$ red. E → int
E + (E ►)$ shift
E + (E) ► $ red. E → E + (E)



#28

Shift-Reduce Example

► int + (int) + (int)$ shift
int ► + (int) + (int)$ red. E → int
E ► + (int) + (int)$ shift 3 times
E + (int ►) + (int)$ red. E → int
E + (E ►) + (int)$ shift
E + (E) ► + (int)$ red. E → E + (E)
E ► + (int)$ shift 3 times
E + (int ►)$ red. E → int
E + (E ►)$ shift
E + (E) ► $ red. E → E + (E)
E ► $ accept



#29

The Stack

- Left string can be implemented as a stack
  - Top of the stack is the ►
- Shift pushes a terminal on the stack
- Reduce pops 0 or more symbols from the stack (production RHS) and pushes a non-terminal on the stack (production LHS)

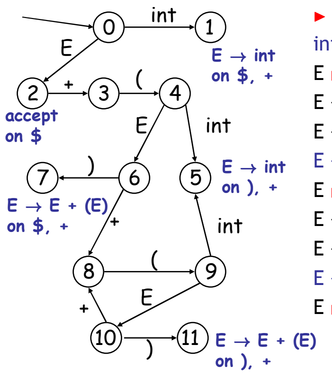
#30

Key Issue: When to Shift or Reduce?

- Decide based on the left string (the stack)
- Idea: use a finite automaton (DFA) to decide when to shift or reduce
  - The DFA input is the stack
  - DFA language consists of terminals and nonterminals
- We run the DFA on the stack and we examine the resulting state X and the token tok after ▸
  - If X has a transition labeled tok then shift
  - If X is labeled with "A → β on tok" then reduce

#31

LR(1) Parsing Example



▸ int + (int) + (int)$	shift
int ▸ + (int) + (int)$	E → int
E ▸ + (int) + (int)$	shift(x3)
E + (int ▸) + (int)$	E → int
E + (E ▸) + (int)$	shift
E + (E) ▸ + (int)$	E → E+(E)
E ▸ + (int)$	shift (x3)
E + (int ▸)$	E → int
E + (E ▸)$	shift
E + (E) ▸ $	E → E+(E)
E ▸ $	accept

#32

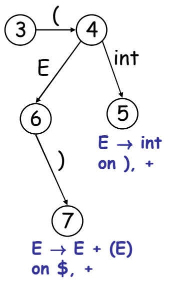
Representing the DFA

- Parsers represent the DFA as a 2D table
  - Recall table-driven lexical analysis
- Lines (rows) correspond to DFA states
- Columns correspond to terminals and non-terminals
- Typically columns are split into:
  - Those for terminals: action table
  - Those for non-terminals: goto table

#33

Representing the DFA. Example

- The table for a fragment of our DFA:



	int	+	(	)	$	E
…						
3			s4			
4	s5					g6
5		$r_{E \to \text{int}}$		$r_{E \to \text{int}}$		
6	s8		s7			
7		$r_{E \to E+(E)}$			$r_{E \to E+(E)}$	
…						

#34

The LR Parsing Algorithm

- After a shift or reduce action we rerun the DFA on the entire stack
  - This is wasteful, since most of the work is repeated
- Optimization: remember for each stack element to which state it brings the DFA
- LR parser maintains a stack

  $< sym_1, state_1 > \ldots < sym_n, state_n >$

  $state_k$ is the final state of the DFA on $sym_1 \ldots sym_k$

#35

The LR Parsing Algorithm

```
Let S = w$ be initial input
Let j = 0
Let DFA state 0 be the start state
Let stack = < dummy, 0 >
   repeat
        match action[top_state(stack), S[j]] with
               | shift k:  push < S[j++], k >
               | reduce X → α:
                     pop |α| pairs,
                     push < X, Goto[top_state(stack), X] >
               | accept: halt normally
               | error: halt and report error
```

#36

LR Parsing Notes

- Can be used to parse more grammars than LL
- Most PL grammars are LR
- Can be described as a simple table
- There are tools for building the table
  - Often called "yacc" or "bison"
- How is the table constructed? Next time!

#37

Homework

- Thursday: WA2 due
  - You may work in pairs.
- Thursday: Read 2.3.4-2.3.5, 2.4.2-2.4.3
- Next Friday: WA3 due
  - Parsing!

#38