LR Parsing

Lecture 7

January 30, 2018

Midterm 1 Next Week

- **in-class exam**
- One sheet letter paper, two sides
- Some review questions online
- Things to know
  - Cool Syntax
  - Compiler/language processor steps
  - Regular Languages
  - NFA/DFAs
  - Regular Expressions
  - Parsing
  - Context-Free Grammars

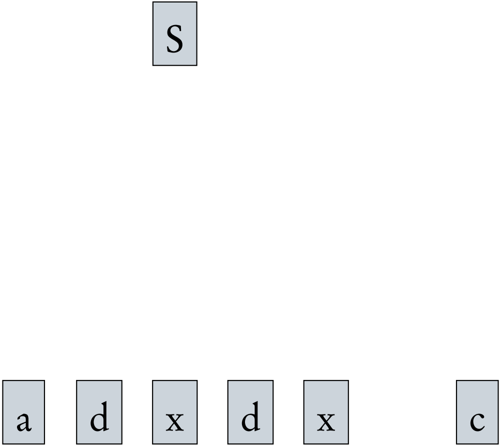
From Last Week

- Top-down Parsing strategies
  - Tells us *how* to get from the **start symbol** of a **grammar** to a sequence of **terminals** following a sequence of **productions**

Top Down Parsing

$$
\begin{array}{lcl}
S & \to & a \;\; B \;\; c \\
B & \to & C \;\; x \;\; B \\
B & \to & \epsilon \\
C & \to & d \\
  & | & a \;\; B \;\; c
\end{array}
$$

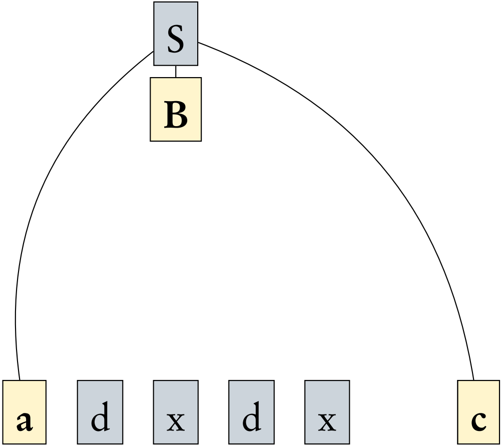
Input string:
"adxdxc"



Top Down Parsing

$$
\begin{array}{lcllll}
S & \rightarrow & a & B & c \\
B & \rightarrow & C & x & B \\
B & \rightarrow & \epsilon \\
C & \rightarrow & d \\
 & | & a & B & c
\end{array}
$$

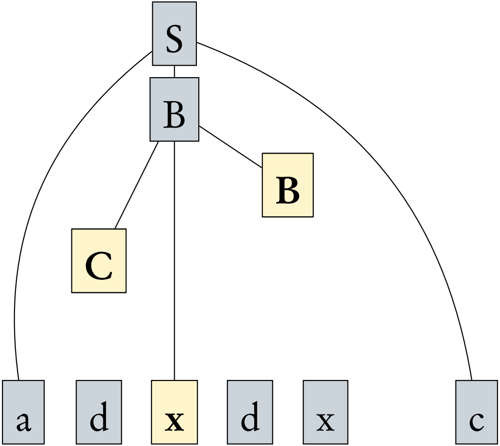
Input string:
"adxdxc"



Top Down Parsing

$$
\begin{array}{lcl}
S & \rightarrow & a \; B \; c \\
B & \rightarrow & C \; x \; B \\
B & \rightarrow & \epsilon \\
C & \rightarrow & d \\
  & | & a \; B \; c
\end{array}
$$

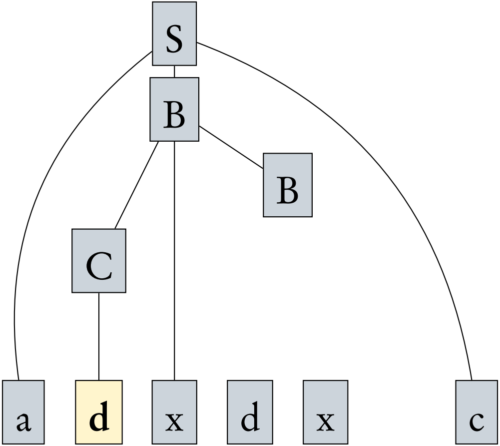
Input string:
"adxdxc"



Top Down Parsing

$$
\begin{array}{lcl}
S & \rightarrow & a \; B \; c \\
B & \rightarrow & C \; x \; B \\
B & \rightarrow & \epsilon \\
C & \rightarrow & d \\
  & | & a \; B \; c
\end{array}
$$

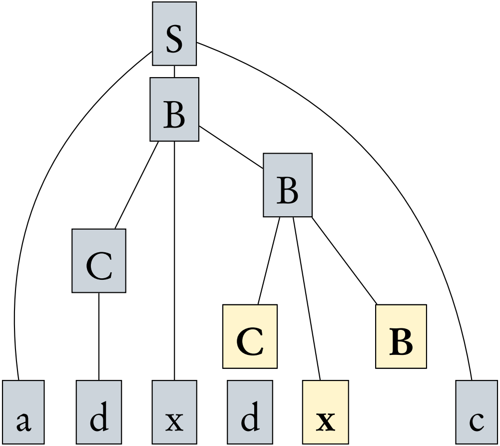
Input string:
"adxdxc"



Top Down Parsing

$$
\begin{array}{llll}
S & \rightarrow & a \; B \; c \\
B & \rightarrow & C \; x \; B \\
B & \rightarrow & \epsilon \\
C & \rightarrow & d \\
  & \mid & a \; B \; c
\end{array}
$$

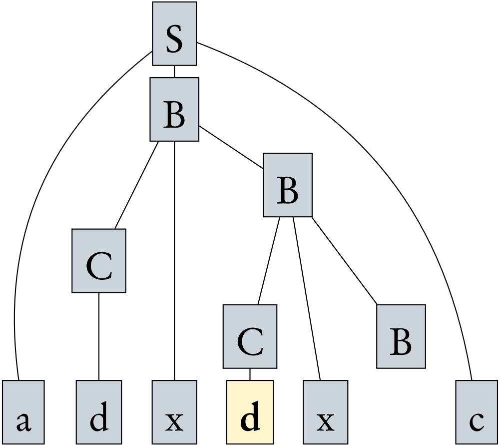
Input string:
“adxdxc”



Top Down Parsing

$$
\begin{array}{llll}
S & \rightarrow & a \; B \; c \\
B & \rightarrow & C \; x \; B \\
B & \rightarrow & \epsilon \\
C & \rightarrow & d \\
 & | & a \; B \; c
\end{array}
$$

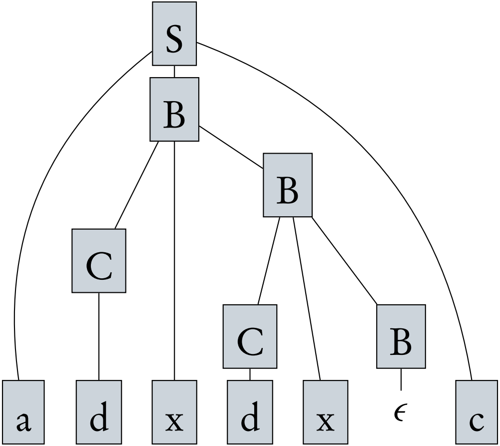
Input string:
“adxdxc”



Top Down Parsing

$$
\begin{array}{lclll}
S & \rightarrow & a & B & c \\
B & \rightarrow & C & x & B \\
B & \rightarrow & \epsilon & & \\
C & \rightarrow & d & & \\
  & | & a & B & c
\end{array}
$$

Input string:
"adxdxc"



Top Down Parsing (2)

$$
\begin{aligned}
E &\rightarrow T + E \mid T \\
T &\rightarrow (E) \mid \texttt{int} \mid \texttt{int} * T
\end{aligned}
$$

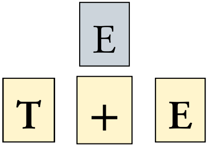
Input: int * int

$$\boxed{E}$$

Top Down Parsing (2)

$$
\begin{aligned}
E &\rightarrow T + E \mid T \\
T &\rightarrow (E) \mid \mathtt{int} \mid \mathtt{int} * T
\end{aligned}
$$

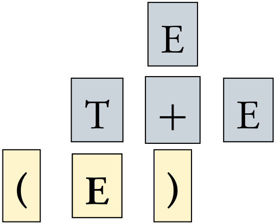
Input: int * int



Top Down Parsing (2)

$$
\begin{aligned}
E &\rightarrow T + E \mid T \\
T &\rightarrow (E) \mid \mathrm{int} \mid \mathrm{int} * T
\end{aligned}
$$

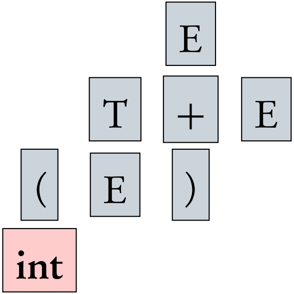
Input: int * int



Top Down Parsing (2)

$$
\begin{aligned}
E &\rightarrow T + E \mid T \\
T &\rightarrow (E) \mid \texttt{int} \mid \texttt{int} * T
\end{aligned}
$$

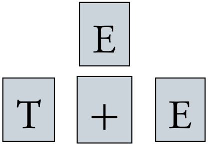
Input: int * int



Top Down Parsing (2)

$$
\begin{aligned}
E &\rightarrow T + E \mid T \\
T &\rightarrow (E) \mid \texttt{int} \mid \texttt{int} * T
\end{aligned}
$$

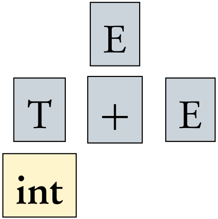
Input: int * int



Top Down Parsing (2)

$$
\begin{aligned}
E &\rightarrow T + E \mid T \\
T &\rightarrow (E) \mid \mathtt{int} \mid \mathtt{int} * T
\end{aligned}
$$

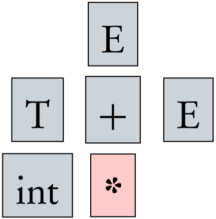
Input: int * int



Top Down Parsing (2)

$$
\begin{aligned}
E &\rightarrow T + E \mid T \\
T &\rightarrow (E) \mid \texttt{int} \mid \texttt{int} * T
\end{aligned}
$$

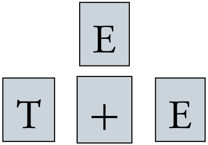
Input: int * int



Top Down Parsing (2)

$$
\begin{aligned}
E &\rightarrow T + E \mid T \\
T &\rightarrow (\, E \,) \mid \texttt{int} \mid \texttt{int} * T
\end{aligned}
$$

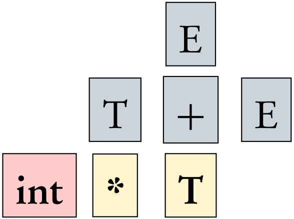
Input: int * int



Top Down Parsing (2)

$$
\begin{aligned}
E &\rightarrow T + E \mid T \\
T &\rightarrow (\; E \;) \mid \texttt{int} \mid \texttt{int} * T
\end{aligned}
$$

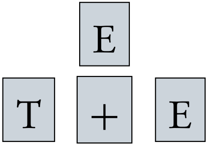
Input: int * int



Top Down Parsing (2)

$$E \rightarrow T + E \mid T$$
$$T \rightarrow (E) \mid \mathtt{int} \mid \mathtt{int} * T$$

Input: int * int



Top Down Parsing (2)

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

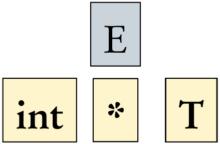
Input: int * int

$\boxed{E}$

Top Down Parsing (2)

$$
\begin{aligned}
E &\rightarrow T + E \mid T \\
T &\rightarrow (E) \mid \texttt{int} \mid \texttt{int} * T
\end{aligned}
$$

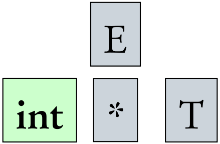
Input: int * int



Top Down Parsing (2)

$$
\begin{aligned}
E &\rightarrow T + E \mid T \\
T &\rightarrow (E) \mid \texttt{int} \mid \texttt{int} * T
\end{aligned}
$$

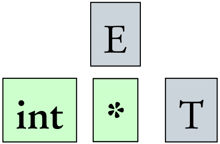
Input: int * int



Top Down Parsing (2)

$$
\begin{array}{lcl}
E & \rightarrow & T + E \mid T \\
T & \rightarrow & (\; E \;) \mid \texttt{int} \mid \texttt{int} * T
\end{array}
$$

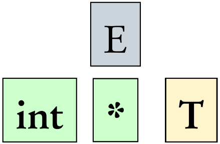
Input: int * int



Top Down Parsing (2)

$$
\begin{aligned}
E &\rightarrow T + E \mid T \\
T &\rightarrow (\; E \;) \mid \texttt{int} \mid \texttt{int} * T
\end{aligned}
$$

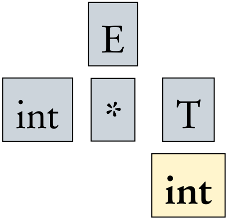
Input: int * int



Top Down Parsing (2)

$$
\begin{aligned}
E &\rightarrow T + E \mid T \\
T &\rightarrow (E) \mid \mathrm{int} \mid \mathrm{int} * T
\end{aligned}
$$

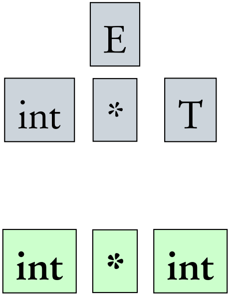
Input: int * int



Top Down Parsing (2)

$$
\begin{array}{lcl}
E & \rightarrow & T + E \mid T \\
T & \rightarrow & (E) \mid \texttt{int} \mid \texttt{int} * T
\end{array}
$$

Input: int * int



Top-Down Summary

- Exhaust all grammatical rule options at each step
- **Notice significant backtracing!**
  - Backtracing results from having multiple options for rules when considering tokens!
- If we restrict our grammar slightly, we can use LL(1) parsing
  - Key idea: *deterministically* select grammar rule based on one token to eliminate backtracing

LL(1) Summary

- We want a parsing strategy that lets us **parse an input string** by considering only **one token at a time**
  - Intuition: If input string $x \in L(G)$, there must be a parse tree. We want to pick a grammar rule one token at a time so we can quickly find if the string is in the language!

- Implementation: **Parse table**
  - Intuition: 2-d table indexed by ‘current’ non-terminal and ‘lookahead’ token. Each cell contains the production rule corresponding to the lookahead.

Parse Table Construction

- Constructing the Parse table requires a **PREDICT** set of every non-terminal in the grammar.
- *We need a way to know which rule must be used to successfully construct the next part of the parse tree*

$$
\begin{array}{lcl}
S & \rightarrow & a \\
 & | & b
\end{array}
$$

Consider! You know exactly which rule to take based on the first thing you see as part of the S rule.

Parse Table Construction

$$
\begin{array}{lcl}
S & \rightarrow & a\ A\ a \\
  & | & b\ A \\
A & \rightarrow & c\ c\ A \\
  & | & b\ b\ A
\end{array}
$$

- Same idea! We always start with S. You can see that the *first* non-terminal in the S rules tells you unambiguously which rule is used!
- Similarly, after you consume one ‘a’, the next non-terminal you consider is A. You can decide which A rule to choose based on the *first* terminal in the A rules!

Parse Table Construction

$$
\begin{array}{lcll}
S & \rightarrow & A & a \\
A & \rightarrow & B & b \\
B & \rightarrow & c & B \\
 & | & a & A
\end{array}
$$

- A bit trickier, but you can still decide which rule to take with only one terminal

Parse Table Construction

- Sounds like we need to know the first terminal that can appear as a result of expanding a non-terminal!
- This is called the **FIRST** set
  FIRST(A) is the set of **terminals** that can appear first in the expansion of the non-terminal A
  - Note this means recursing through other nonterminals!
- $\text{FIRST}(X) = \{\text{b} | X \rightarrow^* \text{b}\alpha\} \cup \{\epsilon | X \rightarrow^* \epsilon\}$

Parse Table Construction

- That includes ϵ rules

$$
\begin{array}{lcl}
S & \rightarrow & a \ A \ a \\
A & \rightarrow & b \ A \\
  & | & \epsilon
\end{array}
$$

Parse Table Construction

- Recall we're trying to make a parse table
  - Almost enough to know FIRST set of all non-terminals. The token will tell us which rule to take next.
  - **However**, how do we know when the token should be an ϵ?

Parse Table Construction

$$
\begin{array}{lcl}
S & \rightarrow & a \; A \; B \; a \\
A & \rightarrow & b \; B \\
 & | & \epsilon \\
B & \rightarrow & c \; A \; d \\
 & | & \epsilon
\end{array}
$$

If my input string is aba, after consuming the 'a' token, how do we know that the next 'b' token results in $B \rightarrow \epsilon$?

Parse Table Construction: FOLLOW

- We also need to know the **terminals** that could **come after** the previous non-terminal

- $\text{FOLLOW}(X) = \{\text{b} | S \rightarrow^* \beta X \text{b} \omega\}$

Computing FOLLOW set

- Compute FIRST sets for all non-terminals
- Add $ to FOLLOW(S)
  - start symbol always ends with end-of-input
- For all productions $Y \rightarrow ...XA_1...A_n$
  - Add FIRST(A_i)-{ϵ} to FOLLOW(X). Stop if $\epsilon \notin FIRST(A_i)$.
  - Add FOLLOW(Y) to FOLLOW(X)

Example FOLLOW Set

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

FOLLOW("+") = { int, (}
FOLLOW("(") = { int, (}
FOLLOW(X) = { \$,) }
FOLLOW(Y) = { +,), \$ }

Back to Parsing Tables

- Recall: We want to build a LL(1) Parsing Table

For each production $A \rightarrow \alpha$ in G do:

- For each terminal $\mathbf{b} \in \mathbf{FIRST}(\alpha)$ do
  - $\mathrm{T[A][b]} = \alpha$
- If $\alpha \rightarrow^* \epsilon$, for each $\mathbf{b} \in \mathbf{FOLLOW}(\mathbf{A})$ do
  - $\mathrm{T[A][b]} = \alpha$

Parsing Table

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

Where do we put $Y \to *T$?

- Well, FIRST(*T) = {*}, thus column * of row Y gets *T

	int	*****	**+**	**(**	**)**	**$**
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		*T	ϵ		ϵ	ϵ

Parsing Table

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

Where do we put $Y \rightarrow \epsilon$?

- Well, FOLLOW(Y) = {$, +,)}, thus columns $, +, and) in row Y get $Y \rightarrow \epsilon$

	int	*****	**+**	**(**	**)**	**$**
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		*T	ϵ		ϵ	ϵ

Another Parse Table Example

$$\begin{array}{lcl} E & \to & T\ X \\ X & \to & +\ T\ X \\ & | & \epsilon \\ T & \to & F\ R \end{array} \qquad \begin{array}{lcl} R & \to & *\ F\ R \\ & | & \epsilon \\ F & \to & \mathtt{int} \\ & | & (\ E\ ) \end{array}$$

	+	*	(	)	int	$
E						
X						
T						
R						
F						

Production	Prediction
$E \to TX$	(, int
$X \to +TX$	+
$X \to \epsilon$	$,)
$T \to FR$	(, int)
$R \to *FR$	*
$R \to \epsilon$	+, $,)
$F \to \text{int}$	int
$F \to (E)$	(

Another Parse Table Example

	+	*	(	)	int	$
E			TX		TX	
X	$+TX$			ϵ		ϵ
T			FR		FR	
R	ϵ		$*FR$	ϵ		ϵ
F			(E)		int	

Production	Prediction
$E \rightarrow TX$	(, int
$X \rightarrow +TX$	+
$X \rightarrow \epsilon$	$,)
$T \rightarrow FR$	(, int)
$R \rightarrow *FR$	*
$R \rightarrow \epsilon$	+, $,)
$F \rightarrow \text{int}$	int
$F \rightarrow (E)$	(

Notes on LL(1) Parsing Tables

- If any entry is **multiply defined** then G is not LL(1)
  - G is ambiguous
  - G is left-recursive
  - G is not left-factored

Ambiguity in parse tables

$$\begin{array}{lllllll} E & \rightarrow & E + T & \quad & T & \rightarrow & F \\ E & \rightarrow & T & & F & \rightarrow & \text{id} \\ T & \rightarrow & T * F & & F & \rightarrow & (E) \end{array}$$

For the E productions, we need FIRST(T) = {(, id} and FIRST(E) = {(, id}

But now, which rule ($E \rightarrow E + T$ or $E \rightarrow T$) gets put in T[E][(] and T[E][id]??

	+	*	(	)	id	$
E			?		?	
T						
F						

LR Parsing

- LL(1) parsing is simple and fast
- However, what if we wanted more expressiveness in the grammar?
- Enter LR Parsing, a bottom-up parsing approach
  - Equally efficient as LL(1)
  - Not quite as easy by hand
  - Preferred practical method (see bison/yacc)

LR Parsing

An **LR Parser** reads tokens from *left to right* and constructs a *bottom-up rightmost* derivation. LR parsers **shift** terminals and **reduce** input by application of productions in **reverse**. LR parsing is fast and easy, and uses a finite automaton (a.k.a. a parse table) augmented with a **stack**. LR works fine with grammars that are left-recursive or not left-factored.

LR Parsing

- LR parsers do not require left-factored grammars and can also handle left-recursive grammars

Consider

$$E \rightarrow E + (E) | \text{int}$$

Can you see why this is **not** LL(1)?

LR Parsing

- Consider input string x
- Loop
  - Identify β in x such that $A \to \beta$ is a production (i.e., $x = \alpha\beta\gamma$)
  - Replace β by A in x (i.e., x becomes $\alpha A\gamma$)
- until $x = S$

LR Parsing Example

$$
\begin{aligned}
S &\rightarrow a \; T \; R \; e \\
T &\rightarrow T \; b \; c \mid b \\
R &\rightarrow d
\end{aligned}
$$

Rightmost derivation:

$$
\begin{aligned}
S &\rightarrow a \; T \; R \; e \\
&\rightarrow a \; T \; d \; e \\
&\rightarrow a \; T \; b \; c \; d \; e \\
&\rightarrow a \; b \; b \; c \; d \; e
\end{aligned}
$$

LR Parsing Example

$$
\begin{aligned}
S &\rightarrow a\ T\ R\ e \\
T &\rightarrow T\ b\ c \mid b \\
R &\rightarrow d
\end{aligned}
$$

Rightmost derivation:

$$
\begin{aligned}
S &\rightarrow a\ T\ R\ e \\
&\rightarrow a\ T\ d\ e \\
&\rightarrow a\ \boxed{T}\ b\ c\ d\ e \\
&\rightarrow a\ \boxed{b}\ b\ c\ d\ e
\end{aligned}
$$

LR Parsing Example

$$
\begin{aligned}
S &\rightarrow a \; T \; R \; e \\
T &\rightarrow T \; b \; c \;\mid\; b \\
R &\rightarrow d
\end{aligned}
$$

Rightmost derivation:

$$
\begin{aligned}
S &\rightarrow a \; T \; R \; e \\
&\rightarrow a \; \boxed{T} \; d \; e \\
&\rightarrow a \; \boxed{T \; b \; c} \; d \; e \\
&\rightarrow a \; b \; b \; c \; d \; e
\end{aligned}
$$

LR Parsing Example

$$
\begin{aligned}
S &\rightarrow a\ T\ R\ e \\
T &\rightarrow T\ b\ c \mid b \\
R &\rightarrow d
\end{aligned}
$$

Rightmost derivation:

$$
\begin{aligned}
S &\rightarrow a\ T\ \boxed{R}\ e \\
&\rightarrow a\ T\ \boxed{d}\ e \\
&\rightarrow a\ T\ b\ c\ d\ e \\
&\rightarrow a\ b\ b\ c\ d\ e
\end{aligned}
$$

LR Parsing Example

Recall $E \rightarrow E + (E) | \text{int}$

Input = int + (int) + (int)

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

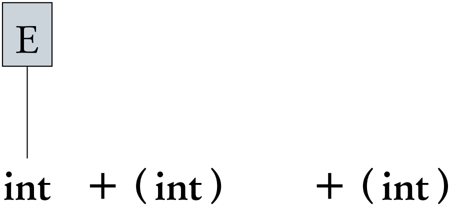
int + (int) + (int)

LR Parsing Example

Recall $E \rightarrow E + (E) | \text{int}$

Input $= \text{int} + (\text{int}) + (\text{int})$

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

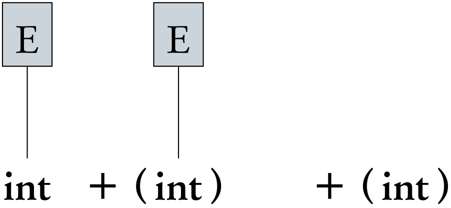


LR Parsing Example

Recall $E \rightarrow E + (E) | \text{int}$

Input $=$ int $+$ (int) $+$ (int)

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

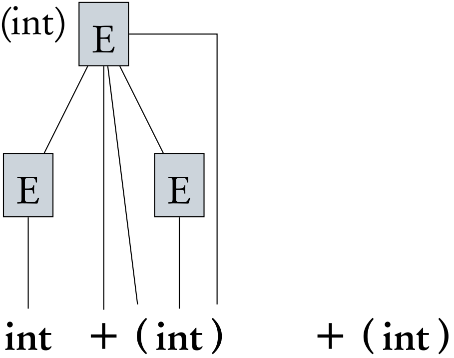


LR Parsing Example

Recall $E \rightarrow E + (E) | \text{int}$

Input = int + (int) + (int)

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

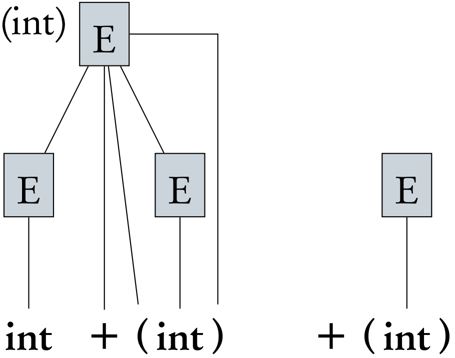


LR Parsing Example

Recall $E \rightarrow E + (E) | \text{int}$

Input = int + (int) + (int)

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

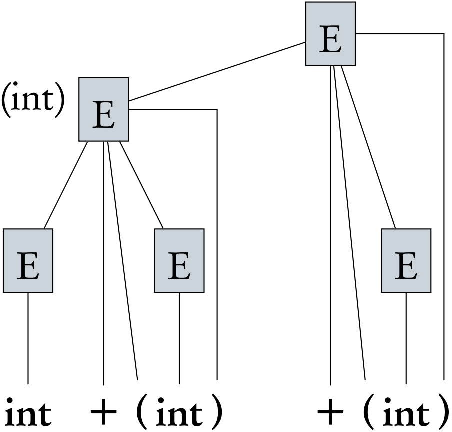


LR Parsing Example

Recall $E \rightarrow E + (E) | \text{int}$

Input = int + (int) + (int)

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



LR Parsing

Important fact about LR parsing:
An LR parser traces a rightmost derivation in reverse.

Notation

Idea: Split the string into two substrings

- **Right substring** (a string of **terminals**) is as yet unexamined by the parser
- **Left substring** has terminals and non-terminals

The dividing point is marked by a $\bullet$

Initially, all input is new: $\bullet\ x_1 x_2 x_3 \ldots x_n$

LR = shift-reduce Parsing

Bottom-up parsing uses only two actions:

1. **Shift**
2. **Reduce**

Shift

Shift: Move $\bullet$ one place to the right

- Shifts a terminal to the left string

$$E + (\bullet \ \text{int} \ )$$

$$\Rightarrow$$

$$E + (\ \text{int} \bullet)$$

Reduce

Apply an **inverse** production at the **right end** of the **left string**
If $T \to E + (E)$ is a production, then

$$E + (E + (E) \bullet)$$

$$E + (T \bullet)$$

Reductions can only happen here!

Shift-Reduce Example

- int + (int) + (int)$ shift

Shift-Reduce Example

$\bullet$ int + (int) + (int)\$	shift
int $\bullet$ + (int) + (int)\$	reduce $E \rightarrow$ int

Shift-Reduce Example

• int + (int) + (int)\$	shift
int • + (int) + (int)\$	reduce $E \rightarrow$ int
E • + (int) + (int)\$	shift (3 times)

Shift-Reduce Example

$\bullet$ int + (int) + (int)\$	shift
int $\bullet$ + (int) + (int)\$	reduce $E \rightarrow \text{int}$
E $\bullet$ + (int) + (int)\$	shift (3 times)
E + (int $\bullet$) + (int)\$	reduce $E \rightarrow \text{int}$

Shift-Reduce Example

$\bullet$ int + (int) + (int)\$	shift
int $\bullet$ + (int) + (int)\$	reduce $E \to$ int
E $\bullet$ + (int) + (int)\$	shift (3 times)
E + (int $\bullet$) + (int)\$	reduce $E \to$ int
E + (E $\bullet$) + (int)\$	shift

Shift-Reduce Example

$\bullet$ int + (int) + (int)\$	shift
int $\bullet$ + (int) + (int)\$	reduce $E \to \text{int}$
E $\bullet$ + (int) + (int)\$	shift (3 times)
E + (int $\bullet$) + (int)\$	reduce $E \to \text{int}$
E + (E $\bullet$) + (int)\$	shift
E + (E) $\bullet$ + (int)\$	reduce $E \to E + (E)$

Shift-Reduce Example

$\bullet$ int + (int) + (int)\$	shift
int $\bullet$ + (int) + (int)\$	reduce $E \rightarrow \text{int}$
E $\bullet$ + (int) + (int)\$	shift (3 times)
E + (int $\bullet$) + (int)\$	reduce $E \rightarrow \text{int}$
E + (E $\bullet$) + (int)\$	shift
E + (E) $\bullet$ + (int)\$	reduce $E \rightarrow E + (E)$
E $\bullet$ + (int)\$	shift (3 times)

Shift-Reduce Example

• int + (int) + (int)$	shift
int • + (int) + (int)$	reduce $E \rightarrow$ int
E • + (int) + (int)$	shift (3 times)
E + (int •) + (int)$	reduce $E \rightarrow$ int
E + (E •) + (int)$	shift
E + (E) • + (int)$	reduce $E \rightarrow E + (E)$
E • + (int)$	shift (3 times)
E + (int •)$	reduce $E \rightarrow$ int

Shift-Reduce Example

$\bullet$ int + (int) + (int)\$	shift
int $\bullet$ + (int) + (int)\$	reduce $E \rightarrow$ int
E $\bullet$ + (int) + (int)\$	shift (3 times)
E + (int $\bullet$) + (int)\$	reduce $E \rightarrow$ int
E + (E $\bullet$) + (int)\$	shift
E + (E) $\bullet$ + (int)\$	reduce $E \rightarrow E + (E)$
E $\bullet$ + (int)\$	shift (3 times)
E + (int $\bullet$)\$	reduce $E \rightarrow$ int
E + (E $\bullet$)\$	shift

Shift-Reduce Example

$\bullet$ int + (int) + (int)\$	shift
int $\bullet$ + (int) + (int)\$	reduce $E \rightarrow$ int
E $\bullet$ + (int) + (int)\$	shift (3 times)
E + (int $\bullet$) + (int)\$	reduce $E \rightarrow$ int
E + (E $\bullet$) + (int)\$	shift
E + (E) $\bullet$ + (int)\$	reduce $E \rightarrow E + (E)$
E $\bullet$ + (int)\$	shift (3 times)
E + (int $\bullet$)\$	reduce $E \rightarrow$ int
E + (E $\bullet$)\$	shift
E + (E) $\bullet$ \$	reduce $E \rightarrow E + (E)$

Shift-Reduce Example

$\bullet$ int + (int) + (int)\$	shift
int $\bullet$ + (int) + (int)\$	reduce $E \rightarrow$ int
E $\bullet$ + (int) + (int)\$	shift (3 times)
E + (int $\bullet$) + (int)\$	reduce $E \rightarrow$ int
E + (E $\bullet$) + (int)\$	shift
E + (E) $\bullet$ + (int)\$	reduce $E \rightarrow E + (E)$
E $\bullet$ + (int)\$	shift (3 times)
E + (int $\bullet$)\$	reduce $E \rightarrow$ int
E + (E $\bullet$)\$	shift
E + (E) $\bullet$ \$	reduce $E \rightarrow E + (E)$
E $\bullet$ \$	**accept**

Stack

The **left string** can be implemented with **a stack**

- Top of the stack is •

Shift

- pushes a **terminal** onto the stack

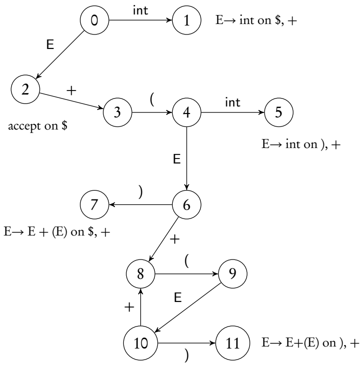
Reduce

- **pop** 0 or more **symbols** from the stack (production RHS)
- **push a non-terminal** on the stack (production LHS)

When to Shift/Reduce?

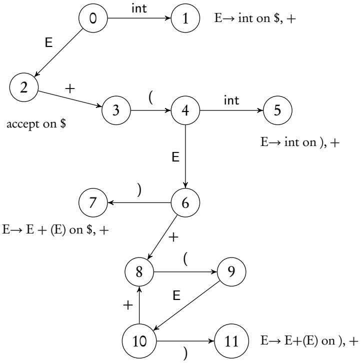
- Decide based on the **left string** (stack)
- Idea: Use a DFA to decide when to shift or reduce
  - The **DFA input** is the **stack**
  - DFA language consists of terminals and non-terminals
- We run the DFA on the stack and we examine the resulting state X and the token **t** after •
  - If X has a transition labeled **t**, then **shift**
  - If X is labeled with "$A \rightarrow \beta$ on **t**", then **reduce**

LR(1) Parsing Example



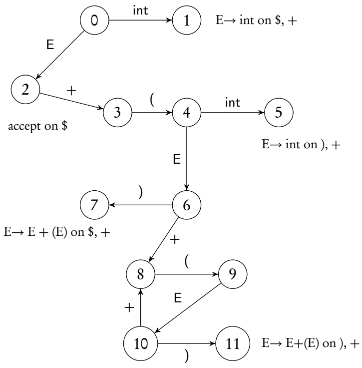
- int + (int) + (int)$ shift

LR(1) Parsing Example



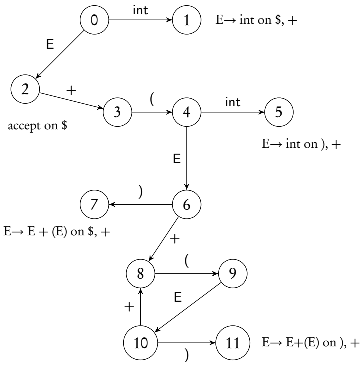
• int + (int) + (int)$	shift
int • + (int) + (int)$	reduce $E \rightarrow$ int

LR(1) Parsing Example



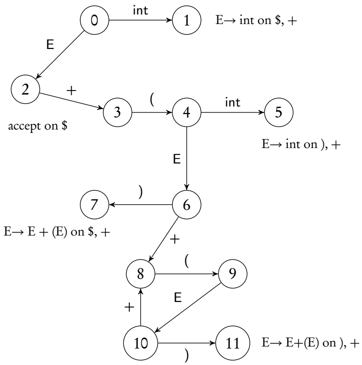
•	int + (int) + (int)$	shift
int •	+ (int) + (int)$	reduce $E \rightarrow \text{int}$
E •	+ (int) + (int)$	shift (3 times)

LR(1) Parsing Example



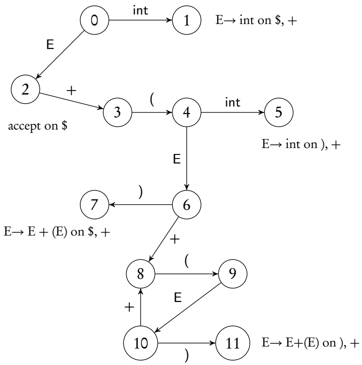
• int + (int) + (int)$	shift
int • + (int) + (int)$	reduce $E \to$ int
E • + (int) + (int)$	shift (3 times)
E + (int •) + (int)$	reduce $E \to$ int

LR(1) Parsing Example



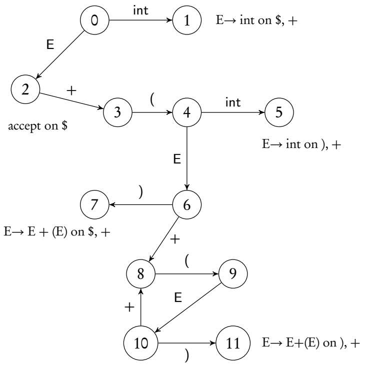
• int + (int) + (int)$	shift
int • + (int) + (int)$	reduce $E \rightarrow$ int
E • + (int) + (int)$	shift (3 times)
E + (int •) + (int)$	reduce $E \rightarrow$ int
E + (E •) + (int)$	shift

LR(1) Parsing Example



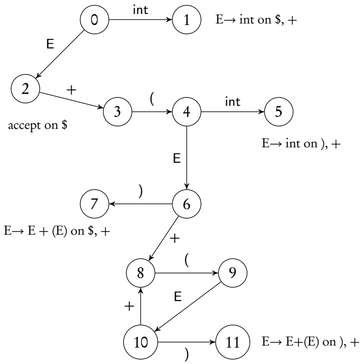
• int + (int) + (int)$	shift
int • + (int) + (int)$	reduce $E \to$ int
E • + (int) + (int)$	shift (3 times)
E + (int •) + (int)$	reduce $E \to$ int
E + (E •) + (int)$	shift
E + (E) • + (int)$	reduce $E \to E + (E)$

LR(1) Parsing Example



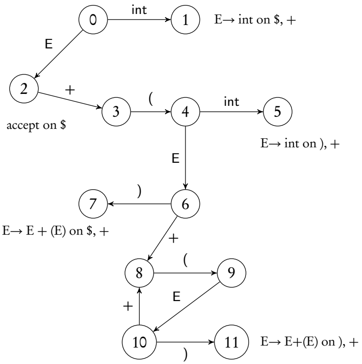
$\bullet$ int + (int) + (int)$	shift
int $\bullet$ + (int) + (int)$	reduce $E \rightarrow \text{int}$
E $\bullet$ + (int) + (int)$	shift (3 times)
E + (int $\bullet$) + (int)$	reduce $E \rightarrow \text{int}$
E + (E $\bullet$) + (int)$	shift
E + (E) $\bullet$ + (int)$	reduce $E \rightarrow E + (E)$
E $\bullet$ + (int)$	shift (3 times)

LR(1) Parsing Example



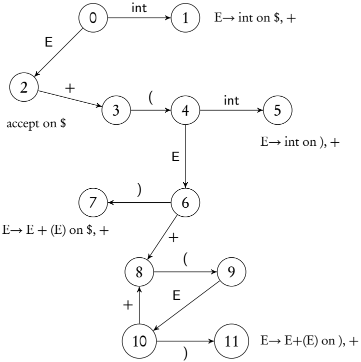
$\bullet$ int + (int) + (int)$	shift
int $\bullet$ + (int) + (int)$	reduce $E \to \text{int}$
E $\bullet$ + (int) + (int)$	shift (3 times)
E + (int $\bullet$) + (int)$	reduce $E \to \text{int}$
E + (E $\bullet$) + (int)$	shift
E + (E) $\bullet$ + (int)$	reduce $E \to E + (E)$
E $\bullet$ + (int)$	shift (3 times)
E + (int $\bullet$)$	reduce $E \to \text{int}$

LR(1) Parsing Example



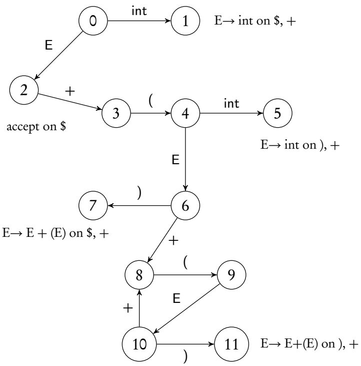
$\bullet$ int + (int) + (int)$	shift
int $\bullet$ + (int) + (int)$	reduce $E \to \text{int}$
E $\bullet$ + (int) + (int)$	shift (3 times)
E + (int $\bullet$) + (int)$	reduce $E \to \text{int}$
E + (E $\bullet$) + (int)$	shift
E + (E) $\bullet$ + (int)$	reduce $E \to E + (E)$
E $\bullet$ + (int)$	shift (3 times)
E + (int $\bullet$)$	reduce $E \to \text{int}$
E + (E $\bullet$)$	shift

LR(1) Parsing Example



• int + (int) + (int)$	shift
int • + (int) + (int)$	reduce $E \rightarrow$ int
E • + (int) + (int)$	shift (3 times)
E + (int •) + (int)$	reduce $E \rightarrow$ int
E + (E •) + (int)$	shift
E + (E) • + (int)$	reduce $E \rightarrow E + (E)$
E • + (int)$	shift (3 times)
E + (int •)$	reduce $E \rightarrow$ int
E + (E •)$	shift
E + (E) • $	reduce $E \rightarrow E + (E)$

LR(1) Parsing Example



• int + (int) + (int)$	shift
int • + (int) + (int)$	reduce $E \rightarrow$ int
E • + (int) + (int)$	shift (3 times)
E + (int •) + (int)$	reduce $E \rightarrow$ int
E + (E •) + (int)$	shift
E + (E) • + (int)$	reduce $E \rightarrow E + (E)$
E • + (int)$	shift (3 times)
E + (int •)$	reduce $E \rightarrow$ int
E + (E •)$	shift
E + (E) • $	reduce $E \rightarrow E + (E)$
E • $	**accept**

Parsing Tables

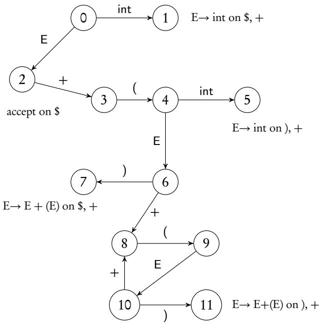
That's right! Represent that DFA as a table

- Parsers represent the DFA as a 2D table
- Rows correspond to DFA states
- Columns correspond to terminals/non-terminals
- Cells contain **actions**
  - Terminals shift or reduce
  - Non-temrinals goto subsequent DFA states

- **Critically** Restart DFA with stack as input *every time* you reduce

Parsing Tables

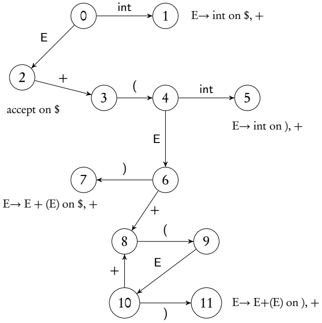
	int	+	(	)	$	E
0	s1					g2
1		$r_{E\to int}$			$r_{E\to int}$	
2		s3			**Accept**	
3		s4				
4	s5					g6
5		$r_{E\to int}$		$r_{E\to int}$		
6	s8		s7			
7		$r_{E\to E+(E)}$		$r_{E\to E+(E)}$		
...						



Parsing Tables

	int	+	(	)	$	E
0	s1					g2
1		$r_{E\to int}$			$r_{E\to int}$	
2		s3			**Accept**	
3		s4				
4	s5					g6
5		$r_{E\to int}$		$r_{E\to int}$		
6	s8		s7			
7		$r_{E\to E+(E)}$		$r_{E\to E+(E)}$		
...						

• int + (int) + (int)$	shift
int • + (int) + (int)$	reduce $E \to$ int
E • + (int) + (int)$	shift (3 times)
E + (int •) + (int)$	reduce $E \to$ int
E + (E •) + (int)$	shift
E + (E) • + (int)$	reduce $E \to E + (E)$
E • + (int)$	shift (3 times)
E + (int •)$	reduce $E \to$ int
E + (E •)$	shift
E + (E) • $	reduce $E \to E + (E)$
E • $	**accept**



Next time

- **Parser generators** construct the parsing DFA/table for you
- For PA3, you use a yacc/bison-based parser generator to implement the COOL grammar!
  - What about the *abstract* syntax *tree*