



NTNU

| Norwegian University of
Science and Technology

Compiler Construction

Lecture 7: Bottom-up parsing

Michael Engel

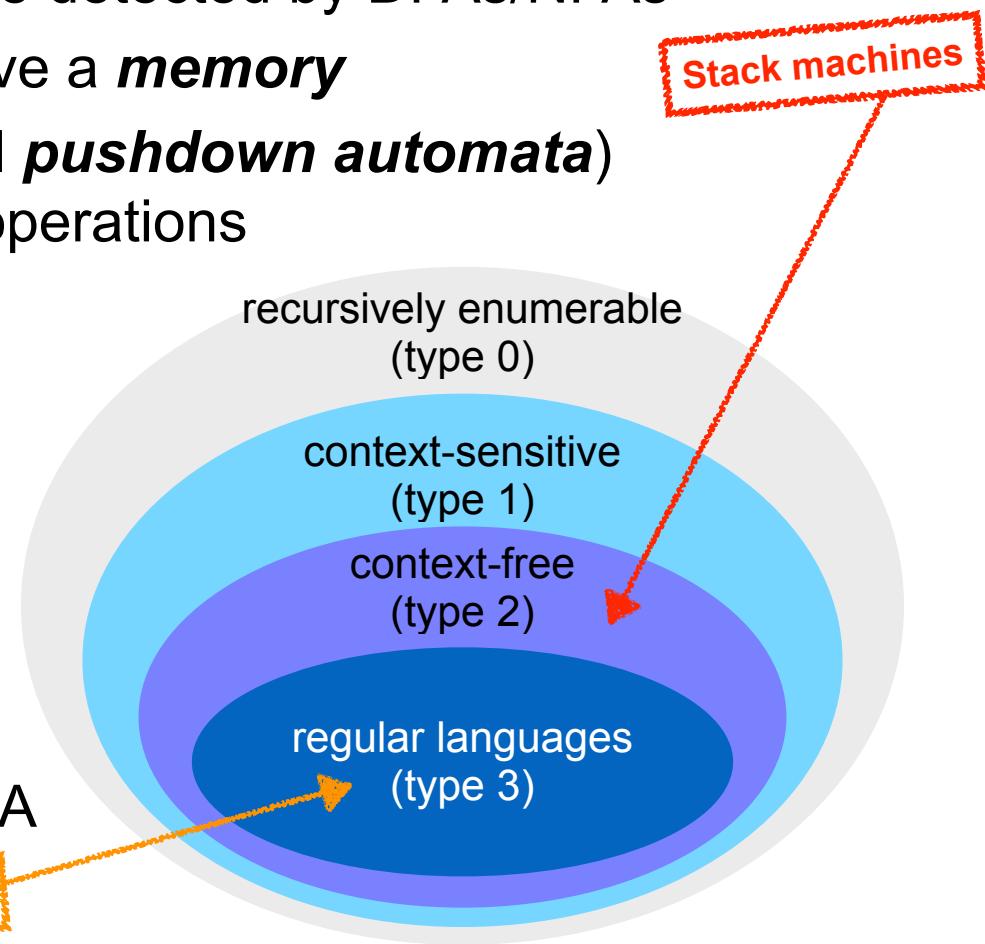
Includes material by
Jan Christian Meyer
and Rich Maclin (UNM)

Overview

- Top-down parsing revisited
- Bottom-up parsing
 - Comparison to top-down parsing
 - Shift-reduce parsers
 - Conflict resolution

Types of languages and automata

- Context-free languages are a *superset* of regular languages
 - Regular languages can be detected by DFAs/NFAs
 - DFAs and NFAs don't have a *memory*
- **Stack machines** (also called *pushdown automata*) add memory by introducing operations ***push*** and ***pop***
 - They enable the stack machine to memorize (trace) the path they took to get to a state (and revert to a previous one)
 - More powerful than D/NFA



Top-down parsing and the stack

- We've seen LL(1) tables and manually built recursive descent parsers
- Another simple example:

```
void parse_A() {
    switch (sym) {
        case 'x':
            add_tree(x,B);
            match(x);
            parse_B();
            break;
        case 'y':
            add_tree(y,C);
            match(y);
            parse_C();
            break;
        case EOF:
            error();
            break;
    }
    return;
}
```

$A \rightarrow xB$	yC
$B \rightarrow xB$	ϵ
$C \rightarrow yC$	ϵ

	x	y	EOF
A	$A \rightarrow xB$	$A \rightarrow yC$	
B	$B \rightarrow xB$		$B \rightarrow \epsilon$
C		$C \rightarrow yC$	$C \rightarrow \epsilon$

```
void parse_B() {
    switch (sym):
        case 'x':
            add_tree(x,B);
            match(x);
            parse_B();
            break;
        case 'y':
            error(); break;
        case EOF:
            return;
    }
    return;
}
```

```
void parse_C() {
    switch (sym):
        case 'x':
            error(); break;
        case 'y':
            add_tree(y,C);
            match(y);
            parse_C();
            break;
        case EOF:
            return;
    }
    return;
}
```

Tracing the recursive descent code

Syntax analysis

- Which derivation do we get when parsing "yyy"?
- $$A \rightarrow yC \rightarrow yyC \rightarrow yyyC \rightarrow yyy$$
- What is the related *hierarchy of function calls*?

$A \rightarrow xB \mid yC$	$B \rightarrow xB \mid \epsilon$	$C \rightarrow yC \mid \epsilon$
Call		

Recur:

	Call		Call		match(y)	Return	Call	Call	match(y)
Call	match(y)	Return	parse_C	parse_C	parse_C	parse_C	parse_C	parse_C	parse_C
Call	parse_A	parse_A	parse_A	parse_A	parse_A	parse_A	parse_A	parse_A	parse_A
Call	parse_A	parse_A	parse_A	parse_A	parse_A	parse_A	parse_A	parse_A	parse_A



Unwind:

...	match(y)	Return							
	parse_C	parse_C	Return						
	parse_C	parse_C	parse_C	Return					
	parse_C	parse_C	parse_C	parse_C	parse_C	Return			
	parse_A	parse_A	parse_A	parse_A	parse_A	parse_A	parse_A	parse_A	Finished



Memory in recursive descent code

- Where is the memory hidden in our parser?
 - We do not explicitly store and retrieve state
- The programming language hides it:
 - When calling (returning) from a function, **state is pushed onto (popped from)** the computer's stack automatically
 - This state includes the **return address** of the call site
- We can also build LL(1) parsers using iterations
 - but then we have to implement our own stack...
- The stack is needed to match beginnings and ends of productions
- Any production of the form $A \rightarrow xBy$ where B can contain further instances of x and y , such as:

Expression \rightarrow (Expression)

Statement \rightarrow {Statement}

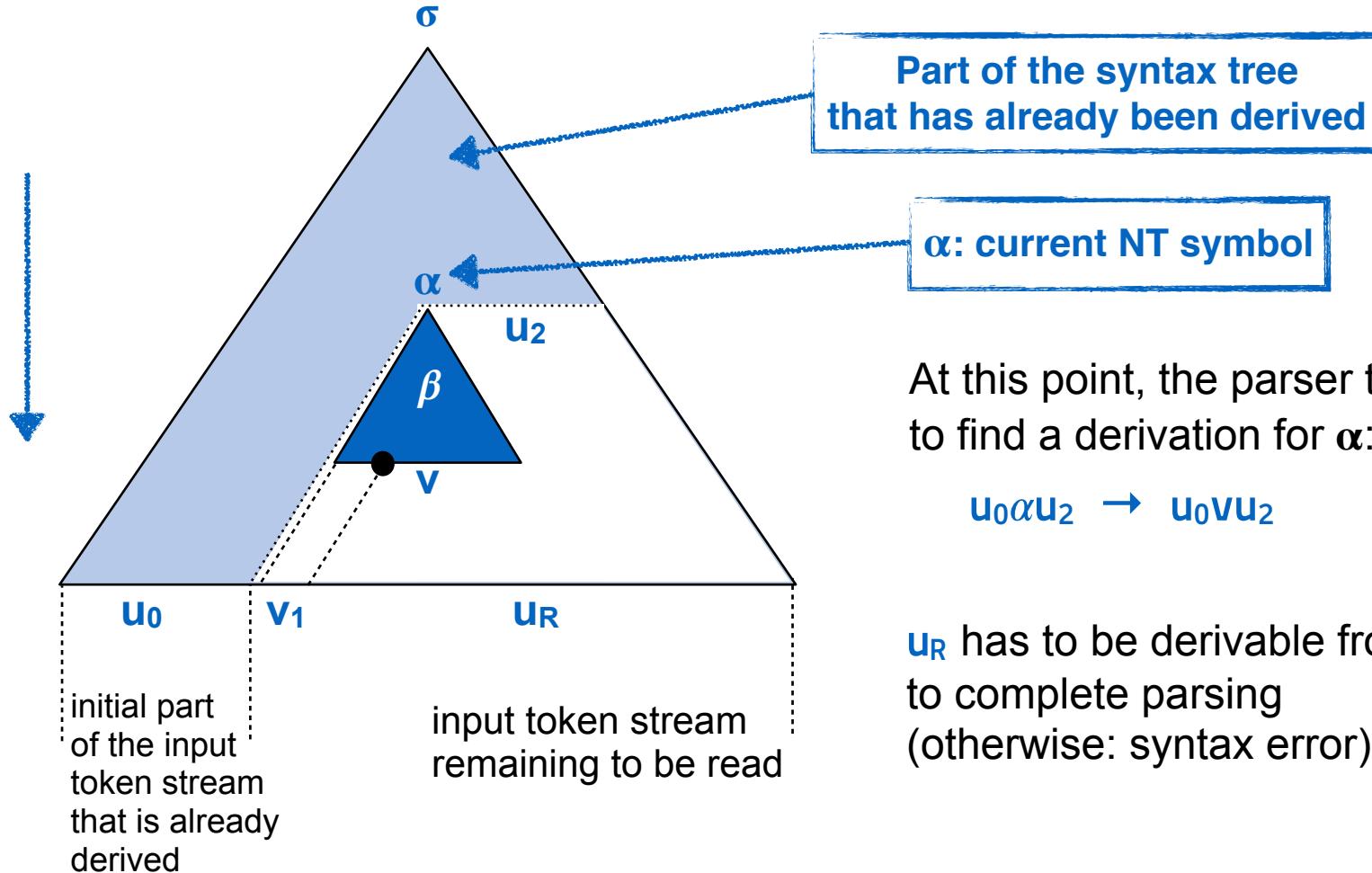
Comment \rightarrow (Comment *)*

	Call	Call	Call	Call	Call	Return
	parse_A	parse_A	parse_A	parse_A	parse_A	parse_A
	parse_A	match(y)	Return	parse_C	parse_C	parse_C
	Call	Call	Call	match(y)	parse_C	parse_C

Top-down parsing and the syntax tree

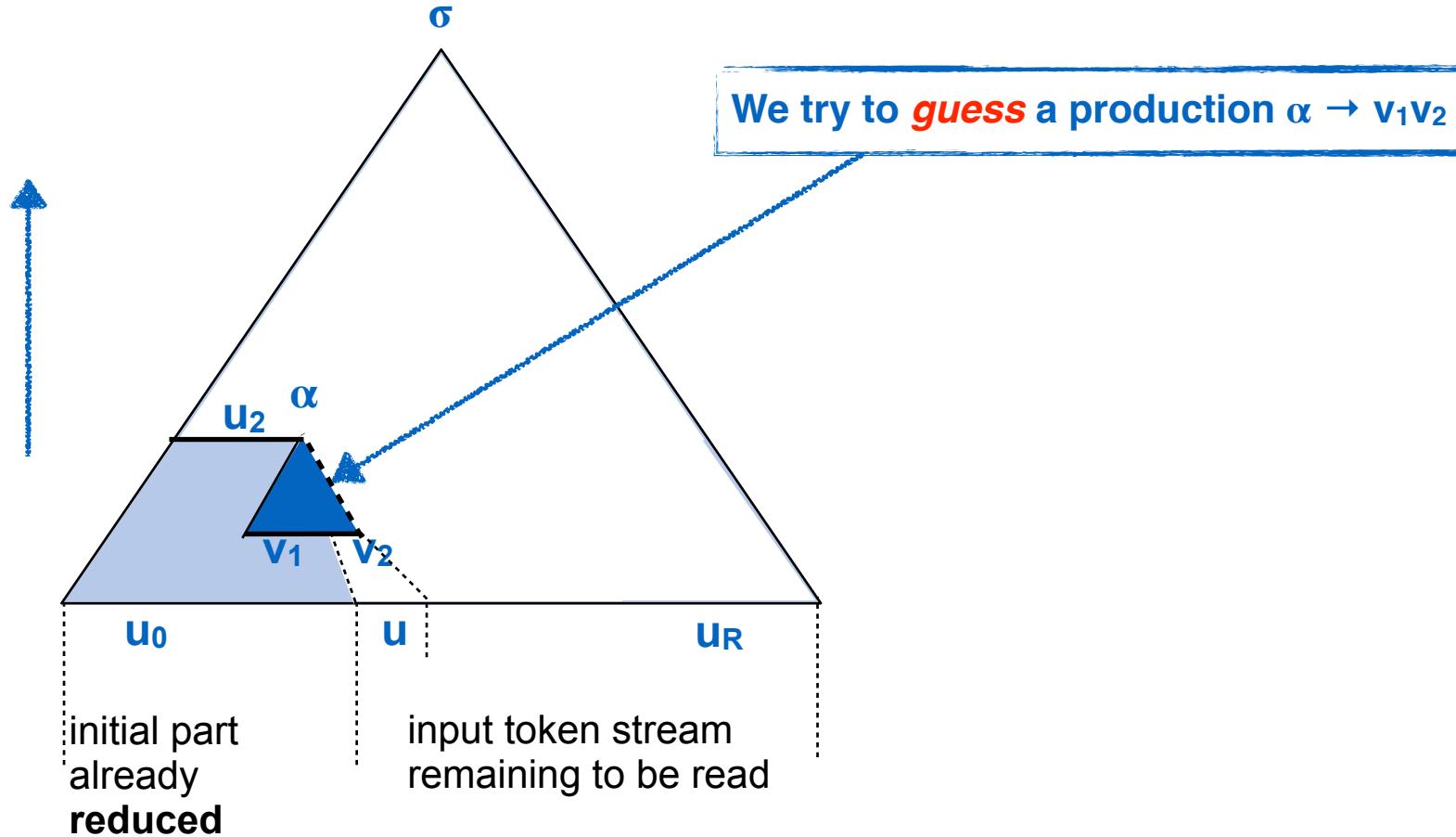
Syntax analysis

LL(1) parsers generate a parse tree from top to bottom:



Bottom-up parsing

Can we also construct the parse tree from bottom to top?

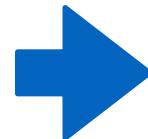


General idea of bottom-up parsing

Syntax analysis

- Bottom-up parsing starts from the input token stream (whereas top-down starts from the grammar start symbol)
- It **reduces** a string to the start symbol by **inverting productions**
 - trying to find a production matching the **right hand side**

$$\begin{array}{l} E \rightarrow T + E \mid T \\ T \rightarrow \text{int} \times T \mid \text{int} \mid \epsilon \end{array}$$



$$\begin{array}{l} E \leftarrow T + E \mid T \\ T \leftarrow \text{int} \times T \mid \text{int} \mid \epsilon \end{array}$$

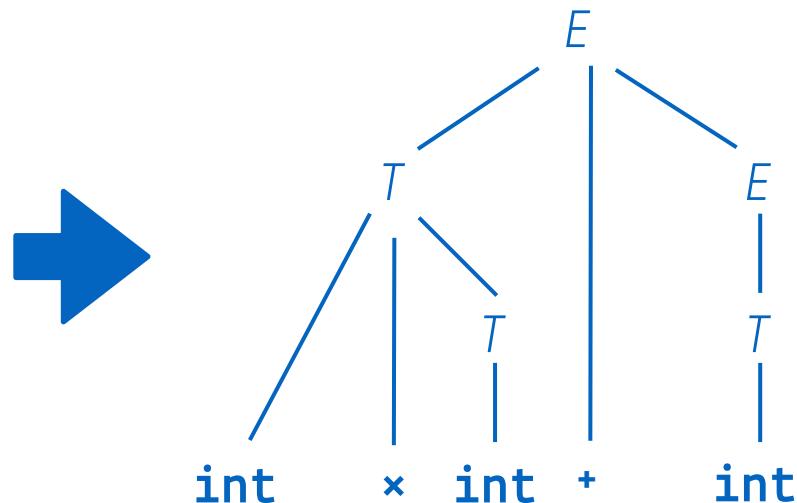
- Consider the input token stream **int * int + int**:
- Reading the productions in reverse (from **bottom** to **top**) gives a **rightmost derivation**

$$\begin{array}{ll} \text{int} \times \text{int} + \text{int} & T \rightarrow \text{int} \\ \text{int} \times T + \text{int} & T \rightarrow \text{int} \times T \\ T + \text{int} & T \rightarrow \text{int} \\ T + T & E \rightarrow T \\ T + E & E \rightarrow T + E \\ E & \end{array}$$

The resulting parse tree

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

```
int × int + int
int × T + int
T + int
T + T
T + E
E
```



A simple bottom-up parsing algor.

- **Idea:** split input string (token stream) into two substrings
 - Right substring (a string of terminal symbols) has not been examined so far
 - Left substring has terminals and nonterminals (generated by **replacing** the right side of a production by the left side)

```
I = input string

repeat
    select a non-empty substring  $\beta$  of I
        where  $X \rightarrow \beta$  is a production in the grammar
        if no such  $\beta$  exists, backtrack
        replace one  $\beta$  by  $X$  in I
until I == "S" /* start symbol */
or all other possibilities exhausted /* error */
```

Bottom-up parsing steps

- Initially, all input is unexamined, written as:

$\uparrow x_1 x_2 x_3 \dots x_n$

Two kinds of operations:

- Shift:** move \uparrow one place to the right

$ABC\uparrow xyz \rightarrow ABCx\uparrow yz$

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

$Cbxy\uparrow ijk \rightarrow CbA\uparrow ijk$

Example with reductions only

$$\begin{array}{l} E \rightarrow T + E \mid T \\ T \rightarrow \text{int} \times T \mid \text{int} \mid \epsilon \end{array}$$

int × int ↑ + int → reduce $T \rightarrow \text{int}$

int × T ↑ + int → reduce $T \rightarrow \text{int} \times T$

T + int ↑ → reduce $T \rightarrow \text{int}$

T + T ↑ → reduce $E \rightarrow T$

T + E ↑ → reduce $E \rightarrow T + E$

Example with shift-reduce parsing

Syntax analysis

↑ int × int + int

shift

int ↑ × int + int

shift

int × ↑ int + int

shift

int × int ↑ + int

reduce $T \rightarrow \text{int}$

int × T ↑ + int

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

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

(arrived at start symbol!)

$$\begin{array}{l} E \rightarrow T + E \mid T \\ T \rightarrow \text{int} \times T \mid \text{int} \mid \epsilon \end{array}$$

Implementing the memory

Idea:

- Left substring can be implemented by a **stack**
 - **shift** pushes a **terminal** symbol onto the stack
 - **reduce** pops zero or more symbols off the stack (the right-hand side of a production) and **pushes a non-terminal symbol** onto the stack (left-hand side of a production)

$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} \times T \mid \text{int} \mid \epsilon \end{aligned}$$

<u>stack contents</u>	<u>input token stream</u>	<u>parser operation: stack operation(s)</u>
[]	↑ int × int + int	shift: push [int]
[int]	int ↑ × int + int	shift: push [×]
[int, ×]	int × ↑ int + int	shift: push [int]
[int, ×, int]	int × int ↑ + int	reduce $T \rightarrow \text{int}$: pop->int, push[T]
[int, ×, T]	int × int ↑ + int	reduce $T \rightarrow \text{int} \times T$: pop, push[$T$]
[T]	int × int ↑ + int	...

Conflicts in parsing

Problem:

- How do we decide when to shift or reduce?
 - Consider the step $\text{int} \uparrow \times \text{int} + \text{int}$
 - We could reduce using $T \rightarrow \text{int}$ giving $T \uparrow \times \text{int} + \text{int}$
 - A fatal mistake: **No way to reduce to the start symbol E**
- Generic shift-reduce strategy:
 - If there is a matching pattern (**handle**) on the stack, reduce
 - Otherwise, shift
- What if there is a choice (between two matching patterns)?
 - If it's 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 and example

Conflicts arise due to:

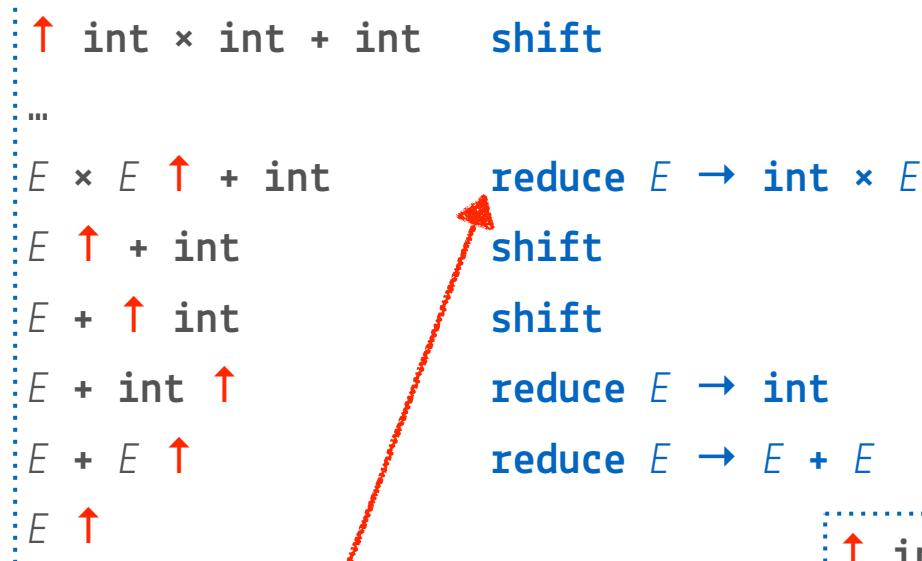
- Ambiguous grammars: always cause conflicts
- But beware, so do many non-ambiguous grammars
- Conflict example

Grammar

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

$\uparrow \text{ int } \times \text{ int } + \text{ int}$	shift
...	
$E \times E \uparrow + \text{ int}$	reduce $E \rightarrow \text{ int } \times E$
$E \uparrow + \text{ int}$	shift
$E + \uparrow \text{ int}$	shift
$E + \text{ int } \uparrow$	reduce $E \rightarrow \text{ int}$
$E + E \uparrow$	reduce $E \rightarrow E + E$
$E \uparrow$	

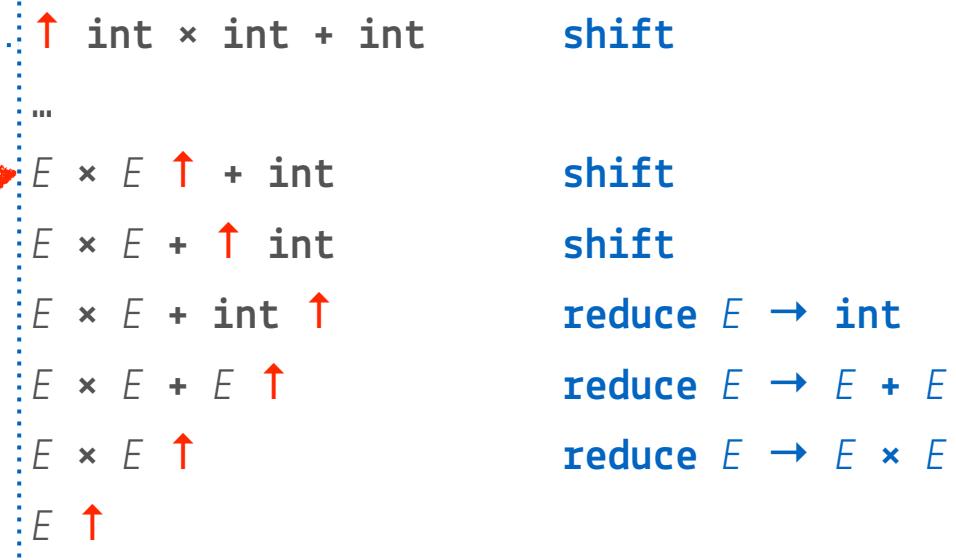
Source of conflicts and example



We can decide to either
shift or reduce in this step

The choice whether to shift or
reduce determines the
associativity of $+$ and \times !

Another derivation
is also possible:



Resolving conflicts: precedence

The choice whether to shift or reduce determines the associativity of `+` and `*`

- We could rewrite the grammar to enforce precedence (as seen with top-down parsing)
- Alternative:
provide ***precedence declarations***
 - these cause shift-reduce parsers to resolve conflicts in certain ways
 - Declaring “`*` has greater precedence than `+`” causes parser to reduce at $E \times E \uparrow + \text{int}$
 - More precisely, precedence declaration is used to resolve conflict between reducing a `*` and shifting a `+`

$$\begin{array}{l} E \rightarrow E + E \\ | \quad E \times E \\ | \quad (E) \\ | \quad \text{int} \end{array}$$

The term “precedence declaration” is misleading. These declarations do not define precedence; they define conflict resolutions

What now?

- Our key ingredients for bottom-up parsing:
 - a stack to shift and reduce symbols on
 - an automaton that can use stacked history to backtrack its footsteps
- The LR(k) family of languages can all be parsed using a shift-reduce parser like this
- The complexity of the grammars you can handle is related to how elaborate your automaton is
 - several variants: SLR, LALR, LR(1)
 - Let's start with a simple one, LR(0), in the next lecture