HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 6: LR Parsing

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Announcements

- Proj 1 out
  - Due today Thursday Sept 20, 11.59pm
- Proj 2 out
  - Due Thursday Oct 4 (14 days away)

Today

- LR Parsing
  - Constructing a DFA and LR parsing table
  - Using Yacc

LR(*k*)

Left-to-right parse

Rightmost derivation

Derivation expands the rightmost non-terminal

(Constructs derivation in reverse order!)

k-symbol lookahead

LR(k)

- Basic idea: LR parser has a stack and input
  - Given contents of stack and k tokens look-ahead parser does one of following operations:
    - Shift: move first input token to top of stack
    - Reduce: top of stack matches rule, e.g., $X \to A\ B\ C$
      - Pop C, pop B, pop A, and push X

Example

$$E \rightarrow \texttt{int}$$
$$E \rightarrow \texttt{(}E\texttt{)}$$
$$E \rightarrow E \texttt{ + } E$$

Stack **Input**

`( 3 + 4 ) + ( 5 + 6 )`

Shift `(` on to stack

Example

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

Stack	**Input**
`(`	`3 + 4 ) + ( 5 + 6 )`

Shift `(` on to stack
Shift `3` on to stack

Example

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

Stack	**Input**
`( 3`	`+ 4 ) + ( 5 + 6 )`

Shift `(` on to stack
Shift `3` on to stack
Reduce using rule $E \rightarrow$ `int`

Example

$E \rightarrow$ `int`

$E \rightarrow$ `(` E `)`

$E \rightarrow E$ `+` E

Stack	**Input**
`(` E	`+4)+(5+6)`

Shift `(` on to stack
Shift `3` on to stack
Reduce using rule $E \rightarrow$ `int`
Shift `+` on to stack

Example

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

Stack **Input**

`(` E `+` `4 ) + ( 5 + 6 )`

Shift `(` on to stack
Shift `3` on to stack
Reduce using rule $E \rightarrow$ `int`
Shift `+` on to stack
Shift `4` on to stack

Example

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

Stack	**Input**
(E + 4	) + (5 + 6)

Shift (on to stack
Shift 3 on to stack
Reduce using rule $E \rightarrow$ `int`
Shift + on to stack
Shift 4 on to stack
Reduce using rule $E \rightarrow$ `int`

Example

$E \rightarrow$ `int`

$E \rightarrow$ `(` E `)`

$E \rightarrow E + E$

Stack	**Input**
`(` $E + E$	`) + ( 5 + 6 )`

Shift `(` on to stack

Shift `3` on to stack

Reduce using rule $E \rightarrow$ `int`

Shift `+` on to stack

Shift `4` on to stack

Reduce using rule $E \rightarrow$ `int`

Reduce using rule $E \rightarrow E + E$

Example

$E \rightarrow$ `int`

$E \rightarrow$ `(` E `)`

$E \rightarrow E + E$

Stack	**Input**
`(` E	`) + ( 5 + 6 )`

Reduce using rule $E \rightarrow E + E$

Shift `)` on to stack

Example

$$E \rightarrow \texttt{int}$$
$$E \rightarrow (E)$$
$$E \rightarrow E + E$$

Stack	**Input**
(E)	$+ (5 + 6)$

Reduce using rule $E \rightarrow E + E$

Shift) on to stack

Reduce using rule $E \rightarrow (E)$

Example

$E \rightarrow$ `int`

$E \rightarrow$ `(` E `)`

$E \rightarrow E + E$

Stack	**Input**
E	`+(5+6)`

Reduce using rule $E \rightarrow E + E$

Shift `)` on to stack

Reduce using rule $E \rightarrow$ `(` E `)`

Shift + on to stack

Example

$$E \to \texttt{int}$$
$$E \to (E)$$
$$E \to E + E$$

Stack	**Input**
$E +$	`(5+6)`

Reduce using rule $E \to E + E$

Shift `)` on to stack

Reduce using rule $E \to (E)$

Shift + on to stack

... and so on ...

Example

$$E \to \texttt{int}$$
$$E \to (E)$$
$$E \to E + E$$

Stack	**Input**
$E + (E$	$+ 6)$

Reduce using rule $E \to E + E$

Shift) on to stack

Reduce using rule $E \to (E)$

Shift + on to stack

… and so on …

Example

$$E \rightarrow \texttt{int}$$
$$E \rightarrow (\,E\,)$$
$$E \rightarrow E + E$$

Stack	**Input**
$E + (E + E$	$)$

Reduce using rule $E \rightarrow E + E$

Shift $)$ on to stack

Reduce using rule $E \rightarrow (\,E\,)$

Shift + on to stack

... and so on ...

Example

$$E \to \texttt{int}$$
$$E \to \texttt{(}\,E\,\texttt{)}$$
$$E \to E + E$$

Stack	**Input**
$E + \texttt{(}\ E$	$\texttt{)}$

Reduce using rule $E \to E + E$

Shift $\texttt{)}$ on to stack

Reduce using rule $E \to \texttt{(}\,E\,\texttt{)}$

Shift + on to stack

... and so on ...

Example

$$E \rightarrow \texttt{int}$$
$$E \rightarrow \texttt{(}E\texttt{)}$$
$$E \rightarrow E + E$$

Stack **Input**

$E + E$

Reduce using rule $E \rightarrow E + E$

Shift `)` on to stack

Reduce using rule $E \rightarrow \texttt{(}E\texttt{)}$

Shift + on to stack

... and so on ...

Example

$$E \rightarrow \texttt{int}$$
$$E \rightarrow \texttt{(}E\texttt{)}$$
$$E \rightarrow E + E$$

Stack	**Input**
E	

Reduce using rule $E \rightarrow E + E$

Shift `)` on to stack

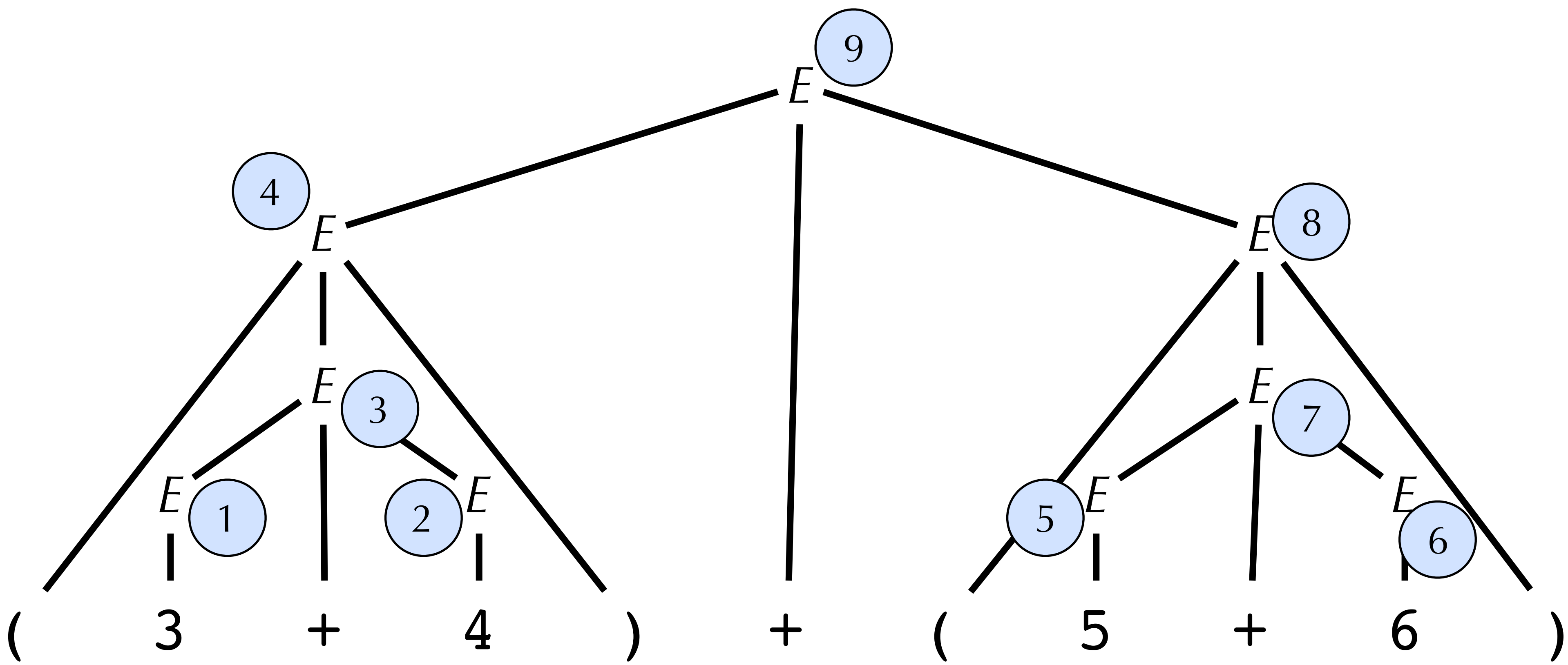
Reduce using rule $E \rightarrow \texttt{(}E\texttt{)}$

Shift + on to stack

... and so on ...

Rightmost derivation

- LR parsers produce a rightmost derivation



- But do reductions in reverse order

What Action to Take?

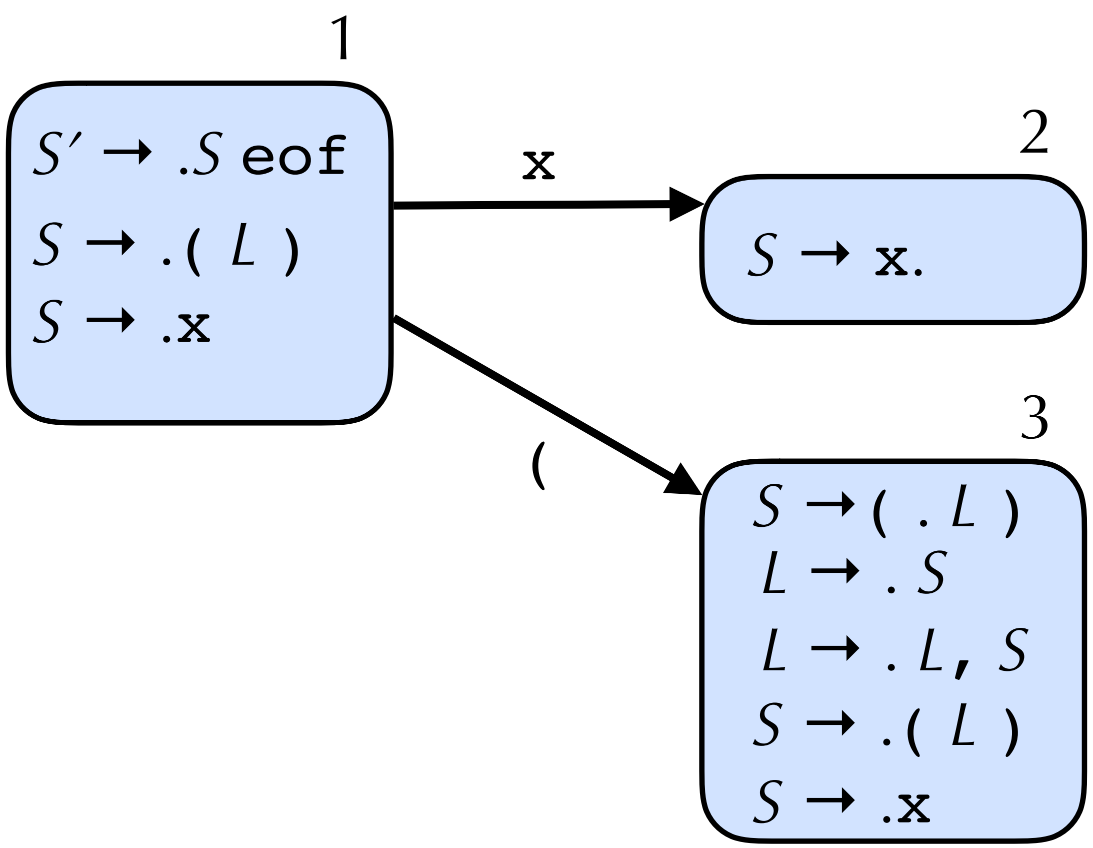
- How does the LR(k) parser know when to shift and to reduce?
- Uses a DFA
  - At each step, parser runs DFA using symbols on stack as input
    - Input is sequence of terminals and non-terminals from bottom to top
  - Current state of DFA plus next *k* tokens indicate whether to shift or reduce

Building the DFA for LR parsing

- Sketch only. For details, see Appel
- States of DFA are sets of **items**
  - An **item** is a production with an indication of current position of parser
  - E.g., Item $E \rightarrow E\ .\ +\ E$ means that for production $E \rightarrow E + E$, we have parsed first expression E have yet to parse + token
  - In general item $X \rightarrow \gamma\ .\ \delta$ means γ is at the top of the stack, and at the head of the input there is a string derivable from δ

Example: LR(0)

Add new start symbol with production to indicate end-of-file



$S' \to S$ `eof`

$S \to$ `(` L `)`

$S \to$ `x`

$L \to S$

$L \to L$ `,` S

First item of first state: at the start of input

State 1: item is about to parse *S*: add productions for *S*

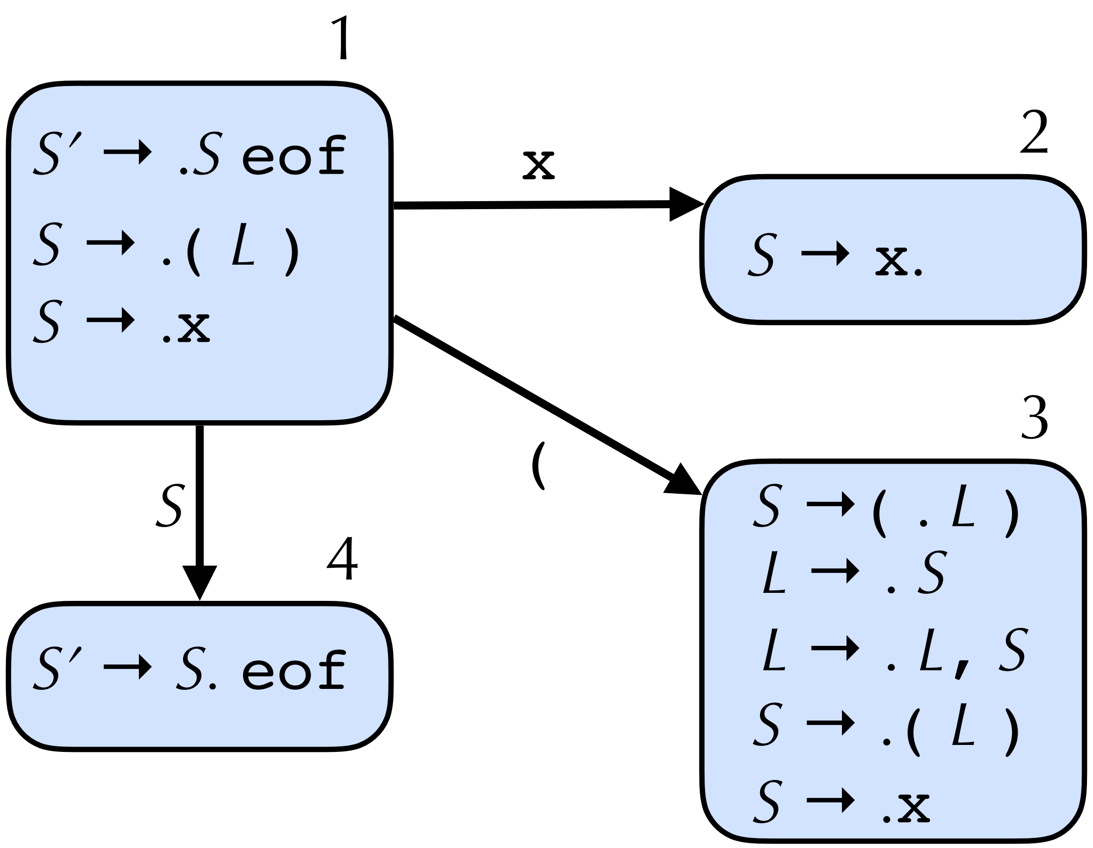
From state 1, can take `x`, moving us to state 2

From state 1, can take `(`, moving us to state 3

State 3: item is about to parse *L*: add productions for *L*

State 3: item is about to parse *S*: add productions for *S*

Example: LR(0)



$S' \to S$ eof

$S \to (L)$

$S \to$ x

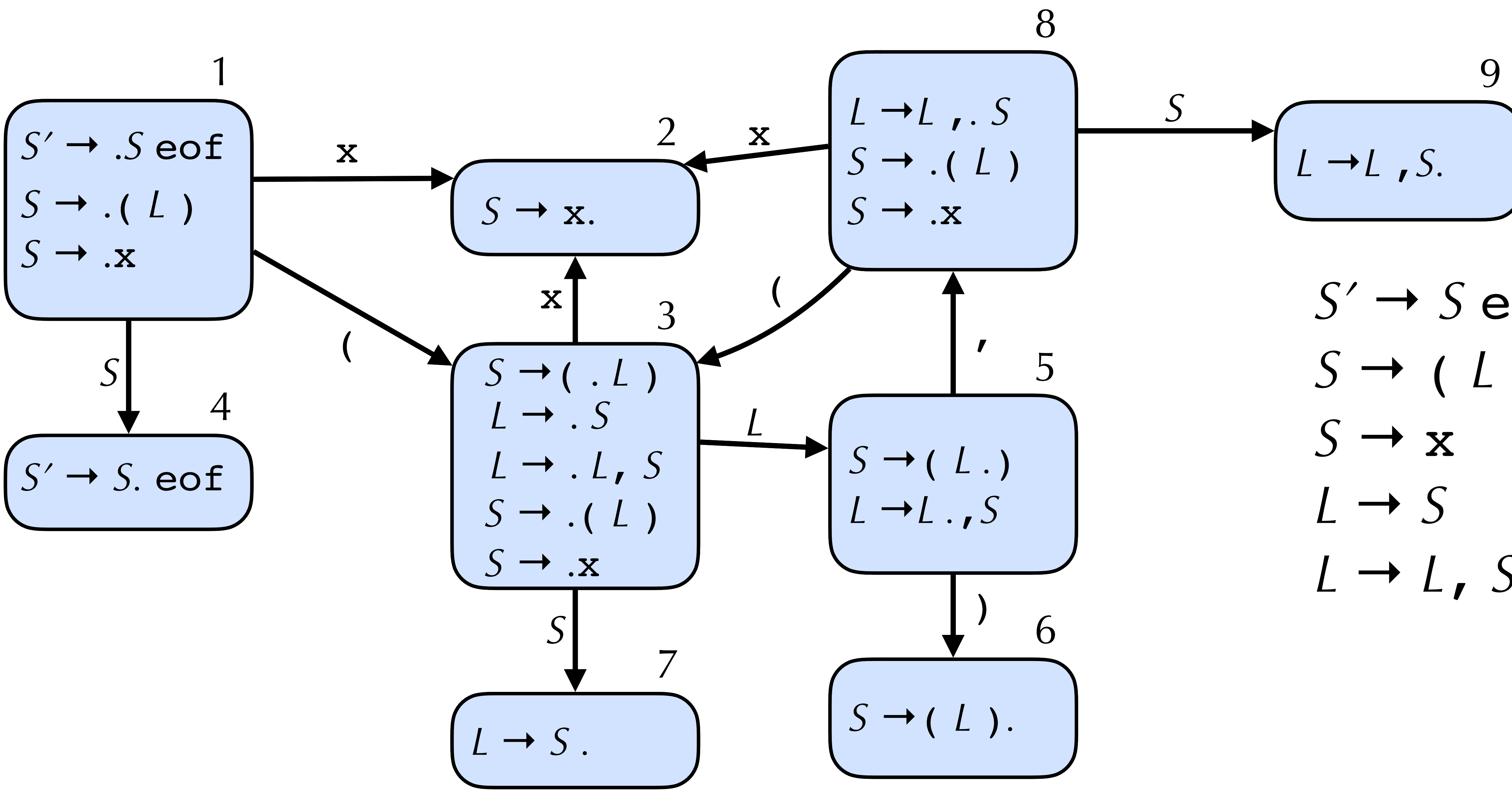
$L \to S$

$L \to L , S$

State 1: can take S, moving us to state 4

State 4 is an accepting state (if at end of input)

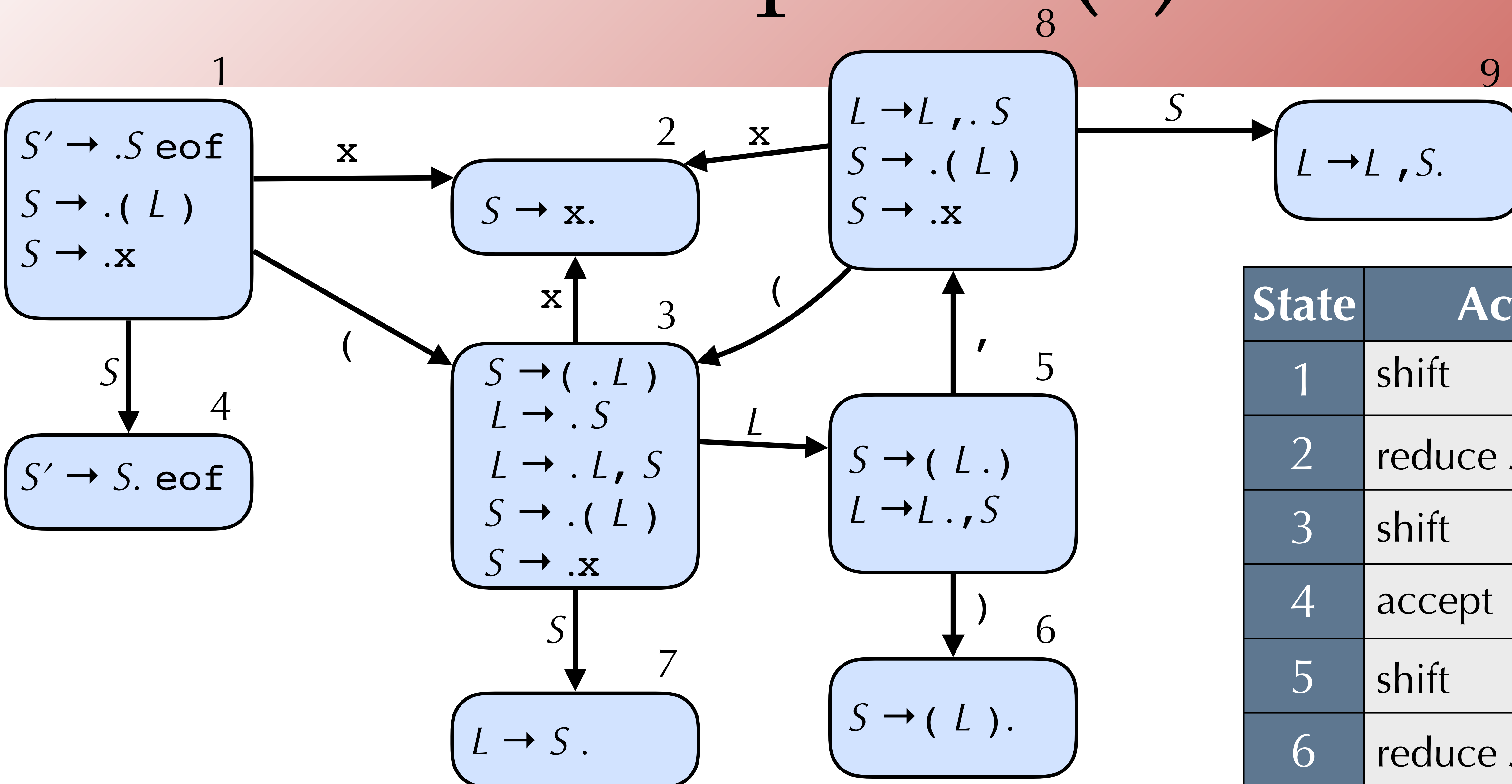
Example: LR(0)



$S' \rightarrow S$ eof
$S \rightarrow$ (L)
$S \rightarrow$ x
$L \rightarrow S$
$L \rightarrow L$, S

Continue to add states based on next symbol in item

Example LR(0)



State	Action
1	shift
2	reduce $S \rightarrow$ `x`
3	shift
4	accept
5	shift
6	reduce $S \rightarrow (L)$
7	reduce $L \rightarrow S$
8	shift
9	reduce $L \rightarrow L , S$

- Build action table
- If state contains item $X \rightarrow \gamma$.`eof` then **accept**
- If state contains item $X \rightarrow \gamma.$ then **reduce** $X \rightarrow \gamma$
- If state *i* has edge to *j* with terminal then **shift**

LR(1)

- In practice, LR(1) is used for LR parsing
  - not LR(0) or LR(k) for k>1

- Item is now pair ($X \rightarrow \gamma$. δ, x)
  - Indicates that γ is at the top of the stack, and at the head of the input there is a string derivable from δx (where x is terminal)
  - Algorithm for constructing state transition table and action table adapted. See Appel for details.
    - Closure operation when constructing states uses FIRST(), incorporating lookahead token
    - Action table columns now terminals (i.e., 1-token lookahead)
    - Note: state transition relation and action table typically combined into single table, called **parsing table**

LR Parsing and Ambiguous Grammars

- If grammar is ambiguous, we can't construct a DFA!
- We get **conflicts**: don't know which action to take
  - Shift-reduce conflicts: don't know whether to shift or reduce
  - Reduce-reduce conflicts: don't know which production to use to reduce

Dangling Else Problem

- Many language have productions such as

  $S \rightarrow$ `if` E `then` S `else` S

  $S \rightarrow$ `if` E `then` S

  $S \rightarrow \ldots$

- Program `if a then if b then s1 else s2` could be

  either `if a then { if b then s1 } else s2`

  or `if a then {if b then s1 else s2 }`

- In LR parsing table there will be a shift-reduce conflict
  - $S \rightarrow$ `if` E `then` S . with lookahead `else`: reduce
  - $S \rightarrow$ `if` E `then` S . `else` S with any lookahead: shift
  - Which action corresponds to which interpretation of

    `if a then if b then s1 else s2` ?

Resolving Ambiguity

- Could rewrite grammar to avoid ambiguity
  - E.g.,

$$S \rightarrow O$$
$$O \rightarrow V \; \texttt{:=} \; E$$
$$O \rightarrow \texttt{if} \; E \; \texttt{then} \; O$$
$$O \rightarrow \texttt{if} \; E \; \texttt{then} \; C \; \texttt{else} \; O$$
$$C \rightarrow V \; \texttt{:=} \; E$$
$$C \rightarrow \texttt{if} \; E \; \texttt{then} \; C \; \texttt{else} \; C$$

Resolving Ambiguity

- Or tolerate conflicts, indicating how to resolve conflict
  - E.g., for dangling else, prefer shift to reduce.
    - i.e., for `if a then if b then s1 else s2`
      prefer `if a then {if b then s1 else s2 }`
      over `if a then { if b then s1 } else s2`
    - i.e., `else` binds to closest `if`
  - Expression grammars can express operator-precedence by resolution of conflicts
- Use sparingly! Only in well-understood cases
  - Most conflicts are indicative of ill-specified grammars

Using Yacc

- **Y**et **A**nother **C**ompiler-**C**ompiler
- Originally developed in early 1970s
- Various versions/reimplimentations
  - Berkeley Yacc, Bison, Ocamlyacc, ...
- From a suitable grammar, constructs an LALR(1) parser
  - A kind of LR parser, not as powerful as LR(1)
  - Most practical LR(1) grammars will be LALR(1) grammars

Ocamlyacc

- Usage: `ocamlyacc` *`options grammar`*`.mly`
- Produces output files
  - *`grammar`*`.ml`: OCaml code for a parser
  - *`grammar`*`.mli`: interface for parser

Structure of ocamlyacc File

```
%{
  header
%}
  declarations
%%
  rules
%%
  trailer
```

- Header and trailer are arbitrary OCaml code, copied to the output file
- Declarations of tokens, start symbols, OCaml types of symbols, associativity and precedence of operators
- Rules are productions for non-terminals, with **semantic actions** (OCaml expressions that are executed with production is reduced, to produce value for symbol)

Ocamlyacc example

- See `Lec03-parser-eg.mll`
  and output files `Lec03-parser-eg.ml`
  and `Lec03-parser-eg.mli`
- Typically, the `.mly` declares the tokens, and the lexer opens the parser module