Lecture E:

LR Parser Construction

CSE401/501m:

Introduction to Compiler Construction

Instructor: Gilbert Bernstein

Administrivia (1)

- Scanners due Thursday, 11:59 pm — how's it going?
  - Make sure to read the MiniJava overview & Scanner assignment — then reread ***again*** when you're "done"
    - Did you implement both kinds of comments
    - Did you handle every kind of token in the MiniJava grammar?
    - Anything "quoted" in the MiniJava grammar should be treated as a reserved word (Token) in MiniJava (even if it's not a single token in full Java)
  - Be sure you can handle comments at the end of the file, and files without newlines at the end (& both)
  - Scanner should continue after "invalid input character" errors
  - Be sure to terminate with correct System.exit code (0=ok, 1=errors) — don't be creative with the spec
  - Take advantage of Flex regex operations that go beyond basic regexes from class if they're useful
  - Don't implement the parser yet!
  - Reminder: you have a partner(!) — take advantage of that
    - On Ed & Email: it's "We have a question" not "I have a question"

Administrivia (2)

- Coming up…
  - ✦ Today & Fri & in sections: LR Parsing and LR Parser construction
  - ✦ HW 2 (grammars, LR Parsing) out tonight or tomorrow morning
  - ✦ Mon — AST visitors (now you know what you need for the Parser)
    - Parser project will be out shortly after that

Administrivia (Friday)

- Hooray! Scanners are done!
    - Was gradescope annoying?
- HW2 is out

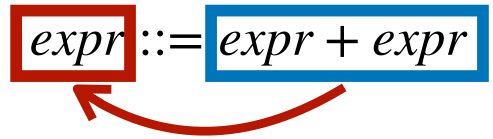
LR Parsing Recap

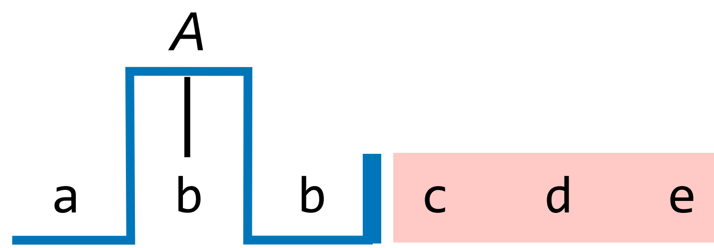
"Bottom-up" Parsing — match **right-hand sides**

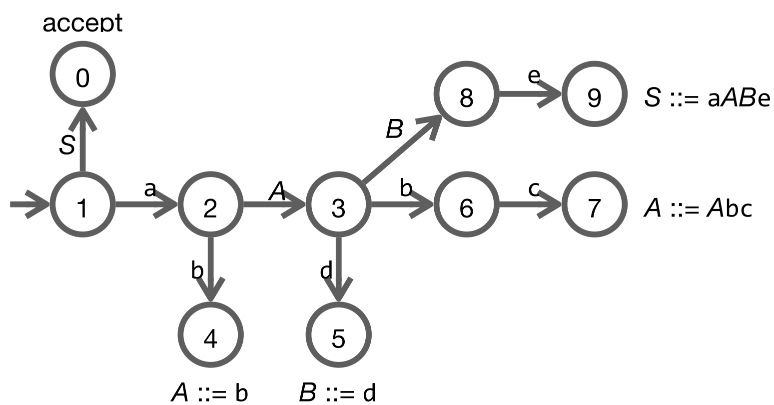
Doing this while scanning **left-to-right** produces a "frontier" (i.e. the stack)

Deciding when to **shift** vs. **reduce** can be decided via a DFA that recognizes valid prefixes

This DFA can be encoded into an **LR table**







State	action						goto		
	a	b	c	d	e	$	A	B	S
0						*acc*			
1	s2								g0
2		s4					g3		
3		s6		s5				g8	
4	r-	r-	r-	r-	r-	r-			
5	r-IV	r-IV	r-IV	r-IV	r-IV	r-IV			
6			s7						
7	r-II	r-II	r-II	r-II	r-II	r-II			
8					s9				
9	r-I	r-I	r-I	r-I	r-I	r-I			

Today's Question

How do we build the DFA (and thus LR table) from a Grammar

Outline

LR(0) State Machine Construction

SLR Variation

- FIRST, FOLLOW, and nullable analyses

LR(k) and LALR Variations

Outline

LR(0) State Machine Construction

SLR Variation

- FIRST, FOLLOW, and nullable analyses

LR(k) and LALR Variations

LR State Machine

- Idea — Build a DFA that
  - ✦ Avoids errors so long as the LR stack is a **viable prefix**
  - ✦ Recognizes and accepts whenever a **reduction** should be performed (aka. a *handle* is recognized)
- Because the language of viable prefixes for a CFG is regular, a DFA will suffice
- Crux of idea — DFA states will correspond to sets of **items**, which keep track of *where* we are in the middle of matching the right-hand side of different production rules

Theory/Terminology (Review)

- Parsing corresponds to a rightmost derivation in reverse
  - ✦ $S \Rightarrow_{\mathrm{rm}} \beta_1 \Rightarrow_{\mathrm{rm}} \cdots \Rightarrow_{\mathrm{rm}} \beta_n$
- Each step is $\alpha A w \Rightarrow_{\mathrm{rm}} \alpha\beta w$ for production $A ::= \beta$
  - ✦ A **viable prefix** is a prefix γ of $\alpha\beta$ for some such step
    - i.e. these are the possible states of the LR stack
  - ✦ The occurrence β in $\alpha\beta w$ is called a **handle**
- An **item** is a marked production (a . in its right-hand side)
  - ✦ e.g. $A ::= .XY$ $A ::= X.Y$ $A ::= XY.$

A New Example Grammar

- Example grammar

$$S' ::= S \; \$$$
$$S ::= (\; L \;)$$
$$S ::= x$$
$$L ::= S$$
$$L ::= L \; \underline{,} \; S$$

- (note: we are now adding a production $S' ::= S \; \$$ to normalize the handling of initial and final states)
- Question: What language does this grammar generate?

Start of LR Parse

$$
\begin{aligned}
(0)\ & S' ::= S\ \$ \\
(I)\ & S ::= (\ L\ ) \\
(II)\ & S ::= x \\
(III)\ & L ::= S \\
(IV)\ & L ::= L\ \underline{,}\ S
\end{aligned}
$$

- Initial State
  - Stack is empty (except for start state number)
  - Initial state contains the item [*S*' ::= . *S* $]
- But, since the position (.) is just before *S*, we are also just before anything that can be derived from *S*
- What else can be derived from *S* (directly or indirectly)?

Initial State

$$
\begin{aligned}
(0)\ & S' ::= S\ \$ \\
(I)\ & S ::= (\ L\ ) \\
(II)\ & S ::= x \\
(III)\ & L ::= S \\
(IV)\ & L ::= L\ ,\ S
\end{aligned}
$$

$[S' ::= .\ S\ \$]$ ← **start**

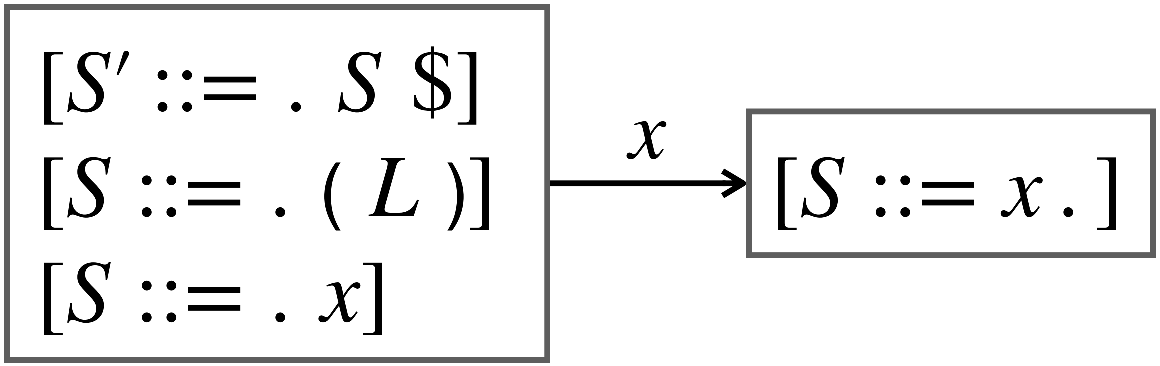
$[S ::= .\ (\ L\ )]$ ← **completion**

$[S ::= .\ x]$ ← **completion**

- A **state** is just a set of **items**
  - **start** — an initial set of items
  - **closure** (aka. completion) — additional productions whose left-hand side nonterminal appears immediately following a dot in some item already in the state

Shift Actions (1)

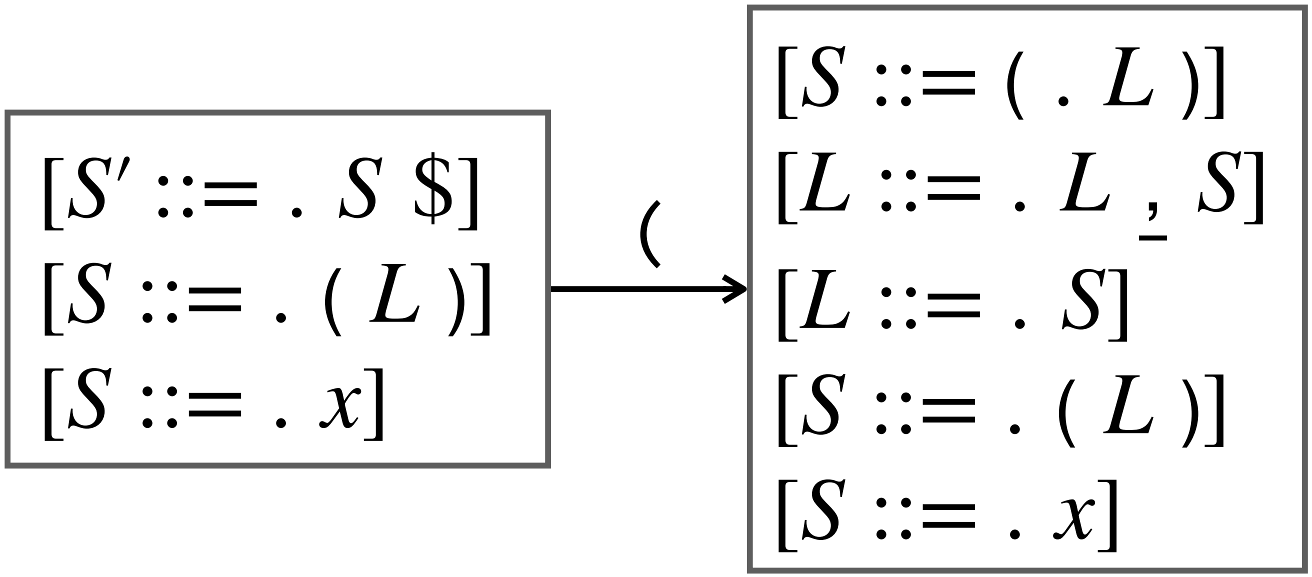
$$
\begin{aligned}
(0)\ & S' ::= S\ \$ \\
(I)\ & S ::= (\ L\ ) \\
(II)\ & S ::= x \\
(III)\ & L ::= S \\
(IV)\ & L ::= L\ ,\ S
\end{aligned}
$$



- To shift on an x, add a new state with appropriate item(s), including their closure
    - In this case, the closure adds no additional items
    - This state will lead to a reduction, since no further shift is possible.

Shift Actions (2)

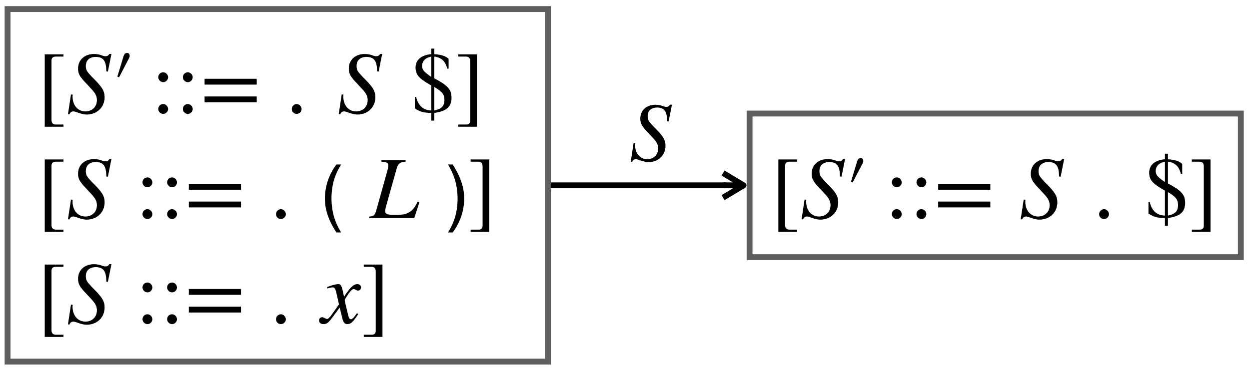
$$
\begin{array}{rl}
(0) & S' ::= S\ \$ \\
(I) & S ::= (\ L\ ) \\
(II) & S ::= x \\
(III) & L ::= S \\
(IV) & L ::= L\ \underline{,}\ S
\end{array}
$$



- If we shift past the (, then we are at the beginning of *L*
- The closure adds all productions that start with *L*
    - which further requires adding all productions starting with *S*

Goto Actions

$$
\begin{aligned}
(0)\ & S' ::= S\ \$ \\
(I)\ & S ::= (\ L\ ) \\
(II)\ & S ::= x \\
(III)\ & L ::= S \\
(IV)\ & L ::= L\ ,\ S
\end{aligned}
$$



- Besides transitioning on terminal symbols, we also want to add transitions on non-terminal symbols. These transitions will get entered into the goto table.
  - remember: these get used to transition after reductions pop the stack

Basic Operations

for Constructing LR States

- `Closure (U)`
  - Returns *U* with all further items implied by *U* included
- `Goto (U, X)`
  - *U* is a set of items
  - *X* is a grammar symbol (terminal or non-terminal)
  - Goto moves the current position (`.`) past the symbol *X* for all items in *U*, discarding the item if *X* is not the next symbol, or including the progressed item if it is

Computing `Closure(U)`

- The Basic Principle
  - If $[A ::= \alpha \,.\, B\,\beta]$ is in `Closure(U)`, (B a non-terminal) and $B ::= \gamma$ is a production, then $[B ::= \,.\,\gamma]$ is in `Closure(U)` as well; (also `U` ⊆ `Closure(U)`)
- Algorithm — **It's a fixed point!** i.e. keep applying the above principle until convergence.

```
Closure(U) {
  repeat {
    for (item [A ::= α . B β] in U)
      for (production B ::= γ)
        U.add([B ::= . γ]);
  } until U does not change;
  return U;
}
```

Computing `Goto(U,X)`

- The Basic Principle
  - If $[A ::= \alpha\,.\,X\beta]$ is in `U`, (X any symbol) and `U'` is the state reached by transitioning on symbol X, then $[A ::= \alpha X\,.\,\beta]$ is in `U'` (& both states should be closed)
- Algorithm
  Not a fixed-point
  (no recursion)

```
Goto(U, X) {
  new_U = empty_set();
  for (item [A ::= α . X β] in U)
    new_U.add([A ::= α X . β]);
  return Closure(new_U);
}
```

- Note: if the computed state already exists, then return that state, not a copy

LR(0) Construction — Init

- First, augment the grammar with an extra start production $S' ::= S\,\$$ so that the start and final states aren't special cases
- Let W be the set of states
  - Initialize W to `Closure(`$[S' ::= .\,S\,\$]$`)`
- Let E be the set of edges/transitions
  - Initialize E to $\{\,\}$, the empty set

LR(0) Construction — Iteration

- Another **fixed-point** algorithm (idea is basic principles)

```
repeat {
  for (U in W)
    for (item [A ::= α . X β] in U)
      let V = Goto(U,X)
      add V to W (if not present)
      add (U→V) to E (if not present)
} until W and E do not change
```

- Special case — For the marker `$`, we don't compute `goto(U,$)`; instead we make this an accept action

Example: States for

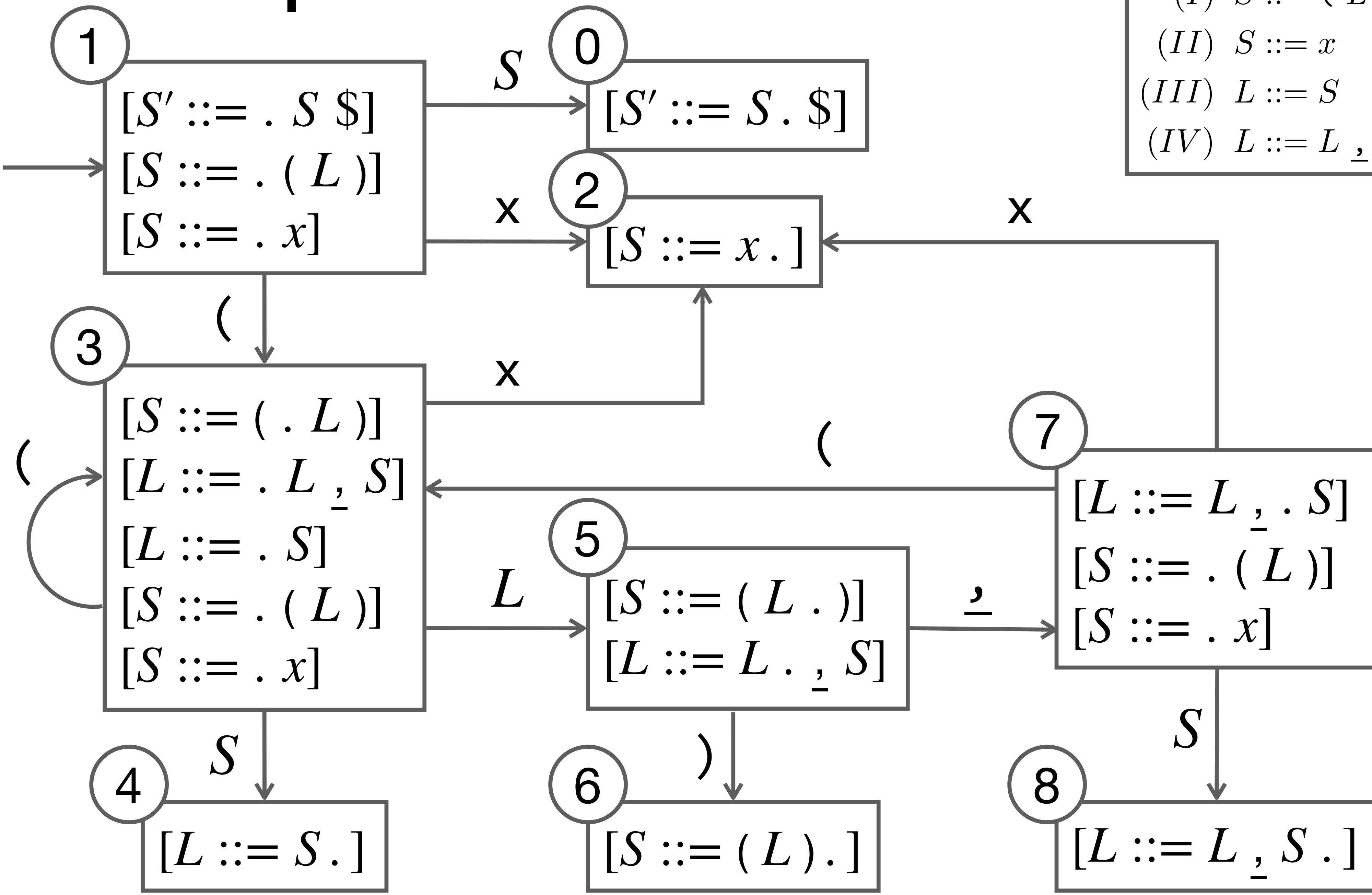
(0) $S' ::= S\ \$$

(I) $S ::= (\ L\ )$

(II) $S ::= x$

(III) $L ::= S$

(IV) $L ::= L\ ,\ S$



Building the Parse Tables (1)

- Let $id(U)$ be the state number we assign to the set of items U
- For each edge $U \xrightarrow{X} V$, let $i = id(U)$ and $j = id(V)$
  - If X is a terminal, then put **s** j into action[i,X] (visually: column X, row i)
  - If X is a non-terminal, then put **g** j into goto[i,X] (visually: column X, row i)

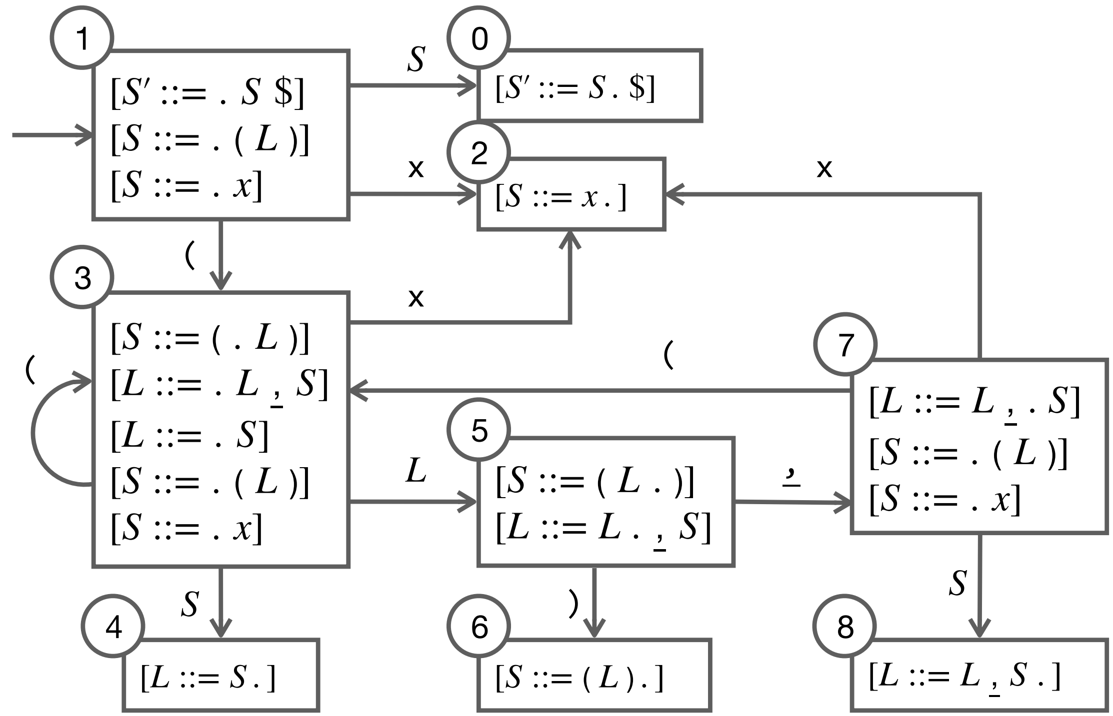
Building the Parse Tables (2)

- For each state $i = id(U)$, with item $[S' ::= S\,.\,\$]$ in U, put **accept** into `action[`i, X`]` (visually: column $\$$, row i)
- For each state $i = id(U)$, with item $[A ::= \gamma\,.\,]$ in U, put action **r-n** (reduce) into every column of row i in the `action` table (n is the **production** number of $[A ::= \gamma\,.\,]$)
  - i.e. when the DFA reaches this state, it has discovered that $A ::= \gamma$ can be applied to reduce $\alpha\gamma$ to αA on the stack

Example: Tables for

$$(0)\ S' ::= S\ \$$$
$$(I)\ S ::= (\ L\ )$$
$$(II)\ S ::= x$$
$$(III)\ L ::= S$$
$$(IV)\ L ::= L\ \underline{,}\ S$$

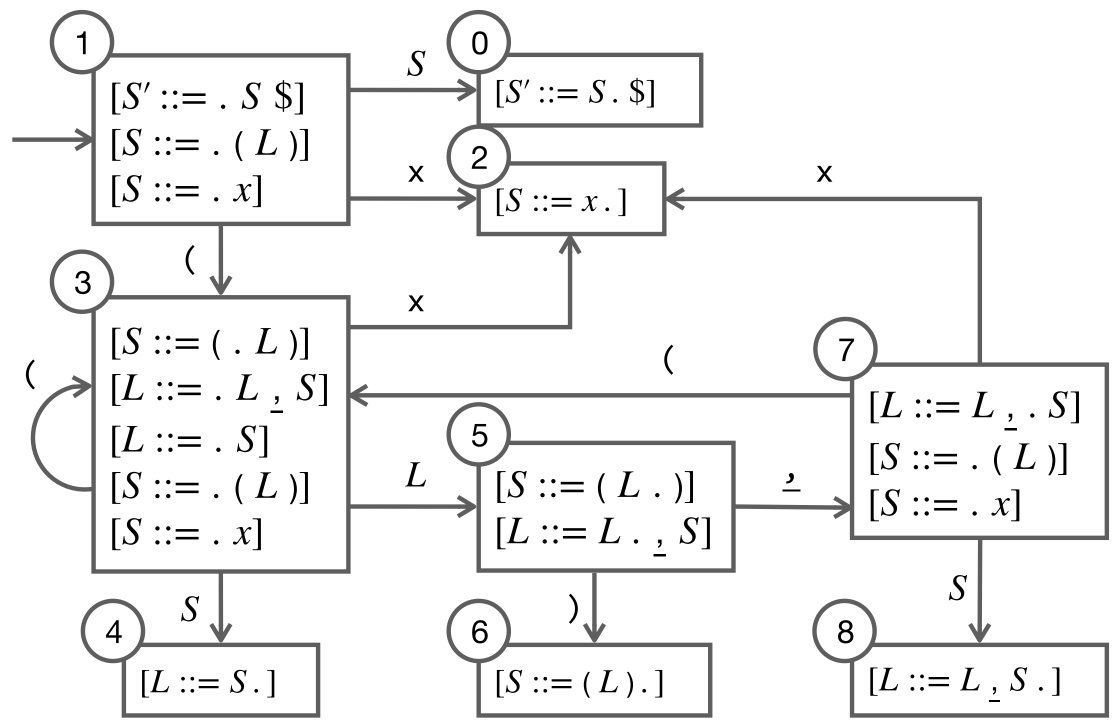
	(	)	x	,	$	S	L
0							
1							
2							
3							
4							
5							
6							
7							
8							



Example: Tables for

$$(0)\ S' ::= S\ \$$$
$$(I)\ S ::= (\ L\ )$$
$$(II)\ S ::= x$$
$$(III)\ L ::= S$$
$$(IV)\ L ::= L\ ,\ S$$

	(	)	x	,	$	S	L
0					acc		
1	s3		s2			g0	
2	r-II	r-II	r-II	r-II	r-II		
3	s3		s2			g4	g5
4	r-III	r-III	r-III	r-III	r-III		
5		s6		s7			
6	r-I	r-I	r-I	r-I	r-I		
7	s3		s2			g8	
8	r-IV	r-IV	r-IV	r-IV	r-IV		



Where do we stand?

- We have built the LR(0) state machine & parser tables
  - No lookahead yet
  - Different variations of LR parsers add lookahead information to *items*, but the basic ideas remain the same: states as sets of items, closure, and goto edges
- A grammar is LR(0) if its LR(0) state machine (equiv. parser tables) has no shift-reduce or reduce-reduce conflicts in it.
  - Note: this is easily decidable, unlike the question of whether a grammar is ambiguous!

Outline

LR(0) State Machine Construction

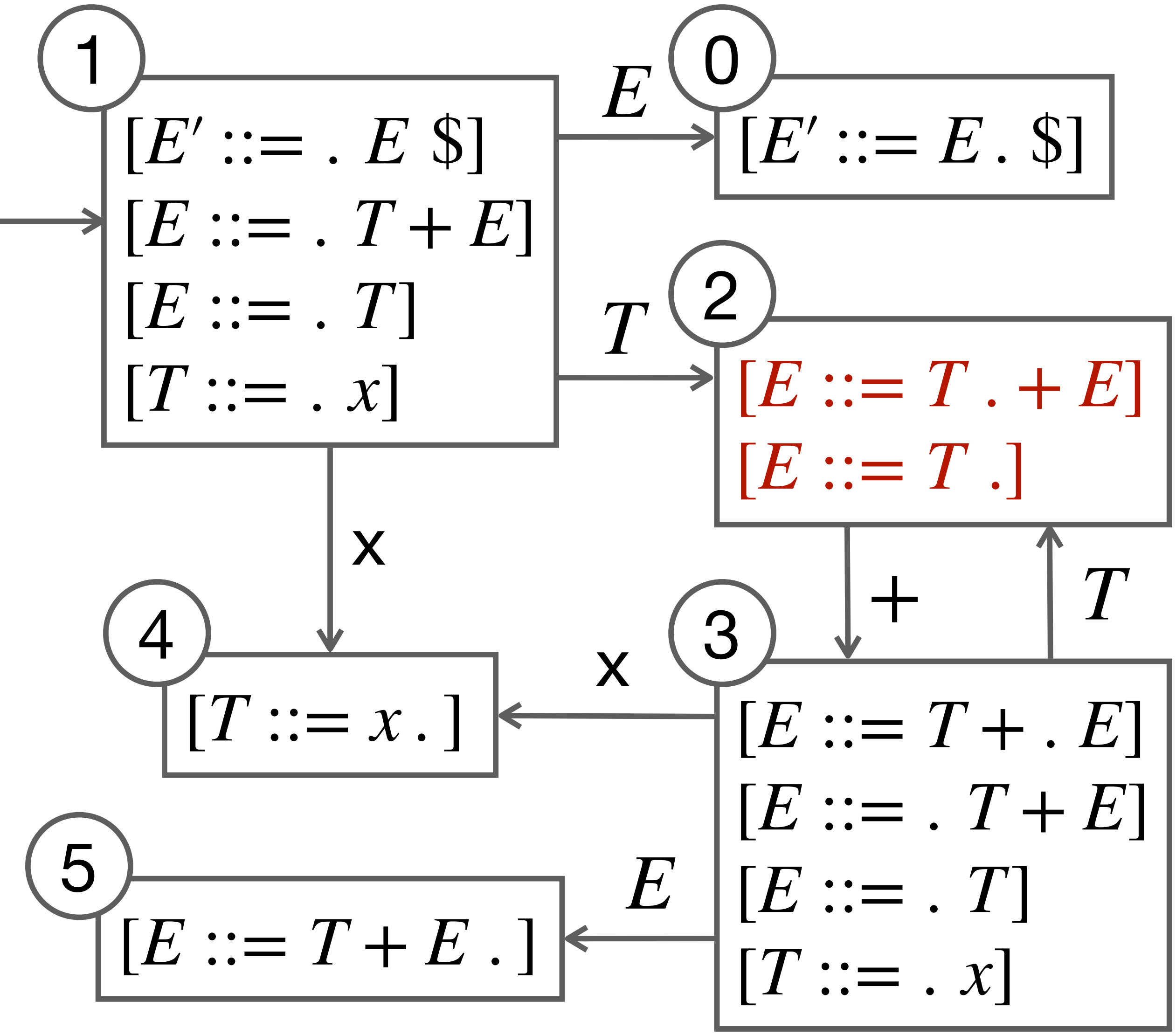
SLR Variation

- FIRST, FOLLOW, and nullable analyses

LR(k) and LALR Variations

A Grammar that is not LR(0)

State 2 has two possible actions on ‘+’: shift 4 or reduce (II)



$(0)\ E' ::= E\ \$$

$(I)\ E ::= T + E$

$(II)\ E ::= T$

$(III)\ T ::= x$

	x	+	$	E	T
0			acc		
1	s4			g0	g2
2	r-II	r-II, s3	r-II		
3	s4			g5	g2
4	r-III	r-III	r-III		
5	r-I	r-I	r-I		

Resolving Conflicts

- Look at the next symbol(s) to help decide whether or not to reduce
- Different schemes — LR(k), LALR(k), SLR
- **SLR** (**Simple LR**) — Only reduce if the next input terminal symbol could possibly follow the resulting non-terminal
  - ✦ e.g. suppose we reach a state with the item $[A ::= \beta\,.\,]$ and the next input token/terminal is x
  - ✦ Then don't reduce, unless Ax appears in some sentence in the derivation. This is the 🔑🗝️ Idea!

SLR Parsers

- Idea (again) — only reduce from $[A ::= \beta\,.\,]$ if the next token x could possibly occur after an A in the derivation
  - ✦ Therefore, we need some way to answer this question
- For each non-terminal A, we want to compute the set $FOLLOW(A)$ of all *terminal* symbols that can follow A in some possible derivation.
  - ✦ How should we compute this?

Outline

LR(0) State Machine Construction

SLR Variation

FIRST, FOLLOW, and nullable analyses

LR(k) and LALR Variations

The Catch

- Consider the grammar

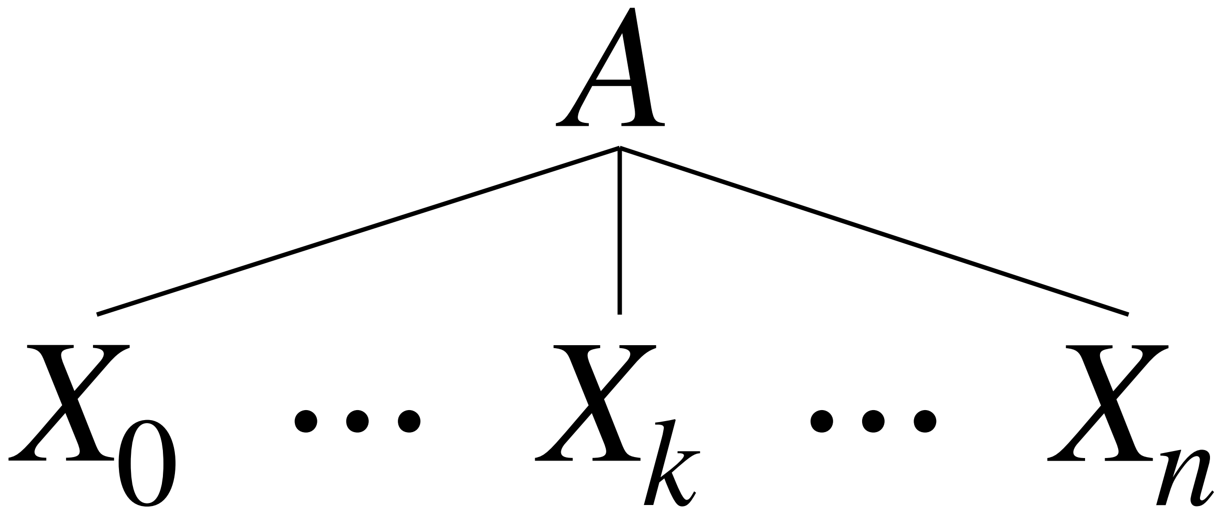
$$A ::= AB$$
$$B ::= bA$$
$$A ::= a$$

- What is the set of all terminals that can **follow** A?
- Well, the non-terminal B can follow A, so we need to know what possible terminals can occur **first** in a sentence derived from B
- What happens if we add a null production $A ::= \epsilon$?
  - Does this change which terminals can occur **first** in a sentence derived from A?
- So, we need to compute whether ϵ can be derived from a non-terminal A, directly or indirectly — is A **nullable**?

A Powerful Habit of Thought

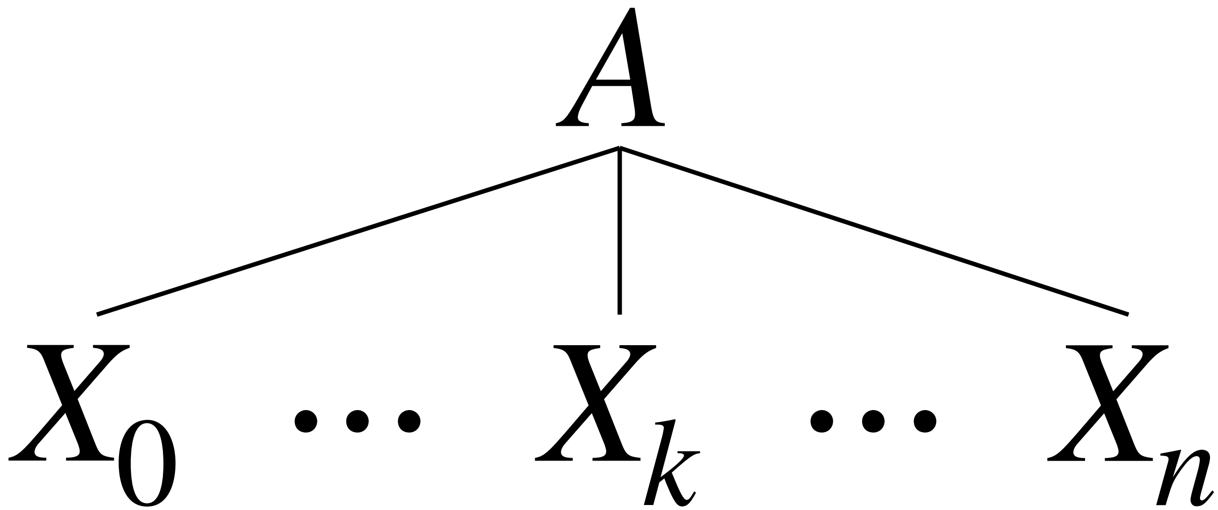
- If you feel like you're thinking in circles, STOP!
- State the ***basic principles*** with which you are thinking ***without*** trying to chase them down the rabbit hole
  - Remember — the power of writing things down!
- Even more basic, let's try to define things first
  - $FOLLOW(A)$ is the set of all *terminals* x that **follow** A in some derived sentence.
  - $FIRST(A)$ is the set of all *terminals* x that occur **first** in some sentence derived from A
  - $NULLABLE(A)$ is true if ϵ derives from A
- *note: use of NULLABLE is different than our textbook*

Consider one Production



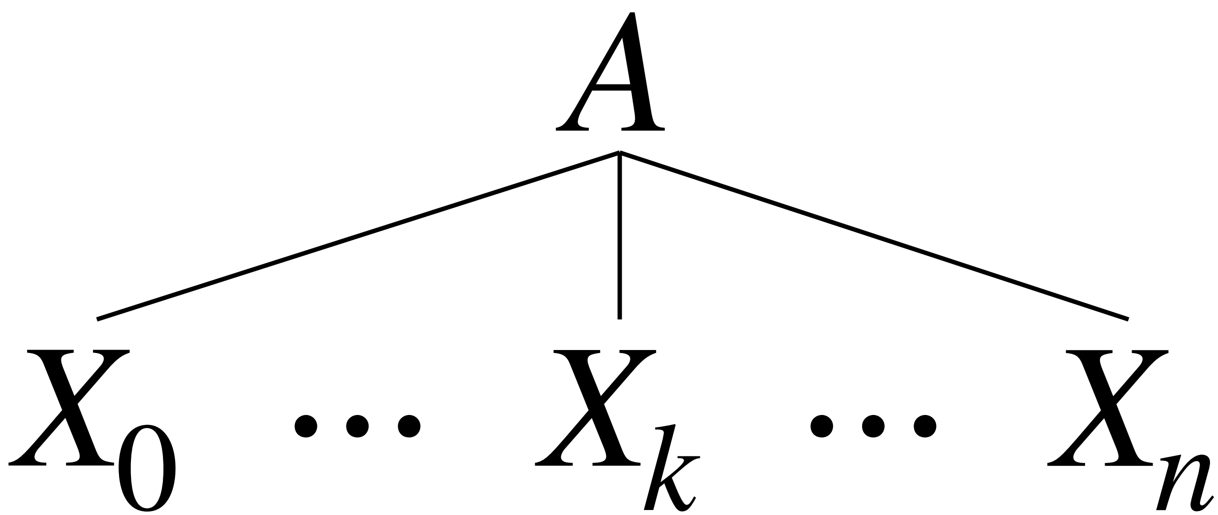
- How do $FOLLOW(A)$, $FIRST(A)$ and $NULLABLE(A)$ relate to those sets on the symbols X_i ?
- e.g. suppose x is in $FOLLOW(A)$. Then what do we know?

NULLABLE Principles



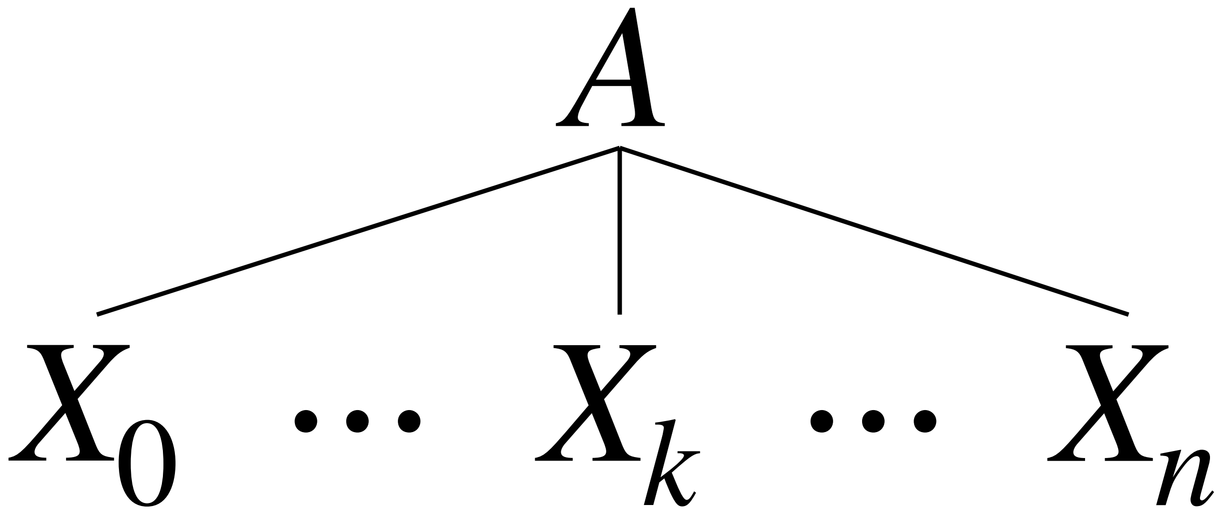
- (base cases) $NULLABLE(\epsilon)$ is true, and $NULLABLE(x)$ for any terminal x is false
- If all of $NULLABLE(X_i)$ are true for $0 \leq i \leq n$, then $NULLABLE(A)$ must be true as well

FIRST Principles



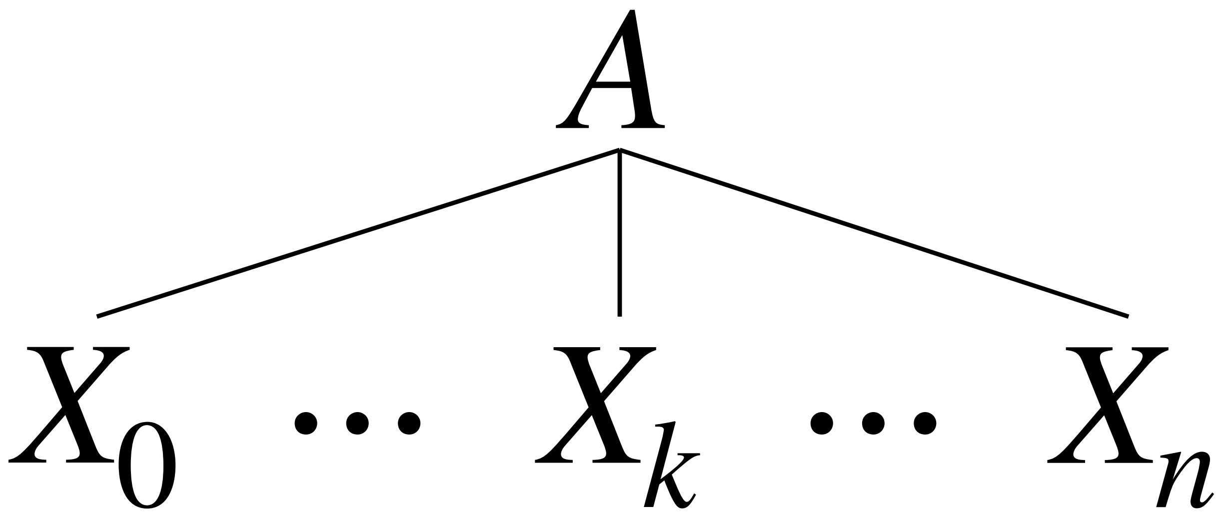
- (base cases) $FIRST(\epsilon) = \{\,\}$ and $FIRST(x) = \{x\}$
- Idea — $FIRST(A) = FIRST(X_0)$?
  - ✦ What if $NULLABLE(X_0)$ is true?
- Correction — If $NULLABLE(X_i)$ for $0 \leq i < k$, then $FIRST(X_k) \subseteq FIRST(A)$

FOLLOW Principle



- no base cases…
- Idea — $FOLLOW(A) \subseteq FOLLOW(X_n)$
- Correction — If $NULLABLE(X_i)$ for $k < i \leq n$, then $FOLLOW(A) \subseteq FOLLOW(X_k)$
- But how do we get anything into $FOLLOW$ to start?

FOLLOW / FIRST Principle



- Idea — intuitively X_{k+1} follows X_k so if the terminal x is in $FIRST(X_{k+1})$ is, it must **follow** X_k
- What if some of the X_k are nullable?
- Correction — If $NULLABLE(X_i)$ for $j < i < k$, then $FIRST(X_k) \subseteq FOLLOW(X_j)$

Basic Principles on One Slide

- $NULLABLE(\epsilon)$ is true
- $FIRST(\epsilon) = \{\,\}$ and $FIRST(x) = \{x\}$
- If $A ::= X_1X_2\cdots X_n$ and for all X_i, $NULLABLE(X_i)$ is true, then $NULLABLE(A)$ is true.
- If $A ::= X_1X_2\cdots X_k\cdots X_n$ and $NULLABLE(X_i)$ for $1 \le i < k$, then $FIRST(X_k) \subseteq FIRST(A)$
- If $A ::= X_1X_2\cdots X_k\cdots X_n$ and $NULLABLE(X_i)$ for $k < i \le n$, then $FOLLOW(A) \subseteq FOLLOW(X_k)$
- If $A ::= X_1X_2\cdots X_j\cdots X_k\cdots X_n$ and $NULLABLE(X_i)$ for $j < i < k$, then $FIRST(X_k) \subseteq FOLLOW(X_j)$

Principles → Algorithm

```
FIRST[A]    = {}      // for all non-terminals A
FOLLOW[A]   = {}      // for all non-terminals A
NULLABLE[A] = false   // for all symbols A
FIRST[x]    = {x}     // for all terminals x

repeat
  for each production A ::= X1, X2, … Xn
    if X1, X2, … Xn are all NULLABLE (or n=0) then
      set NULLABLE[A] = true
    for each k from 1 to n, and each j from 1 to k-1
      if X1, X2, … X(k-1) are all NULLABLE (or k=1) then
        add FIRST[Xk] into FIRST[A]
      if X(k+1), … Xn are all NULLABLE (or k=n) then
        add FOLLOW[A] into FOLLOW[Xk]
      if X(j+1), … X(k-1) are all NULLABLE (or j+1=k) then
        add FIRST[Xk] into FOLLOW[Xj]
until FIRST, FOLLOW, and NULLABLE do not change
```

Example

Grammar

$$Z ::= d$$
$$Z ::= X\ Y\ Z$$
$$Y ::= \epsilon$$
$$Y ::= c$$
$$X ::= Y$$
$$X ::= a$$

	NULLABLE	FIRST	FOLLOW
X	no		
Y	no		
Z	no		

Example

Grammar

$$Z ::= d$$
$$Z ::= X\ Y\ Z$$
$$Y ::= \epsilon$$
$$Y ::= c$$
$$X ::= Y$$
$$X ::= a$$

	NULLABLE	FIRST	FOLLOW
X	yes	a, c	a, c, d
Y	yes	c	a, c, d
Z	no	a, c, d	

Outline

LR(0) State Machine Construction

SLR Variation

- FIRST, FOLLOW, and nullable analyses

LR(k) and LALR Variations

LR(0) Reduce Actions (review)

- In an LR(0) parser, if a state contains a reduction, it is unconditionally applied regardless of the next input symbol
- Algorithm (DFA & Table construction)

```
initialize R to empty
for each state U in W
  for each item [A ::= α .] in U
    add (U, A ::= α) to R
```

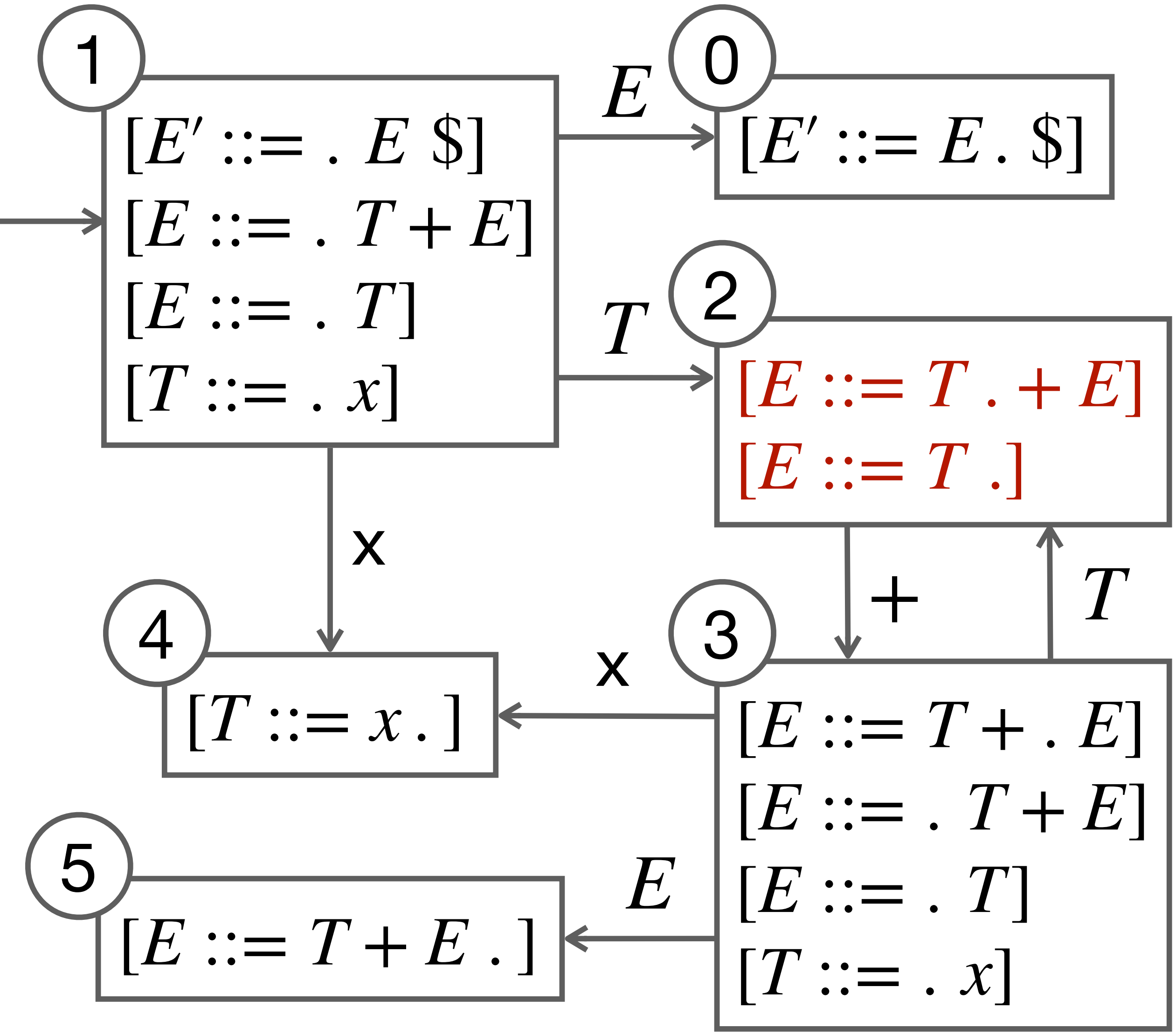
SLR Construction

- This is identical to LR(0) — same construction of states, DFA transitions, etc. Only change calculation of reduce actions
- Algorithm

```
initialize R to empty
for each state U in W
  for each item [A ::= α .] in U
    for each terminal x in FOLLOW[A]
      add (U, x, A ::= α) to R
```

SLR Parser for Earlier Example

Using the **FOLLOW** criteria, we **filter out** some reductions



$$
\begin{aligned}
(0)\ & E' ::= E\ \$ \\
(I)\ & E ::= T + E \\
(II)\ & E ::= T \\
(III)\ & T ::= x
\end{aligned}
$$

	x	+	$	E	T
0			**acc**		
1	**s4**			**g0**	**g2**
2	**r-II**	**s3, r-II**	**r-II**		
3	**s4**			**g5**	**g2**
4	**r-III**	**r-III**	**r-III**		
5	**r-I**	**r-I**	**r-I**		

Outline

LR(0) State Machine Construction

SLR Variation

- FIRST, FOLLOW, and nullable analyses

LR(k) and LALR Variations

On to LR(1)

- Many practical grammars are SLR
- But LR(1) is even more powerful
- Similar construction, but the notion of an item is now more complex in order to incorporate Look-ahead information (LR(1) = LR with ***one*** lookahead)
  - Now lookahead information is associated with specific items rather than using **FOLLOW** for the non-terminal
  - using **FOLLOW** is less powerful, because it doesn't track as much context about where a given terminal appears in the derivation

LR(1) Items

- A general LR(0) item is $[A ::= \alpha \,.\, \beta]$
- A general LR(1) item is $[A ::= \alpha \,.\, \beta,\ x]$, consisting of
  - a grammar production $A ::= \alpha\beta$
  - a right-hand side position (the dot)
  - a lookahead terminal symbol (x)
- Idea — This item indicates that α is on the top of the stack, and it would still be possible to match the next sequence of tokens with βx
- For a full construction, see the book

LR(1) Tradeoffs

- LR(1)
  - Pro — more precise; LR(k) admits the largest number of grammars
  - Con — can produce ***very*** large parse tables with many states

LALR(1)

- Variation of LR(1), but merge any two states that differ only in lookahead
  - e.g. these two would be merged

$$[A ::= x\ .\ y,\ a]$$

$$[A ::= x\ .\ y,\ b]$$

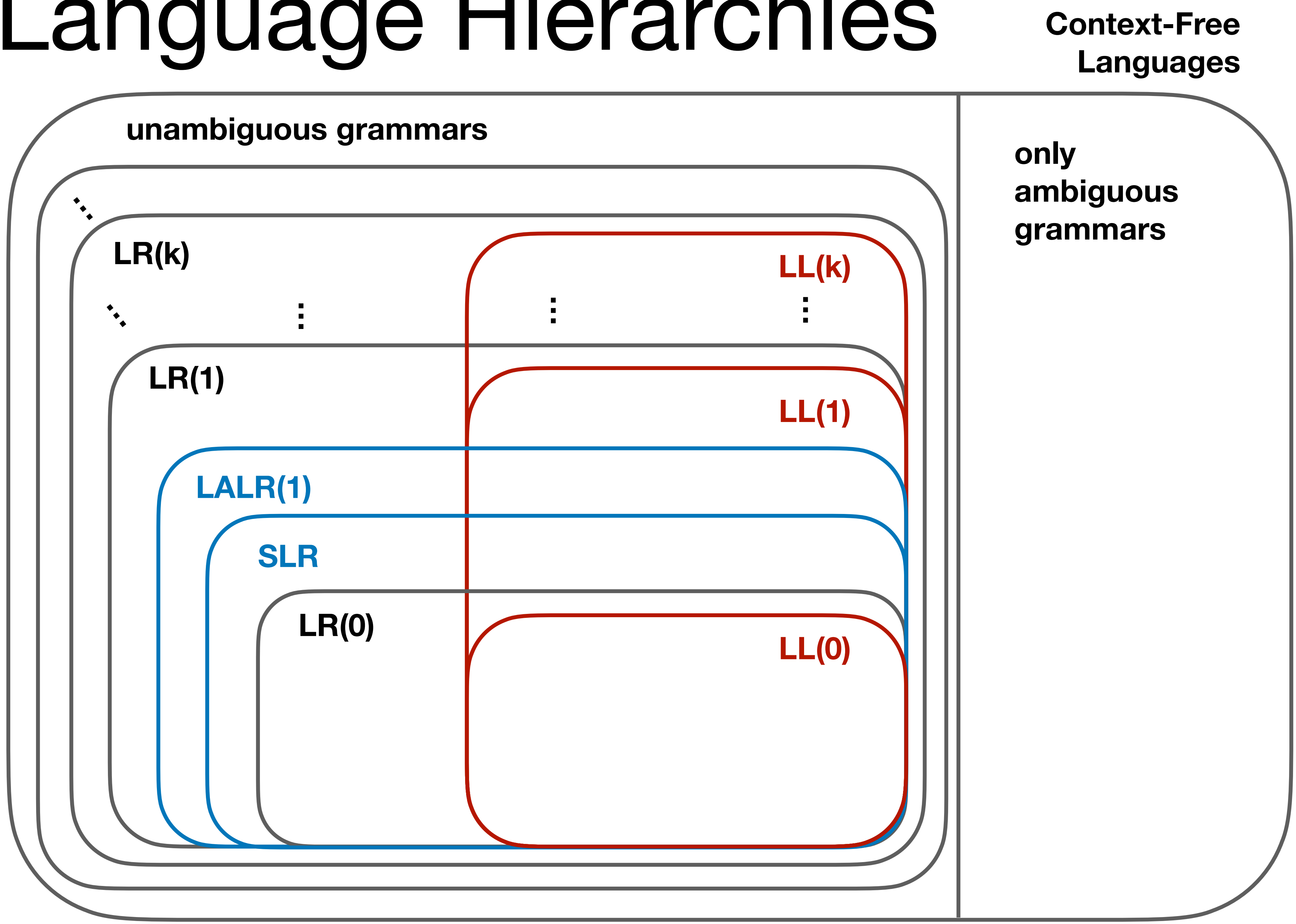
to produce

$$[A ::= x\ .\ y,\ ab]$$

LALR(1) vs. LR(1)

- LALR(1) tables can have many fewer states than LR(1)
  - somewhat surprising result — will actually have the same number of states as SLR parsers, even though LALR(1) is more powerful, because of more fine-grained lookahead information in states
- LALR(1) may have reduce conflicts where LR(1) would not (but in practice this doesn't happen often)
- Most practical bottom-up parser generator tools use LALR(1) parser construction (e.g. yacc, bison, CUP, …)

Language Hierarchies



Next Week

- Lecture
  - ASTs & Visitor Pattern
  - LL(k) Parsing — Top-Down parser generators
  - Recursive Descent Parsers
    - What to do if you want a parser in a hurry
- Sections Next Week
  - AST Construction — What your parser actually does!
  - Visitor Pattern details — how to traverse ASTs for further processing (in type checking, code gen, etc.)