

# CSE P 501 – Compilers

LR Parser Construction

Hal Perkins

Spring 2018

# Agenda

- LR(0) state construction
- FIRST, FOLLOW, and nullable
- Variations: SLR, LR(1), LALR

# LR State Machine

- Idea: Build a DFA that recognizes handles
  - Language generated by a CFG is generally not regular, but
  - Language of viable prefixes for a CFG is regular
    - So a DFA can be used to recognize handles
  - LR Parser reduces when DFA accepts a handle

# Prefixes, Handles, &c (review)

- If  $S$  is the start symbol of a grammar  $G$ ,
  - If  $S \Rightarrow^* \alpha$  then  $\alpha$  is a *sentential form* of  $G$
  - $\gamma$  is a *viable prefix* of  $G$  if there is some derivation  $S \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm}^* \alpha \beta w$  and  $\gamma$  is a prefix of  $\alpha \beta$ .
  - The occurrence of  $\beta$  in  $\alpha \beta w$  is a *handle* of  $\alpha \beta w$
- An *item* is a marked production (a  $\cdot$  at some position in the right hand side)
  - $[A ::= \cdot X Y]$   $[A ::= X \cdot Y]$   $[A ::= X Y \cdot]$

# Building the LR(0) States

- Example grammar

$S' ::= S \$$

$S ::= ( L )$

$S ::= x$

$L ::= S$

$L ::= L , S$

- We add a production  $S'$  with the original start symbol followed by end of file ( $\$$ )
  - We accept if we reach the end of this production
- Question: What language does this grammar generate?

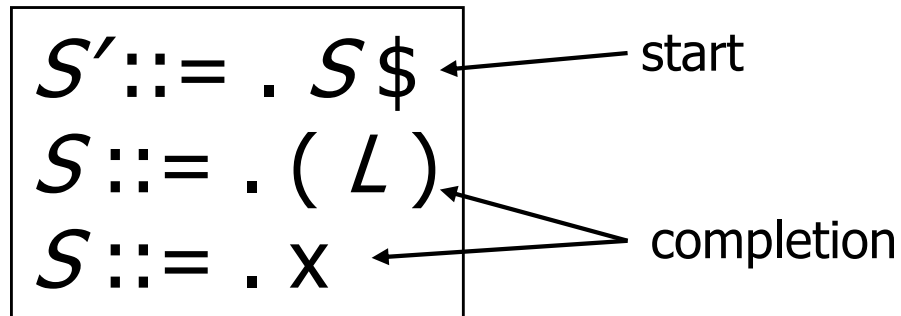
# Start of LR Parse

0.  $S' ::= S \$$
1.  $S ::= ( L )$
2.  $S ::= x$
3.  $L ::= S$
4.  $L ::= L , S$

- Initially
  - Stack is empty
  - Input is the right hand side of  $S'$ , i.e.,  $S \$$
  - Initial configuration is  $[S' ::= . S \$]$
  - But, since position is just before  $S$ , we are also just before anything that can be derived from  $S$

# Initial state

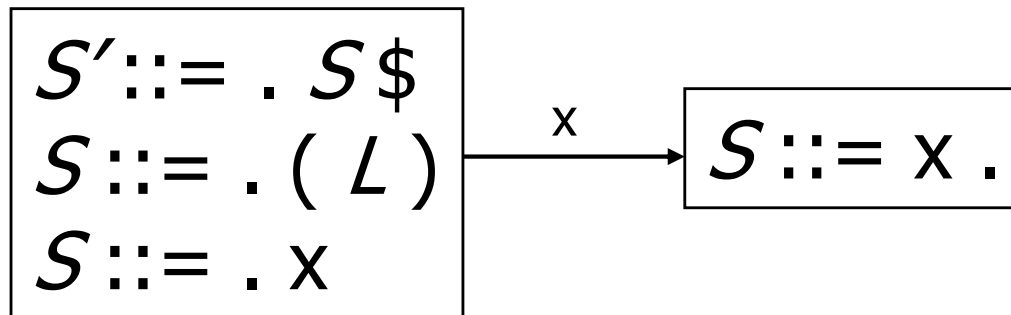
0.  $S' ::= S \$$
1.  $S ::= ( L )$
2.  $S ::= x$
3.  $L ::= S$
4.  $L ::= L , S$



- A state is just a set of items
  - Start: an initial set of items
  - Completion (or closure): additional productions whose left hand side appears to the right of the dot in some item already in the state

# Shift Actions (1)

0.  $S' ::= S \$$
1.  $S ::= ( L )$
2.  $S ::= x$
3.  $L ::= S$
4.  $L ::= L , S$

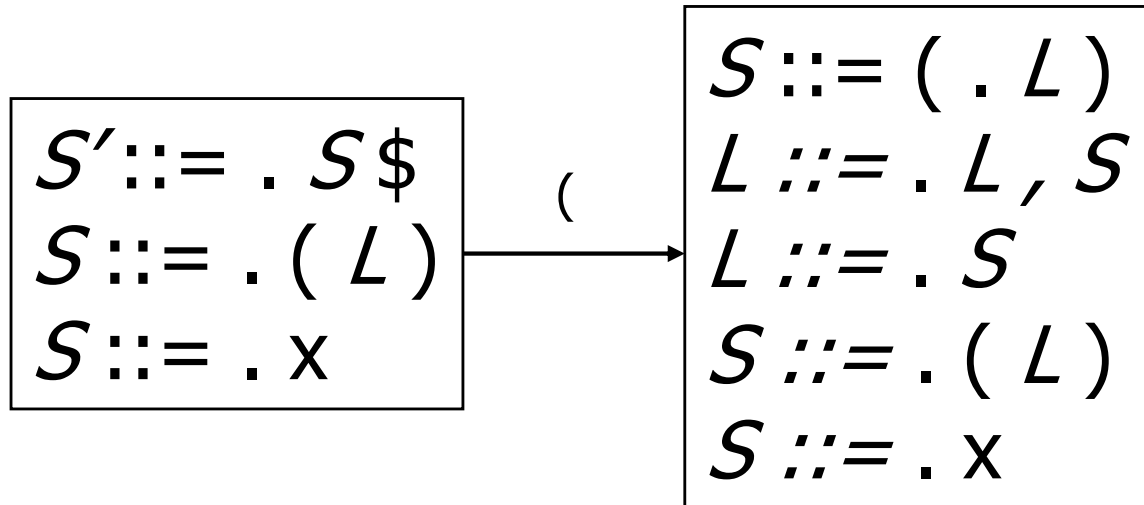


- To shift past the  $x$ , add a new state with appropriate item(s), including their closure
  - In this case, a single item; the closure adds nothing
  - This state will lead to a reduction since no further shift is possible



# Shift Actions (2)

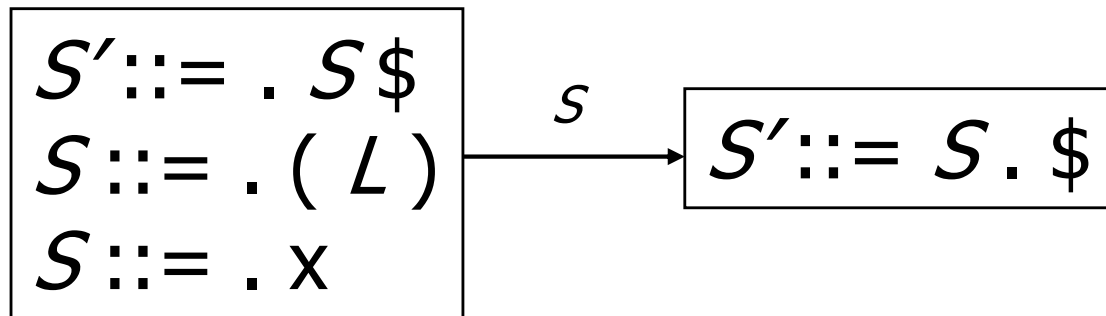
0.  $S' ::= S \$$
1.  $S ::= ( L )$
2.  $S ::= x$
3.  $L ::= S$
4.  $L ::= L , S$



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

# Goto Actions

0.  $S' ::= S \$$
1.  $S ::= ( L )$
2.  $S ::= x$
3.  $L ::= S$
4.  $L ::= L , S$



- Once we reduce  $S$ , we'll pop the rhs from the stack exposing the first state. Add a *goto* transition on  $S$  for this.

# Basic Operations

- *Closure* ( $S$ )
  - Adds all items implied by items already in  $S$
- *Goto* ( $I, X$ )
  - $I$  is a set of items
  - $X$  is a grammar symbol (terminal or non-terminal)
  - *Goto* moves the dot past the symbol  $X$  in all appropriate items in set  $I$

# Closure Algorithm

- *Closure* ( $S$ ) =
  - repeat
    - for any item  $[A ::= \alpha . B \beta]$  in  $S$ 
      - for all productions  $B ::= \gamma$ 
        - add  $[B ::= . \gamma]$  to  $S$
  - until  $S$  does not change
  - return  $S$
- Classic example of a fixed-point algorithm

# Goto Algorithm

- $Goto(I, X) =$ 
  - set  $new$  to the empty set
  - for each item  $[A ::= \alpha . X \beta]$  in  $I$ 
    - add  $[A ::= \alpha X . \beta]$  to  $new$
  - return  $Closure(new)$
- This may create a new state, or may return an existing one

# LR(0) Construction

- First, augment the grammar with an extra start production  $S' ::= S \$$
- Let  $T$  be the set of states
- Let  $E$  be the set of edges
- Initialize  $T$  to *Closure* (  $[S' ::= . S \$]$  )
- Initialize  $E$  to empty

# LR(0) Construction Algorithm

repeat

  for each state  $I$  in  $T$

    for each item  $[A ::= \alpha . X \beta]$  in  $I$

      Let  $new$  be  $Goto(I, X)$

      Add  $new$  to  $T$  if not present

      Add  $I \xrightarrow{X} new$  to  $E$  if not present

until  $E$  and  $T$  do not change in this iteration

- Footnote: For symbol  $\$$ , we don't compute  $goto(I, \$)$ ; instead, we make this an *accept* action.

# Example: States for

0.  $S' ::= S\$$
1.  $S ::= (L)$
2.  $S ::= x$
3.  $L ::= S$
4.  $L ::= L, S$



# Building the Parse Tables (1)

- For each edge  $I \xrightarrow{X} J$ 
  - if  $X$  is a terminal, put  $sj$  in column  $X$ , row  $I$  of the action table (shift to state  $j$ )
  - If  $X$  is a non-terminal, put  $gj$  in column  $X$ , row  $I$  of the goto table (go to state  $j$ )

# Building the Parse Tables (2)

- For each state  $I$  containing an item  $[S' ::= S . \$]$ , put *accept* in column  $\$$  of row  $I$
- Finally, for any state containing  $[A ::= \gamma .]$  put action *rn* (reduce) in every column of row  $I$  in the table, where  $n$  is the *production* number (*not* a state number)

# Example: Tables for

0.  $S' ::= S\$$
1.  $S ::= (L)$
2.  $S ::= x$
3.  $L ::= S$
4.  $L ::= L, S$

# Where Do We Stand?

- We have built the LR(0) state machine and parser tables
  - No lookahead yet
  - Different variations of LR parsers add lookahead information, but basic idea of states, closures, and edges remains the same
- A grammar is LR(0) if its LR(0) state machine (equiv. parser tables) has no shift-reduce or reduce-reduce conflicts.

# A Grammar that is not LR(0)

- Build the state machine and parse tables for a simple expression grammar

$$S ::= E \$$$

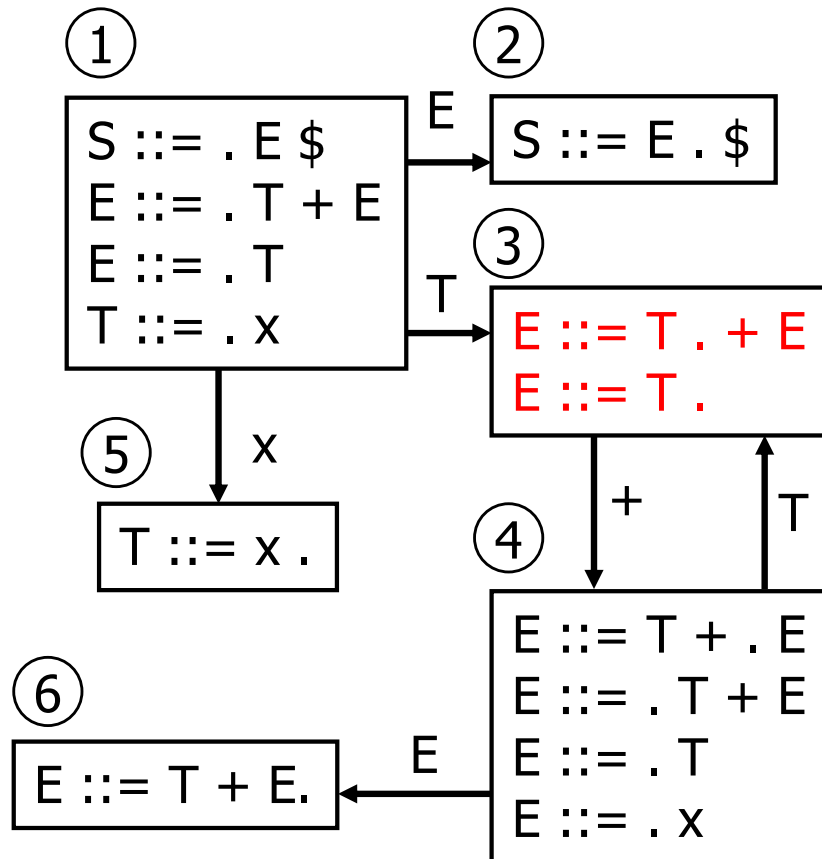
$$E ::= T + E$$

$$E ::= T$$

$$T ::= x$$

# LR(0) Parser for

0.  $S ::= E \$$
1.  $E ::= T + E$
2.  $E ::= T$
3.  $T ::= x$



	x	+	\$	E	T
1	s5			g2	G3
2			acc		
3	r2	s4,r2	r2		
4	s5			g6	G3
5	r3	r3	r3		
6	r1	r1	r1		

- State 3 is has two possible actions on +
  - shift 4, or reduce 2
- $\therefore$  Grammar is not LR(0)

# How can we solve conflicts like this?

- Idea: look at the next symbol after the handle before deciding whether to reduce
- Easiest: SLR – Simple LR. Reduce only if next input terminal symbol could follow the nonterminal on the left of the production in some possible derivation(s)
- More complex: LR and LALR. Store lookahead symbols in items to keep track of what can follow a particular instance of a reduction
  - LALR used by YACC/Bison/CUP; we won't examine in detail
  - see your favorite compiler book for explanations

# SLR Parsers

- Idea: Use information about what can follow a non-terminal to decide if we should perform a reduction; don't reduce if the next input symbol can't follow the resulting non-terminal
- We need to be able to compute  $\text{FOLLOW}(A)$  – the set of symbols that can follow  $A$  in any possible derivation
  - i.e.,  $t$  is in  $\text{FOLLOW}(A)$  if any derivation contains  $At$
  - To compute this, we need to compute  $\text{FIRST}(\gamma)$  for strings  $\gamma$  that can follow  $A$



# Calculating FIRST( $\gamma$ )

- Sounds easy... If  $\gamma = X Y Z$ , then FIRST( $\gamma$ ) is FIRST( $X$ ), right?
  - But what if we have the rule  $X ::= \epsilon$ ?
  - In that case, FIRST( $\gamma$ ) includes anything that can follow  $X$ , i.e. FOLLOW( $X$ ), which includes FIRST( $Y$ ) and, if  $Y$  can derive  $\epsilon$ , FIRST( $Z$ ), and if  $Z$  can derive  $\epsilon$ , ...
  - So computing FIRST and FOLLOW involves knowing FIRST and FOLLOW for other symbols, as well as which ones can derive  $\epsilon$ .

# FIRST, FOLLOW, and nullable

- **nullable( $X$ )** is true if  $X$  can derive the empty string
- Given a string  $\gamma$  of terminals and non-terminals, **FIRST( $\gamma$ )** is the set of terminals that can begin any strings derived from  $\gamma$ 
  - For SLR we only need this for single terminal or non-terminal symbols, not arbitrary strings  $\gamma$
- **FOLLOW( $X$ )** is the set of terminals that can immediately follow  $X$  in some derivation
- All three of these are computed together

# Computing FIRST, FOLLOW, and nullable (1)

- Initialization
  - set FIRST and FOLLOW to be empty sets
  - set nullable to false for all non-terminals
  - set FIRST[a] to a for all terminal symbols a
- Repeatedly apply four simple observations to update these sets
  - Stop when there are no further changes
  - Another fixed-point algorithm

# Computing FIRST, FOLLOW, and nullable (2)

```
repeat
  for each production  $X := Y_1 Y_2 \dots Y_k$ 
    if  $Y_1 \dots Y_k$  are all nullable (or if  $k = 0$ )
      set nullable[X] = true
    for each  $i$  from 1 to  $k$  and each  $j$  from  $i + 1$  to  $k$ 
      if  $Y_1 \dots Y_{i-1}$  are all nullable (or if  $i = 1$ )
        add FIRST[ $Y_i$ ] to FIRST[X]
      if  $Y_{i+1} \dots Y_k$  are all nullable (or if  $i = k$ )
        add FOLLOW[X] to FOLLOW[ $Y_i$ ]
      if  $Y_{i+1} \dots Y_{j-1}$  are all nullable (or if  $i+1=j$ )
        add FIRST[ $Y_j$ ] to FOLLOW[ $Y_i$ ]
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

Y

Z

# LR(0) Reduce Actions (review)

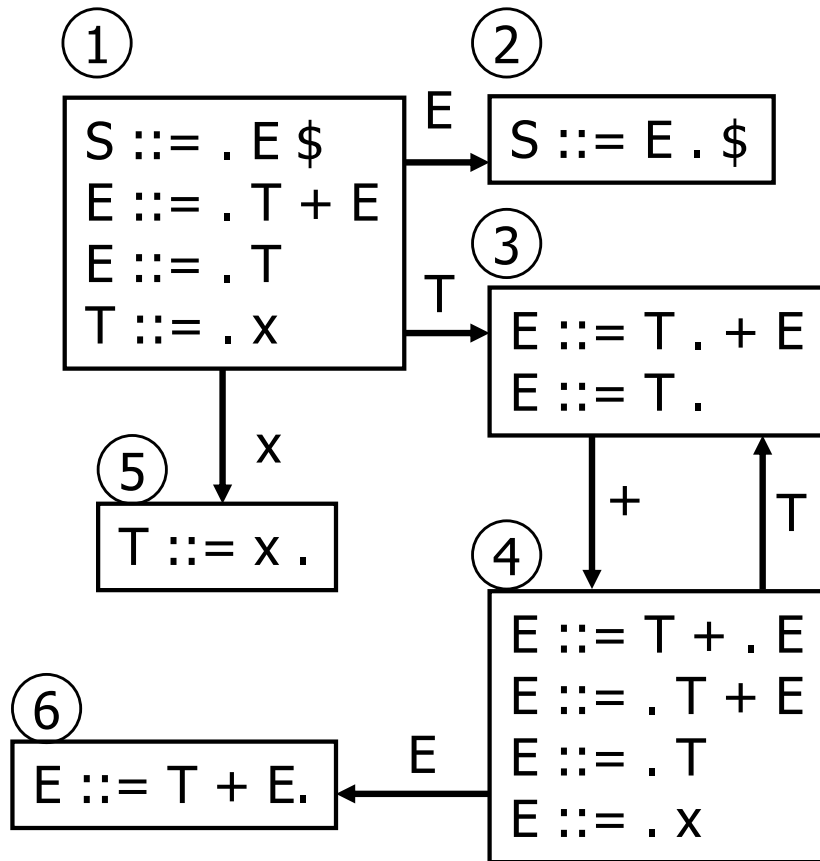
- In a LR(0) parser, if a state contains a reduction, it is unconditional regardless of the next input symbol
- Algorithm:
  - Initialize  $R$  to empty
  - for each state  $I$  in  $T$ 
    - for each item  $[A ::= \alpha .]$  in  $I$ 
      - add  $(I, A ::= \alpha)$  to  $R$

# SLR Construction

- This is identical to LR(0) – states, etc., except for the calculation of reduce actions
- Algorithm:
  - Initialize  $R$  to empty
  - for each state  $I$  in  $T$ 
    - for each item  $[A ::= \alpha .]$  in  $I$ 
      - for each terminal  $a$  in  $\text{FOLLOW}(A)$ 
        - add  $(I, a, A ::= \alpha)$  to  $R$
  - i.e., reduce  $\alpha$  to  $A$  in state  $I$  only on lookahead  $a$

# SLR Parser for

0.  $S ::= E \$$
1.  $E ::= T + E$
2.  $E ::= T$
3.  $T ::= x$



	x	+	\$	E	T
1	s5			g2	g3
2			acc		
3	r2	s4,r2	r2		
4	s5			g6	g3
5	r3	r3	r3		
6	r1	r1	r1		



# On To LR(1)

- Many practical grammars are SLR
- LR(1) is more powerful yet
- Similar construction, but notion of an item is more complex, incorporating lookahead information

# LR(1) Items

- An LR(1) item  $[A ::= \alpha . \beta, a]$  is
  - A grammar production ( $A ::= \alpha\beta$ )
  - A right hand side position (the dot)
  - A lookahead symbol ( $a$ )
- Idea: This item indicates that  $\alpha$  is the top of the stack and the next input is derivable from  $\beta a$ .
- Full construction: see the book

# LR(1) Tradeoffs

- LR(1)
  - Pro: extremely precise; largest set of grammars
  - Con: potentially **very** large parse tables with many states

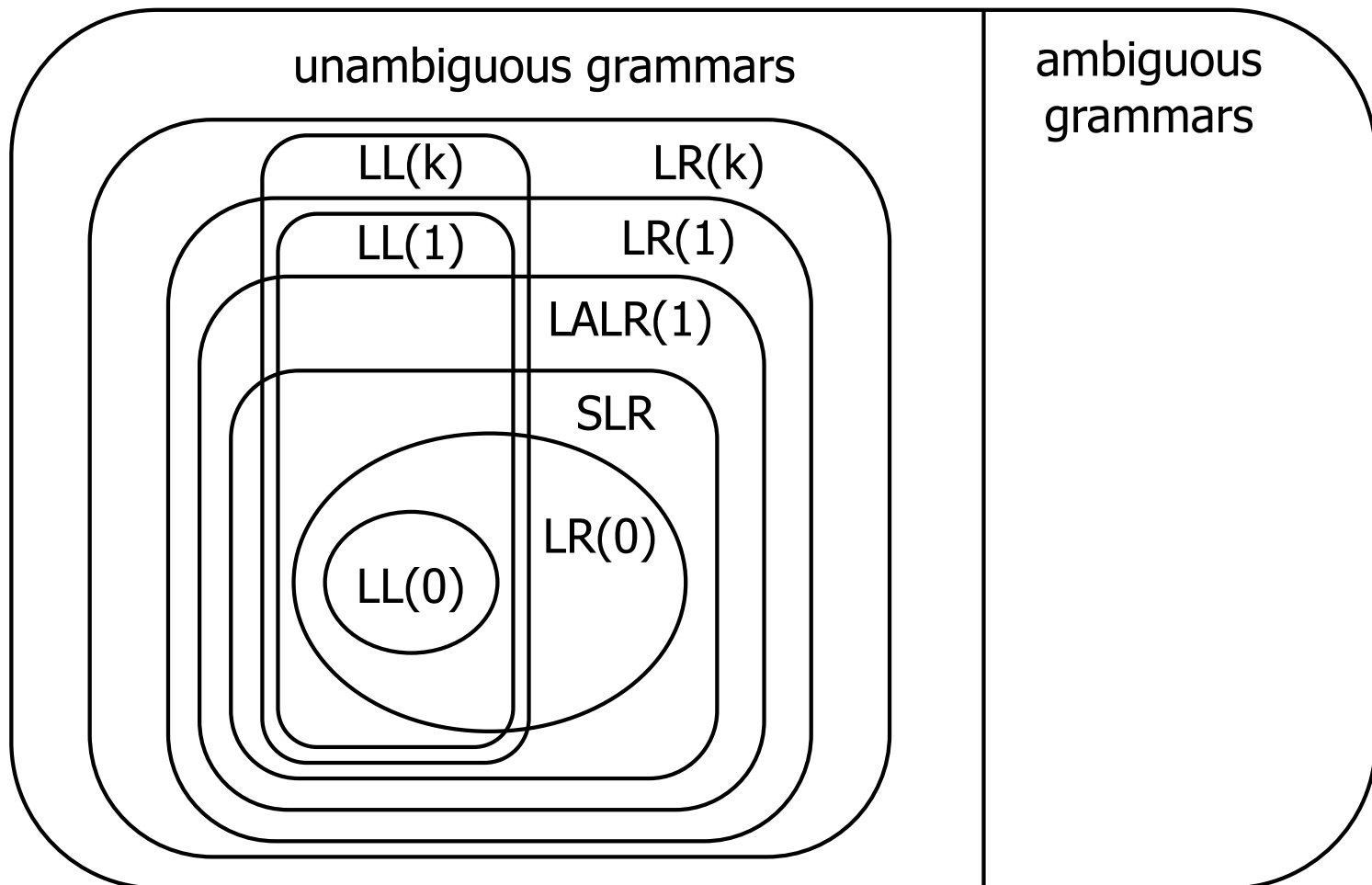
# LALR(1)

- Variation of LR(1), but merge any two states that differ only in lookahead
  - Example: these two would be merged
    - [A ::= x . , a]
    - [A ::= x . , b]

# LALR(1) vs LR(1)

- LALR(1) tables can have many fewer states than LR(1)
  - Somewhat surprising result: will actually have same number of states as SLR parsers, even though LALR(1) is more powerful
  - After the merge step, acts like SLR parser with “smarter” FOLLOW sets (can be specific to particular handles)
- LALR(1) may have reduce conflicts where LR(1) would not (but in practice this doesn't happen often)
- Most practical bottom-up parser tools are LALR(1) (e.g., yacc, bison, CUP, ...)

# Language Heirarchies



# Coming Attractions

Rest of Parsing...

- LL(k) Parsing – Top-Down
- Recursive Descent Parsers
  - What you can do if you want a parser in a hurry

Then...

- AST construction – what do do while you parse!
- Visitor Pattern – how to traverse ASTs for further processing (type checking, code generation, ...)