# CSE P 501 – Compilers

LR Parser Construction

Hal Perkins

Autumn 2025

# Administrivia (1)

- HW1 sample solutions handed out during break today. Might be of some use finishing up scanner project. HW1 grades/feedback out now.

- Scanners due tomorrow night. Last-minute hints / checklist
  - Must read MiniJava overview as well as scanner assignment & reread *again* when you think you're "done"
    - Be sure to implement both kinds of comments
    - Be sure to look carefully at MiniJava grammar to discover tokens
    - Anything "quoted" in the MiniJava project grammar should be treated as a reserved word (token) in MiniJava, even if it's not that in full Java (e.g., "System.out.println" (!) )
  - Be sure you can handle comments at end of file, also files with and without newlines at the end (and with and without comments at the end!)

# Administrivia (2)

- Scanners (cont.)
  - Scanner should continue after "invalid input character" errors
    - Error handling for invalid characters is already in the starter code – use it!
  - Be sure to terminate with correct System.exit code (0=ok, 1=errors)
    - Don't get "creative" with the specs – compiler must work as required
  - Take advantage of JFlex regexp operations that go beyond basic regexps presented in class and on hw1 if they are useful
  - Don't implement the parser just yet – plenty of time for that…
  - Reminder: you have a partner(!) – be sure to take advantage
    - Discussion board/email: never "I have a question" or "I am confused"
    - Rather: "We are confused" or "We have a question" ☺

# Administrivia (3)

- We've got one student who still needs a partner for the project. Best solution is to find a group that would be willing to take on a 3rd partner.

  - If your group can help out, please reply (anonymously if you wish) to the thread on the discussion board or send email to csep501-staff [at] cs. We'll send email to get everyone in touch and figure out how to make this work. Thanks!

# 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 $S =>^* \alpha$ then $\alpha$ is a *sentential form* of *G*
  - $\gamma$ is a *viable prefix* of *G* if there is some derivation
    $S =>^*_{rm} \alpha A w =>_{rm} \alpha \beta w$ and $\gamma$ is a prefix of $\alpha \beta$.
    - These are the strings that can appear on the LR parser stack
  - The occurrence of $\beta$ in $\alpha \beta w$ is the right side of a *handle* of $\alpha \beta w$

- An *item* is a marked production (a . at some position in the right hand side)
  - [*A* ::= . *X Y* ]   [*A* ::= *X* . *Y* ]   [*A* ::= *X Y* . ]

# 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?
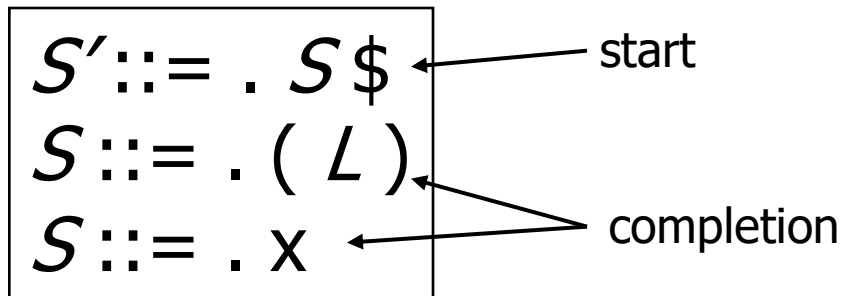
# Start of LR Parse

- Initially
  - Stack is empty
    - (except for start state number usually)
  - 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 $S$
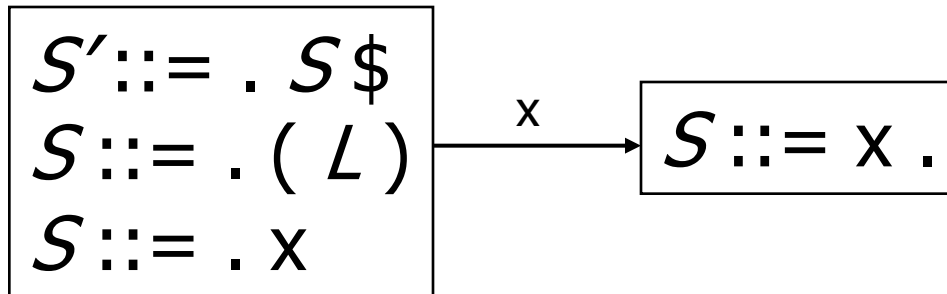
# Initial state

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

$S' ::= .\ S\ \$$      ← start
$S ::= .\ (\ L\ )$
$S ::= .\ x$      ← completion

- A state is just a set of items
  - Start: an initial set of items
  - Completion (or closure): additional productions whose left-hand side nonterminal appears immediately following a dot in some item already in the state
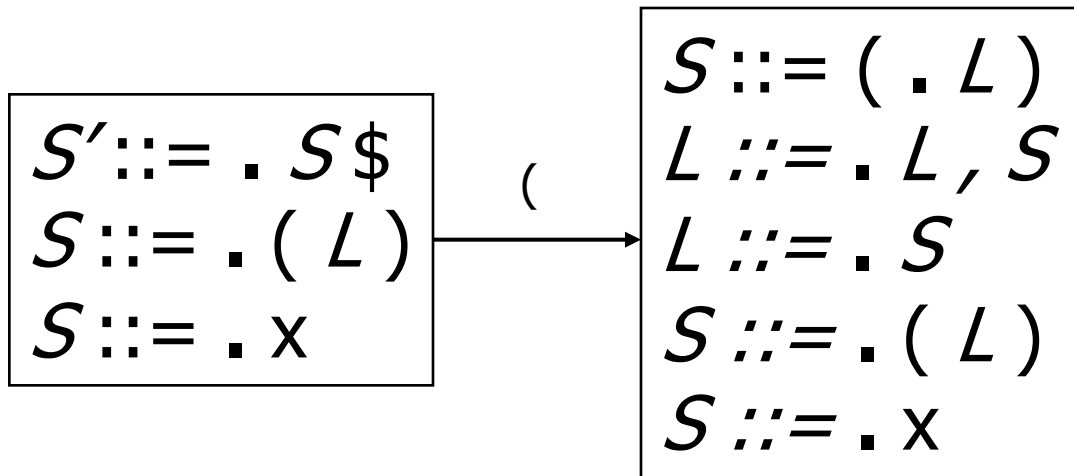
# Shift Actions (1)

$$
\boxed{\begin{aligned}
&S' ::= .\ S \$ \\
&S ::= .\ ( L ) \\
&S ::= .\ x
\end{aligned}}
\xrightarrow{\ x\ }
\boxed{S ::= x\ .}
$$

- 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
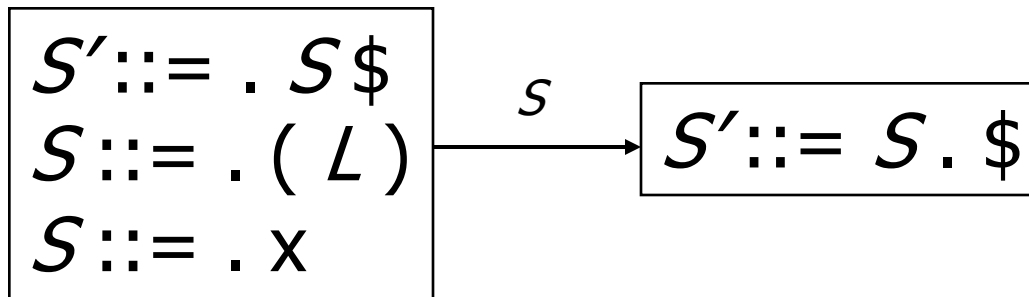  - This state will lead to a reduction since no further shift is possible

# Shift Actions (2)

$$S' ::= \textbf{.} \, S \, \$$$
$$S ::= \textbf{.} \, ( \, L \, )$$
$$S ::= \textbf{.} \, x$$

$$\xrightarrow{\ (\ }$$

$$S ::= ( \textbf{.} \, L \, )$$
$$L ::= \textbf{.} \, L \, , \, S$$
$$L ::= \textbf{.} \, S$$
$$S ::= \textbf{.} \, ( \, L \, )$$
$$S ::= \textbf{.} \, x$$

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

# Goto Actions

$$S' ::= . S \$$$
$$S ::= . ( L )$$
$$S ::= . x$$

$\xrightarrow{S}$

$$S' ::= S . \$$$

- Once we reduce $S$, we'll pop the rhs from the stack exposing a previous state. Add a *goto* transition on $S$ for this (i.e., if we back up into this state having reduced a rhs to $S$, then we need a goto transition on $S$ to another state).

# 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* ( [$S'$ ::= . $S$ $] )
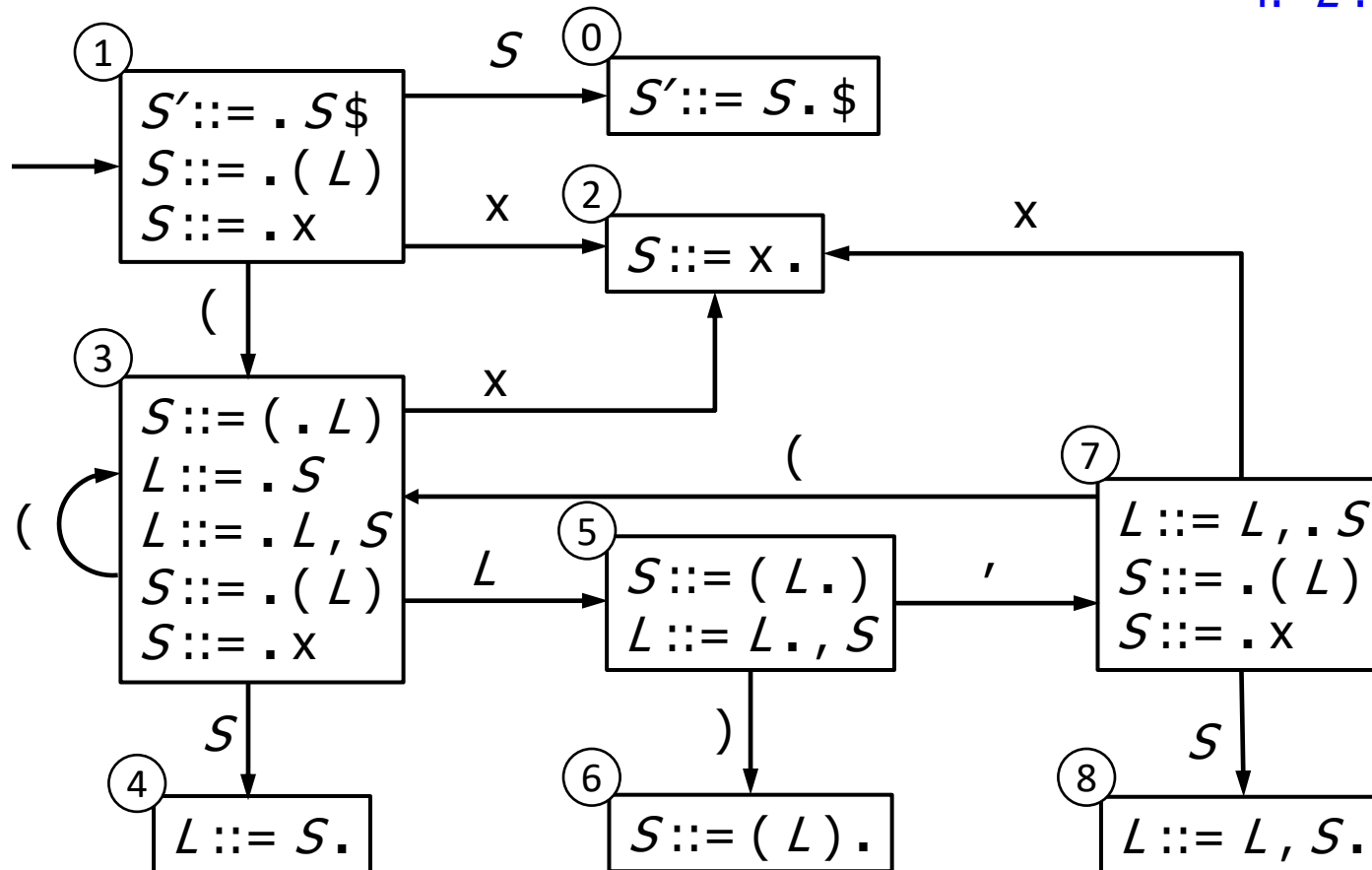
- Initialize $E$ to empty

# 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* ::= γ **.**] put action r*n* (reduce) in every
  column of row *I* in the table, where *n* is the
  *production* number (*not* a state number)
  - i.e., when it reaches this state, the DFA has
    discovered that *A* ::= γ is a *handle*, so the parser
    should reduce γ to *A*

# Example: Tables for

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

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

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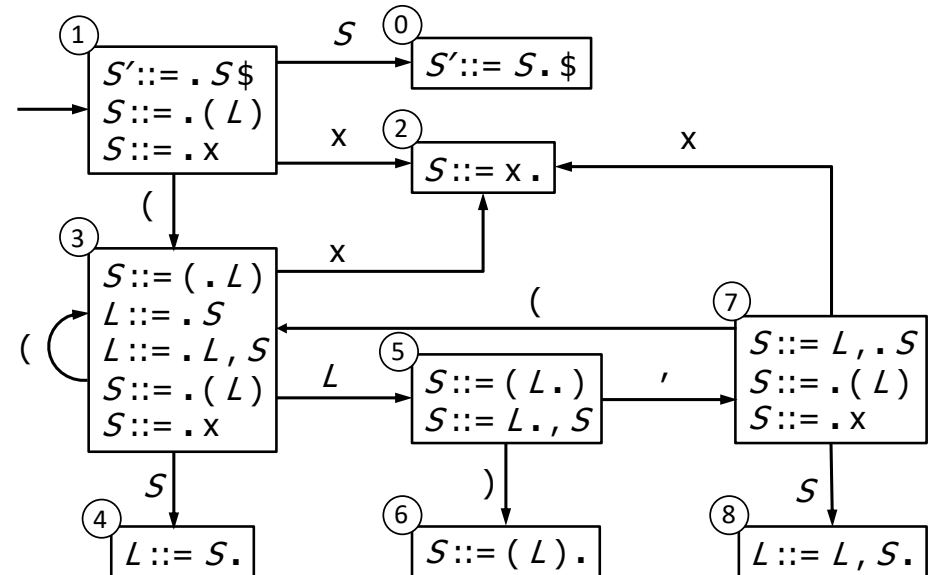
(0) $S' ::= S\,.\,\$$

(2) $S ::= x\,.$

(3)
$S ::= (\,.\,L\,)$
$L ::= .\,S$
$L ::= .\,L\,,\,S$
$S ::= .\,(\,L\,)$
$S ::= .\,x$

(5)
$S ::= (\,L\,.\,)$
$S ::= L\,.\,,\,S$

(7)
$S ::= L\,,\,.\,S$
$S ::= .\,(\,L\,)$
$S ::= .\,x$

(4) $L ::= S\,.$

(6) $S ::= (\,L\,)\,.$

(8) $L ::= L\,,\,S\,.$

# Example: Tables for

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

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 0 | | | | | acc | | |
| 1 | s3 | | s2 | | | g0 | |
| 2 | r2 | r2 | r2 | r2 | r2 | | |
| 3 | s3 | | s2 | | | g4 | g5 |
| 4 | r3 | r3 | r3 | r3 | r3 | | |
| 5 | | s6 | | s7 | | | |
| 6 | r1 | r1 | r1 | r1 | r1 | | |
| 7 | s3 | | s2 | | | g8 | |
| 8 | r4 | r4 | r4 | r4 | r4 | | |

(1)
$S' ::= . S\,\$$
$S ::= . ( L )$
$S ::= . x$

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

(2)
$S ::= x .$

(3)
$S ::= ( . L )$
$L ::= . S$
$L ::= . L , S$
$S ::= . ( L )$
$S ::= . x$

(5)
$S ::= ( L . )$
$S ::= L . , S$

(7)
$S ::= L , . S$
$S ::= . ( L )$
$S ::= . x$

(4)
$L ::= S .$

(6)
$S ::= ( L ) .$

(8)
$L ::= L , S .$
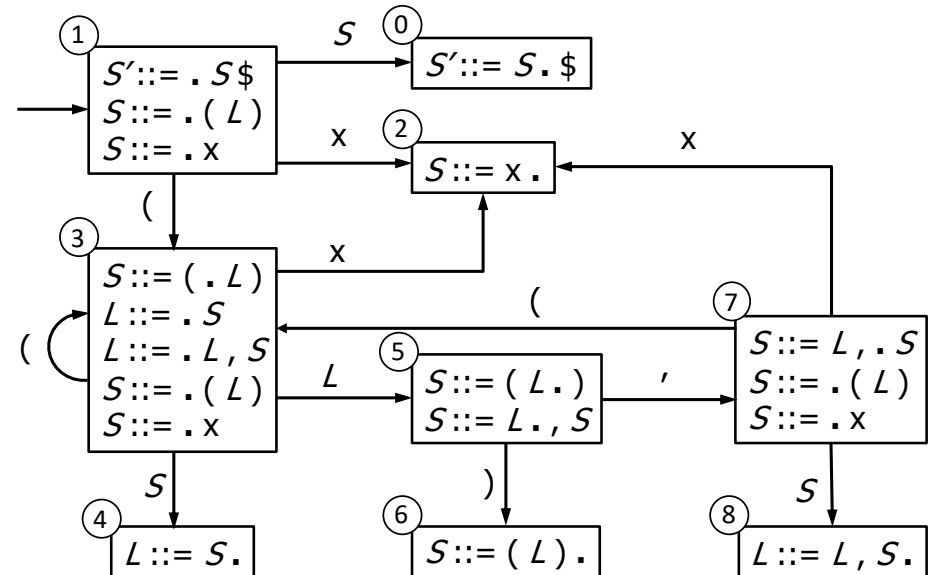
# Where Do We Stand?

- 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
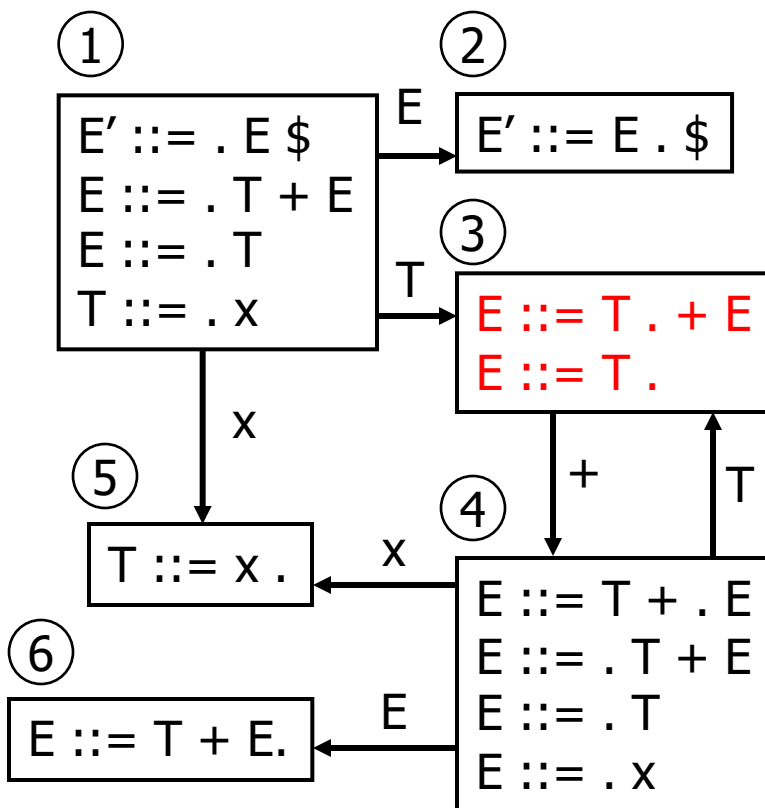
    $E' ::= E \$$

    $E ::= T + E$

    $E ::= T$

    $T ::= x$

# LR(0) Parser for

0. $E' ::= E \$$
1. $E ::= T + E$
2. $E ::= T$
3. $T ::= x$

**State 1** (E' ::= . E $, E ::= . T + E, E ::= . T, T ::= . x)
**State 2** (E' ::= E . $)
**State 3** (E ::= T . + E, E ::= T .)
**State 5** (T ::= x .)
**State 4** (E ::= T + . E, E ::= . T + E, E ::= . T, E ::= . x)
**State 6** (E ::= T + E.)

| | 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
- ∴ 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 resulting nonterminal
  - Suppose we've reached [$A$ ::= $\beta$ **.**] and the next input is x
  - Don't reduce unless $Ax$ can appear in some sentential form
    - This is the 🔑 🗝 idea!

- 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

# 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

- To decide, for each non-terminal $A$, we need to know FOLLOW($A$) – the set of terminal symbols that can follow $A$ in some possible derivation

  - i.e., t is in FOLLOW($A$) if there is some possible derivation that contains $At$

  - To compute this, we need to compute 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 $X ::= \varepsilon$?

    – In that case, FIRST($\gamma$) includes anything that can follow $X$, i.e. FOLLOW($X$), which includes FIRST($Y$) and, if $Y$ can derive $\varepsilon$, FIRST($Z$), and if $Z$ can derive $\varepsilon$, …

    – So computing FIRST and FOLLOW involves knowing FIRST and FOLLOW for other symbols, as well as which ones can derive $\varepsilon$

# 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

- Footnote: Textbook doesn't use a separate nullable(X) attribute, instead it indicates nullable by including ε in FIRST(X).  Both will wind up with same results, but one or the other might be easier to follow, so to speak.

# 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 $X := Y_1\ Y_2\ Y_3\ ...\ Y_{k-2}\ Y_{k-1}\ Y_k$
      if $Y_1\ ...\ Y_k$ are all nullable (or if $k = 0$)
        set nullable[$X$] = true    ①
      for each $i$ from 1 to k and each $j$ from $i$ +1 to $k$
        if $Y_1\ ...\ Y_{i-1}$ are all nullable (or if $i = 1$)
          add FIRST[$Y_i$] to FIRST[$X$]    ②
        if $Y_{i+1}\ ...\ Y_k$ are all nullable (or if $i = k$ )
          add FOLLOW[$X$] to FOLLOW[$Y_i$]    ③
        if $Y_{i+1}\ ...\ Y_{j-1}$ are all nullable (or if i+1=j)
          add FIRST[$Y_j$] to FOLLOW[$Y_i$]    ④
  Until FIRST, FOLLOW, and nullable do not change
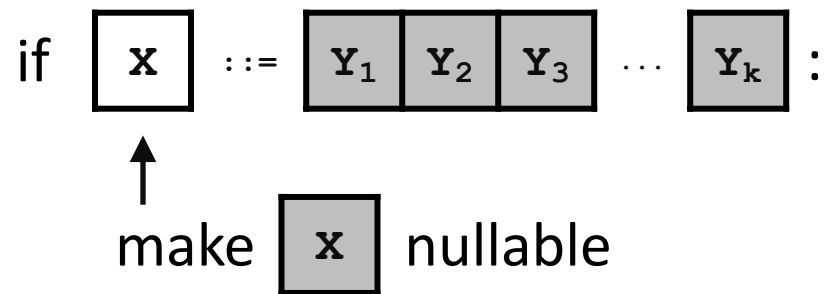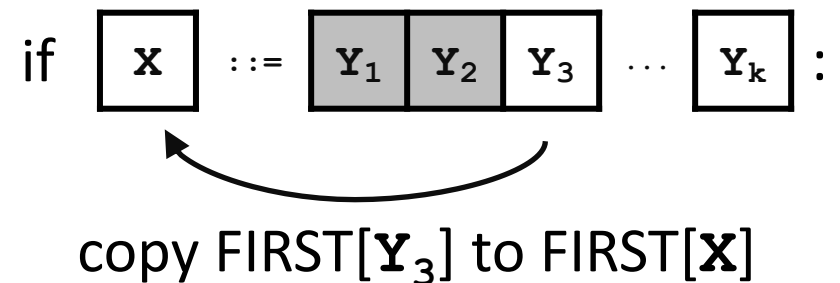
# Computing FIRST, FOLLOW, & nullable (3)

$\boxed{Y}$ = nullable

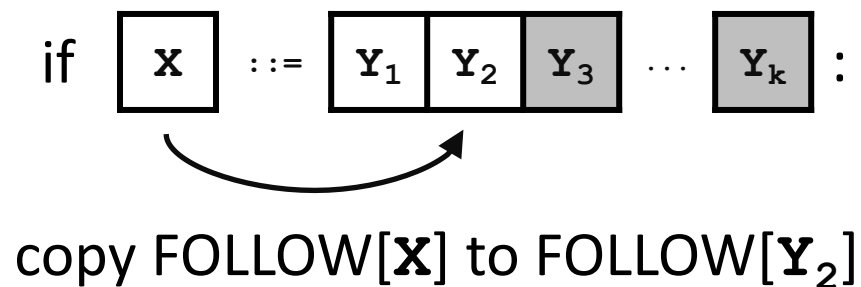**①**

if $\boxed{X}$ ::= $\boxed{Y_1}$ $\boxed{Y_2}$ $\boxed{Y_3}$ $\cdots$ $\boxed{Y_k}$ :

make $\boxed{x}$ nullable

**②**

if $\boxed{X}$ ::= $\boxed{Y_1}$ $\boxed{Y_2}$ $\boxed{Y_3}$ $\cdots$ $\boxed{Y_k}$ :

copy FIRST[$Y_3$] to FIRST[$X$]

**③**

if $\boxed{X}$ ::= $\boxed{Y_1}$ $\boxed{Y_2}$ $\boxed{Y_3}$ $\cdots$ $\boxed{Y_k}$ :

copy FOLLOW[$X$] to FOLLOW[$Y_2$]

**④**

if $\boxed{X}$ ::= $\boxed{Y_1}$ $\boxed{Y_2}$ $\boxed{Y_3}$ $\cdots$ $\boxed{Y_k}$ :

copy FIRST[$Y_3$] to FOLLOW[$Y_1$]
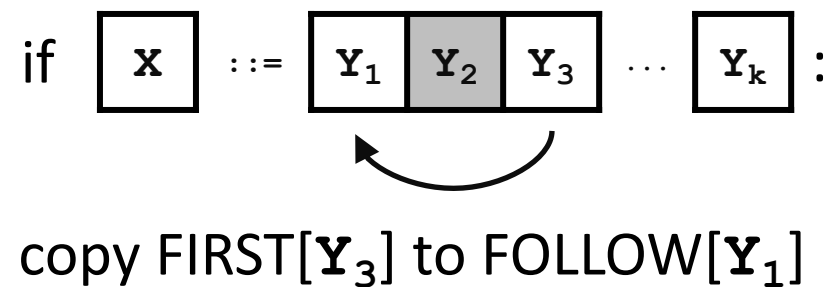
# Example (initial)

- Grammar

| | nullable | FIRST | FOLLOW |
|---|---|---|---|
| *Z* ::= d | | | |
| *Z* ::= *X Y Z* | | | |
| *Y* ::= ε | | | |
| *Y* ::= c | | | |
| *X* ::= *Y* | | | |
| *X* ::= a | | | |

| | nullable | FIRST | FOLLOW |
|---|---|---|---|
| *X* | no | | |
| *Y* | no | | |
| *Z* | no | | |

# Example (final)

- Grammar

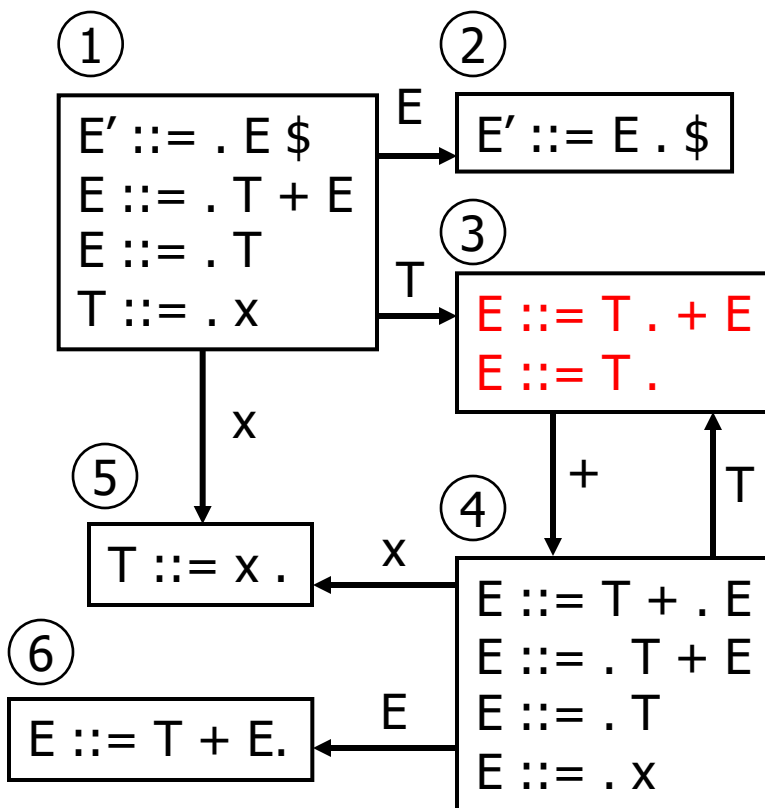|   | nullable | FIRST | FOLLOW |
|---|---|---|---|
| *Z* ::= d | | | |
| *Z* ::= *X Y Z* | *X*   ~~no~~ yes | a, c | a, c, d |
| *Y* ::= ε | | | |
| *Y* ::= c | *Y*   ~~no~~ yes | c | a, c, d |
| *X* ::= *Y* | | | |
| *X* ::= a | *Z*   no | a, c, d | |

# 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 FOLLOW(*A*)    ☞ new!

          add (*I*, a, *A* ::= $\alpha$) to *R*

    – i.e., reduce $\alpha$ to *A* in state *I* only on lookahead a

# SLR Parser for

0. E' ::= E $
1. E ::= T + E
2. E ::= T
3. T ::= x

① 
E' ::= . E $
E ::= . T + E
E ::= . T
T ::= . x

—E→ ②
E' ::= E . $

—T→ ③
E ::= T . + E
E ::= T .

—x→ ⑤
T ::= x .

④
E ::= T + . E
E ::= . T + E
E ::= . T
E ::= . x

—E→ ⑥
E ::= T + E.

| | 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 | | |

Ghost yellow = reductions omitted
in SLR parser because next terminal
is not in FOLLOW(non-terminal)

# 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

  – So lookahead information is associated with specific items rather than FOLLOW for a non-terminal regardless of specific context where that non-terminal appears in the derivation

# 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]

    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 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 Hierarchies
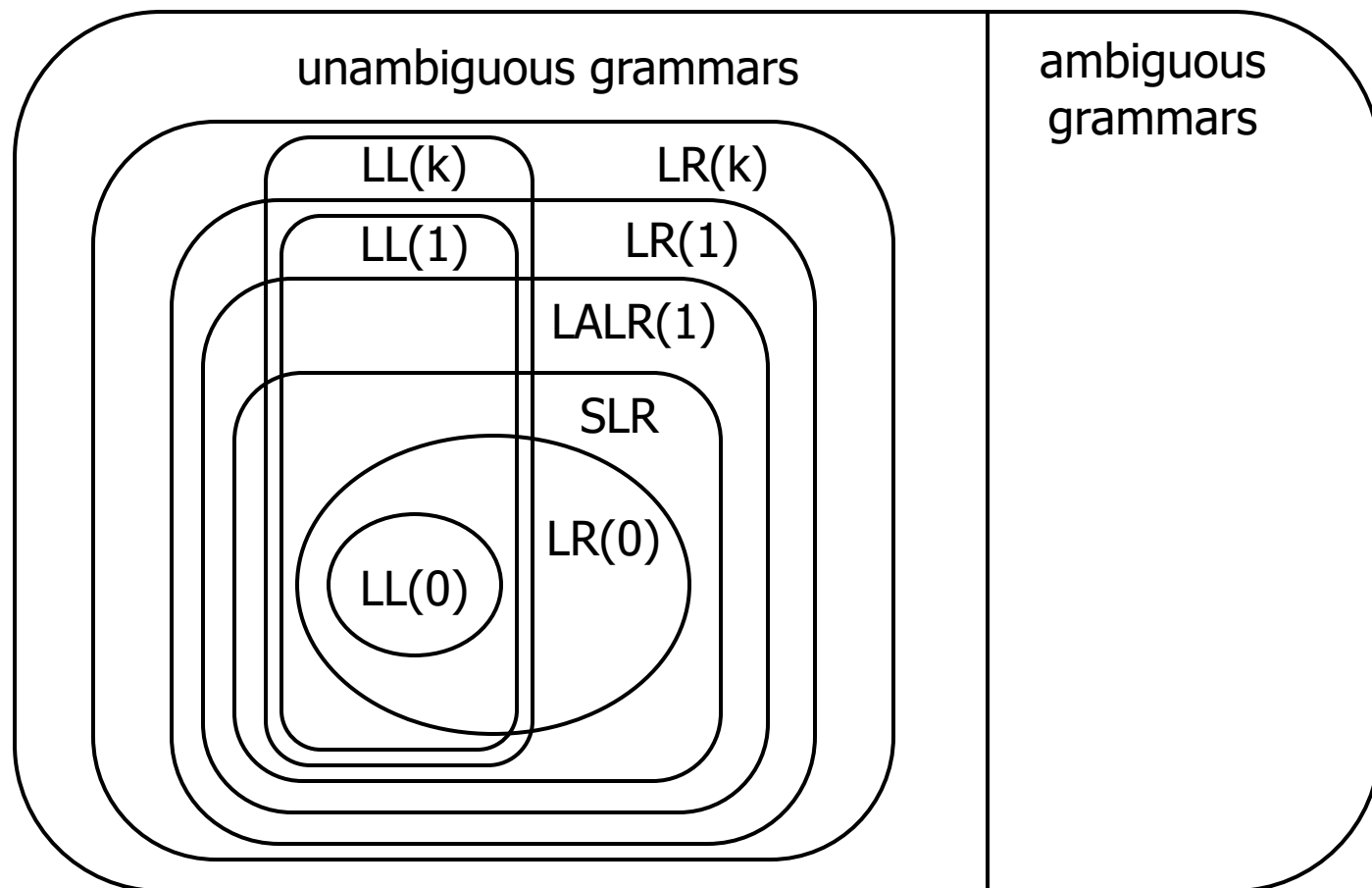
# Coming Attractions

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, …)