Bottom Up (Shift Reduce) Parsing
Bottom-Up Parsing

- A **bottom-up parser** creates the parse tree of the given input starting from leaves towards the root.

- A bottom-up parser tries to find the **right-most derivation** of the given input in the reverse order.

  \[ S \Rightarrow ... \Rightarrow \omega \quad \text{(the right-most derivation of } \omega) \]

  \[ \quad \leftarrow \quad \text{(the bottom-up parser finds the right-most derivation in the reverse order)} \]
Bottom Up Parsing

- LR Parsing
  - Also called “Shift-Reduce Parsing”

- Find a rightmost derivation
- Finds it in reverse order

- LR Grammars
  - Can be parsed with an LR Parser

- LR Languages
  - Can be described with LR Grammar
  - Can be parsed with an LR Parser
LR Parsing Techniques

- **LR Parsing**
  - Most General Approach

- **SLR**
  - Simpler algorithm, but not as general

- **LALR**
  - More complex, but saves space
LL vs. LR

• LR (shift reduce) is more powerful than LL (predictive parsing)

• Can detect a syntactic error as soon as possible.

• LR is difficult to do by hand (unlike LL) and
• LL accepts a much smaller set of grammars.
Rightmost Derivation

**Rules Used:**
- $E \rightarrow T$
- $T \rightarrow T \ast F$
- $F \rightarrow \text{id}$
- $T \rightarrow F$
- $F \rightarrow (E)$
- $E \rightarrow E + T$
- $T \rightarrow F$
- $F \rightarrow \text{id}$
- $E \rightarrow T$
- $T \rightarrow F$
- $F \rightarrow \text{id}$

**Right-Sentential Forms:**
- $E$
- $T$
- $T \ast F$
- $T \ast \text{id}$
- $F \ast \text{id}$
- $(E) \ast \text{id}$
- $(E + T) \ast \text{id}$
- $(E + F) \ast \text{id}$
- $(T + \text{id}) \ast \text{id}$
- $(F + \text{id}) \ast \text{id}$
- $(\text{id} + \text{id}) \ast \text{id}$
- $(E) + T$
- $T F$
- $F \text{id}$
- $(E + \text{id}) \ast \text{id}$
- $(T + \text{id}) \ast \text{id}$
- $(F + \text{id}) \ast \text{id}$
- $(\text{id} + \text{id}) \ast \text{id}$

**Diagram:**

```
E -> E + T
2. E -> T
3. T -> T * F
4. T -> F
5. F -> (E)
6. F -> id
```
Rightmost Derivation In reverse

Rules Used:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Right-Sentential Forms</th>
</tr>
</thead>
<tbody>
<tr>
<td>F → id</td>
<td>*(id + id) * id</td>
</tr>
<tr>
<td>T → F</td>
<td>*(F + id) * id</td>
</tr>
<tr>
<td>E → T</td>
<td>*(T + id) * id</td>
</tr>
<tr>
<td>F → id</td>
<td>*(E + id) * id</td>
</tr>
<tr>
<td>T → F</td>
<td>*(E + F) * id</td>
</tr>
<tr>
<td>E → E + T</td>
<td>*(E + T) * id</td>
</tr>
</tbody>
</table>

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → ( E )
6. F → id
LR parsing corresponds to rightmost derivation in reverse
Reduction

- A reduction step replaces a specific substring (matching the body of a production)

\[
\begin{align*}
(id + id) \ast id & \quad (E) \ast id \\
(F + id) \ast id & \quad F \ast id \\
(T + id) \ast id & \quad T \ast id \\
(E + id) \ast id & \quad T \ast F \\
(E + F) \ast id & \quad T \\
(E + T) \ast id & \quad E
\end{align*}
\]

- Reduction is the opposite of derivation
- Bottom up parsing is a process of reducing a string \( \omega \) to the start symbol \( S \) of the grammar
Shift-Reduce Parsing

• Bottom-up parsing is also known as **shift-reduce parsing** because its two main actions are shift and reduce.

• data structures: input-string and stack

• Operations
  – At each **shift** action, the current symbol in the input string is pushed to a stack.
  – At each **reduction** step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will replaced by the non-terminal at the left side of that production.
  – **Accept**: Announce successful completion of parsing
  – **Error**: Discover a syntax error and call error recovery
Shift Reduce Parsing Example

S → a T R e
T → T b c | b
R → d

Remaining input: a b b c d e

Rightmost derivation:
S → a T R e
→ a T d e
→ a T b c d e
→ a b b c d e
Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

→ Shift a

Remaining input: bbcde

Rightmost derivation:
S → a T R e
→ a T d e
→ a T b c d e
→ a b b c d e
Shift Reduce Parsing

Remaining input: bcde

→ Shift a, Shift b

Rightmost derivation:
\[
S \rightarrow a T R e \\
S \rightarrow a T R e \\
S \rightarrow a T R e \\
S \rightarrow a T R e \\
S \rightarrow a T R e
\]

\[
\rightarrow a b b c d e
\]
Shift Reduce Parsing

\[ S \rightarrow a \ T \ R \ e \]
\[ T \rightarrow T \ b \ c \ | \ b \]
\[ R \rightarrow d \]

\[ \text{Shift } a, \text{ Shift } b \]

\[ \text{Reduce } T \rightarrow b \]

Remaining input: \( b c d e \)

Rightmost derivation:
\[ S \rightarrow a \ T \ R \ e \]
\[ \rightarrow a \ T \ d \ e \]
\[ \rightarrow a \ T \ b \ c \ d \ e \]
\[ \rightarrow a \ b \ b \ c \ d \ e \]
Shift Reduce Parsing

$S \rightarrow aTRe$
$T \rightarrow Tbcb | b$
$R \rightarrow d$

$\rightarrow$ Shift a, Shift b
$\rightarrow$ Reduce T $\rightarrow$ b
$\rightarrow$ Shift b

Remaining input: cde

Rightmost derivation:
$S \rightarrow aTRe$
$\rightarrow aTde$
$\rightarrow aTbcede$
$\rightarrow abbcde$
Shift Reduce Parsing

\[
S \rightarrow aTRe \\
T \rightarrow Tbcb|b \\
R \rightarrow d
\]

⇒ Shift a, Shift b
⇒ Reduce T \rightarrow b
⇒ Shift b, Shift c

Remaining input: de

Rightmost derivation:

\[
\begin{align*}
S & \rightarrow aTRe \\
& \rightarrow aTd e \\
& \rightarrow aTb c d e \\
& \rightarrow abbcd e
\end{align*}
\]
Shift Reduce Parsing

\[ S \rightarrow a T R e \]
\[ T \rightarrow T b c | b \]
\[ R \rightarrow d \]

- Shift a, Shift b
- Reduce T \( \rightarrow b \)
- Shift b, Shift c
- Reduce T \( \rightarrow T b c \)

Remaining input: de

Rightmost derivation:

\[ S \rightarrow a T R e \]
\[ \rightarrow a T d e \]
\[ \rightarrow a T b c d e \]
\[ \rightarrow a b b c d e \]
Shift Reduce Parsing

\[ S \rightarrow a \ T \ R \ e \]
\[ T \rightarrow T \ b \ c \mid b \]
\[ R \rightarrow d \]

- Shift a, Shift b
- Reduce T \rightarrow b
- Shift b, Shift c
- Reduce T \rightarrow T b c
- Shift d

Remaining input: e

Rightmost derivation:
\[ S \rightarrow a \ T \ R \ e \]
\[ \rightarrow a \ T \ d \ e \]
\[ \rightarrow a \ T \ b \ c \ d \ e \]
\[ \rightarrow a \ b \ b \ c \ d \ e \]
Shift Reduce Parsing

\[ S \rightarrow a \ T \ R \ e \]
\[ T \rightarrow T \ b \ c \ | \ b \]
\[ R \rightarrow d \]

- \text{Shift } a, \text{ Shift } b
- \text{Reduce } T \rightarrow b
- \text{Shift } b, \text{ Shift } c
- \text{Reduce } T \rightarrow T \ b \ c
- \text{Shift } d
- \text{Reduce } R \rightarrow d

Remaining input: \( e \)

Rightmost derivation:

- \( S \rightarrow a \ T \ R \ e \)
- \( a \ T \ d \ e \)
- \( a \ T \ b \ c \ d \ e \)
- \( a \ b \ b \ c \ d \ e \)
Shift Reduce Parsing

\[
S \rightarrow a \ T \ R \ e \\
T \rightarrow T \ b \ c \ | \ b \\
R \rightarrow d
\]

- Shift a, Shift b
- Reduce T \rightarrow b
- Shift b, Shift c
- Reduce T \rightarrow T b c
- Shift d
- Reduce R \rightarrow d
- Shift e

Remaining input:

Rightmost derivation:

\[
S \rightarrow a \ T \ R \ e \\
\rightarrow a \ T \ d \ e \\
\rightarrow a \ T \ b \ c \ d \ e \\
\rightarrow a \ b \ b \ c \ d \ e
\]
Shift Reduce Parsing

\[ S \rightarrow a \ T \ R \ e \]
\[ T \rightarrow T \ b \ c \mid b \]
\[ R \rightarrow d \]

- Shift a, Shift b
- Reduce T \rightarrow b
- Shift b, Shift c
- Reduce T \rightarrow T b c
- Shift d
- Reduce R \rightarrow d
- Shift e
- Reduce S \rightarrow a \ T \ R \ e

Remaining input:

\[ S \]
\[ T \]
\[ R \]

Rightmost derivation:

\[ S \rightarrow a \ T \ R \ e \]
\[ \rightarrow a \ T \ d \ e \]
\[ \rightarrow a \ T \ b \ c \ d \ e \]
\[ \rightarrow a \ b \ b \ c \ d \ e \]
Example Shift-Reduce Parsing

Consider the grammar:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$id_1 + id_2$</td>
<td>shift</td>
</tr>
<tr>
<td>$id_1$</td>
<td>+ id_2$</td>
<td>reduce 6</td>
</tr>
<tr>
<td>$F$</td>
<td>+ id_2$</td>
<td>reduce 4</td>
</tr>
<tr>
<td>$T$</td>
<td>+ id_2$</td>
<td>reduce 2</td>
</tr>
<tr>
<td>$E$</td>
<td>+ id_2$</td>
<td>shift</td>
</tr>
<tr>
<td>$E +$</td>
<td>id_2$</td>
<td>shift</td>
</tr>
<tr>
<td>$E + id_2$</td>
<td></td>
<td>reduce 6</td>
</tr>
<tr>
<td>$E + F$</td>
<td></td>
<td>reduce 4</td>
</tr>
<tr>
<td>$E + T$</td>
<td></td>
<td>reduce 1</td>
</tr>
<tr>
<td>$E$</td>
<td></td>
<td>accept</td>
</tr>
</tbody>
</table>

Grammar rules:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow ( E )$
6. $F \rightarrow id$
Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsers cannot be used.
- Stack contents and the next input symbol may not decide action:
  - **shift/reduce conflict**: Whether make a shift operation or a reduction.
  - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.
- If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar.

  ![Diagram](left to right scanning) ![Diagram](right-most derivation) ![Diagram](k lookhead)

- An ambiguous grammar can never be a LR grammar.
Shift-Reduce Conflict in Ambiguous Grammar

\[
stmt \rightarrow \text{if expr then stmt} \\
| \quad \text{if expr then stmt else stmt} \\
| \quad \text{other}
\]

STACK  
....if expr then stmt  
INPUT  
else .....$

• We can’t decide whether to shift or reduce?
Reduce-Reduce Conflict in Ambiguous Grammar

1. \( stmt \rightarrow id(parameter\_list) \)
2. \( stmt \rightarrow expr:=expr \)
3. \( parameter\_list \rightarrow parameter\_list, parameter \)
4. \( parameter\_list \rightarrow parameter \)
5. \( parameter\_list \rightarrow id \)
6. \( expr \rightarrow id(expr\_list) \)
7. \( expr \rightarrow id \)
8. \( expr\_list \rightarrow expr\_list, expr \)
9. \( expr\_list \rightarrow expr \)

STACK

\[ \ldots id \ ( id \ldots \) \]

INPUT

\[ \ldots , id \) \ldots \$ \]

• We can’t decide which production will be used to reduce id?
Shift-Reduce Parsers

There are two main categories of shift-reduce parsers

1. **Operator-Precedence Parser**
   - simple, but only a small class of grammars.

2. **LR-Parsers**
   - covers wide range of grammars.
     - SLR – simple LR parser
     - LR – most general LR parser
     - LALR – intermediate LR parser (lookahead LR parser)
   - SLR, LR and LALR work same, only their parsing tables are different.
LR Parsers

LR parsing is attractive because:
- LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.
  \[ \text{LL}(1)\text{-Grammars} \subset \text{LR}(1)\text{-Grammars} \]
- An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.
- LR parsers can be constructed to recognize virtually all programming language constructs for which CFG grammars can be written.

Drawback of LR method:
- Too much work to construct LR parser by hand
  - Fortunately tools (LR parsers generators) are available
LR Parsing Algorithm

stack

S_m
X_m
S_{m-1}
X_{m-1}
.
.
S_1
X_1
S_0

input \( a_1 \ldots a_i \ldots a_n \$ \)

LR Parsing Algorithm

Action Table

- terminals and $ 
- four different actions

Goto Table

- non-terminal
- each item is a state number

output
Bottom-Up Parsing: LR(0) Table Construction
Constructing SLR Parsing Tables – LR(0) Item

• An **LR(0)** item of a grammar G is a production of G a dot at the same position of the right side.
• Ex: \( A \rightarrow aBb \) Possible LR(0) Items:
  \( A \rightarrow \cdot aBb \) (four different possibility)
  \( A \rightarrow a \cdot Bb \)
  \( A \rightarrow aB \cdot b \)
  \( A \rightarrow aBb \cdot \)

• Sets of LR(0) items will be the states of action and goto table of the SLR parser.
  – States represent sets of “items”
• LR parser makes shift-reduce decision by maintaining states to keep track of where we are in a parsing process
Constructing SLR Parsing Tables – LR(0) Item

- An item indicates how much of a production we have seen at a given point in the parsing process.
- For Example the item $A \rightarrow X \cdot YZ$
  - We have already seen on the input a string derivable from $X$
  - We hope to see a string derivable from $YZ$
- For Example the item $A \rightarrow \cdot XYZ$
  - We hope to see a string derivable from $XYZ$
- For Example the item $A \rightarrow XYZ \cdot$
  - We have already seen on the input a string derivable from $XYZ$
  - It is possibly time to reduce $XYZ$ to $A$

- Special Case:
  - Rule: $A \rightarrow \varepsilon$ yields only one item
    - $A \rightarrow \cdot$
Constructing SLR Parsing Tables

- A collection of sets of LR(0) items (the canonical LR(0) collection) is the basis for constructing SLR parsers.
- Canonical LR(0) collection provides the basis of constructing a DFA called LR(0) automaton
  - This DFA is used to make parsing decisions
- Each state of LR(0) automaton represents a set of items in the canonical LR(0) collection
- To construct the canonical LR(0) collection for a grammar
  - Augmented Grammar
  - CLOSURE function
  - GOTO function
Grammar Augmentation

Augment the grammar by adding...

- A new start symbol, $S'$
- A new rule $S' \rightarrow S$

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T \ast F$
4. $T \rightarrow F$
5. $F \rightarrow ( E )$
6. $F \rightarrow \text{id}$

```
0.  $S' \rightarrow E$
1.  $E \rightarrow E + T$
2.  $E \rightarrow T$
3.  $T \rightarrow T \ast F$
4.  $T \rightarrow F$
5.  $F \rightarrow ( E )$
6.  $F \rightarrow \text{id}$
```

Our goal is to find an $S'$, followed by $\$$.  
$S' \rightarrow \cdot E, \$ $

Whenever we are about to reduce using rule 0...  
Accept! Parse is finished!
The Closure Operation

- If $I$ is a set of LR(0) items for a grammar $G$, then $\text{closure}(I)$ is the set of LR(0) items constructed from $I$ by the two rules:

1. Initially, every LR(0) item in $I$ is added to $\text{closure}(I)$.
2. If $A \rightarrow \alpha.B\beta$ is in $\text{closure}(I)$ and $B \rightarrow \gamma$ is a production rule of $G$; then $B \rightarrow .\gamma$ will be in the $\text{closure}(I)$. We will apply this rule until no more new LR(0) items can be added to $\text{closure}(I)$. 
The Closure Operation -- Example

\[ E' \rightarrow E \]
\[ E \rightarrow E + T \]
\[ E \rightarrow T \]
\[ T \rightarrow T * F \]
\[ T \rightarrow F \]
\[ F \rightarrow (E) \]
\[ F \rightarrow id \]

\[ \text{closure} \{ E' \rightarrow \bullet E \} = \]
\[ \{ E' \rightarrow \bullet E \} \leftarrow \text{kernel items} \]
\[ E \rightarrow \bullet E + T \]
\[ E \rightarrow \bullet T \]
\[ T \rightarrow \bullet T * F \]
\[ T \rightarrow \bullet F \]
\[ F \rightarrow \bullet (E) \]
\[ F \rightarrow \bullet id \]
GOTO Operation

- If $I$ is a set of LR(0) items and $X$ is a grammar symbol (terminal or non-terminal), then $\text{GOTO}(I,X)$ is defined as follows:
  - If $A \rightarrow \alpha \cdot X \beta$ in $I$
    then every item in $\text{closure}(\{A \rightarrow \alpha X \cdot \beta\})$ will be in $\text{GOTO}(I,X)$.

Example:

$I = \{\ E' \rightarrow \cdot E, \ E \rightarrow \cdot E+T, \ E \rightarrow \cdot T, \ T \rightarrow \cdot T*F, \ T \rightarrow \cdot F, \ F \rightarrow \cdot (E), \ F \rightarrow \cdot \text{id} \}$

$\text{GOTO}(I,E) = \{ \ E' \rightarrow E \cdot, \ E \rightarrow E \cdot +T \}$

$\text{GOTO}(I,T) = \{ \ E \rightarrow T \cdot, \ T \rightarrow T \cdot *F \}$

$\text{GOTO}(I,F) = \{ T \rightarrow F \cdot \}$

$\text{GOTO}(I,\cdot) = \{ \ F \rightarrow \cdot E, \ E \rightarrow \cdot E+T, \ E \rightarrow \cdot T, \ T \rightarrow \cdot T*F, \ T \rightarrow \cdot F, \ F \rightarrow \cdot (E), \ F \rightarrow \cdot \text{id} \}$

$\text{GOTO}(I,\text{id}) = \{ \ F \rightarrow \text{id} \cdot \}$
LR(0) Automation

- Start with **start rule** & compute **initial state with closure**
- Pick one of the items from the states and **move “.” to the right**
  - **one symbol** (as if you parsed the symbol)
    - this creates a **new item**
    - … and a **new state** when you **compute the closure of the new item**
    - mark the edge between the two states with:
      - a terminal T, if you moved “.” over T
      - a non-terminal X, if you moved “.” over x
- **Continue until there are no further ways to move “.” across items** and **generate the new states or new edges in the automation.**
Grammar:

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

S' ::= @ S $
S ::= @ ( L )
S ::= @ x
Grammar:

0. S' ::= S $  
   •  S ::= ( L )  
   •  S ::= x  
   •  L ::= S  
   •  L ::= L , S
Grammar:

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

```
S' ::= @ S $  
S ::= @ ( L )  
S ::= @ x  
```
Grammar:

0. $S' ::= \text{@ } S \text{ $}$
   • $S ::= ( \text{ L })$
   • $S ::= \text{@ } x$
   • $L ::= S$
   • $L ::= L, S$

```
S' ::= @ S $
S ::= @ ( L )
L ::= S
L ::= L , S
```

```
S' ::= @ S $
S ::= @ ( L )
L ::= S
L ::= @ L , S
S ::= @ x
```
Grammar:

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

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

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

Grammar:

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

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

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

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

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

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

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

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

S ::= x @

S ::= ( L )
L ::= @ S
L ::= @ L, S
S ::= @ ( L )
S ::= @ x
Grammar:

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

```
S ::= x @
S ::= @ (L)
S ::= @ x
L ::= S
L ::= @ S
L ::= @ L, S
S ::= @ (L)
S ::= @ x
L ::= L @
L ::= L @, S
```

```
S' ::= S @$
S' ::= @ S $
S ::= @ (L)
S ::= @ x
S' ::= S @$
```

```
L ::= S
L ::= @ S
L ::= @ L, S
S ::= @ (L)
S ::= @ x
L ::= L @
L ::= L @, S
```
Grammar:

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

S ::= x @

S ::= ( @ L )
L ::= @ S
L ::= @ L \, S
S ::= @ ( L )
S ::= @ x

S ::= @ x

S ::= @ S$
S ::= @ ( L )
S ::= @ x

S' ::= @ S$

S ::= @ S$

S' ::= S @$

L ::= S @

S ::= ( L @ )
L ::= L @ \, S
Grammar:

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

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

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

Diagram:

- \( S' ::= @ S \) $
- \( S ::= @ ( L ) \)
- \( S ::= @ x \)
- \( S ::= x @ \)
- \( S ::= @ L \)
- \( L ::= @ S \)
- \( L ::= @ L , S \)
- \( S ::= ( @ L ) \)
- \( S ::= @ x \)
- \( S ::= ( L @ ) \)
- \( L ::= L @ , S \)
- \( S ::= ( L ) @ \)
Grammar:

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

```
S ::= x @
L ::= @ S
L ::= @ L\, ,\, S
S ::= ( @ L )
S ::= ( L @ )
S ::= ( L ) @
L ::= L \, @ \, S
```
Grammar:

0. $S' ::= S \, \epsilon$

- $S ::= ( \, L )$
- $S ::= x$
- $L ::= S$
- $L ::= L \, , \, S$

```
S ::= x \, @
S ::= ( \, L )
S ::= @ x
L ::= S
L ::= L \, , \, S
```
Grammar:

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

Diagram:
Grammar:

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

- $S ::= ( L )$
- $S ::= x$
- $L ::= S$
- $L ::= L , S$

$S' ::= @ S \$

$S ::= x @$

$L ::= S @$

$L ::= L , S @$

$L ::= L @ , S$

$L ::= L , S$

$L ::= ( L @)$

$L ::= ( L ) @$

$L ::= L @ , S$

$L ::= L , S$

$S ::= ( L )$

$S ::= ( L )$

$S ::= @ ( L )$

$S ::= @ x$

$S ::= @ ( L )$

$S ::= @ x$

$S ::= ( L )$

$S ::= ( L )$

$S ::= @ x$
Grammar:

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

```
S ::= @
S ::= ( @ L )
S ::= @ x
L ::= S
L ::= L \, S
S ::= @ ( L )
S ::= @ x
S ::= ( L @ )
L ::= L \, S @
S ::= ( L ) @
S ::= S @$
L ::= L \, @ S
S ::= @ ( L )
S ::= @ x
```
Grammar:

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

Assigning numbers to states:
Computing Parse table

- At every point in the parse, the LR parser table tells us what to do next according to the automaton state at the top of the stack
  - shift
  - reduce
  - accept
  - error
Computing Parse table

- State \( i \) contains \( X ::= s @ \$ \Longrightarrow \text{table}[i,\$] = a \)
- State \( i \) contains rule \( k: X ::= s @ \Longrightarrow \text{table}[i,T] = rk \) for all terminals \( T \)
- Transition from \( i \) to \( j \) marked with \( T \Longrightarrow \text{table}[i,T] = sj \)
- Transition from \( i \) to \( j \) marked with \( X \Longrightarrow \text{table}[i,X] = gj \) for all nonterminals \( X \)

| states | Terminal seen next | Non-terminals | X, Y, Z ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( \text{sn} = \text{shift \&amp; goto state n} )</td>
<td>( \text{gn} = \text{goto state n} )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( \text{rk} = \text{reduce by rule k} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>( \text{a} = \text{accept} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>( = \text{error} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Parse Table

• Reducing by rule k is broken into two steps:
  – current stack is:
    \[ A \ 8 \ B \ 3 \ C \ \ldots \ \ 7 \ \text{RHS} \ 12 \]
  – rewrite the stack according to \( X ::= \text{RHS} \):
    \[ A \ 8 \ B \ 3 \ C \ \ldots \ \ 7 \ X \]
  – figure out state on top of stack (ie: goto 13)
    \[ A \ 8 \ B \ 3 \ C \ \ldots \ \ 7 \ X \ 13 \]

<table>
<thead>
<tr>
<th>states</th>
<th>Terminal seen next ID, NUM, ::= ...</th>
<th>Non-terminals X,Y,Z ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>gn = goto state n</td>
</tr>
<tr>
<td>2</td>
<td>sn = shift &amp; goto state n</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>rk = reduce by rule k</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>a = accept</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>= error</td>
<td></td>
</tr>
</tbody>
</table>
0. $S' ::= S \, \$$
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L, S

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

2. $S ::= x \, @$

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

4. $S' ::= S \, @\$  

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

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

7. $L ::= S @$

8. $L ::= L, @ S$
   $S ::= @ (L)$
   $S ::= @ x$

9. $L ::= L, S @$

---

<table>
<thead>
<tr>
<th>states</th>
<th>( )</th>
<th>x</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
0. \( S' ::= S \) $
- \( S ::= (L) \)
- \( S ::= x \)
- \( L ::= S \)
- \( L ::= L, S \)

<table>
<thead>
<tr>
<th>states</th>
<th>( )</th>
<th>x</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
S ::= @ S $
S ::= @ ( L )
S ::= @ x
S ::= x @
S ::= S @
L ::= S @
L ::= @ S
L ::= @ L , S
S ::= @ ( L )
S ::= @ x
L ::= L @ , S
L ::= L , @ S
S ::= @ ( L )
S ::= @ x
L ::= L @
S ::= ( L @ )
S ::= ( @ L )
S ::= ( L @ )
S ::= @ S
S ::= @ x
L ::= L@ , S
L ::= @ L , S
S ::= @ ( L )
S ::= @ x
L ::= L @
S ::= ( L @ )
S ::= @ S
S ::= @ x

states | ( | ) | x | , | $ | S | L
-------|---------|---------|---------|---------|-----|-----
1      | s3      | s2      |         |         |     |     |
2      |         |         |         |         |     |     |
3      |         |         |         |         |     |     |
4      |         |         |         |         |     |     |
...    |         |         |         |         |     |     |
0. $S' ::= S \$$
- \hspace{10pt} S ::= (L)
- \hspace{10pt} S ::= x$
- \hspace{10pt} L ::= S$
- \hspace{10pt} L ::= L, S$

\begin{tabular}{|c|c|c|c|c|}
\hline
states & ( ) & x & , & $\$ \hline
1 & s3 & s2 & & g4 \\
2 & & & & \\
3 & & & & \\
4 & & & & \\
\ldots & & & & \\
\hline
\end{tabular}
0. \( S' ::= S \) 
- \( S ::= ( L ) \)
- \( S ::= x \)
- \( L ::= S \)
- \( L ::= L, S \)

<table>
<thead>
<tr>
<th>states</th>
<th>( )</th>
<th>x</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
0. S' ::= S $
- S ::= ( L )
- S ::= x
- L ::= S
- L ::= L , S

1 S' ::= @ S $  
   S ::= @ ( L )
   S ::= @ x

2 S ::= x @  

3 S ::= ( @ L )
   L ::= @ S
   L ::= @ L , S
   S ::= @ ( L )
   S ::= @ x

4 S' ::= S @ $  

5 L ::= @ S
   S ::= @ ( L )
   S ::= @ x

6 S ::= ( L @ )

7 L ::= L , S @
   S ::= L @
   S ::= @ S
   S ::= @ x

8 L ::= L , @ S
   S ::= @ S
   S ::= @ x

9 L ::= L , S @
   S ::= @ S
   S ::= @ x

states | ( | ) | x | , | $ | S | L
---|---|---|---|---|---|---|---
1 | s3 | s2 | | | | g4 |
2 | r2 | r2 | r2 | r2 | r2 |
3 | s3 | s2 |
4 |
...
0. \( S' ::= S \) $ \\
1. \( S' ::= @ S \) $ \\
2. \( S ::= x @ \) \\
3. \( S ::= ( @ L ) \) \\
4. \( S ::= @ x \) \\
5. \( S ::= @ ( L ) \) \\
6. \( S ::= @ x \) \\
7. \( S ::= ( L @ ) \) \\
8. \( L ::= @ S, S \) \\
9. \( L ::= L, S @ \)

<table>
<thead>
<tr>
<th>states</th>
<th>( )</th>
<th>x</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
<td>r2</td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g7</td>
<td>g5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>states</td>
<td>(   )</td>
<td>x</td>
<td>,</td>
<td>$</td>
<td>S</td>
<td>L</td>
</tr>
<tr>
<td>--------</td>
<td>-------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g7</td>
<td>g5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td>g9</td>
</tr>
<tr>
<td>9</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
</tbody>
</table>

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

Input: $( x , x ) \$ 
Stack: 1
### Parsing Table

<table>
<thead>
<tr>
<th>states</th>
<th>( )</th>
<th>x</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td>g7</td>
<td>g5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
</tbody>
</table>

## Grammar

- $S' ::= S \,$
- $S ::= ( L )$
- $S ::= x$
- $L ::= S$
- $L ::= L , S$

### Example

**Input:**

$$( x , x ) \,$$

**Stack:**

1 ( 3
### Grammar

- **S’ ::= S $**
- **S ::= ( L )**
- **S ::= x**
- **L ::= S**
- **L ::= L , S**

### Transition Table

<table>
<thead>
<tr>
<th>states</th>
<th>( )</th>
<th>x</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g7</td>
<td>g5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>s6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
</tbody>
</table>

### Input

- **input:** `( x , x ) $`

### Stack

- **stack:** `1 ( 3 x 2 )`
0. \( S' ::= S \ $ \)
- \( S ::= ( \ L ) \)
- \( S ::= x \)
- \( L ::= S \)
- \( L ::= L , S \)

yet to read

<table>
<thead>
<tr>
<th>states</th>
<th>( )</th>
<th>x</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g7</td>
<td>g5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>s6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
</tbody>
</table>

input: \( ( x , x ) \ $ \)

stack: 1 ( 3 S
yet to read

input: \( ( \ x \ , \ x \ ) \ $

stack: 1 ( 3 S 7

0. \( S' ::= S \$
  • \( S ::= ( \ L \)$
  • \( S ::= x \)
  • \( L ::= S \)
  • \( L ::= L , S \)
### Grammar Rules

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

### Transition Table

<table>
<thead>
<tr>
<th>states</th>
<th>(  )</th>
<th>x</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td>$</td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td>g7</td>
<td>g5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
</tbody>
</table>

### Input and Stack

Input: $( \, x \, , \, x \, ) \, \$$

Stack: 1 ( 3 L

yet to read
yet to read

input: ( x , x )$

stack: 1 ( 3 L 5

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

states | ( ) x , $ S | L
---|---|---|---|
1 | s3 | s2 | g4 |
2 | r2 | r2 | r2 | r2 |
3 | s3 | s2 | g7 | g5 |
4 | | | a |
5 | s6 | s8 |
6 | r1 | r1 | r1 | r1 |
7 | r3 | r3 | r3 | r3 |
8 | s3 | s2 | g9 |
9 | r4 | r4 | r4 | r4 |
yet to read

input: ( x , x ) $

stack: 1 ( 3 L 5 , 8

0. S' ::= S $

- S ::= ( L )$
- S ::= x$
- L ::= S$
- L ::= L , S
<table>
<thead>
<tr>
<th>states</th>
<th>( )</th>
<th>x</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g7</td>
<td>g5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
</tbody>
</table>

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

yet to read

input: \( ( x , x ) \) $

stack: 1 ( 3 L 5 , 8 x 2 )
yet to read

input: \( ( \; x \; , \; x \; ) \; $

stack: \( 1 \; ( \; 3 \; L \; 5 \; , \; 8 \; S \) \)

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
states & ( ) & x & , & $ & S & L \\
\hline
1 & s3 & s2 & & & g4 & \\
\hline
2 & r2 & r2 & r2 & r2 & r2 & \\
\hline
3 & s3 & s2 & & & g7 & g5 \\
\hline
4 & & & & & a & \\
\hline
5 & & s6 & & s8 & \\
\hline
6 & r1 & r1 & r1 & r1 & r1 & \\
\hline
7 & r3 & r3 & r3 & r3 & r3 & \\
\hline
8 & s3 & s2 & & & g9 & \\
\hline
9 & r4 & r4 & r4 & r4 & r4 & \\
\hline
\end{tabular}

0. \( S' ::= S \; $ \)
- \( S ::= ( \; L \; ) \)
- \( S ::= x \)
- \( L ::= S \)
- \( L ::= L \; , \; S \)
<table>
<thead>
<tr>
<th>states</th>
<th>( )</th>
<th>x</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g7</td>
<td>g5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>s6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
</tbody>
</table>

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

yet to read

input: \((x, x)\) $

stack: 1 (3 \ L 5, 8 \ S 9)
### States Transition Table

<table>
<thead>
<tr>
<th>states</th>
<th>( )</th>
<th>x</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g7</td>
<td>g5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
</tbody>
</table>

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

### Example

- **Input:** $( \ x, x \ ) \$  
- **Stack:** 1 ( 3 L
<table>
<thead>
<tr>
<th>states</th>
<th>( )</th>
<th>x</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g7</td>
<td>g5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
</tbody>
</table>

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

yet to read

input: \( ( x , x ) \  S \)

stack: \( 1 ( 3 L 5 \) \)

etc ......
LR(0)

- Even though we are doing LR(0) parsing we are using some look ahead (there is a column for each non-terminal)
- however, we only use the terminal to figure out which state to go to next, not to decide whether to shift or reduce

<table>
<thead>
<tr>
<th>states</th>
<th>(</th>
<th>)</th>
<th>x</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>s2</td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g7</td>
<td>g5</td>
</tr>
</tbody>
</table>
LR(0)

- Even though we are doing LR(0) parsing we are using some look ahead (there is a column for each non-terminal)
- however, we only use the terminal to figure out which state to go to next, not to decide whether to shift or reduce

<table>
<thead>
<tr>
<th>states</th>
<th>( )</th>
<th>x</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td>g7</td>
<td>g5</td>
</tr>
</tbody>
</table>

ignore next automaton state

<table>
<thead>
<tr>
<th>states</th>
<th>no look-ahead</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shift</td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>reduce 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>shift</td>
<td>g7</td>
<td>g5</td>
</tr>
</tbody>
</table>
Even though we are doing LR(0) parsing we are using some look ahead (there is a column for each non-terminal). However, we only use the terminal to figure out which state to go to next, not to decide whether to shift or reduce. If the same row contains both shift and reduce, we will have a conflict \(\Rightarrow\) the grammar is not LR(0). Likewise if the same row contains reduce by two different rules.

<table>
<thead>
<tr>
<th>states</th>
<th>no look-ahead</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shift, reduce 5</td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>reduce 2, reduce 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>shift</td>
<td>g7</td>
<td>g5</td>
</tr>
</tbody>
</table>
SLR

- SLR (simple LR) is a variant of LR(0) that reduces the number of conflicts in LR(0) tables by using a tiny bit of look ahead.
- To determine when to reduce, 1 symbol of look ahead is used.
- Only put reduce by rule \( X ::= \text{RHS} \) in column T if T is in Follow(X).

<table>
<thead>
<tr>
<th>states</th>
<th>( )</th>
<th>x</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td>$</td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>s5</td>
<td>r2</td>
<td></td>
<td></td>
<td>g4</td>
</tr>
<tr>
<td>3</td>
<td>r1</td>
<td>r1</td>
<td>r5</td>
<td>r5</td>
<td>g7</td>
<td>g5</td>
</tr>
</tbody>
</table>

cuts down the number of rk slots & therefore cuts down conflicts
LR(1) & LALR

• LR(1) automata are identical to LR(0) except for the “items” that make up the states
• LR(0) items:
  \[ X ::= s_1 . s_2 \]
• LR(1) items
  \[ X ::= s_1 . s_2, \ T \]
  – Idea: sequence \( s_1 \) is on stack; input stream is \( s_2 \ T \)
• Find closure with respect to \( X ::= s_1 . Y s_2, \ T \) by adding all items \( Y ::= s_3, U \) when \( Y ::= s_3 \) is a rule and \( U \) is in \( \text{First}(s_2 \ T) \)
• Two states are different if they contain the same rules but the rules have different look-ahead symbols
  – Leads to many states
  – \( \text{LALR}(1) = \text{LR}(1) \) where states that are identical aside from look-ahead symbols have been merged
  – ML-Yacc & most parser generators use LALR
• READ: Appel 3.3 (and also all of the rest of chapter 3)
Summary

• LR parsing is more powerful than LL parsing, given the same look ahead
• to construct an LR parser, it is necessary to compute an LR parser table
• the LR parser table represents a finite automaton that walks over the parser stack
• ML-Yacc uses LALR, a compact variant of LR(1)