Last Time

- Parsing
  - Top-down or bottom-up
- Top-down parsing
  - Recursive descent
  - LL(k) grammars
  - LL(k) parsing with pushdown automata
- LL(k) parsers
  - Cannot deal with common prefixes and left recursion
  - Left-recursion removal might result in complicated grammar

Parser Classes – Reminder

- Top-down (predictive)
- Bottom-up (shift-reduce)

LR(k) Grammars

- A grammar is in the class LR(k) when it can be derived via:
  - Bottom-up analysis
  - Scanning the input from left to right (L)
  - Producing the rightmost derivation (R)
  - With lookahead of k tokens (k)
- A language is said to be LR(k) if it has an LR(k) grammar
- The simplest case is LR(0), which we discuss next
Any LL(k) language is also in LR(k) (and not vice versa), i.e., LL(k) ⊂ LR(k).

The lookahead is counted differently in the two cases:
- With LL(k), the algorithm sees k tokens of the right-hand side of the rule and then must select the derivation rule.
- With LR(k), the algorithm sees all right-hand side of the derivation rule plus k more tokens.

The LR family of parsers is more popularly used today.

Example: a Simple LR(0) Grammar

\[ E \rightarrow E \ast B \mid E + B \mid B \]

\[ B \rightarrow 0 \mid 1 \]

Let us number the rules:
1. \( E \rightarrow E \ast B \)
2. \( E \rightarrow E + B \)
3. \( E \rightarrow B \)
4. \( B \rightarrow 0 \)
5. \( B \rightarrow 1 \)

Goal: Reduce the Input to the Start Symbol

Example:
\[ 0 + 0 \ast 1 \]

\[ B + 0 \ast 1 \]

\[ E + 0 \ast 1 \]

Go over the input so far, and upon seeing a right-hand side of a rule, “invoke” the rule and replace the right-hand side with the left-hand side (which is a single non-terminal).

Shift & Reduce

In each step, we either shift a symbol from the input to the stack, or reduce according to one of the rules. Example: “0 + 0 \ast 1”.

Shift / Reduce Parser — Intuition

Gather input token by token
- until we find a right-hand side of a rule
- then, replace it with the non-terminal on the left hand side
- Going over a token and recording it in the stack is a shift
- Each shift moves to a state that records what we’ve seen so far
- A reduce replaces a string on the stack with a nonterminal that derives it
For a production rule \( N \rightarrow \alpha \beta \) in the grammar:

- **Input**:
  - Already matched
  - To be matched

\[
N \rightarrow \alpha \beta
\]

So far we've matched \( \alpha \), expecting to see \( \beta \)

**LR(0) Item**

\[
E \rightarrow E \ast B \mid E + B \mid B
\]

\[
B \rightarrow 0 \mid 1
\]

- \( E \rightarrow E \ast B \) **Shift Item**
- \( E \rightarrow E \ast B \) **Reduce Item**

Example: Parsing with LR(0) Items

\[
Z \rightarrow \text{expr} $
\]

\[
\text{expr} \rightarrow \text{term} \mid \text{expr} + \text{term}
\]

\[
\text{term} \rightarrow \text{ID} \mid ( \text{expr} )
\]

Z → E $
E → T | E + T
T → i | ( E )

(just a shorthand of the grammar on top)

Example: Parsing with LR(0) Items

**Input**

\[
\text{expr} \rightarrow \text{term} \mid \text{expr} + \text{term}
\]

Z → E $
E → T | E + T
T → i | ( E )

**Closure**

\[
Z \rightarrow \text{E} $
\]

E → T | E + T
T → i | ( E )

**Shift**

\[
\text{expr} \rightarrow \text{term} \mid \text{expr} + \text{term}
\]

Z → E $
E → T | E + T
T → i | ( E )

**Reduce Item**
Reducing the initial rule means accept
How does the parser know what to do?

- Pushdown Automaton!
  - A state will keep the info gathered so far
  - A table will tell it "what to do" based on current state and next token
  - Some info will be kept in a stack

Why do we need a stack?

- Suppose so far we have discovered $E \rightarrow B \rightarrow 0$ and $+$;
  - So we have constructed sentential form "$E +$".
- In the given grammar this can only mean $E \rightarrow E + B$.
- Suppose current state $q_6$ represents this situation.
- Now, the next token is $0$, and we need to ignore $q_6$ for a minute, and work on $B \rightarrow 0$ to obtain $E + B$.
- Therefore, we push $q_6$ to the stack, and after identifying $B$, we pop it to continue.

The Stack

- The stack contains states
- For readability we also include variables and tokens (the recognizer does not need them)
- The initial stack contains $q_0$ only
- Apart from $q_0$ at the bottom of the stack, the rest of the stack contains pairs of (state, token) or (state, nonterminal)
The ACTION Table

- At each step we need to decide whether to **shift** the next token to the stack (and move to the appropriate state) or **reduce** a production rule from the grammar
- The ACTION table tells us what to do based on current state and next token:

  - **shift** $n$: shift and move to $q_n$
  - **reduce** $m$: reduce according to production rule (m)

  (also: accept and error conditions)

The GOTO Table

- Defines what to do on **reduce** actions
- After reducing a right-hand side to the deriving non-terminal, we need to decide what the next state is
- This is determined by the previous state (which is on the stack) and the variable we got
  - Suppose we reduce according to $N \rightarrow \beta$;
  - We remove $\beta$ from the stack, and look at the state $q$ that is now at the top. GOTO[$q, N$] specifies the next state.

  
  - *Note – this can be a little confusing:*
  - $q$ is the state after popping $\beta$
  - $N$ is the left-hand side of the rule just used in **reduce**

For example...

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>+</td>
</tr>
<tr>
<td>$q_0$</td>
<td>r1</td>
</tr>
<tr>
<td>$q_1$</td>
<td>r1</td>
</tr>
<tr>
<td>$q_2$</td>
<td>r1</td>
</tr>
<tr>
<td>$q_3$</td>
<td>r1</td>
</tr>
<tr>
<td>$q_4$</td>
<td>r1</td>
</tr>
<tr>
<td>$q_5$</td>
<td>r1</td>
</tr>
<tr>
<td>$q_6$</td>
<td>r1</td>
</tr>
<tr>
<td>$q_7$</td>
<td>r1</td>
</tr>
</tbody>
</table>

The Algorithm, Formally

- Initialize the stack to $q_0$
- Repeat until halting:
  - Consider ACTION[$q, t$] for $q$ at the top of stack and $t$ the next token
    - **shift $n$**: Remove $t$ from the input and push $t$ and then $q_n$ to the stack.
    - **reduce $m$**, where rule $(m)$ is $N \rightarrow \beta$:
      - Remove $\beta$ pairs from the stack; let $q$ be the state at the top of the stack.
      - Push $M$ and the state GOTO[$q, M$] to the stack.
    - **accept**: halt successfully.
    - empty cell: halt with an error.

Using LR Items to Build the Tables

- Typically a state consists of several LR items
- For example, if we identified a string that is reduced to $E$, then we may be in one of the following LR items:

  $E \rightarrow E + B$ or $E \rightarrow E * B$

- Therefore one state would be:

  $q = \{E \rightarrow E + B , E \rightarrow E * B\}$

- But if the current state includes $E \rightarrow E + B$, then we must allow $B$ to be derived too → **Closure**
Construct the Closure

• Proposition: a closure set of LR(0) items has the following property — if the set contains an item of the form
  \[ A \rightarrow \alpha \cdot B \beta \]
  then it must also contain an item
  \[ B \rightarrow \delta \]
  for each rule of the form \( B \rightarrow \delta \) in the grammar.

• Building the closure set for a given item set is recursive, as \( \delta \) may also begin with a variable.

Closure: an example

• The closure of the set \( C \) is
  \[ \text{clos}(C) = \{ E \rightarrow E + B , \ B \rightarrow B , \ B \rightarrow 0 , \ B \rightarrow 1 \} \]
  This will become another parser state

Extended Grammar

• Goal: simple termination condition
  ‣ Assume that the initial variable only appears in a single rule.
  This guarantees that the last reduction can be (easily) detected.
  ‣ Any grammar can be (easily) extended to have such structure.

Example: the grammar

```
(1) E \rightarrow E \times B
(2) E \rightarrow E + B
(3) E \rightarrow B
(4) B \rightarrow 0
(5) B \rightarrow 1
```

Can be extended into

```
(0) S \rightarrow E
(1) E \rightarrow E \times B
(2) E \rightarrow E + B
(3) E \rightarrow B
(4) B \rightarrow 0
(5) B \rightarrow 1
```

The Initial State

• To build the ACTION/GOTO table, we go through all possible states during derivation
  • Each state represents a (closure) set of LR(0) items
  • The initial state \( q_0 \) is the closure of the initial rule
  • In our example the initial rule is \( S \rightarrow E \), and therefore the initial state is
    \[ q_0 = \text{clos}(S \rightarrow E) = \{ S \rightarrow E , \ E \rightarrow E \times B , \ E \rightarrow \times + B , \ E \rightarrow B , \ B \rightarrow 0 , \ B \rightarrow 1 \} \]
  • We build all possible next states by following a single symbol (token or variable)

The Next States

• For each possible terminal or variable \( X \), and each possible state (closure set) \( q \),
  1. Find all items in the set of \( q \) in which the dot is before \( X \).
     We denote this set by \( q \cdot X \)
  2. Move the dot ahead of \( X \) in all items in \( q \cdot X \)
  3. Find the closure of the obtained set:
     this is the state into which we move from \( q \) upon seeing \( X \)

• Formally, the next set of a set \( C \) and next symbol \( X \)
  \[ \text{nextSet}(C, X) = \text{clos}(\text{step}(C, X)) \]
Recall that in our example
\[ q_0 = \text{clos}(S \rightarrow \#E) = \]
\[ \{ S \rightarrow \#E, E \rightarrow \#E \ast B, E \rightarrow \#E + B, E \rightarrow \#0, B \rightarrow \#1 \} \]
Let us check which states are reachable from it.

States reachable from \( q_0 \) in the example

From these new states there are more reachable states

Finally

Automaton
# Building the Tables

- A row for each state.
- If $q_j$ was obtained at $q_i$ upon seeing $x$, then in row $q_i$ and column $x$ we write $j$.

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 2 3 4 5 6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

# Building the tables: accept

- Add accept in column $S$ for each state that has $S \rightarrow E$ as an item.

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 2 3 4 5 6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

# Building the Tables: Shift

- Any number $n$ in the action table becomes shift $n$.

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$s1$ $s2$</td>
<td>3 4</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

# Building the Tables: Reduce

- For any state whose set includes the item $A \rightarrow \alpha$, such that $A \rightarrow \alpha$ is production rule ($m$):
  - Fill all columns of that state in the ACTION table with reduce $m$.

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$s1$ $s2$</td>
<td>3 4</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

# Note on LR(0)

- When a reduce is possible, we execute it without checking the next token.

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$s1$ $s2$</td>
<td>3 4</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### GOTO/ACTION Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Action</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ET</td>
<td>q0</td>
<td>s5</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>q2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>s3</td>
<td>q3</td>
<td>s5</td>
<td></td>
</tr>
<tr>
<td>s7</td>
<td>q7</td>
<td>s5</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

* s = shift to state n
  
  r = reduce using rule number m

### Are we done?

- Can make a transition diagram for any grammar
- Can make a GOTO table for every grammar
- ... but the states are not always clear on what to do
  
  ⇒ Cannot make a deterministic ACTION table for every grammar

### LR(0) Conflicts

shift/reduce conflict

### View in Action/Goto Table

- shift/reduce conflict...

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Action</th>
<th>Action</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ET</td>
<td>q0</td>
<td>s5</td>
<td>s7</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>q2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>s3</td>
<td>q3</td>
<td>s5</td>
<td>s7</td>
<td></td>
</tr>
<tr>
<td>s7</td>
<td>q7</td>
<td>s5</td>
<td>s7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
LR(0) Conflicts

Z → • E
E → • T
E → • E + T
T → • i
V → • i
T → • (E)

View in Action/Goto Table

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>15</td>
</tr>
</tbody>
</table>

Can there be a shift/shift conflict?

LR(0) vs. ε-Rules

- Whenever a nonterminal has an ε production, it will be reduced as soon as it is reached in the grammar (without looking at the next token).
- If the variable has another production with a terminal prefix, there is an inherent shift/reduce conflict

Coming Up

Yet More LR Parsing

THEORY OF COMPILATION
LECTURE 04
Syntax Analysis
Bottom-Up Parsing
Reminder – Parser Classes

- Top-down (predictive)
- Bottom-up (shift-reduce)

Reminder – LR(0) Parsing

Input

Stack

Output

ACTION Table

GOTO Table

LR(0) Parsing Algorithm

<table>
<thead>
<tr>
<th>S → E</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E + B</td>
</tr>
<tr>
<td>E → E * B</td>
</tr>
<tr>
<td>E → T</td>
</tr>
<tr>
<td>T → 0</td>
</tr>
<tr>
<td>T → 1</td>
</tr>
</tbody>
</table>

Reminder – LR(0) Conflicts

<table>
<thead>
<tr>
<th>S → E $</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → T</td>
</tr>
<tr>
<td>E → E + T</td>
</tr>
<tr>
<td>T → 0</td>
</tr>
<tr>
<td>T → 1</td>
</tr>
</tbody>
</table>

shift/reduce conflict
Reminder – LR(0) Conflicts

Back to Action/Goto Table

SLR Grammars

GOTO/ACTION Table

GOTO/ACTION Table
The tokens that can follow $E$ are '+' ')' and '$$$.  

[1]

\[ Z \rightarrow E \$

[2]

\[ E \rightarrow T \$

[3]

\[ E \rightarrow E + T \$

[4]

\[ T \rightarrow i \$

[5]

\[ T \rightarrow (\ E \ ) \$

Now let's add "\( T \rightarrow i \ [\ E \] \)"

\[ Z \rightarrow E \$

\[ E \rightarrow T \$

\[ E \rightarrow E + T \$

\[ T \rightarrow i \$

\[ T \rightarrow (\ E \ ) \$

Now let's add "\( T \rightarrow i \ [\ E \] \)"

SLR: check next token when reducing

- Simple LR(1), or SLR(1), or SLR.
- Example demonstrates elimination of a shift/reduce conflict.
- Can eliminate reduce/reduce conflicts when conflicting rules' left-hand sides satisfy:
  \[ \text{FOLLOW}(T) \cap \text{FOLLOW}(V) \neq \emptyset. \]
- But cannot resolve all conflicts.
Consider this non-LR(0) grammar

\[
\begin{align*}
S' &\rightarrow S \\
S &\rightarrow L = R \\
S &\rightarrow R \\
L &\rightarrow \ast R \\
L &\rightarrow \text{id} \\
R &\rightarrow L
\end{align*}
\]

Shift/reduce conflict

\[
\begin{align*}
S &\rightarrow L \ast R \quad \text{vs.} \quad R \rightarrow L \ast R \\
\text{FOLLOW}(R) \text{ contains } \ast \Rightarrow \text{SLR cannot resolve the conflict either}
\end{align*}
\]

Resolving the Conflict

- In SLR, a reduce item \(N \rightarrow \alpha \ast\) is applicable when the lookahead is in \(\text{FOLLOW}(N)\).
- But there is a whole sentential form that we have discovered so far.
- We can ask what the next token may be given all previous reductions.
- For example, even looking at the \(\text{FOLLOW}\) of the entire sentential form is more restrictive than looking at the \(\text{FOLLOW}\) of the last variable.
- In a way, \(\text{FOLLOW}(N)\) merges look-ahead for all possible occurrences of \(N\):
\[
\text{FOLLOW}(\sigma N) \subseteq \text{FOLLOW}(N)
\]
- LR(1) keeps look-ahead with each LR item

LR(1) Item

\[
N \rightarrow \alpha \ast \beta, \sigma
\]

So far we've matched \(\alpha\), expecting to see \(\beta\), followed by the lookahead \(\sigma\)
LR(1) Item

- Example: the production $L \rightarrow \text{id}$ yields the following LR(1) items:

<table>
<thead>
<tr>
<th>Productions</th>
<th>Tag</th>
<th>ACTION</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S' \rightarrow S$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S \rightarrow R$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S' \rightarrow S$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L \rightarrow \text{id}$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L \rightarrow \text{R}$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L \rightarrow \text{id}$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S \rightarrow \text{R}$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L \rightarrow \text{id}$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S' \rightarrow S$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L \rightarrow \text{id}$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L \rightarrow \text{R}$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L \rightarrow \text{id}$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S \rightarrow \text{R}$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L \rightarrow \text{id}$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S' \rightarrow S$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L \rightarrow \text{id}$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L \rightarrow \text{R}$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L \rightarrow \text{id}$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S \rightarrow \text{R}$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L \rightarrow \text{id}$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Creating the states for LR(1)

- We start with the initial state:
  
  $q_0$ will be the closure of: $(S' \rightarrow S, S)$

- Closure for LR(1):
  
  - For every $(A \rightarrow \alpha, \epsilon)$ in the state:
    
    * For every production $B \rightarrow \beta$ and every token $b \in \text{FIRST}[\beta]$
    
    * $[B \rightarrow \beta, b]$ should also be in the state

Closure of $(S' \rightarrow S, S)$

- We would like to add rules that start with $S$, but keep track of possible lookahead.
  
  * $S' \rightarrow S, S$ – Rules for $S$
  
  * $S' \rightarrow \text{R}, S$ – Rules for $L$
  
  * $S' \rightarrow \text{id}, S$ – Rules for $R$
  
  * $[L \rightarrow \text{id}, \epsilon]$ – Rules for $L$
  
  * $[R \rightarrow \text{id}, \epsilon]$ – Rules for $R$
  
  * $[S \rightarrow \text{id}, \epsilon]$ – Rules for $L$
  
  * $[S \rightarrow \text{id}, \epsilon]$ – Rules for $L$

The State Machine
The State Machine

Back to the conflict

Building the Tables

Building the Table

Bottom-up Parsing
Chomsky Hierarchy

Grammar Hierarchy

Building the Parse Tree
- Done at the time of `reduce`.

Building the Abstract Syntax Tree
- Generally — just "skip over" the creation of some internal nodes and you get an AST
### Summary

- Bottom up derivation
- LR(k) can decide on a reduce after seeing the entire right side of the rule plus k look-ahead tokens.
- Particularly LR(0).
- Using a table and a stack to derive.
- LR items and the automaton.
- Creating the table from the automaton.
- LR parsing with pushdown automata
- LR(0), SLR, LR(1) – different kinds of LR items, same basic algorithm.
- LALR: in the tutorial.

---

### Coming Up

- Semantic Analysis