**CSC 4181**
**Compiler Construction**

**Parsing**

---

### Outline
- Top-down v.s. Bottom-up
  - Top-down parsing
    - Recursive-descent parsing
    - LL(1) parsing
      - LL(1) parsing algorithm
      - First and follow sets
      - Constructing LL(1) parsing table
      - Error recovery
  - Bottom-up parsing
    - Shift-reduce parsers
    - LR(0) parsing
      - LR(0) items
      - Finite automata of items
      - LR(0) parsing algorithm
      - LR(0) grammar
      - SLR(1) parsing
      - SLR(1) parsing algorithm
      - SLR(1) grammar
      - Parsing conflict

---

### Introduction
- Parsing is a process that constructs a syntactic structure (i.e. parse tree) from the stream of tokens.
- We already learned how to describe the syntactic structure of a language using (context-free) grammar.
- So, a parser only needs to do this?

**Stream of tokens** → **Context-free grammar** → **Parser** → **Parse tree**

---

### Parse Trees and Derivations

#### Top-down Parsing
- E → E + E
- id + E
- id + E * E
- id + id * E
- id + id * id
- E + E
- E + E * E
- E + id * id
- E + id * id

#### Bottom-up Parsing
- E → E + E
- id + E
- id + E * E
- id + id * E
- id + id * id
- E + E
- E + E * E
- E + id * id
- E + id * id

---

### Top-down Parsing
- What does a parser need to decide?
  - Which production rule is to be used at each point of time?
- How to guess?
  - What is the next token?
    - Reserved word if, open parentheses, etc.
  - What is the structure to be built?
    - If statement, expression, etc.
**Top-down Parsing**

- Why is it difficult?
  - Cannot decide until later
  - Next token: if Structure to be built: St
    St → MatchedSt | UnmatchedSt
    UnmatchedSt → if (E) St | if (E) MatchedSt else UnmatchedSt
  - MatchedSt → if (E) MatchedSt else MatchedSt [...]
  - Production with empty string
    Next token: id Structure to be built: par
    par → parList | λ
    parList → exp , parList | exp

**Recursive-Descent**

- Write one procedure for each set of productions with the same nonterminal in the LHS
- Each procedure recognizes a structure described by a nonterminal.
- A procedure calls other procedures if it needs to recognize other structures.
- A procedure calls match procedure if it needs to recognize a terminal.

**Recursive-Descent: Example**

- For this grammar:
  - We cannot decide which rule to use for E, and
  - If we choose E → E O F, it leads to infinitely recursive loops.
  - Rewrite the grammar into EBNF

**Match procedure**

- procedure match(expTok)
  - if (token==expTok) then getToken else error

- The token is not consumed until getToken is executed.

**Problems in Recursive-Descent**

- Difficult to convert grammars into EBNF
- Cannot decide which production to use at each point
- Cannot decide when to use λ-production A → λ

**LL(1) Parsing**

- LL(1)
  - Read input from (L) left to right
  - Simulate (L) leftmost derivation
  - 1 lookahead symbol
  - Use stack to simulate leftmost derivation
    - Part of sentential form produced in the leftmost derivation is stored in the stack.
    - Top of stack is the leftmost nonterminal symbol in the fragment of sentential form.
**Concept of LL(1) Parsing**
- Simulate leftmost derivation of the input.
- Keep part of sentential form in the stack.
- If the symbol on the top of stack is a terminal, try to match it with the next input token and pop it out of stack.
- If the symbol on the top of stack is a nonterminal $X$, replace it with $Y$ if we have a production rule $X \rightarrow Y$.
  
  Which production will be chosen, if there are both $X \rightarrow Y$ and $X \rightarrow Z$?

**Example of LL(1) Parsing**

$(n + (n)) * n$

**LL(1) Parsing Algorithm**

Push the start symbol into the stack

WHILE stack is not empty ($\$$ is not on top of stack) and the stream of tokens is not empty (the next input token is not $\$$)

SWITCH (Top of stack, next token)

CASE (terminal $a$, $a$):
- Pop stack;
- Get next token

CASE (nonterminal $A$, terminal $a$):
- IF the parsing table entry $M[A, a]$ is not empty THEN
  - Get $A \rightarrow X_1 \ldots X_n$ from the parsing table entry $M[A, a]$ Pop stack;
  - Push $X_n \ldots X_2 X_1$ into stack in that order
  - ELSE Error

CASE ($\$$,$\$$):
- Accept

OTHER: Error

**LL(1) Parsing Table**

If the nonterminal $N$ is on the top of stack and the next token is $t$, which production rule to use?

Choose a rule $N \rightarrow X$ such that

- $X \Rightarrow^* tY$ or
- $X \Rightarrow^* \lambda$ and $S \Rightarrow^* WNtY$

**First Set**

Let $X$ be $\lambda$ or be in $V$ or $T$.

First($X$) is the set of the first terminal in any sentential form derived from $X$.

- If $X$ is a terminal or $\lambda$, then First($X$) = \{ $X$ \}.
- If $X$ is a nonterminal and $X \rightarrow X_1 X_2 \ldots X_n$ is a rule, then
  - First($X_i$) - $\{ \lambda \}$ is a subset of First($X$)
  - First($X_i$) - $\{ \lambda \}$ is a subset of First($X$) if there's any $j < i$ where First($X_j$) contains $\lambda$
  - $\lambda$ is in First($X$) if for all $j \leq n$ First($X_j$) contains $\lambda$

**Examples of First Set**

$exp \rightarrow$ exp addop term | term
addop $\rightarrow$ + | -
term $\rightarrow$ term mulop factor | factor
mulop $\rightarrow$ *
factor $\rightarrow$ (exp) | num
First(exp) = $\{ +, - \}$
First(addop) = $\{ +, - \}$
First(mulop) = $\{ * \}$
First(factor) = $\{ (, num) \}$
First(term) = $\{ (, num) \}$
First(exp) = $\{ (, num) \}$

$st \rightarrow$ ifst | other
ifst $\rightarrow$ if (exp) st elsepart
elsepart $\rightarrow$ else st | $\lambda$
exp $\rightarrow$ 0 | 1
First(exp) = $\{ 0, 1 \}$
First(elsepart) = $\{ else, \lambda \}$
First(ifst) = $\{ if \}$
First(st) = $\{ if, other \}$
### Algorithm for finding First(A)

- For all terminals $a$, $\text{First}(a) = \{a\}$
- For all nonterminals $A$, $\text{First}(A) := \{}$
- While there are changes to any $\text{First}(A)$
  - For each rule $A \rightarrow X_1 X_2 \ldots X_n$
    - For each $X_i$ in $\{X_1, X_2, \ldots, X_n\}$
      - If for all $j < i$, $\text{First}(X_j)$ contains $\lambda$, then $\text{First}(A)$ contains $\text{First}(X_i) - \{\lambda\}$
      - If $\lambda$ is in $\text{First}(X_1), \text{First}(X_2), \ldots$, and $\text{First}(X_n)$ contain $\lambda$, then $\text{First}(A)$ also contains $\lambda$
  - If $A$ is a terminal or $\lambda$, then $\text{First}(A) = \{A\}$
  - If $A$ is a nonterminal, then for each rule $A \rightarrow X_1 X_2 \ldots X_n$, $\text{First}(A)$ contains $\text{First}(X_i) - \{\lambda\}$
  - If also for some $i < n$, $\text{First}(X_i), \text{First}(X_{i+1}), \ldots$, and $\text{First}(X_n)$ contain $\lambda$, then $\text{First}(A)$ contains $\text{First}(X_{i+1}) - \{\lambda\}$

### Finding First Set: An Example

<table>
<thead>
<tr>
<th>Symbol</th>
<th>First</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>(num)</td>
<td>$$</td>
</tr>
<tr>
<td>exp'</td>
<td>$\lambda$</td>
<td>$$</td>
</tr>
<tr>
<td>addop</td>
<td>$+$</td>
<td>$$</td>
</tr>
<tr>
<td>term</td>
<td>(num)</td>
<td>$$</td>
</tr>
<tr>
<td>factor</td>
<td>$\lambda$</td>
<td>$$</td>
</tr>
<tr>
<td>mulop</td>
<td>$*$</td>
<td>$$</td>
</tr>
</tbody>
</table>

### Follow Set

1. Let $\$ denote the end of input tokens
2. If $A$ is the start symbol, then $\$ is in $\text{Follow}(A)$.
3. If there is a rule $B \rightarrow X A Y$, then $\text{First}(Y) - \{\lambda\}$ is in $\text{Follow}(A)$.
4. If there is production $B \rightarrow X A Y$ and $\lambda$ is in $\text{First}(Y)$, then $\text{Follow}(A)$ contains $\text{Follow}(B)$.

### Algorithm for Finding Follow(A)

- $\text{Follow}(\$) = $\$
- FOR each $A$ in $V - \{S\}$
  - $\text{Follow}(A) = \{\}$
- WHILE change is made to some $\text{Follow}$ sets
  - FOR each production $A \rightarrow X_1 X_2 \ldots X_n$
    - FOR each nonterminal $X_i$
      - Add $\text{First}(X_{i+1} X_{i+2} \ldots X_n) - \{\lambda\}$ into $\text{Follow}(X_i)$
      - IF $\lambda$ is in $\text{First}(X_{i+1} X_{i+2} \ldots X_n)$ THEN add $\text{Follow}(A)$ to $\text{Follow}(X_i)$
  - IF $A$ is the start symbol, then $\$ is in $\text{Follow}(A)$.
  - IF there is a rule $A \rightarrow Y X Z$, then $\text{First}(Z) - \{\lambda\}$ is in $\text{Follow}(X)$.
  - IF there is production $B \rightarrow X A Y$ and $\lambda$ is in $\text{First}(Y)$, then $\text{Follow}(A)$ contains $\text{Follow}(B)$.

### Finding Follow Set: An Example

### Constructing LL(1) Parsing Tables

FOR each nonterminal $A$ and a production $A \rightarrow X$
FOR each token $a$ in $\text{First}(X)$
  - $A \rightarrow X$ is in $M(A, a)$
  - IF $\lambda$ is in $\text{First}(X)$ THEN
    - FOR each element $a$ in $\text{Follow}(A)$
      - Add $a$ to $M(A, a)$

Example: Constructing LL(1) Parsing Table

<table>
<thead>
<tr>
<th>First (exp)</th>
<th>Follow (exp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>+ - * n $</td>
</tr>
<tr>
<td>exp</td>
<td>exp' addop</td>
</tr>
<tr>
<td>term</td>
<td>term' factor</td>
</tr>
<tr>
<td>factor</td>
<td>mulop</td>
</tr>
</tbody>
</table>

| 1 exp → term exp' |
| 2 exp' → addop term exp' |
| 3 exp' → λ |
| 4 addop → + |
| 5 addop → - |
| 6 term → factor term' |
| 7 term' → mulop factor term' |
| 8 term' → λ |
| 9 mulop → * |
| 10 factor → ( exp ) |
| 11 factor → num |

LL(1) Grammar

A grammar is an LL(1) grammar if its LL(1) parsing table has at most one production in each table entry.

LL(1) Parsing Table for non-LL(1) Grammar

<table>
<thead>
<tr>
<th>First</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>+ - * num $</td>
</tr>
<tr>
<td>exp</td>
<td>1 1.2</td>
</tr>
<tr>
<td>term</td>
<td>3.4 3.4</td>
</tr>
<tr>
<td>factor</td>
<td>5 6</td>
</tr>
<tr>
<td>addop</td>
<td>7 8</td>
</tr>
<tr>
<td>mulop</td>
<td>9</td>
</tr>
</tbody>
</table>

First(exp) = { (, num )}
First(term) = { (, num )}
First(factor) = { (, num )}
First(addop) = { +, - }
First(mulop) = { * }

Causes of Non-LL(1) Grammar

What causes grammar being non-LL(1)?
- Left-recursion
- Left factor

Left Recursion

- Immediate left recursion
  - A → AX | Y  A=Y X*
  - A → AX_1 | AX_2 |...| AX_n
  - Y_1 | Y_2 |...| Y_m
  - A=(Y_1, Y_2,..., Y_m) (X_1, X_2,..., X_n)*
- General left recursion
  - A → X =⇒ X =⇒* A Y

Can be removed when there is no empty-string production and no cycle in the grammar.

Removal of Immediate Left Recursion

exp → exp + term | exp - term | term
term → term * factor | factor
factor → ( exp ) | num

Remove left recursion
exp → term exp’ exp’ → + term exp’ | - term exp’ | λ
term → factor term’ term’ → * factor term’ | λ
factor → ( exp ) | num
General Left Recursion

- **Bad News!**
  - Can only be removed when there is no empty-string production and no cycle in the grammar.

- **Good News!!!!**
  - Never seen in grammars of any programming languages

Left Factoring

- **Left factor causes non-LL(1)**
  - Given $A \rightarrow X Y \mid X Z$. Both $A \rightarrow X Y$ and $A \rightarrow X Z$ can be chosen when $A$ is on top of stack and a token in First(X) is the next token.

  \[ A \rightarrow X Y \mid X Z \]  
  \[ \text{can be left-factored as} \]  
  \[ A \rightarrow X A' \quad \text{and} \quad A' \rightarrow Y \mid Z \]

Example of Left Factor

ifSt $\rightarrow$ if ( exp ) st else | if ( exp ) st  
  can be left-factored as  
ifSt $\rightarrow$ if ( exp ) st elsePart  
elsePart $\rightarrow$ else st | \( \lambda \)  

seq $\rightarrow$ st ; seq | st  
  can be left-factored as  
seq $\rightarrow$ st seq'  
seq' $\rightarrow$ ; seq | \( \lambda \)

Bottom-up Parsing

- Use explicit stack to perform a parse  
- Simulate rightmost derivation (R) from left (L) to right, thus called LR parsing  
- More powerful than top-down parsing  
  - Left recursion does not cause problem  
  - Two actions  
    - Shift: take next input token into the stack  
    - Reduce: replace a string B on top of stack by a nonterminal A, given a production $A \rightarrow B$

Example of Shift-reduce Parsing

<table>
<thead>
<tr>
<th>Grammar</th>
<th>$S' \rightarrow S$</th>
<th>$S \rightarrow (S)S \mid \lambda$</th>
<th>Parsing actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Input</td>
<td>Action</td>
<td>Stack</td>
</tr>
<tr>
<td>$$</td>
<td></td>
<td></td>
<td>$$</td>
</tr>
<tr>
<td>$(())$</td>
<td>$$</td>
<td></td>
<td>$(())$</td>
</tr>
<tr>
<td>$((()))$</td>
<td>$(())$</td>
<td></td>
<td>$((()))$</td>
</tr>
<tr>
<td>$(())$</td>
<td>$$</td>
<td></td>
<td>$(())$</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$$</td>
<td></td>
<td>$(S)$</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$$</td>
<td></td>
<td>$(S)$</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$$</td>
<td></td>
<td>$(S)$</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$$</td>
<td></td>
<td>$(S)$</td>
</tr>
<tr>
<td>$S$</td>
<td>$$</td>
<td></td>
<td>$S$</td>
</tr>
<tr>
<td>$S$</td>
<td>10 $S'$</td>
<td>$S$</td>
<td></td>
</tr>
</tbody>
</table>

Reverse of rightmost derivation from left to right

Example of Shift-reduce Parsing

<table>
<thead>
<tr>
<th>Grammar</th>
<th>$S' \rightarrow S$</th>
<th>$S \rightarrow (S)S \mid \lambda$</th>
<th>Parsing actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Input</td>
<td>Action</td>
<td>Stack</td>
</tr>
<tr>
<td>$$</td>
<td></td>
<td>shift</td>
<td>$$</td>
</tr>
<tr>
<td>$(())$</td>
<td>$$</td>
<td>$(())$</td>
<td>$(())$</td>
</tr>
<tr>
<td>$(())$</td>
<td>$$</td>
<td>$(())$</td>
<td>$(())$</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$$</td>
<td>$(S)$</td>
<td>$(S)$</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$$</td>
<td>$(S)$</td>
<td>$(S)$</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$$</td>
<td>$(S)$</td>
<td>$(S)$</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$$</td>
<td>$(S)$</td>
<td>$(S)$</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$$</td>
<td>$(S)$</td>
<td>$(S)$</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$$</td>
<td>$(S)$</td>
<td>$(S)$</td>
</tr>
<tr>
<td>$S$</td>
<td>$$</td>
<td>$(S)$</td>
<td>$S$</td>
</tr>
<tr>
<td>$S$</td>
<td>$$</td>
<td>$(S)$</td>
<td>$S$</td>
</tr>
<tr>
<td>$S$</td>
<td>10 $S'$</td>
<td>accept</td>
<td>$S$</td>
</tr>
</tbody>
</table>

Viable prefix
**Terminologies**

- Right sentential form
  - sentential form in a rightmost derivation
- Viable prefix
  - sequence of symbols on the parsing stack
- Handle
  - right sentential form at position where reduction can be performed + production used for reduction
- LR(0) item
  - production with distinguished position in its RHS

**Shift-reduce parsers**

- There are two possible actions:
  - shift and reduce
- Parsing is completed when
  - the input stream is empty and
  - the stack contains only the start symbol
- The grammar must be augmented
  - a new start symbol $S'$ is added
  - a production $S' \rightarrow S$ is added
  - To make sure that parsing is finished when $S'$ is on top of stack because $S'$ never appears on the RHS of any production.

**LR(0) parsing**

- Keep track of what is left to be done in the parsing process by using finite automata of items
  - An item $A \rightarrow w . B \ y$ means:
    - $A \rightarrow w B \ y$ might be used for the reduction in the future,
    - at the time, we know we already construct $w$ in the parsing process,
    - if $B$ is constructed next, we get the new item $A \rightarrow w B \ . \ y$

**LR(0) items**

- LR(0) item
  - production with a distinguished position in the RHS
- Initial Item
  - Item with the distinguished position on the leftmost of the production
- Complete Item
  - Item with the distinguished position on the rightmost of the production
- Closure Item of $x$
  - Item $x$ together with items which can be reached from $x$ via $\lambda$-transition
- Kernel Item
  - Original item, not including closure items

**Finite automata of items**

Grammar:

- $S' \rightarrow S$
- $S \rightarrow (S)S$
- $S \rightarrow \lambda$

Items:

- $S' \rightarrow S$
- $S' \rightarrow S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow \lambda$

**DFA of LR(0) Items**

- LR(0) item
  - production with a distinguished position in the RHS
- Initial Item
  - Item with the distinguished position on the lefmost of the production
- Complete Item
  - Item with the distinguished position on the rightmost of the production
- Closure Item of $x$
  - Item $x$ together with items which can be reached from $x$ via $\lambda$-transition
- Kernel Item
  - Original item, not including closure items
**LR(0) parsing algorithm**

<table>
<thead>
<tr>
<th>Item in state</th>
<th>token</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>A -&gt; x.By where B is terminal</td>
<td>B</td>
<td>shift B and push state s containing A -&gt; xB.y</td>
</tr>
<tr>
<td>A -&gt; x.By where B is terminal</td>
<td>not B</td>
<td>error</td>
</tr>
<tr>
<td>A -&gt; x.</td>
<td>-</td>
<td>reduce with A -&gt; x (i.e., pop x, backup to the state s on top of stack) and push A with new state d(s,A)</td>
</tr>
<tr>
<td>S' -&gt; S.</td>
<td>none</td>
<td>accept</td>
</tr>
<tr>
<td>S' -&gt; S.</td>
<td>any</td>
<td>error</td>
</tr>
</tbody>
</table>

**LR(0) Parsing Table**

```
<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Rule</th>
<th>( a )</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>shift</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>reduce</td>
<td>A' -&gt; A</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>reduce</td>
<td>A -&gt; a</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>shift</td>
<td>2</td>
<td>3 4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>shift</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>reduce</td>
<td>A -&gt; (A)</td>
<td>3 2 4</td>
<td></td>
</tr>
</tbody>
</table>
```

**Example of LR(0) Parsing**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>(((a))$</td>
<td>shift</td>
<td>3</td>
</tr>
<tr>
<td>$0</td>
<td>3</td>
<td>(a)$</td>
<td>shift</td>
</tr>
<tr>
<td>$0</td>
<td>3</td>
<td>2</td>
<td>a$</td>
</tr>
<tr>
<td>$0</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>$0</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$0</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$0</td>
<td>4</td>
<td>1</td>
<td>$</td>
</tr>
</tbody>
</table>

**Non-LR(0) Grammar**

- Conflict
  - Shift-reduce conflict
    - A state contains a complete item A -> x, and a shift item A -> x.By
  - Reduce-reduce conflict
    - A state contains more than one complete items.

A grammar is a LR(0) grammar if there is no conflict in the grammar.

**SLR(1) parsing algorithm**

- Simple LR with 1 lookahead symbol
- Examine the next token before deciding to shift or reduce
  - If the next token is the token expected in an item, then it can be shifted into the stack.
  - If a complete item A -> x is constructed and the next token is in Follow(A), then reduction can be done using A -> x.
  - Otherwise, error occurs.
- Can avoid conflict

**SLR(1) parsing**

<table>
<thead>
<tr>
<th>Item in state</th>
<th>token</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>A -&gt; x.By where B is terminal</td>
<td>B</td>
<td>shift B and push state s containing A -&gt; xB.y</td>
</tr>
<tr>
<td>A -&gt; x.By where B is terminal</td>
<td>not B</td>
<td>error</td>
</tr>
<tr>
<td>A -&gt; x.</td>
<td>Follow(A)</td>
<td>reduce with A -&gt; x (i.e., pop x, backup to the state s on top of stack) and push A with new state d(s,A)</td>
</tr>
<tr>
<td>A -&gt; x.</td>
<td>not in Follow(A)</td>
<td>error</td>
</tr>
<tr>
<td>S' -&gt; S.</td>
<td>none</td>
<td>accept</td>
</tr>
<tr>
<td>S' -&gt; S.</td>
<td>any</td>
<td>error</td>
</tr>
</tbody>
</table>
**SLR(1) grammar**

Conflict
- Shift-reduce conflict
  - A state contains a shift item $A \rightarrow x.Wy$ such that $W$ is a terminal and a complete item $B \rightarrow z$ such that $W$ is in Follow($B$).
- Reduce-reduce conflict
  - A state contains more than one complete item with some common Follow set.

A grammar is an SLR(1) grammar if there is no conflict in the grammar.

---

**SLR(1) Parsing Table**

| State | ( a ) | $|$ | A |
|-------|-------|----|----|
| 0     | S3   | S2 | 1  |
| 1     | AC   |    |    |
| 2     | R2   |    |    |
| 3     | S3   | S2 | 4  |
| 4     | S5   |    |    |
| 5     | R1   |    |    |

---

**Disambiguating Rules for Parsing Conflict**

- **Shift-reduce conflict**
  - Prefer shift over reduce
    - In case of nested if statements, preferring shift over reduce implies most closely nested rule for dangling else
- **Reduce-reduce conflict**
  - Error in design

---

**Dangling Else**

<table>
<thead>
<tr>
<th>State</th>
<th>if</th>
<th>else</th>
<th>other</th>
<th>S</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S4</td>
<td>S3</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>R1</td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>R2</td>
<td>R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S4</td>
<td>S3</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S6</td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S4</td>
<td>S3</td>
<td>7</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>R4</td>
<td>R4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>