 Parsing

Outline

- Top-down v.s. Bottom-up
  - Top-down parsing
    - Recursive-descent 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?
Top–Down Parsing
- A parse tree is created from root to leaves
- The traversal of parse trees is a preorder traversal
- Tracing leftmost derivation
- Two types:
  - Backtracking parser
  - Predictive parser

Bottom–Up Parsing
- A parse tree is created from leaves to root
- The traversal of parse trees is a reversal of postorder traversal
- Tracing rightmost derivation
- More powerful than top-down parsing

Parse Trees and Derivations

Top-down parsing:
- $E \Rightarrow E + E$
- $\Rightarrow id + E$
- $\Rightarrow id + E * E$
- $\Rightarrow id + id * E$
- $\Rightarrow id + id * id$
- $E + E$
- $\Rightarrow E + E * E$
- $\Rightarrow E + E * id$
- $\Rightarrow id + id * id$

Bottom-up parsing:
- $E \Rightarrow E + E$
- $\Rightarrow id + E$
- $\Rightarrow id + E * E$
- $\Rightarrow id + id * E$
- $\Rightarrow id + id * id$
- $E + E$
- $\Rightarrow E + E * E$
- $\Rightarrow E + E * id$
- $\Rightarrow id + id * id$

TOP DOWN PARSING
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 guess based on?
- 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,
- If we choose \( E \to E \cdot O \cdot F \), it leads to infinitely recursive loops.

Rewrite the grammar into EBNF

```
procedure E
{ F;
  while (token==+ or token=-)
  { O; F; }
}
```

```
E ::= F \{O F\}
O ::= + | -
F ::= ( E ) | id
```

Procedure F
```
{ switch token
    { case (: match('(');
    E;
    match(')');
    case id: match(id);
    default: error;
    }
}
```

Procedure E
```
{ E; O; F; }
```

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 \( \lambda \)-production \( A \to \lambda \).
**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**

<table>
<thead>
<tr>
<th>E</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>X</td>
</tr>
<tr>
<td>( )</td>
<td>+</td>
</tr>
</tbody>
</table>

Finished

- \( E \rightarrow TX \)
- \( X \rightarrow ATX | \lambda \)
- \( A \rightarrow + | - \)
- \( T \rightarrow FN \)
- \( N \rightarrow MFN | \lambda \)
- \( M \rightarrow \cdot \)
- \( F \rightarrow (E) | n \)
**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 $)
SWITCH (Top of stack, next token)
   CASE (terminal a, a):
      Pop stack; Get next token
   CASE (nonterminal A, terminal a):
      IF the parsing table entry M[A, a] is not empty THEN
         Get A → X₁ X₂ ... Xₙ from the parsing table entry M[A, a]; Pop stack;
         Push Xₙ ... X₂ X₁ into stack in that order
      ELSE Error
   CASE ($,$): 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 → X
such that
   • X⇌* tY or
   • X⇌* λ and S⇌* WNY

**First Set**

Let X be λ 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 λ, then First(X) = {X}.
   • If X is a nonterminal and X → X₁ X₂ ... Xₙ is a
      rule, then
         • First(X₁) - {λ} is a subset of First(X)
         • First(X₁) - {λ} is a subset of First(X) if for all j<i
             First(X₁) contains {Xᵢ}
         • λ is in First(X) if for all j≠n First(X) contains λ.
### Examples of First Set

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>First Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>{0, 1}</td>
</tr>
<tr>
<td>exp'</td>
<td>{\lambda}</td>
</tr>
<tr>
<td>addop</td>
<td>{+,-}</td>
</tr>
<tr>
<td>term</td>
<td>{(), num}</td>
</tr>
<tr>
<td>term'</td>
<td>{*, (exp)}</td>
</tr>
<tr>
<td>mulop</td>
<td>{*}</td>
</tr>
<tr>
<td>factor</td>
<td>{(), num}</td>
</tr>
</tbody>
</table>

### Algorithm for finding First(A)

1. For all terminals $a$, First($a$) = \{a\}
2. For all nonterminals $A$, First($A$) := {}  
3. While there are changes to any First($A$)
4.   For each rule $A \rightarrow X_1 X_2 ... X_n$
5.      For each $X_i$ in \{X_1, X_2, ..., X_n\}  
6.         If for all $j < i$ First($X_j$) contains $\lambda$,  
7.               Then add First($X_i$)-{$\lambda$} to First($A$)  
8.         If $\lambda$ is in First($X_1$), First($X_2$), ..., and First($X_n$),  
9.               Then add $\lambda$ to First($A$)  

### Finding First Set: An Example

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>First Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>{0, 1}</td>
</tr>
<tr>
<td>exp'</td>
<td>{\lambda}</td>
</tr>
<tr>
<td>addop</td>
<td>{+,-}</td>
</tr>
<tr>
<td>term</td>
<td>{(), num}</td>
</tr>
<tr>
<td>term'</td>
<td>{*, (exp)}</td>
</tr>
<tr>
<td>mulop</td>
<td>{*}</td>
</tr>
<tr>
<td>factor</td>
<td>{(), num}</td>
</tr>
</tbody>
</table>
**Follow Set**

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

**Algorithm for Finding Follow(A)**

Follow(S) = {$$
FOR each A in V \{-S\}
Follow(A)=\{
WHILE change is made to some Follow sets
    FOR each production $A \rightarrow X_1 X_2 ... X_n$
        FOR each nonterminal $X_i$
            Add First($X_{i+1} X_{i+2} ... X_n$) - \{\lambda\} into Follow($X_i$).
(NOTE: If $im: X_{i+1} X_{i+2} ... X_n = \lambda$)
    IF $\lambda$ is in First($X_{i+1} X_{i+2} ... X_n$) THEN
        Add Follow(A) to Follow($X_i$).

If A is the start symbol, then $\$ \$ is in Follow(A).
If there is a rule $A \rightarrow Y X Z$, then First(Z) - \{\lambda\} is in Follow(X).
If there is production $B \rightarrow X A Y$ and $\lambda$ is in First(Y), then Follow(A) contains Follow(B).

**Finding Follow Set: An Example**

```
exp \rightarrow \text{term exp'}
exp' \rightarrow \text{addop term exp'} \mid \lambda.
addop \rightarrow + \mid -
term \rightarrow \text{factor term'}
term' \rightarrow \text{mulop factor term'} \mid \lambda.
mulop \rightarrow *
factor \rightarrow (\exp) \mid \text{num}
```

<table>
<thead>
<tr>
<th>First</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>(num $$)</td>
</tr>
<tr>
<td>exp'</td>
<td>$\lambda$ + $$</td>
</tr>
<tr>
<td>addop</td>
<td>$\lambda$ + $$</td>
</tr>
<tr>
<td>term</td>
<td>(num $\lambda$ - $$)</td>
</tr>
<tr>
<td>term'</td>
<td>$\lambda$ *</td>
</tr>
<tr>
<td>mulop</td>
<td>$\lambda$ *</td>
</tr>
<tr>
<td>factor</td>
<td>(num</td>
</tr>
</tbody>
</table>
Constructing LL(1) Parsing Tables

FOR each nonterminal A and a production A → X
FOR each token a in First(X)
    A → X is in M(A, a)
    IF $ is in First(X) THEN
        FOR each element a in Follow(A)
            Add A → X to M(A, a)

Example: Constructing LL(1) Parsing Table

<table>
<thead>
<tr>
<th>First</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( ) + - * n $</td>
</tr>
<tr>
<td>exp</td>
<td>exp'</td>
</tr>
<tr>
<td>exp'</td>
<td>exp</td>
</tr>
<tr>
<td>addop</td>
<td>term</td>
</tr>
<tr>
<td>term'</td>
<td>term</td>
</tr>
<tr>
<td>mulop</td>
<td>factor</td>
</tr>
<tr>
<td>factor</td>
<td>num</td>
</tr>
<tr>
<td>1 exp</td>
<td>term exp'</td>
</tr>
<tr>
<td>2 exp'</td>
<td>addop term exp'</td>
</tr>
<tr>
<td>3 exp</td>
<td>term</td>
</tr>
<tr>
<td>4 addop</td>
<td>addop</td>
</tr>
<tr>
<td>5 addop</td>
<td>addop</td>
</tr>
<tr>
<td>6 term</td>
<td>factor term'</td>
</tr>
<tr>
<td>7 term'</td>
<td>mulop factor term'</td>
</tr>
<tr>
<td>8 term'</td>
<td>term'</td>
</tr>
<tr>
<td>9 mulop</td>
<td>mulop</td>
</tr>
<tr>
<td>10 factor</td>
<td>factor</td>
</tr>
<tr>
<td>11 factor</td>
<td>factor</td>
</tr>
</tbody>
</table>

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></th>
<th>(</th>
<th>)</th>
<th>*</th>
<th>-</th>
<th>num</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>num</td>
<td>$</td>
</tr>
<tr>
<td>term</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>factor</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addop</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulop</td>
<td>9</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></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**
  - $A \rightarrow A X \ | \ Y \quad A=Y X^*$
  - $A \rightarrow A X_1 \ | \ A X_2 \ | \ldots \ | \ A X_n \ | \ Y_1 \ | \ Y_2 \ | \ldots \ | \ Y_m$
  - $A=(Y_1, Y_2, \ldots, Y_m) (X_1, X_2, \ldots, X_n)^*$
- **General left recursion**
  - $A \Rightarrow X \Rightarrow \ldots \Rightarrow Y \ A$
- **Can be removed very easily**
  - $A \rightarrow Y A' \ | \ X A' \ | \ X A' \ | \ A A' \ | \ \lambda$
  - $A \rightarrow Y_1 A' \ | \ Y_2 A' \ | \ldots \ | \ Y_m A' \ | \ A' \rightarrow X_1 A' \ | \ X_2 A' \ | \ldots \ | \ X_n A' \ | \ \lambda$
- **Can be removed when there is no empty-string production and no cycle in the grammar**
### Removal of Immediate Left Recursion

\[
\begin{align*}
\text{exp} & \rightarrow \text{exp} + \text{term} | \text{exp} - \text{term} | \text{term} \\
\text{term} & \rightarrow \text{term} \ast \text{factor} | \text{factor} \\
\text{factor} & \rightarrow (\text{exp}) | \text{num} \\
\text{Remove left recursion} \\
\text{exp} & \rightarrow \text{term} \text{exp}' \\
\text{exp}' & \rightarrow + \text{term} \text{exp}' | - \text{term} \text{exp}' | \lambda \\
\text{term} & \rightarrow \text{factor} \text{term}' \\
\text{term}' & \rightarrow \ast \text{factor} \text{term}' | \lambda \\
\text{factor} & \rightarrow (\text{exp}) | \text{num}
\end{align*}
\]

### 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 \( A \rightarrow X Y | X Z \). 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.

\[
A \rightarrow X Y | X Z
\]

can be left-factored as

\[
A \rightarrow X A' \text{ and } A' \rightarrow Y | Z
\]
Example of Left Factor

ifSt → if ( exp ) st else st | if ( exp ) st
    can be left-factored as
ifSt → if ( exp ) st elsePart
elsePart → else st | λ.

seq → st ; seq | st
    can be left-factored as
seq → st seq’
seq’ → ; seq | λ.

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 → B
### Example of Shift-reduce Parsing

#### Grammar

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

#### Parsing actions

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$(())$</td>
<td>shift</td>
</tr>
<tr>
<td>$(())$</td>
<td>$(())$</td>
<td>reduce $S \rightarrow \lambda$</td>
</tr>
<tr>
<td>$(())$</td>
<td>$S$</td>
<td>shift</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$(())$</td>
<td>reduce $S \rightarrow \lambda$</td>
</tr>
<tr>
<td>$((S)S)$</td>
<td>$(())$</td>
<td>reduce $S \rightarrow (S)S$</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$(S)$</td>
<td>shift</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$(S)$</td>
<td>reduce $S \rightarrow \lambda$</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$(S)$</td>
<td>reduce $S \rightarrow (S)S$</td>
</tr>
<tr>
<td>$S$</td>
<td>$S$</td>
<td>accept</td>
</tr>
</tbody>
</table>

#### Reverse of rightmost derivation

1. $(())$  
2. $((())$  
3. $((S)$  
4. $((S)$  
5. $((S)S)$  
6. $(S)$  
7. $(S)$  
8. $(S)$  
9. $(S)$  
10. $S' \Rightarrow S$

### Terminology

- **Right sentential form**
  - sentential form in a rightmost derivation

- **Viable prefix**
  - sequence of symbols on the parsing stack

- **Handle**
  - right sentential form + position where reduction can be performed + production used for reduction

- **LR(0) item**
  - production with distinguished position in its RHS

- **Viable prefix**
  - sequence of symbols on the parsing stack

- **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:

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

**DFA of LR(0) Items**

**LR(0) parsing algorithm**

<table>
<thead>
<tr>
<th>Item in state</th>
<th>token</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow x.B$ where $B$ is terminal $B$</td>
<td>shift $B$ and push state $s$ containing $A \rightarrow xB,y$</td>
<td></td>
</tr>
<tr>
<td>$A \rightarrow x.B$ where $B$ is terminal not $B$</td>
<td>error</td>
<td></td>
</tr>
<tr>
<td>$A \rightarrow x.$</td>
<td>reduce with $A \rightarrow x$ (i.e. pop $x$, backup to the state $s$ on top of stack) and push $A$ with new state $d(s,A)$</td>
<td></td>
</tr>
<tr>
<td>$S' \rightarrow S.$</td>
<td>none</td>
<td>accept</td>
</tr>
<tr>
<td>$S' \rightarrow S.$</td>
<td>any</td>
<td>error</td>
</tr>
</tbody>
</table>
LR(0) Parsing Table

Example of LR(0) Parsing

Non-LR(0)Grammar
**SLR(1) parsing**

- 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 $A \rightarrow x.$ is constructed and the next token is in $\text{Follow}(A)$, then reduction can be done using $A \rightarrow x.$
  - Otherwise, error occurs.
- Can avoid conflict

---

**SLR(1) parsing algorithm**

<table>
<thead>
<tr>
<th>Item in state</th>
<th>token</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow x. B \text{y}$ (B is terminal)</td>
<td>B</td>
<td>shift B and push state s containing $A \rightarrow xB.y$</td>
</tr>
<tr>
<td>$A \rightarrow x. B \text{y}$ (B is terminal)</td>
<td>not B</td>
<td>error</td>
</tr>
<tr>
<td>$A \rightarrow x.$ in $\text{Follow}(A)$</td>
<td></td>
<td>reduce with $A \rightarrow x.$ (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>$A \rightarrow x.$ not in $\text{Follow}(A)$</td>
<td></td>
<td>error</td>
</tr>
<tr>
<td>$S' \rightarrow S.$</td>
<td>none</td>
<td>accept</td>
</tr>
<tr>
<td>$S' \rightarrow S.$</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 $\text{Follow}(B)$.
  - Reduce-reduce conflict
    - A state contains more than one complete item with some common $\text{Follow}$ set.
- A grammar is an SLR(1) grammar if there is no conflict in the grammar.
**SLR(1) Parsing Table**

A → (A) | a

<table>
<thead>
<tr>
<th>State</th>
<th>(</th>
<th>a</th>
<th>)</th>
<th>$</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S3</td>
<td>S2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>AC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>R2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S3</td>
<td>S2</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>S5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SLR(1) Grammar not LR(0)**

S' → S
S → (S)S
S → .S
S → (.)S
S → .S
S → (S)S
S → .S
S → (S.)S
S → (S)S.

<table>
<thead>
<tr>
<th>S</th>
<th>(</th>
<th>)</th>
<th>$</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>S'</td>
<td></td>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>(S)S</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>(S)S</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>(S)S</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>(S)S</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>(S)S</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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>$</th>
<th>S</th>
<th>1</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>R1</td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>R2</td>
<td>R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S4</td>
<td>S3</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S6</td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>S4</td>
<td>S3</td>
<td>7</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>R4</td>
<td>R4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LALR(1) parsing

- Goal: reduce number of states in LR(1) parser.
- Some states LR(1) automaton have the same core items and differ only in the possible lookahead.
  - States $I_3$, $I_3'$, $I_5$, $I_5'$, $I_7$, $I_7'$, $I_8$ and $I_8'$
- We shrink our parser by merging such states.
- SLR: 10 states, LR(1): 14 states, LALR(1): 10 states

Conflicts in LALR(1) parsing

- Most conflicts that existed in LR(1) parser can be eliminated with LALR(1)
- Can LALR(1) parsers introduce conflicts that did not exist in the LR(1) parser?
  - Unfortunately YES.
  - BUT, only reduce/reduce conflicts.
- YACC generates LALR(1) parser