LR Parsing, Part 1

LR parsing concept
Using a parser generator
Parse Tree Generation

What is LR-parsing?
- L – Left-to-right scanning
- R – Rightmost derivation in reverse, i.e., bottom-up parsing
  - 12 <=rm <digit> 2 <=rm <no> 2 <digit> <no> <no> <=rm <number>

- LR(1) – LR parsing with 1 token lookahead
- LR(k) – LR parsing with k tokens lookahead

LR-parsing is the most general nonbacktracking shift-reduce parsing method known
An LR-parser detects a syntactic error as soon as possible

LR-Grammar Definition
A grammar for which a unique LR-table can be constructed is called an LR grammar (LR(0), SLR(1), LALR(1), LR(1), ...).
- No ambiguous grammars are LR grammars.
- There are unambiguous grammars which are not LR grammars.
- The state at the top of the stack contains all the information we need.

LR-Parsing is Bottom-up Parsing (from Lecture 5)
- Example: Bottom-up parsing with input: 3 - 6 * 2

Pushdown Automaton for LR-Parsing
Finite-state pushdown automaton
- Stack contains alternatingly states and symbols in N∪Σ (w.l.o.g.)
- 2 sorts of transitions:
  - Actions (shift, reduce, accept, error)
  - Goto’s (ε-transitions to find new state after reductions)
Configurations of the LR-Parse

- Configuration: (stack contents, remaining input)
  \( = (s_0 X_1 s_1 ... X_m s_m a_i a_{i+1} ... a_n) \)

- **Shift**: read current input symbol \( a_i \) and push it with new state \( s \):
  \( = (s_0 X_1 s_1 ... X_m s_m a_i s a_{i+1} ... a_n) \)

- **Reduce**: read \( \epsilon \), pop 2 stack symbols for handle \( X_{m-r+1} ... X_m \), push LHS nonterminal + new state (see below)

**Invariants**:
- Nonterminals on stack + remaining input \( (X_1 ... X_m a_i a_{i+1} ... a_n) \) is a rightmost-derived sentential form of \( G \).
- State on top of stack represents a viable prefix of \( G \)
  - Needs to be reconstructed after a reduce, using the GOTO table

**Example: An SLR(1) Grammar**

- **Terminals**: , a, b
- **Nonterminals**: <list> (or L) (is also the start symbol)
  - <element> (or E)

**Productions**:
1. <list> \( \rightarrow \) <list> , <element>
2. | <element>
3. <element> \( \rightarrow \) a
4. | b

Example (cont.): Extend Grammar with new start symbol

- **Terminals**: | (end-of-input symbol)
  - a, b
- **Nonterminals**: <SYS> (or S') (new start symbol)
  - <list> (or L)
  - <element> (or E)

**Productions**:
0. <SYS> \( \rightarrow \) <list> |
1. <list> \( \rightarrow \) <list> , <element>
2. | <element>
3. <element> \( \rightarrow \) a
4. | b

**Example: Tables (given)**

Parsing input string a,b

**GOTO table**:

<table>
<thead>
<tr>
<th>State</th>
<th>L</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**ACTION table**:

<table>
<thead>
<tr>
<th>State</th>
<th>L</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a, b</td>
<td>S4</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>S2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>R1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>R3</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>R3</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>R3</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>R2</td>
</tr>
</tbody>
</table>

Step: Stack: Input: Table entries
1. ---| 0 | a, b | ACTION[ 0, a ] = S4
2. ---| 0 | a | ACTION[ 0, a ] = S4
3. ---| 0 | a | ACTION[ 0, a ] = S4
4. ---| 0 | a | ACTION[ 0, a ] = S4
5. ---| 0 | a | ACTION[ 0, a ] = S4
6. ---| 0 | a | ACTION[ 0, a ] = S4

Reconstruct successor state after a Reduce from top stack items via the GOTO table
Handle, Viable Prefix

- Consider a rightmost derivation $S \Rightarrow^*_{\alpha} \beta | \alpha u \Rightarrow^*_{\beta} \beta | \alpha u$ for a context-free grammar $G$.
- $\alpha$ is called a handle of the right sentential form $\beta | \alpha u$, associated with the rule $X \Rightarrow^*_{\alpha} \alpha$.
- Each prefix of $\alpha \beta$ is called a viable prefix of $G$.

**Example:** Grammar $G$ with productions \( S \rightarrow aSb \mid c \)
- Right sentential forms: e.g., c, abc, aSb, aaaaSbbbb, ....
- For c: Handle: c Viable prefixes: c, c
- For abc: Handle: c Viable prefixes: c, a, ac
- For aSb: Handle: aSb Viable prefixes: c, a, aS, aSb
- For aaSb: Handle: aSb Viable prefixes: c, a, aa, aaS, aaSb
- ...

Recognizing Handles

- How to recognize if a handle appears as the top elements on the stack?
  - **Naive approach:** Examine the entire stack (e.g., from top to bottom, or vice versa) at every step
  - Leads to unnecessarily long worst-case parsing time
  - Need to actually store grammar symbols on stack
  - **Idea:** Incremental handle recognition
    - Keep information about partially recognized handles (= viable prefixes) on top of stack, encoded in state
    - Characteristic automaton (NFA) to recognize viable prefix
    - DFA by subset construction of $\varepsilon$-closure
    - Parser tables (ACTION / GOTO)

Rightmost Derivation, Handle

(from Lecture 3)

- Reverse rightmost derivation
  - $S \Rightarrow_{\alpha} aA \Rightarrow_{\beta} b \Rightarrow_{\epsilon} \epsilon$ such that $\alpha \Rightarrow_{\hat{\epsilon}} \beta \Rightarrow_{\hat{\epsilon}} \epsilon$.
- **Handles**
  - Consist of two parts:
    1. A production $A \rightarrow \beta$
    2. $\alpha \hat{\Rightarrow}_0 \beta$
  - Example: The handle of $<\text{no}> <\text{digit}> 2$ is the production $\Rightarrow_{\epsilon} \Rightarrow_{\text{digit}}$ together with the position after $\text{digit}$ is a handle of $\text{digit}$.

Small Exercise on LR-parsing

---

GOTO table: state L E
0 1 6
1 1 3
2 3
3 2
4 5
5 4
6 3
7 2
8 1

GOTO table: state L E
0 1 6
1 1 3
2 3
3 2
4 5
5 4
6 3
7 2
8 1

ACTION table:

---

Step Stack Input Table entries
1 $\Rightarrow_{\epsilon} a, b \Rightarrow_{\epsilon} a, b \Rightarrow_{\epsilon} a, b$
2 $\Rightarrow_{\epsilon} a, b \Rightarrow_{\epsilon} a, b \Rightarrow_{\epsilon} a, b$
3 $\Rightarrow_{\epsilon} a, b \Rightarrow_{\epsilon} a, b \Rightarrow_{\epsilon} a, b$

... and so on ...

---

Example: Tables (given);
Parsing input string $a,b$

Step Stack Input Table entries
1 $\Rightarrow_{\epsilon} a, b \Rightarrow_{\epsilon} a, b \Rightarrow_{\epsilon} a, b$
2 $\Rightarrow_{\epsilon} a, b \Rightarrow_{\epsilon} a, b \Rightarrow_{\epsilon} a, b$
3 $\Rightarrow_{\epsilon} a, b \Rightarrow_{\epsilon} a, b \Rightarrow_{\epsilon} a, b$

...
A NFA Recognizing Viable Prefixes

- A.k.a. the "characteristic finite automaton" for a grammar G
- States: LR(0) items (= context-free items) of extended grammar
- Input stream: The grammar symbols on the stack
- Start state: [S']  Final state: [S']
- Transitions:
  - "move dot across symbol" if symbol found next on stack: 
    A\rightarrow\alpha B\gamma 
    to  A\rightarrow\alpha B\beta
  - \epsilon-transitions to LR(0)-items for nonterminal productions 
    from items where the dot precedes that nonterminal:
    A\rightarrow\alpha B\gamma
- (Example and construction of DFA later, in Lecture 7)

Using a Parser Generator

- Example: UNIX Yacc, GNU Bison for LALR(1) grammars

Example Grammar for Yacc

```c
yacc mygrammar.y
```

Parser Executable

```c
y.tab.c
```

Using a Parser Generator

- Example: UNIX Yacc, GNU Bison for LALR(1) grammars

Example Grammar for Yacc

```c
yacc mygrammar.y
```

Using a Parser Generator

- Example: UNIX Yacc, GNU Bison for LALR(1) grammars

Example Grammar for Yacc

```c
yacc mygrammar.y
```

Using a Parser Generator

- Example: UNIX Yacc, GNU Bison for LALR(1) grammars

Example Grammar for Yacc

```c
yacc mygrammar.y
```
Construction of parse tree (Bottom-up)

- **Shift operations:**
  
  Create a one-node tree containing the shifted symbol.

- **Reduce operations:**
  
  When reducing a handle β to A (as in A → β), create a new node A whose "children" are those nodes that were created in the handle.

- During the analysis we have a forest of sub-trees. Each entry in the stack points to its corresponding sub-tree. When we accept, the whole parse tree is completed.

---

### Parse Tree Construction

#### Parsing input string a,b

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Input</th>
<th>Table entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-L,E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>a</td>
<td>ACTION[1, a]</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>,b</td>
<td>ACTION[1, ,b]</td>
</tr>
</tbody>
</table>

**ACTION table:**

- **s → L → L → E:**
  - L → L, E
  - E → a

**Parse Tree Construction**

#### Parsing input string a,b

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Input</th>
<th>Table entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-L,E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>a</td>
<td>ACTION[1, a]</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>,b</td>
<td>ACTION[1, ,b]</td>
</tr>
</tbody>
</table>

**ACTION table:**

- **s → L → L → E:**
  - L → L, E
  - E → a

**Reduce [X → a]:** Create a new tree node for X whose children are those former root nodes pointed to from the handle elements' semantic stack fields (in Yacc: $1, $2, ...)

**Reduce [X → b]:** Create a new tree node for X whose children are those nodes that were created in the handle.

**During parsing:** Forest of sub-trees, roots pointed from the semantic stack.

**At accept [S’ → S.]:** Emit the parse tree computed for S.

---

**Small Exercise on Parse Tree Construction**

- **Parse Tree Construction:**
  - During parsing, a forest of sub-trees is created. Each entry in the stack points to its corresponding sub-tree. When we accept, the whole parse tree is completed.
  - **Shift operations:** Create a one-node tree containing the shifted symbol.
  - **Reduce operations:** When reducing a handle β to A (as in A → β), create a new node A whose "children" are those nodes that were created in the handle.

---

**Notes:**

- The notes and figures are related to the construction and parsing of a parse tree in the context of compiler construction.
- The parse tree is constructed in a bottom-up manner, starting from the input string and building up to the root node, which is the start symbol S.
- The process involves shifting and reducing symbols according to the grammar rules and the ACTION and GOTO tables.
- The parse tree is a visualization of the parse tree construction process, showing the structure of the parsed input.
- The figures illustrate the steps involved in the parse tree construction, including the shift and reduce operations, the stack entries, and the parse tree structure.
- The notes explain the concepts of shift and reduce operations, the forest of subtrees, and the role of the GOTO table in determining the next action.

---

**References:**

- The source of the text and figures is likely a textbook or course material on compiler construction, as indicated by the mention of "TDDD55/TDDB44 Compiler Construction, 2011 P. Fritzson, C. Kessler, IDA, Linköpings universitet."