Parsing
Tomita’s Parser: Generalized LR Parsing

Laura Kallmeyer

Heinrich-Heine-Universität Düsseldorf

Winter 2017/18
Table of contents

1. Motivation
2. Graph-structured stack
3. The parse forest
4. Conclusion
Motivation

- LR-parsing with one lookahead is deterministic for LR(1) grammars. But there are CFLs that cannot be generated by LR(1)-grammars.
- If a grammar is not LR(1), we can construct a LR(1) parse table with more than one entry in some of the fields. This can be used for non-deterministic parsing.
- However, since we don’t have tabulation, partial results get computed several times and the complexity is exponential.
- Tomita’s idea: Use a graph-structured stack to avoid computing partial results more than once.

Tomita’s parser is an LR parser with tabulation
Graph-structured stack (1)

The stack is a directed acyclic graph (DAG) with the leaves being the topmost elements.

A directed acyclic graph consists of

- A set of nodes (or vertices) \( V \) (here finite), and
- a set of edges \( E \subseteq V \times V \), such that
  a) for all \( v \in V \): \( \langle v, v \rangle \notin E \), and
  b) for every sequence \( v_1, \ldots, v_k \in V \) with \( \langle v_1, v_2 \rangle, \ldots, \langle v_{k-1}, v_k \rangle \in V \): \( v_1 \neq v_k \).

In our case, the vertices of the DAG are labelled with states, non-terminals or terminals.
Our parsing is incremental, i.e., processes the input one by one from left to right.

For every word in the input, before processing that word, we have $k$ possible states.

- We first do the possible reductions for each of the states while leaving the original stack if there is a shift possible. In case of a reduce/reduce or shift/reduce conflict, we branch. If several branches lead to the same states, we identify these.

- We then do the possible shifts. Again, if several lead to the same states, we identify these.
### Graph-structured stack (3)

Example: 1. \( S \rightarrow AB \), 2. \( S \rightarrow SC \), 3. \( B \rightarrow BC \),
4. \( A \rightarrow a \), 5. \( B \rightarrow b \), 6. \( C \rightarrow c \)

<table>
<thead>
<tr>
<th></th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>$</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>s6</td>
<td>acc</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parse</td>
<td>r1, s6</td>
<td>r1</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>table:</td>
<td>4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>r6</td>
<td>r6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>r2</td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For input $w = abcc$, at some point (after shifting the first $c$) the stack is the following:

```
S ← 2
0  ← 2  c ← 6
A ← 1  ← B ← 3
```
Problems (infinite loops) in generalized LR parsing can arise from

- Loops: Productions $A \rightarrow B$, $B \rightarrow A$ would lead to an infinite reduce-loop.

- Hidden left-recursion: $A \rightarrow \alpha A\beta$ with $\alpha \Rightarrow^* \epsilon$ would lead to an infinite loop of reducing $\epsilon$ to $\alpha$ since $A \rightarrow \alpha \bullet A\beta$ and $A \rightarrow \bullet \alpha A\beta$ would be in the same state.
The parse forest (1)

- The dag-structure avoids an explosion in the number of stacks.
- However, we can still have exponentially many parse trees for a given input.
- Therefore, a compact representation of parse forests is needed.
- Tomita uses two techniques: sub-tree sharing and local ambiguity packing.
The parse forest (2)

Example: Take the preceding grammar, \( w = abcc \)

Three parse trees:

Sub-tree sharing: Common sub-trees are represented only once.
Result of sub-tree sharing:

Local ambiguity packing: whenever the same category spans the same input (possibly with different analyses), the corresponding nodes are put into one **packed node**.
The parse forest (4)

Result of local ambiguity packing:

```
  S S
   / \
  /   \
 S   B
   / \
  /   \
A   B
   / \ \
  /   \ \
 a   b  C
```

```
  C
```

```
  C
```
Packed parse forests are easy to construct within an LR-parser with graph-structured stack: Whenever a subtree is shared or different subtrees are packed into one node, there will be a corresponding shared node in the stack graph. More precisely,

- Whenever a node is shared, we create a shared sub-tree, and
- whenever two or more branches get identified into a single new branch, we create a packed node.

Instead of non-terminals or terminals we use pointers to identifiers of parse trees as stack vertex labels. This way, in different places we can have pointers to the same parse tree.
Example: $w = abc$.

<table>
<thead>
<tr>
<th>Stack</th>
<th>analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s4</td>
</tr>
<tr>
<td>0</td>
<td>1 4 r4</td>
</tr>
<tr>
<td>0</td>
<td>2 1 s5</td>
</tr>
<tr>
<td>0</td>
<td>2 1 3 5 r5</td>
</tr>
<tr>
<td>0</td>
<td>2 1 4 3 r1, s6</td>
</tr>
<tr>
<td>0</td>
<td>2 1 4 3 s6</td>
</tr>
</tbody>
</table>

- 5 2 s6 5: S(2,4)
The parse forest (7)

```
0   2   1   4   3   6   6   r6
     5   2
6: c

0   2   1   4   3   7   8   r3
     5   2   7   7   r2
7: C(6)

0   2   1   8   3   r1
     9   2   acc
8: B(4,7), 9: S(5, 7)

0   11   2   acc
10: S(2,8), 11: [10, 9]
```
Conclusion

Tomita’s algorithm

- is a general LR(1) parser that works for every CFG;
- uses a graph-structured stack to avoid the explosion otherwise linked to non-determinism;
- uses a compact parse forest representation to avoid the explosion arising from ambiguous grammars.

Reference: