Bottom-Up Parsing

A. Pnueli

Honors Compilers, NYU, Fall, 2009

Lecture 4: Bottom-Up Parsing

A. Pnueli

Honors Compilers, NYU, Fall, 2009

Lecture 4: Bottom-Up Parsing

Bottom-Up Parsing

Handles

We may encounter sentential forms with two handles, e.g.

\[
p \mid q \mid r \mid s \leftarrow T
\]

On the other hand, for an unambiguous grammar such as

\[
in an unambiguous grammar the handle is always uniquely defined.
\]

and there exist a rightmost derivation

A production \( A \) is a handle in a sentential form if \( A \) is a non-terminal

String that may appear in a rightmost derivation.

A (right) sentential form is a mixed (terminals and non-terminals)

Techniques more general than LL(\( k \)).

A production forms two actions:

from left to right.

Synthesize free from fragments, reconstruct rightmost derivation

\( S \)

Bottom-Up Parsing

The operations here are shift (copy from input to stack) and reduce (invert a production).

The operations here are shift (copy from input to stack) and reduce (invert a production).

bottom-up recognition by a shift-reduce PDA is given by:

 reversing a rightmost derivation.

bottom-up recognition by a shift-reduce PDA

A bottom-up recognition by a shift-reduce PDA is given by:

\[
((())) \leftarrow ((S)) \leftarrow (SS) \leftarrow (S) \leftarrow S
\]

Shift | Input | Action
--- | --- | ---
| | (()) | shift
| | (()()) | shift, shift
| | SS | reduce S
| | SS | reduce S
| | () | reduce S
| | (S) | reduce S
| | () | reduce S
| | (()) | reduce S
| | S | accept

Automaton starts with an empty stack.

Automaton equals to \( S \) or \( \epsilon \).

\( S \)

The operations here are shift (copy from input to stack) and reduce (invert a production).

Shift | Input | Action
--- | --- | ---
| | (()) | shift
| | (()()) | shift, shift
| | SS | reduce S
| | SS | reduce S
| | () | reduce S
| | (S) | reduce S
| | () | reduce S
| | (()) | reduce S
| | S | accept

Automaton stops when all input has been consumed and the stack equals to \( S \) or \( \epsilon \).

reversing a rightmost derivation.

Bottom-Up Parsing

Handles

Techniques more general than LL(\( k \)).

A production forms two actions:

from left to right.

Synthesize free from fragments, reconstruct rightmost derivation

### Bottom-Up Parsing

- **Techniques more general than LL(\( k \)).**
- **Handles:**
  - If accepts when all input has been consumed and the stack equals to \( S \) or \( \epsilon \).
  - Automaton starts with an empty stack.
  - Automaton equals to \( S \) or \( \epsilon \).
  - \( S \)
  - The operations here are shift (copy from input to stack) and reduce (invert a production).
  - Automaton stops when all input has been consumed and the stack equals to \( S \) or \( \epsilon \).
  - Automaton forms two actions:
    - from left to right:
      - Synthesize free from fragments, reconstruct rightmost derivation
      - **Handles:**
  - **Handles:**
LR parsing: we expect all stack contents to be viable prefixes. In an LR parsing, we refer to a prefix of a reducible string as a viable prefix. In an LR grammar, the set of reducible strings is a regular language.

Claim 7. [Regularity of reducible strings] The set of reducible strings is a regular language.

Reduction:

The main problem in the implementation of a shift-reduce parser is to determine whether the current stack is a reducible string. If it is a reducible string, we refer to it as a reducible string.

For each rightmost derivation \( S = A y \), we refer to \( y \) as a reducible string.

On the other hand, the language \( \{ x \in \mathcal{L}(G) : \exists y \text{ derivable from } x \text{ such that } x = A y \} \) does not uniquely determine whether the handle is \( C \) or \( A \), which generalizes the language \( \{ 0 < \ell \ \mid \text{prefix of } \ell \text{ is reducible} \} \cap \{ 0 < \ell \ \mid \text{prefix of } \ell \text{ is reducible} \} \).

Propositions of LR(\( k \)) Grammars

There exist unambiguous grammars which are not LR(\( k \)).

Properties of LR(\( k \)) Grammars

There exist unambiguous grammars which are not LR(\( k \)).

otherwise, reject.

If the stack contains \( S \), and the input is empty, then accept.

If the stack contains \( A \), and the input is not empty, then shift.

If the stack contains \( \epsilon \), and the stack is not empty, then reduce.

Thus means that if there are two rightmost derivations

Characters beyond the handle.

An unambiguous grammar is called an LR(\( k \)) grammar.

### Implementation of Shift-Reduce Parsing

We use a stack to hold the prefix of the analyzable sentential form up to the handle. Initially, the stack is empty.

If the stack contains \( S \), and the input is empty, then accept.

Otherwise, reject.

If the next input token is a right sentential form – shift the next input token to the stack.

If the stack contains a non-terminal at its top, but is still a prefix of a rightmost derivation, the corresponding non-terminial grammar, this may require looking at the first \( k \) input characters.

Then we follow the rules:

1. If the stack contains \( A \), and the input is not empty, then reduce.
2. If the stack contains \( \epsilon \), and the stack is not empty, then reduce.
3. If the stack contains \( S \), and the input is empty, then accept.
4. Otherwise, reject.

For each rightmost derivation \( S = A y \), we refer to \( y \) as a reducible string.

The main problem in the implementation of a shift-reduce parser is to determine whether the current stack is a reducible string. If it is a reducible string, we refer to it as a reducible string.

For each rightmost derivation \( S = A y \), we refer to \( y \) as a reducible string.

On the other hand, the language \( \{ x \in \mathcal{L}(G) \) derivable from \( x \) such that \( x = A y \} \) does not uniquely determine whether the handle is \( C \) or \( A \), which generalizes the language \( \{ 0 < \ell \ \mid \text{prefix of } \ell \text{ is reducible} \} \cap \{ 0 < \ell \ \mid \text{prefix of } \ell \text{ is reducible} \} \).

Propositions of LR(\( k \)) Grammars

There exist unambiguous grammars which are not LR(\( k \)).

Properties of LR(\( k \)) Grammars

There exist unambiguous grammars which are not LR(\( k \)).
A Classifier via Right Linear Grammar

We will show how to construct a finite-state classifier for the set of reducible strings of an LR(0) grammar. This is a DFA with several accepting states. Each accepting state identifies the production \( A \rightarrow \beta \) which has been applied last when generating the reducible string \( \alpha \beta \) by a rightmost derivation.

For each production \( X \rightarrow Y_1 \cdots Y_n \), for each non terminal \( Y_i \), construct a right-linear rule

\[ \langle X \rangle \rightarrow Y_1 \cdots Y_{i-1} \langle Y_i \rangle \]

Eliminate any productions of the form \( \langle X \rangle \rightarrow \langle X \rangle \).

\[ \langle X \rangle \rightarrow Y_1 \cdots Y_n \{ X \rightarrow Y_0 \} \]

Construct also the rule

\[ \langle X \rangle \rightarrow \]

Out of this grammar we construct an NFA and then determinize it into a DFA.
Initially, stack is empty.

At any step, if stack contains $E_0$ (start symbol) and input is empty, accept.

Otherwise, let $a$ be the next incoming input character, and $I_j$ be the (classier) state at the top of the stack.

4. Any step: if stack contains $I_0$ and input is empty, accept.

5. Initially, stack contains state $I_0$.

Replace Symbol Stack by State Stack

Instead of running the LR(0) classier on the stack at each step, we can keep a stack of the classier states. This leads to the following parsing algorithm:

1. Initially, stack contains state $I_0$.

2. At any step, if stack contains $I_j$ and input is empty, accept.

3. Initially, stack is empty.

Example of a Parse

<table>
<thead>
<tr>
<th>Action</th>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a$</td>
<td>$a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A$</td>
<td>$a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
</tbody>
</table>

This parse cannot be completed by pure LR(0) parsing. To resolve, switch to SLR(1) parsing.

<table>
<thead>
<tr>
<th>Action</th>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a$</td>
<td>$a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A$</td>
<td>$a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
</tbody>
</table>

This parse can be completed by pure SLR(1) parsing. To resolve, switch to SLR(1) parsing.

<table>
<thead>
<tr>
<th>Action</th>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a$</td>
<td>$a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A$</td>
<td>$a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
<tr>
<td>reduce by $A \rightarrow a$</td>
<td>$a \rightarrow A + a$</td>
<td>$a + a$</td>
</tr>
</tbody>
</table>
For every non-terminal $A$ such that $\exists a \in \text{Follow}(A)$, $I_f = I$ set $\text{GOTO}[I_f,a]$ to 

$\text{ERROR}$. 

For every non-terminal $A$ such that $\forall a \in \text{Follow}(A)$, $I_f = I$ set $\text{REDUCE}$. 

For every terminal $a$, if $I_f = (I,Y)$, then set 

$\text{GOTO}[I_f,a]$ to shift. 

For every terminal $a$, if $I_f = (I,Y)$, then set $\text{REDUCE}$. 

The parsing actions are determined as follows: 

- State $I$ corresponds to $I_f$. 
- The parsing actions for the grammar:
  
  **Constructing the LR(0) classifier for the grammar:** 

**LR Parsing Tables**

**Parsing Tables**

**A Parse with a State Stack**

**Table for Arithmetic Expressions**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Accept</th>
<th>Shift</th>
<th>Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$</td>
<td>$L + E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$*$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$($</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$)$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$+$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$*$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$($</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$)$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$+$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$*$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$($</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$)$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
</tbody>
</table>

**A Parse with a State Stack**

**Table for Arithmetic Expressions**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Accept</th>
<th>Shift</th>
<th>Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$</td>
<td>$L + E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$*$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$($</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$)$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$+$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$*$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$($</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$)$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$+$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$*$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$($</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$)$</td>
<td>$E$</td>
<td>$E$</td>
<td>$E$</td>
</tr>
</tbody>
</table>
A grammar is SLR(0) if its LR(0) classifier satisfies the following requirements:

- For every accepting state $I$ annotated by production $A \rightarrow \alpha$ and terminal $a$ such that $(I, a)$ is defined,
  
  $$ A \cap \text{Follow}(A) \neq \emptyset $$

- For every accepting state $I$ annotated by production $A \rightarrow \alpha$ and terminal $a$ such that $I \cap a$ is defined,
  
  $$ \forall A \in \text{Follow}(A) \cup \text{Follow}(a) \ ; \ I \cap a' \rightarrow a' $$

That is, the classifier has no shift-reduce conflict and all reduce-reduce conflicts are resolved by the function `follow`. Use of ambiguous grammars follow.
When SLR(0) parsing is inadequate, there are LR grammars which are not SLR(0).

Consider the following grammar:

\[ S \rightarrow aAb \]
\[ A \rightarrow c \]
\[ B \rightarrow c \]

Let us construct an LR(0) classier for this grammar. As a first step, let us construct an LR(1) classier which accepts the string \( av \) under the mode \( \delta \):

\[ \delta \rightarrow \varepsilon \]

For every rightmost derivation, we can extend the analysis of sentential forms to full consideration of the terminal immediately following the handle. When we can extend the analysis of sentential forms to full consideration of the terminal immediately following the handle, we can extend the classier to be a DFA which, in order to construct an LR(1) classier, we generate.

The LR(1) classier for the problem grammar is generated as follows:

Assume a context-free grammar \( G \). The right-linear grammar \( G_{LR(1)} \) is generated as follows:

\[ \{ \{p, e\} = \{p, a\} \} \]

The FOLLOW function is generated as follows:

\[ \text{Note that we have a conflict at state } I_2 \text{ which cannot be resolved by the FOLLOW function.} \]

\[ \langle p \rangle \rightarrow a \]
\[ \langle p \rangle \rightarrow a \]

An LR(0) classier for the problem grammar:

When LR(0) parsing is inadequate, we construct a right-linear grammar for all rightmost sentential forms. We can extend the analysis of sentential forms to full consideration of the terminal immediately following the handle. We can extend the classier to be a DFA which, in order to construct an LR(1) classier, we generate.

\[ \delta \rightarrow \varepsilon \]

For every rightmost derivation, we can extend the analysis of sentential forms to full consideration of the terminal immediately following the handle. When we can extend the analysis of sentential forms to full consideration of the terminal immediately following the handle, we can extend the classier to be a DFA which, in order to construct an LR(1) classier, we generate.
Applying to the Problem Grammar

Reconsider the previously considered problem grammar:

\[
S \rightarrow aAd \\
S \rightarrow bBd \\
S \rightarrow aBe \\
S \rightarrow bAe \\
A \rightarrow c \\
B \rightarrow c \\
S \rightarrow C \rightarrow cC \\
S \rightarrow D \rightarrow cD \\
C \rightarrow cC \\
D \rightarrow cD \\
\]

Constructing the induced grammar \( G'_{LR(1)} \) we obtain:

\[
\begin{align*}
P & \rightarrow \varnothing \rightarrow \varnothing \\
P & \rightarrow S \rightarrow \varnothing \\
S & \rightarrow \varnothing \rightarrow S \\
S & \rightarrow C \rightarrow \varnothing \\
S & \rightarrow D \rightarrow \varnothing \\
C & \rightarrow \varnothing \rightarrow C \\
C & \rightarrow \varnothing \rightarrow \varnothing \\
D & \rightarrow \varnothing \rightarrow D \\
D & \rightarrow \varnothing \rightarrow \varnothing \\
\end{align*}
\]

The Resulting \( LR(1) \) Classifier

For every non-terminal \( A \) such that \( (I,A) \) set \( \text{GOTO} \) to err.

- All entries not set by the above rules are set to err.
- If any conflicting actions result from the above rules, then the grammar is not \( LR(1) \).
- If \( I \) is an accepting state annotated by \( \text{ACTION}[a] \) then set action \( a \rightarrow \varnothing \).
- If \( I \) is an accepting state annotated by \( A \rightarrow \varnothing, \text{ACTION}[a] \) to shift.
- Every terminal \( a \) follows: \( \text{ACTION}[a] = I \), the parsing actions are determined as follows.
- State \( I \) corresponds to \( I \) for the grammar.

Constructing the \( LR(1) \) classifier for the grammar:

\[
\begin{align*}
\text{GOTO} & = [A \rightarrow C, C \rightarrow D, D \rightarrow C] \\
\text{ACTION} & = [A \rightarrow \varnothing, C \rightarrow \text{shift}] \\
\end{align*}
\]

Reconsidering the previously considered problem grammar:

\[
\begin{align*}
S & \rightarrow aAd \\
S & \rightarrow bBd \\
S & \rightarrow aBe \\
S & \rightarrow bAe \\
A & \rightarrow c \\
B & \rightarrow c \\
S & \rightarrow C \rightarrow cC \\
S & \rightarrow D \rightarrow cD \\
C & \rightarrow cC \\
D & \rightarrow cD \\
\end{align*}
\]
Note that we can merge together states $I_4$ with $I_7$ and state $I_8$ with $I_9$ without creating conflicts (but giving up some error-detection options). This leads to an LALR classier.