Parsing review

**Top-down parsers**

- start at the root of derivation tree and fill in
- choosing production for nonterminal is the key problem
- LL (predictive) parsers use lookahead to choose production

**Bottom-up parsers**

- start at leaves and fill in
- choosing right-hand side (rhs) of production is the key problem
- LR parsers use current stack and lookahead to choose production

LR(k) parsers are more powerful than LL(k) parsers because they can see the entire rhs before choosing a production.
Some definitions

For a grammar $G$, with start symbol $S$, any string $\alpha$ such that $S \Rightarrow^* \alpha$ is called a sentential form.

- If $\alpha$ contains only terminal symbols, $\alpha$ is a sentence in $L(G)$.
- If $\alpha$ contains one or more non-terminals, it is just a sentential form (not a sentence in $L(G)$).

A left-sentential form is a sentential form that occurs in the leftmost derivation of some sentence.

A right-sentential form is a sentential form that occurs in the rightmost derivation of some sentence.
Bottom-up parsing

Goal:

Given an input string $w$ and a grammar $G$, construct a parse tree by starting at the leaves and working to the root.

The parser repeatedly matches the right-hand side $rhs$ of a production against a substring in the current right-sentential form.

At each match, it applies a reduction to build the parse tree.

- each reduction replaces the matched substring with the nonterminal on the left-hand side $lhs$ of the production
- each reduction adds an internal node to the current parse tree
- the result is another right-sentential form

The final result is a rightmost derivation, in reverse.
Example

Consider the grammar

1  \langle \text{goal} \rangle ::=: a \langle A \rangle \langle B \rangle e
2  \langle A \rangle ::=: \langle A \rangle b c
3  \text{ | } b
4  \langle B \rangle ::=: d

and the input string \text{abbcde}.

<table>
<thead>
<tr>
<th>Prod’n.</th>
<th>Sentential Form</th>
<th>Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>\text{abbcde}</td>
<td>3,2</td>
</tr>
<tr>
<td>3</td>
<td>a\langle A\rangle bcde</td>
<td>2,4</td>
</tr>
<tr>
<td>2</td>
<td>a\langle A\rangle de</td>
<td>4,3</td>
</tr>
<tr>
<td>4</td>
<td>a\langle A\rangle \langle B\rangle e</td>
<td>1,4</td>
</tr>
<tr>
<td>1</td>
<td>\langle \text{goal} \rangle</td>
<td>—</td>
</tr>
</tbody>
</table>

The trick appears to be scanning the input and finding valid sentential forms.
Handles

We trying to find a substring $\alpha$ of the current right-sentential form where:

- $\alpha$ matches some production $A ::= \alpha$
- reducing $\alpha$ to $A$ is one step in the reverse of a rightmost derivation.

We will call such a string a handle.

Formally,

- a handle of a right-sentential form $\gamma$ is a production $A ::= \beta$ and a position in $\gamma$ where $\beta$ may be found.
- If $(A ::= \beta, k)$ is a handle, then replacing the $\beta$ in $\gamma$ at position $k$ with $A$ produces the previous right-sentential form in a rightmost derivation of $\gamma$.

Because $\gamma$ is a right-sentential form, the substring to the right of a handle contains only terminal symbols.
Handles

Provable fact:

If $G$ is unambiguous, then every right-sentential form has a unique handle.

Proof: (by definition)

1. $G$ is unambiguous $\Rightarrow$ rightmost derivation is unique.

2. $\Rightarrow$ a unique production $A ::= \beta$ applied to take $\gamma_{i-1}$ to $\gamma_i$

3. $\Rightarrow$ a unique position $k$ at which $A ::= \beta$ is applied

4. $\Rightarrow$ a handle $(A ::= \beta, k)$
Example

The left-recursive expression grammar

*(original form, before left factoring)*

\[
\begin{align*}
1 & \quad \langle \text{goal} \rangle ::= \langle \text{expr} \rangle \\
2 & \quad \langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{term} \rangle \\
3 & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{\mid} \quad \langle \text{expr} \rangle - \langle \text{term} \rangle \\
4 & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{\mid} \quad \langle \text{term} \rangle \\
5 & \quad \langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle \\
6 & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{\mid} \quad \langle \text{term} \rangle / \langle \text{factor} \rangle \\
7 & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{\mid} \quad \langle \text{factor} \rangle \\
8 & \quad \langle \text{factor} \rangle ::= \text{num} \\
9 & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{\mid} \quad \text{id}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Prod’n.</th>
<th>Sentential Form</th>
<th>Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>(\langle \text{goal} \rangle)</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>(\langle \text{expr} \rangle)</td>
<td>1,1</td>
</tr>
<tr>
<td>3</td>
<td>(\langle \text{expr} \rangle - \langle \text{term} \rangle)</td>
<td>3,3</td>
</tr>
<tr>
<td>5</td>
<td>(\langle \text{expr} \rangle - \langle \text{term} \rangle * \langle \text{factor} \rangle)</td>
<td>5,5</td>
</tr>
<tr>
<td>9</td>
<td>(\langle \text{expr} \rangle - \langle \text{term} \rangle * \text{id})</td>
<td>9,5</td>
</tr>
<tr>
<td>7</td>
<td>(\langle \text{expr} \rangle - \langle \text{factor} \rangle * \text{id})</td>
<td>7,3</td>
</tr>
<tr>
<td>8</td>
<td>(\langle \text{expr} \rangle - \text{num} * \text{id})</td>
<td>8,3</td>
</tr>
<tr>
<td>4</td>
<td>(\langle \text{term} \rangle - \text{num} * \text{id})</td>
<td>4,1</td>
</tr>
<tr>
<td>7</td>
<td>(\langle \text{factor} \rangle - \text{num} * \text{id})</td>
<td>7,1</td>
</tr>
<tr>
<td>9</td>
<td>\text{id} - \text{num} * \text{id})</td>
<td>9,1</td>
</tr>
</tbody>
</table>
Handle-pruning

The process we use to construct a bottom-up parse is called handle-pruning.

To construct a rightmost derivation

\[ S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w, \]

we set \( i \) to \( n \) and apply the following simple algorithm

\[
\text{do } i = n \text{ to } 1 \text{ by } -1 \\
\quad (1) \text{ find the handle } (A_i ::= \beta_i, k_i) \text{ in } \gamma_i \\
\quad (2) \text{ replace } \beta_i \text{ with } A_i \text{ to generate } \gamma_{i-1}
\]

This takes \( 2n \) steps, where \( n \) is the length of the derivation
Shift-reduce parsing

One scheme to implement a handle-pruning, bottom-up parser is called a \textit{shift-reduce} parser.

Shift-reduce parsers use a \textit{stack} and an input buffer

1. initialize stack with $\$

2. Repeat until the top of the stack is the goal symbol and the input token is “end of file”
   
   a) \textit{find the handle}
      
      if we don’t have a handle on top of the stack, shift an input symbol onto the stack
   
   b) \textit{prune the handle}
      
      if we have a handle ($A ::= \beta, k$) on the stack, reduce
      
      i) pop $\beta$ symbols off the stack
      
      ii) push $A$ onto the stack

Back to “x - 2 * y”

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id - num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$id</td>
<td>- num * id</td>
<td>9,1</td>
<td>reduce 9</td>
</tr>
<tr>
<td>$(factor)</td>
<td>- num * id</td>
<td>7,1</td>
<td>reduce 7</td>
</tr>
<tr>
<td>$(term)</td>
<td>- num * id</td>
<td>4,1</td>
<td>reduce 4</td>
</tr>
<tr>
<td>$(expr)</td>
<td>- num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$(expr) -</td>
<td>num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$(expr) - num</td>
<td>* id</td>
<td>8,3</td>
<td>reduce 8</td>
</tr>
<tr>
<td>$(expr) - (factor)</td>
<td>* id</td>
<td>7,3</td>
<td>reduce 7</td>
</tr>
<tr>
<td>$(expr) - (term)</td>
<td>* id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$(expr) - (term) *</td>
<td>id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$(expr) - (term) * id</td>
<td>9,5</td>
<td>reduce 9</td>
<td></td>
</tr>
<tr>
<td>$(expr) - (term) * (factor)</td>
<td>5,5</td>
<td>reduce 5</td>
<td></td>
</tr>
<tr>
<td>$(expr) - (term)</td>
<td></td>
<td>3,3</td>
<td>reduce 3</td>
</tr>
<tr>
<td>$(expr)</td>
<td></td>
<td>1,1</td>
<td>reduce 1</td>
</tr>
<tr>
<td>$(goal)</td>
<td></td>
<td>none</td>
<td>accept</td>
</tr>
</tbody>
</table>

1. Shift until top of stack is the right end of a handle

2. Find the left end of the handle and reduce

5 shifts + 9 reduces + 1 accept
Shift-reduce parsing

*Shift-reduce parsers*

- are simple to understand
- have a simple, table-driven, *shift-reduce* skeleton
- encode grammatical knowledge in tables

A shift-reduce parser has just four canonical actions:

1. *shift* — next input symbol is shifted onto the top of the stack
2. *reduce* — right end of handle is on top of stack; locate left end of handle within the stack; pop handle off stack and push appropriate non-terminal *lhs*
3. *accept* — terminate parsing and signal success
4. *error* — call an error recovery routine
LR(1) grammars

Informally, we say that a grammar $G$ is LR(1) if, given a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_n = w,$$

we can, for each right-sentential form in the derivation,

1. *isolate the handle of each right-sentential form*, and

2. *determine the production by which to reduce*

by scanning $\gamma_i$ from left to right, going at most 1 symbol beyond the right end of the handle of $\gamma_i$.

Formality will come later.
Why study LR(1) grammars?

LR(1) grammars are often used to construct shift-reduce parsers.

We call these parsers LR(1) parsers.

- everyone’s favorite parser (EFP)
- virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars that can be parsed by a non-backtracking, shift-reduce parser
- efficient shift-reduce parsers can be implemented for LR(1) grammars
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- LR grammars describe a proper superset of the languages recognized by LL (predictive) parsers
Table-driven $LR(1)$ parsing

A table-driven $LR(1)$ parser looks like

```
source code → scanner → table-driven parser → il
```

Stack two items per state: state and symbol

Table building tools are readily available (yacc)

We’ll learn how to build these tables by hand!
LR(1) parsing

The skeleton parser:

```python
token = next_token()
repeat forever
    s = top of stack
    if action[s,token] = "shift $s_i$" then
        push token
        push $s_i$
        token = next_token()
    else if action[s,token] = "reduce $A ::= \beta" then
        pop 2 * |$\beta$| symbols
        s = top of stack
        push $A$
        push goto[s,$A$]
    else if action[s, token] = "accept" then
        return
    else error()

This takes $k$ shifts, $l$ reduces, and 1 accept, where $k$ is the length of the input string and $l$ is the length of the reverse rightmost derivation.

Note: Equivalent to Figure 4.30, Aho, Sethi, and Ullman
## Example tables

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>id + * $</td>
<td>&lt;expr&gt; &lt;term&gt; &lt;factor&gt;</td>
</tr>
</tbody>
</table>

| $S_0$ | s4 — — — | 1 2 3 |
| $S_1$ | — — — acc | — — — |
| $S_2$ | — s5 — r3 | — — — |
| $S_3$ | — r5 s6 r5 | — — — |
| $S_4$ | — r6 r6 r6 | — — — |
| $S_5$ | s4 — — — | 7 2 3 |
| $S_6$ | s4 — — — | — 8 3 |
| $S_7$ | — — — r2 | — — — |
| $S_8$ | — r4 — r4 | — — — |

## The Grammar

1. $<\text{goal}> ::= <\text{expr}>$
2. $<\text{expr}> ::= <\text{term}> + <\text{expr}>$
3. $<\text{term}> ::= <\text{factor}> * <\text{term}>$
4. $<\text{factor}> ::= \text{id}$