Bottom-up Parsing

Roadmap (Where are we?)

- Predictive top-down parsing
  - FIRST, FOLLOW, FIRST
  - The LL(1) condition
  - Table-driven LL(1) parsers
- Bottom-up parsing
  - Finding reductions
  - Shift-reduce parsers

Parsing Techniques: Bottom-up parsers

LR(1) operator precedence
1 input symbol lookahead
construct rightmost derivation (backwards)
input: read left-to-right

LR(2) operator precedence
1 input symbol lookahead
construct rightmost derivation (backwards)
input: read left-to-right

A bottom-up parser builds a derivation by working from the input sentence back toward the start symbol $S$.

To reduce $\gamma_i$ to $\gamma_{i+1}$ match some rhs $\beta$ against $\gamma_i$ then replace $\beta$ with its corresponding lhs, $A$. (assuming the production $A \rightarrow \beta$)

In terms of the parse tree, this is working from leaves to root
- Nodes with no parent in a partial tree form its upper fringe
- Since each replacement of $\beta$ with $A$ shrinks the upper fringe, we call it a reduction.

The parse tree need not be built, it can be simulated

$|\text{parse tree nodes}| = |\text{words}| + |\text{reductions}|$
Finding Reductions

Consider the simple grammar

<table>
<thead>
<tr>
<th></th>
<th>Goal</th>
<th>Next Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
<td>$\Rightarrow A \alpha \beta$</td>
</tr>
<tr>
<td>2</td>
<td>$A$</td>
<td>$\Rightarrow A \alpha \beta$</td>
</tr>
<tr>
<td>3</td>
<td>$\alpha$</td>
<td>$\Rightarrow \alpha$</td>
</tr>
<tr>
<td>4</td>
<td>$\beta$</td>
<td>$\Rightarrow \beta$</td>
</tr>
</tbody>
</table>

And the input string $\text{abode}$

The trick is scanning the input and finding the next reduction
The mechanism for doing this must be efficient

---

Finding Reductions (Handles)

The parser must find a substring $\beta$ of the tree's frontier that
matches some production $A \rightarrow \beta$ that occurs as one step
in the rightmost derivation.
Informally, we call this substring $\beta$ a handle.

Formally:
A handle of a right-sentential form $\gamma$ is a pair $(A \rightarrow \beta, k)$ where
$A \rightarrow \beta = P$ and $k$ is the position in $\gamma$ of $\beta$'s rightmost symbol.

If $(A \rightarrow \beta, k)$ is a handle, then replacing $\beta$ at $k$ with $A$ produces the right
sentential form from which $\gamma$ is derived in the rightmost derivation.

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

$\Rightarrow$ the parser doesn't need to scan past the handle (only look-ahead)

---

Example (a very busy slide)

<table>
<thead>
<tr>
<th></th>
<th>Goal</th>
<th>Exp'</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
<td>$\Rightarrow$</td>
<td>Exp'</td>
</tr>
<tr>
<td>2</td>
<td>$\Rightarrow$</td>
<td>$\Rightarrow$</td>
<td>Exp'</td>
</tr>
<tr>
<td>3</td>
<td>$\Rightarrow$</td>
<td>$\Rightarrow$</td>
<td>Exp'</td>
</tr>
<tr>
<td>4</td>
<td>$\Rightarrow$</td>
<td>$\Rightarrow$</td>
<td>Exp'</td>
</tr>
<tr>
<td>5</td>
<td>$\Rightarrow$</td>
<td>$\Rightarrow$</td>
<td>Exp'</td>
</tr>
<tr>
<td>6</td>
<td>$\Rightarrow$</td>
<td>$\Rightarrow$</td>
<td>Exp'</td>
</tr>
<tr>
<td>7</td>
<td>$\Rightarrow$</td>
<td>$\Rightarrow$</td>
<td>Exp'</td>
</tr>
<tr>
<td>8</td>
<td>$\Rightarrow$</td>
<td>$\Rightarrow$</td>
<td>Exp'</td>
</tr>
<tr>
<td>9</td>
<td>$\Rightarrow$</td>
<td>$\Rightarrow$</td>
<td>Exp'</td>
</tr>
<tr>
<td>10</td>
<td>$\Rightarrow$</td>
<td>$\Rightarrow$</td>
<td>Exp'</td>
</tr>
</tbody>
</table>

The expression $\text{abode}$ Handles for rightmost derivation of a $\text{abode}$

This is the inverse of Figure 3.9 in EAC

---

Handle-pruning, Bottom-up Parsers

The process of discovering a handle & reducing it to the
appropriate left-hand side is called handle pruning.

Handle pruning forms the basis for a bottom-up parsing method

To construct a rightmost derivation

$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_n \Rightarrow \gamma$.

Apply the following simple algorithm

for $i$ from $n$ to $1$ by $-1$

Find the handle $(A \rightarrow \beta, k)$ in $\gamma_i$.
Replace $\beta$ with $A$ to generate $\gamma_i$.$\gamma_i$.

This takes $O(n)$ steps.

---

Handle-pruning, Bottom-up Parsers

One implementation technique is the shift-reduce parser

push INVALID
repeat until (top of stack = Goal and token = EOF)
if the top of the stack is a handle $A \rightarrow \beta$ then
// reduce $\beta$ to $A$
pop $\beta$, symbols off the stack
push $A$ onto the stack
else if (token = EOF) then
// shift
push token
taken -- next, taken ++
else // need to shift but out of input
report an error

Figure 3.11 in EAC
### Back to $x \cdot 2^y$

**Stack**  
| $\$ | $\text{Id} \cdot \text{Num} \cdot \text{Id} | none | shift |

1. Shift until the top of the stack is the right end of a handle  
2. Find the left end of the handle & reduce

| $\$ | $\text{Id} \cdot \text{Factor} | 9.1 | \text{red} \ 9 |
| $\text{Term} \cdot \text{Num} | 4.1 | \text{red} \ 4 |
| $\text{Exp}_r \cdot \text{Term} | none | shift |
| $\text{Exp}_r \cdot \text{Num} | 8.3 | \text{red} \ 8 |
| $\text{Exp}_r \cdot \text{Factor} | 7.3 | \text{red} \ 7 |
| $\text{Exp}_r \cdot \text{Term} | none | shift |
| $\text{Exp}_r \cdot \text{Id} | none | shift |

### Back to $x \cdot 2^y$

**Stack**  
| $\$ | $\text{Id} \cdot \text{Num} \cdot \text{Id} | none | shift |

1. Shift until the top of the stack is the right end of a handle  
2. Find the left end of the handle & reduce

| $\$ | $\text{Id} | 9.1 | \text{red} \ 9 |
| $\text{Factor} | 7.1 | \text{red} \ 7 |
| $\text{Term} | 4.1 | \text{red} \ 4 |
| $\text{Exp}_r | none | shift |
| $\text{Exp}_r \cdot \text{Num} | 8.3 | \text{red} \ 8 |
| $\text{Exp}_r \cdot \text{Factor} | 7.3 | \text{red} \ 7 |
| $\text{Exp}_r \cdot \text{Term} | none | shift |
| $\text{Exp}_r \cdot \text{Id} | 9.5 | \text{red} \ 9 |

### Back to $x \cdot 2^y$

**Stack**  
| $\$ | $\text{Id} \cdot \text{Num} \cdot \text{Id} | none | shift |

1. Shift until the top of the stack is the right end of a handle  
2. Find the left end of the handle & reduce

| $\$ | $\text{Id} | 9.1 | \text{red} \ 9 |
| $\text{Factor} | 7.1 | \text{red} \ 7 |
| $\text{Term} | 4.1 | \text{red} \ 4 |
| $\text{Exp}_r | none | shift |
| $\text{Exp}_r \cdot \text{Num} | 8.3 | \text{red} \ 8 |
| $\text{Exp}_r \cdot \text{Factor} | 7.3 | \text{red} \ 7 |
| $\text{Exp}_r \cdot \text{Term} | none | shift |
| $\text{Exp}_r \cdot \text{Id} | 9.5 | \text{red} \ 9 |

### Back to $x \cdot 2^y$

**Stack**  
| $\$ | $\text{Id} \cdot \text{Num} \cdot \text{Id} | none | shift |

1. Shift until the top of the stack is the right end of a handle  
2. Find the left end of the handle & reduce

| $\$ | $\text{Id} | 9.1 | \text{red} \ 9 |
| $\text{Factor} | 7.1 | \text{red} \ 7 |
| $\text{Term} | 4.1 | \text{red} \ 4 |
| $\text{Exp}_r | none | shift |
| $\text{Exp}_r \cdot \text{Num} | 8.3 | \text{red} \ 8 |
| $\text{Exp}_r \cdot \text{Factor} | 7.3 | \text{red} \ 7 |
| $\text{Exp}_r \cdot \text{Term} | none | shift |
| $\text{Exp}_r \cdot \text{Id} | 9.5 | \text{red} \ 9 |

9 shifts + 9 reduces + 1 accept.
Back to $x \cdot 2 \cdot y$

### Stack

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$</td>
<td>id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$id$</td>
<td>num</td>
<td>9.1</td>
<td>red. 9</td>
</tr>
<tr>
<td>$_id$</td>
<td>num</td>
<td>7.1</td>
<td>red. 7</td>
</tr>
<tr>
<td>$_term$</td>
<td>num</td>
<td>4.1</td>
<td>red. 4</td>
</tr>
<tr>
<td>$_exp$</td>
<td>num</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$_exp_num$</td>
<td>num</td>
<td>8.3</td>
<td>red. 8</td>
</tr>
<tr>
<td>$_exp_id$</td>
<td>num</td>
<td>7.3</td>
<td></td>
</tr>
<tr>
<td>$_exp_term$</td>
<td>id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$_exp_term_id$</td>
<td>id</td>
<td>9.5</td>
<td>red. 9</td>
</tr>
<tr>
<td>$_exp_term_factor$</td>
<td>id</td>
<td>5.5</td>
<td>red. 5</td>
</tr>
<tr>
<td>$_exp_term_factor_id$</td>
<td>id</td>
<td>3.3</td>
<td>red. 3</td>
</tr>
<tr>
<td>$_term_factor$</td>
<td>num</td>
<td>1.1</td>
<td>red. 1</td>
</tr>
<tr>
<td>$_term_factor_id$</td>
<td>num</td>
<td>none</td>
<td>accept</td>
</tr>
<tr>
<td>$_term_factor_id_num$</td>
<td>num</td>
<td>reduce</td>
<td>here</td>
</tr>
</tbody>
</table>

1. Shift until the top of the stack is the right end of a handle.
2. Find the left end of the handle & reduce.

---

### Shift-reduce Parsing

**Shift-reduce parsers are easily built and easily understood**

A shift-reduce parser has just four actions:

- **Shift** — next word is shifted onto the stack.
- **Reduce** — right end of handle is at top of stack.
- **Pop** handle off stack & push appropriate i/o
- **Accept** — stop parsing & report success

**Accept & Error are simple.**
Shift is just a push and a call to the scanner.
Reduce takes |r/A| pops & 1 push.

If handle-finding requires state, put it in the stack ⇒ 2x work.

---

### LR(1) Parsers

- **LR(1) parsers** are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition.
- **LR(1) parsers** recognize languages that have an LR(1) grammar.

**Informal definition:**

A grammar is LR(1) if, given a rightmost derivation

\[ S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_n \Rightarrow \text{sentence} \]

We can:

1. Isolate the handle of each right-sentential form $\gamma_i$ 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 $\gamma_i$.

---

### An Important Lesson about Handles

To be a handle, a substring of a sentential form $\gamma$ must have two properties:

- It must match the right-hand side $\beta$ of some rule $A \rightarrow \beta$.
- There must be some rightmost derivation from the goal symbol that produces the sentential form $\gamma$ with $A \rightarrow \beta$ as the last production applied.

**Simply looking for right-hand sides that match strings is not good enough.**

**Critical Question:** How can we know when we have found a handle without generating lots of different derivations?

- Answer:
  - LR(1) parsers build a DFA that runs over the stack & finds them. One token look-ahead may be used to determine the next action (shift or reduce) in each state of the DFA.
  - LR(0) parsers: no look-ahead.

---

### LR(1) Parsers

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

```
source code               Scanner               Table-driven parser
```

**Grammar Tables**

- **Parser Generator** creates ACTION & GOTO tables.
- **Tables** can be built by hand.
- However, this is a perfect task to automate.
LR(1) Skeleton Parser

stack push(0x91a033); stack pop(0x92);
not_found = true;
token = scanner.next_token();
do while (not_found)
    if (not_stack_top())
        stack pop(0x91a033); // pop 2^k[i] symbols
        stack push(0x91a033); stack pop(0x91a033);
    else if (ACTION(i, token) = "reduce k") then |
        stack push(token); stack pop(0x91a033); |
        token = scanner.next_token(); |
    else if (ACTION(i, token) = "shift k") then |
        stack push(token); stack pop(0x91a033); |
        token = scanner.next_token(); |
    else if (ACTION(i, token) = "accept"
            & token == EOF) |
        then not_found = false; |
    else report a syntax error and recover;
report success;

The skeleton parser
* uses ACTION & GOTO tables
* does (word) shifts
* does (derivation) reductions
* does 1 accept
* detects errors by failure of 3 other cases