Parsing III
Bottom-up Parsing
Parsing Techniques

Top-down parsers (LL(1), recursive descent)
• Start at the root of the parse tree and grow toward leaves
• Pick a production & try to match the input
• Bad “pick” ⇒ may need to backtrack
• Some grammars are backtrack-free (predictive parsing)

Bottom-up parsers (LR(1), operator precedence)
• Start at the leaves and grow toward root
• As input is consumed, encode possibilities in an internal state
• Start in a state valid for legal first tokens
• Bottom-up parsers handle a large class of grammars
Bottom-up Parsing (definitions)

The point of parsing is to construct a derivation

A derivation consists of a series of rewrite steps

\[ S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence} \]

- Each \( \gamma_i \) is a sentential form
  - If \( \gamma \) contains only terminal symbols, \( \gamma \) is a sentence in \( L(G) \)
  - If \( \gamma \) contains \( \geq 1 \) non-terminals, \( \gamma \) is a sentential form

- To get \( \gamma_i \) from \( \gamma_{i-1} \), expand some NT \( A \in \gamma_{i-1} \) by using \( A \rightarrow \beta \)
  - Replace the occurrence of \( A \in \gamma_{i-1} \) with \( \beta \) to get \( \gamma_i \)
  - In a leftmost derivation, it would be the first NT \( A \in \gamma_{i-1} \)

A left-sentential form occurs in a leftmost derivation.
A right-sentential form occurs in a rightmost derivation.
Bottom-up Parsing

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

\[ S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence} \]

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

In terms of the parse tree, this is working from leaves to root

- Nodes with no parent in a partial tree form its frontier
- Since each replacement of $\beta$ with $A$ shrinks the current frontier, we call it a reduction.
Finding Reductions

Consider the simple grammar

1. Goal $\rightarrow$ a A B e
2. A $\rightarrow$ A b c
3. b
4. B $\rightarrow$ d

And the input string abbcde

<table>
<thead>
<tr>
<th>Sentential Form</th>
<th>Next Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prod' n</td>
<td>Pos' n</td>
</tr>
<tr>
<td>abbcde</td>
<td>3</td>
</tr>
<tr>
<td>a A bcde</td>
<td>2</td>
</tr>
<tr>
<td>a A de</td>
<td>4</td>
</tr>
<tr>
<td>a A B e</td>
<td>1</td>
</tr>
<tr>
<td>Goal</td>
<td>—</td>
</tr>
</tbody>
</table>

The trick is scanning the input and finding the next reduction. The mechanism for doing this should 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

We call this substring \( \beta \) a handle

Formally,
A handle is a pair \(<A \rightarrow \beta, k>\) where \( A \rightarrow \beta \in P \) and \( k \) is position in tree’s current frontier of \( \beta \)’s rightmost (last) symbol.
Replacing \( \beta \) at \( k \) with \( A \) in the bottom-up parse represents the next step in the reverse rightmost derivation.
Finding Reductions (Handles)

Critical Insight

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

If we can find those handles, we can build a derivation!

Sketch of Proof:

1. $G$ is unambiguous $\Rightarrow$ rightmost derivation is unique
2. $\Rightarrow$ a unique production $A \rightarrow \beta$ applied to derive $\gamma_i$ from $\gamma_{i-1}$
3. $\Rightarrow$ a unique position $k$ at which $A \rightarrow \beta$ is applied
4. $\Rightarrow$ a unique handle $\langle A \rightarrow \beta, k \rangle$

This all follows from the definitions
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

\[ S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence} \]

Apply the following simple algorithm

for \( i \leftarrow n \) to 1 by -1

Find the handle \( \langle A_i \rightarrow \beta_i, k_i \rangle \) in \( \gamma_i \)

Replace \( \beta_i \) with \( A_i \) to generate \( \gamma_{i-1} \)
Handle-pruning, Bottom-up Parsers

One implementation technique is the shift-reduce parser

```plaintext
push INVALID
word ← NextWord()
repeat until (top of stack = Goal and word = EOF)
    if a handle for A → β on top of the stack then
        // reduce β to A
        pop |β| symbols off the stack
        push A onto the stack
    else if (word ≠ EOF) then // shift
        push word
        word ← NextWord()
    else // either no handle or no input
        report an error
```

How do errors show up?
- failure to find a handle
- hitting EOF & needing to shift (final else clause)
Either generates an error

Figure 3.11 in EAC
Example

<table>
<thead>
<tr>
<th>Rule</th>
<th>Right Hand Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>$Expr$</td>
</tr>
<tr>
<td>$Expr$</td>
<td>$Expr + Term$</td>
</tr>
<tr>
<td></td>
<td>$Expr - Term$</td>
</tr>
<tr>
<td></td>
<td>$Term$</td>
</tr>
<tr>
<td>Term</td>
<td>$Term * Factor$</td>
</tr>
<tr>
<td></td>
<td>$Term / Factor$</td>
</tr>
<tr>
<td></td>
<td>$Factor$</td>
</tr>
<tr>
<td>Factor</td>
<td>$number$</td>
</tr>
<tr>
<td></td>
<td>$id$</td>
</tr>
<tr>
<td></td>
<td>$(Expr)$</td>
</tr>
</tbody>
</table>

The expression grammar

$<id, x> - <num, 2> * <id, y>$
Example $<\text{id,x}> - <\text{num,2}> * <\text{id,y}>$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$</td>
<td>$\text{id}$</td>
<td>shift</td>
</tr>
<tr>
<td>$$ $\text{id}$</td>
<td>$\text{id} - \text{num} * \text{id}$</td>
<td>red. 9</td>
</tr>
<tr>
<td>$$ $\text{Factor}$</td>
<td>$\text{id} - \text{num} * \text{id}$</td>
<td>red. 7</td>
</tr>
<tr>
<td>$$ $\text{Term}$</td>
<td>$\text{id} - \text{num} * \text{id}$</td>
<td>red. 4</td>
</tr>
<tr>
<td>$$ $\text{Expr}$</td>
<td>$\text{id} - \text{num} * \text{id}$</td>
<td>shift</td>
</tr>
<tr>
<td>$$ $\text{Expr =}$</td>
<td>$\text{id} - \text{num} * \text{id}$</td>
<td>shift</td>
</tr>
<tr>
<td>$$ $\text{Expr - num}$</td>
<td>$\text{id} - \text{num} * \text{id}$</td>
<td>red. 8</td>
</tr>
<tr>
<td>$$ $\text{Expr - Factor}$</td>
<td>$\text{id} - \text{num} * \text{id}$</td>
<td>red. 7</td>
</tr>
<tr>
<td>$$ $\text{Expr_ Term}$</td>
<td>$\text{id} - \text{num} * \text{id}$</td>
<td>shift</td>
</tr>
<tr>
<td>$$ $\text{Expr_ Term*}$</td>
<td>$\text{id} - \text{num} * \text{id}$</td>
<td>shift</td>
</tr>
<tr>
<td>$$ $\text{Expr_ Term* id}$</td>
<td>$\text{id} - \text{num} * \text{id}$</td>
<td>red. 9</td>
</tr>
<tr>
<td>$$ $\text{Expr_ Term* Factor}$</td>
<td>$\text{id} - \text{num} * \text{id}$</td>
<td>red. 5</td>
</tr>
<tr>
<td>$$ $\text{Expr_ Term}$</td>
<td>$\text{id} - \text{num} * \text{id}$</td>
<td>red. 3</td>
</tr>
<tr>
<td>$$ $\text{Expr}$</td>
<td>$\text{id} - \text{num} * \text{id}$</td>
<td>red. 1</td>
</tr>
<tr>
<td>$$ $\text{Goal}$</td>
<td>$\text{id} - \text{num} * \text{id}$</td>
<td>accept</td>
</tr>
</tbody>
</table>

Diagram:

```
+-----------------+     +-----------------+
| Goal            |     | Expr            |
|                 |     | Term            |
| *               |     | Fact.           |
| Fact.           |     | <id,y>          |

<id,x> <num,2>
```

Diagram of the parse tree for the given input.
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
  - Locate left end of handle within the stack
  - Pop handle off stack & push appropriate $lhs$
- **Accept** — stop parsing & report success
- **Error** — call an error reporting/recovery routine

**Accept & Error** are simple

*Shift* is just a push and a call to the scanner

*Reduce* takes $|rhs|$ pops & 1 push

---

Handle finding is key
- handle is on stack
- finite set of handles
  $$\Rightarrow$$ use a DFA!
An Important Lesson about Handles

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

$\rightarrow$ It must match the right hand side $\beta$ of some rule $A \rightarrow \beta$

$\rightarrow$ 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?

$\rightarrow$ **Answer:** we use look ahead in the grammar along with tables produced as the result of analyzing the grammar.

$\rightarrow$ LR(1) parsers build a DFA that runs over the stack & finds them