Bottom Up Parsing
Ch 4 p 195-215 + handouts!
Read this by Monday.

March 6, 2009
The Plan

• Bottom Up Parsing
• Handles
• Shift/Reduce
• Operator Precedence Parsing
• Building Operator Precedence Tables
• Wednesday: Build an OPT in class
Bottom Up Parsing

• Build parse tree from leaves and work up!
  – *Reduce* a string, w, to the start symbol, S

• Reduction: replace a substring that matches the RHS of a production with the LHS of that production
  – *Right most derivation* is run in reverse.

\[
\begin{align*}
S &\rightarrow aABe \\
A &\rightarrow Abc \mid b \\
B &\rightarrow d
\end{align*}
\]

abbcde
Algorithm

1) Look for a substring in \( w \) that matches the right side of any production.

2) Repeat step 1) with the new string \( w' \) until the start symbol \( S \) is produced (accept) or we run out of matching possibilities (reject)

\[
\begin{align*}
S & \rightarrow aABe \\
A & \rightarrow Abc \mid b \\
B & \rightarrow d
\end{align*}
\]

abbcde
– Problems?
Handle

- A **handle** is a substring of a string that matches some production's right side such that a reduction to a nonterminal on the left can be done in one step along the reverse of a rightmost derivation.

```
S -> aABe
A -> Abc | b
B -> d
```

abbcde
Practice

- page 196/198

### Right Most Derivation

- Remember, we are doing bottom up parsing, so we start right here

```latex
E \rightarrow E + E
E \rightarrow E \ast E
E \rightarrow ( E )
E \rightarrow id
```

- Ambiguous grammar so $1+ \Rightarrow_{rm}$

- Handle Pruning
How to choose a Handle

• Add a restriction

We'll see an *implementable* algorithm for this later.

• Defn[Aho]: "A handle of a right-sentential form $\gamma$ is a production $A \rightarrow \beta$ and a position of $\gamma$ where the string $\beta$ may be found and replaced by $A$ to produce the *previous right-sentential form in the rightmost derivation* of $\gamma$.
That is, if $S \Rightarrow^{*}_{rm} \alpha A w \Rightarrow_{rm} \alpha \beta w$, then $A \rightarrow \beta$ in the position following $\alpha$ is a *handle* of $\alpha \beta w$."
Shift Reduce Parsing

- p199 ex4.24

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id + id * id $</td>
<td>shift</td>
</tr>
<tr>
<td>$id</td>
<td>+ id * id $</td>
<td>Reduce</td>
</tr>
<tr>
<td>$E</td>
<td>+ id * id $</td>
<td>shift</td>
</tr>
</tbody>
</table>
CONFLICTS!

- Some CFGs have unresolvable conflicts
  - shift/reduce
  - reduce/reduce

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ ... if expr then stmt</td>
<td>else ... $</td>
<td>?????</td>
</tr>
</tbody>
</table>

stmt -> if expr then stmt
| if expr then stmt else stmt |
| other |
Operator Precedence Parsing

• Form of Shift/Reduce parsing

Two important properties for these shift-reduce parsers is that $\varepsilon$ does not appear on the right side of any production and no production has two adjacent nonterminals.

$E \rightarrow E + E$

$T \rightarrow + T T$

This allows us to find handles!
Precedence

We need to define three different precedence relations between pairs of terminals.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt;. b</td>
<td>a yields precedence to b</td>
</tr>
<tr>
<td>a =. b</td>
<td>a has the same precedence as b</td>
</tr>
<tr>
<td>a &gt;. b</td>
<td>a takes precedence over b</td>
</tr>
</tbody>
</table>

They look like >, <, and == but are very different.
Why?

- Identify each handle using the precedence rules and reduce the right-sentential string, based on the precedence relations, to a start (accept) state.
## Precedence Table

Define precedence relations between terminals.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td></td>
<td></td>
<td>&gt;.</td>
<td>&gt;.</td>
<td>&gt;.</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td>&lt;.</td>
<td>&gt;.</td>
<td>&lt;.</td>
<td>&gt;.</td>
</tr>
<tr>
<td>*</td>
<td></td>
<td>&lt;.</td>
<td>&gt;.</td>
<td>&gt;.</td>
<td>&gt;.</td>
</tr>
<tr>
<td>$</td>
<td></td>
<td>&lt;.</td>
<td>&lt;.</td>
<td>&lt;.</td>
<td>accept</td>
</tr>
</tbody>
</table>
How does this work? (high level) p205

$ id + id * id $

$ <. id >. + <. id >. * <. id >. $

• Scan from left (to right) until the first $>. is found
• Then, scan backwards (left) until a $<. is found
• The handle is everything to the left of the $>. and right of the $<.
  – Including surrounding nonterminals
In Code, p206 Algo 4.5

• How to find a handle
• Use a stack

• If the precedence relation <. or =. holds between the topmost terminal on the stack and the next input symbol, SHIFT
• If the relation >. holds, REDUCE
• No relation, SYNTAX ERROR

This is the solution to the bottom up assignment!
## Example

<table>
<thead>
<tr>
<th>Handle/Output</th>
<th>Stack</th>
<th>Input</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id + id * id $</td>
<td>Start State</td>
<td></td>
</tr>
<tr>
<td>$ id</td>
<td>+id*id$</td>
<td>$ &lt;. id</td>
<td></td>
</tr>
</tbody>
</table>

- `$` is the start state.
- The current input is `id + id * id $`.
- The stack contains `id`.
- The current input is consumed (`$ <. id`).

The process continues with the next input and stack state.
Unary Operators

• In your grammar: *, &, -

• Example: Unary prefix operator
  \( \sim \) (not operator. Is not also a binary operator)
  \( X <. \sim \) for any op X
  \( \sim >. \) X if \( \sim \) has higher precedence than X,
  and \( \sim <. \) X otherwise
Unary op is also a binary op

- * is dereference and multiplication
- - is negation and subtraction
- & is not also binary

Use lexer to return different token for
  - Dereference/Multiplication
  - Negation/Subtraction

How?
  - Lexer needs to remember the previous token!
  - Cannot look ahead
Define

- Operator-precedence grammar is an ε-free operator grammar in which the precedence relations <,.=,.> constructed as previous are disjoint. For any pair of terminals, a and b, never more than one of the relations a <.b, a=.b ,a >. b is true.
Create Table

• Let G be an ε-free operator grammar

• For each two terminals a and b we say:
  
  • a =. b if there exists a RHS: αaβbyγ where β is either ε or a single nonterminal.
  
  • a <. b if for some NT A, a RHS αaAβ exists, and A=>+ γbδ, where γ is either ε or a single NT
  
  • a >. b if for some NT A, a RHS αAbβ exists, and A=>+ γaδ, where δ is either ε or a single NT
LEADING, TRAILING

• LEADING: for each NT, those terminals that can be the first terminal in a string derived from that NT

• TRAILING: for each NT, those terminals that can be the last terminal in a string derived from that NT
Leading/Trailing

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>First terminal</th>
<th>Last terminal</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

E -> E + T | T
T -> T * F | F
F -> (E) | id
Compute Precedence

- For \( = \). look for RHS with two terminals separated by nothing or a NT

- \( < \). Look for RHS with a terminal immediately to the left of a NT (a, A in rule above) For each, a is \( < \). to any terminal LEADING(A)

- \( > \). Look for a RHS with a nonterminal immediately to the left of a terminal (A, b rule above). Every terminal TRAILING(A) \( > \). b
Compute Precedence

- Algo 5.2 on handout!
• EXTRA
Create Operator Precedence Table

- Page 207: heuristic for arithmetic expressions
- Precedence & Associativity

- If op X has higher precedence than op Y, make $X >. Y$ and $Y <. X$
- If op X and op Y have equal precedence, make $X >. Y$ and $Y >. X$ if they are left assoc.
  $X <. Y$ and $Y <. X$ if they are right assoc.
- $X <. \text{id}$, $\text{id} >. X$, $X <. (, (\text{<. X, } ) >.X, X >. )$, $X >. \$, $\$ <. X$, for all op X
- More on page 207
# Build the table!

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>right</td>
</tr>
<tr>
<td>* /</td>
<td>left</td>
</tr>
<tr>
<td>+ -</td>
<td>left</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>^</th>
<th>id</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>