LL Parsing:
A piece of cake after LR

Dr. Sean Peisert – ECS 142 – Spring 2009
LL Parsing

- Still specified using a CFG
- Still reads left-to-right (Lx)
- Now is leftmost derivation (xL) rather than rightmost (xR)
- Constructed from the top in the order that tokens appear in the token stream
### Parsing Algorithms

<table>
<thead>
<tr>
<th>LL</th>
<th>LR</th>
</tr>
</thead>
<tbody>
<tr>
<td>𝐸</td>
<td>𝐸 → 2</td>
</tr>
<tr>
<td>𝐸 + 𝐸</td>
<td>𝐸 → 3</td>
</tr>
<tr>
<td>2 + 𝐸</td>
<td>𝐸 → 𝐸 + 𝐸</td>
</tr>
<tr>
<td>2 + 𝐸 * 𝐸</td>
<td>𝐸 → 8</td>
</tr>
<tr>
<td>2 + 3 * 𝐸</td>
<td>𝐸 → 𝐸 * 𝐸</td>
</tr>
<tr>
<td>2 + 3 * 8</td>
<td>𝐸 → 𝐸</td>
</tr>
</tbody>
</table>
LR vs. LL

- LR (bison/CUP): harder to implement, but allows more grammars
- LL: easier to write, but less flexible in the grammars
Recursive Descent Parsing

• Consider the grammar:
  • $E \rightarrow T + E \mid T$
  • $T \rightarrow \text{int} \mid \text{int} \ast T \mid (E)$

• Token stream is int$5 \ast$ int$2$
• Start with top-level non-terminal E
• Try the rules E in order
Recursive Descent Parsing

- $E_0 \rightarrow T_1 + E_2$
- Then try a rule for $T_1 \rightarrow (E_3)$
  - But ( does not match the input token int
- Try $T_1 \rightarrow \text{int}$. Token matches.
  - But + after $T_1$ does not match the input token *
- Try $T_1 \rightarrow \text{int} \ast T_2$
  - This will match but + after $T_1$ will be unmatched
- Have exhausted the choices for $T_1$
- Backtrack to choice for $E_0$
Recursive Descent Parsing

- $E_0 \rightarrow T_1$
- Follow same steps as before for $T_1$
- And succeed with $T_1 \rightarrow \text{int} \ast T_2$ and $T_2 \rightarrow \text{int}$

Diagram:

```
        E_0
         /
        /  /
       /    /
      T_1   T_2
     /      / 7
   int_5 *  int_2
```
Recursive Descent Parsing

- Parsing: given a string of tokens $t_1, t_2, t_3, t_4, t_5$, find its parse tree
- Recursive descent parsing. Try all of the productions exhaustively:
  - At a given moment, the finge of the parse tree is: $t_1, t_2, ... t_k, A$
  - Try all the productions for $A$: if $A \rightarrow BC$ is a production, the new finge is $t_1, t_2, ... t_k, BC$
  - Backtrack where the finge doesn’t match the string.
  - Stop when there are no more non-terminals.
Construction of Decision Table

• for each rule $X \rightarrow \alpha$

• for each $a$ in $\text{First}(\alpha)$

• $\text{table}(X, a) = X \rightarrow \alpha$

• so if we see the very first symbol, match the rule

• if $\varepsilon$ is in $\text{First}(\alpha)$

• for each $b$ in $\text{Follows}(X)$

• $\text{table}(X, b) = X \rightarrow \alpha$

• That’s it!
Non-LL Grammars

• There are a lot of non-LL grammars. The patterns to look for are:
  • Left-Recursive rules (goes into an infinite loop)
    • which must be rewritten as right-recursive
  • Common prefix rules
    • which must be rewritten to move the common prefix to one rule
Left Recursion

• $A \rightarrow Aa$
• $A \rightarrow b$

• For LL, $\text{First}(A \rightarrow Aa) = b$
• For LL, $\text{First}(A \rightarrow b) = b$
• this gives a conflict
Rewrite the Grammar

- A → bA'
- A' → aA'
- A' → ε
Common Prefix

- $X \rightarrow abc$
- $X \rightarrow abd$

- This works for LR.
- For LL, the parser only looks at the First set of rules, not the entire rule. So the rules cause a conflict.
Rewritten Grammar

- $X \rightarrow abY$
- $Y \rightarrow c$
- $Y \rightarrow d$
Recursive Descent

- Simple and general, but unpopular (and slow) because of backtracking
- Often we can avoid backtracking (somewhat)
Predictive Parsing

• Like recursive-descent, but parser can “predict” which production to use”
• By looking at the next few tokens
• No backtracking
• Predictive parsers accept LL(k) grammars
LL(1) Languages

• In recursive-descent, for each non-terminal and input token, there may be a choice of production

• LL(1) means that for each non-terminal and token, there is only one production that could lead to success.

• Can be specified as a 2D table
  • 1 dim for current non-terminal to expand
  • 1 dim for next token
  • A table entry contains one production
Example Grammar #1

Grammar:

- $E \rightarrow T + E \mid T$
- $T \rightarrow \text{int} \mid \text{int} \ast T \mid (E)$

Impossible to predict because $T$ has two productions with int. For $E$ it is not clear how to predict.

Grammar must be left-factored
Left Factoring

- **Grammar**
  - $E \rightarrow T + E \mid T$
  - $T \rightarrow \text{int} \mid \text{int} \ast T \mid (E)$

- **Factor out common prefixes of productions:**
  - $E \rightarrow T X$
  - $X \rightarrow + E \mid \varepsilon$
  - $T \rightarrow (E) \mid \text{int} Y$
  - $Y \rightarrow \ast T \mid \varepsilon$
LL(1) Parse Table Example

- Left-Factored Grammar
  - $E \rightarrow T \ X \quad X \rightarrow + \ E \mid \varepsilon$
  - $T \rightarrow (E) \mid \text{int} \ Y \quad Y \rightarrow * \ T \mid \varepsilon$
- The LL(1) parsing table ($\$ \$ is a special end marker):

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>T X</td>
<td></td>
<td></td>
<td></td>
<td>(E)</td>
<td>$</td>
</tr>
<tr>
<td>X</td>
<td>T X</td>
<td></td>
<td>+ E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>* T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Wednesday, April 22, 2009
LL(1) Parse Table

• Consider the [E, int] entry:
  • “When current non-terminal is E and next input is int, use production $E \rightarrow TX$”
  • This production can generate an int in the first place

• Consider the [Y, +] entry:
  • “When current non-terminal is y and current token is +, get rid of Y”
  • We’ll see later why this is so
LL(1) Parse Table

• Blank entries indicate error situations
• E.g., consider the [E, *] entry
• “There is no way to derive a string starting with * from non-terminal E”
Using Parse Tables

• Similar to recursive descent, except:
  • For each non-terminal S
  • We look at the next token a
  • And choose the production shown at [S,a]
  • Use a stack to keep track of pending non-terminals
  • Reject when we encounter an error state
  • Accept when we encounter an end-to-input
LL(1) Parsing Algorithm

initialize stack = <S, $> and next (pointer to tokens)
repeat
  case stack of
    <X, rest> if T[X, *next] = Y \ldots Y_n
      then stack ← Y \ldots Y_n rest> ;
    else error();
    <t, rest> if T == *next ++
      then stack ← <rest> ;
    else error();
  until stack == < >
# LL(1) Parsing Example

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$</td>
<td>int * int $</td>
<td>TX</td>
</tr>
<tr>
<td>TX$</td>
<td>int * int $</td>
<td>int Y</td>
</tr>
<tr>
<td>int YX$</td>
<td>int * int *</td>
<td>terminal</td>
</tr>
<tr>
<td>YX$</td>
<td>*int$</td>
<td>*T</td>
</tr>
<tr>
<td>*TX$</td>
<td>*int$</td>
<td>terminal</td>
</tr>
<tr>
<td>TX$</td>
<td>int$</td>
<td>int Y</td>
</tr>
<tr>
<td>int YX$</td>
<td>int$</td>
<td>terminal</td>
</tr>
<tr>
<td>YX$</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>X$</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>ACCEPT</td>
</tr>
</tbody>
</table>
Example Grammar #2

- $D \rightarrow v \ L \ c \ T$
- $L \rightarrow L \ m \ i$
- $L \rightarrow i$
- $T \rightarrow r$

- $v = \text{Var}$, $L = \text{IdList}$
- $c = \text{colon}$, $T = \text{type}$
- $m = \text{comma}$, $i = \text{id}$, $r = \text{REAL}$
After fixing L
(which was right-recursive)

- $D \rightarrow v L c T$
- $L \rightarrow i X$
- $X \rightarrow m i X$
- $X \rightarrow \varepsilon$
- $T \rightarrow r$
Algorithm

- start with the start symbol on the stack
- repeat until done
  - if top of stack == first char of input
    - pop 1 of stack
    - advance input
  - else if top of stack == non-terminal T
    - consult (T, input)
    - if (T,input) == T → α
      - pop T
      - push input in reverse order
      - print T → α reduce
    - else if (T,input) == empty
      - error - reject
  - else if stack is empty and input is empty
    - accept
  - else if stack is empty and input is not empty
    - reject
## LL Decision Table

<table>
<thead>
<tr>
<th>D</th>
<th>v</th>
<th>i</th>
<th>m</th>
<th>c</th>
<th>r</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D-&gt;vLcT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>L-&gt;iX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>X-&gt;miX</td>
<td></td>
<td></td>
<td>T-&gt;r</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td>X-&gt;e</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Wednesday, April 22, 2009
<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>vimicr</td>
<td>$D, v=\text{match } D \rightarrow vLcT, \text{pop LHS}(D), \text{push RHS }(vLcT) \text{ in reverse order}$</td>
</tr>
</tbody>
</table>
| TcL  | imicr  | $L, i=\text{match } L \rightarrow iX$  
|      |        | pop $L$, push $iX$ in reverse order |
| TcXi | imicr  | $i, i=\text{match}, \text{pop stack, advance}$ |
| TcX  | micr   | $X, m=\text{match } X \rightarrow miX$  
|      |        | pop $X$, push $miX$ in reverse order |
| TcXim| micr   | $m, m=\text{match}$  
|      |        | pop stack, advance |
| TcXi | icr    | $i, i=\text{match}$  
|      |        | pop stack, advance |
| TcX  | cr     | $X, c=\text{match } X \rightarrow \varepsilon$  
|      |        | pop $X$, push $\Sigma$ in reverse order |
| Tc   | cr     | $c, c=\text{match}$  
|      |        | pop stack, advance |
| T    | r      | $T, r=\text{match } T \rightarrow r$  
|      |        | pop $T$, push $r$ in reverse order |
| r    | r      | $r, r=\text{match}$  
|      |        | pop stack, advance |
Done! Accept!
Status

- Project 2 Due Friday, Apr. 24, 11:55pm