Bottom-Up Parsing

CS 132: Compiler Design

January 31, 2011
Review: LL(1)

A grammar is said to be LL(1) if one can always predict the next grammar production by peeking at one character of input.

A grammar is said to be LL(k) if we can make the prediction based on the next k tokens of the input.

A language is said to be LL(k) if it has an LL(k) grammar (which generates the right strings, but not necessarily the right parse trees)
Massaging Grammars

\[
\begin{align*}
S & \rightarrow E \ \$ \\
E & \rightarrow n \\
& \quad \mid E - n \\
\downarrow \\
\text{Eliminate “left recursion”} \\
\downarrow \\
S & \rightarrow E \ \$ \\
E & \rightarrow n \\
& \quad \mid n - E
\end{align*}
\]

✓ In this case, a non-LL grammar becomes LL(2).
✓ Note: for recursive descent, LL(2) usually isn’t that bad!
✓ But, this transformation can change the parse tree.
Massaging Grammars

\[
\begin{align*}
S &\rightarrow E \ $ \\
E &\rightarrow n \\
&\quad | \ n - E \\
\downarrow \quad \downarrow \\
&\quad \text{“Left-factor”} \\
\downarrow \\
S &\rightarrow E \ $ \\
E &\rightarrow n \ E' \\
E' &\rightarrow \epsilon \\
&\quad | \ - E'
\end{align*}
\]

✓ In this case, an LL(2) grammar becomes LL(1).
✓ Possibly useful if predictions go via a big lookup table.
✓ Parse tree is even less pretty.
Alternative: EBNF

S -> E $
E -> E
| E - n
↓

S -> E $
E -> n { - n }
or

S -> E $
E -> n ( - n )* 

✓ Comparatively easy to handle in recursive descent
   (Take one n, and then loop to gather as many - n’s as you can.)
✓ Result is a “list” of subtractions; can turn these into a proper tree.
A (Contrived) Grammar

Given the grammar:

\[ S \rightarrow a \ C \ E \ | \ b \ D \]
\[ C \rightarrow b \ c \ | \ C \ d \]
\[ D \rightarrow d \ | \ c \ D \ e \]
\[ E \rightarrow e \]

Draw the parse trees for the inputs \( a \ b \ c \ d \ e \) and \( b \ c \ d \ e \).
LR PARSING

S -> a C E | b D
C -> b c | C d
D -> d | c D e
E -> e

The essence of LR parsing is to build parse trees in a bottom-up fashion:
✓ Start with the leaves (a b c d e and b c d e respectively)
✓ Repeatedly combine subtrees into bigger trees using the productions.

Sadly, we can’t naively “just run productions backwards”
**Shift-Reduce Parsing**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action (Shift/Reduce)</th>
</tr>
</thead>
</table>

Top-Down Parsing

Bottup-Up Parsing

Practical Issues
**Some Terminology**

Assume we have a parse tree.

A *handle* of a parse tree is the leftmost, complete and nontrivial subtree.
Some Terminology

Assume we have a parse tree.

A handle of a parse tree is the leftmost, complete and nontrivial subtree.

We prune the handle of a parse tree by removing the children/leaves.
FROM TREES TO PRODUCTION SEQUENCES

S -> a C E | b D
C -> b c | C d
D -> d | c D e
E -> e

Start with a parse tree (say, for a b c d e).
FROM TREES TO PRODUCTION SEQUENCES

\[ S \rightarrow a \ C \ E \mid b \ D \]
\[ C \rightarrow b \ c \mid C \ d \]
\[ D \rightarrow d \mid c \ D \ e \]
\[ E \rightarrow e \]

Start with a parse tree (say, for \( a \ b \ c \ d \ e \)).

Repeatedly prune the handle.
**From Trees to Production Sequences**

S -> a C E | b D  
C -> b c | C d  
D -> d | c D e  
E -> e  

Start with a parse tree (say, for a b c d e).

Repeatedly prune the handle.

How can we get a rightmost production sequence out of this?
**From Trees to Production Sequences**

\[ S \rightarrow a \ C \ E \mid b \ D \]
\[ C \rightarrow b \ c \mid C \ d \]
\[ D \rightarrow d \mid c \ D \ e \]
\[ E \rightarrow e \]

Start with a parse tree (say, for \( a \ b \ c \ d \ e \)).

Repeatedly prune the handle.

How can we get a rightmost production sequence out of this?

If only we could find handles in the original string, without precomputing the parse tree…
**Viable Prefixes**

Assume that $\tilde{\alpha}$ and $\tilde{\beta}$ are sequences of terminals and non-terminals.
Viable Prefixes

Assume that $\vec{\alpha}$ and $\vec{\beta}$ are sequences of terminals and non-terminals.

We say that $\vec{\alpha}\vec{\beta}$ is a viable prefix if there is a sequence $\ldots$ of terminals such that

$$\vec{\alpha}\vec{\beta}\ldots$$

has a parse tree with $\vec{\beta}$ as the handle.
Viable Prefixes

Assume that $\vec{\alpha}$ and $\vec{\beta}$ are sequences of terminals and non-terminals.

We say that $\vec{\alpha} \vec{\beta}$ is a viable prefix if there is a sequence ... of terminals such that

$$\vec{\alpha} \vec{\beta} \ldots$$

has a parse tree with $\vec{\beta}$ as the handle.

✓ What are some viable prefixes for trees whose root is $E$? $C$? $D$? $S$?

- $S \rightarrow a\ C\ E\ |\ b\ D$
- $C \rightarrow b\ c\ |\ C\ d$
- $D \rightarrow d\ |\ c\ D\ e$
- $E \rightarrow e$
Why Should You Care?
Why Should You Care?

✓ The viable prefixes of every context-free grammar are regular!

✓ That is, a finite state machine can recognize viable prefixes. It can even tell you what the handle is.
Why Should You Care?

✓ The viable prefixes of every context-free grammar are regular!

✓ That is, a finite state machine can recognize viable prefixes. It can even tell you what the handle is.

✓ Every parseable string (by definition) has a viable prefix.

✓ Some grammars have a property that no viable prefix extends another. These are called LR(0) grammars.

✓ If so, every derivable string starts with a unique viable prefix. This gives us a parsing algorithm for LR(0) grammars.
**Parsing Automaton: NFA**

\[
S \rightarrow aCE \mid bD \\
C \rightarrow bc \mid Cd \\
D \rightarrow d \mid cDe \\
E \rightarrow e
\]
BUILDING A PARSING AUTOMATON
### Using the Parsing Automaton

Parse \text{b c d e} and \text{a b c d e} using the parsing automaton.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action (Shift/Reduce)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LR(0)

A grammar is LR(0) if the parsing automaton (as described) gives unambiguous instructions.

BTW, why doesn’t anyone ever consider LL(0) grammars?
A Non-LR(0) Example

Sadly, few useful grammars are LR(0). For example,

\[
S \rightarrow E$
\]
\[
E \rightarrow n
\]
\[
E \rightarrow n+E
\]
\[
S \rightarrow E$
\]
\[
S \rightarrow E$
\]
\[
E \rightarrow n+E
\]
\[
E \rightarrow n+.E
\]
\[
E \rightarrow .n
\]
\[
E \rightarrow .n+E
\]
\[
E \rightarrow n.
\]
\[
E \rightarrow n.+En
\]

✓ Show a parse tree for an input starting with \( n \), where \( n \) is the handle.

✓ Show a parse tree for an input starting with \( n \), where \( n \) isn’t the handle.

Thus, in practice we need to peek past the handle.
SLR(1)

Reduce using a rule $B \rightarrow \vec{\beta}$ only if the next input token is in FOLLOW($B$).

SLR(1):
FOLLOW($E$) = {$}
not LR(0)
LR(k)

Add lookahead tokens to the LR(0) items in the NFA.

Also, LALR(k): Merge similar states ("lossy compression")
Dangling Else

\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S \\
| \text{if } E \text{ then } S \\
| ... \\
\]

How to parse

if a then if b then s1 else s2
Dangling Else

S -> if E then S else S
| if E then S
| ...

How to parse

if a then if b then s1 else s2

No ambiguous grammar is LR(k), so you’d get a shift-reduce conflict

S -> if E then S . else S <any>
S -> if E then S . else
Solutions

Hack the grammar

\[ S \rightarrow M \]
\[ S \rightarrow U \]

\[ M \rightarrow \text{if } E \text{ then } M \text{ else } M \]
\[ M \rightarrow \ldots \]

\[ U \rightarrow \text{if } E \text{ then } S \]
\[ U \rightarrow \text{if } E \text{ then } M \text{ else } U \]
Solutions

Hack the grammar

S -> M
S -> U

M -> if E then M else M
M -> ...

U -> if E then S
U -> if E then M else U

Hack the language

S -> if E then S else S fi
  | if E then S fi
  | ...
  | ...
Solutions

Hack the grammar

\[ S \rightarrow M \]
\[ S \rightarrow U \]

\[ M \rightarrow \text{if } E \text{ then } M \text{ else } M \]
\[ M \rightarrow \ldots \]

\[ U \rightarrow \text{if } E \text{ then } S \]
\[ U \rightarrow \text{if } E \text{ then } M \text{ else } U \]

Hack the language

\[ S \rightarrow \text{if } E \text{ then } S \text{ else } S \text{ fi} \]
\[ | \text{if } E \text{ then } S \text{ fi} \]
\[ | \ldots \]

Hack the parser:
Whenever you reach this “conflicting” state, just go ahead and shift.
**Ambiguous Expression Grammars**

\[ E \rightarrow E + E \mid E - E \]
\[ \mid E \ast E \mid E / E \]
\[ \mid ( E ) \mid n \]
Ambiguous Expression Grammars

\[ E \rightarrow E + E \mid E - E \]
\[ \mid E * E \mid E / E \]
\[ \mid ( E ) \mid n \]

All sorts of shift-reduce conflicts:

<table>
<thead>
<tr>
<th>Production</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ E \rightarrow E . + E ]</td>
<td>&lt;any&gt;</td>
</tr>
<tr>
<td>[ E \rightarrow E * E . + ]</td>
<td></td>
</tr>
<tr>
<td>[ E \rightarrow E . * E ]</td>
<td>&lt;any&gt;</td>
</tr>
<tr>
<td>[ E \rightarrow E + E . * ]</td>
<td></td>
</tr>
<tr>
<td>[ E \rightarrow E . + E ]</td>
<td>&lt;any&gt;</td>
</tr>
<tr>
<td>[ E \rightarrow E + E . + ]</td>
<td></td>
</tr>
</tbody>
</table>
Solutions

Hack the grammar

\[
\begin{align*}
E & \rightarrow E \ + \ T \ | \ E \ - \ T \ | \ T \\
T & \rightarrow T \ * \ F \ | \ T \ / \ F \ | \ F \\
F & \rightarrow (E) \ | \ n
\end{align*}
\]

\[
\begin{align*}
E & \rightarrow T \ ( + \ T \ | \ - \ T )^* \\
T & \rightarrow F \ ( * \ F \ | \ / \ F )^* \\
F & \rightarrow (E) \ | \ n
\end{align*}
\]
Solutions

Hack the grammar

\[
\begin{align*}
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T * F \mid T / F \mid F \\
F &\rightarrow (E) \mid n
\end{align*}
\]

\[
\begin{align*}
E &\rightarrow T (+ T \mid - T)^* \\
T &\rightarrow F (* F \mid / F)^* \\
F &\rightarrow (E) \mid n
\end{align*}
\]

Hack the language

\[
E \rightarrow + E E \mid * E E \mid \ldots
\]
Solutions

Hack the grammar

\[
E \to E + T \mid E - T \mid T \quad E \to T ( + T \mid - T )* \\
T \to T * F \mid T / F \mid F \\
F \to ( E ) \mid n \\
\]

Hack the language

\[
E \to + E E \mid * E E \mid \ldots \\
\]

Hack the parser to shift or reduce as desired.
**Solutions**

Hack the grammar

\[
\begin{align*}
E & \rightarrow E + T \mid E - T \mid T \\
T & \rightarrow T * F \mid T / F \mid F \\
F & \rightarrow ( E ) \mid n
\end{align*}
\]

\[
\begin{align*}
E & \rightarrow T (+ T \mid - T)^* \\
T & \rightarrow F (* F \mid / F)^* \\
F & \rightarrow ( E ) \mid n
\end{align*}
\]

Hack the language

\[
E \rightarrow + E E \mid * E E \mid \ldots
\]

Hack the parser to shift or reduce as desired. YACC support:

```yacc
%left PLUS MINUS
%left STAR SLASH
```
Solutions

Hack the grammar

\[
\begin{align*}
E & \rightarrow E + T \mid E - T \mid T \\
T & \rightarrow T * F \mid T / F \mid F \\
F & \rightarrow ( E ) \mid n
\end{align*}
\]

\[
\begin{align*}
E & \rightarrow T (+ T \mid - T)* \\
T & \rightarrow F (* F \mid / F)* \\
F & \rightarrow ( E ) \mid n
\end{align*}
\]

Hack the language

\[
E \rightarrow + E E \mid * E E \mid \ldots
\]

Hack the parser to shift or reduce as desired. YACC support:

%left PLUS MINUS
%left STAR SLASH

✓ Terminals listed in increasing precedence
Solutions

Hack the grammar

\[
\begin{align*}
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T * F \mid T / F \mid F \\
F &\rightarrow (E) \mid n
\end{align*}
\]

\[
\begin{align*}
E &\rightarrow T (+ T \mid - T)^* \\
T &\rightarrow F (* F \mid / F)^* \\
F &\rightarrow (E) \mid n
\end{align*}
\]

Hack the language

\[
E \rightarrow + E E \mid * E E \mid \ldots
\]

Hack the parser to shift or reduce as desired. YACC support:

%left PLUS MINUS
%left STAR SLASH

✓ Terminals listed in increasing precedence
✓ Precedence of a rule is that of its last terminal
Solutions

Hack the grammar

\[
\begin{align*}
E & \rightarrow E + T \mid E - T \mid T \\
T & \rightarrow T * F \mid T / F \mid F \\
F & \rightarrow (E) \mid n
\end{align*}
\]

\[
\begin{align*}
E & \rightarrow T (+ T \mid - T)* \\
T & \rightarrow F (* F \mid / F)* \\
F & \rightarrow (E) \mid n
\end{align*}
\]

Hack the language

\[
\begin{align*}
E & \rightarrow + E E \mid * E E \mid \ldots
\end{align*}
\]

Hack the parser to shift or reduce as desired. YACC support:

\[
\begin{align*}
%left & \ \text{PLUS \ MINUS} \\
%left & \ \text{STAR \ SLASH}
\end{align*}
\]

✓ Terminals listed in increasing precedence

✓ Precedence of a rule is that of its last terminal

✓ Reduce if rule precedence greater than next token, or if same and rule precedence is left-associative. Otherwise shift.
**Unary minus**

We want these rules to be in increasing precedence:

```
exp: exp MINUS exp
  | exp TIMES exp
  | MINUS exp
```

But YACC would give the same precedence to the first and third rules.
Solution: Imaginary Tokens + Override Rule Precedence

%left PLUS MINUS
%left STAR SLASH
%left UNARY_MINUS

exp: exp MINUS exp
    | exp TIMES exp
    | MINUS exp %prec UNARY_MINUS

(The lexer never produces a UNARY_MINUS token!)