Class problems so far (1)

Write the flex rules to strip out all comments of the form /*, */ from an input program

Hints: Action ECHO copies input token to output
       Think of using 2 states
       Keyword BEGIN “state” takes you to that state

Email solution to tnm@umich.edu for feedback before Monday 28th 2008
Class Problems so far (2)

Convert this NFA to a DFA

Email solution to tnm@umich.edu for feedback before Monday 28th 2008

Class Problems so far (3)

» S → E + S | E
» E → number | (S)

Derive: \((1 + 2 + (3 + 4)) + 5\)

See later but send the completed version to tnm@umich.edu before Monday 28th 2008
Class Problems so far (4)

» S → E + S | E
» E → number | (S) | -S

❖ Do the rightmost derivation of: 1 + (2 + -(3 + 4)) + 5
❖ Email solution to tnm@umich.edu for feedback before Monday 28th 2008

Reading/Announcements

❖ Reading: Section 4.4 (top-down parsing)
❖ Working example posted on webpage for Monday 28th
» Converts expressions with infix notation to expression with prefix notation
» Running the example
  • bison –d example.y
    ♦ Creates example.tab.c and example.tab.h
  • flex example.l
    ♦ Creates lex.yy.c
  • g++ example.tab.c lex.yy.c –lfl
    ♦ g++ required here since user code uses C++ (new,<)
  • a.out < ex_input.txt
Bison Overview

Format of .y file
(same structure as lex file)

declarations

%%

rules

%%

support code

Ambiguity Review: Class Problem

S → if (E) S
S → if (E) S else S
S → other

Anything wrong with this grammar?
Grammar for Closest-if Rule

- Want to rule out: if (E) if (E) S else S
- Impose that unmatched “if” statements occur only on the “else” clauses
  - statement → matched | unmatched
  - matched → if (E) matched else matched | other
  - unmatched → if (E) statement |
    if (E) matched else unmatched

Parsing Top-Down

Goal: construct a leftmost derivation of string while reading in sequential token stream

\[
S \rightarrow E + S \mid E \\
E \rightarrow \text{num} \mid (S)
\]

<table>
<thead>
<tr>
<th>Partly-derived String</th>
<th>Lookahead</th>
<th>parsed part unparsed part</th>
</tr>
</thead>
<tbody>
<tr>
<td>E + S</td>
<td>(</td>
<td>(1+2+(3+4))+5</td>
</tr>
<tr>
<td>(S) + S</td>
<td>1</td>
<td>(1+2+(3+4))+5</td>
</tr>
<tr>
<td>(E+S)+S</td>
<td>1</td>
<td>(1+2+(3+4))+5</td>
</tr>
<tr>
<td>(1+S)+S</td>
<td>2</td>
<td>(1+2+(3+4))+5</td>
</tr>
<tr>
<td>(1+E+S)+S</td>
<td>2</td>
<td>(1+2+(3+4))+5</td>
</tr>
<tr>
<td>(1+2+S)+S</td>
<td>2</td>
<td>(1+2+(3+4))+5</td>
</tr>
<tr>
<td>(1+2+E)+S</td>
<td>(</td>
<td>(1+2+(3+4))+5</td>
</tr>
<tr>
<td>(1+2+(S))+S</td>
<td>3</td>
<td>(1+2+(3+4))+5</td>
</tr>
<tr>
<td>(1+2+(E+S))+S</td>
<td>3</td>
<td>(1+2+(3+4))+5</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem with Top-Down Parsing

Want to decide which production to apply based on next symbol

\[ S \rightarrow E + S \mid E \]
\[ E \rightarrow \text{num} \mid (S) \]

Ex1: “(1)”
\[ S \rightarrow E \rightarrow (S) \rightarrow (E) \rightarrow (1) \]

Ex2: “(1)+2”
\[ S \rightarrow E + S \rightarrow (S) + S \rightarrow (E) + S \rightarrow (1) + E \rightarrow (1) + 2 \]

How did you know to pick \( E + S \) in Ex2?
If you picked \( E \) followed by \( (S) \), you couldn’t parse it?

Grammar is Problem

\[ S \rightarrow E + S \mid E \]
\[ E \rightarrow \text{num} \mid (S) \]

- This grammar cannot be parsed top-down with only a single look-ahead symbol!
- Not \( \text{LL}(1) = \text{Left-to-right scanning, Left-most derivation, 1 look-ahead symbol} \)
- Is it \( \text{LL}(k) \) for some \( k \)?
- If yes, then can rewrite grammar to allow top-down parsing: create \( \text{LL}(1) \) grammar for same language
Making a Grammar LL(1)

S → E + S
S → E
E → num
E → (S)

• Problem: Can’t decide which S production to apply until we see the symbol after the first expression

S → ES’
S’ → ε
S’ → +S
E → num
E → (S)

• Left-factoring: Factor common S prefix, add new non-terminal S’ at decision point. S’ derives (+S)*

• Also: Convert left recursion to right recursion

Parsing with New Grammar

<table>
<thead>
<tr>
<th>S → ES’</th>
<th>S’ → ε</th>
<th>+S</th>
<th>E → num</th>
<th>(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partly-derived String</td>
<td>Lookahead</td>
<td>parsed part</td>
<td>unparsed part</td>
<td></td>
</tr>
<tr>
<td>→ ES’</td>
<td>(</td>
<td>(1+2+(3+4))+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ (S)S’</td>
<td>1</td>
<td>1</td>
<td>(1+2+(3+4))+5</td>
<td></td>
</tr>
<tr>
<td>→ (ES’)S’</td>
<td>1</td>
<td>(1+2+(3+4))+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ (1S’)S’</td>
<td>+</td>
<td>(1+2+(3+4))+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ (1+ES’)S’</td>
<td>2</td>
<td>(1+2+(3+4))+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ (1+2S’)S’</td>
<td>+</td>
<td>(1+2+(3+4))+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ (1+2+S)S’</td>
<td></td>
<td>(1+2+(3+4))+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ (1+2+ES’)S’</td>
<td>(</td>
<td>(1+2+(3+4))+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ (1+2+(S)S’)S’</td>
<td>3</td>
<td>(1+2+(3+4))+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ (1+2+(ES’)S’)S’</td>
<td>3</td>
<td>(1+2+(3+4))+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ (1+2+(3S’)S’)S’</td>
<td>+</td>
<td>(1+2+(3+4))+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ (1+2+(3+E)S’)S’</td>
<td>4</td>
<td>(1+2+(3+4))+5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Class Problem

Are the following grammars LL(1)?

\[
S \rightarrow A bc | a A cb \\
A \rightarrow b | c | \varepsilon
\]

\[
S \rightarrow a A S | b \\
A \rightarrow a | b S A
\]

Email solution to tnm@umich.edu for feedback before Monday 28\textsuperscript{th} 2008

Predictive Parsing

- LL(1) grammar:
  - For a given non-terminal, the lookahead symbol uniquely determines the production to apply
  - Top-down parsing = predictive parsing
  - Driven by predictive parsing table of
    - non-terminals x terminals \rightarrow productions
Parsing with Table

| $S \rightarrow ES'$ | $S' \rightarrow ε | +S$ | $E \rightarrow$ num $|$ (S) |
|---------------------|------------------------|--------------------------|
| Partly-derived String | Lookahead | parsed part unparsed part |
| $\rightarrow ES'$ | ( | $1+2+(3+4))+5$ |
| $\rightarrow (S)S'$ | 1 | $1+2+(3+4))+5$ |
| $\rightarrow (ES')S'$ | 1 | $1+2+(3+4))+5$ |
| $\rightarrow (1S')S'$ | + | $1+2+(3+4))+5$ |
| $\rightarrow (1+ES')S'$ | 2 | $1+2+(3+4))+5$ |
| $\rightarrow (1+2S')S'$ | + | $1+2+(3+4))+5$ |

How to Implement This?

- Table can be converted easily into a recursive descent parser
- 3 procedures: parse_S(), parse_S'(), and parse_E()
Recursive-Descent Parser

void parse_S() {
  switch (token) {
    case num: parse_E(); parse_S’(); return;
    case ‘(’ : parse_E(); parse_S’(); return;
    default: ParseError();
  }
}

<table>
<thead>
<tr>
<th></th>
<th>num</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>$\rightarrow ES’$</td>
<td>$\rightarrow ES’$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S’</td>
<td>$\rightarrow +S$</td>
<td>$\rightarrow E$</td>
<td>$\rightarrow E$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>$\rightarrow num$</td>
<td>$\rightarrow (S)$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recursive-Descent Parser (2)

void parse_S’() {
  switch (token) {
    case ‘+’ : token = input.read(); parse_S(); return;
    case ‘(’ : return;
    case EOF: return;
    default: ParseError();
  }
}

<table>
<thead>
<tr>
<th></th>
<th>num</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>$\rightarrow ES’$</td>
<td>$\rightarrow ES’$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S’</td>
<td>$\rightarrow +S$</td>
<td>$\rightarrow E$</td>
<td>$\rightarrow E$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>$\rightarrow num$</td>
<td>$\rightarrow (S)$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recursive-Descent Parser (3)

```c
void parse_E() {
    switch (token) {
        case num:
            token = input.read(); return;
        case '(': token = input.read(); parse_S();
            if (token != ')') ParseError();
            token = input.read(); return;
        default: ParseError();
    }
}
```

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>→ ES'</td>
<td>→ ES'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S'</td>
<td>→ +S</td>
<td>→ ε</td>
<td>→ ε</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>→ num</td>
<td>→ (S)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Call Tree = Parse Tree

```
```
How to Construct Parsing Tables?

Needed: Algorithm for automatically generating a predictive parse table from a grammar

Constructing Parse Tables

- Can construct predictive parser if:
  - For every non-terminal, every lookahead symbol can be handled by at most 1 production
  - FIRST(β) for an arbitrary string of terminals and non-terminals β is:
    - Set of symbols that might begin the fully expanded version of β
  - FOLLOW(X) for a non-terminal X is:
    - Set of symbols that might follow the derivation of X in the input stream
Parse Table Entries

- Consider a production $X \rightarrow \beta$
- Add $\beta$ to the $X$ row for each symbol in $\text{FIRST}(\beta)$
- If $\beta$ can derive $\varepsilon$ ($\beta$ is nullable), add $\beta$ for each symbol in $\text{FOLLOW}(X)$
- Grammar is LL(1) if no conflicting entries

<table>
<thead>
<tr>
<th></th>
<th>num</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>ES'</td>
<td></td>
<td></td>
<td></td>
<td>$S$</td>
</tr>
<tr>
<td>$S'$</td>
<td>ES'</td>
<td></td>
<td></td>
<td></td>
<td>$S'$</td>
</tr>
<tr>
<td>$E$</td>
<td>num</td>
<td>+</td>
<td></td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(S)</td>
<td></td>
</tr>
</tbody>
</table>

Computing FIRST

- Determining $\text{FIRST}(X)$
  1. if $X$ is a terminal, then add $X$ to $\text{FIRST}(X)$
  2. if $X \rightarrow \varepsilon$ then add $\varepsilon$ to $\text{FIRST}(X)$
  3. if $X$ is a nonterminal and $X \rightarrow Y_1 Y_2 ... Y_k$ then $a$ is in $\text{FIRST}(X)$ if $a$ is in $\text{FIRST}(Y_i)$ and $\varepsilon$ is in $\text{FIRST}(Y_j)$ for $j = 1 ... i-1$ (i.e., its possible to have an empty prefix $Y_1 ... Y_i$)
  4. if $\varepsilon$ is in $\text{FIRST}(Y_1 Y_2 ... Y_k)$ then $\varepsilon$ is in $\text{FIRST}(X)$
FIRST Example

\[ S \rightarrow ES' \]
\[ S' \rightarrow \varepsilon | +S \]
\[ E \rightarrow \text{number} | (S) \]

Apply rule 1: \( \text{FIRST}(\text{num}) = \{\text{num}\}, \text{FIRST}(+) = \{+\}, \text{etc.} \)
Apply rule 2: \( \text{FIRST}(S') = \{\varepsilon\} \)
Apply rule 3: \( \text{FIRST}(S) = \text{FIRST}(E) = \{\} \)
\[ \text{FIRST}(S') = \text{FIRST}(\varepsilon) + \{\varepsilon\} = \{\varepsilon, +\} \]
\[ \text{FIRST}(E) = \text{FIRST}(\text{num}) + \text{FIRST}(\varepsilon) = \{\text{num}, (\} \]
Rule 3 again: \( \text{FIRST}(S) = \text{FIRST}(E) = \{\text{num}, (\} \)
\[ \text{FIRST}(S') = \{\varepsilon, +\} \]
\[ \text{FIRST}(E) = \{\text{num}, (\} \]

Computing FOLLOW

- Determining FOLLOW(X)
  1. if \( S \) is the start symbol then \( $ \) is in FOLLOW(\( S \))
  2. if \( A \rightarrow \alpha B\beta \) then add all \( \text{FIRST}(\beta) \neq \varepsilon \) to FOLLOW(\( B \))
  3. if \( A \rightarrow \alpha B \) or \( \alpha B\beta \) and \( \varepsilon \) is in FIRST(\( \beta \)) then add FOLLOW(\( A \)) to FOLLOW(\( B \))
FOLLOW Example

\[
\begin{align*}
S & \rightarrow ES' \\
S' & \rightarrow \varepsilon | +S \\
E & \rightarrow \text{number} | (S)
\end{align*}
\]

$\text{FIRST}(S) = \{\text{num, (}\}$
$\text{FIRST}(S') = \{\varepsilon, +\}$
$\text{FIRST}(E) = \{\text{num, (}\}$

Apply rule 1: $\text{FOL}(S) = \{\}$
Apply rule 2: $S \rightarrow ES'$

$\text{FOL}(E) += \{\text{FIRST}(S') - \varepsilon\} = \{ + \}$

$S' \rightarrow \varepsilon | +S$

$E \rightarrow \text{num} | (S)$

$\text{FOL}(S) += \{\text{FIRST}(\varepsilon) - \varepsilon\} = \{\}$

Apply rule 3: $S \rightarrow ES'$

$\text{FOL}(E) += \text{FOL}(S) = \{+,\}$

(because $S'$ is nullable)

$\text{FOL}(S') += \text{FOL}(S) = \{\}$

Putting it all Together

$\text{FIRST}(S) = \{\text{num, (}\}$
$\text{FIRST}(S') = \{\varepsilon, +\}$
$\text{FIRST}(E) = \{\text{num, (}\}$

$\text{FOLLOW}(S) = \{\}$

$\text{FOLLOW}(S') = \{\}$

$\text{FOLLOW}(E) = \{+,\}$

$\text{FOLLOW}(S) = \{\}$

$\text{FOLLOW}(S') = \{\}$

$\text{FOLLOW}(E) = \{+,\}$

\begin{itemize}
  \item Consider a production $X \rightarrow \beta$
  \item Add $\rightarrow \beta$ to the $X$ row for each symbol in $\text{FIRST}(\beta)$
  \item If $\beta$ can derive $\varepsilon$ ($\beta$ is nullable), add $\rightarrow \beta$ for each symbol in $\text{FOLLOW}(X)$
\end{itemize}

\begin{center}
\begin{tabular}{|c|c|c|c|}
  \hline
  & num & + & ( \\
  \hline
  S & ES' & ES' & \\
  \hline
  S' & +S & \varepsilon & \varepsilon \\
  \hline
  E & num & (S) & \\
  \hline
\end{tabular}
\end{center}
Ambiguous Grammars

Construction of predictive parse table for ambiguous grammar results in conflicts in the table (ie 2 or more productions to apply in same cell)

\[ S \rightarrow S + S \mid S * S \mid \text{num} \]

\[ \text{FIRST}(S+S) = \text{FIRST}(S*S) = \text{FIRST}(\text{num}) = \{ \text{num} \} \]

Class Problem

\[ E \rightarrow E + T \mid T \]
\[ T \rightarrow T * F \mid F \]
\[ F \rightarrow (E) \mid \text{num} \mid \varepsilon \]

1. Compute FIRST and FOLLOW sets for this G
2. Compute parse table entries

Email solution to tnm@umich.edu for feedback before Monday 28\textsuperscript{th} 2008