Class Information

REMINDERS

• Second homework deadline extension?
• Don’t forget to work on your Linux skills (ilab)
Basic Idea:

- The parse tree is constructed from the root, expanding **non-terminal** nodes on the tree’s frontier following a left-most derivation.

- The input program is read from left to right, and input tokens are read (consumed) as the program is parsed.

- The next **non-terminal** symbol is replaced by one of its rules. The particular choice has to be unique, and uses parts of the input (partially parsed program), for instance the first **token** of the remaining input.
Review: Predictive Parsing

Basic idea:

For any two productions \( A ::= \alpha | \beta \) with \( \alpha \in (T \cup N)^* \) and \( \beta \in (T \cup N)^* \), we would like a distinct way of choosing the correct production to expand.

For \( \alpha \in (T \cup N)^* \), define \( \text{FIRST}(\alpha) \) as the set of tokens that appear as the first token in some string derived from \( \alpha \).

That is

\[
\begin{align*}
\text{x} & \in \text{FIRST}(\alpha) \quad \text{iff} \quad \alpha \Rightarrow^* x\gamma \quad \text{for some} \quad \gamma \in (T \cup N)^* \\
\text{and x is a token (x} & \in T) \text{, and} \\
\epsilon & \in \text{FIRST}(\alpha) \quad \text{iff} \quad \alpha \Rightarrow^* \epsilon
\end{align*}
\]

For a non-terminal \( A \), define \( \text{FOLLOW}(A) \) as the set of terminals that can appear immediately to the right of \( A \) in some sentential form.

Thus, a non-terminal’s FOLLOW set specifies the tokens that can legally appear after it.

A terminal symbol has no FOLLOW set

\( \text{FIRST} \) and \( \text{FOLLOW} \) sets can be constructed automatically
Predictive Parsing (cont.)

Key Property:
Whenever two productions \( A ::= \alpha \) and \( A ::= \beta \) both appear in the grammar, we would like

- \( \text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset \), and
- if \( \alpha \Rightarrow^* \epsilon \) then \( \text{FIRST}(\beta) \cap \text{FOLLOW}(A) = \emptyset \)
- Analogue case for \( \beta \Rightarrow^* \epsilon \). Note: due to first condition, at most one of \( \alpha \) or \( \beta \) can derive \( \epsilon \).

This would allow the parser to make a correct choice with a lookahead of only one symbol!
LL(1) Grammar

Define $FIRST^+(\delta)$ for rule $A ::= \delta$

- $FIRST(\delta) - \{\epsilon\} \cup \text{Follow}(A)$, if $\epsilon \in FIRST(\delta)$
- $FIRST(\delta)$ otherwise

A grammar is LL(1) iff

$(A ::= \alpha \text{ and } A ::= \beta)$ implies

$FIRST^+(\alpha) \cap FIRST^+(\beta) = \emptyset$
Back to Our Example

S ::= a S b | \( \epsilon \)

\[
\begin{align*}
FIRST(aSb) &= \{a\} \\
FIRST(\epsilon) &= \{\epsilon\} \\
FOLLOW(S) &= \{\text{eof, b}\}
\end{align*}
\]

\[
\begin{align*}
FIRST^+(aSb) &= \{a\} \\
FIRST^+(\epsilon) &= (FIRST(\epsilon) - \{\epsilon\}) \cup FOLLOW(S) = \\
&\quad \{\text{eof, b}\}
\end{align*}
\]

Is the grammar LL(1)?
Table-Driven LL(1) Parser

LL(1) parse table

Example:

\[ S ::= a \ S \ b \mid \epsilon \]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>eof</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>aSb</td>
<td>ε</td>
<td>ε</td>
<td>error</td>
</tr>
</tbody>
</table>

How to parse input a a a b b b ?
Table-driven predictive parsing algorithm

Input: a string \( w \) and a parsing table \( M \) for \( G \)

1. push \( \text{eof} \)
2. push \( \text{Start Symbol} \)
3. \( \text{token} \leftarrow \text{next_token}() \)
4. \( X \leftarrow \text{top-of-stack} \)
5. repeat
   1. if \( X \) is a terminal then
      1. if \( X = \text{token} \) then
         1. pop \( X \)
         2. \( \text{token} \leftarrow \text{next_token}() \)
         3. else error()
      2. else /* \( X \) is a non-terminal */
         1. if \( M[X, \text{token}] = X \rightarrow Y_1Y_2\cdots Y_k \) then
            1. pop \( X \)
            2. push \( Y_k, Y_{k-1}, \ldots, Y_1 \)
            3. else error()
   2. \( X \leftarrow \text{top-of-stack} \)
6. until \( X = \text{eof} \)
7. if \( \text{token} \neq \text{eof} \) then error()

See also Aho, Lam, Sethi, and Ullman, Figure 4.20, page 227
Predictive Parsing

Now, a predictive parser looks like:

Rather than writing code, we build tables.

Building tables can be automated!
Generating a Table-Driven Parser

A parser generator system often looks like:

This is true for both top down and bottom up parsers

**LL(1):** left to right, leftmost derivation, lookahead(1)

**LR(1):** left to right, reverse rightmost derivation, lookahead(1)
Recursive Descent Parsing

Now, we can produce a simple recursive descent parser from our favorite LL(1) expression grammar.

Recursive descent is one of the simplest parsing techniques used in practical compilers:

• Each non–terminal has an associated parsing procedure that can recognize any sequence of tokens generated by that non–terminal.

• There is a main routine to initialize all globals (e.g.: token) and call the start symbol. On return, check whether token == eof, and whether errors occurred. (Note: left-to-right evaluation of expressions).

• Within a parsing procedure, both non–terminals and terminals can be “matched”:
  – non–terminal A — call parsing procedure for A
  – token t — compare t with current input token; if match, consume input, otherwise ERROR

• Parsing procedures may contain code that performs some useful “computation” (syntax directed translation).
Next Lecture

Things to do:
Start programming in C. Check out the web for tutorials.

Next time:

- Recursive descent parsing;
- Programming in C, pointers, explicit memory allocation
- Read Scott 5.1 - 5.3 (some background - chapter on CD)