REMINDERS

- Third homework problem set will be posted by tomorrow.
- Sample solutions of first homework have been posted / will be posted soon.
Review - Predictive Parsing

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 $\alpha$.

- $x \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* x\gamma$ for some $\gamma \in (T \cup N)^*$ and $x$ is a token ($x \in T$), and
- $\epsilon \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* \epsilon$

For a **non-terminal** $A$, define $\text{FOLLOW}(A)$ as the set of terminals (or $\text{eof}$) that can appear immediately to the right of $A$ in some derivation step (sentential form) from the start symbol $S$.

- $x \in \text{FOLLOW}(A)$ iff $S \Rightarrow^* \alpha Ax\gamma$ for some $\alpha$ and $\gamma \in (T \cup N)^*$ and $x$ is a token ($x \in T$) or $\text{eof}$.

Thus, a non-terminal’s FOLLOW set specifies the tokens that can legally appear after it. Note: A terminal symbol has no FOLLOW set.

$\text{FIRST}$ and $\text{FOLLOW}$ sets can be constructed automatically.
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

\[(A := \alpha \text{ and } A := \beta) \text{ implies } FIRST^+(\alpha) \cap FIRST^+(\beta) = \emptyset\]

More general: A grammar is LL(1) if the $FIRST^+$ sets of all right-hand sides of the productions for a nonterminal symbol are pairwise disjoint. This allows a deterministic selection of the rule for a nonterminal symbol during top-down parsing.
Back to Our Example

$S ::= a \ S \ b \ | \ \epsilon$

$FIRST(aSb) = \{a\}$
$FIRST(\epsilon) = \{\epsilon\}$
$FOLLOW(S) = \{\text{eof, b}\}$

$FIRST^+(aSb) = \{a\}$
$FIRST^+(\epsilon) = (FIRST(\epsilon) - \{\epsilon\}) \cup FOLLOW(S) = \{\text{eof, b}\}$

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

LL(1) parse table

Example:
S ::= a S b | ϵ

<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 $G$

```plaintext
push eof
push Start Symbol
token ← next_token()

X ← top-of-stack
repeat
  if X is a terminal then
    if X = token then
      pop X
      token ← next_token()
    else error()
  else /* X is a non-terminal */
    if $M[X, \text{token}] = X \rightarrow Y_1Y_2\cdots Y_k$ then
      pop X
      push $Y_k, Y_{k-1}, \cdots, Y_1$
    else error()

  X ← top-of-stack
until X = eof

if token ≠ 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).
Recursive Descent Parsing (pseudo code)

\[
\begin{array}{c|cc|c|c}
 & a & b & \text{eof} & \text{other} \\
\hline
S & aSb & \epsilon & \epsilon & \text{error} \\
\end{array}
\]

main: {
    token := next_token( );
    if (S( ) and token == eof) print ‘’accept’’ else print ‘’error’’;
}

bool S:
    switch token {
        case a:
            token := next_token( );
            if (not S( )) return false; // recursive call to S;
            if token == b {
                token := next_token( )
                return true;
            }
            else
                return false;
        break;
        case b,
        case eof:return true;
        break;
        default: return false;
    }

How to parse input **a a a b b b**?
Next Lecture

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

Next time:

- Recursive descent parser and syntax-directed translation examples
- Programming in C, pointers, explicit memory allocation
- Read Scott 5.1 - 5.3 (some background - chapter on CD)