Announcements

• Homework set 3 has been posted.

• Updated homework set 2 solutions have been posted (sakai → Resources).

• The first programming project new deadline: Monday, October 29.

• Midterm exam, October 23, in class, 80 minutes. No lecture after the exam. You are responsible for all material up to, but not including bottom-up parsing.

• Exam review session: Monday, October 22, 1:30-2:30pm, CoRE 305 (small conference room)? Any conflicts?
Top-down parsers

- start at the root of derivation tree and fill in
- picks a production and tries to match the input
- some grammars are backtrack-free  \((\text{predictive})\)
- **LL Parsing**: reads input from **left to right** and constructs **leftmost** derivation (forwards); LL Parsing is predictive.

\[
S \Rightarrow^*_l \ xA\beta \Rightarrow^l \ x\delta\beta \Rightarrow^*_l \ xy
\]

- \(x, y \in T^*; S, A \in NT; \delta, \beta \in (T \cup NT)^*; A \rightarrow \delta \in P\)
Review - LL(1) grammars

Features

- input parsed from left to right
- leftmost derivation (forward)
- one token lookahead

Definition

A grammar $G$ is $LL(1)$ if and only if for each set of productions $A ::= \alpha_1 | \alpha_2 | \cdots | \alpha_n$

1. $\text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), \cdots, \text{FIRST}(\alpha_n)$ are all pairwise disjoint, and
2. if $\alpha_i \Rightarrow^* \epsilon$, then in addition
   $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$, for all
   $1 \leq j \leq n, i \neq j$.

What rule to select for a given non-terminal and input token can be represented in a parse table $M$.

Algorithm for $LL(1)$ parse table construction must not result in multiple entries for any $M[A, a]$ or $M[A, \text{eof}]$ (Aho, Sethi, and Ullman, Algorithm 4.4).

$\Rightarrow$ Whether a grammar is $LL(1)$ or not is decidable.
Table-driven predictive parser — LL(1)

Input: a string $w$ and a parsing table $M$ for $G$

push $\text{eof}$
push $\text{Start Symbol}$
token $\leftarrow$ next_token()

$X \leftarrow \text{top-of-stack}$
repeat
  if $X$ is a terminal then
    if $X = \text{token}$ then
      pop $X$
      token $\leftarrow$ 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()

  $X \leftarrow \text{top-of-stack}$
until $X = \text{eof}$

if token $\neq \text{eof}$ then error()

Aho, Sethi, and Ullman, Algorithm 4.3
Recursive descent parsing — LL(1)

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.

• 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)

<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>

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 b b b ?
Recursive descent parser

For our larger example grammar

goal:
    token ← next_token();
    if (expr() = ERROR | token ≠ EOF) then
        return ERROR;
    else return OK;

expr:
    if (term() = ERROR) then
        return ERROR;
    else return expr_prime();

expr_prime:
    if (token = PLUS) then
        token ← next_token(); return expr();
    else if (token = MINUS) then
        token ← next_token(); return expr();
    else if (token = eof) then
        return OK;
    else return ERROR;
Recursive Descent Parsing (Cont.)

term:
   if (factor() = ERROR) then
       return ERROR;
   else return term_prime();

term_prime:
   if (token = MULT) then
       token ← next_token(); return term();
   else if (token = DIV) then
       token ← next_token(); return term();
   else if (token = eof) then
       return OK;
   if (token = PLUS) then
       return OK;
   if (token = MINUS) then
       return OK;
   else return ERROR;

factor:
   if (token = NUM) then
       token ← next_token(); return OK;
   else if (token = ID) then
       token ← next_token(); return OK;
   else return ERROR;
**LL(1) grammars**

**Provable facts about LL(1) grammars:**

- no left recursive grammar is LL(1)
- no ambiguous grammar is LL(1)
- LL(1) parsers operate in linear time
- an \(\epsilon\)-free grammar where each alternative expansion for \(A\) begins with a distinct terminal is a *simple* LL(1) grammar

**Not all grammars are LL(1)**

- \( S ::= aS \mid a \)
  
is not LL(1)
  
  \[ \text{FIRST}(aS) = \text{FIRST}(a) = \{a\} \]

- \( S ::= aS' \)
  
- \( S' ::= aS' \mid \epsilon \)
  
  accepts the same language and is LL(1)
**LL grammars**

**LL(1) grammars**

- may need to rewrite grammar
  
  (left recursion removal, left factoring)

- resulting grammar larger, less maintainable

**LL(k) grammars**

- $k$-token lookahead

- more powerful than LL(1) grammars

- example:

  $S ::= ac \mid abc$ is LL(2)

**Not all grammars are LL(k)**

- example:

  Set of productions of form: $S ::= a^i b^j$ for $i \geq j$

- problem - must choose production after $k$ tokens of lookahead

**Bottom-up parsers avoid some of these problems**
Bottom-up parsers

- start at the leaves and fill in
- construct rightmost derivation in reverse
- find the next right-hand side of a production (handle) such that its replacement by left-hand side nonterminal will yield previous right-sentential form
- as input is consumed, change state to encode possibilities (recognize valid prefixes); if handle is found, REDUCE, otherwise SHIFT (or ERROR)

\[
S \Rightarrow_{rm}^* \alpha By \Rightarrow_{rm} \alpha \gamma y \Rightarrow_{rm}^* xy
\]

- **LR parsing**: Reads input from *left to right* and constructs **rightmost** derivation in reverse
Example

Consider the context-free grammar (in BNF notation)

1 \[ \langle \text{goal} \rangle ::= \langle A \rangle \langle B \rangle e \]
2 \[ \langle A \rangle ::= \langle A \rangle b c \]
3 \[ \quad | \quad b \]
4 \[ \langle B \rangle ::= d \]

and the input string \texttt{abbcde}.

<table>
<thead>
<tr>
<th>Prod’n.</th>
<th>Sentential Form</th>
<th>Handle(^\dagger)</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>\texttt{abcde}</td>
<td>(3,2)</td>
</tr>
<tr>
<td>3 [ a \langle A \rangle b c d e ]</td>
<td>(2,4)</td>
<td></td>
</tr>
<tr>
<td>2 [ a \langle A \rangle d e ]</td>
<td>(4,3)</td>
<td></td>
</tr>
<tr>
<td>4 [ a \langle A \rangle \langle B \rangle e ]</td>
<td>(1,4)</td>
<td></td>
</tr>
<tr>
<td>1 [ \langle \text{goal} \rangle ]</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>

Why is (3,3) not a handle for \texttt{a(A)b c d e}?

The trick appears to be scanning the input and finding valid right-sentential forms.

\(^\dagger\) (rule, position of right end of handle in input string).
Handles

We trying to find a substring $\alpha$ of the current right-sentential form where:

- $\alpha$ matches some production $A ::= \alpha$
- reducing $\alpha$ to $A$ is one step in the reverse of a rightmost derivation.

We will call such a string a *handle*.

Formally,

- a *handle* of a right-sentential form $\gamma$ is a production $A ::= \beta$ and a position in $\gamma$ where $\beta$ may be found. Convention: position specifies the right end of handle.
- If $(A ::= \beta, k)$ is a handle, then replacing the $\beta$ in $\gamma$ at position $k$ with $A$ produces the previous right-sentential form in a rightmost derivation of $\gamma$. 
Handles

Provable fact:

The substring to the right of a handle contains only terminal symbols.

Proof: Follows from the fact that all $\gamma_i$ are right-sentential forms.

Corollary

The right end of a handle is to the right of the previously reduced variable.
One scheme to implement a handle-pruning, bottom-up parser is called a \textit{shift-reduce} parser.

Shift-reduce parsers use a \textit{stack} and an \textit{input buffer}

1. initialize stack with $\$

2. Repeat until the top of the stack is the goal symbol and the input token is \texttt{eof}

   a) \textit{find the handle}
      
      if we don’t have a handle on top of the stack, \textit{shift} an input symbol onto the stack

   b) \textit{prune the handle}
      
      if we have a handle $(A ::= \beta, k)$ on top of the stack, \textit{reduce}
      
      i) pop $|\beta|$ symbols off the stack
      ii) push $A$ onto the stack