Class Information

▶ Third homework will be posted after today’s class.
▶ Section 1 new TA’s office hour and contact information announced.
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.
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

$(A ::= \alpha$ and $A ::= \beta)$ implies

$FIRST^+(\alpha) \cap FIRST^+(\beta) = \emptyset$
Review: LL(1) Parsing

Basic idea:

For any two productions $A ::= \alpha | \beta$, we would like a distinct way of choosing the correct production to expand.

For some rhs $\alpha \in G$, define $\text{FIRST}(\alpha)$ as the set of tokens that appear as the first symbol in some string derived from $\alpha$. That is

$x \in \text{FIRST}(\alpha) \iff \alpha \Rightarrow^* x\gamma$ for some $\gamma$, and

$\epsilon \in \text{FIRST}(\alpha) \iff \alpha \Rightarrow^* \epsilon$

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.

FIRST and FOLLOW sets can be constructed automatically
**FIRST set construction**

For a string of grammar symbols $\alpha$, define FIRST($\alpha$) as

- the set of terminal symbols that begin strings derived from $\alpha$
- if $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in$ FIRST($\alpha$)

FIRST($\alpha$) contains the set of tokens valid in the first position of $\alpha$

**STEP 1**: Build FIRST($X$) for all grammar symbols $X$:

1. if $X$ is a terminal, FIRST($X$) is \{X\}
2. if $X ::= \epsilon$, then $\epsilon \in$ FIRST($X$)
3. **iterate** until no more terminals or $\epsilon$ can be added to any FIRST($X$):
   
   if $X ::= Y_1 Y_2 \cdots Y_k$ then
   
   $a \in$ FIRST($X$) if $a \in$ FIRST($Y_1$) or $a \in$ FIRST($Y_i$) and $\epsilon \in$ FIRST($Y_j$) for all $1 \leq j < i$
   
   $\epsilon \in$ FIRST($X$) if $\epsilon \in$ FIRST($Y_i$) for all $1 \leq i \leq k$

   end iterate

(If $\epsilon \not\in$ FIRST($Y_1$), then FIRST($Y_i$) is irrelevant, for $1 < i$)

Zheng Zhang 6 CS@Rutgers University
The FIRST set

STEP 2: Build \textsc{FIRST}(\alpha) for \alpha = X_1X_2 \cdots X_n:

\begin{itemize}
  \item $a \in \textsc{FIRST}(\alpha)$ if $a \in \textsc{FIRST}(X_i)$
  and $\epsilon \in \textsc{FIRST}(X_j)$ for all $1 \leq j < i$
  \item $\epsilon \in \textsc{FIRST}(\alpha)$ if $\epsilon \in \textsc{FIRST}(X_i)$ for all $1 \leq i \leq n$
\end{itemize}
**FOLLOW set construction**

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 $\text{FOLLOW}$ set specifies the tokens that can legally appear after it. A terminal symbol has no $\text{FOLLOW}$ set.

To build $\text{FOLLOW}(X)$ for non-terminal $X$:

1. place eof in $\text{FOLLOW}(<\text{start}\>)$

   iterate until no more terminals or $\epsilon$ can be added to any $\text{FOLLOW}(X)$:

2. if $A ::= \alpha B \beta$ then
   
   if $\epsilon \in \text{FIRST}(\beta)$
   
   put $\{\text{FIRST}(\beta) - \epsilon\}$ in $\text{FOLLOW}(B)$
   
   put $\text{FOLLOW}(A)$ in $\text{FOLLOW}(B)$
   
   else
   
   put $\{\text{FIRST}(\beta)\}$ in $\text{FOLLOW}(B)$

3. if $A ::= \alpha B$ then

   put $\text{FOLLOW}(A)$ in $\text{FOLLOW}(B)$

end iterate
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.

▶ 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( );
    call 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

- Review on LL(1) parsing
- Read Scott 2.3.1 - 2.3.2; ALSU: 4.1 - 4.4