CS 536 Announcements for Wednesday, March 6, 2024

Last Time
- wrap up CYK
- classes of grammars
- top-down parsing

Today
- review grammar transformations
- building a predictive parser
- FIRST and FOLLOW sets

Next Time
- predictive parsing and syntax-directed translation

LL(1) Predictive Parser

Predict the parse tree top-down

Parser structure
- 1 token lookahead
- parse-selector table
- stack tracking current parse tree's frontier

Necessary conditions
- left-factored
- free of left-recursion
Review of LL(1) grammar transformations

Necessary (but not sufficient conditions) for LL(1) parsing

- free of left recursion – no left-recursive rules
- left-factored – no rules with a common prefix, for any nonterminal

Why left-recursion is a problem

Outside/high-level view

CFG snippet: \( \text{xlist} \rightarrow \text{xlist} \ X \mid \ X \)

Current parse tree: \( \text{xlist} \)  
Current token: \( \text{X} \)

How to grow parse tree?

\( \text{xlist} \)  
\( \text{xlist} \)  
\( \text{X} \)

or

\( \text{xlist} \)  
\( \text{xlist} \)  
\( \text{X} \)

Depends on if there are more \( \text{X} \)'s

⇒ need more lookahead

Inside/algorithmic-level view

CFG snippet: \( \text{xlist} \rightarrow \text{xlist} \ X \mid \ X \ \epsilon \)

Current parse tree: \( \text{xlist} \)  
Current token: \( \text{X} \)

Parse Table

\( \text{xlist} \)  
\( \text{xlist} \ X \mid \ X \ \epsilon \)

\( \text{X} \)

\( \text{EOF} \)

⇒ stack overflow
Removing left-recursion (review)

Replace

\[ A \rightarrow A \alpha \mid \beta \]

with

\[ A \rightarrow A' \]
\[ A' \rightarrow \alpha A' \mid \epsilon \]

where \( \beta \) does not start with \( A \) (or may be \( \epsilon \))

Preserves the language (as a list of \( \alpha \)'s, starting with a \( \beta \)), but uses right recursion

Example

\[ \text{xlist} \rightarrow \text{xlist X} \]
\[ \text{xlist} \rightarrow \text{E xlist'} \]
\[ \text{xlist'} \rightarrow \text{X xlist'} \mid \epsilon \]
\[ \Rightarrow \text{xlist} \rightarrow \text{X xlist'} \mid \epsilon \]
Left factoring (review)

Removing a common prefix from a grammar

Replace

\[ A \rightarrow \alpha \beta_1 | \alpha \beta_2 | ... | \alpha \beta_n | \gamma_1 | \gamma_2 | ... | \gamma_m \]

with

\[ A \rightarrow \alpha A' | \gamma_1 | \gamma_2 | ... | \gamma_m \]

\[ A' \rightarrow \beta_1 | \beta_2 | ... | \beta_n \]

where \( \beta_i \) and \( \gamma_i \) are sequence of symbols with no common prefix

Note: \( \gamma_i \) may not be present, and one of the \( \beta_i \) may be \( \epsilon \)

Idea: combine all "problematic" rules that start with \( \alpha \) into one rule \( \alpha A' \)

\( A' \) now represents the suffix of the problematic rules

Example 1

\[ \text{exp} \rightarrow \textless A \rvert \textless B \rvert \textless C \rvert D \]

\[ \text{exp} \rightarrow \textless \text{exp}' \rvert \text{ID} \]

\[ \text{exp}' \rightarrow A \rvert B \rvert C \]

Example 2

\[ \text{stmt} \rightarrow \text{ID ASSIGN exp} \rvert \text{ID ( elist )} \rvert \text{return} \]

\[ \text{exp} \rightarrow \text{INTLIT} \rvert \text{ID} \]

\[ \text{elist} \rightarrow \text{exp} \rvert \text{exp COMMA elist} \]

\[ \text{stmt} \rightarrow \text{ID stmt'} \rvert \text{return} \]

\[ \text{stmt'} \rightarrow \text{ASSIGN exp} \rvert ( \text{elist} ) \]

\[ \text{exp} \rightarrow \text{INTLIT} \rvert \text{ID} \]

\[ \text{elist} \rightarrow \text{exp elist'} \]

\[ \text{elist'} \rightarrow \epsilon \rvert \text{COMMA elist} \]
Building the parse table

Goal: given production $lhs \rightarrow rhs$, determine what terminals would lead us to choose that production

ie, figure out $T$ such that $table[lhs][T] = rhs$

- also what terminals could indicate an error at this point?

- what terminals could $rhs$ possibly start with?
- What terminals could possibly come after $lhs$?

Idea: $FIRST(rhs) =$ set of terminals that begin sequences derivable from $rhs$

Suppose top-of-stack symbol is nonterminal $p$ and the current token is $A$ and we have

- Production 1: $p \rightarrow \alpha$
- Production 2: $p \rightarrow \beta$

FIRST lets us disambiguate:

- if $A \in FIRST(\alpha)$, then $production 1$ is a viable choice
- if $A \in FIRST(\beta)$, then $production 2$ is a viable choice
- if $A$ is in just one of them, then $we$ can $predict which production to use$

FIRST sets

$FIRST(\alpha) =$ the set of terminals that begin the strings derivable from $\alpha$, and also, if $\alpha$ can derive $\varepsilon$, then $\varepsilon$ is in $FIRST(\alpha)$.

Formally,

$FIRST(\alpha) = \begin{cases} \varepsilon, T \in \Sigma & \text{if } \alpha \Rightarrow^* \varepsilon \\ T \in \Sigma & \text{if } \alpha \Rightarrow^* T \beta \\ \varepsilon \text{ or } \varepsilon \text{ if } \varepsilon \text{ is in FIRST}(\alpha) \end{cases}$

For a symbol $X$

- if $X$ is terminal: $FIRST(X) = \{X\}$
- if $X$ is $\varepsilon$: $FIRST(X) = \{\varepsilon\}$
- if $X$ is nonterminal: for each production $X \rightarrow Y_1Y_2Y_3..Y_n$
  - put $FIRST(Y_1) - \varepsilon$ into $FIRST(X)$
  - if $\varepsilon$ is in $FIRST(Y_1)$, put $FIRST(Y_2) - \varepsilon$ into $FIRST(X)$
  - if $\varepsilon$ is in $FIRST(Y_2)$, put $FIRST(Y_3) - \varepsilon$ into $FIRST(X)$
  - ...$\newline$
  - if $\varepsilon$ is in $FIRST(Y_i)$ for all $i$, put $\varepsilon$ into $FIRST(X)$
Example

Original CFG

expr → expr + term
  | term

term → term * factor
  | factor

factor → exponent ^ factor
  | exponent

exponent → INTLIT
  | ( expr )

Transformed CFG

expr → expr' expr'

expr' → + term expr'
  | ε

term → factor term'

factor → exponent factor'
  | exponent

exponent → INTLIT
  | ( expr )

FIRST FOLLOW

<table>
<thead>
<tr>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>INTLIT (</td>
</tr>
<tr>
<td>expr'</td>
<td>+ ε</td>
</tr>
<tr>
<td>term</td>
<td>INTLIT (</td>
</tr>
<tr>
<td>term'</td>
<td>* ε</td>
</tr>
<tr>
<td>factor</td>
<td>INTLIT (</td>
</tr>
<tr>
<td>factor'</td>
<td>^ ε</td>
</tr>
<tr>
<td>exponent</td>
<td>INTLIT (</td>
</tr>
</tbody>
</table>

FIRST

expr → term expr'
  | INTLIT ( |
expr' → + term expr'
  | + |
expr' → ε
  | ε |
term → factor term'
  | INTLIT ( |
term' → * factor term'
  | * |
term' → ε
  | ε |
factor → exponent factor'
  | INTLIT ( |
factor' → ^ factor
  | ^ |
factor' → ε
  | ε |
exponent → INTLIT
  | INTLIT |
exponent → ( expr )
  | ( |
Computing FIRST(\(\alpha\)) (continued)

Extend FIRST to strings of symbols \(\alpha\)

- want to define FIRST for all RHS of productions

Let \(\alpha = Y_1 Y_2 Y_3 \ldots Y_n\)
- put FIRST\((Y_1) - \epsilon\) into FIRST\((\alpha)\)
  - if \(\epsilon\) is in FIRST\((Y_1)\), put FIRST\((Y_2) - \epsilon\) into FIRST\((\alpha)\)
  - if \(\epsilon\) is in FIRST\((Y_2)\), put FIRST\((Y_3) - \epsilon\) into FIRST\((\alpha)\)
  - ...
  - if \(\epsilon\) is in FIRST\((Y_i)\) for all \(i\), put \(\epsilon\) into FIRST\((\alpha)\)

Given two productions for nonterminal \(p\)
- Production 1: \(p \rightarrow \alpha\)
- Production 2: \(p \rightarrow \beta\)

If only 1 has it, pick that production
If both have it, grammar is not LL(1)
If neither have it, if one FIRST set has \(\epsilon\) in it, look at what terminals can follow \(p\)

FOLLOW sets

For single nonterminal \(a\), FOLLOW\((a)\) is the set of terminals that can appear immediately to the right of \(a\)

Formally,

\[
\text{FOLLOW}(a) = \epsilon \cup \left( \left\{ t \in \Sigma : \alpha \rightarrow t \right\} \cup \{ T = E O F \land s \Rightarrow \ast \cdot a \} \right)
\]
Computing FOLLOW sets

To build FOLLOW(a)

- if a is the start non-term, put EOF in FOLLOW(a)
- for each production x → α a β
  - put FIRST(β) – ε into FOLLOW(a)
  - if ε is in FIRST(β), put FOLLOW(x) into FOLLOW(a)
- for each production x → α a
  - put FOLLOW(x) into FOLLOW(a)

Building the parse table

for each production x → α {
  for each terminal T in FIRST(α) {
    put α in table[x][T]
  }
  if ε is in FIRST(α) {
    for each terminal T in FOLLOW(x) {
      put α in table[x][T]
    }
  }
}