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: xlist → xlist X | X

Current parse tree: xlist Current token: X

Inside/algorithmic-level view

CFG snippet: xlist → xlist X | X

Current parse tree: xlist Current token: X

Removing left-recursion (review)

Replace

$$A \rightarrow A\ \alpha\ \mid\ \beta$$

with

$$A \rightarrow \beta\ A'$$
$$A' \rightarrow \alpha\ A'\ \mid\ \varepsilon$$

where β does not start with A (or may be ε)

Preserves the language (as a list of α's, starting with a β), but uses right recursion

Example

xlist → xlist X | ε

Left factoring (review)

Removing a common prefix from a grammar

Replace

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_m$$

with

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

where β_i and γ_i are sequence of symbols with no common prefix

Note: γ_i may not be present, and one of the β_i may be ε

Idea: combine all "problematic" rules that start with α into one rule αA'
A' now represents the suffix of the problematic rules

Example 1

exp → < A > | < B > | < C > | D

Example 2

stmt → ID ASSIGN exp | ID (elist) | return

exp → INTLIT | ID

elist → exp | exp COMMA elist

Building the parse table

Goal: given production *lhs* → *rhs*, determine what terminals would lead us to choose that production

- 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* → α
- Production 2: *p* → β

FIRST lets us disambiguate:

- if **A** ∈ FIRST(α), then
- if **A** ∈ FIRST(β), then
- if **A** is in just one of them, then

FIRST sets

FIRST(α) is the set of terminals that begin the strings derivable from α, and also, if α can derive ε, then ε is in FIRST(α).

Formally,

FIRST(α) =

For a symbol X

- if X is terminal: FIRST(X) = {X}
- if X is ε : FIRST(X) = {ε}
- if X is nonterminal : for each production X → $Y_1Y_2Y_3..Y_n$
  - put FIRST(Y_1) – ε into FIRST(X)
  - if ε is in FIRST(Y_1), put FIRST(Y_2) – ε into FIRST(X)
  - if ε is in FIRST(Y_2), put FIRST(Y_3) – ε into FIRST(X)
  - ...
  - if ε is in FIRST(Y_i) for all i, put ε into FIRST(X)

Example

Original CFG

expr → expr + term
| term
term → term * factor
| factor
factor → exponent ^ factor
| exponent
exponent → INTLIT
| (expr)

Transformed CFG

	FIRST	FOLLOW
expr		
expr'		
term		
term'		
factor		
factor'		
exponent		

	FIRST
expr → term expr'	
expr' → + term expr'	
expr' → ε	
term → factor term'	
term' → * factor term'	
term' → ε	
factor → exponent factor'	
factor' → ^ factor	
factor' → ε	
exponent → INTLIT	
exponent → (expr)	

Computing FIRST(α) (continued)

Extend FIRST to strings of symbols α

Let $\alpha = Y_1Y_2Y_3..Y_n$

- put $\text{FIRST}(Y_1) - \varepsilon$ into FIRST(α)
  - if ε is in $\text{FIRST}(Y_1)$, put $\text{FIRST}(Y_2) - \varepsilon$ into FIRST(α)
  - if ε is in $\text{FIRST}(Y_2)$, put $\text{FIRST}(Y_3) - \varepsilon$ into FIRST(α)
  - ...
  - if ε is in $\text{FIRST}(Y_i)$ for all i, put ε into FIRST(α)

Given two productions for nonterminal *p*

- Production 1: *p* → α
- Production 2: *p* → β

FOLLOW sets

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

Formally,

FOLLOW(*a*) =

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]
         }
     }
}
```