CS 536 Announcements for Wednesday, March 22, 2023

Programming Assignment 3 – due Thursday, March 23

Midterm 2 – Wednesday, March 29

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

Today
- review parse table construction
- predictive parsing and syntax-directed translation

Next Time
- static semantic analysis
- exam review

Recap of where we are

Predictive parser builds the parse tree top-down
- 1 token lookahead
- parse(selector table
- stack tracking current parse tree's frontier

Building the parse table – given production $lhs \rightarrow rhs$, determine what terminals would lead us to choose that production

\[ \text{FIRST}(\alpha) = \{ T \mid (T \in \Sigma \land \alpha \Rightarrow^* T\beta) \lor (T = \varepsilon \land \alpha \Rightarrow^* \varepsilon) \} \]

\[ \text{FOLLOW}(a) = \{ T \mid (T \in \Sigma \land s \Rightarrow^* aaT\beta) \lor (T = \text{EOF} \land s \Rightarrow^* aa) \} \]
FIRST and FOLLOW sets

FIRST(α) for α = y₁ y₂ ... yₖ
Add FIRST(y₁) – { ε }
If ε is in FIRST(y₁ to i-1), add FIRST(yᵢ) – { ε }
If ε is in all RHS symbols, add ε

FOLLOW(a) for x → α a β
If a is the start, add EOF
Add FIRST(β) – { ε }
Add FOLLOW(x) if ε is in FIRST(β) or β is empty

Note that

FIRST sets
• only contain alphabet terminals and ε
• defined for arbitrary RHS and nonterminals
• constructed by started at the beginning of a production

FOLLOW sets
• only contain alphabet terminals and EOF
• defined for nonterminals only
• constructed by jumping into production

Putting it all together
• Build FIRST sets for each nonterminal
• Build FIRST sets for each production's RHS
• Build FOLLOW sets for each nonterminal
• Use FIRST and FOLLOW sets to fill parse table for each production

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]
        }
    }
}
Example

CFG

<table>
<thead>
<tr>
<th>Production</th>
<th>First sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>s → aC</td>
<td>a</td>
</tr>
<tr>
<td>a → AB</td>
<td>A</td>
</tr>
<tr>
<td>b → D</td>
<td>D</td>
</tr>
</tbody>
</table>

FIRST and FOLLOW sets

<table>
<thead>
<tr>
<th>Production</th>
<th>FIRST sets</th>
<th>FOLLOW sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>s → aC</td>
<td>a</td>
<td>s</td>
</tr>
<tr>
<td>s → b a</td>
<td>b</td>
<td>s</td>
</tr>
<tr>
<td>a → AB</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>a → C s</td>
<td>a</td>
<td>s</td>
</tr>
<tr>
<td>b → D</td>
<td>b</td>
<td>D</td>
</tr>
<tr>
<td>b → (\varepsilon)</td>
<td>b</td>
<td>(\varepsilon)</td>
</tr>
</tbody>
</table>

Parse table

for each production \(x \rightarrow \alpha\)

for each terminal \(T\) in FIRST(\(\alpha\))

put \(\alpha\) in table[\(x\)][\(T\)]

if \(\varepsilon\) is in FIRST(\(\alpha\))

for each terminal \(T\) in FOLLOW(\(x\))

put \(\alpha\) in table[\(x\)][\(T\)]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

CFG

\[ s \rightarrow (s) | \{s\} | \varepsilon \]

FIRST and FOLLOW sets

<table>
<thead>
<tr>
<th>FIRST sets</th>
<th>FOLLOW sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td></td>
</tr>
<tr>
<td>s \rightarrow (s)</td>
<td></td>
</tr>
<tr>
<td>s \rightarrow {s}</td>
<td></td>
</tr>
<tr>
<td>s \rightarrow \varepsilon</td>
<td></td>
</tr>
</tbody>
</table>

Parse table

for each production \( x \rightarrow \alpha \)

for each terminal \( T \) in \( \text{FIRST}(\alpha) \)

put \( \alpha \) in \( \text{table}[x][T] \)

if \( \varepsilon \) is in \( \text{FIRST}(\alpha) \)

for each terminal \( T \) in \( \text{FOLLOW}(x) \)

put \( \alpha \) in \( \text{table}[x][T] \)

|     | ( | ) | { | } | EOF |
|-----|----|----|----|----|
| s   |    |    |    |    |    |
Parsing and syntax-directed translation

Recall syntax-directed translation (SDT)

To translate a sequence of tokens
- build the parse tree
- use translation rules to compute the translation of each non-terminal in the parse tree, bottom up
- the translation of the sequence is the translation of the parse tree’s root non-terminal

CFG: SDT rules:

```
expr  →  expr + term
       |  term

term  →  term * factor
       |  factor

factor →  INTLIT
        |  ( expr )
```

```
expr₁.trans = expr₂.trans + term.trans
expr.trans = term.trans

term₁.trans = term₂.trans * factor.trans
term.trans = factor.trans

factor.trans = INTLIT.value
factor.trans = expr.trans
```

The LL(1) parser never needed to explicitly build the parse tree – it was implicitly tracked via the stack.

Instead of building parse tree, give parser a second, semantic stack.

SDT rules are converted to actions

CFG: SDT actions:

```
expr  →  expr + term
       |  term

term  →  term * factor
       |  factor

factor →  INTLIT
        |  ( expr )
```

```
tTrans = pop; eTrans = pop; push(eTrans + tTrans)
tTrans = pop; push(tTrans)
fTrans = pop; tTrans = pop; push(tTrans * fTrans)
fTrans = pop; push(fTrans)
push( INTLIT.value)
eTrans = pop; push(eTrans)
```
Parsing and syntax-directed translation (cont.)

Augment the parsing algorithm
- number the actions
- when RHS of production is pushed onto symbol stack, include the actions
- when action is the top of symbol stack, pop & perform the action

CFG: SDT actions:
expr → expr + term
     | term
    tTrans = pop; eTrans = pop; push(eTrans + tTrans)

| term → term * factor
     | factor
    fTrans = pop; tTrans = pop; push(tTrans * fTrans)

| factor → INTLIT
     | ( expr )
    push( INTLIT.value)

Placing the action numbers in the productions
- action numbers go
  - after their corresponding non-terminals
  - before their corresponding terminal

Building the LL(1) parser

1) Define SDT using the original grammar
   - write translation rules
   - convert translation rules to actions that push/pop using semantic stack
   - incorporate action #s into grammar rules

2) Transform grammar to LL(1)

3) Compute FIRST and FOLLOW sets

4) Build the parse table
Example SDT on transformed grammar

Original CFG:

expr → expr + term #1  
  |  term

term → term * factor #2  
  |  factor

factor → #3 INTLIT  
  |  ( expr )

Transformed CFG:

expr → term expr'
expr' → + term #1 expr' | ε

term → factor term'
term' → * factor #2 term' | ε

factor → #3 INTLIT | ( expr )

Transformed CFG:

expr → term expr'
expr' → + term #1 expr' | ε

term → factor term'
term' → * factor #2 term' | ε

factor → #3 INTLIT | ( expr )

SDT actions:

#1 : tTrans = pop;  
eTrans = pop;  
push(eTrans + tTrans)

#2 : fTrans = pop;  
tTrans = pop;  
push(tTrans * fTrans)

#3 : push(INTLIT.val)

Parse table

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th></th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>INTLIT</th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>expr'</td>
<td></td>
</tr>
<tr>
<td>expr'</td>
<td>+</td>
<td></td>
<td>term</td>
<td>expr'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>term</td>
<td></td>
<td></td>
<td></td>
<td>term'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>term'</td>
<td>ε</td>
<td></td>
<td></td>
<td>factor term'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>term'</td>
<td></td>
<td>ε</td>
<td></td>
<td>* factor #2 term'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>factor</td>
<td></td>
<td></td>
<td></td>
<td>( expr )</td>
<td></td>
<td>#3 INTLIT</td>
<td></td>
</tr>
</tbody>
</table>
What about ASTs?

Push and pop AST nodes on the semantic stack
Keep references to nodes that we pop

**Original CFG:**

- \( \text{expr} \rightarrow \text{expr} + \text{term} \)
  - \( \text{term} \)

- \( \text{term} \rightarrow \#2 \text{INTLIT} \)

**Transformed CFG:**

- \( \text{expr} \rightarrow \text{term} \text{expr'} } \)
  - \( \text{expr'} \rightarrow + \text{term} \#1 \text{expr'} \)
  - \( \varepsilon \)
  - \( \text{term} \rightarrow \#2 \text{INTLIT} \)

**SDT actions:**

- \#1 : tTrans = pop;
  - eTrans = pop;
  - push(
- \#2 : push(

**Parse table:**