CS 536 Announcements for Monday, March 11, 2024

Programming Assignment 3 – due Friday, March 15
Midterm 2 – Thursday, March 21

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

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

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

\[
\text{Follow}(a) = \{ T \mid (T \in \Sigma \land s \Rightarrow^* a T \beta) \lor (T = \text{EOF} \land s \Rightarrow^* a) \}
\]

Current token: D

\[ s \rightarrow aC \]

Look at First(aC) and First(ba)

If D is in neither

\[ a \rightarrow \star \epsilon \]

then look at Follow(ba)
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

\[
\begin{align*}
    s & \rightarrow a \ C \ | \ ba \\
    a & \rightarrow A \ B \ | \ Cs \\
    b & \rightarrow D \ | \ \epsilon
\end{align*}
\]

FIRST and FOLLOW sets

<table>
<thead>
<tr>
<th>FIRST sets</th>
<th>FOLLOW sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s)</td>
<td>(A, C, D)</td>
</tr>
<tr>
<td>(a)</td>
<td>(A, C)</td>
</tr>
<tr>
<td>(b)</td>
<td>(D, \epsilon)</td>
</tr>
</tbody>
</table>

| \(s \rightarrow a\ C\) | \(A, C\) |
| \(s \rightarrow b\ a\) | \(D, \alpha, C\) |
| \(a \rightarrow A\ B\) | \(A\) |
| \(a \rightarrow C\ s\) | \(C\) |
| \(b \rightarrow D\)    | \(D\) |
| \(b \rightarrow \epsilon\) | \(\epsilon\) |

Parse table

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

for each terminal \(T\) in FIRST(\(\alpha\))
    put \(\alpha\) in table[\(x\)][\(T\)]

if \(\epsilon\) is in FIRST(\(\alpha\))
    for each terminal \(T\) in FOLLOW(\(x\))
        put \(\alpha\) in table[\(x\)][\(T\)]

not LL(1)

<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>aC, ba</td>
<td>aC, ba</td>
<td>ba</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td>AB</td>
<td>aC, ba</td>
<td>Cs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td>(\epsilon)</td>
<td>aC, ba</td>
<td>(\epsilon)</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>
Example

CFG

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

FIRST and FOLLOW sets

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

<table>
<thead>
<tr>
<th>Grammar</th>
<th>SDT rules:</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr → expr + term</td>
<td>expr₁.trans = expr₂.trans + term.trans</td>
</tr>
<tr>
<td></td>
<td>expr.trans = term.trans</td>
</tr>
<tr>
<td>term → term * factor</td>
<td>term₁.trans = term₂.trans * factor.trans</td>
</tr>
<tr>
<td></td>
<td>term.trans = factor.trans</td>
</tr>
<tr>
<td>factor → INTLIT</td>
<td>factor.trans = INTLIT.value</td>
</tr>
<tr>
<td></td>
<td>factor.trans = expr.trans</td>
</tr>
</tbody>
</table>

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

- holds translations of nonterms

SDT rules are converted to actions

- pop translations of RHS nonterms
- push compared translation of LHS nonterm

CFG: SDT actions:

<table>
<thead>
<tr>
<th>Grammar</th>
<th>SDT actions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr → expr + term</td>
<td>tTrans = pop; eTrans = pop; push(eTrans + tTrans)</td>
</tr>
<tr>
<td></td>
<td>tTrans = pop; push(tTrans)</td>
</tr>
<tr>
<td>term → term * factor</td>
<td>fTrans = pop; tTrans = pop; push(tTrans * fTrans)</td>
</tr>
<tr>
<td></td>
<td>fTrans = pop; push(fTrans)</td>
</tr>
<tr>
<td>factor → INTLIT</td>
<td>push(INTLIT.value)</td>
</tr>
<tr>
<td></td>
<td>eTrans = pop; push(eTrans)</td>
</tr>
</tbody>
</table>
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:
```
expr  →  expr + term
|   term

term  →  term * factor
|   factor

factor  →  INTLIT
|   ( expr )
```

SDT actions:
```
expr + term #1  tTrans = pop; eTrans = pop; push(eTrans + tTrans)

term * factor #2  fTrans = pop; tTrans = pop; push(tTrans * fTrans)

INTLIT #3  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)
- treating action #s like terminals

3) Compute FIRST and FOLLOW sets
- treating action #s like ε

4) Build the parse table
Example SDT on transformed grammar

Original CFG:

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} + \text{term} \#1 \\
& \mid \text{term} \\
\text{term} & \rightarrow \text{term} \ast \text{factor} \#2 \\
& \mid \text{factor} \\
\text{factor} & \rightarrow \#3 \text{ INTLIT} \\
& \mid ( \text{expr} )
\end{align*}
\]

Transformed CFG:

\[
\begin{align*}
\text{expr} & \rightarrow \text{term} \text{expr}' \\
\text{expr}' & \rightarrow + \text{term} \#1 \text{expr}' \mid \varepsilon \\
\text{term} & \rightarrow \text{factor} \text{term}' \\
\text{term}' & \rightarrow * \text{factor} \#2 \text{term}' \mid \varepsilon \\
\text{factor} & \rightarrow \#3 \text{ INTLIT} \mid ( \text{expr} )
\end{align*}
\]

SDT actions:

\[
\begin{align*}
#1 : & \text{tTrans} = \text{pop} ; \\
& \text{eTrans} = \text{pop} ; \\
& \text{push}(\text{eTrans} + \text{tTrans}) \\
#2 : & \text{fTrans} = \text{pop} ; \\
& \text{tTrans} = \text{pop} ; \\
& \text{push}(\text{tTrans} \ast \text{fTrans}) \\
#3 : & \text{push}(\text{INTLIT}.\text{val})
\end{align*}
\]

Parse table

<table>
<thead>
<tr>
<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></td>
</tr>
<tr>
<td>expr'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>term</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>term'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>factor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input: 5 + 3 × 2  \text{EOF}

\[
\begin{align*}
#1 : & \text{eTrans} = 5 \rightarrow 5 + 6 = 11 \\
#2 : & \text{fTrans} = 2 \rightarrow 3 \times 2 = 6
\end{align*}
\]
What about ASTs?

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

Original CFG:

expr → expr + term #1
| term

term → #2 INTLIT

Transformed CFG:

expr → term expr'
expr' → + term #1 expr'
| ε
term → #2 INTLIT

SDT actions:

#1 : tTrans = pop;
eTrans = pop;
push(new PlusNode(eTrans, tTrans))
#2 : push(new InLiLitNode(INTLIT, value))

Parse table: