CS415 Compilers

Syntax Analysis

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• **Midterm:** **Wednesday, March 21**, class time slot, in Tillett hall (two rooms); closed book, closed notes, 80 minutes
  • Tillett 257: Section 01
  • Tillett 264: Section 02
You must attend the exam for the section you are REGISTERED in!

All electronics and jackets must be stored in front of the class room (Yes, your phone is an electronic device!)

• No recitation on Wednesday March 21 after the exam

• Homework 5 has been posted. Covers a recursive descent parser. You are not responsible for this material for the midterm.
Parsing
(Syntax Analysis)

EAC Chapters 3.4
LR(1), operator precedence

1 input symbol lookahead

construct rightmost derivation (backwards)

input: read left-to-right

rule $B ::= \gamma$

$S \Rightarrow_{\text{rm}}^{*} \alpha \ B \ y \ \Rightarrow_{\text{rm}} \ \alpha \ \gamma \ y \ \Rightarrow_{\text{rm}}^{*} \ x \ y$
Parsing Techniques: Bottom-up parsers

LR(1), operator precedence

1 input symbol lookahead
construct rightmost derivation (backwards)
input: read left-to-right

\[
S \Rightarrow^{*_{rm}} \alpha B y \Rightarrow_{rm} \alpha \gamma y \Rightarrow^{*_{rm}} x y
\]

rule  \( B ::= \gamma \)

upper fringe
LR(1) Parser Example

Is the following grammar LL(1), L(2), or LR(1)?

\[ S ::= a \ b \mid a \ b \ c \]

Is the following grammar LR(1)?

\[ S ::= a \ S \ b \mid \varepsilon \]

Basic idea:

*shift* symbols from input onto the stack until top of the stack is a RHS of a rule; if so, “apply” rule backwards (*reduce*) by replacing top of the stack by the LHS non-terminal.

*Challenge*: When to shift, and when to reduce
Consider the simple grammar

<table>
<thead>
<tr>
<th></th>
<th>Goal</th>
<th>→</th>
<th>a A B e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>→</td>
<td>A b c</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>l</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>→</td>
<td>d</td>
</tr>
</tbody>
</table>

And the input string abbcde

<table>
<thead>
<tr>
<th>Sentential Form</th>
<th>Next Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbcde</td>
<td>3</td>
</tr>
<tr>
<td>a A bcde</td>
<td>2</td>
</tr>
<tr>
<td>a A de</td>
<td>4</td>
</tr>
<tr>
<td>a A B e Goal</td>
<td>1</td>
</tr>
</tbody>
</table>

The trick is scanning the input and finding the next reduction. The mechanism for doing this must be efficient.
The parser must find a substring $\beta$ of the tree’s frontier that matches some production $A \rightarrow \beta$ that occurs as one step in the rightmost derivation.

Informally, we call this substring $\beta$ a handle.

Formally,

A handle of a right-sentential form $\gamma$ is a pair $\langle A \rightarrow \beta, k \rangle$ where $A \rightarrow \beta \in P$ and $k$ is the position in $\gamma$ of $\beta$’s rightmost symbol.

If $\langle A \rightarrow \beta, k \rangle$ is a handle, then replacing $\beta$ at $k$ with $A$ produces the right sentential form from which $\gamma$ is derived in the rightmost derivation.

Because $\gamma$ is a right-sentential form, the substring to the right of a handle contains only terminal symbols.

$\Rightarrow$ the parser doesn’t need to scan past the handle (only lookahead).

$\Rightarrow$ The right end of the handle will be on top of the stack, not within the stack. Need lookahead to determine whether we reached the handle.
Critical Insight  

*(Theorem?)*  

If $G$ is unambiguous, then every right-sentential form has a unique handle.

If we can find those handles, we can build a derivation!

**Sketch of Proof:**

1. $G$ is unambiguous $\Rightarrow$ rightmost derivation is unique
2. $\Rightarrow$ a unique production $A \rightarrow \beta$ applied to derive $\gamma_i$ from $\gamma_{i-1}$
3. $\Rightarrow$ a unique position $k$ at which $A \rightarrow \beta$ is applied
4. $\Rightarrow$ a unique handle $\langle A \rightarrow \beta, k \rangle$

This all follows from the definitions
Revisit previous example

Consider the simple grammar

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
<td>→</td>
<td>a A B e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>→</td>
<td>A b c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>l</td>
<td>→</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>→</td>
<td>d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

And the input string abbcde

The trick is scanning the input and finding the next reduction
The mechanism for doing this must be efficient
LR(0) items and LR(0) canonical collection

S0: \{[Goal \to \cdot a A B e]\}

S1: \{[Goal \to a \cdot A B e], [A \to \cdot A b c], [A \to \cdot b]\}

S2: \{[Goal \to a A \cdot B e], [A \to A \cdot b c], [B \to \cdot d]\}

S3: \{[A \to b \cdot]\}

S4: \{[Goal \to a A B \cdot e]\}

S5: \{[A \to A b \cdot c]\}

S6: \{[B \to d \cdot]\}

S7: \{[A \to A b c \cdot]\}

S8: \{[Goal \to a A B e \cdot]\}
LR(1) Skeleton Parser

stack.push(INVALID); stack.push(s₀);
not_found = true;
token = scanner.next_token();
do while (not_found) {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {
        stack.popnum(2*|β|); // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);
        stack.push(GOTO[s,A]);
    }
    else if ( ACTION[s,token] == "shift s_i" ) then {
        stack.push(token); stack.push(s_i);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
            & token == EOF )
        then not_found = false;
    else report a syntax error and recover;
} report success;

The skeleton parser
• uses ACTION & GOTO tables
• does |words| shifts
• does |derivation| reductions
• does 1 accept
• detects errors by failure of 3 other cases
More Syntax Analysis (bottom-up)

Read EaC: Chapter 3.4