Lecture 6: Lexers & Parsers

CSC 131
Spring, 2019
Kim Bruce

Homework

- First line:
  - module Hmwk2 where
  - Next line should be name as comment
  - Name of program file should be Hmwk2.hs

Analysis

Source Program → Lexical Analysis → Syntax Analysis → Parse Tree → Semantic Analysis → Annotated Parse Tree

Symbol Table
Other Tables

Synthesis


Symbol Table
Other Tables
Step 1: Lexical Analysis

Lexing

- Lexer returns a list of all tokens from the input stream.
- Build from either regular expressions or (equivalently) finite automaton recognizing the tokens.
- See program LexArith.hs in class examples.
  - Haskell program uses modules to hide info

Explaining LexArith

- module LexArith(...) where
  - lists funcs and types exported (includes constructors)
- code details follow in file
  - getid :: [Char] -> [Char] -> ([Char], [Char])
    - takes string and prefix of id to first full id and rest of string to be processed
  - getnum :: [Char] -> Int -> (Int, [Char])
    - similar
  - getToken :: [Char] -> (Token, [Char])
    - takes string to pair of first recognized token and rest of list to be processed

Parsing
Parsing

- Build parse tree from an expression
- Interested in abstract syntax tree
  - drops irrelevant details from parse tree

Arithmetic grammar

<exp> ::= <exp> <addop> <term>
  | <term>
<term> ::= <term> <mulop> <factor>
  | <factor>
<factor> ::= ( <exp> )
  | NUM
  | ID
<addop> ::= + | -
<mulop> ::= * | /

Look at parse tree & abstract syntax tree for 2 * 3 + 7

Recursive Descent Parser

Base recognizer (ignore building tree now) on productions:
<exp> ::= <exp> <addop> <term>

addop (fst:rest) = if fst=='+' or fst=='-' then rest
  else error ...

exp input = let
  inputAfterExp = exp input
  inputAfterAddop = addOp inputAfterExp
  rest = term inputAfterAddop
  in
  rest
  or
  fun exp input = term(addOp(exp input));

Problems

- How do we select which production to use when alternatives?
- Left-recursive - never terminates
Rewrite Grammar

\[
\begin{align*}
<exp> & ::= <term> <termTail> & (1) \\
<termTail> & ::= <addop> <term> <termTail> & (2) \\
| & \varepsilon & (3) \\
<term> & ::= <factor> <factorTail> & (4) \\
<factorTail> & ::= <mulop> <factor> <factorTail> & (5) \\
| & \varepsilon & (6) \\
<factor> & ::= ( <exp> ) & (7) \\
| & \text{NUM} & (8) \\
| & \text{ID} & (9) \\
<addop> & ::= + | - & (10) \\
<mulop> & ::= * | / & (11) \\
\end{align*}
\]

No left recursion

How do we know which production to take?

Predictive Parsing

Goal: \( a_1a_2...a_n \)

\[
S \rightarrow \alpha \\
\quad \ldots \\
\quad \rightarrow a_1a_2X\beta
\]

Want next terminal character derived to be \( a_3 \)

\( a_3 \) in \( \text{First}(\gamma) \)

Need to apply a production \( X ::= \gamma \) where

1) \( \gamma \) can eventually derive a string starting with \( a_3 \) or
2) If \( X \) can derive the empty string, and also
   if \( \beta \) can derive a string starting with \( a_3 \).

\( a_3 \) in \( \text{Follow}(X) \)

FIRST

\* Intuition: \( b \in \text{First}(X) \) iff there is a derivation \( X \rightarrow^* b\omega \) for some \( \omega \).

1. \( \text{First}(b) = b \) for \( b \) a terminal or the empty string

2. If have \( X ::= \omega_1 | \omega_2 | \ldots | \omega_n \) then
   \[ \text{First}(X) = \text{First}(\omega_1) \cup \ldots \cup \text{First}(\omega_n) \]

3. For any right hand side \( u_1u_2...u_n \)
   - \( \text{First}(u_i) \subseteq \text{First}(u_iu_2...u_n) \)
   - if all of \( u_1, u_2..., u_{i-1} \) can derive the empty string then
     also \( \text{First}(u_i) \subseteq \text{First}(u_iu_2...u_n) \)
   - empty string is in \( \text{First}(u_iu_2...u_n) \) iff all of \( u_1, u_2..., u_n \)
     can derive the empty string

First for Arithmetic

\[
\begin{align*}
\text{FIRST}(<addop>) & = \{ +, - \} \\
\text{FIRST}(<mulop>) & = \{ *, / \} \\
\text{FIRST}(<factor>) & = \{ (, \text{NUM}, \text{ID} \} \\
\text{FIRST}(<term>) & = \{ (, \text{NUM}, \text{ID} \} \\
\text{FIRST}(<exp>) & = \{ (, \text{NUM}, \text{ID} \} \\
\text{FIRST}(<termTail>) & = \{ +, -, \varepsilon \} \\
\text{FIRST}(<factorTail>) & = \{ *, /, \varepsilon \}
\end{align*}
\]
**Follow**

- **Intuition:** A terminal $b \in \text{Follow}(X)$ iff there is a derivation $S \rightarrow^* vXb\omega$ for some $v$ and $\omega$.

1. If $S$ is the start symbol then put EOF $\in \text{Follow}(S)$
2. For all rules of the form $A ::= wXv$,
   - $a.$ Add all elements of First($v$) to Follow($X$)
   - $b.$ If $v$ can derive the empty string then add all els of Follow($A$) to Follow($X$)

- Follow($X$) only used if can derive empty string from $X$.

**Follow for Arithmetic**

- Only needed to calculate for $\langle \text{termTail} \rangle$, $\langle \text{factorTail} \rangle$!

FOLLOW($\langle \text{exp} \rangle$) = \{EOF, \}
FOLLOW($\langle \text{termTail} \rangle$) = FOLLOW($\langle \text{exp} \rangle$) = \{EOF, \}
FOLLOW($\langle \text{term} \rangle$) = FIRST($\langle \text{termTail} \rangle$) $\cup$
  - FOLLOW($\langle \text{term} \rangle$) $\cup$ FOLLOW($\langle \text{termTail} \rangle$)
    - {+, -, EOF, \}

FOLLOW($\langle \text{factorTail} \rangle$) = {+, -, EOF, \}
FOLLOW($\langle \text{factor} \rangle$) = {*, /, +, -, EOF, \}
FOLLOW($\langle \text{addop} \rangle$) = {+, -, EOF, \}
FOLLOW($\langle \text{mulop} \rangle$) = {/, NUM, ID, \} \} Not needed!

**Predictive Parsing, redux**

Goal: $a_1a_2...a_n$

$S \rightarrow \alpha$

...$
\rightarrow a_1a_2X\beta$

Want next terminal character derived to be $a_3$

Need to apply a production $X ::= \gamma$ where
1) $\gamma$ can eventually derive a string starting with $a_3$ or
2) If $X$ can derive the empty string, then see if $\beta$ can derive a string starting with $a_3$.

**Building Table**

- Put $X ::= \alpha$ in entry $(X,a)$ if either
  - $a$ in First($\alpha$), or
  - $e$ in First($\alpha$) and $a$ in Follow($X$)

- Consequence: $X ::= \alpha$ in entry $(X,a)$ iff there is a derivation s.t. applying production can eventually lead to string starting with $a$. 
Need Unambiguous

- No table entry should have more than one production to ensure it’s unambiguous, as otherwise we don’t know which rule to apply.

- Laws of predictive parsing:
  - If $A ::= \alpha_1 | ... | \alpha_n$ then for all $i \neq j$,
    \[
    \text{First}(\alpha_i) \cap \text{First}(\alpha_j) = \emptyset.
    \]
  - If $X \rightarrow^* \varepsilon$, then $\text{First}(X) \cap \text{Follow}(X) = \emptyset$.

See ArithParse.hs

<table>
<thead>
<tr>
<th>Non-terminals</th>
<th>ID</th>
<th>NUM</th>
<th>Addop</th>
<th>Mulop</th>
<th>(</th>
<th>)</th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;exp&gt;</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;termTail&gt;</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>&lt;term&gt;</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;factorTail&gt;</td>
<td>6</td>
<td>5</td>
<td></td>
<td></td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>&lt;factor&gt;</td>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;addop&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;mulop&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>II</td>
<td></td>
</tr>
</tbody>
</table>

Read off from table which production to apply!

Laws of predictive parsing:
- If $A ::= \alpha_1 | ... | \alpha_n$ then for all $i \neq j$,
  \[
  \text{First}(\alpha_i) \cap \text{First}(\alpha_j) = \emptyset.
  \]
- If $X \rightarrow^* \varepsilon$, then $\text{First}(X) \cap \text{Follow}(X) = \emptyset$.

2nd is OK for arithmetic:
- $\text{FIRST}(<\text{termTail}>) = \{+, -, \varepsilon\}$
- $\text{FOLLOW}(<\text{termTail}>) = \{\text{EOF}\}$
- $\text{FIRST}(<\text{factorTail}>) = \{*, /, \varepsilon\}$
- $\text{FOLLOW}(<\text{factorTail}>) = \{+, -, \text{EOF}\}$

More Options

- Parser Combinators
  - Domain specific language for parsing.
  - Even easier to tie to grammar than recursive descent
  - Build into Haskell and Scala, definable elsewhere
    - Talk about when cover Scala
Parser Combinators in Scala

```scala
def multOp = ("*" | "/")
def addOp = ("+" | "-")
def factor = "(" ~> expr <~ ")" | numericLit ^^ {
  ...
}
def term = factor ~ (factorTail*) ^^ {
  ...
}
def factorTail = multOp ~ factor ^^ {
  ...
}
def expr  = term ~ (termTail*) ^^ {
  ...
}
def termTail = addOp ~ term ^^ {
  ...
}
def termTail = addOp ~ term ^^ {
  ...
}
```

Syntax tree building code omitted