Lecture 13: Parsing in Haskell

CSCI 101
Spring, 2019

Kim Bruce

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

Predictive Parsing

- Want to be able to parse languages without backtracking (even done efficiently).
- Talked earlier about deterministic pda's.
  - No choice as to what to do at each step.
- Some languages seem deterministic, but don't quite work with that definition.
  - Book uses $L = a^* \cup \{a^n b^n \mid n \geq 0\}$
  - Empty stack when at end if no b's.

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

```
      <exp> ::= <term> <termTail>                 (1)
 <termTail> ::= <addop> <term> <termTail>         (2)
              | ε                                 (3)
     <term> ::= <factor> <factorTail>             (4)
<factorTail> ::= <mulop> <factor> <factorTail>    (5)
              | ε                                 (6)
   <factor> ::= ( <exp> )                         (7)
              | NUM                               (8)
              | ID                                (9)
    <addop> ::= + | -                             (10)
    <mulop> ::= * | /                             (11)
```

No left recursion

How do we know which production to take?

Predictive Parsing

Goal: $a_1a_2...a_n$

$S \rightarrow \alpha$

...

$\rightarrow a_1a_2X\beta$

Want next terminal character derived to be a_3

a_3 in First(γ)

Need to apply a production $X ::= \gamma$ where

1) γ can eventually derive a string starting with a_3 or

2) If X can derive the empty string, and also

if β can derive a string starting with a_3.

a_3 in Follow(X)

FIRST

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

1. First(b) = b for b a terminal or the empty string
2. If have $X ::= \omega_1 \mid \omega_2 \mid \dots \mid \omega_n$ then
   $\text{First}(X) = \text{First}(\omega_1) \cup \dots \cup \text{First}(\omega_n)$
3. For any right hand side $u_1u_2\dots u_n$
   - $\text{First}(u_1) \subseteq \text{First}(u_1u_2\dots u_n)$
   - if all of $u_1, u_2, \dots, u_{i-1}$ can derive the empty string then also $\text{First}(u_i) \subseteq \text{First}(u_1u_2\dots u_n)$
   - empty string is in $\text{First}(u_1u_2\dots u_n)$ iff all of $u_1, u_2, \dots, u_n$ can derive the empty string

First for Arithmetic

FIRST(<addop>) = { +, - }

FIRST(<mulop>) = { *, / }

FIRST(<factor>) = { (, NUM, ID }

FIRST(<term>) = { (, NUM, ID }

FIRST(<exp>) = { (, NUM, ID }

FIRST(<termTail>) = { +, -, ε }

FIRST(<factorTail>) = { *, /, ε }

Technically, should write down production giving the terminal — leave out here for clarity.

Follow

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

1. If S is the start symbol then put $\text{EOF} \in \text{Follow}(S)$
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 elts 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 <termTail>, <factorTail> !

FOLLOW(<exp>) = { EOF,) }

FOLLOW(<termTail>) = FOLLOW(<exp>) = { EOF,) }

FOLLOW(<term>) = FIRST(<termTail>) ∪ FOLLOW(<exp>) ∪ FOLLOW(<termTail>) = { +, -, EOF,) }

FOLLOW(<factorTail>) = { +, -, EOF,) }

*FOLLOW(<factor>) = { *, /, +, -, EOF }*

FOLLOW(<addop>) = { (, NUM, ID }

FOLLOW(<mulop>) = { (, NUM, ID }

Not needed!

Predictive Parsing, redux

Goal: $a_1 a_2 \ldots a_n$

$S \rightarrow \alpha$
$\ldots$
$\rightarrow a_1 a_2 X \beta$

Want next terminal character derived to be a_3

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

Building Table

- Put $X ::= \alpha$ in entry (X,a) if either
  - a in First(α), or
  - e in First(α) 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 $A ::= \alpha_1 \mid \ldots \mid \alpha_n$ then for all $i \neq j$, $First(\alpha_i) \cap First(\alpha_j) = \emptyset$.
  - If $X \rightarrow^* \varepsilon$, then $First(X) \cap Follow(X) = \emptyset$.

- Laws of predictive parsing:
  - If $A ::= \alpha_1 \mid \ldots \mid \alpha_n$ then for all $i \neq j$, $First(\alpha_i) \cap First(\alpha_j) = \emptyset$.
  - If $X \rightarrow^* \varepsilon$, then $First(X) \cap Follow(X) = \emptyset$.
- 2nd is OK for arithmetic:
  - FIRST(<termTail>) = { +, -, ε }
  - FOLLOW(<termTail>) = { EOF,) }
  - FIRST(<factorTail>) = { *, /, ε }
  - FOLLOW(<factorTail>) = { +, -, EOF,) }

no overlap!

See ArithParse.hs

Non-terminals	ID	NUM	Addop	Mulop	(	)	EOF
<exp>	1	1			1		
<termTail>			2			3	3
<term>	4	4			4		
<factTail>			6	5		6	6
<factor>	9	8			7		
<addop>			10				
<mulop>				11			

Read off from table which production to apply!

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

Syntax tree building code omitted

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