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

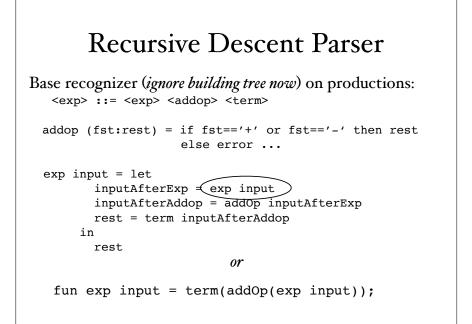
- Build parse tree from an expression
- Interested in abstract syntax tree
  - drops irrelevant details from parse tree

## **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 =  $a^* \cup \{a^nb^n \mid n \ge 0\}$
  - Empty stack when at end if no b's.

### Arithmetic grammar

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

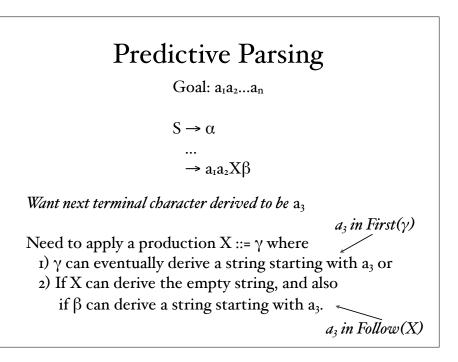


## Problems

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

#### Rewrite Grammar

<exp></exp>	::=	<term> <termtail></termtail></term>	(1)					
<termtail></termtail>	::=	<addop> <term> <termtail></termtail></term></addop>	(2)					
		ε	(3)					
<term></term>	::=	<factor> <factortail></factortail></factor>	(4)					
<factortail></factortail>	::=	<mulop> <factor> <factortail></factortail></factor></mulop>	(5)					
		3	(6)					
<factor></factor>	::=	( <exp> )</exp>	(7)					
		NUM	(8)					
		ID	(9)					
<addop></addop>	::=	+   _	(10)					
<mulop></mulop>	::=	*   /	(11)					
No left recursion								
How do we know which production to take?								



## FIRST

- Intuition:  $b \in First(X)$  iff there is a derivation  $X \rightarrow^* b\omega$  for some  $\omega$ .
- I.First(b) = b for b a terminal or the empty string
- 2. If have X ::=  $\omega_1 | \omega_2 | ... | \omega_n$  then First(X) = First( $\omega_1$ )  $\cup ... \cup$  First( $\omega_n$ )
- 3. For any right hand side  $u_1u_2...u_n$ 
  - $First(u_1) \subseteq First(u_1u_2...u_n)$
  - if all of  $u_{1}, u_{2}..., u_{i^{-1}}$  can derive the empty string then also  $First(u_{i})\subseteq First(u_{1}u_{2}...u_{n})$
  - empty string is in  $First(u_{1}u_{2}...u_{n})$  iff all of  $u_{1},\,u_{2}...,\,u_{n}$  can derive the empty string

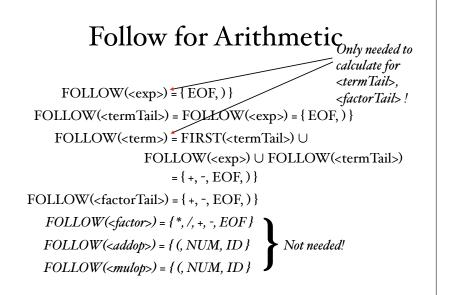
# First for Arithmetic

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  $b \in Follow(X)$  iff there is a derivation  $S \rightarrow^* vXb\omega$  for some v and  $\omega$ .
- *I*. If S is the start symbol then put  $EOF \in Follow(S)$
- 2. For all rules of the form A ::= wXv,
  - *d*.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.



## Predictive Parsing, redux

Goal:  $a_1a_2...a_n$ 

 $S \rightarrow \alpha$ 

 $\rightarrow a_1 a_2 X \beta$ 

Want next terminal character derived to be a3

Need to apply a production X ::= γ where
1) γ can eventually derive a string starting with a<sub>3</sub> or
2) If X can derive the empty string, then see if β can derive a string starting with a<sub>3</sub>.

# **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 ::= α 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  $A ::= \alpha_i \mid ... \mid \alpha_n$  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  $A ::= \alpha_i \mid ... \mid \alpha_n$  then for all  $i \neq j$ , First $(\alpha_i) \cap$  First $(\alpha_j) = \emptyset$ .
  - If  $X \rightarrow^* \epsilon$ , then  $First(X) \cap Follow(X) = \emptyset$ .

no overlap!

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

Non- terminals	ID	NUM	Addop	Mulop	(	)	EOF
<exp></exp>	Ι	I			Ι		
<termtail></termtail>			2			3	3
<term></term>	4	4			4		
<facttail></facttail>			6	5		6	6
<factor></factor>	9	8			7		
<addop></addop>			IO				
<mulop></mulop>				II			

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

