Lecture 7: Parsers & Lambda Calculus

CSC 131
Spring, 2019

Kim Bruce

Homework

- First line:
  - module Hmwk3 where
  - Next line should be name as comment
  - Name of program file should be Hmwk3.hs

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: $a_1a_2 \ldots a_n$

$S \rightarrow \alpha$

$\ldots$

$\rightarrow a_1a_2X\beta$

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 & Follow

- *Intuition*:
  - $b \in$ First(X) iff there is a derivation $X \rightarrow^* b\omega$ for some ω.
    - Let $X \rightarrow \gamma$ be first step
  - A terminal $b \in$ Follow(X) iff there is a derivation $S \rightarrow^* vXb\omega$ for some v and ω.
    - Only used if $X \rightarrow^* \varepsilon$

Using First & Follow

- If next character to be matched is b and X is left-most non-terminal
  - if $b \in$ First(X), apply $X \rightarrow \gamma$ as first step to derive b
  - if $X \rightarrow^* \varepsilon$ and $b \in$ Follow(X) then apply first step in derivation of ε
  - If neither then stuck, if both then ambiguous

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 }

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,) }

$$\left.\begin{array}{l}\textit{FOLLOW(<factor>) = \{ *, /, +, -, EOF \}}\\ \textit{FOLLOW(<addop>) = \{ (, NUM, ID \}}\\ \textit{FOLLOW(<mulop>) = \{ (, NUM, ID \}}\end{array}\right\}\ \textit{Not needed!}$$

Predictive Parsing, redux

Goal: $a_1a_2\ldots a_n$

$S \rightarrow \alpha$
$\ldots$
$\rightarrow a_1a_2X\beta$

Want next terminal character derived to be a_3

Need to apply a production X ::= γ 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 ::= α in entry (X,a) if either
  - a in First(α), or
  - ε in First(α) 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 $A ::= \alpha_1 \mid \ldots \mid \alpha_n$ then for all $i \neq j$, $\text{First}(\alpha_i) \cap \text{First}(\alpha_j) = \emptyset$.
  - If $X \rightarrow^* \varepsilon$, then $\text{First}(X) \cap \text{Follow}(X) = \emptyset$.

- Laws of predictive parsing:
  - If A ::= α_1 | …| α_n then for all i≠ j, First(α_i) ∩ First(α_j) = ∅.
  - If X →* ε, then First(X) ∩ Follow(X) = ∅.
- 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!

See Haskell Recursive Descent Parser, ParseArith.hs on web page

getTokens ::

More Options

- Stack-based parsers (pda's)
- 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 ^^{...}
```

Where are we?

Formal Syntax

- Syntax:
  - Readable, writable, easy to translate, unambiguous, ...
- Formal Grammars:
  - Backus & Naur, Chomsky
  - First used in ALGOL 60 Report - formal description
  - Generative description of language.
- Language is set of strings. (E.g. all legal C++ programs)

Example

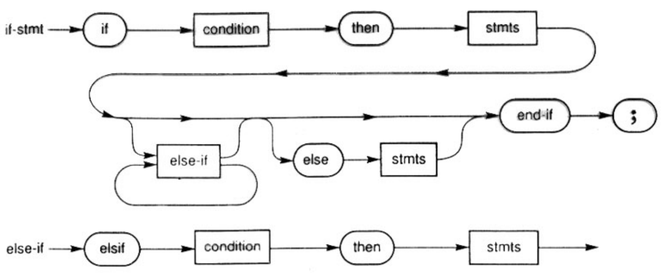
```
<exp>      ⇒  <term> | <exp> <addop> <term>
<term>     ⇒  <factor> | <term> <multop> <factor>
<factor>   ⇒  <id> | <literal> | (<exp>)
<id>       ⇒  a | b | c | d
<literal>  ⇒  <digit> | <digit> <literal>
<digit>    ⇒  0 | 1 | 2 | ... | 9
<addop>    ⇒  + | - | or
<multop>   ⇒  * | / | div | mod | and
```

Extended BNF

- Extended BNF handy:
  - item enclosed in square brackets is optional
    - <conditional> ⇒ if <expression> then <statement> [else <statement>]
  - item enclosed in curly brackets means zero or more occurrences
    - <literal> ⇒ <digit> { <digit> }

Syntax Diagrams

- Syntax diagrams - alternative to BNF.
  - Syntax diagrams are never directly recursive, use "loops" instead.



Ambiguity

```
<statement> ⇒ <unconditional> | <conditional>

 <unconditional> ⇒ <assignment> | <for loop> |
                              "{" { <statement> } "}"

 <conditional> ⇒ if (<expression>) <statement> |
                   if (<expression>)  <statement>
                                else <statement>
```

How do you parse:

```
if (exp1)
    if (exp2)
       stat1;
  else
      stat2;
```

Resolving Ambiguity

- Pascal, C, C++, and Java rule:
  - else attached to nearest then.
  - to get other form, use { ... }
- Modula-2 and Algol 68
  - No "{", only "}" (except write as "end")
- Not a problem in LISP/Racket/ML/Haskell conditional *expressions*
- Ambiguity in general is undecidable

Chomsky Hierarchy

- Chomsky developed mathematical theory of programming languages:
  - type 0: recursively enumerable
  - type 1: context-sensitive
  - type 2: context-free
  - type 3: regular
- BNF = context-free, recognized by pda

Beyond Context-Free

- Not all aspects of PL's are context-free
  - Declare before use, goto target exist
- Formal description of syntax allows:
  - programmer to generate syntactically correct programs
  - parser to recognize syntactically correct programs
- Parser-generators: LEX, YACC, ANTLR, etc.
  - formal spec of syntax allows automatic creation of recognizers

---

Specifying Semantics: Lambda Calculus

Defining Functions

- In math and LISP:
  - f(n) = n * n
  - (define (f n) (* n n))
  - (define f (<u>lambda (n) (* n n)</u>))
- In lambda calculus
  - λn. n * n
  - ((λn. n * n) 12) ⇒ 144

Pure Lambda Calculus

- Terms of pure lambda calculus
  - M ::= v | (M M) | λv. M
  - Impure versions add constants, but not necessary!
  - Turing-complete
- Left associative: M N P = (M N) P.
- Computation based on substituting actual parameter for formal parameters

Free Variables

- Substitution easy to mess up!
- Def: If M is a term, then FV(M), the collection of free variables of M, is defined as follows:
  - FV(x) = { x }
  - FV(M N) = FV(M) ∪ FV(N)
  - FV(λv. M) = FV(M) - {v}

Substitution

- Write [N/x] M to denote result of replacing all free occurrences of x by N in expression M.
  - [N/x] x = N,
  - [N/x] y = y, if y ≠ x,
  - [N/x] (L M) = ([N/x] L) ([N/x] M),
  - [N/x] (λy. M) = λy. ([N/x] M), if y≠ x and y ∉ FV(N),
  - [N/x] (λx. M) = λx. M.

Computation Rules

- Reduction rules for lambda calculus:

(α) λx. M → λy. ([y/x] M), if y ∉ FV(M).

change name of parameters if new not capture old

(β) (λx. M) N → [N/x] M.

computation by subst function argument for formal parameter

(η) λx. (M x) → M.

Optional rule to get rid of excess λ's