Lecture 13: Parsing in Haskell

CSCI 101
Spring, 2019

Kim Bruce

Midterm

- 24 hour exam
- Open book
  - Need to study!!
  - Similar to homework
- Can take in any 24 hour period between Monday @ 8:30 a.m. and Wednesday at 5 p.m.

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?

First for Arithmetic

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

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

FIRST(<factor>) = { (, NUM, ID } *rules 7, 8, 9*

FIRST(<term>) = { (, NUM, ID } *rules 4, 4, 4*

FIRST(<exp>) = { (, NUM, ID } *rules 1, 1, 1*

FIRST(<termTail>) = { +, -, ε } *rules 2, 2, 3*

FIRST(<factorTail>) = { *, /, ε } *rules 5, 5, 6*

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

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: $a_1a_2\ldots a_n$

$S \to \alpha$

…

$\to a_1a_2X\beta$

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
  - e 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 ::= α_1 | …| α_n then for all i≠ j, First(α_i) ∩ First(α_j) = ∅.
  - If $X \to^* \varepsilon$, then First(X) ∩ Follow(X) = ∅.

- 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!

Writing a Parser

- Use table to drive parser:
  - Emulate pda: StackParseArith.hs
  - Recursive descent: ParseArith.hs
    - *Build Abstract Syntax Tree!*

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 ^^{...}
```

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

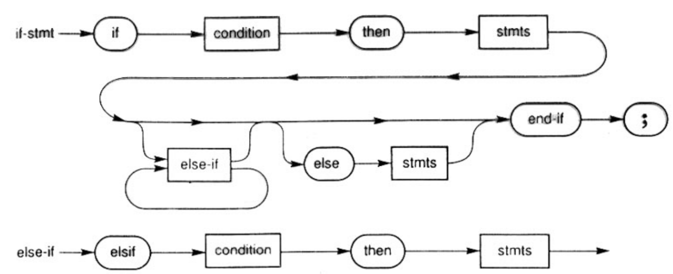
```
<exp>      ⇒  <term> | <exp> <addop> <term>
<term>     ⇒  <factor> | <term> <multop> <factor>
<factor>   ⇒  <id> | <literal> | (<exp>)
<id>       ⇒  a | b | c | d
<literal>  ⇒  <digit> | <digit> <literal>
<digit>    ⇒  0 | 1 | 2 | ... | 9
<addop>    ⇒  + | - | or
<multop>   ⇒  * | / | div | mod |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

Turing Machines

Beyond PDA's

- Grammars and machine models rich enough to represent every effective algorithm
- FSM's have no extra storage space
- PDA's can use unbounded push-down stack
- Expand to unrestricted (but finite) storage

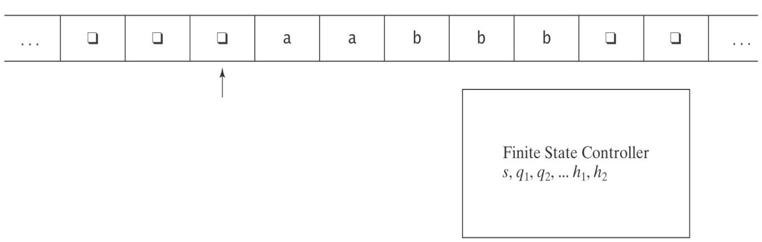
Models

- Many possible:
  - RAM: FSM with potentially infinite memory directly addressable.
  - Turing Machine: FSM with potentially infinite (both directions) tape for storage.
  - TM historically most important, but RAM more natural today.
  - Many other models possible -- but all equivalent!!
    - While language, lambda calculus, ...

What is good model?

- Powerful enough to describe all computations
- Simple enough that we can reason formally about it

Turing Machines



- At each step, the machine must:
  - choose its next state,
  - write on the current square, and
  - move left or right.

Definition

- Turing machine M is sixtuple (K, Σ, Γ, δ, s, H):
  - K is a finite set of states;
  - Σ is the input alphabet, which does not contain □;
    - □ represents "blank"
  - $\Gamma \supseteq \Sigma \cup \{\square\}$ is the tape alphabet.
  - $s \in K$ is the initial state;
  - $H \subseteq K$ is the set of halting states;
  - δ is ...