Writing a Lexical Analyzer in Haskell

Today

– (Finish up last Thursday) User-defined datatypes
– (Finish up last Thursday) Lexicographical analysis for punctuation and keywords in Haskell
– Regular languages and lexicographical analysis part I

This week

– HW2: Due tonight
– PA1: It is due in 6 days!
– PA2 has been posted. We are starting to cover concepts needed for PA2.
User-defined Datatypes in Haskell

Kind of like enumerate types but can have fields

```haskell
data Bool = False | True

data Shape = Point | Rect Int Int Int Int | Circle Int
```

Can derive handy properties

```haskell
data Color = Blue | Red | Yellow deriving (Show)

main = print Yellow

data Color = Blue | Red | Yellow deriving (Show,Eq)

if (Yellow==Blue) then ... else ...
```

Constructors can be used in pattern matching

```haskell
foo :: Shape -> String

foo Point = “Point”

foo Rect p1 p2 p3 p4 = “Rect “ ++ (show p1) ++ ...
```
Structure of a Typical Compiler

**Analysis**
- Character stream
  - Lexical analysis
    - Tokens
      - "Words"
    - Syntactic analysis
      - AST
        - "Sentences"
      - Semantic analysis
        - Annotated AST
          - Interpreter

**Synthesis**
- IR code generation
  - IR
    - Optimization
    - Code generation
      - Target language
**Tokens for Example MeggyJava program**

```java
import meggy.Meggy;

class PA3Flower {
    public static void main(String[] whatever)
    {
        // Upper left petal, clockwise
        Meggy.setPixel( (byte)2, (byte)4, Meggy.Color.VIOLET );
        Meggy.setPixel( (byte)2, (byte)1, Meggy.Color.VIOLET);
        ...
    }
}
```

**Tokens:** TokenImportKW, TokenMeggyKW, TokenSemi, TokenClassKW, TokenID "PA3Flower", TokenLBrace, ...
Some Lexical Analysis with Haskell *(why is this broken?)*

module Lexer where

import Data.Char -- needed for isSpace function

data Token
  = TokenIfKW
  | TokenComma
  -- TODO: constructors for all other tokens
  deriving (Show,Eq)

lexer :: String -> [Token]
lexer [] = []
lexer ('i':'f':rest) = TokenIfKW : lexer rest
  -- TODO: patterns for other keyword and punctuation tokens
lexer (c:rest) = if isSpace c then lexer rest else lexer (c:rest)
General Approach for Lexical Analysis

Regular Languages

Finite State Machines
  – DFAs: Deterministic Finite Automata
  – Complications when doing lexical analysis
  – NFAs: Non Deterministic Finite State Automata

From Regular Expressions to NFAs

From NFAs to DFAs
About The Slides on Languages and Finite Automata

Slides Originally Developed by Prof. Costas Busch (2004)
  – Many thanks to Prof. Busch for developing the original slide set.
Adapted with permission by Prof. Dan Massey (Spring 2007)
  – Subsequent modifications, many thanks to Prof. Massey for CS 301 slides
Adapted with permission by Prof. Michelle Strout (Spring 2011)
  – Adapted for use in CS 453
Adapted by Wim Bohm (added regular expr \( \rightarrow \) NFA \( \rightarrow \) DFA, Spr2012)
Added slides from Profs. Christian Colberg and Saumya Debray (Fall 2016)
Languages

A language is a set of strings (sometimes called sentences)

**String:** A finite sequence of letters

Examples: “cat”, “dog”, “house”, ...

Defined over a fixed alphabet:

\[ \Sigma = \{a, b, c, \ldots, z\} \]
Empty String

A string with no letters: $\varepsilon$

Observations: $|\varepsilon| = 0$

$\varepsilon w = w \varepsilon = w$

$\varepsilon abba = abba \varepsilon = abba$
Regular Expressions

Regular expressions describe regular languages
You have probably seen them in OSs / editors

Example: \((a \mid (b)(c))^*\)

describes the language

\[L((a \mid (b)(c))^*) = \{\varepsilon, a, bc, aa, abc, bca, \ldots\}\]
Recursive Definition for Specifying Regular Expressions

**Primitive regular expressions:** $\emptyset$, $\varepsilon$, $\alpha$

where $\alpha \in \Sigma$, some alphabet

Given regular expressions $r_1$ and $r_2$

\[
\begin{align*}
    r_1 & \mid r_2 \\
    r_1 r_2 & \\
    r_1^* & \\
    (r_1) &
\end{align*}
\]

Are regular expressions.
Regular operators

- **choice:** $A | B$  
  a string from $L(A)$ or from $L(B)$

- **concatenation:** $A B$  
  a string from $L(A)$ followed by a string from $L(B)$

- **repetition:** $A*$  
  0 or more concatenations of strings from $L(A)$

- $A^+$  
  1 or more

- **grouping:** $(A)$

Concatenation has precedence over choice: $A | B C$ vs. $(A | B)C$

More syntactic sugar, used in scanner generators:

- $[abc]$ means a or b or c
- $[\t\n]$ means tab, newline, or space
- $[a-z]$ means a,b,c, …, or z
Example Regular Expressions and Regular Definitions

Regular definition:

name : regular expression
name can then be used in other regular expressions

Keywords “print”, “while”

Operations: “+”, “-”, “*”

Identifiers:

let : [a-zA-Z] // chose from a to z or A to Z
dig : [0-9]
id : let (let | dig)*

Numbers: dig+ = dig dig*
Finite Automaton, or Finite State Machine (FSM)

Input

String

Finite Automaton

Output

String
Finite State Machine

Input

String

Finite Automaton

Output

“Accept” or “Reject”
abba - Finite Accepter

initial state

transition

final state “accept”
Initial Configuration

Input String

\[ a \ b \ b \ a \]

CS453 Lecture  Regular Languages and Lexical Analysis  17
Reading the Input

\[ \begin{array}{cccc}
  a & b & b & a \\
\end{array} \]
The diagram illustrates a finite automaton with states labeled $q_0, q_1, q_2, q_3, q_4,$ and $q_5$. The transitions are labeled with inputs $a$ and $b$, and the states $q_3$ and $q_5$ are marked as accept states. The input string $aba$ is shown at the top of the diagram.
\begin{align*}
q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_3 \xrightarrow{a} q_4 \\
q_4 \xrightarrow{a,b} q_5 \\
q_5 \xrightarrow{a,b} q_5
\end{align*}
Input finished

Input: $a b b a$

Output: “accept”
String Rejection

\[
\begin{array}{c|c|c}
  a & b & a \\
\end{array}
\]
Regular Languages and Lexical Analysis
Input finished

Input: $a b a$

Output: "reject"
The Empty String

\[ q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{a} q_3 \xrightarrow{b} q_4 \xrightarrow{a, b} q_5 \]

\[ \varepsilon \]
Output: “reject”

Would it be possible to accept the empty string?
Another Example

$\begin{array}{|c|c|c|}
  a & a & b \\
\end{array}$
Input finished

Output: “accept”
Rejection

\[
\begin{array}{c}
q_0 \\
\text{a} \\
b \\
q_1 \\
a, b \\
q_2
\end{array}
\]

\[
\begin{array}{c}
b \\
a \\
b \\
q_0
\end{array}
\]
Input finished

Which strings are accepted?

Output: “reject”
Formalities

Deterministic Finite Automaton (DFA)

\[ M = (Q, \Sigma, \delta, q_0, F) \]

- \( Q \): set of states
- \( \Sigma \): input alphabet
- \( \delta \): transition function
- \( q_0 \): initial state
- \( F \): set of final (accepting) states
Input Alphabet

\[ \Sigma = \{a, b\} \]
Set of States

\[ Q = \{ q_0, q_1, q_2, q_3, q_4, q_5 \} \]
Initial State
Set of Final States

\[ F = \{ q_4 \} \]
Transition Function

\[ \delta : Q \times \Sigma \rightarrow Q \]
\[ \delta(q_0, a) = q_1 \]
\[ \delta(q_0, b) = q_5 \]
\[ \delta(q_2, b) = q_3 \]
Transition Function / Table

<table>
<thead>
<tr>
<th>δ</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₀</td>
<td>q₁</td>
<td>q₅</td>
</tr>
<tr>
<td>q₁</td>
<td>q₅</td>
<td>q₂</td>
</tr>
<tr>
<td>q₂</td>
<td>q₅</td>
<td>q₃</td>
</tr>
<tr>
<td>q₃</td>
<td>q₄</td>
<td>q₅</td>
</tr>
<tr>
<td>q₄</td>
<td>q₅</td>
<td>q₅</td>
</tr>
<tr>
<td>q₅</td>
<td>q₅</td>
<td>q₅</td>
</tr>
</tbody>
</table>
Complications

1. "1234" is an NUMBER but what about the “123” in “1234” or the “23”, etc. Also, the scanner must recognize many tokens, not one, only stopping at end of file.

2. "if" is a keyword or reserved word IF, but "if" is also defined by the reg. exp. for identifier ID. We want to recognize IF.

3. We want to discard white space and comments.

4. "123" is a NUMBER but so is "235" and so is "0", just as "a" is an ID and so is "bcd", we want to recognize a token, but add attributes to it.
Before Next Time

HW2: Due tonight!

PA1: It is due in 6 days. Should be almost done.

Read Chapters 2 and 3 in the online book.