Regular Expressions

Motivation

- Begin work on building a compiler

Lexical Analysis

- First stage of parsing: *Lexical analysis*
- First stage of lexical analysis: *Tokenization*
- What is tokenization?

Definition

- Tokens = terminals
  - Recall: CFG has terminals and nonterminals
- When we *tokenize* the input string:
  - We determine how input characters are grouped into terminals
  - We call these groups of characters *tokens*

Example

- Suppose we have this CFG:
  - Terminals: NUM, PLUS
  - Nonterminal: S
  - Productions:

    S→S PLUS NUM

    S→NUM

Example

- Suppose we have input:
  12 + 3 + 456
  - Tokens: NUM, ADD, NUM, ADD, NUM
- Or, we have input:
  1 + 2 + 3
  - Tokens: NUM, ADD, NUM, ADD, NUM
- Hmm. The inputs are different, but the tokens are the same.

Tokens

- It’s not sufficient to just have the token types
- We usually consider a token to have several pieces of information:
  - Symbol: The grammar symbol (NUM, ADD, etc.)
  - Lexeme: The actual text of the input
  - Line number: For reporting errors/diagnostics

Example

- Input: 1 + 23 + 4
- Tokens:
  - (sym=NUM, lexeme= “1”)
  - (sym=PLUS, lexeme= “+”)
  - (sym=NUM, lexeme= “23”)
  - (sym=PLUS, lexeme= “+”)
  - (sym=NUM, lexeme= “4”)
- Sometimes, we create a special pseudo-token representing ”end of file”
  - Conventionally, it has a symbol of “$”

Tokenization

- How do we tokenize?
- Typically, we define terminals (tokens) by means of *regular expressions*
- We'll review these now...

Ordinary Characters

- Ordinary characters match themselves
- Ordinary means all characters except:
  $ ^ * ( ) + [ ] { } \ | . ?

.

- Dot matches any single character
- fo.
  - foo
  - fox
  - fo!
  - foolish (only matches first three letters!)

[ ]

- Character class: Denoted with [ ]
  - [aeiou] → matches lowercase vowels
  - [AEIOUaeiou] → vowels
  - [a-z] → Any lowercase letter
  - [A-Za-z0-9] → Any letter or digit
  - [^aeiou] → Any character NOT a lowercase vowel
  - [^a-zA-Z] → Any character NOT a letter
- Note: You only get one character from the [ ]'s
  - [abc] matches exactly ONE a, b, or c – not several in a row!

Shortcut

- Some character classes are so commonly used we have shortcuts for them:
  - \d → Digit: Like [0-9]
  - \D → Not a digit: Like [^0-9]
  - \s → Whitespace (space, tab, newline, ...)
  - \S → Not whitespace
  - \w → “Word” character (letter, digit, _)
  - \W → Not a “word” character

?

- Question mark indicates optional
  - xy?z → xz or xyz
  - x(ab)?z → xz or xabz

*

- Asterisk indicates zero or more repetitions of immediately preceding item
  - x* → Any number of x's (even zero!)
  - ab*c → ac, abc, abbc, abbbc, ...
  - a(bc)*d → ad, abcd, abcbcd, abcbcbcd, ...

+

- Plus indicates one or more repetitions of immediately preceding item
  - x+ → Any positive number of x’s
  - ab+c → abc, abbc, abbbc, ...
  - a(bc)+d → abcd, abcbcd, abcbcbcd, ...
- This is just a shortcut: We can always get same result by using *
- Ex: These are equivalent:
  - ab+c
  - abb*c

{ }

- We can control repetition by using braces
  - ab{3,5}c → abbbc, abbbbc, abbbbbc
  - ab{,3}c → ac, abc, abbc, abbbc
  - ab{3,}c → abbbc, abbbbc, abbbbbc, ...
    - No upper limit to repetition

|

- Pipe gives alternation
  - a(bc|def)g → abcg or adefg
  - foo(bar|zim) → foobar or foozim
- May need to use parentheses to control precedence
  - ab|cd → ab or cd
  - a(b|c)d → abd or acd

Anchors

- ^ means “beginning of string” (when it’s not in [ ]’s)
- $ means “end of string”
- Ex: ^abc matches only if string begins with “abc”
- Ex: xyz$ matches only if string ends with “xyz”

Boundary

- \b matches on a boundary
  - Transition between \w and \W
- Regex of \bis\b
  - Matches “this is the isolated stuff”
- Regex of \bis
  - Matches “this is the isolated stuff”
- Regex of is\b
  - Matches “this is the isolated stuff”
- Regex of is
  - Matches “this is the isolated stuff”
- Note: Start and end of text is also a boundary point
  - “is\b” not the same as “is\s”
  - Different if text ends with “is” (with no trailing space)

Uses

- We often use \b when specifying reserved words
- Ex: Suppose we want to match the keyword “if”
- Why would we not want to specify the regex as: “if”?

Problem

- Suppose we have:
  iffy = 42
- This will parse as token “IF”, followed by token “FY”
- Better: Specify regex for ’if’ keyword:
  \bif\b

Modifiers

- Some platforms (Python, Java, C#) allow regex to start with special prefix:
    - (?i) → Case insensitive
    - (?m) → Multiline: ^,$ match on each line too
    - (?s) → Dotall: Dot matches newlines
    - (?x) → Verbose: Ignore spaces in regex (unless in []'s or preceded with \) and treat # (when outside []'s) as comment character (until end-of-line)
    - These can be combined: (?si)
- Other platforms (C++, Javascript) require special modifiers when creating regex

Greed

- Quantifiers (*, +, {}, ?) are greedy: Consume as much input as possible.
- Example:
  - Regex: <a.*>
  - Input: Click <a href=”www.xyz.com”>here</a> now!
- But:
  - Regex: <a .*?>
  - Input: Click <a href=”www.xyz.com”>here</a> now!

Greed

- Nongreedy (“lazy”) stops as soon as it can
- Greedy stops only when it must
  - Greedy will never be so greedy as to cause match to fail if it could somehow succeed
  - Lazy will never be so lazy as to cause match to fail if it could somehow succeed

Backreferences

- ( ) create *capture groups*
- We can reference them later with \1, \2, etc.
- Ex: (\w+)\1
  - Matches “foo foo” but not “foo bar”
- Ex: (\w+)\s(\w+)\s\1\2
  - Matches “foo bar foo bar”

Lookaround

- (?=...stuff...) Matches if stuff found, does not consume
- (?!...stuff...) Matches if stuff not found, does not consume
- How are these useful?

Lookaround

- Ex: Suppose language contains != for not equal and ! for factorial
- Regular expression for factorial token:
  - !(?!=)
  - Exclamation but only if NOT followed by equals sign
- Ex: Suppose we support < for “less than” and << for “left shift”
  - Regex for less operator: <(?!<)
  - Regex for shift operator: <<

Usage

- Now we'll look at using regex for a few toy problems
- Our application: Search a phone number, print area code + number
- Our regex:
  - (\(?\d{3}\)?)?\s*(\d{3})-?(\d{4})
- A trick: We can often use [ ] to get literals instead of \ escape: ([(]?\d{3}[)]?)?\s*(\d{3})-?(\d{4})
- Sometimes it's better to repeat ourselves instead of using {}: ([(]?\d\d\d[)]?)?\s*(\d\d\d)-?(\d\d\d\d)
- Let's break this up in pieces...

Regex

- (\(?\d{3}\)?)? \s*(\d{3})-?(\d{4})
- Outer ()'s define a capture group so we can refer to area code later
- \(? and \)? specify optional ()'s around area code
- Since ( ) are metachars, we must escape them
- Note: We don't protect against mismatched parens (opening without closing or vice versa)

Regex

- (\(?\d{3}\)?)? \s* (\d{3})-?(\d{4})
- Optional: Any number of spaces (even zero)

Regex

- (\(?\d{3}\)?)?\s* (\d{3}) -?(\d{4})
- Exactly three digits
- We define capture group so we can refer to them later

Regex

- (\(?\d{3}\)?)?\s*(\d{3})-?(\d{4})
- Optional hyphen

Regex

- `(\(?\d{3}\)?)?\s*(\d{3})-?(\d{4})`
- Exactly four digits
- Define capture group to get them

Note

- What if we received this input:
- (800)555-1234567
- Our regex matches it!
- How to fix?

Fix

- (\(?\d{3}\)?)?\s*(\d{3})-?(\d{4})(?!\d)
    - Add negative lookahead: No digits
    - But this doesn’t prevent things like: (800)555-1234abcdef
- ^(\(?\d{3}\)?)?\s*(\d{3})-?(\d{4})$
    - Anchor beginning and end of input

Setup

- We need to create a regex object and then use it
- Since regex creation is costly, we usually do it once, at startup

```
using System.Text.RegularExpressions;
...
var rex = new Regex("(\\(?\\d{3}\\)?)?\\s*(\\d{3})-?(\\d{4})");
```

- Notice how we must escape backslashes
  - Two rounds of interpretation: The C# parser and then the regex engine

Better

- Since this is so annoying, C# provides *verbatim strings* (like Python r-strings)

```
using System.Text.RegularExpressions;
...
var rex = new Regex(@"(\(?\d{3}\)?)?\s*(\d{3})-?(\d{4})");
```

- To get quotation mark in verbatim string: Use two “ marks, one after the other

Match

- See if we have a match and print the phone number in a standard format:

```
var m = rex.Match(s, start_idx);
string areacode, exchange, extension;
if( m.Success ){
    if( m.Groups[1].Success )
        areacode = m.Groups[1].Value;
    else
        areacode = "";
    exchange =  m.Groups[2].Value;
    extension = m.Groups[3].Value;
    Console.WriteLine(areacode+exchange+extension);
}
```

- Note: Group 0 = entire matched string

Assignment

- Write a C# program which takes a single *command line argument*. This will be the name of a file.
- Each line in the file will be of the form:
  lhs -> someregex
- Read the lines of the file into some sort of collection. Print an error message if:
  - Any of the regexes are invalid
  - Any lhs's are repeated
- Stop when either:
  - You reach the end of the file
  - You see a blank line (be careful of \r\n vs. \n: I suggest you use Trim())
- Code follows...

Note

- If you want to support both GUI file selection and command line arguments:

```
//Need to right click solution, "Add reference",
//choose System.Windows.Forms
using System.Windows.Forms;
using System;
class Main{
    [STAThread]
    public static void Main(string[] args)
    {
        string infile;
        if(args.Length == 0) {
            OpenFileDialog dlg = new OpenFileDialog();
            dlg.Filter = "All files|*.*";
            dlg.ShowDialog();
            infile = dlg.FileName;
            if(infile.Trim().Length == 0)
                return;
            dlg.Dispose();
        } else {
            infile = args[0];
        }
        ...
    }
}
```

Sources

- Aho, Lam, Sethi, Ullman. Compilers: Principles, Techniques, and Tools. 2nd ed.
- K. Louden. Compiler Construction: Principles and Practice
- Python Tutorial. http://www.python.org
- Java Documentation. http://java.sun.com
- Boost Documentation. http://www.boost.org
- PCRE Documentation
- Regexp. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp
- http://en.cppreference.com/w/cpp/regex/regex_search

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org