Practical Regular Expressions:

$\boxed{\text{a Virtual Machine Approach}}$

by

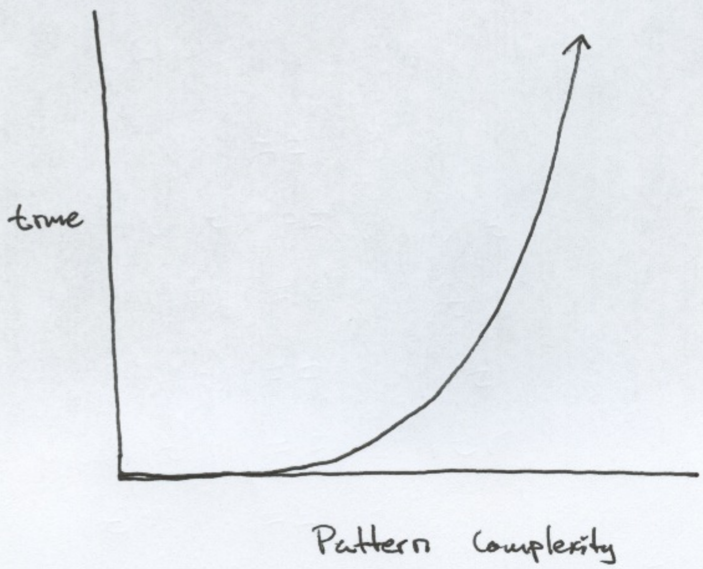
Tim Henderson

tim.tadh@gmail.com

inspired by:

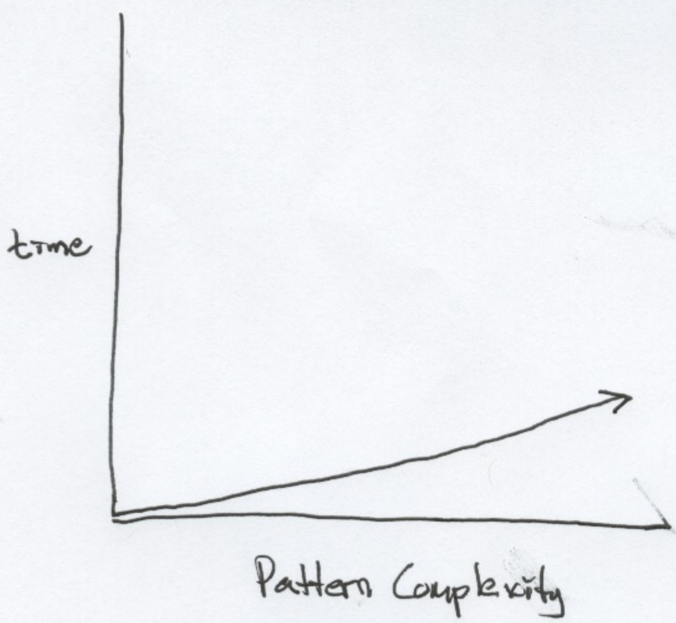
Russ Cox : swtch.com/~rsc/regexp/
Jeff Ullman et all. : "Dragon Book" ch. 3
Ken Thompson : original inventor of technique
↳ "Regular Expression Search Algorithm"
Comm. of ACM June <u>1968</u> pp. 419-422
Kenneth Rosen : Descrete Mathematics + its Applications.

Motivation



Python
Perl
Ruby
PCRE

$O(2^{nm})$



Thompson NFA Simulation
US!
Gnu Grep
Gnu awk, awk
TCL

$O(n \cdot m)$

Part 1: Review

Today we will use the Kernal of Regular Exp pattern Matching Langause.

- All other "regular" features can be derived from this Kernel.

alphabet = {a, b} ....

Charater → any letter from the Alphabet ⇒ $\underline{c}$

$\underline{c}$ Matches one charater

ex:

a → {a}

b → {b}

Concatenation

$\underline{c}\,\underline{c} \ldots \underline{c}$ matches a string of charaters

ex:

aaa → {aaa}

aba → {aba}

Union

$\underline{c}_1 | \underline{c}_2$ Matches either $\underline{c}_1$ or $\underline{c}_2$ but not both

ex:

a|b → {a, b}

b|c → {b, c}

ab|aa → {aa, ab}

Part 1 Cont...

Grouping

(r) Matches the exp. r

ex:

(a) → {a}

(a|b) → {a, b}

(a|b)b → {ab, bb}

Kleene Star | Repetition operator

r* Matches the exp r 0 to ∞ times

a* → {∅, a, aa, aaa, ...}

(a|b)* → {∅, a, b, ab, ba, ...}

aa, bb abb

One or None

r? Matches the exp 0 or 1 times

a?b → {b, ab}

(a|b)?b → {b, ab, bb}

Part 1 cont...

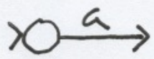
Algebraic Rules / Laws

Law	
$r\|s == s\|r$	Comm.
$r\|(s\|t) == (r\|s)\|t$	Assoc.
$r(st) == (rs)t$	Assoc.
$r(s\|t) == rs\|rt$	dist.
$(s\|t)r == sr\|tr$	dist.
$\epsilon r == r$	
$r\epsilon == r$	ϵ is the empty string
$(r\|\epsilon)^* == r^*$	
$r^{**} == r^*$	

Part 1 Cont... NFA Matching

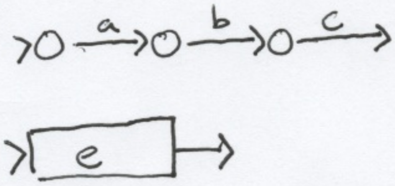
Singe Character

a



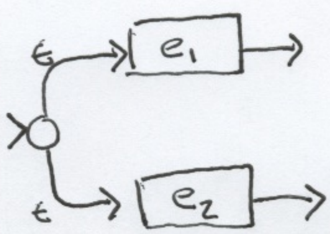
~~Character~~ Group

(abc) = e



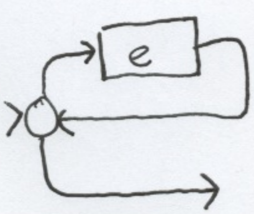
Union

$e_1 | e_2$



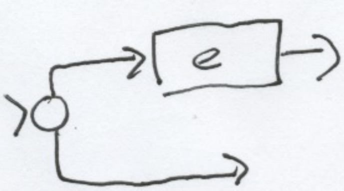
Kleene Star

e^*

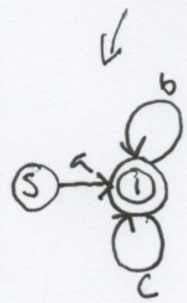


One or None

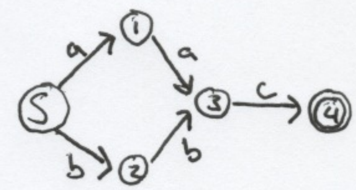
e?



a(b|c)*



(aa|bb)c



a b c

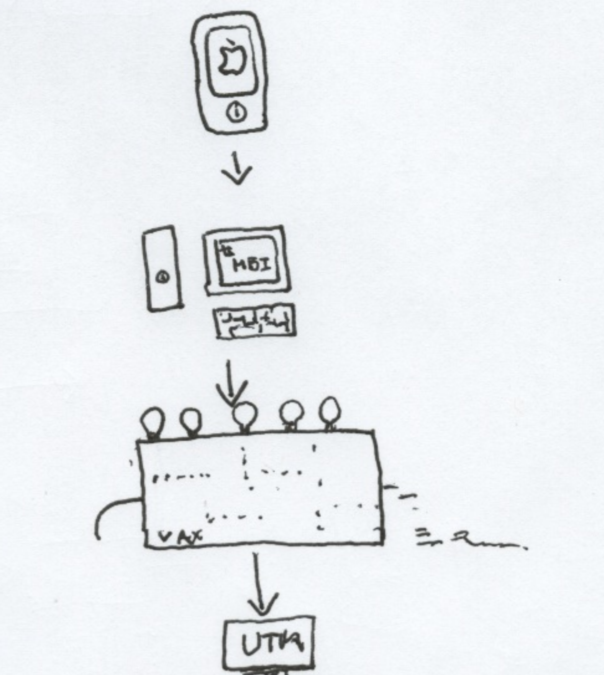
yes

Question: How do we simulate these with a compute?

Part 2

VM's

What exactly are VM's?



it is UTM's
all the
way
d
o
w
n

Steps toward excuting Regular exp with Finite Automata

Step 1: Define Machine Language

Step 2: Define excution Semantics.

Step 3: Define Conversion from NFA to Machine language.

Step 4: Develop Algorithm to execute Language.

Step 5: Profit.

Part 3

Lets

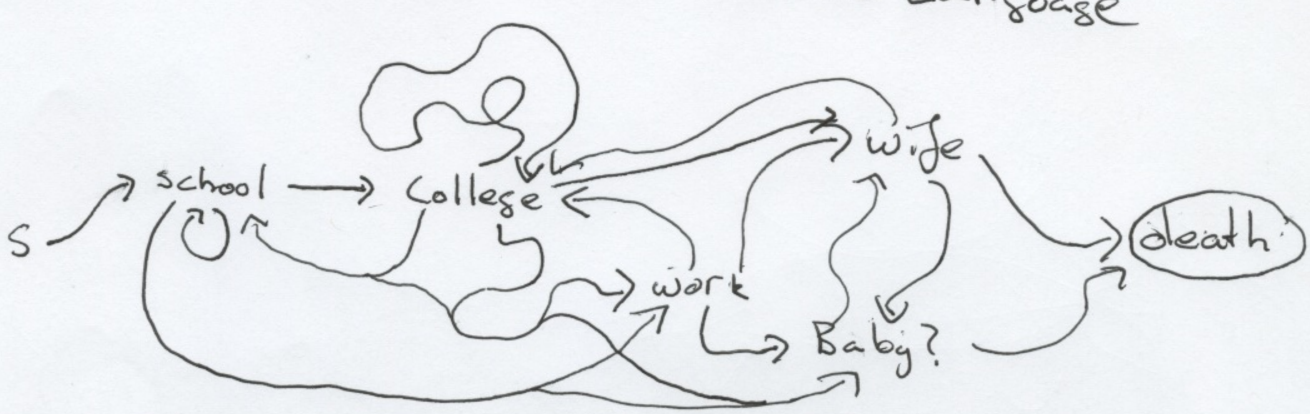
Get

Down

to

Business

From NFA → Machine Language

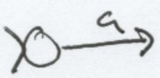


Part 3

Language

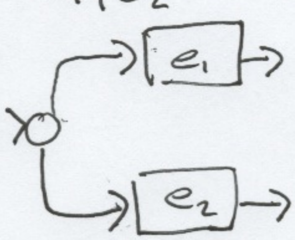
OP	Args
CHAR	Character
SPLIT	left, Right
JMP	X
Match	

a



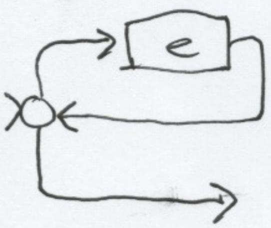

```
Char a
```

$e_1 | e_2$



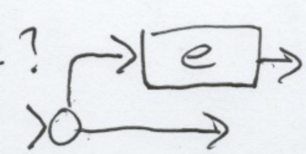

```
    Split e1, e2
e1:   :
    Jmp end
e2:   :
end:
```

e^*




```
L1: Split e, end
e:    :
    Jmp L1
end:
```

e?



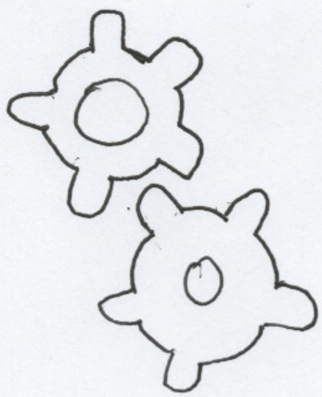

```
Split e, end
e:   :
end:
```

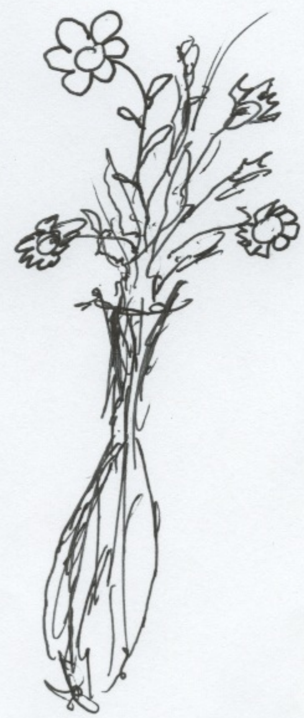
a(b|c)*

```
0 Char a
1 Split 2, 7
2 split 3, 5
3 Char b
4 JMP 6
5 Char C
6 Jmp 1
7 Match
```

Part 4:

The Hard PaRT (tm)





The

Matching

Algorithm

```
package inst

import "fmt"

const (
    CHAR    = iota   = 0
    SPLIT
    JMP              = 1
    MATCH
)                    = 2
                     = 3
type Inst struct {
    Op  uint8
    X   uint32
    Y   uint32
}

type InstSlice []*Inst

func New(op uint8, x, y uint32) *Inst {
    self := new(Inst)
    self.Op = op
    self.X = x
    self.Y = y
    return self
}

func (self Inst) String() (s string) {
    switch self.Op {
        case CHAR:
            s = fmt.Sprintf("CHAR   %c", byte(self.X))
        case SPLIT:
            s = fmt.Sprintf("SPLIT  %v, %v", self.X, self.Y)
        case JMP:
            s = fmt.Sprintf("JMP    %v", self.X)
        case MATCH:
            s = "MATCH"
    }
    return
}

func (self InstSlice) String() (s string) {
    s = "{\n"
    for i, inst := range self {
        if inst == nil { continue }
        if i < 10 {
            s += fmt.Sprintf("    0%v %v\n", i, inst)
        } else {
            s += fmt.Sprintf("    %v %v\n", i, inst)
        }
    }
    s += "}"
    return
}
```

Program Text yes/no

```go
package machines

import "inst"

func recursive(program inst.InstSlice, text []byte, pc, tc uint32) bool {
    if int(pc) >= len(program) || int(tc) > len(text) {
        return false
    }
```

sanity check

cast

```go
    switch program[pc].Op {
        case inst.MATCH:
            if tc == uint32(len(text)) {
                return true
            }
            return false
```

builtin

```go
        case inst.CHAR:
            if tc == uint32(len(text)) || text[tc] != byte(program[pc].X) {
                return false
            }
            return recursive(program, text, pc+1, tc+1)
        case inst.JMP:
            return recursive(program, text, program[pc].X, tc)
        case inst.SPLIT:
```

3 3

```go
            if recursive(program, text, program[pc].X, tc) {
                return true
            }
```

3 3

```go
            return recursive(program, text, program[pc].Y, tc)
    }
    return false
}

func Recursive(program inst.InstSlice, text []byte) bool {
    return recursive(program, text, 0, 0)
}
```

```go
package thread

import "container/list"

type Thread struct {
    Pc uint32
    Tc uint32
}

func NewThread(pc, tc uint32) *Thread {
    self := new(Thread)
    self.Pc = pc
    self.Tc = tc
    return self
}

type Stack struct {
    list *list.List
}

func NewStack() *Stack {
    self := new(Stack)
    self.list = list.New()
    return self
}

func (self *Stack) Empty() bool { return self.list.Len() <= 0 }

func (self *Stack) Push(t *Thread) {
    self.list.PushFront(t)
}

func (self *Stack) Pop() *Thread {
    e := self.list.Front()
    t, _ := e.Value.(*Thread)
    self.list.Remove(e)
    return t
}
```

← constructor

stack.Empty()

Methods

```go
package machines

import . "inst"
import . "thread"

func Backtracking(program InstSlice, text []byte) bool {
    var stack *Stack = NewStack()
    var thread *Thread
    stack.Push(NewThread(0, 0))

    for !stack.Empty() {
        thread = stack.Pop()
        inner: for {
            if int(thread.Pc) >= len(program) || int(thread.Tc) > len(text) { return false
            }
            inst := program[thread.Pc]
            switch inst.Op {
                case CHAR:
                    if int(thread.Tc) >= len(text) || text[thread.Tc] != byte(inst.X) {
                        break inner
                    }
                    thread.Pc++
                    thread.Tc++
                case MATCH:
                    if thread.Tc == uint32(len(text)) {
                        return true
                    }
                    break inner
                case JMP:
                    thread.Pc = inst.X
                case SPLIT:
                    stack.Push(NewThread(inst.Y, thread.Tc))
                    thread.Pc = inst.X
            }
        }
    }
    return false
}
```

PC TC

infinite loop

label

Program

Text

```
package queue

import "container/list"

type Queue struct {
    list    *list.List
    set     map[uint32] bool
}

func New() *Queue {
    self := new(Queue)
    self.list = list.New()
    self.set = make(map[uint32] bool)
    return self
}

func (self *Queue) Empty() bool { return self.list.Len() <= 0 }

func (self *Queue) Push(pc uint32) {
    if _, ok := self.set[pc]; ok { return }
    self.set[pc] = true, true
    self.list.PushBack(pc)
}

func (self *Queue) Pop() uint32 {
    e := self.list.Front()
    pc, _ := e.Value.(uint32)
    self.list.Remove(e)
    self.set[pc] = false, false
    return pc
}

func Swap(a, b *Queue) {
}
```

Applys to The current Text Position

Applies to the next text position

current

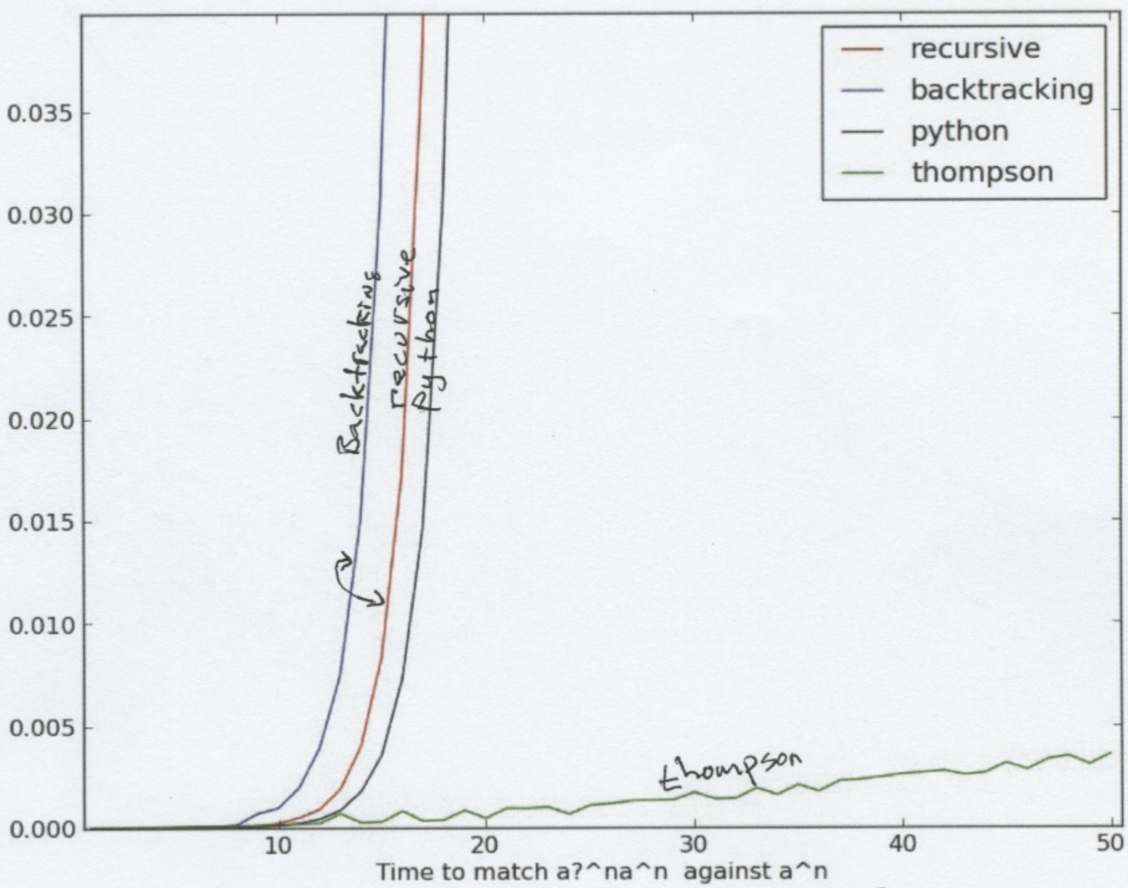
new

```
package machines

import . "inst"
import "queue"

func Thompson(program InstSlice, text []byte) bool {
    var cqueue, nqueue *queue.Queue = queue.New(), queue.New()
    cqueue.Push(0)
    for tc := 0; tc <= len(text); tc++ {
        for !cqueue.Empty() {
            pc := cqueue.Pop()
            inst := program[pc]
            switch inst.Op {
                case CHAR:
                    if int(tc) >= len(text) || text[tc] != byte(inst.X) { break }
                    nqueue.Push(pc+1)
                case MATCH:
                    if tc == len(text) { return true }
                case JMP:
                    cqueue.Push(inst.X)
                case SPLIT:
                    cqueue.Push(inst.X)
                    cqueue.Push(inst.Y)
            }
        }
        cqueue, nqueue = nqueue, cqueue
    }
    return false
}
```

swaps current, new



$a?^n a^n$ $n=3$

$\underbrace{a?a?a?aaa}$

aaa

$(a^*)?(a^*)?\ldots$

```go
package main

import "fmt"
import "os"
import "math"
import "inst"
import . "machines"

func test_case(n int) (inst.InstSlice, []byte) {
    program := make(inst.InstSlice, n*3+1)
    text := make([]byte, n)
    i := uint32(0)
    for j := 0; j < n; j++ {
        program[i]   = inst.New(inst.SPLIT, i+1, i+2)
        program[i+1] = inst.New(inst.CHAR, 'a', 0)
        i += 2
    }
    for j := 0; j < n; j++ {
        text[j] = 'a'
        program[i] = inst.New(inst.CHAR, 'a', 0)
        i++
    }
    program[i] = inst.New(inst.MATCH, 0, 0)
    return program, text
}

func time() float64 {
    sec, nsec, _ := os.Time()
    return float64(sec) + float64(nsec)*math.Pow(10.0, -9)
}

func main() {
//      fmt.Println("test, recursive, backtracking, thompson")
    for i := 1; i <= 50; i++ {
        program, text := test_case(i)
        var t1, t2, t3 float64
        if i <= 20 {
            {
                s := time()
                Recursive(program, text)
                e := time()
                t1 = e-s
            }
            {
                s := time()
                Backtracking(program, text)
                e := time()
                t2 = e-s
            }
        } else {
            t1 = 0.0
            t2 = 0.0
        }
        {
```

a?a?aa
0 split 1,2
1 char a
2 split 3,4
3 char a
4 char a
5 char a
6 match

17

```
            s := time()
            Thompson(program, text)
            e := time()
            t3 = e-s
        }                                       R B A
        fmt.Printf("%v, %f, %f, %f\n", i, t1, t2, t3)
    }
}
```

```
Page 1:
    What are Regular Expressions?
        - They answer yes or no questions about a piece of text.
        - Does the text match a pattern

Page 4:
    What is an Automaton?
        - A self operating machine, or program
    What is an NFA?
        - Non-Deterministic Finite Automaton
        - A machine whose operation may or may not perform the same way given the same
        input
            ie. non-deterministic
    Why NFAs?
        - Kleene's Theorem:
            A language is regular if and only if it is recongnized by a finite state
            automaton
        - NFA's are equivalent to DFA's
        - It is simipler to construct a NFA

Page 5:
    Turing Machine say wah?
        - A more powerful kind of automaton than finite automaton
        - details are unimportant
        - finite automaton can be executed on on a turing machine
        - the Church-Turing result tells us:
            - any problem solvable with a effective algorithm can be run on a turing
            machine
        - this is the basis for runing mac programs on windows and vice versa
        - it also means we can use a approach similer to running windows programs on a mac
            to implement regex
    - Step 1
        - Define a machine language
    - Step 2
        - Define execution semantics for the language
    - Step 3
        - Define conversion from NFA to the machine language
    - Step 4
        - Develop an algorithm or machine to execute the machine language
```