Writing an Interpreter in Go

John Scott

Unix/Database Consultant
Founder, SetSpace, Inc, 1998-now
Founder, august.{com,net}, Inc, 1998-2006

Capitol One Conference Center
Plano, Texas, USA
April 5, 2016

Paul Graham - YCombinator

```
Your mind is like a compiled program you've lost the source of.
It works, but you don't know why.
```

What is a Compiler?

- program that transforms a formal language into another formal language
- immutable - same input language always yields same output language
- output language - called the *target* - executes "fastest"

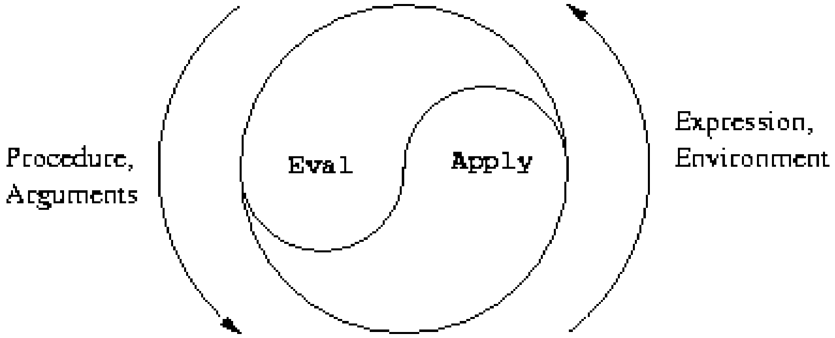
Does a compilier define semantics of either source or target language?

Examples of Typically Compiled Languages

- go
- c
- java
- perl5 is **NOT** compiled
- **perl6** will eventually be compiled
- YACC compiles Backus-Naur into Go (and many other languages)
- Ken Thompson (gofather) compilied regular expressions into pdp11 assembler
- compiling SQL into native code (Oracle, PostgreSQL)

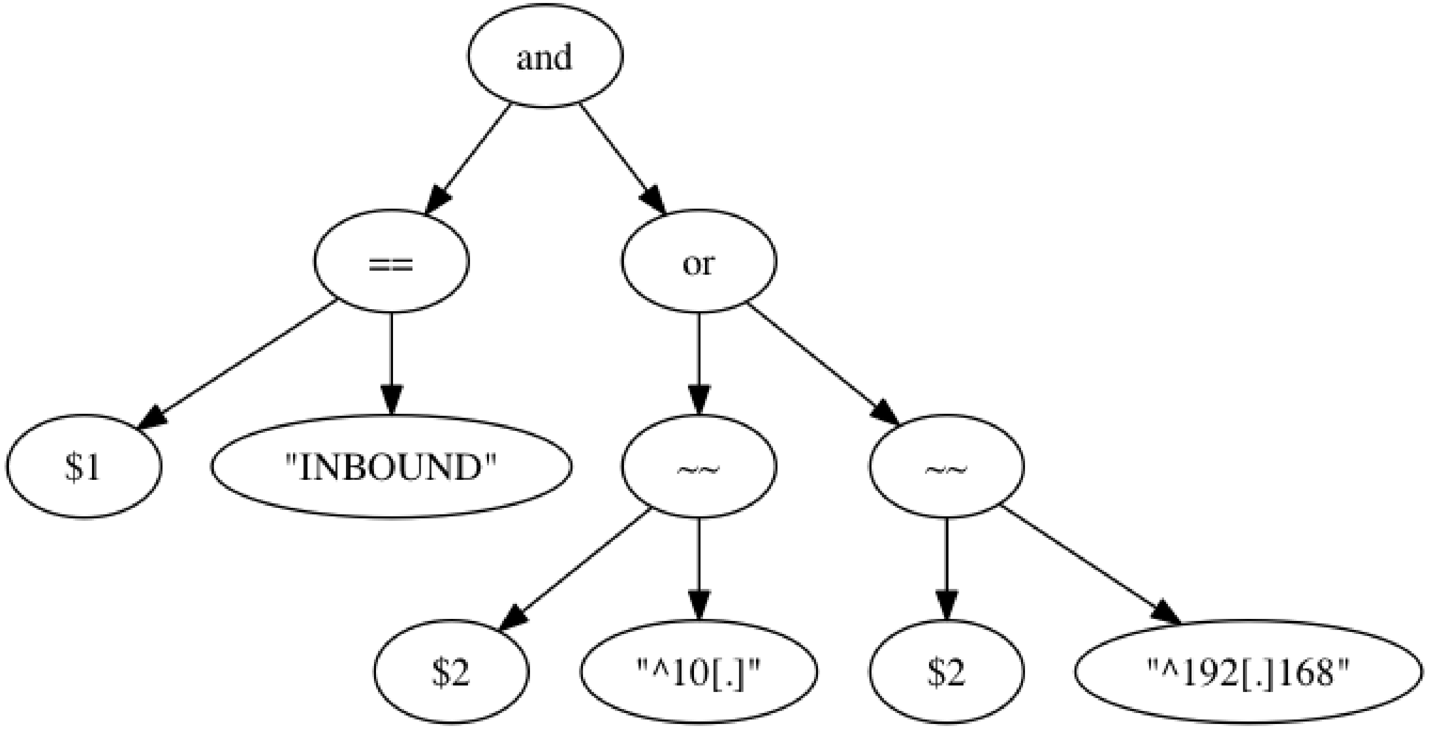
What is an Interpreter?

- program that directly executes the source program (often text)



- bourne shell, awk, javascript, ruby, SQL
- lisp was the first interpreter
- the java command is an interpreter (strictly) of compiled java class files

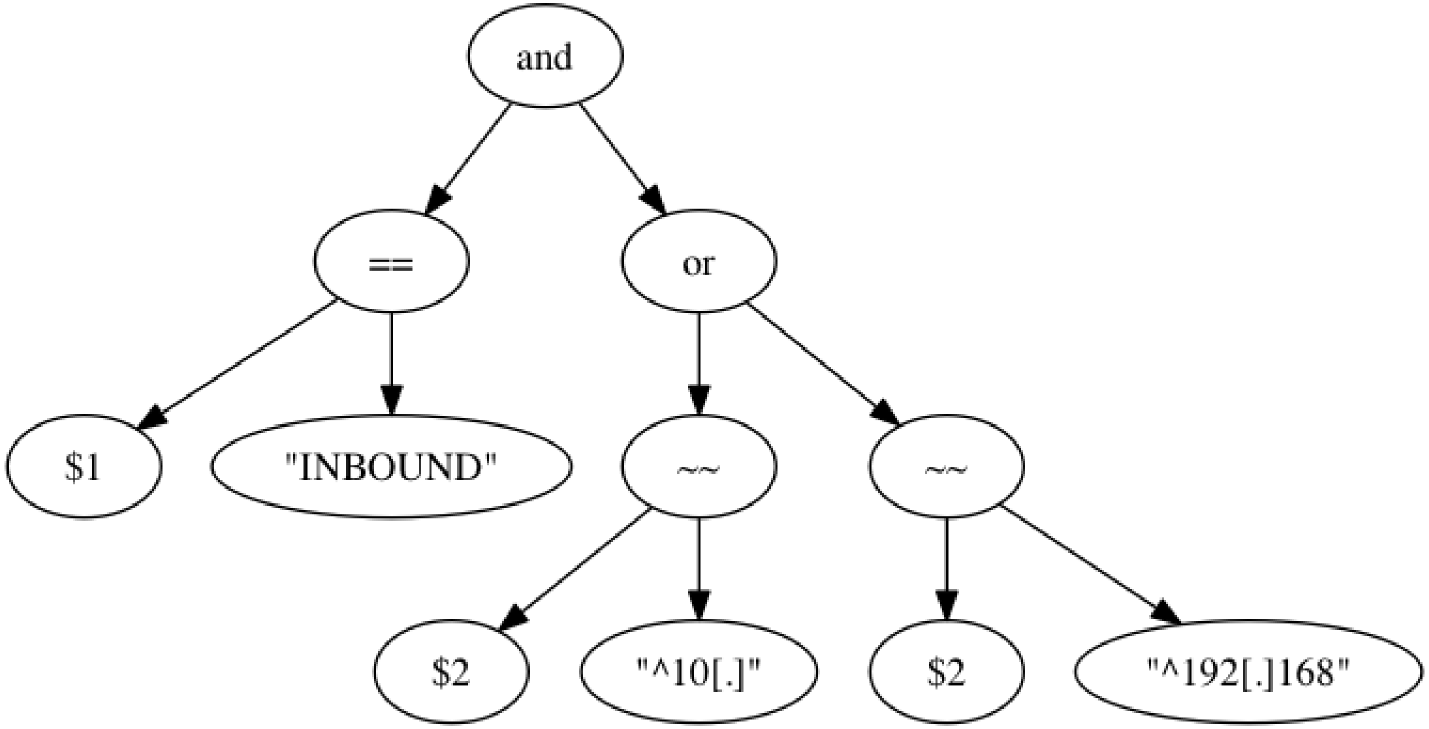
Interpreter Translates Source into Internal Format



Abstract Syntax Tree (perl5, awk)

- line by line (many unix shells)
- virtual machine instruction (python, ruby, vbasic, , perl6)
- java translates jvm opcodes into native hardware instructions

Abstract Syntax Tree of a Query Fragment in HOQ



Query Fragment

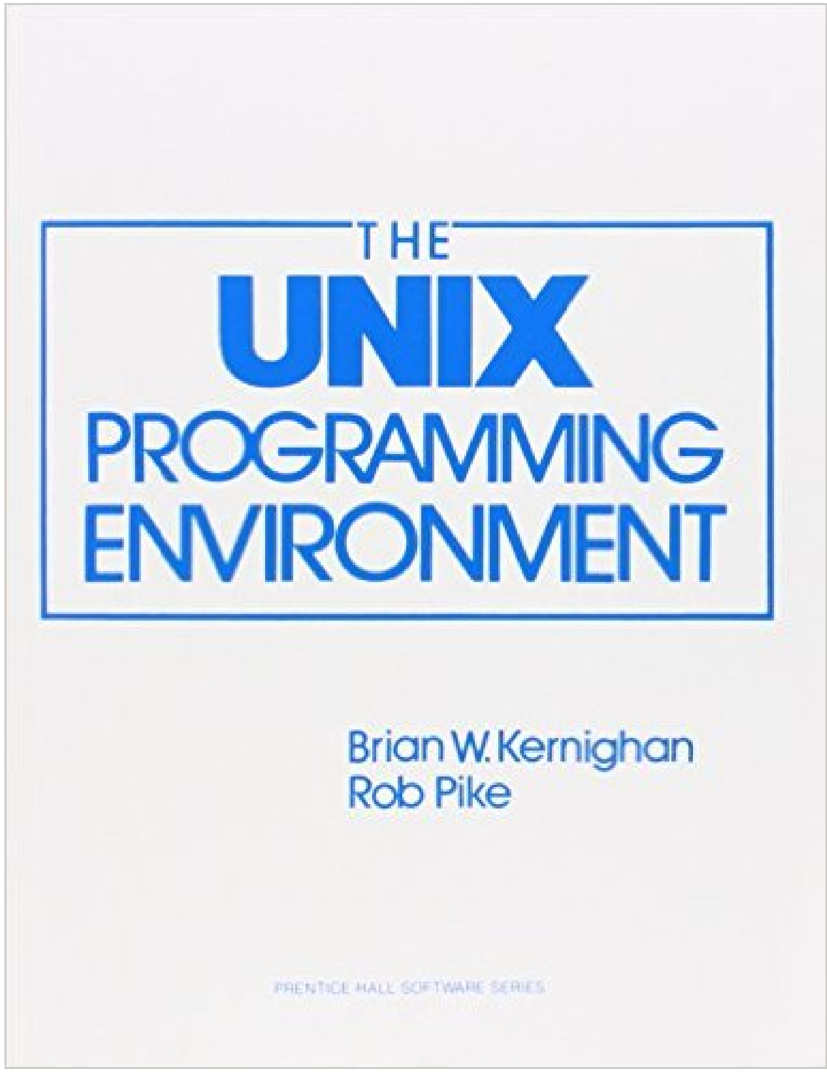
```
$1 == "INBOUND" and ( $2 ~~ "^10[.]" or $2 ~~ "^192[.]168" )
```

What is HOQ?

- Higher Order Query
- toy interpreter that demonstrates the famous YACC compiler for Go language
- also, hopefully, demonstrates "Communicating Sequential Processes"

Did I bite off more than I can chew?

Inspired by "HOC" Calculator in Unix Programming Environment



Higher Order Calculator - Chapter 8

Mighty Fine Book

How Do We Invoke HOQ?

- command line program invoked with script as first argument

```
hoq file.hoq < data.in
```

- hoq script *may* invoke other unix programs and wait for their exit status

HOQ Execution is Driven by Standard Input

- consuming text drives the execution of the whole hoq script

```
tr ',' '\t' quarterly.csv | hoq quarterly.hoq
```

Terminal

- hoq terminates after reading the the final line and waiting for all processes to exit.

HOQ Splits Input Into Fields

- input lines `strings.Split()` on tab separated boundaries

```
line = strings.TrimRight(line, "\n")
...
fields:   strings.SplitN(line, "\t", 255),
```

- $1 is first tab separated field, $2 is second field, ...
- $0 is whole, current line, including tabs, minus terminating new line

HOQ is Mostly Declarative

- qualify on patterns in tab separated field ($1, $2)
- qualify on process exit status codes (`uint8`)
- boolean combinations (logical and, or, not) on any qualifications
- subprocess are executed when boolean qualifications are true

Execution order of subprocess is directed acylic graph (DAG)

Hello, World

```
command say
{
    path = "echo";
}

exec say("hello, world");
```

say.hoq

Invoke Say

```
command say
{
    path = "echo";
}

exec say("hello, world");
```

say.hoq

```
$ echo | hoq say.hoq
hello, world
```

Terminal

Good Bye, Cruel World

```
command say1 {
    path = "echo";
}
command say2 {
    path = "echo";
}

exec say2("good bye, cruel world");
exec say1("hello, world");
```

say-bye.hoq

```
$ echo | hoq say-bye.hoq
hello, world
good bye, cruel world

$ echo | hoq say-bye.hoq
good bye, cruel world
hello, world
```

Trinity

```
command say1 {
    path = "echo";
}
command say2 {
    path = "echo";
}
command say3 {
    path = "echo";
}

exec say1("hello, world");         #  always executes

exec say2("to be or not to be")    when say1.exit_status == 0;
exec say3("good bye, cruel world") when say2.exit_status == 0;
```

say-trinity.hoq

```
$ echo | hoq say-trinity.hoq
hello, world
to be or not to be
good bye, cruel world
```

A Complex Qualification

```
command blob_on_network {
        path = "true";                                     #  path to /bin/true
}
command merge_blob      {
        path = "merge-blob";                               #  path to executable program
}

exec blob_on_network()
  when ((                                                  #  $3 is a blob request action
           $3 == "put" or $3 == "get" or
           $3 == "eat" or $3 == "wrap" or
           $3 == "roll"
    )
        and $5 == "ok"                                     #  $5 is request status
  ) or (
        $3 == "give" and $5 == "ok,ok"
  );

exec merge_blob($1, $2)            #  merge blob request into database
  when
        blob_on_network.exit_status == 0;
```

Edited from https://github.com/jmscott/setspace/blob/master/schema/setspace/setspace.flow.example

Is Wu Wei Nothing?

Idiom #1 - Turn Any Function into a Channel

```
package main
import "fmt"

func fib(i uint64) uint64 {

    if i < 2 {
        return 1
    }

    return fib(i - 1) + fib(i - 2)
}

func main() {

    fmt.Println(fib(24))

}
```

Run

Fibonacci is a Sequential Function

Fibonacci Becomes a Channel

```
func fib_chan(in chan int) (out chan int) {

    out = make(chan int)

    go func() {
        defer close(out)

        for i := range in {
            out <- fib(i)
        }
    }()

    return out
}
```

Idiom #1 - Turn Any Function into a Channel

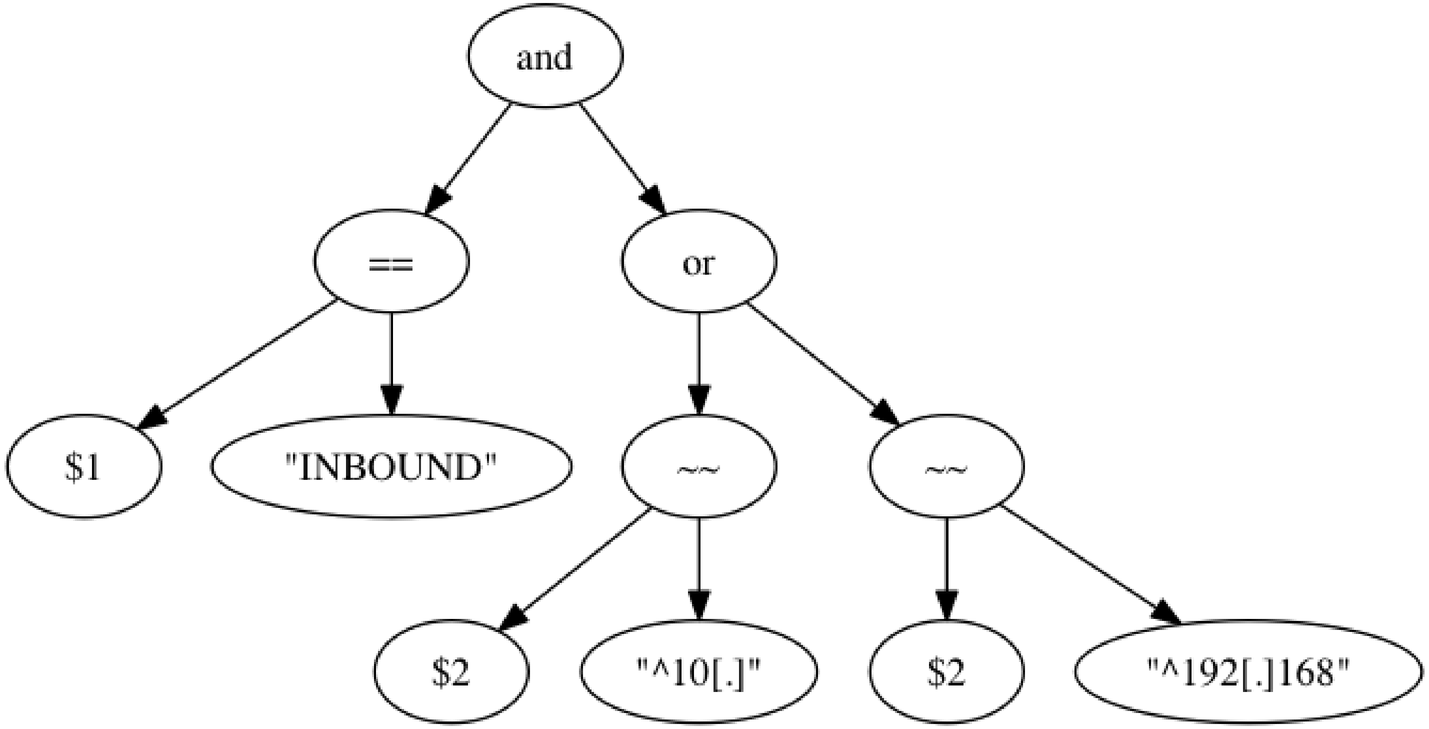
Idiom #1 - Run the Fibonacci Series

```
func main() {
    in := make(chan int)

    out := fib_chan(in)

    i := 1;
    for {
        in <- i
        fmt.Println(i, "=", <-out)
        i++
    }
}
```

Run

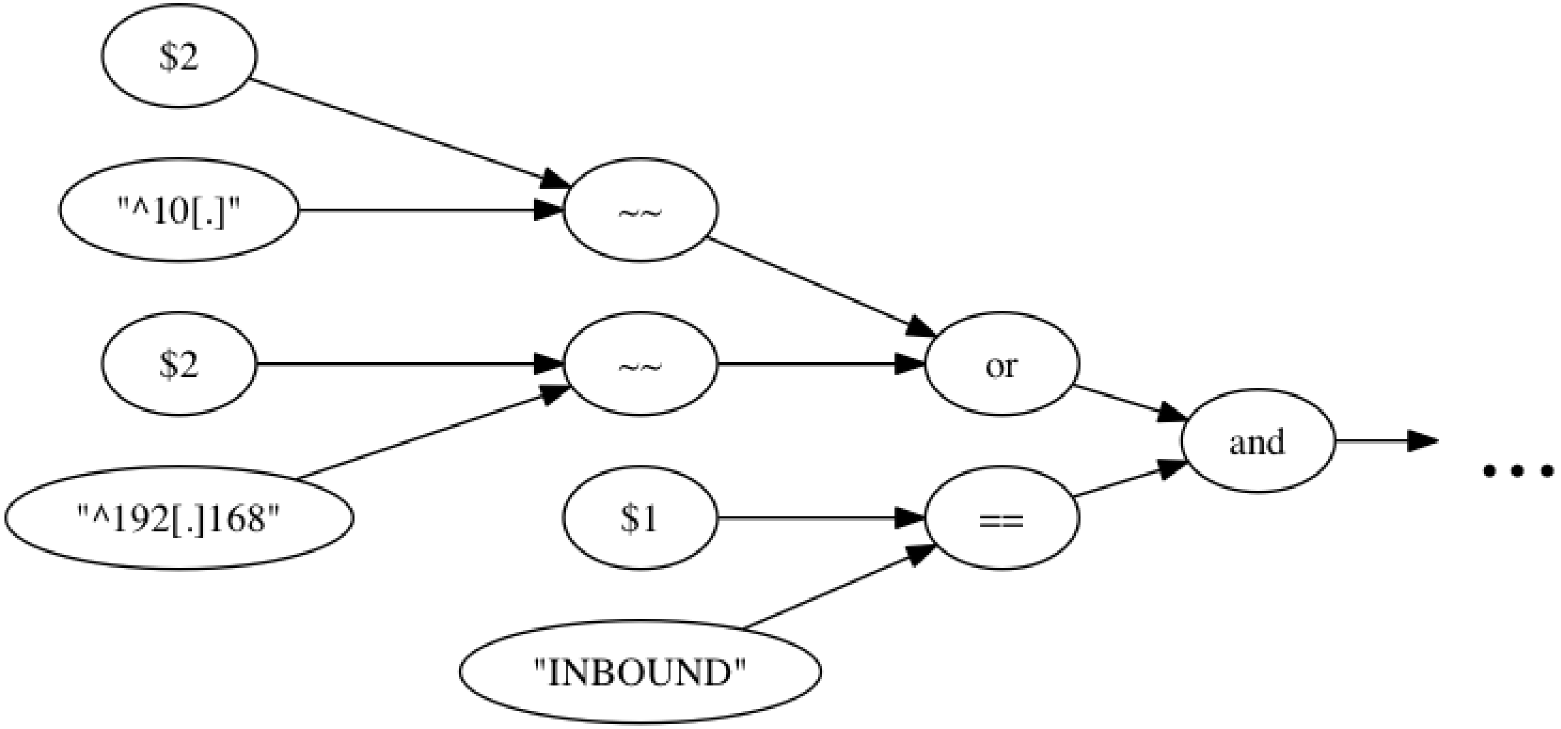
Idiom #1 - Turn Any Function into a Channel

Abstract Syntax Tree is a Flow Graph



```
$1 == "INBOUND" and ( $2 ~~ "^10[.]" or $2 ~~ "^192[.]168[.]" )
```

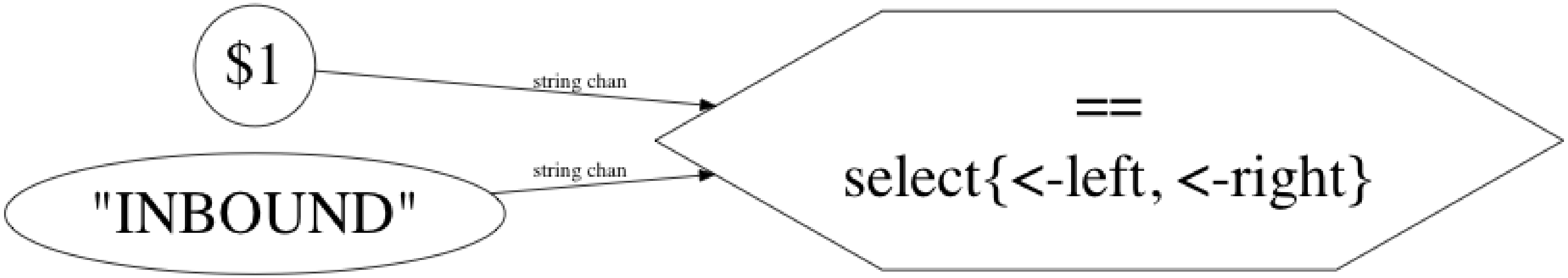
Data Flows From Leaves to Root



Each Edge is a Go Channel

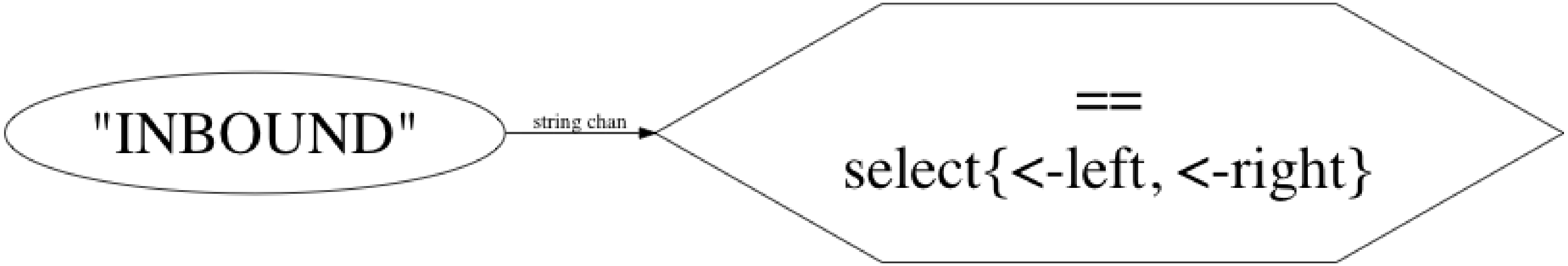
```
$1 == "INBOUND" and ( $2 ~~ "^10[.]" or $2 ~~ "^192[.]168[.]" )
```

AST Query Fragment: Data Flows Towards String Equality Operand



```
$1 == "INBOUND"
```

AST Query Fragment: Send String Constant to Equality Operand



Similar to Push in Sequential Machine

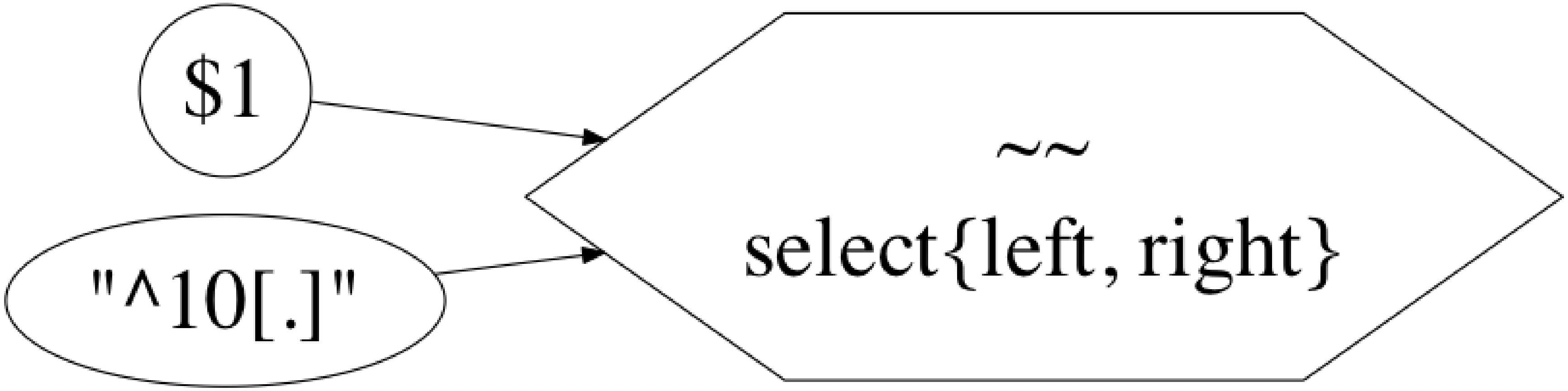
Opcode: const_string()

```
type string_value struct {
    string
    is_null bool
}
type string_chan chan *string_value
```

```
func (flo *flow) const_string(s string) (out string_chan) {

    out = make(string_chan)

    go func() {
        defer close(out)

        for flo = flo.get(); flo != nil; flo = flo.get() {
            out <- &string_value{                       //  send string upstream to next opcode
                string: s,
            }
        }
    }()

    return out
}
```

'Push' a String Constant

Idiom #2 - select{} on Two Channels Allows Concurrent Operands



```
$1 ~~ "^10[.]"
```

The regex.Match() is a Binary Relation on Two Strings

```
import "regexp"

func re_match(sample, re string) bool {

    matched, err := regexp.MatchString(re, sample)
    if err != nil {
        panic(err)
    }

    return matched
}
```

import "regexp"

```
type bool_value struct {
    bool
    is_null bool
}
type bool_chan chan *bool_value
```

Idiom #2 - select{} on Two Channels Allows Concurrent Operands

- OpCode: Relational Binary Operator on Two Strings

```
func (flo *flow) string_rel2(
    rel2 func(left, right string) bool,
    in_left,
    in_right string_chan,
) (out bool_chan) {

    out = make(bool_chan)
    go func() {
        defer close(out)
```

Upper Half of string_rel2()

... Main Loop in Next Slide ... ⇒

```
    }()
    return out
}
```

Lower Half of string_rel2()

Idiom #2 - select{} on Two Channels Allows Concurrent Operands

Relational Binary Operator on Two Strings - Main Loop (Idiom #2)

```
        for flo = flo.get(); flo != nil; flo = flo.get() {
            var left, right *string_value

            for left == nil || right == nil {
                select {
                case lv := <-in_left:
                    if lv == nil {
                        return
                    }
                    left = lv
                case rv := <-in_right:
                    if rv == nil {
                        return
                    }
                    right = rv
                }
            }
            bv := &bool_value {is_null: left.is_null || right.is_null}
            if bv.is_null == false {
                bv.bool = rel2(left.string, right.string)
            }
            out <- bv        //  send channel to next opcode
        }
```

16 Opcodes of Flow Machine

- const_string, const_bool, const_uint8
- to_string_uint8, to_string_bool
- dollar, dollar0
- string_rel2, uint8_rel2, bool_rel2
- argv0, argv1, argv
- exec
- fanout_uint8, fanin_uint8

Exec() of Command Sends Exit Status When Qualification is True

```
command xtrue {
    path = "true";
}
command say1 ("hello from say1:") {
    path = "echo";
}
command say2 ("hello from say2:") {
    path = "echo";
}

exec xtrue();      #  always called

exec say1("xtrue exited 0")
  when
      xtrue.exit_status == 0
;
exec say2("xtrue exited 1")
  when
      xtrue.exit_status == 1;
```

```
$ echo | hoq exec-xtrue.hoq
hello from say1: xtrue exited 0
```

Exec() of Command Sends Null When Qualification is False or Null

```
command xtrue {
    path = "true";
}
command say1 ("hello from say1:") {
    path = "echo";
}
command say2 ("hello from say2:") {
    path = "echo";
}

 #  xtrue never called, so exit_status sends null bool upstream to all qualifications
exec xtrue() when false;

exec say1("xtrue exited 0")
  when
      xtrue.exit_status == 0
;
exec say2("xtrue exited 1")
  when
      xtrue.exit_status == 1
;
```

```
$ echo | hoq exec-false.hoq
```

Logical Boolean Operators Follow Strict SQL Semantics

Logical AND

```
false and *  =>   false
* and false  =>   false
null and *   =>   null
* and null   =>   null
*            =>   true
```

Logical OR

```
true or *    =>   true
* or true    =>   true
null or *    =>   null
* or null    =>   null
*            =>   false
```

All other binary operators are null if either operand is null

Idiom #3 - Channels Over Channels

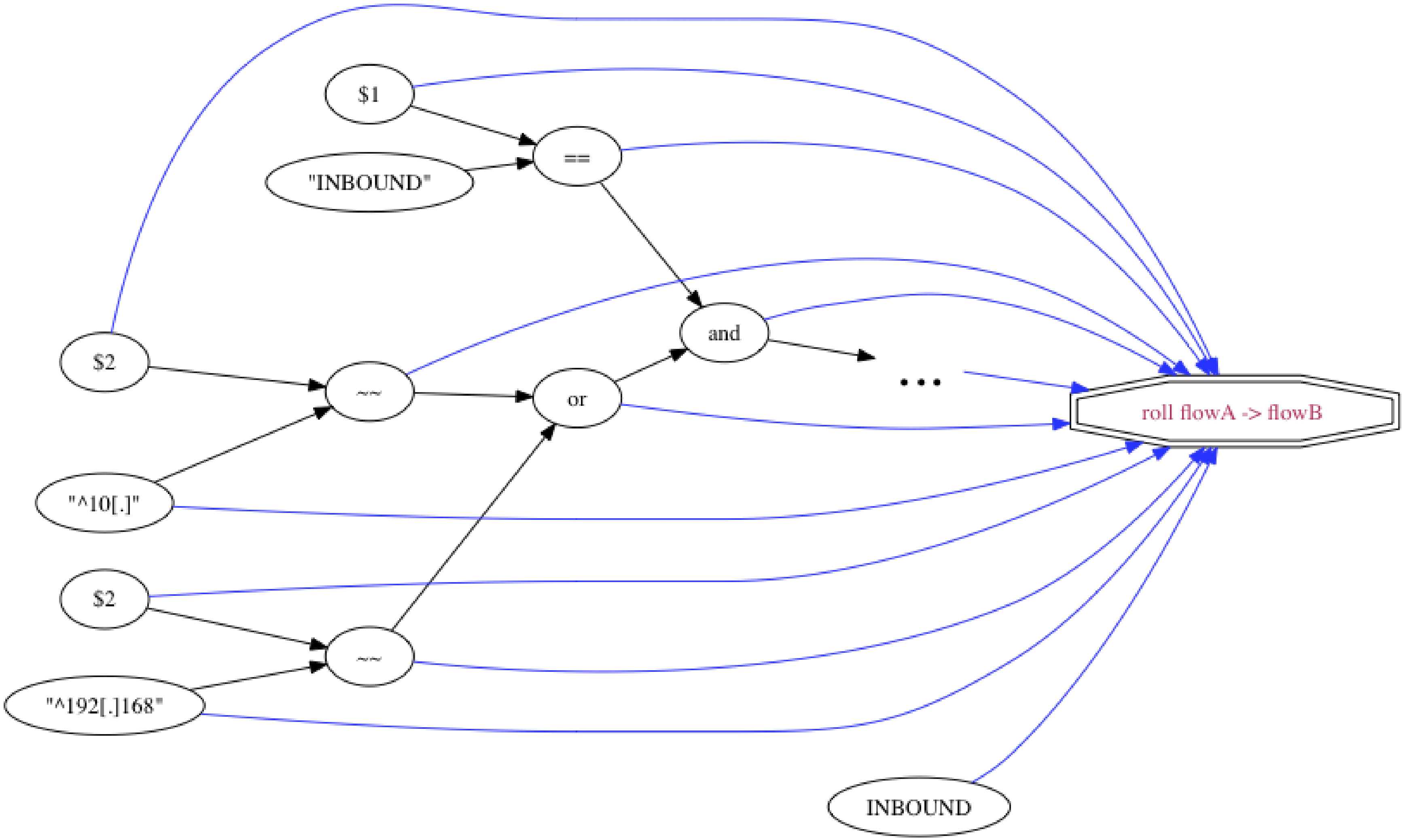
```
//  each opcode requests another flow

func (flo *flow) get() *flow {

    //  wait for entire query to resolve

    <-flo.resolved

    //  next active flow arrives on this channel

    reply := make(flow_chan)

    //  request another flow by sending reply channel to main()

    flo.next <- reply

    //  return next flow to the this opcode

    return <-reply
}
```

Flow Structure Synchronizes Operators

```
//  a flow tracks the firing of rules over a single line of input text.

type flow struct {

    //  request a new flow from this channel, reading reply on sent side-channel
    next chan flow_chan

    //  channel is closed when all exec()s make no further progress
    resolved chan struct{}

    //  the whole line of input with trailing new line removed
    line string

    //  tab separated fields split out from the line read from standard input
    fields []string

    //  count of go routines/operators still resolving qualifications
    confluent_count int
}
type flow_chan chan *flow
```

Idiom #3 - Channels Over Channels

Each Opcode Syncs with main() on Each "Tick"



$1 == "INBOUND" and ($2 ~~ "^10[.]" or $2 ~~ "^192[.]168[.]")

Idiom #4 - Close(channel) is a Cheap Broadcast

```
//  set up first flow to compile all ast nodes into a single flow graph.
//  each flow terminates by sending a count of the fired unix processes.

flowA := &flow{
    next:     make(chan flow_chan),
    resolved: make(chan struct{}),
}
uc := flowA.compile(ast, depend_order)          //  compile using gnu tsort
close(flowA.resolved)                           //  flowA never runs

//  start pumping standard input to the flow graph of nodes

in := bufio.NewReader(os.Stdin)
for {
    line, err := in.ReadString('\n')
    ...
```

... Main Loop in Next Slide ... ⟹

```
}
```

Flow A and Flow B Run Concurrently

```
    line = strings.TrimRight(line, "\n")        //  trim and split the input line of text
    flowB := &flow{
        line:     line,
        fields:   strings.SplitN(line, "\t", 255),
        next:     make(chan flow_chan),
        resolved: make(chan struct{}),
    }
    for flowA.confluent_count > 0 {            //  push flowA to flowB
        reply := <-flowA.next
        flowA.confluent_count--
        reply <- flowB
        flowB.confluent_count++
    }
    if <-uc == nil {                           //  wait for flowB to finish
        break
    }
    close(flowB.resolved)                     //  broadcast to all nodes in flowB the flow is don
    flowA = flowB                             //  and so the wheel turns
```

Idiom #4 - Close(channel) is a Cheap Broadcast

What is Go YACC?

- Compiles a Flavor of Backus-Naur Form (BNF) into Go Code
- Original Golang Grammar written in YACC
- Domain Specific Languages (chemical reactions, SAT solvers)
- Mutation Coverage

Mechanics of YACC

- YACC generated Go code reads a stream of integers called <TOKEN>s
- Patterns are recognized in stream of <TOKEN>s
- Each pattern has an associated block of manually written Go code
- When the pattern is recognized in the stream then associated block of Go is invoked
- Patterns are recusive

Backus-Naur Grammar of Backus-Naur Form

```
alpha:           'a' | 'b' |  ... 'Z'
             ;
alphanum:        alpha | '0' | '1'  ... '9'
             ;
name:            alpha
               |
                 name  alphanum
             ;
term:            <TOKEN>
               |
                 name
               |
                 term  term
             ;
expression:      term
               |
                 expression  '|'  term
             ;
production:     name  ':'  expression  ';'
             ;
grammar:         production
               |
                 grammar  production
             ;
```

Whitespace/Comment not specified. <TOKEN> is scanned integers, 'a'=97

YACC Grammar for HOQ Language (Backus-Naur Form)

```
exp:
      TRUE    |    FALSE    |    STRING    |    UINT8    |    '$' UINT8                 |
      XCOMMAND '.' EXIT_STATUS                                                          |
      exp RE_MATCH exp     |     exp RE_NMATCH exp                                      |
      exp AND exp |  exp OR exp  |  exp EQ exp  |  exp NEQ exp                          |
      NOT exp     |     '(' exp ')'                                                     ;
exp_list:
      exp    |    exp_list ',' exp                                                      ;
argv:
      /*empty*/    |    exp_list                                                        ;
qualification:
      /*empty*/    |    WHEN  exp                                                       ;
string_list:
      STRING    |    string_list  ','  STRING                                           ;
command_argv:
      /*empty*/    |    '(' ')'    |    '(' string_list ')'                             ;
statement:
      COMMAND NAME command_argv '{' PATH '=' STRING ';' '}'                             |
      EXEC  XCOMMAND '('  argv  ')'  qualification  ';'                                 ;
statement_list:
      statement    |    statement_list statement                                        ;
```

Stripped from https://github.com/jmscott/play/blob/master/hoq/src/parser.y

YACC/Go Code Snippet of Qualification Expression in HOQ Grammar

```
  exp  AND  exp                  //  logical and of two qualifications
  {
        l := yylex.(*yyLexState)
        $$ = l.bool_node(AND, $1, $3)
        if $$ == nil {
                return 0
        }

        if $1.go_type != reflect.Bool {
                l.error("logical 'and' requires boolean operands")
                return 0
        }
  }
|
  '('  exp  ')'                  //  change precedence of qualification
  {
        $$ = $2
  }
```

In YACC $1 is value of first term; $2 is value of second term ...

$$ is value of new, reduced term

YACC Needs a Lexer

- lexer easy to write by hand !!!
- just a big switch{}
- busts text into stream of integer tokens
- scanned token has a simple value related to scan yylval.(string, float, uint8)
- token const defined by YACC (see %token in grammar)
- co-routines invented to write lexers (Rob Pike)

Go Files on github.com/jmscott

- hoq.go
- ast.go
- command.go
- compile.go
- opcode.go
- parser.y
- rummy.go
- tsort.go

github.com/jmscott/play/tree/master/hoq/src (https://github.com/jmscott/play/tree/master/hoq/src)

YACC Resources

Early GoLang in YACC (https://docs.google.com/document/d/1P3BLR31VA8cvLJLfMibSuTdwTuF7WWLux71CYD0eeD8/edit)

Book 'Unix Programming Environment" by Kernighan and Pike (http://www.amazon.com/Unix-Programming-Environment-Prentice-Hall-Software/dp/013937681X)

C Written in Yacc (http://heim.ifi.uio.no/inf2270/programmer/historien-om-C.pdf)

Simple Go Calculator (https://github.com/golang-samples/yacc)

Go AST Library (https://golang.org/pkg/go/ast/)

EBNF in GoLang (https://godoc.org/golang.org/x/exp/ebnf)

Lex Resources

Lexer Talk by Rob Pike (https://www.youtube.com/watch?v=HxaD_trXwRE)

Ragle is a Lexical Compiler (http://www.colm.net/open-source/ragel/)

Nex (New Lex) (http://www-cs-students.stanford.edu/~blynn/nex/)

CSP Resources

research.swtch.com/power (http://research.swtch.com/power)

Thank you

John Scott
jmscott@setspace.com (mailto:jmscott@setspace.com)
https://github.com/jmscott (https://github.com/jmscott)

Unix/Database Consultant
Founder, SetSpace, Inc, 1998-now
Founder, august.{com,net}, Inc, 1998-2006

Capitol One Conference Center
Plano, Texas, USA
April 5, 2016