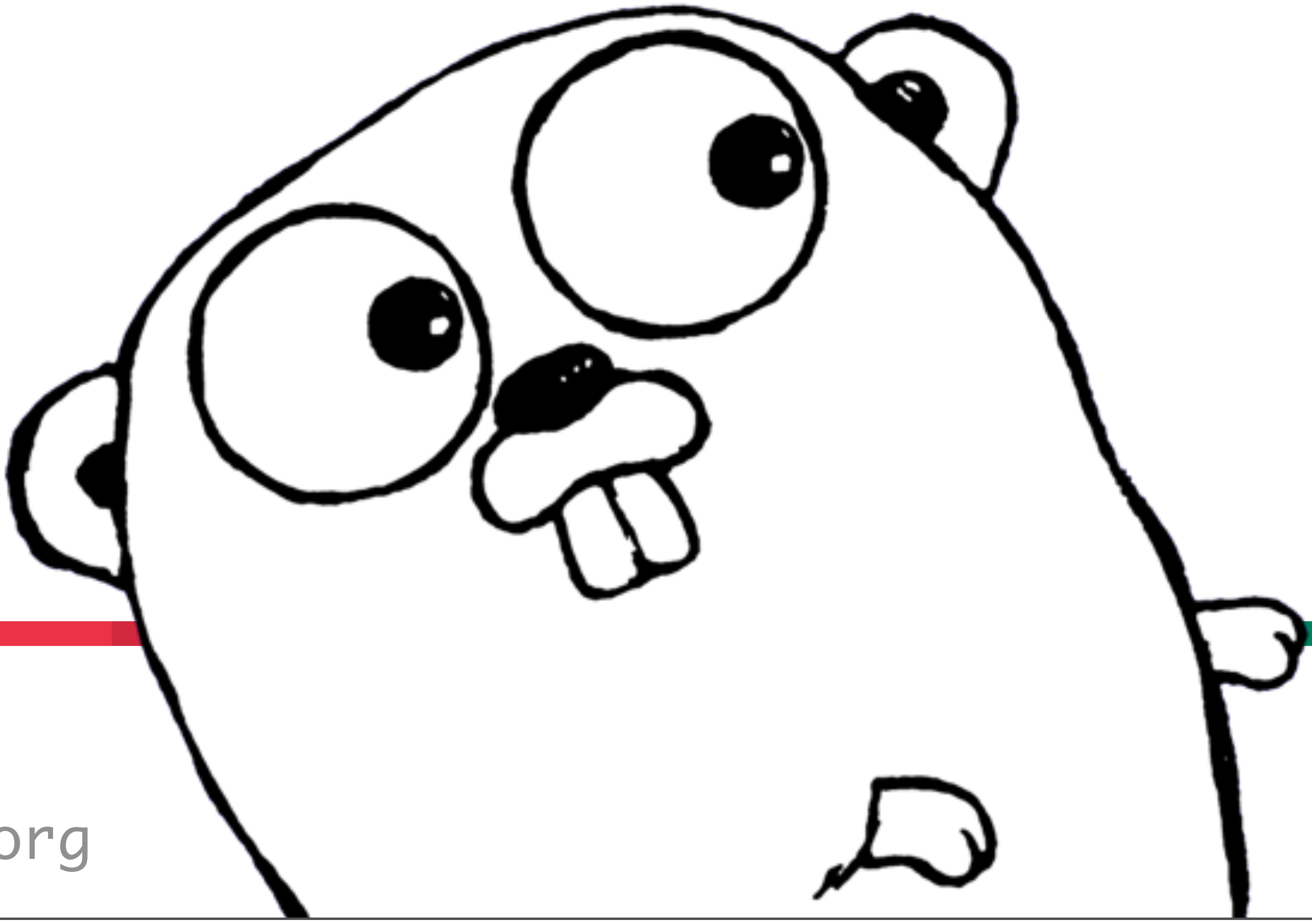
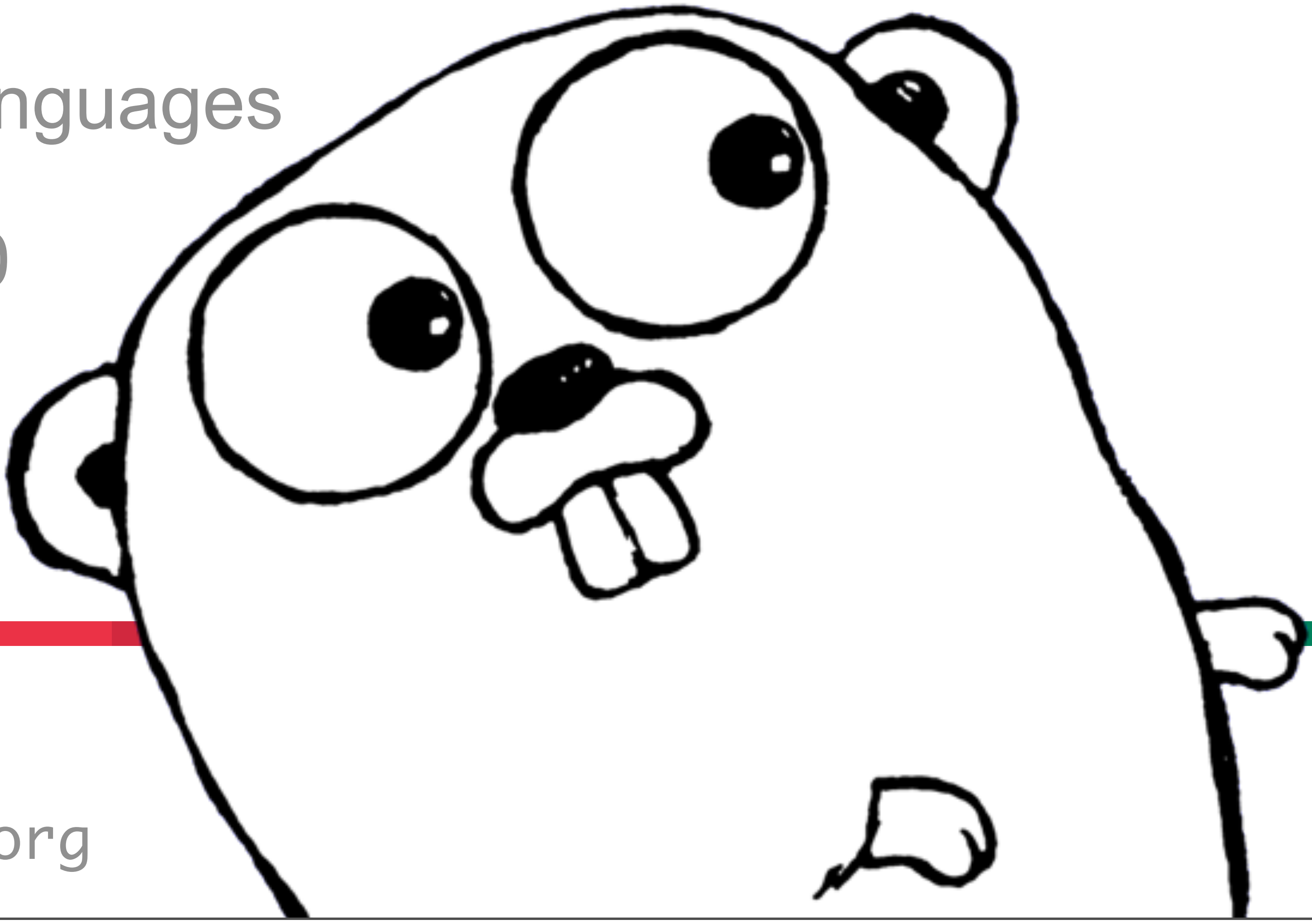


http://golang.org

Google

Go

Rob Pike
Emerging Languages
OSCON
July 21, 2010



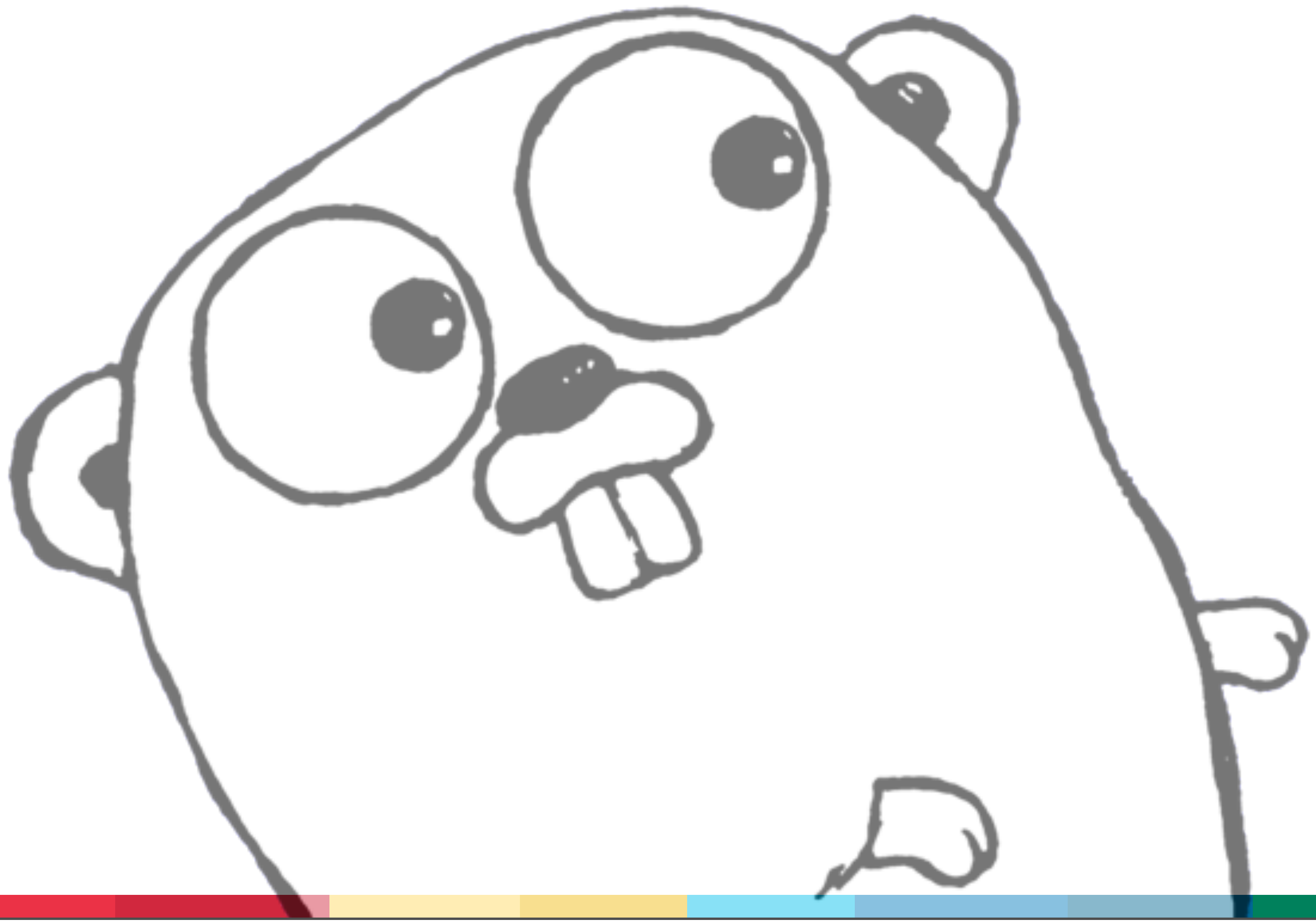
`http://golang.org`

Google

Concurrency

Where did Go's concurrency model come from?

It's got a long history.



Parallelism

In the late 1970s, multiprocessors were a research topic.

Programming multiprocessors seemed related to issues of operating systems research, interrupt handling, I/O systems, message passing, ...

Ideas in the air:
- semaphores (Dijkstra, 1965)
- monitors (Hoare, 1974)
- locks (mutexes)
- message passing

(Lauer & Needham 1979 showed that message-passing and what we now call threads&locks are equivalent.)

Communicating Sequential Processes (CSP)

Seminal paper by C.A.R. Hoare, CACM 1978.

A model programming language that promoted input, output as fundamental elements of computing.

"Parallel composition of communicating sequential processes."

Communication is I/O.
Parallel composition is multiprocessing.
That's all you need!

More ideas in this paper than in any other 10 good papers you are likely to find.

Communicating Sequential Processes (CSP)

Mathematical, concise, elegant.

Generalization of Dijkstra's guarded commands, with `!` for sending and `?` for receiving a message.

`p!value`	sends value to process `p`
`p?var`	receives value from `p`, stores in variable `var`
`[A;B]`	runs `A` followed by `B`
`[A||B]`	runs `A` in parallel with `B` (composition)
`*[A]`	runs `A` repeatedly
`[a → A [] b → B]`	guarded command (`if a then A elif b then B fi`, but in parallel)

Communication is synchronization.
Each command can succeed or fail.

Coroutines

```
COPY:: *[c: character; west?c → east!c]

DISASSEMBLE::
*[cardimage:(1..80)character; cardfile?cardimage →
    i:integer; i := 1;
    *[i≤80 → X!cardimage(i); i := i+1]
    X!space ]

ASSEMBLE:: lineimage(1..125)character;
i:integer; i := 1;
*[c:character; X?c →
    lineimage(i) := c;
    [ i≤24 → i := 1+1
    [] i=125 → lineprinter!lineimage; i := 1
]   ];
[ i=1 → skip;
[] i>1 → *[i≤125 → lineimage(i) := space; i := i+1];
         lineprinter!lineimage ]

[west::DISASSEMBLE||COPY||east::ASSEMBLE]  # pipe!
```

Ports and patterns

The "ports" used in communication are just single connections to predefined processes - the names are process names.

Can write a prime sieve for 1000 primes but not N primes; a matrix multiplier for 3x3 but not NxN, etc. (Arrays of processes do the bookkeeping.)

Pattern matching to analyze/unpack messages:

```
  [ c?(x, y) → A ]
```

More general conditions:

```
  [ i≥100; c?(x, y) → A ]
```

Cannot use send as a guard.

Recap

Parallel composition of independent processes

Communication synchronizes

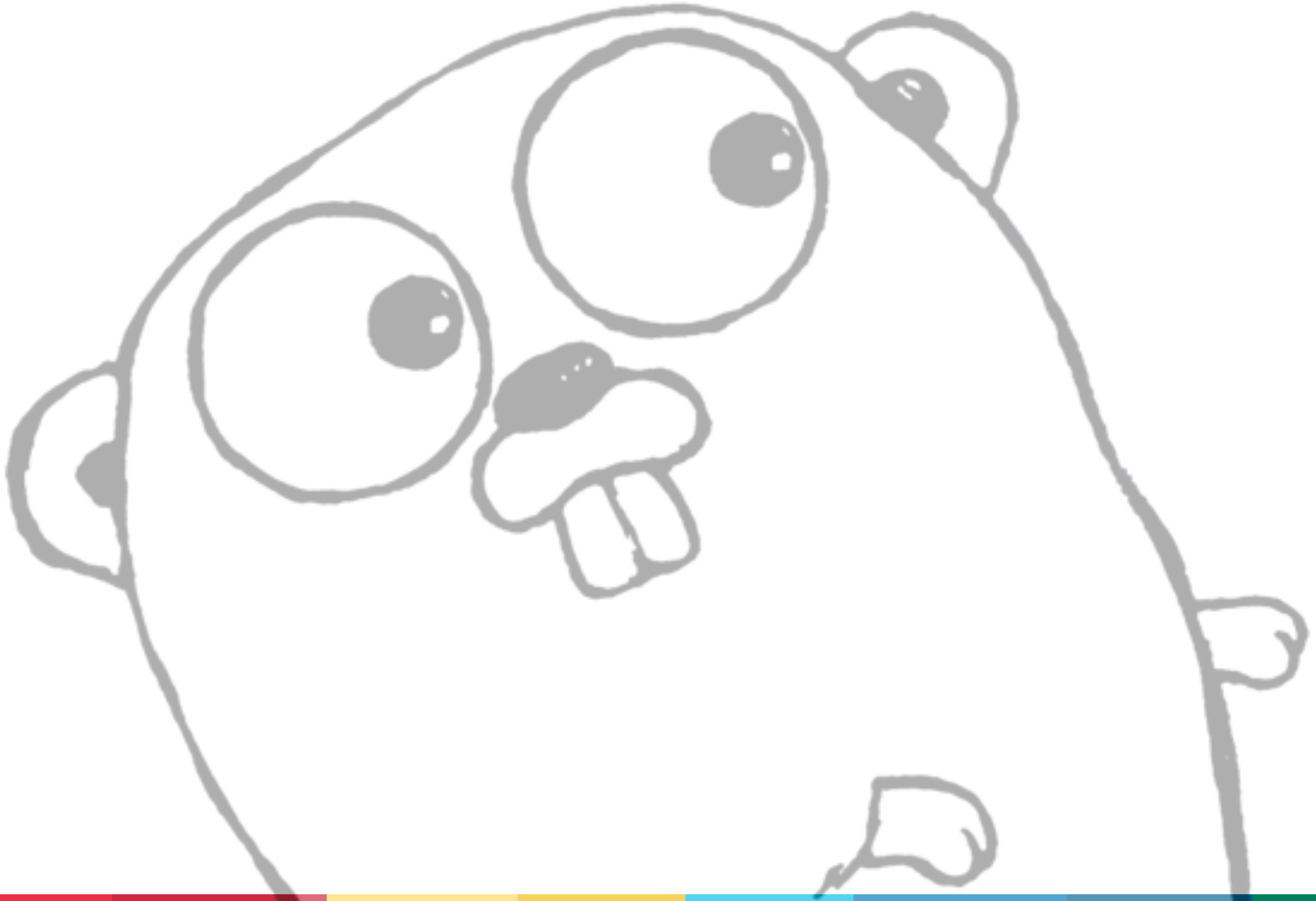
No sharing of memory

Not threads and Not mutexes!

Now we come to a fork in the road.

The Occam branch

Distinct sets of languages emerge from CSP.
One leads us to Occam, very close to basic CSP.

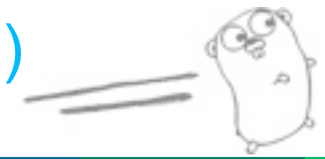


Thursday, July 22, 2010

Occam

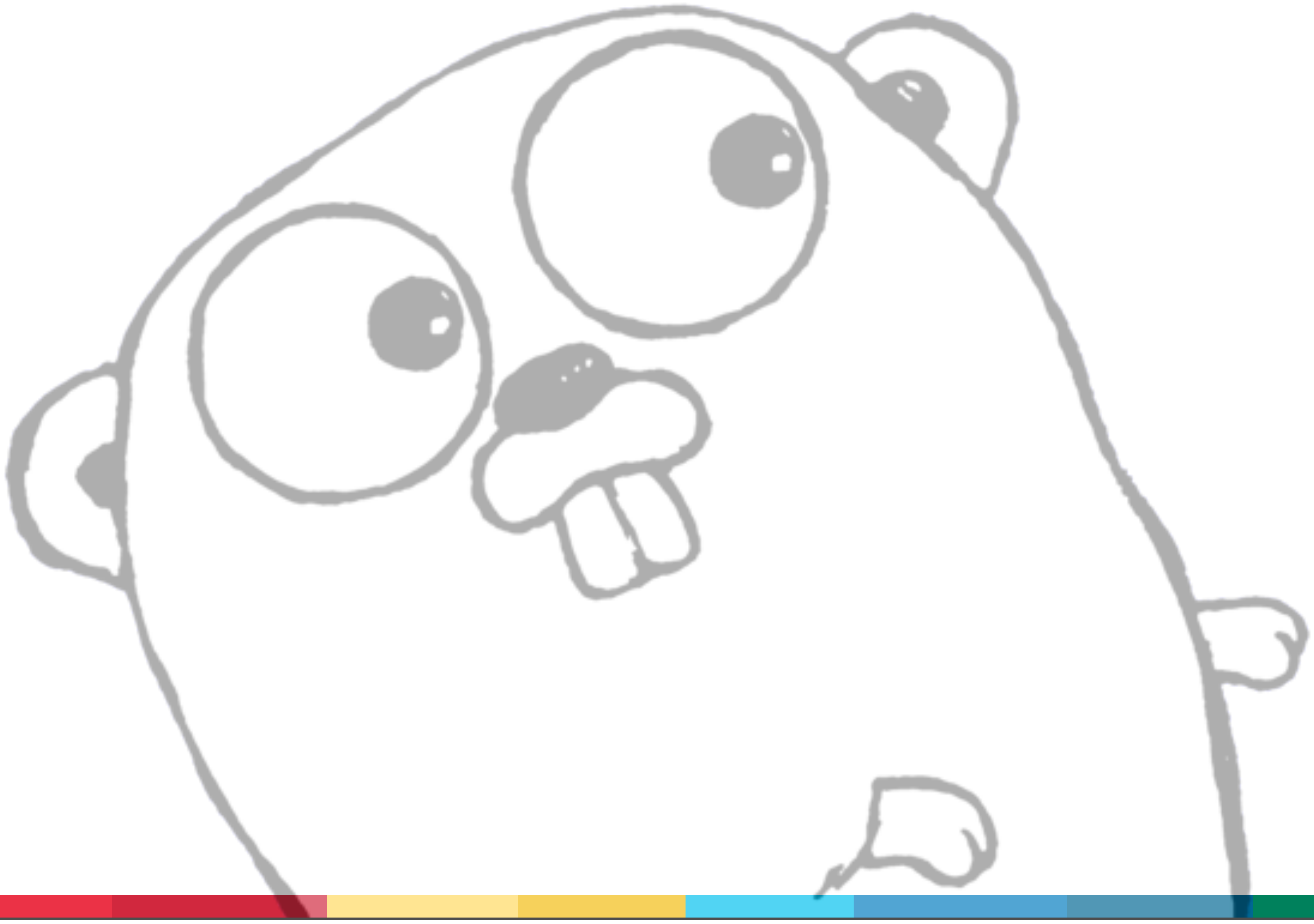
Inmos, a hardware company, designed the Transputer, and programmed it in Occam (1983).
Parallel architecture; nodes on the chip communicate with !?.
Ports correspond to hardware.
Language quite close to CSP (advised by Hoare).

```
ALT
   count1 < 100 & c1 ? data
     SEQ
       count1 := count1 + 1
       merged ! data
   count2 < 100 & c2 ? data
     SEQ
       count2 := count2 + 1
       merged ! data
   status ? request
     SEQ
       out ! count1
       out ! count2      -- (note white space for structure!)
```



The Erlang branch

Another leads us to Erlang: networked, pattern-matched.



Erlang

Developed at Ericsson (late 1980s). Took the functional side of CSP and used "mailboxes". Processes use pattern matching to unpack messages. Send is to a process ID.

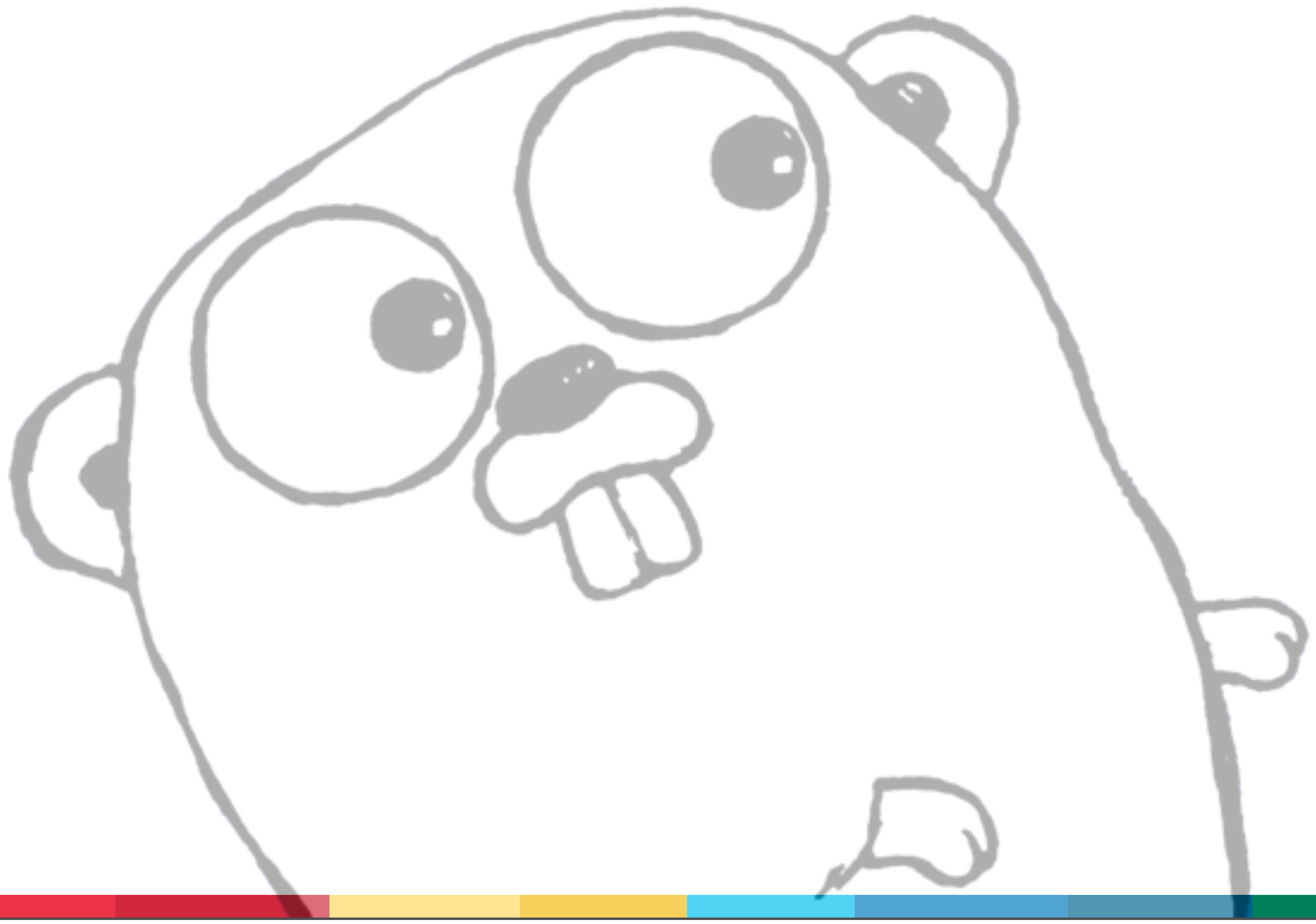
```
ServerProcess = spawn(web, start_server, [Port, MaxConnections]),

RemoteProcess = spawn(RemoteNode, web, start_server, [Port, MaxConnections]),

ServerProcess ! {pause, 10},
 receive
        a_message -> do_something;
        {data, DataContent} -> handle(DataContent);
        {hello, Text} -> io:format("Got hello message: ~s", [Text]);
        {goodbye, Text} -> io:format("Got goodbye message: ~s", [Text])
 end.
```

The Newsqueak/Limbo/Go branch

Another branch leads us to eventually to Go.
Here the focus is on channels.



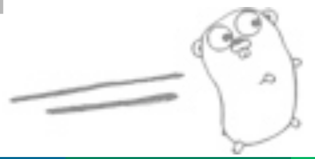
Squeak

Squeak* (Cardelli and Pike (1985)) was a toy language used to demonstrate the use of concurrency to manage the input streams to a user interface.

```
proc Mouse = DN? . M?p . moveTo!p . UP? . Mouse
proc Kbd(s) = K?c .
    if c==NewLine then typed!s . Kbd(emptyString)
    else Kbd(append(s, c))
    fi
proc Text(p) =
    < moveTo?p . Text(p)
    :: typed?s . {drawString(s, p)? . Text(p) >

type = Mouse & Kbd(emptyString) & Text(nullPt)
```

*unrelated to the much later Squeak Smalltalk implementation



Newsqueak

Newsqueak (1989) looked syntactically like C but was applicative and concurrent. Idea: a research language to make the concurrency ideas of Squeak practical.

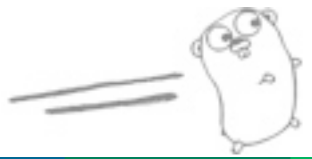
Had lambdas called `progs`, a `select` statement corresponding to the CSP alternation, but guards must be communication only (sends work).

Long-lived syntactic inventions:

Communication operator is left arrow `<-`. Information flows in direction of arrow. Also `<-c` (receive) is an expression.

Introduces := for "declare and initialize":

```
x: int = 1
x := 1
```



Prime sieve

```
counter := prog(c:chan of int) {
    i:int; for(i = 2;;) c<-=i++;
};
filter := prog(prime:int, listen,send:chan of int) {
    i:int; for(i=0 ;;) if((i=<-listen)%prime) send<-=i;
};
sieve := prog() of chan of int {
    c := mk(chan of int);
    begin counter(c);
    prime := mk(chan of int);
    begin prog(){
        p: int;
        newc: chan of int;
        for(;;){
            prime<- = p = <-c;
            newc = mk();
            begin filter(p, c, newc);
            c = newc;
        }
    }();
    become prime;
};
```

Channels as first-class values

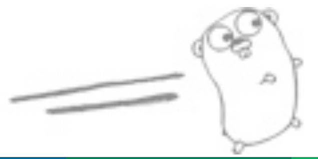
Unlike in the other languages, Newsqueak had the concept of channels as first-class values in a CSP-like model.

In Newsqueak and its descendants, can send a channel on a channel.

```
c: chan of int;
cc: chan of chan of int;

cc<- = c;  # Sends channel c on channel cc.
           # Recipient can then use c.
```

Makes multiplexers easy to construct. I can send you the channel to use to reply to me. It's a capability, like a file descriptor: "I grant you permission to communicate with me." (Erlang process IDs can grant ability to send but not receive; Newsqueak channels are fully symmetric.)

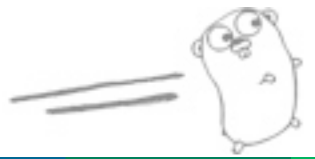


Alef

Early 1990s: Alef (Phil Winterbottom) grafted the concurrency and communications model of Newsqueak onto a more traditional compiled C-like language.

Problem: with C's memory model in a concurrent world, hard to know when to free items.

All the other languages in this talk are garbage-collected, which is essential to easy concurrent programming.



Limbo

Limbo (Dorward, Pike, Winterbottom 1996) was a VM language (contemporaneous with Java) that was closer to Newsqueak in overall design.

Used as an embedded language in communication products.

As in Newsqueak and Alef, the key idea is that channels are first-class.

Go

Go (Griesemer, Pike, Thompson 2009) is a compiled, object-oriented language with a concurrent runtime.

Makes it easy to use the tools of CSP efficiently, in concert with regular systems code. Channels are first class! (So are functions, which can run in parallel.)

Compilation makes execution efficient (e.g., cryptographic calculations are quick).

The runtime makes concurrency easy (stacks, communication, scheduling, etc. are all automatic).

Garbage-collected, naturally.

Best of all worlds!

A card reassembly example in Go

```
func copy(west, east chan byte) { for { east <- <-west } }

func assemble(X chan byte, printer chan []byte) {
    lineimage := make([]byte, 125)
    for i := 0;; {
        lineimage[i] = <-X
        if i < 124 { i++ } else { printer <- lineimage; i = 0 }
    }
}

func disassemble(cardfile chan []byte, X chan byte) {
    for {
        cardimage := <-cardfile
        i := 0
        for i < len(cardimage) { X <- cardimage[i]; i++ }
        for i < 80 { X <- ' '; i++ }
    }
}

go disassemble(cardreader, chars1)
go copy(chars1, chars2)
go assemble(chars2, lineprinter)
```

Summary

Go's concurrency structures have a long history dating back to a branch in the CSP family tree in the 1980s. Multiple real languages have built on CSP's ideas.

Channels as first-class values are the distinguishing feature of the Go branch.

Go pulls together elements from several predecessors, coupling high-level concurrency operations with a compiled object-oriented language.

To use concurrency gracefully, language must have garbage collection and automatic stack management.

Go

There's much more to Go than concurrency!

Two more OSCON talks tomorrow about it.

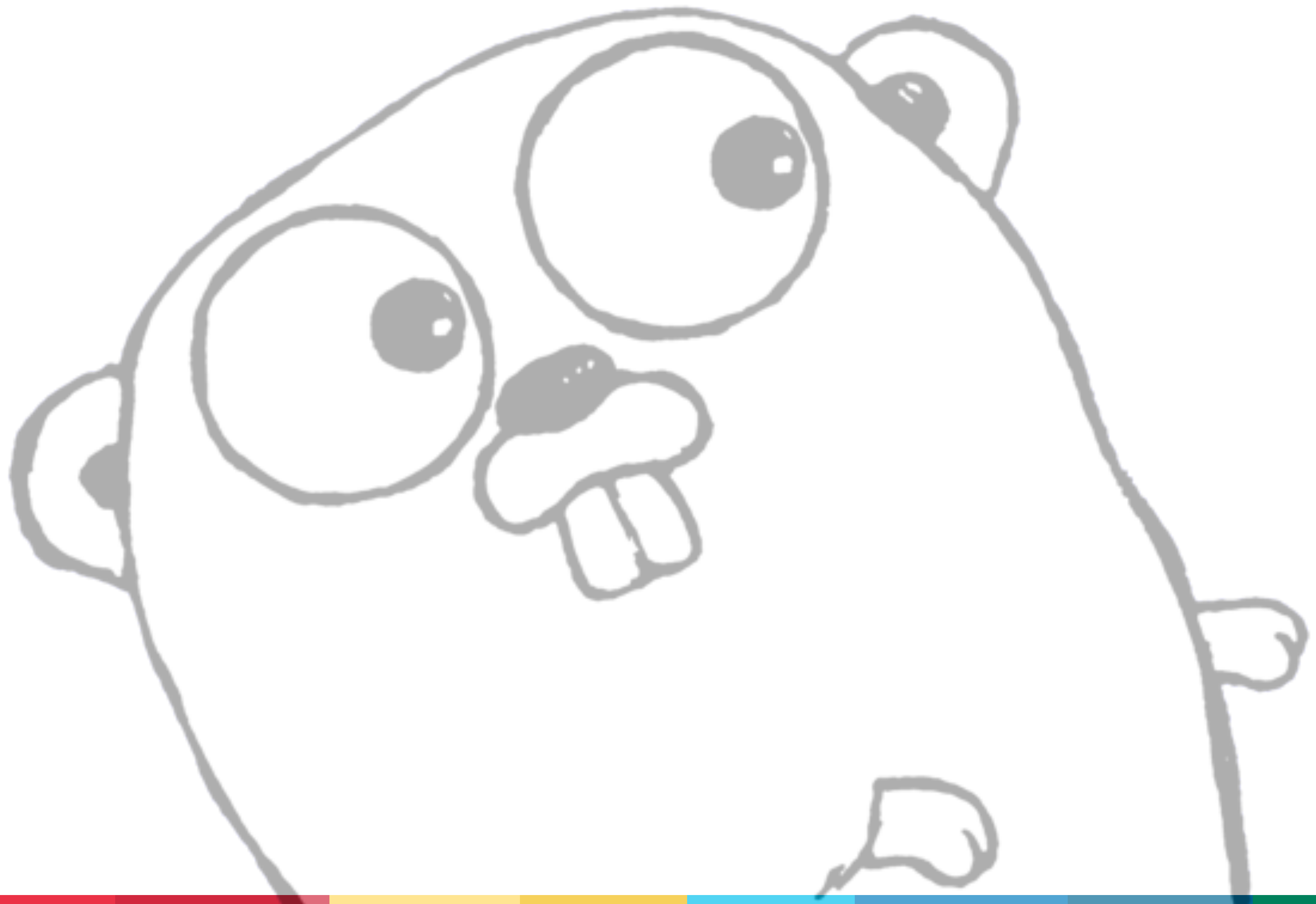
In the meantime, see

`http://golang.org`

for lots more information.

Read the 1978 CSP paper

It is deep and wise

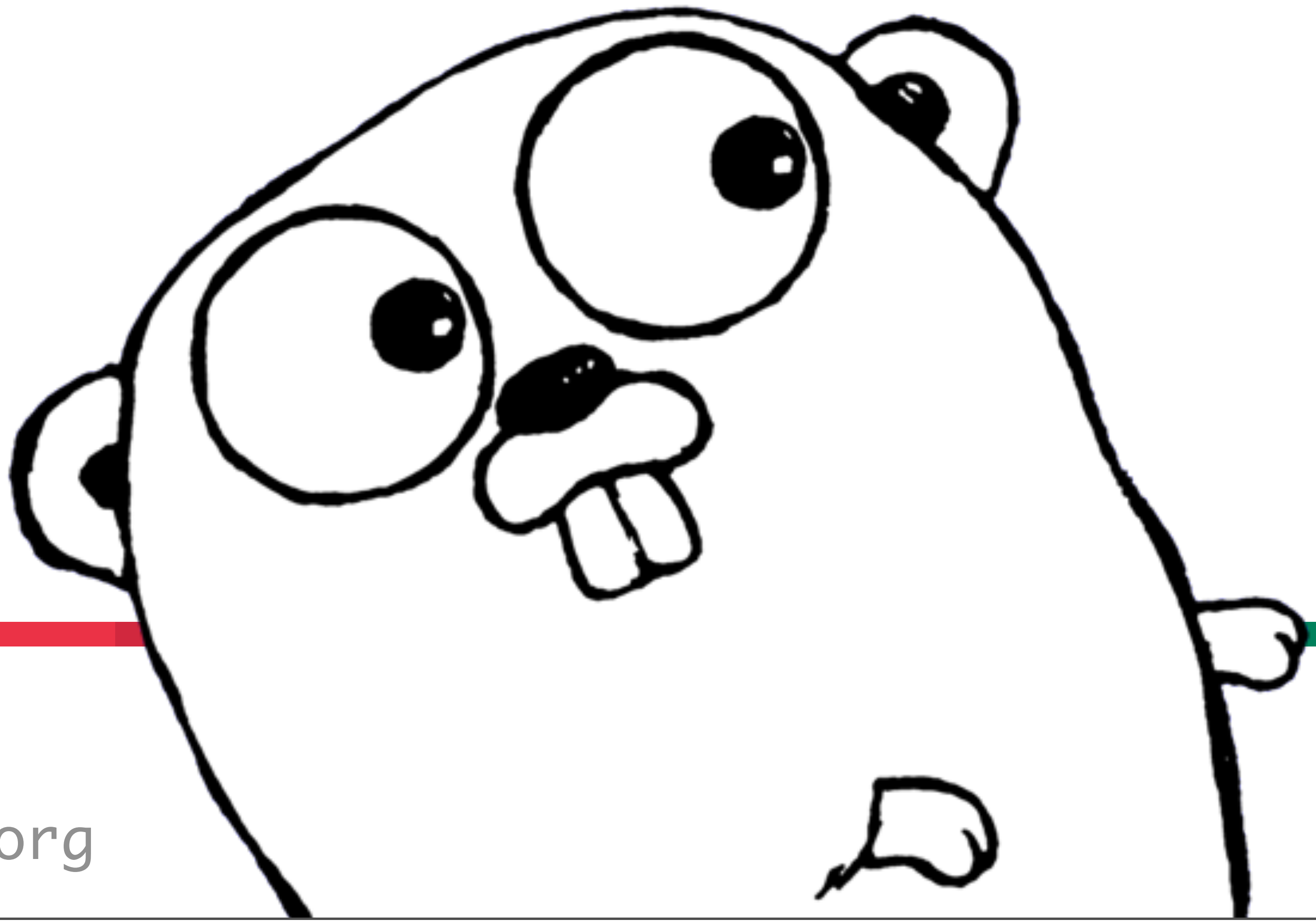


Quotes from the CSP paper

[Processes] may not communicate with each other by updating global variables.

In parallel programming coroutines appear as a more fundamental program structure than subroutines, which can be regarded as a special case.

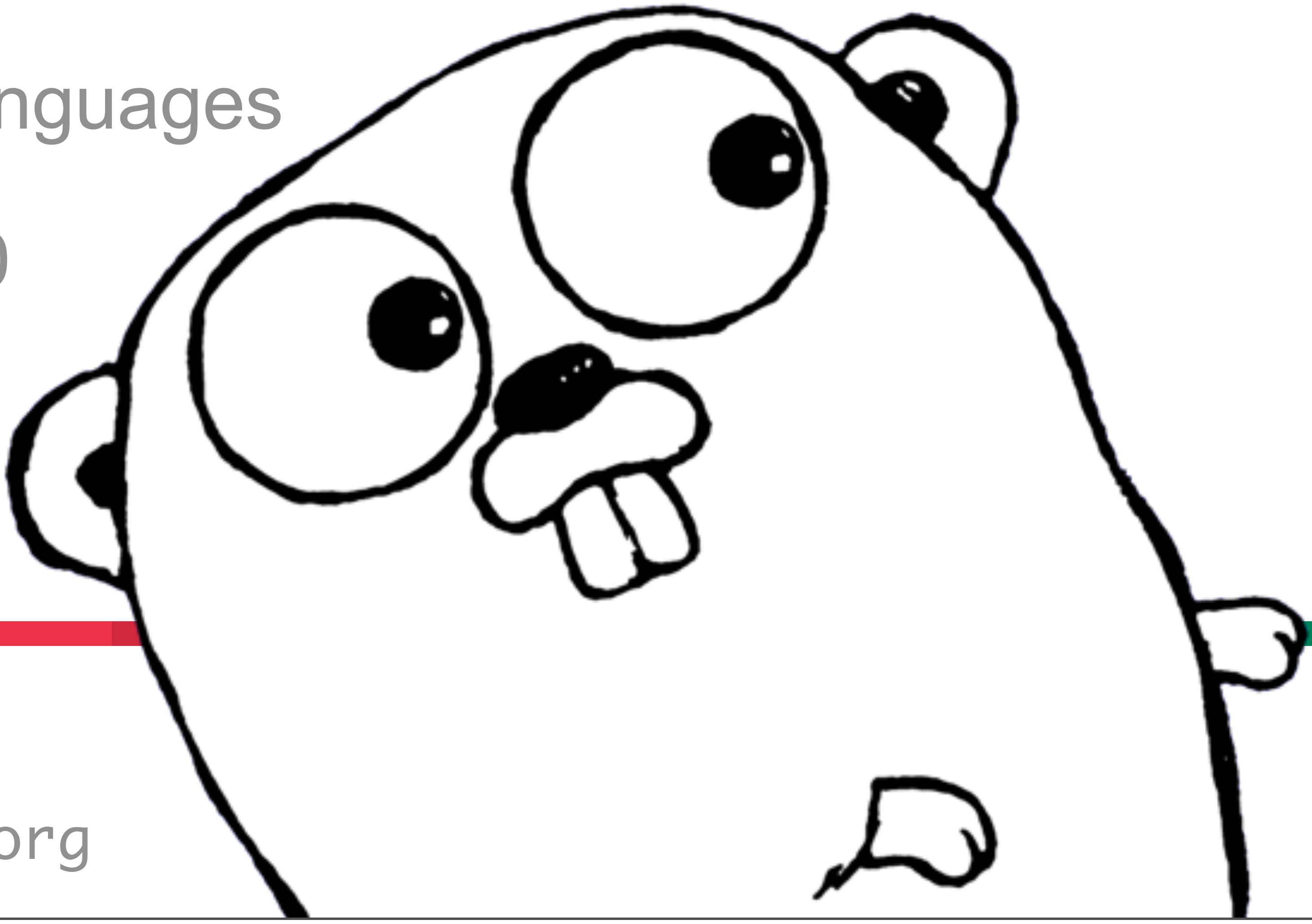
[A coroutine] may use input commands to achieve the effect of "multiple entry points" ... [and be] used like a SIMULA class instance as a concrete representation for abstract data.



http://golang.org

Go

Rob Pike
Emerging Languages
OSCON
July 21, 2010



http://golang.org

Google