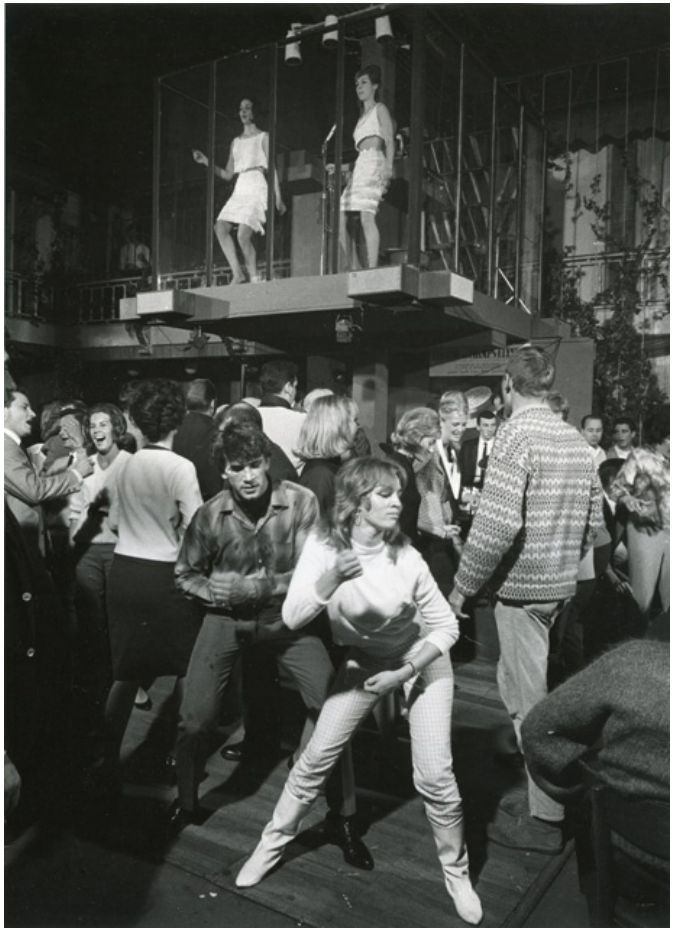
rs of Go:
g a new language by writing cool stuff in it

ollier-Brown
@spamcop.net

The Supposed Problem

- We needed to parse other people's XML and json at work. Unfortunately,
  - It constantly changed
  - It was often <borked \>
- Hidden agenda: I wanted a better language than Java or C++
  - > Java makes me unproductive
  - > C++ is a trade-off with malloc/free bugs
  - > Shell and Awk don't scale that well

Drew said “go go”



- I said “go-go dancers are so 1960s!”
- He meant the language

OK, now how do I evaluate a language?

- One way is to look for cool or elegant stuff
- Another is to rewrite something you know in it
- A third is to try writing something that was hard elsewhere in it

Look for cool or elegant stuff

- An elegant lexer/parser from Rob Pike
- Pipes as a built-in (from the Pike talk)
- Elliotte Rusty Harold's simplification of XML attributes as funny elements
- "Explain" what to compile, from model checking

Another is to rewrite something

- I’ve written path expressions at least two times before, in Java and C++
- It’s lots bigger than “Hello, world”

- It looks like
- /universe/galaxy[planet=earth]/timelord

A third is to do something that was hard elsewhere

- I had the “apptrace” in-context logger,
- … which was going to be in Solaris and Java before Oracle took over.

OK, path expressions in Go

- Aimed at validating a business scheme
  - The boss actually invalidated the hypothesis and trashed the project, saving us 6 man-months
- Perhaps a test to see if we should adopt another language
  - We should, but we can *staff* java and js
- The thing I really want to know:
  - Does it make me more productive?

Rob's Elegant Lexer/Parser

- Rob Pike has a talks about lexers, parsers and concurrency as a design tool at https://www.youtube.com/watch?v=HxaD_trXwRE
- To me concurrency is either pipes (good) or monitors (hard)
- To Rob, concurrency is pipes inside the program. "Communicating Sequential Processes" (Tony Hoare, circa 1978 and ignored by the underwhelming thereafter)

Cool thing 1: a 3-line lexer

```
for state := lexTag; state != nil; {
        state = state(lex)
}
close(lex.Pipe)
```

- What this says is “the lexing function returns the function to use next in the lexing”
- AKA, a 1-character look-ahead tells us what we need to do next

LexTag() lexes out XML tags

- We found ‘<’, it’s maybe an xml begin tag

```
tokenTypeFound = token.BEGIN
// Subject to change, though
if lex.Next() == '/' {
        tokenTypeFound = token.END
        lex.Ignore()
```

- Nope, it was an xml ‘</’ end tag

And at the end, it decides what's next

```
if lex.HasPrefix(">") {
        lex.Emit(tokenTypeFound, lex.Current())
        lex.Next();  lex.Ignore()
        return lexText // we expect text next
}
...
if lex.Next() == eof {
        return nil // we expect to stop now
```

Cool thing 2, pipes

- Remember I said close(lex.Pipe) ?
- The lexer writes tokens into a pipe to the parser
- The parser is
  - isolated from the lexer,
  - runs in parallel with it and
  - yet communicates just like a standalone Unix program does when reading stdin.
- No semaphores, no protecteds, no monitors.

The null parser is just a while loop

```
for Tok = range lex.Pipe {
        slice = append(slice, tok)
        if tok.Typ == token.EOF {
                break
        }
}
return slice // returns at EOF
```

- range is an iterator

This is “CSP” inside a language

- Invented in 1978 by Tony Hoare
  - Ignored by the “concurrency should be hard” crowd
  - Good for concurrency, not parallel programming
- Loved by Thompson and Richie
  - A subset in Unix since forever
  - This is a subset too

Supported by “Goroutines”

- Extra-lightweight threads
- As many as you’ve memory for
- Can all write to a channel, just like a Unix file: writes are atomic
- Therefor can do parallelism easily if there aren’t cross-dependencies
- Does have shared variable, locks, etc for the hard cases with cross-dependencies (and the hard-cases who like to write them)

Cool thing 3, composing classes

- I have two (well, almost three) lexers, which are different enough to not be subclasses of anything
- But they use a common bottom-half by including it in the struct that defines them

```
type jLex struct {          type xLex struct {
*lexer.Lexer                *lexer.Lexer
}                           }
```

- I also composed in a tracer, just for fun

Cool thing 4, “explain”

- This was originally from the AI world
- One asks a model checker to “explain” the steps it took to make the program fail
- I ask an interpreter to explain
  - The steps it took, and therefore
  - What it would generate if it was a compiler

Explain timelords?

```
$ jxpath -explain /universe/galaxy[world=earth]/timelord <who.xml

explanation:

path :=     pathExpr.NewPath(lexer.Lex(input));

value := path.FindFirst("universe").FindSuchThat("galaxy", "world",
"earth").FindFirst("timelord").TextValue()
```

- Get an input, and optionally save it for fetching multiple values
- Then get a single value, text in this case

Easily split into good production code

- Can be split into
  - > Find earth

    subPath := path.FindFirst("universe").FindSuchThat("galaxy", "world", "earth")

  - > Find timelords and other folks there

    subPath.FindFirst(“timelord”).TextValue()

    subPath.FindFirst(“Ford Prefect”).TextValue()

Recreating things: in-context tracing

- We did this in c as part of apptrace(1)

```
| starting with world:”earth",timelord:"who"...
| | begin lexer.(*Lexer).SkipOver()
| | | skipped whitespace ' '
| | end lexer.(*Lexer).SkipOver
| | begin lexer.(*Lexer).AcceptQstring()
| | | starting with "earth",timelord:"who"...
| | | returning "earth"
| | end lexer.(*Lexer).AcceptQstring
| | begin lexer.(*Lexer).Pop()
```

This is a class with three members

- Begin
- Print
- Printf
- Begin does the begin and end processing
  - Via defer, which runs one chunk of code now, and another at the return from the function
  - Think of “finally”, but not just for error-handling

Defined by an interface ...

- Which allows me to “optimize” the code

```
func New(fp io.Writer, expand bool) Trace {
        var real RealTrace
        var fake FakeTrace
        if fp == ioutil.Discard || fp == nil {
                // do discarding less expensively
               return &fake
```

- A factory method in one if-statement

Which does nothing, cleanly

```
type FakeTrace struct {}

func (* FakeTrace) Begin(args ...interface{}) func() {
        return func() {}
}
func (* FakeTrace) Printf(format string, v ...interface{}) {
}
```

Live demo

- Goes here...

What's Next?

- In-line performance measurement function for servers
- Rather simple interface
  - request(nBytes)
  - response(nBytes)
- Returns the full data set from my performance talks
  - Latency
  - Transfer time
  - Think time between requests
  - TPS and BPS

But how's the *language*, Dave?

- "Not half bad", as the English say
  - Not perfect (I dislike closures and captured variables)
- Way less academic experimentation
  - But some more major capabilities still need to be doped out
- More for professionals writing servers
  - I'm more productive than in Java or C++
  - That's huge, as I want to do a start-up