Thirteen Hours of Go: learning a new language by writing cool things in it.

I wanted to learn Go, and the best way for me to learn a language is to write something interesting in it. In the case of Go, I had listened to the talk by Rob Pike on lexers and parsers, and I had a use for an xpath-like program that would need a parser. This, therefor, was an chance to

- use an elegant lexer/parser combination from Rob Pike's talk,
- try out concurrency as a design tool, from the same talk,
- use "explain", a concept from model checkers, for code generation, and
- write a path expression engine for the third time.

Path expressions, like xml's *xpath*, are old friends, something I've written several times before. The idea of doing path expressions for json, xml and csv appealed to me: it was also something a particular customer might need.

Thirteen man-hours later, I had a minimal viable product, knew I was going to like Go, and had in the process had also created a less-difficult versions of an in-context logger from my *apptrace* days at Sun. So I sat down to write this blog.

The Problem to Solve.

APIs are popular these days, and at least one customer needed to extract small bits of information from large masses of json, xml and csv files.

The information we typically wanted was the same, typically about five things, but every provider had their own xml or json, and also buried them in large quantities other data. That made every case different.

Added to that, the xml and json wasn't always regular or even *correct*. One company infamously used a "<borked \>" tag to indicate an error, which promptly borked all my xml parsers. It should have been "/>", you see.

That gave me a good use case for a parser based on a talk by Rob Pike, https://talks.golang.org/2011/lex.slide1

The Lexer.

Rob was talking about design, and the design of the lexer was particularly simple. The main loop was just

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

We set a variable to (point at) the funtion startState(), then run it in the body of the loop. It return either the next state, or nil at end of file. When we get to the end of the file, we close something called “Pipe”.

Each “state function” lexes something, like a tag or a value, until it reaches the end. For example, lexTag() lexes an xml tag until it reaches the end. The next character is sees tells it what is to happen next. If the character is whitespace, we want to try to do text next and so we return lexText. If the next character is a ‘<’, then we want to try to lex a tag, and we return lexTag.

At the end, each state function has captured one or more characters and writes them down a channel I named “Pipe”, along with the lexical type of what it found.

The parser gets a stream of lexical objects, organizes them into tokens and append them to a list that the path expression engine can interpret. It does so without semaphores, protected and monitors: like a real pipe, all it does is read stdin and process what it gets.

The simplest possible parser is just a for-loop:

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

If the lexer fails to write an EOF token, there is no edge case: the close causes the range operator to report that it’s hit the end, the for-loop completes and the “slice” is returned.

At this point we have our input in a list-like data structure in memory, and we can traverse it. We don’t have to build a full DOM to apply a path-expression to it.

All that is cool, and it’s all either peculiar to go or is made easier by go. And it didn’t take long: after thirteen hours I had done a minimum viable product (MVP) and a couple of features that would be part of a real product.

What does it look like?

The minimum viable program does xpath-like extraction from the command-line, quickly.

Imagine we have an xml file about popular scienc-fiction shows and the path expression ”/universe/galaxy[planet=earth]/timelord”, which means “find me a galaxy where there is a planet named earth, and see if it had a timelord”.

We run *jxpath /universe/galaxy[planet=earth]/timelord <who.xml*

After lexing, we have a sequence of tokens.

```
{"BEGIN", "universe"} {"BEGIN", "galaxy"} {"BEGIN", "world"} {"VALUE", "earth"}
{"END", "world"} {"BEGIN", "timelord"} {"VALUE", "who"} {"END", "timelord"} {"END",
"galaxy"}
```

and typically a huge amount of other, uninteresting, data.

The path expression engine’s first step is to find the first universe. The second step is, within that universe, to find the first galaxy with a planet whose value is “earth” and select the slice from BEGIN “galaxy” to END “galaxy”.

Its last step is to find any timelords inside that slice and return their value. In this case it will find (doctor) “who”.

Another cool thing, done easily

As it happens, I wanted to not just interpret path expressions, but also compile a program for use in production, that extracts useful data like the name of the local timelord.

In the AI world one can ask a model checking program to “explain” the steps it took to make something fail the test.

You can do the same thing here. With any interpreter, you can have it record each of the methods it called and their parameters. If you say “--explain” you get a code snippit like

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

path := pathExpr.NewPath(lexer.Lex(input));
value := path.FindFirst("universe").FindSuchThat("galaxy",
         "world","earth").FindFirst("timelord").TextValue()
```

This is easy to edit into more efficient production code that doesn’t re-find the galaxy.

```
subPath := path.FindFirst("universe").FindSuchThat("galaxy", "world", "earth")
subPath.FindFirst(“timelord”).TextValue()
subPath.FindFirst(“Ford Prefect”).TextValue()
```

I could have written explain code in any language, but the great succinctness of Go gave me the extra time to add it, which then paid off by making it easy to see what code the expression engine was executing for a given input.

Explain is now part of my standard set of debugging tools!

Concurrent design: in-context trace.

At about the point where I had the lexer and parser communicating with each other as co-routines, I realized what Rob had meant about using concurrency as a design tool.

He had a lexer and parser that was best written as a set of pipeline stages.

I had an in-context-trace program that was ugly data-race code in most languages, except for one case where I had split into a c part and an awk part, which I called via the pipe() system call.

Sure enough, go test -race flagged my usual implementation as not thread-safe. When I moved the bookkeeping into a goroutine of its own, fed via a pipe, it happily wrote this to a logger:

```
|    right now, start is at "world>earth</world><timelord>who" ...
|   |    begin lexer.(*Lexer).Emit("BEGIN", world)
|   |    end lexer.(*Lexer).Emit
|    end xml.lexTag
|    begin xml.lexText()
|    Input="earth</world><world/><timelord>who</time" ...
|   |    begin lexer.(*Lexer).Emit("VALUE", earth)
|   |    end lexer.(*Lexer).Emit
|    end xml.lexText
|    begin xml.lexTag()
|    right now, start is at "/world><timelord>who</timelord><" ...
|   |    begin lexer.(*Lexer).Emit("END", world)
|   |    end lexer.(*Lexer).Emit
|    end xml.lexTag
```

Programs still need one logger each for the trace from each logical stream of processing, but that's normal, and a logical stream can be several goroutines in some cases.

I love interfaces.

In the process of rewriting the tracer, I also discovered just how easy it is to make it efficient. Previously it was purely a debugging tool, as it generated a lot of I/O that was dumped to /dev/null, and made the whole program slow.

By declaring the tracer to meet ("be") an interface, I could detect redirection to ioutil.Discard, a go equivalent of dev/null, and return a fake tracer:

```
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!

With this and a separate goroutine to do I/O, I can now see exactly how to write a transaction-performance measurement tool without letting it slow down the transactions it's measuring.

Things I haven't mentioned

The first is that at 13 hours, I definitely don't have a production program: the lexer and parser need re-balancing, to spread out the work properly.

I also haven't mentioned all the other cool things in Go: slices, for example, are bounded sub-lists, something my colleagues and I wanted in C since our university days.

Depending on what's important to you, your mileage will vary. If you dislike varags but love optional parameters, you might hate go. You'll have to make tha t decision yourself.

Conclusions

If you find you need to try out a language, I *will* recommend

- find something about it that's cool, and that you want to use
- rewrite something you know well, and
- try writing something you previously found hard.

It didn't take very long to see I was as productive in Go as in C, shell or Awk, and way more so than in C++ or Java.

The huge advantage for me was using concurrency as a design tool instead of something to be struggled with. Channels and *communicating sequential processes* are easy to reason about, and we've had years of experience with the subset that pipes provide in Unix.

Other things vary: I like slices, but I dislike closures capturing variables.

To sum it all up, this is another "Kernighan" language, one for professional programmers, with trade-offs, but with trade-offs that the authors considered carefully.

I'm going to enjoy doing more with Go.