Pause-able Parsing and Elegant Interpreters in Go: Using Goroutines as Coroutines

20 September 2016

Jason E. Aten, Ph.D.
Principal Engineer, Sauce Labs

problem: implementating an interpreter efficiently

- suppose your code is running, and deep inside a nested set of possibly mutually recursive calls...
- and you run out of input.
- ... do you start all over?
- ... and take O(n^2) time to parse an n-line program? Ouch.
- you want to save your state, and resume later, exactly where you left off...
- this is exactly what happens at the interpreter prompt

generally

- how to refactor your straight line code...
- to pause - and - resume gracefully
- to be interuptable
- to be lazy

benefits of this style

- more coherency: keep the readability of straight-line code
- insert pause points after the fact
- easier to read => means easier to maintain, refactor, and extend

context: zygomys interpreter

- an interpreted scripting language
- built in Go, for steering Go
- reflect to invoke compiled Go code
- zygomys has closures with lexical scope
- for loops
- higher order functions
- readable math: anything inside curly braces {} is infix. example: a = 2 * 5 + 4 / 2

https://github.com/glycerine/zygomys (https://github.com/glycerine/zygomys)

context II: architecture / overview of zygomys implementation

- a) lexer produces tokens
- b) parser produces lists and arrays of symbols <<<== focus of this talk
- c) macros run at definition type
- d) codegen produces s-expression byte-code
- e) a virtual machine executes the byte-code

what specifically changes to make code pause-able? And more importantly, resumable?

original parseArray (only 50% shown/fits on a screen)

```
// (original straight-line code:) parseArray handles `[2, 4, 5, "six"]` arrays of expressions
func (parser *Parser) parseArray(depth int) (Sexp, error) {
    for { // get the next token, then break
    getTok:
        for {
            tok, err = parser.lexer.peekNextToken()
            if err != nil {
                return SexpEnd, err
            }

            if tok.typ == TokenComma {
                // pop off the ,
                _, _ = parser.lexer.getNextToken()
                continue getTok
            }

            if tok.typ != TokenEnd {
                break getTok // got a token
            } else {
                return nil, io.EOF // <<<<<<<<<<<<<   sad, done before finding ']'
            }
        }

        if tok.typ == TokenRSquare {
            // pop off the ]
            _, _ = parser.lexer.getNextToken()
            break
```

```
        }

        expr, err := parser.parseExpression(depth + 1)
        if err != nil {
            return SexpNull, err
        }
        arr = append(arr, expr)
    }

    return &SexpArray{Val: arr, Env: parser.env}, nil
}
```

before, closeup

```
// BEFORE: original straight-line code
func (parser *Parser) parseArray(depth int) (Sexp, error) {
...
            if tok.typ != TokenEnd {
                break getTok // got a token
            } else {
                return nil, io.EOF // <<<<<<<<<<<<   sad, done before finding ']'
            }
...
```

after, closeup

```
// AFTER: we call getMoreInput()
func (parser *Parser) parseArray(depth int) (Sexp, error) {
...

            if tok.typ != TokenEnd {
                break getTok
            } else {
                // we ask for more, and then loop
                err = parser.getMoreInput(nil, ErrMoreInputNeeded) <<<<=== key change
                switch err {
                case ParserHaltRequested:
                    return SexpNull, err
                case ResetRequested:
                    return SexpEnd, err
                }
            }
...
```

zoom out: after in full context

```
// AFTER in context
func (parser *Parser) parseArray(depth int) (Sexp, error) {
    for { // get the next token, then break
    getTok:
        for {
            tok, err = parser.lexer.peekNextToken()
            if err != nil {
                return SexpEnd, err
            }
            if tok.typ == TokenComma {
                // pop off the ,
                _, _ = parser.lexer.getNextToken()
                continue getTok
            }
            if tok.typ != TokenEnd {
                break getTok // got a token
            } else {
                // we ask for more, and then loop
                err = parser.getMoreInput(nil, ErrMoreInputNeeded) // <<<=== key change
                switch err {
                case ParserHaltRequested:
                    return SexpNull, err
                case ResetRequested:
                    return SexpEnd, err
                }
            }
        }
```

```
        if tok.typ == TokenRSquare {
            // pop off the ]
            _, _ = parser.lexer.getNextToken()
            break
        }

        expr, err := parser.parseExpression(depth + 1)
        if err != nil {
            return SexpNull, err
        }
        arr = append(arr, expr)
    }

    return &SexpArray{Val: arr, Env: parser.env}, nil
}
```

the key was getMoreInput() call instead of returning io.EOF... simple enough, but...

that begs the question, how does getMoreInput() work...

apparently the real magic is in getMoreInput(). It must be doing the heavy lifting...

getMoreInput()

```
// getMoreInput does I/O: it is called by the Parser routines mid-parse to get the user's next line
func (p *Parser) getMoreInput(deliverThese []Sexp, errorToReport error) error {
    if len(deliverThese) == 0 && errorToReport == nil {
        p.FlagSendNeedInput = true
    } else {
        p.sendMe = append(p.sendMe,ParserReply{Expr: deliverThese,Err:  errorToReport})
    }
    for {
        select {
        case <-p.reqStop:
            return ParserHaltRequested
        case input := <-p.AddInput:
            p.lexer.AddNextStream(input)
            p.FlagSendNeedInput = false
            return nil
        case input := <-p.ReqReset:
            p.lexer.Reset()
            p.lexer.AddNextStream(input)
            p.FlagSendNeedInput = false
            return ResetRequested
        case p.HaveStuffToSend() <- p.sendMe:  // a conditional send!
            p.sendMe = make([]ParserReply, 0, 1)
            p.FlagSendNeedInput = false
}}}
```

HaveStuffToSend() is easy...

```
func (p *Parser) HaveStuffToSend() chan []ParserReply {
    if len(p.sendMe) > 0 || p.FlagSendNeedInput {
        return p.ParsedOutput
    }
    return nil
}
```

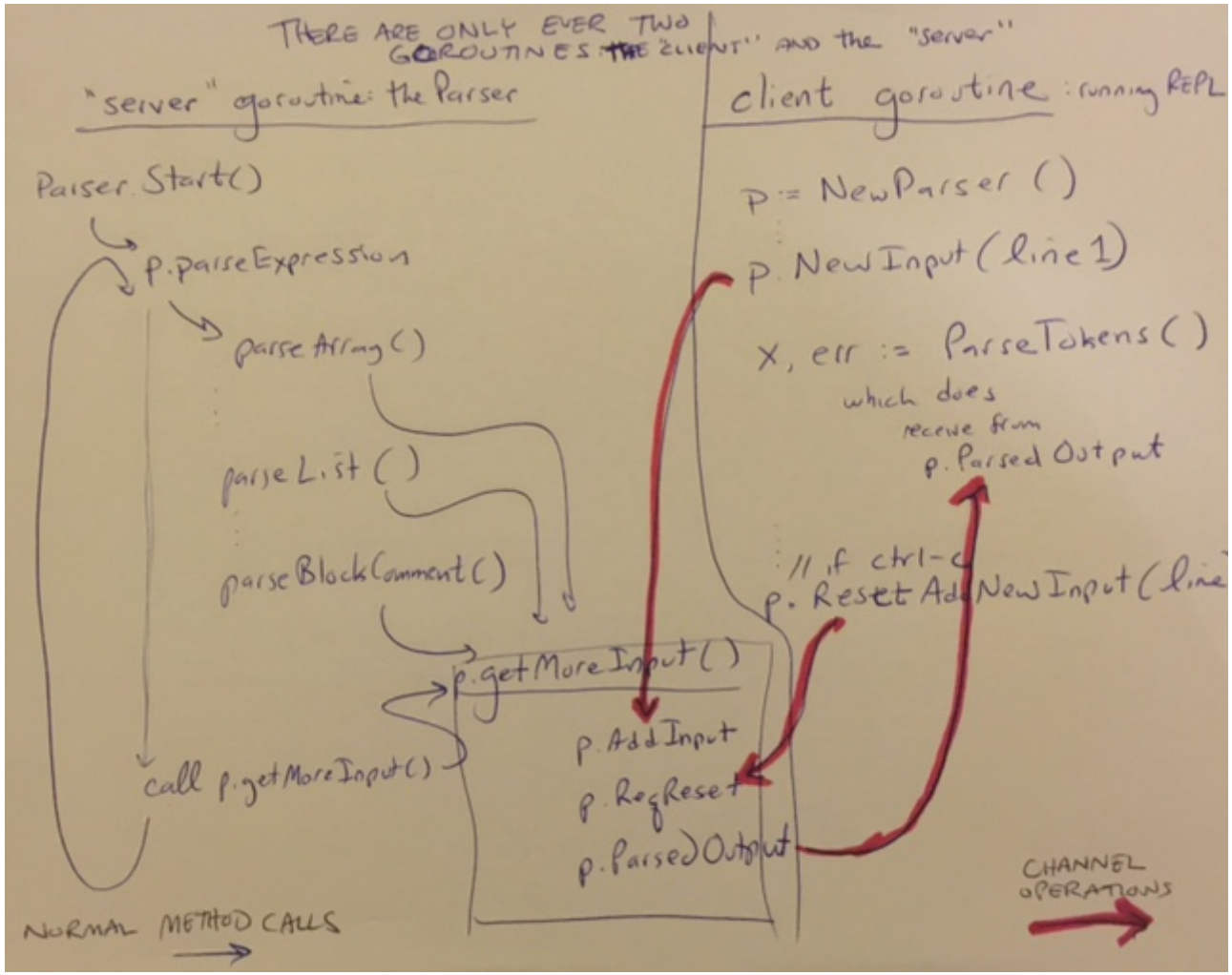
what is unusual about getMoreInput()

- it can be called from multiple places
- callers get to retain the entire context of their call stack
- getMoreInput() returns to its caller precisely once the caller can continue
- and in the meantime, it does the channel work in a select{} to get more input from an asynchronous source
- In my humble experience, this is rare: a co-routine pattern
- Caller's code gets to pause. And then resume, right where it left off.

supporting player: a background goroutine running an infinite loop that drives parsing. It also calls getMoreInput() to start top-level parsing.

```
// Start() commences the background infinite loop of parsing
func (p *Parser) Start() {
    go func() {
    defer close(p.Done)
    expressions := make([]Sexp, 0, SliceDefaultCap)
    for {
        expr, err := p.parseExpression(0)
        if err != nil || expr == SexpEnd {
            if err == ParserHaltRequested {
                return
            }
            err = p.getMoreInput(expressions, err) // SexpEnd means we need more input
            if err == ParserHaltRequested {
                return
            }
            expressions = make([]Sexp, 0, SliceDefaultCap)
        } else {
            expressions = append(expressions, expr)
        }
    }
}
```

call graph


THERE ARE ONLY EVER TWO GOROUTINES THE "CLIENT" AND the "server"
"server" goroutine: the Parser
client goroutine: running REPL
Parser.Start()
p.parseExpression
parseArray()
parseList()
parseBlockComment()
p.getMoreInput()
call p.getMoreInput()
p.AddInput
p.ReqReset
p.ParsedOutput
p := NewParser()
p.NewInput(line1)
x, err := ParseTokens()
which does receive from p.ParsedOutput
// if ctrl-c
p.Reset AddNewInput(line
NORMAL METHOD CALLS
CHANNEL OPERATIONS

so what does the Parser API look like from the outside?

here is what the user sees:

what the Parser API looks like

```
// Start() commences the background parse loop goroutine.
func (p *Parser) Start()


// ParseTokens is the main service the Parser provides.
// Currently returns first error encountered, ignoring
// any expressions after that.
func (p *Parser) ParseTokens() ([]Sexp, error)


// NewInput is the principal API function to
// supply parser with addition textual
// input lines
func (p *Parser) NewInput(s io.RuneScanner)


// ResetAddNewInput is the principal API function to
// tell the parser to forget everything it has stored,
// reset, and take as new input the scanner s.
func (p *Parser) ResetAddNewInput(s io.RuneScanner) {


// Stop gracefully shutsdown the parser and its background goroutine.
func (p *Parser) Stop() error
```

conclusion

- coroutine patterns are viable in Go
- we can avoid O(n^2) time interpreter parsing
- other uses: functional programming patterns like (lazy) generators[1]

-

-

[1] John Hughes, Why Functional Programming Matters

www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf (https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf)

Thank you

Jason E. Aten, Ph.D.
Principal Engineer, Sauce Labs
j.e.aten@gmail.com (mailto:j.e.aten@gmail.com)
@jasonaten_ (http://twitter.com/jasonaten_)

https://github.com/glycerine/zygomys (https://github.com/glycerine/zygomys)

(#ZgotmplZ) github.com/glycerine/zygomys/wiki. (https://github.com/glycerine/zygomys/wiki.) The wiki has details, examples, and discussion.