Parsing and interpretation with goyacc and closures (continued)

Vadim Vygonets
August 2022

stage 4: Error handling

stage 4: functions

Let's make the fun return `(int, error)`:

```
type fun func() (int, error)
```

And convert `varMap.Get()` and `list.Run()` to return errors:

```
func (vl varMap) Get(s string) (int, error) {
    if n, ok := vl[s]; ok {
        return n, nil
    }
    return 0, fmt.Errorf("unknown variable %s", s)
}
```

```
func (l list) Run() error {
    for _, v := range l {
        if _, err := v(); err != nil {
            return err
        }
    }
    return nil
}
```

stage 4: functions

To avoid boilerplate, let's write a function that will wrap `func(int, int) (int, error)` in a closure calling `left` and `right` operands and propagating errors.

```
func newFun(left, right fun, f func(int, int) (int, error)) fun {
    return func() (int, error) {
        a, err := left()
        if err != nil {
            return 0, err
        }
        b, err := right()
        if err != nil {
            return 0, err
        }
        return f(a, b)
    }
}
```

stage 4: parser

Now creating arithmetic `fun`s is a bit simpler:

```
|       expr '+' expr2
        {
                $$ = newFun($1, $3, func(a, b int) (int, error) {
                        return a + b, nil
                })
        }
```

Division is a little more elaborate:

```
|       expr2 '/' expr3
        {
                $$ = newFun($1, $3, func(a, b int) (int, error) {
                        if b == 0 {
                                return 0, ErrZeroDivision
                        }
                        return a / b, nil
                })
        }
```

stage 4: for loops

for loops are a mess though:

```
|       FOR stmt2 ';' expr ';' stmt2 block
        {
                a, b, c := $2, $4, append($7, $6)
                $$ = func() (int, error) {
                        if _, err := a(); err != nil {
                                return 0, err
                        }
                        for {
                                if v, err := b(); err != nil || v == 0 {
                                        return 0, err
                                }
                                if err := c.Run(); err != nil {
                                        return 0, err
                                }
                        }
                }
        }
```

stage 4: blocks

To make `for` loops simpler, I also changed `block` to be a list and moved it under `stmts`.

```
|       stmts block
        {
                $$ = append($1, $2...)
        }
```

```
block:
        '{' stmts '}'
        {
                $$ = $2
        }
```

stage 4: demo time...

```
s := `
for i = 0; i - 5; i = i + 1 {
    for j = -2; j; j = j + 1 {
        i; j; j*10/(i+2*j)
    }
}
`
go yy.run(bytes.NewBufferString(s))
fmt.Println("parser returned", yyParse(&yy))
if err := runtime.top.Run(); err != nil {
    fmt.Println(err)
}
return
```

Run

stage 5: Interactive calculator

stage 5: interactive calculator

An interactive calculator such as `bc(1)` receives input lines and runs them. This essentially means sending `$end` after each input line and running `yyParse` in a loop. There are several complications though.

- We should not send `$end` within a block, which requires us to track `'{'` and `'}'` tokens.
- After sending `$unk` we should skip the rest of the line.
- Parsing can fail at any point, so the lexer should receive parse status and skip the rest of the line on failures.
    - Unless we're at the beginning of a line, which means the error is on the previous line.
- We should do this only in interactive sessions.
- We should distinguish between `$end` and EOF.

stage 5: end of file

To handle EOF we introduce the notion of a command token, a piece of code that is sent to the parser in its own parsing session.

```
%token <fun> CMD
```

```
top:
        stmts
        {
                runtime.top = $1
        }
|       CMD
        {
                runtime.top = append(runtime.top[:0], $1)
        }
```

The EOF command will set a boolean variable that the parse loop will check.

```
func cmdEOF() (int, error) {
    runtime.eof = true
    return 0, nil
}
```

stage 5: input loop

Let's separate the input into its own goroutine sending input lines to the lexer on a channel. It will signal EOF by sending an empty string, and replace empty lines with a string containing a single space.

```
func (yy *yyLex) input() {
    sc := bufio.NewScanner(yy.r)
    for sc.Scan() {
        s := sc.Text()
        if s == "" {
            s = " "
        }
        yy.in <- s
    }
    if err := sc.Err(); err != nil {
        fmt.Fprintln(os.Stderr, err)
    }
    yy.in <- ""
}
```

stage 5: parse loop

The parser will run `yyParse` in a loop, sending its return value to a channel, until `eof`.

```
func (yy *yyLex) parse() {
    go yy.input()
    go yy.run()
    for !runtime.eof {
        status := yyParse(yy)
        if status == 0 {
            if err := runtime.top.Run(); err != nil {
                fmt.Fprintln(os.Stderr, err)
            }
        }
        yy.ps <- status | lexParserStatus
    }
}
```

stage 5: parser status

The lexer will try to receive the parser status when operating on other channels.

```
const (
    lexOK           = iota                   // operation completed
    lexParseSuccess = iota | lexParserStatus // parser signalled success
    lexParseError                            // parser signalled failure

    lexParserStatus = 0x02 // flag for parser status
)
```

stage 5: sendToken()

A parse error may happen after any token, therefore we must be ready to receive the parse status in `sendToken()`.

Go `select` statement will run one of the two alternatives, depending on which channel becomes ready first. It will either receive the parse status (if parser sent it), or send the token (if `Lex()` is called).

In the first case, the function may be called later for the same token. We'll introduce a new member to `yyLex` to hold it, called `next`.

```
func (yy *yyLex) sendToken() int {
    select {
    case status := <-yy.ps:
        return status
    case yy.c <- yy.next:
        yy.last = yy.next
        return lexOK
    }
}
```

stage 5: sendToken() helpers

A couple of helper functions: one to send an arbitrary token,

```
func (yy *yyLex) send(tok token) int {
    yy.next = tok
    return yy.sendToken()
}
```

and another to send $end and wait for the parser status.

```
func (yy *yyLex) sendEnd() int {
    if status := yy.send(token{}); status != lexOK {
        return status
    }
    return <-yy.ps
}
```

stage 5: getLine()

Another convenient point to receive parse status would be while waiting for input.

We'll put the input string in the yyLex structure too, to avoid passing it around.

```
func (yy *yyLex) getLine() int {
    select {
    case status := <-yy.ps:
        return status
    case yy.s = <-yy.in:
        return lexOK
    }
}
```

stage 5: nextToken()

We'll change `nextToken()` to use the new `yyLex` members and return `bool`.

```
func (yy *yyLex) nextToken() bool {
    s := strings.TrimSpace(yy.s)
    if s == "" {
        return false
    }
```

`[...]`

```
    tok.s, yy.s = s[:tlen], s[tlen:]
    yy.next = tok
    return true
}
```

stage 5: yyLex

The lexer struct suddently has a lot of members.

```
type yyLex struct {
    r    io.Reader   // input
    tty  bool        // interactive session with a human at a teletype
    in   chan string // channel for input lines
    c    chan token  // channel for tokens sent to the parser
    ps   chan int    // channel for parser status
    s    string      // input string
    next token       // next token to send
    last token       // last token sent
}
```

This requires an initialisation function.

stage 5: newLexer()

Note that the channels are created unbuffered to keep the goroutines synchronised. This is especially important for the `token` channel `c` during interactive sessions, so that when parsing fails there aren't a bunch of tokens queued up.

```
func newLexer(r io.Reader) *yyLex {
    yy := yyLex{
        r:  r,
        c:  make(chan token),
        in: make(chan string),
        ps: make(chan int),
    }
    if f, ok := r.(*os.File); ok {
        yy.tty = isatty.IsTerminal(f.Fd())
    }
    return &yy
}
```

`isatty` is a package telling whether a file descriptor ***is a t**ele**ty**pe*.

```
    "github.com/mattn/go-isatty"
```

This is a teletype, BTW.

Bundesarchiv Bild 183-2008-0516-500, Fernschreibmaschine mit Telefonanschluss

stage 5: lexer

Now we're ready to start looking at the lexer.

I tried writing it as a state machine, but serial code is easier to understand.

stage 5: lexer

Process lines in a loop.

If a parse status is received while waiting for input, reset the lexer. We do it with a `goto`, because `goto` rules.

On end of file we `break`.

```
func (yy *yyLex) run() {
    var (
        depth int
        first bool
    )
    for {
        if yy.getLine() != lexOK {
            goto reset
        } else if yy.s == "" {
            break
        }
```

stage 5: lexer

While there are tokens on the input line, send them. In an interactive session, if parser status is received while trying to send the first token on the input line, try again.

```
        first = true
        for yy.nextToken() {
            for yy.sendToken() != lexOK {
                /*
                 * when sending the first token in an input
                 * line fails, it means the error is on the
                 * previous line.  if in an interactive
                 * session, reset depth and try sending again.
                 */
                if yy.tty && first {
                    depth = 0
                    continue
                }
                // otherwise reset (skip line or bail out)
                goto reset
            }
```

stage 5: lexer

What did we just send?

If we just told the parser to stop, wait for the status and reset.

For curly braces, keep track of the nesting depth.

```
            switch yy.last.typ {
            case 0, 1:
                // sent $end or $unk: wait for status and reset
                <-yy.ps
                goto reset
            case '{':
                depth++
            case '}':
                depth--
            }
            first = false
        }
```

stage 5: lexer

We have reached the end of the line.

If the last token is $end or $unk, this means we have sent no tokens in this session. Go straight to reading the next input line.

Unless the last token was a semicolon, inject one. Reset if parser stopped.

```
        // end of line
        switch yy.last.typ {
        case 0, 1:
            // if we haven't sent any tokens, read next line
            continue
        case ';':
            // no semicolon needed
        default:
            // inject semicolon at EOL
            if yy.send(token{typ: ';'}) != lexOK {
                goto reset
            }
        }
```

stage 5: lexer

In interactive sessions, send $end if the depth is zero. Negative depth should cause parse errors, but it's more robust to handle it here too.

sendEnd( ) returns the parse status (it never returns `lexOK`). If parsing failed, reset.

Then loop back to reading the next input line.

```
        if yy.tty && depth <= 0 {
            // interactive and not within a block:
            // send $end and reset depth
            if yy.sendEnd() != lexParseSuccess {
                goto reset
            }
            depth = 0
        }
        continue
```

stage 5: lexer

How do we `reset`?

In non-interactive sessions we don't. We just quit. It's a parse error in a file.

In interactive session we reset the depth and keep going (from the next line).

```
    reset:
        if !yy.tty {
            break
        }
        depth = 0
    }
```

stage 5: lexer

End of file! End the current parse session and send the EOF command in its own session.

```
    // EOF
    // we could check yy.last here to avoid sending $end
    // after $end or $unk, but this is simpler and more robust.
    yy.sendEnd()                            // send $end
    yy.send(token{typ: CMD, fun: cmdEOF}) // send EOF command
    yy.sendEnd()                            // send $end
}
```

stage 5: demo? does anybody care at this point?

```
        s := `2+2
for i = 0; i - 5; i = i + 1 {
    for j = -2; j; j = j + 1 {
        i; j
    }
}
for i = 0; i - 5; i = i + 1 {
    i - * j
i * 2
`
        yy := newLexer(newSlowReader(s))
        yy.tty = true
        yy.parse()
        return
```

Run

stage 6: Floats and more operators

stage 6: floats

Let's have some floating point numbers.

```
type number struct {
    i       int
    f       float64
    isFloat bool
}
```

```
type fun func() (number, error)
```

This is nice, but the yacc code in the last stage was a bit tiresome. I can't imagine what it would be like to add floats to it.

Let's backtrack a little and start with reducing boilerplate.

stage 6a: Binary integer operators

stage 6a: binary integer operators

All the parser blocks for binary operators are the same, only the code is different.

But operators *are* code, right?

Let's treat them as such.

```
type op func(int, int) (int, error)
```

We'll pass them to the parser.

```
%union {
        num  int
        word string
        op   op
        fun  fun
        list list
}
```

stage 6a: binary integer operators: a better parser?

Assign this type to tokens and rules...

```
%token <op> '+' '-' '*' '/' '%'
```

```
%type <op> op1 op2
```

Change the order of parameters in newFun, and we can simplify the parser:

```
expr:
        expr2
|       expr op1 expr2
        {
                $$ = newFun($1, $2, $3)
        }

op1:    '+' | '-'
```

stage 6a: newFun

This is what newFun looks like now:

```
func newFun(left fun, f op, right fun) fun {
    return func() (int, error) {
        a, err := left()
        if err != nil {
            return 0, err
        }
        b, err := right()
        if err != nil {
            return 0, err
        }
        return f(a, b)
    }
}
```

stage 6a: binary integer operators: implementation

Helps to have a map!

```
type opMap map[byte]op

var ops = opMap{
    '+': func(a, b int) (int, error) { return a + b, nil },
    '-': func(a, b int) (int, error) { return a - b, nil },
    '*': func(a, b int) (int, error) { return a * b, nil },
    '/': func(a, b int) (int, error) {
        if b == 0 {
            return 0, ErrZeroDivision
        }
        return a / b, nil
    },
    '%': func(a, b int) (int, error) {
        if b == 0 {
            return 0, ErrZeroDivision
        }
        return a % b, nil
    },
}
```

stage 6a: binary integer operators: passing it on

Now we can add the op to the `token` structure and pass it to the parser.

yyLex.`nextToken`:

```
    case strings.Index(bareTokens, s[:1]) != -1:
        tok.typ = int(s[0])
        if op, ok := ops[s[0]]; ok {
            tok.op = op
        }
```

stage 6b: Floating point

stage 6b: floating point

We already have `number` and `fun`:

```
type number struct {
    i       int
    f       float64
    isFloat bool
}
```

```
type fun func() (number, error)
```

Let's print numbers.

```
func (a number) String() string {
    if a.isFloat {
        return strconv.FormatFloat(a.f, 'g', -1, 64)
    }
    return strconv.FormatInt(int64(a.i), 10)
}
```

stage 6b: floating point: nextToken

Now we need to add them to `yyLex.nextToken`. We'll accept `'.'` and try to parse the number as `uint` and as `float64`.

```
    case s[0] >= '0' && s[0] <= '9':
        for tlen < len(s) &&
            (s[tlen] >= '0' && s[tlen] <= '9' || s[tlen] == '.') {
            tlen++
        }
        if u, err := strconv.ParseUint(s[:tlen], 10, 63); err == nil {
            tok.typ = NUM
            tok.n.i = int(u)
            break
        }
        if f, err := strconv.ParseFloat(s[:tlen], 64); err == nil {
            tok.typ = NUM
            tok.n.f = f
            tok.n.isFloat = true
        } else {
            fmt.Fprintln(os.Stderr, err)
        }
```

stage 6b: floating point

Let's define the simplest op type for binary operators, and the types of functions we'll build it from:

```
type (
    binIntFun   func(int, int) (int, error)
    binFloatFun func(float64, float64) (float64, error)
    op          struct {
        i binIntFun
        f binFloatFun
    }
)
```

Let's put newFun under op, so that we can all it like this:

```
expr:
        expr2
|       expr op1 expr2          { $$ = $2.NewFun($1, $3) }

op1:    '+' | '-'
```

stage 6b: floating point: op.NewFun

The closure it returns will start as before, by calling the operands and checking errors.

```
func (o op) NewFun(left fun, right fun) fun {
    return func() (number, error) {
        a, err := left()
        if err != nil {
            return number{}, err
        }
        b, err := right()
        if err != nil {
            return number{}, err
        }
```

Now we have to choose which function to call according to the types of the operands.

But what if the types of the arguments don't match?

stage 6b: floating point: op.NewFun

Cast them to the same type.

If both operands are `int`, we'll call `o.i`.

Otherwise we'll call `o.f`, converting one of the operands to `float64` if required.

```
        switch {
        case !a.isFloat && !b.isFloat:
            a.i, err = o.i(a.i, b.i)
            return a, err
        case !a.isFloat:
            a = number{f: float64(a.i), isFloat: true}
        case !b.isFloat:
            b = number{f: float64(b.i), isFloat: true}
        }
        a.f, err = o.f(a.f, b.f)
        return a, err
    }
}
```

stage 6b: floating point: opMap

Now we only have to change `int` to `number` in many places, including `opMap`, which now has entries like this:

```
type opMap map[byte]op
```

```
    '%': {
        func(a, b int) (int, error) {
            if b == 0 {
                return 0, ErrZeroDivision
            }
            return a % b, nil
        },
        func(a, b float64) (float64, error) {
            if b == 0 {
                return 0, ErrZeroDivision
            }
            return math.Mod(a, b), nil
        },
    },
```

stage 6b: demo

```
func main() {
    yyErrorVerbose = true
    if true {
        s := `
        for i = 0; i - 5; i = i + 1 {
            for j = -2; j; j = j + 1 {
                i; j / 2.0
            }
        }
        `
        yy := newLexer(bytes.NewBufferString(s))
        yy.parse()
        return
    }
    yy := newLexer(os.Stdin)
    yy.parse()
}
```

Run

stage 6: Floats and ops

stage 6: floats and ops

Ok, so now we have binary integer and floating point operators. But we also need:

- operators that can be unary or binary (like `'-'`)

To have usable `for` loops it'll be nice to have:

- comparison and logic operators that return `int` for any operands (`'<'`, ==, &&)
- same but unary (`'!'`)
- fancy assignment operators like +=, perhaps ++ and `--`

If we want all operators that Go has, we also need:

- operators that do only `int`, like `'^'` and <<

The only operators that need internal error checking are `'/'` and `'%'`, so it would also be nice not to have to return errors from every function.

stage 6: op type

We need a more general op type. But what should it be?

Hmm.

Maybe one of the Go authors can help us?

> *Is a sortable array an array that sorts or a sorter represented by an array? [...]*
>
> *I believe that's a preposterous way to think about programming. What matters isn't the ancestor relations between things but what they can do for you.*
>
> — Rob Pike, *Less is exponentially more*, 2012

Thanks, Rob. I have seen the light, and I shall define a proper type.

stage 6: the interface

At this point we all know what a binary op should do for us. This.

```
type op interface {
    NewFun(fun, fun) fun
}
```

What about unary operators?

- We can define another interface. But this sounds like it would complicate operators that could be both unary and binary.
- We can add another function like `NewUnaryFun(fun) fun` to the interface. But it looks like we'll need many types, so adding another function to each of them sounds cumbersome.
- We can **reuse** `NewFun` **for unary operators** but ignore one of the arguments. This sounds simplest, so let's do this.

stage 6: basic op types

Right. Let's see what we can do with it.

It looks like we will need many kinds of operators with different behaviour. But all of them will have to do error checking, so let's define unary and binary operator types as generically as possible: as functions that receive and return `number`.

Let's also define the corresponding functions for ints and floats.

```
type (
    unOp        func(number) number
    binOp       func(number, number) number
    unIntFun    func(int) int
    unFloatFun  func(float64) float64
    binIntFun   func(int, int) int
    binFloatFun func(float64, float64) float64
)
```

We'll deal with comparisons later.

stage 6: unary ops

Let's start with the simple case: unary operators. `NewFun()` ignores the `right` operand, but otherwise there's nothing new here.

```
func (f unOp) NewFun(left, right fun) fun {
    return func() (number, error) {
        a, err := left()
        if err != nil {
            return number{}, err
        }
        return f(a), nil
    }
}
```

With this we can already define an op that prints:

```
    printOp unOp = func(a number) number { fmt.Println(a); return a }
```

Wait, did we just define a method on a function type? Yes we did. Thanks for the inspiration, `http.HandlerFunc`.

stage 6: unary ops

Add conversions from `bool` to `number`: `false` → `int(0)`, `true` → `int(1)`;
and from `number` to `bool`: `int(0)`, `float64(0.0)` → `false`, non-zero → `true`.

```
func boolToNumber(b bool) number {
    if b {
        return number{i: 1}
    }
    return number{}
}

func (a number) Bool() bool {
    if a.isFloat {
        return a.f != 0
    }
    return a.i != 0
}
```

...and we can define `'!'`:

```
    notOp   unOp = func(a number) number { return boolToNumber(!a.Bool()) }
```

stage 6: unary ops

Now we just need to create an unOp from two functions,

```
func newUnOp(uif unIntFun, uff unFloatFun) unOp {
    return func(a number) number {
        if a.isFloat {
            a.f = uff(a.f)
        } else {
            a.i = uif(a.i)
        }
        return a
    }
}
```

and then we can define an unary minus:

```
        newUnOp(
            func(a int) int { return -a },
            func(a float64) float64 { return -a },
        ),
```

stage 6: unary ops

With a little helper function and a constructor we can create integer-only unary ops.

```
func (a number) Int() int {
    if a.isFloat {
        return int(a.f)
    }
    return a.i
}
```

```
func newUnIntOp(f unIntFun) unOp {
    return func(a number) number {
        return number{i: f(a.Int())}
    }
}
```

Like `'^'`. (That's bitwise "not", BTW. If unary minus is "*all-zeros* `minus` n", unary `'^'` can well be "*all-ones* `xor` n".)

```
        newUnIntOp(func(a int) int { return ^a }),
```

stage 6: binary ops

Now let's do binary operators. `binOp.NewFun` and `newBinIntOp` are like their unary counterparts but with two arguments. We can use them to implement bitwise operators.

```
    lShiftOp = newBinIntOp(func(a, b int) int { return a << b })
    rShiftOp = newBinIntOp(func(a, b int) int { return a >> b })
    andOp    = newBinIntOp(func(a, b int) int { return a & b })
    bicOp    = newBinIntOp(func(a, b int) int { return a &^ b })
    orOp     = newBinIntOp(func(a, b int) int { return a | b })
```

Onto `newBinOp`.

stage 6: binary ops

Casting operands to the same type can be useful for many kinds of operators, so let's move it to its own function.

This function takes a `binOp` and wraps in a type casting. If the types of the operands don't match, one gets converted to `float64`.

```
func castToSame(f binOp) binOp {
    return func(a, b number) number {
        if a.isFloat != b.isFloat {
            if !a.isFloat {
                a = number{f: float64(a.i), isFloat: true}
            } else {
                b = number{f: float64(b.i), isFloat: true}
            }
        }
        return f(a, b)
    }
}
```

stage 6: binary ops

Then we can build our `binOp`.

```
func newBinOp(bif binIntFun, bff binFloatFun) binOp {
    return castToSame(func(a, b number) number {
        if a.isFloat {
            a.f = bff(a.f, b.f)
        } else {
            a.i = bif(a.i, b.i)
        }
        return a
    })
}
```

```
    addOp = newBinOp(
        func(a, b int) int { return a + b },
        func(a, b float64) float64 { return a + b },
    )
```

stage 6: division

The functions types we use, such as `binIntFun` (or `func(int, int) int`), don't return errors, so we can't implement `'/'` and `'%'` the same way we did before.

We could write a different NewFun, of course. Or we could wrap the denominator `fun` in a `fun` that will return `ErrZeroDivision` if the result is zero. Like this:

```
func (f fun) Denominator() fun {
    return func() (number, error) {
        n, err := f()
        if !n.Bool() && err == nil {
            err = ErrZeroDivision
        }
        return n, err
    }
}
```

stage 6: division

Other than that, a division op is the same as `binOp` and is constructed the same way.

```
type divModOp binOp

func newDivModOp(bif binIntFun, bff binFloatFun) divModOp {
    return divModOp(newBinOp(bif, bff))
}
```

Its NewFun just calls `binOp.NewFun` after wrapping the `right fun` in `Denominator`.

```
func (f divModOp) NewFun(left, right fun) fun {
    return binOp(f).NewFun(left, right.Denominator())
}
```

Now we can define division and modulo operators.

```
    modOp = newDivModOp(
        func(a, b int) int { return a % b },
        func(a, b float64) float64 { return math.Mod(a, b) },
    )
```

stage 6: wait, how does it work, again?

Ok, so that was too much. Let's see how a `divModOp` is constructed and run in detail.

So we call `newDivModOp` with two functions as arguments. These functions are anonymous, but for clarity let's pretend they and the generated closures have names.

```
func intMod(a, b int) int {
    return a % b
}

func floatMod(a, b float64) float64 {
    return math.Mod(a, b)
}
```

stage 6: constructing an op

newDivModOp calls newBinOp, which first wraps the two functions in a closure calling one according to the type of the argument.

```
func chooseMod(a, b number) number {
    if a.isFloat {
        a.f = intMod(a.f, b.f)
    } else {
        a.i = floatMod(a.i, b.i)
    }
    return a
}
```

stage 6: constructing an op

It then calls `castToSame` to wrap it in type casting, and returns the result.

```
func castMod(a, b number) number {
    if a.isFloat != b.isFloat {
        if !a.isFloat {
            a = number{f: float64(a.i), isFloat: true}
        } else {
            b = number{f: float64(b.i), isFloat: true}
        }
    }
    return chooseMod(a, b)
}
```

`newDivModOp` returns it as type `divModOp`.

```
    return divModOp(castMod)
```

stage 6: instantiating a fun

When NewFun is called for the modulo operator, it first calls `fun.Denominator` to wrap `right` in a closure.

```
func denominatorRight() (number, error) {
    n, err := right()
    if !n.Bool() && err == nil {
        err = ErrZeroDivision
    }
    return n, err
}
```

stage 6: instantiating a fun

It then calls `binOp.NewFun` to construct the closure that will actually run.

```
func runMod() (number, error) {
    a, err := left()
    if err != nil {
        return number{}, err
    }
    b, err := denominatorRight()
    if err != nil {
        return number{}, err
    }
    return castMod(a, b), nil
}
```

stage 6: running a fun

At runtime:

- `runMod` calls `left`.
- `runMod` calls `denominatorRight`.
    - `denominatorRight` calls `right`.
- `runMod` calls `castMod`.
    - `castMod` calls `chooseMod`.
        - `chooseMod` calls `intMod` or `floatMod`.

Quite a lot of code for computing a modulo.

If we ran the code repeatedly, it would make sense to track the types of numbers that the funs return and only convert them as necessary, avoiding `castMod` and `chooseMod`. But for a calculator the compile time – run time distinction doesn't matter.

stage 6: unary and binary

Let's continue.

Some operators like `'-'` can be both unary and binary, and at the lexing stage we don't know which it will be. It's basically two ops wrapped in one.

```
type multiOp struct {
    un, bin op
}
```

Thus, NewFun is where we'll have to decide which `fun` to retun. NewFun is called with `right` equal to `nil` for unary operators, so we'll check that.

```
func (f multiOp) NewFun(left, right fun) fun {
    if right == nil {
        return f.un.NewFun(left, nil)
    }
    return f.bin.NewFun(left, right)
}
```

stage 6: unary and binary

Now we can define `'-'` and `'^'`.

```
    subOp = multiOp{
        newUnOp(
            func(a int) int { return -a },
            func(a float64) float64 { return -a },
        ),
        newBinOp(
            func(a, b int) int { return a - b },
            func(a, b float64) float64 { return a - b },
        ),
    }
```

```
    xorOp    = multiOp{
        newUnIntOp(func(a int) int { return ^a }),
        newBinIntOp(func(a, b int) int { return a ^ b }),
    }
```

stage 6: opMap

Now that we have a bunch of ops, let's lex them.

We're going to have operators longer than one character (<=, &&), so let's change opMap accordingly and define some operators.

```
type opMap map[string]struct {
    typ int
    op  op
}
```

```
var ops = opMap{
    "+":   {'+', addOp},
    "-":   {'-', subOp},
    "*":   {'*', mulOp},
    "/":   {'/', divOp},
    "%":   {'%', modOp},
    "&":   {'&', andOp},
    "^":   {'^', xorOp},
    "&^":  {BIC, bicOp},
    "|":   {'|', orOp},
```

stage 6: opMap

Then we can write a function to find the longest operator and call it from `yyLex.nextToken`. If none is found, it will return a single-character tokens with op set to `nil`, so that we don't have to add tokens like `'('`, `')'` and `';'` to the map.

```
func (m opMap) find(s string) (token, int) {
    tlen := len(s)
    if tlen > 3 {
        tlen = 3
    }
    for tlen > 0 {
        if o, ok := m[s[:tlen]]; ok {
            return token{typ: o.typ, op: o.op}, tlen
        }
        tlen--
    }
    return token{typ: int(s[0])}, 1
}
```

```
    case strings.Index(bareTokens, s[:1]) != -1:
        tok, tlen = ops.find(s)
```

stage 6: assignments

To implement assignment, first we should have a function to assign the return value of a fun to a variable.

```
func (vl varMap) NewSet(s string, f fun) fun {
    return func() (number, error) {
        n, err := f()
        if err != nil {
            return number{}, err
        }
        vl[s] = n
        return n, nil
    }
}
```

We shoud distinguish between three kinds of assignments:

- simple assignment (`'='`)
- fancy assignment (+=, -=, etc.)
- post-increment and post-decrement (++, --)

stage 6: assignments

To implement fancy operators like +=, we should get the current value of the variable, run the + operator and assign the result to the variable. We can reuse the code for +.

++ adds 1 to the variable's value, so we'll make it an unary operator.

For = we don't get the variable's value or run a function, so we'll leave op as `nil`.

```
func NewAssign(lval string, op op, rval fun) fun {
    /*
     * Possibilities:
     * op == nil:    '=' operator, rval is the value
     * rval == nil:  "++" or "--", unOp.NewFun() uses left as the value
     * both non-nil: operator like "+=", uses Get(lval) and rval
     */
    if op != nil {
        rval = op.NewFun(runtime.vars.NewGet(lval), rval)
    }
    return runtime.vars.NewSet(lval, rval)
}
```

stage 6: assignments

Add them to the opMap. We don't have to add `'='`, as its op is `nil`.

```
    "+=":  {ADDEQ, addOp},
    "-=":  {SUBEQ, subOp},
    "*=":  {MULEQ, mulOp},
    "/=":  {DIVEQ, divOp},
    "%=":  {MODEQ, modOp},
    "&=":  {ANDEQ, andOp},
    "^=":  {XOREQ, xorOp},
    "&^=": {BICEQ, bicOp},
    "|=":  {OREQ, orOp},
    "<<=": {LSHIFTEQ, lShiftOp},
    ">>=": {RSHIFTEQ, rShiftOp},
    "++": {INC, newUnOp(
        func(a int) int { return a + 1 },
        func(a float64) float64 { return a + 1 },
    )},
    "--": {DEC, newUnOp(
        func(a int) int { return a - 1 },
        func(a float64) float64 { return a - 1 },
    )},
```

stage 6: assignments

Then we call NewAssign from the parser.

```
assign:
        IDENT assop expr        { $$ = NewAssign($1, $2, $3) }
|       IDENT postop            { $$ = NewAssign($1, $2, nil) }

assop:
        '=' | ADDEQ | SUBEQ | MULEQ | DIVEQ | MODEQ
|       ANDEQ | XOREQ | BICEQ | OREQ | LSHIFTEQ | RSHIFTEQ

postop: INC | DEC
```

stage 6: logic operators

How far can we push this framework?

Let's add short circuit logic. The way && works is:

- The left operand is run.
- If its result is `false`, the overall result is `false` and the right operand is skipped.
- If it's `true`, the overall result is the result of the right operand.

|| is the same, except with `true` and `false` reversed.

So a logic operator is essentially a boolean.

```
type logicOp bool

const (
    logicalOr  = logicOp(false)
    logicalAnd = logicOp(true)
)
```

stage 6: logic operators

It requires a custom NewFun.

```
func (cont logicOp) NewFun(left, right fun) fun {
    return func() (number, error) {
        a, err := left()
        if err != nil {
            return number{}, err
        }
        ans := a.Bool()
        if ans == bool(cont) {
            a, err = right()
            if err != nil {
                return number{}, err
            }
            ans = a.Bool()
        }
        return boolToNumber(ans), nil
    }
}
```

stage 6: comparison operators

Let's write comparison functions for == and `'<'`. `'>'` is the same as `'<'` but with operands reversed. We can't use `newBinOp` here because they return `int` for any operands, but the operands still have to be cast to the same type.

```
var (
    equalOp = castToSame(func(a, b number) number {
        if a.isFloat {
            return boolToNumber(a.f == b.f)
        }
        return boolToNumber(a.i == b.i)
    })
    lessOp = castToSame(func(a, b number) number {
        if a.isFloat {
            return boolToNumber(a.f < b.f)
        }
        return boolToNumber(a.i < b.i)
    })
    greaterOp binOp = func(a, b number) number {
        return lessOp(b, a)
    }
)
```

stage 6: comparison operators

We define a comparison operator as a bitfield.

```
type compareOp uint8

const (
    Equal = compareOp(1 << iota)
    Less
    Greater
)
```

Thus "<=" is `Less|Equal`, "!=" is `Less|Greater`, etc.

stage 6: comparison operators

Let's turn it into `binOp`.

`"!="` is the opposite of `"=="`, `"<="` of `'>'`, etc. So for a `compareOp` that has more than one bit set, we can run the opposite operator and negate the result.

```
func (f compareOp) BinOp() binOp {
    var (
        bf  binOp
        not bool
    )
    if (f & (f - 1)) != 0 {
        f ^= Equal | Less | Greater
        not = true
    }
```

To tell if a number has only one bit set, we clear the lowest set bit and compare the result to zero. Here is how it works:

```
  Equal = 1<<0 = binary 001        Greater = binary 100 = 4        Less | Greater = binary 110 = 6
   Less = 1<<1 = binary 010        4-1 = 3 = binary 011                   6-1 = 5 = binary 101
Greater = 1<<2 = binary 100          4 & 3 = binary 000 = 0               6 & 5 = binary 100 ≠ 0
```

stage 6: comparison operators

We then choose the appropriate function and negate it if needed.

```
    switch f {
    case Equal:
        bf = equalOp
    case Less:
        bf = lessOp
    case Greater:
        bf = greaterOp
    }
    if not {
        return func(a, b number) number {
            ans := bf(a, b)
            ans.i ^= 1
            return ans
        }
    }
    return bf
}
```

stage 6: comparison operators

To instantiate it, we run NewFun on the resulting `binOp`.

```
func (f compareOp) NewFun(left, right fun) fun {
    return f.BinOp().NewFun(left, right)
}
```

Now we can add them to the opMap.

```
    "<":   {'<', Less},
    ">":   {'>', Greater},
    "<=":  {LE, Less | Equal},
    ">=":  {GE, Greater | Equal},
    "==":  {EQ, Equal},
    "!=":  {NE, Less | Greater},
```

We run `compareOp.BinOp` every time NewFun is called, instead of once. It's not the most efficient way to do it, and I could build the ops with `(Less | Equal).BinOp()`, but that doesn't look as pretty and I think I've earned some fun.

stage 6: for loop

Is a `for` loop an op? Of course it is!

```
type forLoop struct{}

func (forLoop) NewFun(expr, block fun) fun {
    return func() (number, error) {
        for {
            if v, err := expr(); err != nil || !v.Bool() {
                return number{}, err
            }
            if _, err := block(); err != nil {
                return number{}, err
            }
        }
    }
}
```

stage 6: for loop

We just need to tokenize it as one.

```
        switch s[:tlen] {
        case "for":
            tok.typ = FOR
            tok.op = forLoop{}
        default:
            tok.typ = IDENT
        }
```

Define `list.NewFun` and we're ready to go.

```
func (l list) NewFun() fun {
    return func() (number, error) {
        return number{}, l.Run()
    }
}
```

```
|       FOR expr block          { $$ = $1.NewFun($2, $3.NewFun()) }
```

stage 6: for loop

This is only the short kind of `for` loop. How does the other work?

```
|       FOR stmt2 ';' expr ';' stmt2 block
```

- After running the `block`, we run the second `stmt2`. append can take care of that.

```
        {
                block := append($7, $6)
```

- With this, we can create the loop `fun`.

```
                loop := $1.NewFun($4, block.NewFun())
```

- Before the loop we run the first `stmt2`. This is just a list.

```
                $$ = list{$2, loop}
        }
```

stage 6: for loop

We can keep the result as a `list`.

A `block` is a `list` and can appear in the same place as a `for` loop, so we change the name in the `stmts` rule from `block` to `list`, and define the `list` rule.

```
stmts:
                                { }
|       stmts ';'
|       stmts stmt ';'           { $$ = append($1, $2) }
|       stmts list               { $$ = append($1, $2...) }

list:
        block
|       FOR stmt2 ';' expr ';' stmt2 block
        {
                $$ = list{$2, $1.NewFun($4, append($7, $6).NewFun())}
        }
```

The short `for` loop will stay under `stmt`.

stage 6: demo...

```
func main() {
    yyErrorVerbose = true
    if true {
        s := `
        for i = 0; i < 5; i++ {
            for j = -2; j != 0; j++ {
                i; j / 2.0
            }
        }
        `
        yy := newLexer(bytes.NewBufferString(s))
        yy.parse()
        return
    }
    yy := newLexer(os.Stdin)
    yy.parse()
}
```

Run

Thank you

Vadim Vygonets
August 2022