Parsing and interpretation with goyacc and closures

Vadim Vygonets
August 2022

First of all,

I don't know how to write slides (proof below). I only know how to write text.

Don't read this drivel. Just look at the code.

Unless you're not attending the lecture but only have the slides, in which case my incompetence is helpful, I guess.

1.1. hello there

What's up?

This is based on a parser/interpreter I once wrote for a s00per s1kr1t project, that had a simple grammar with nested parentheses. My goals with this presentation are:

- To introduce this nice trick for writing interpreters.
- To introduce you to parsing in general, and give you the names of some concepts and tools in case you choose to research this further.
- To make you consider the techniques of processing input more carefully.
- To make you fall in love with formal grammars.
- To show off.

If you'll leave this talk understanding grammars better and hating Noam Chomsky, I'll consider it a success.

What does the title mean?

What is parsing?

Parsing, or lexical analysis, is:

- receiving an input
- understanding what's up with it
    - presumably to do something useful with it later

If you've dealt with input, *any* input, you've done parsing.

And we should be careful doing that, you say?

Yes. Very.

Not processing input correctly is where many security issues come from.

> *A very large class of attacks against systems are really input validation attacks.*
>
> — Robert J. Hansen, Meredith L. Patterson
> *Guns and Butter: Towards Formal Axioms of Input Validation*
> Presented at the Black Hat conference USA, 2005

Patterson knows what she's talking about. She wrote an SQL firewall.

An ***SQL firewall***.

What is interpretation?

Running code without compiling it. What a shell does.

What are closures?

A closure is a function that captures variables from its environment.

Example: the second argument of `sort.Search` from the Go standard library is a callback function.

```
func Search(n int, f func(int) bool) int // from package "sort"
```

The variables a and k are present in the "environment" inside `findMeAnInt`.

```
// findMeAnInt returns the index of the first number >=k in the sorted array a.
func findMeAnInt(a []int, k int) int {
```

It calls `sort.Search` with a closure that captures a and k.

```
    return sort.Search(len(a),
        func(i int) bool {
            return k < a[i]
        })
}
```

Closures capture variables, not values

```
func clos(s string, i int) func() {
    return func() {
        fmt.Println(s, "=", i) // this closure captures s and i
        i++                    // it increments i each time it runs
    }
}

func main() {
    i := 42             // Each call to clos returns a new closure
    clos("i", i)()      // that has captured
    clos("i", i)()      // its own variables.
    f := clos("\tf", 5) // Therefore f and g
    g := clos("g", 23)  // will increment their
    f()                 // own integer variables
    f()                 // if run repeatedly.
    g()
    f()
    g()
    g()
    f()
}
```

Run

What is goyacc?

It's a yacc.

For Go.

What is Go?

You've got to be kidding me.

What is yacc?

yacc is a parser generator for Unix written in early 1970s by Stephen C. Johnson. It takes a description of a grammar and writes C code of a parser.

That is, it compiles grammar descriptions to C.

The parsers it generates are LALR (more on this later), which is the sort of parsers that are often used for parsing context-free grammars such as most sane programming languages (i.e., not C++). So they're used in compilers.

So one might say that yacc compiles compilers.

It's yet another one of the programs that do it.

Thus the name: Yet Another Compiler Compiler.

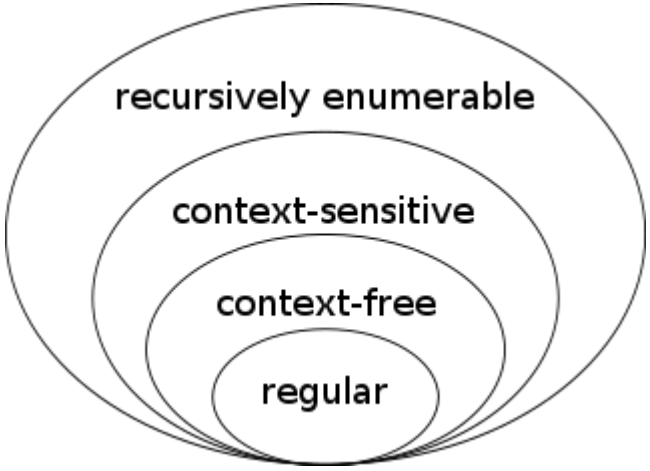
Як як?

Як як як.

Chomsky hierarchy of grammars

Chomsky hierarchy of grammars

Any piece of data has a grammar, and it better be formal. The hierarchy of grammars categorises them into 4 groups.

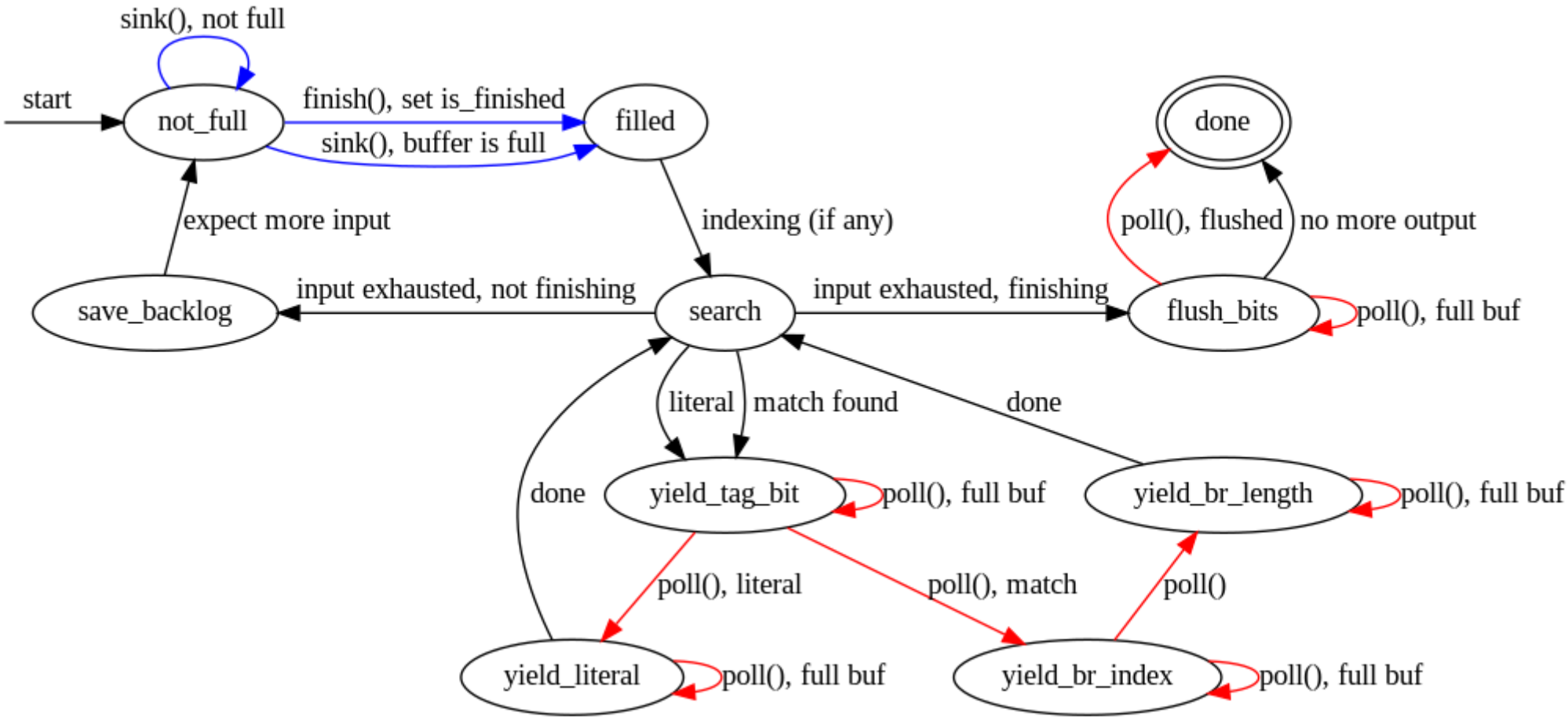


Chomsky-hierarchy.svg Copyright © 2010, J. Finkelstein, CC BY-SA 3.0

It's useful to view the simpler categories as subsets of the more complex ones, but I will present them from the simplest (regular) to the most complex (recursively enumerable).

But first, state machines!

State machine example: heatshrink encoder (Scott Vokes, 2013–2015)



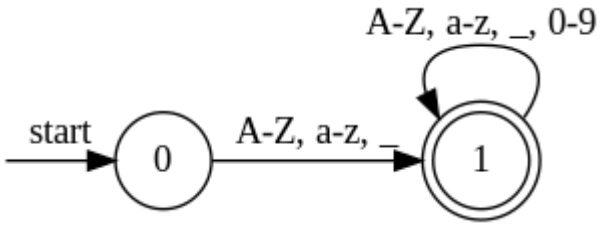

```
heatshrink embedded data compression library
Copyright © 2013-2021, Scott Vokes <vokes.s@gmail.com>
```

Regular languages

- A **Regular** grammar can be parsed by a finite state machine. Regular expression engines generate those.

Example: a C identifier that starts with an ASCII letter or an underscore, followed by zero or more ASCII letters, underscores or digits.

State machine (a double circle represents a terminal state):



Regexp: `[A-Za-z_][A-Za-z_0-9]*`

Context-free languages

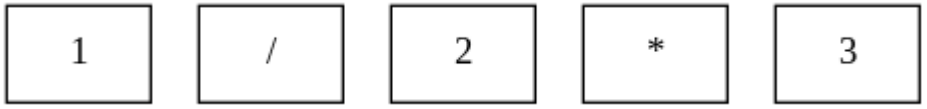
- A **Context-free** grammar can be parsed by a finite state machine with a stack. Technically, a "non-deterministic pushdown automation".

Example: Arithmetic expressions with parentheses.

Context-free grammars are often specified using BNF (Backus–Naur form), so that's what we'll use to describe this one.

```
<expression> ::= <expr1> | <expression> <op0> <expr1>
       <op0> ::= "+" | "-"
     <expr1> ::= <expr2> | <expr1> <op1> <expr2>
       <op1> ::= "*" | "/"
     <expr2> ::= <NUMBER> | "(" <expression> ")"
```

Let's look at this example in detail using the expression `1 / 2 * 3`.

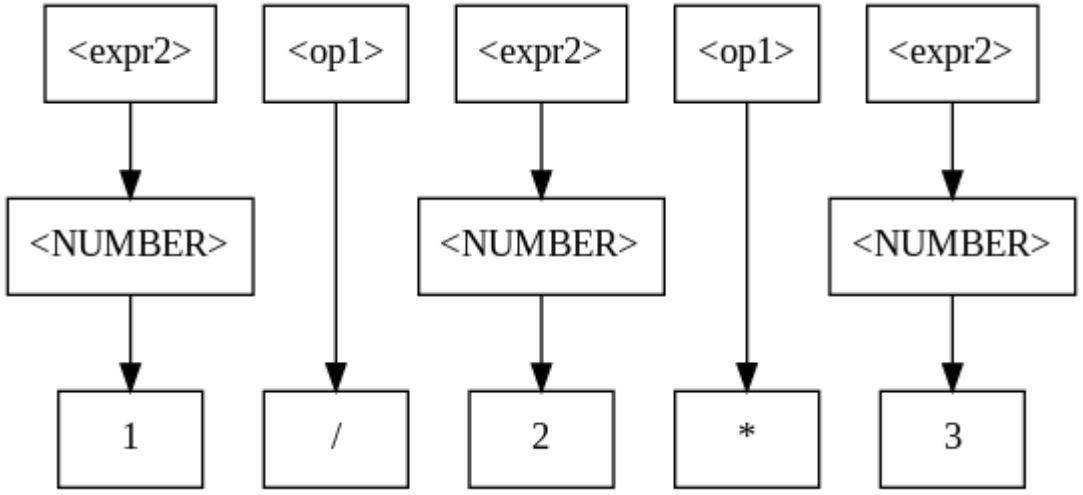


Context-free languages: example

An <op1> is either "*" or "/".

A level 2 expression <expr2> is either a <NUMBER> or a top level expression in parentheses. The latter variant makes the whole grammar recursive.

```
  <op1> ::= "*" | "/"
<expr2> ::= <NUMBER> | "(" <expression> ")"
```

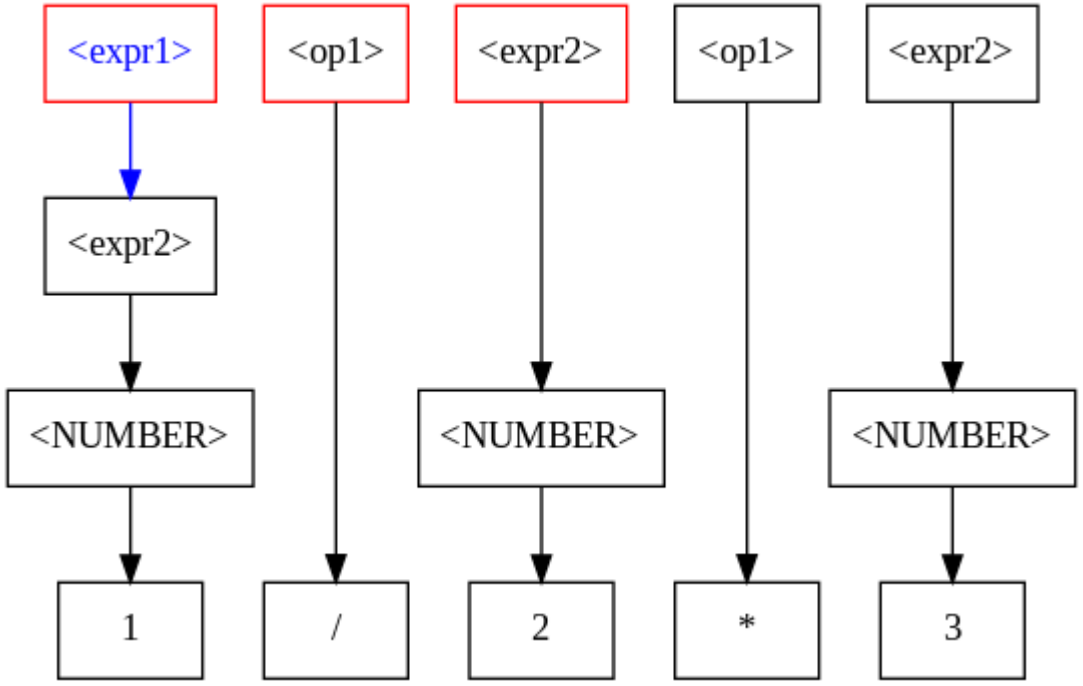


Context-free languages: example

An <expr1> is either an <expr2>, or an <expr1>, then an <op1>, then an <expr2>.

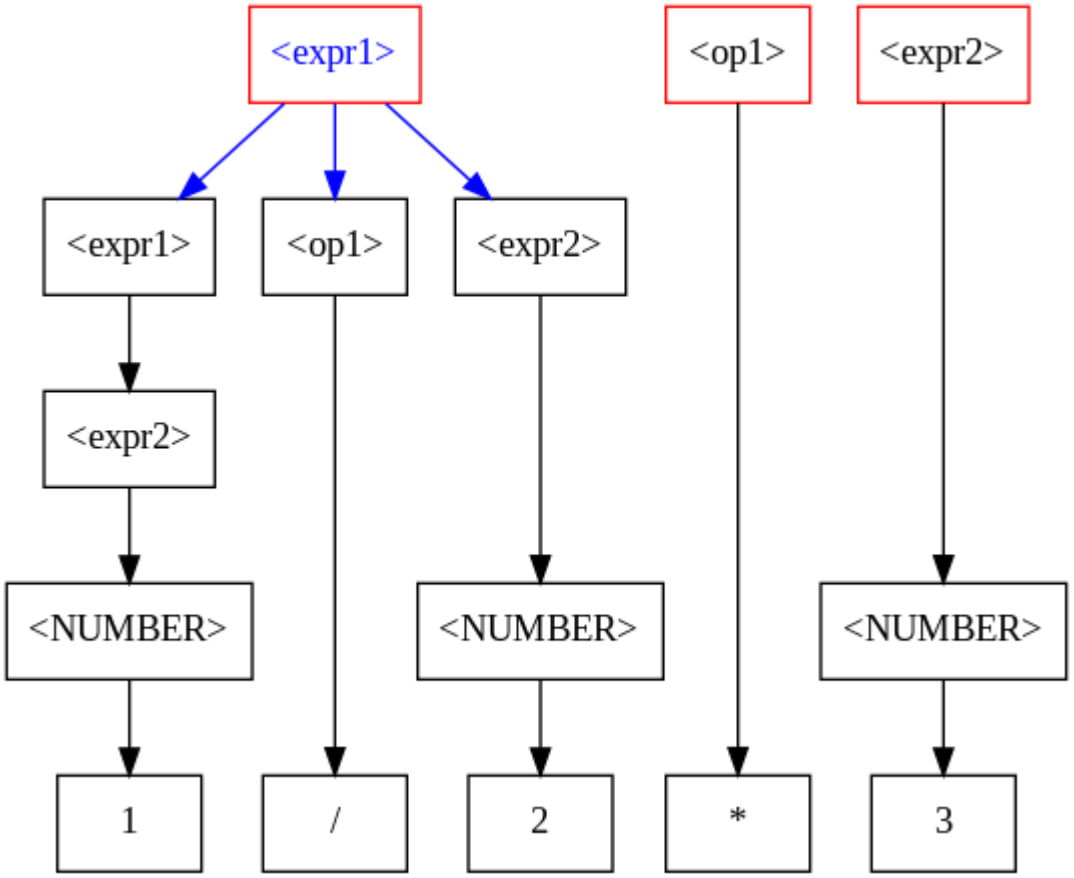
```
<expr1> ::= <expr2> | <expr1> <op1> <expr2>
```

Let's start real parsing by applying the first variant to the first token and painting it blue.



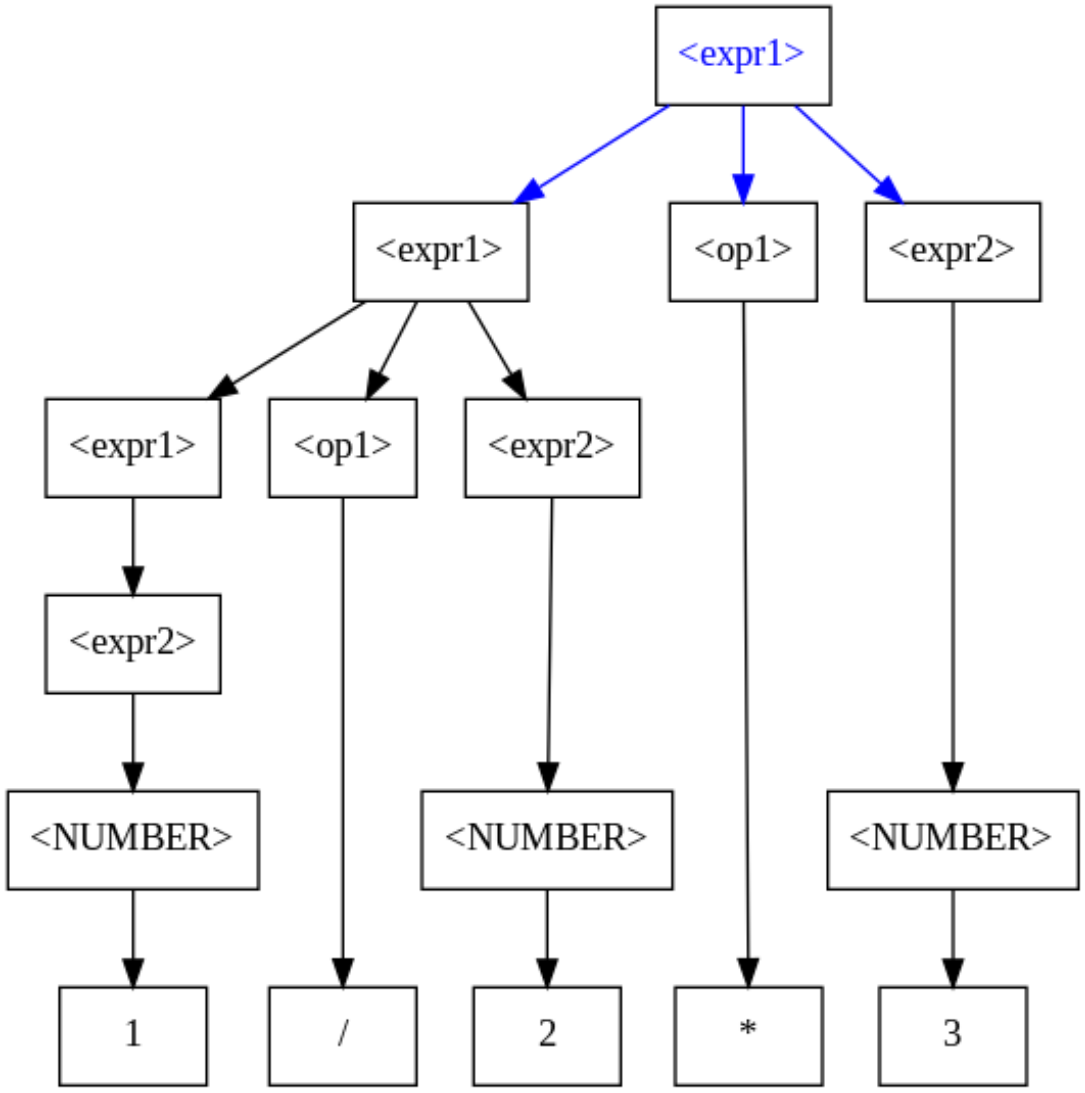
Looks like we can apply the second variant to the red boxes.

Context-free languages: example



And again.

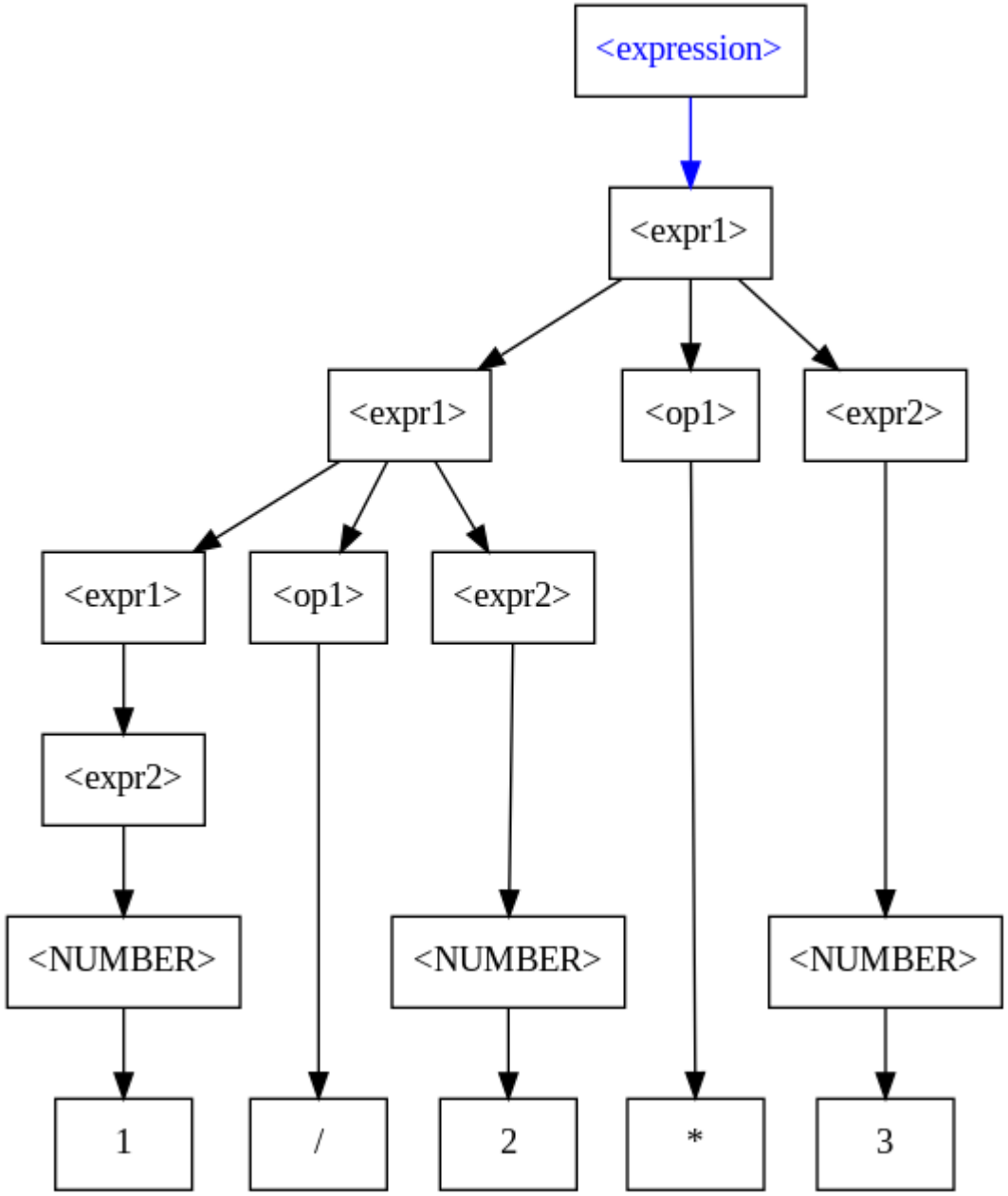
Context-free languages: example



Context-free languages: example

It's not an <expression> yet. So we have to apply this rule, and then we're really done.

```
<expression> ::= <expr1> | <expression> <op0> <expr1>
```


<expression>
<expr1>
<expr1>
<op1>
<expr2>
<expr1>
<op1>
<expr2>
<expr2>
<NUMBER>
<NUMBER>
<NUMBER>
1
/
2
*
3

Context-free languages: where's the state machine? and the stack?

yacc compiles grammar descriptions into those.

Context-sensitive languages

- A **Context-sensitive** grammar can be parsed by all of the above, plus some context, e.g., a dictionary.

Example: The ANSI C programming language, because of `typedef`. Before ANSI, a type declaration always started with a known keyword, like `int` or `unsigned` or `struct`. Now it can be anything, so a dictionary is needed.

Another example: IP packets, because they include a length field. *(Which is also an example of a formall grammar that is not text.)*

example: IPv4 packet- I mean, INTERNET DATAGRAM

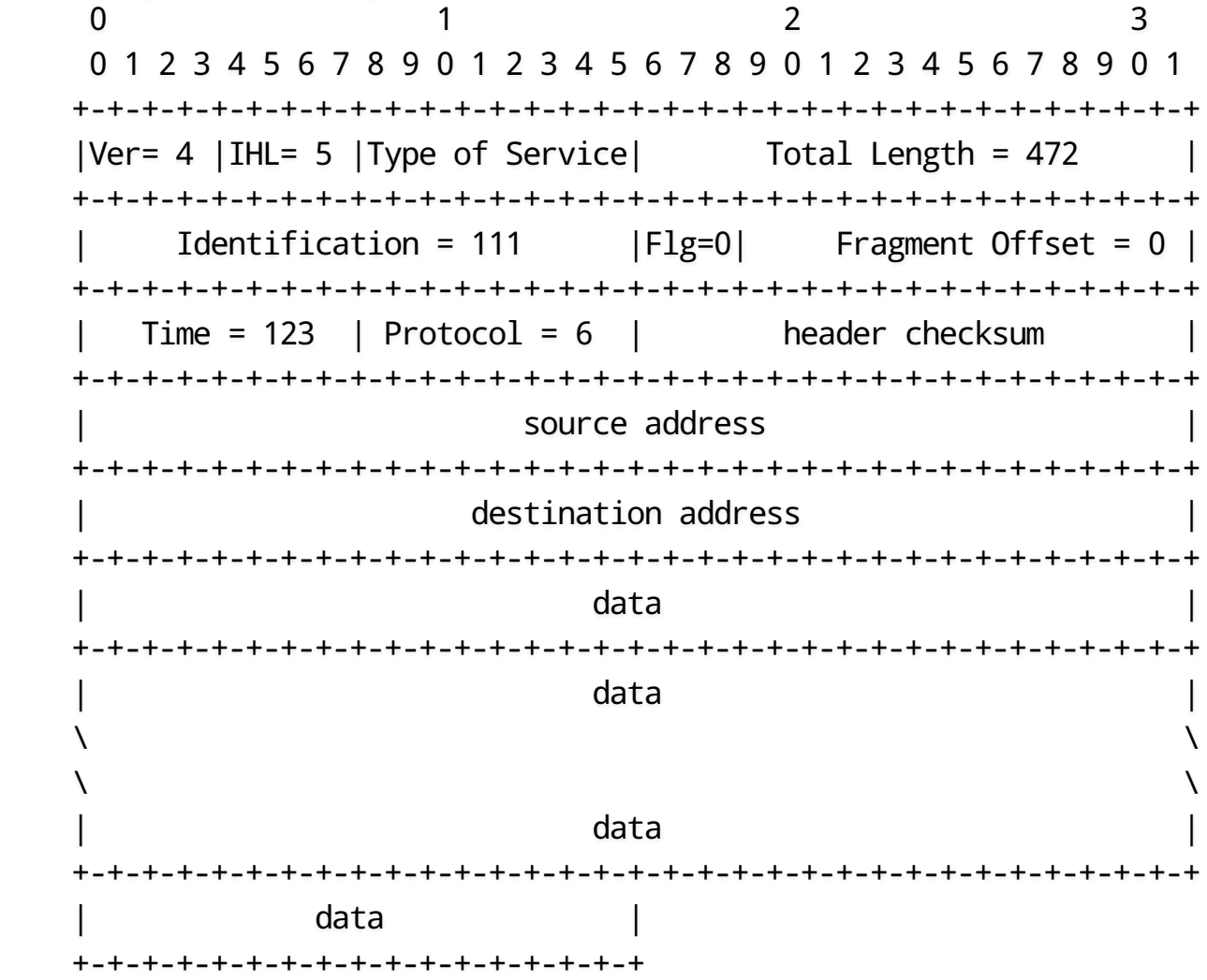
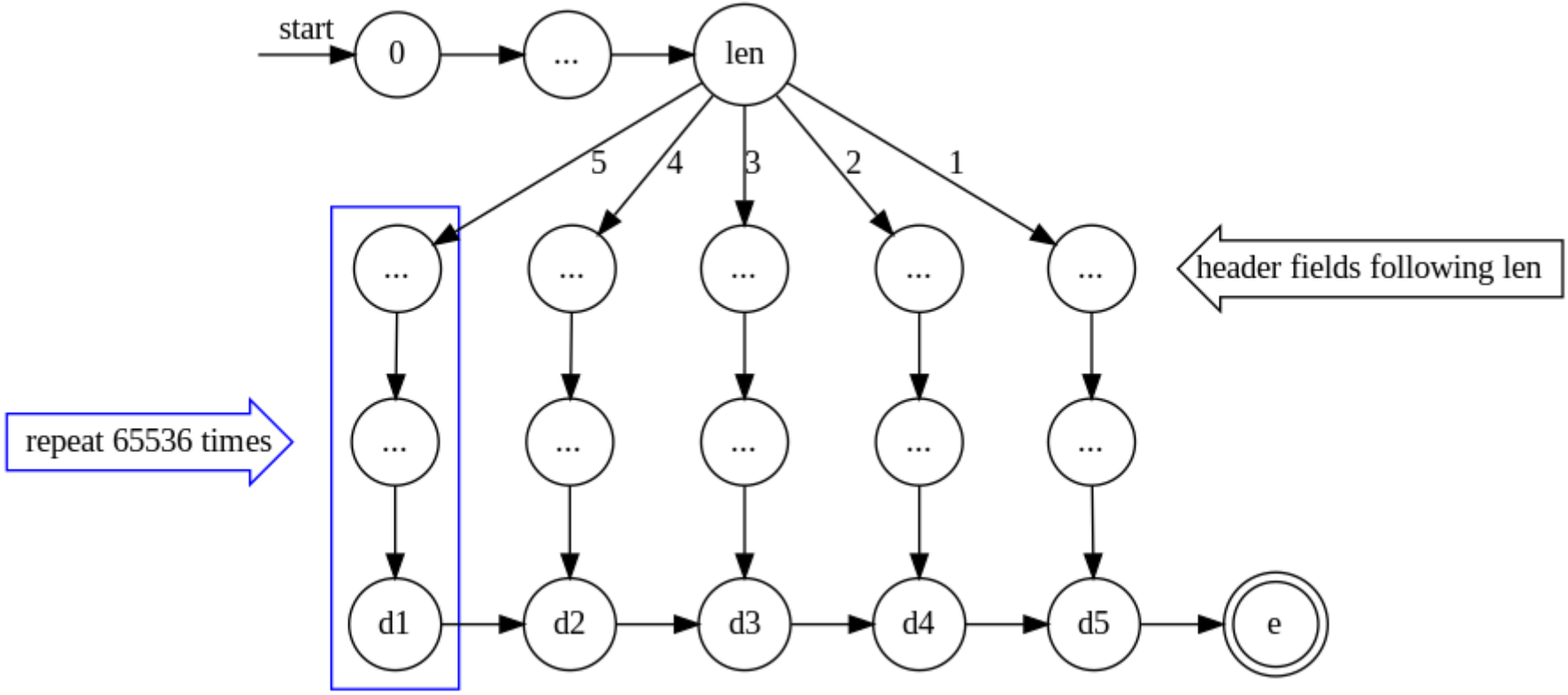


Figure 6: Example Internet Datagram. RFC 791: Internet Protocol, Jon Postel (Editor), 1981.

length field: context-sensitive?

Although, I guess, it could be made context-free by exploding the number of states, but this would be ridiculous.



Recursively enumerable languages

- A **Recursively enumerable** grammar can be parsed by a computer.

Example: C++ *[citation needed]*.

> *Outstandingly complicated grammar*
>
> "Outstandingly" should be interpreted literally, because *all popular languages* have context-free (or "nearly" context-free) grammars, while C++ has undecidable grammar.
>
> — Yossi Kreinin, *Defective C++*

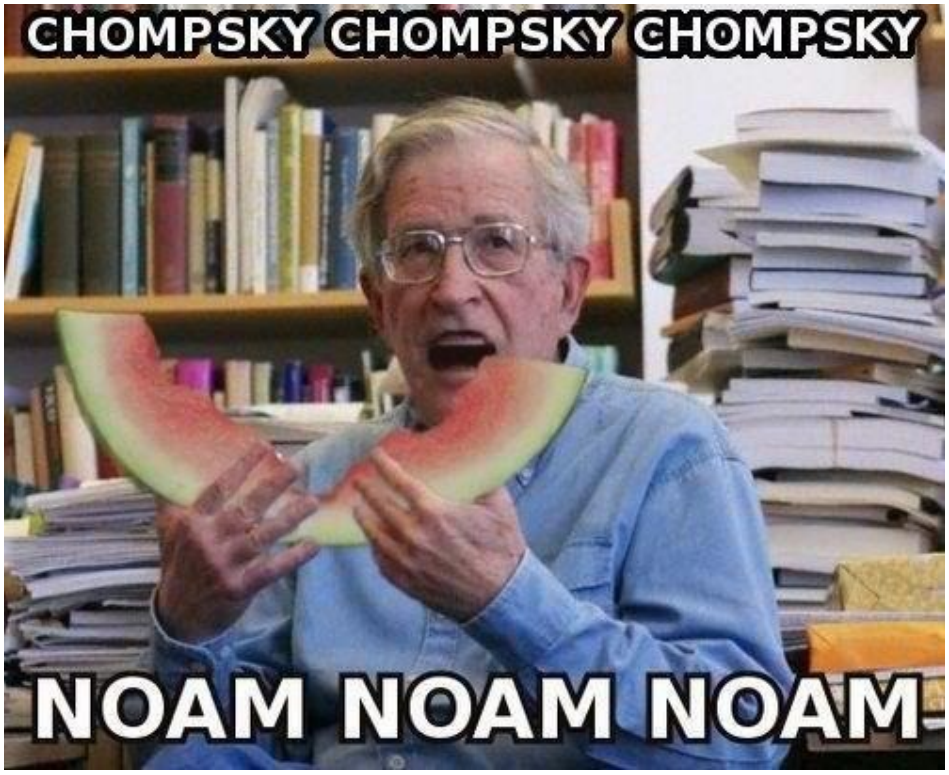
So who's this Chomsky guy? He sounds quite smart. I don't know why you want us to hate him.

To be honest, I don't know myself.

Prof. Noam Chomsky, born in 1928, is at MIT since 1955. His various achievements include pioneering the theories of Universal Grammar and Generative Linguistics. He is known as:

- The father of modern linguistics
- The Foremost Intellectual Of Our Time™
- The Consciousness of the Western Left
- Genocide denier

Noam Chomsky



Chomsky, who always emphasizes how one has to be empirical, accurate [...] I don't think I know a guy who was so often empirically wrong in his descriptions!

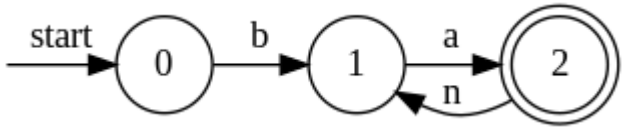
— Slavoj Žižek

Noam Chomsky

Turns out I'm a vindictive bastard with an axe to grind. Source: personal conversation with Professor *"Tempest"* (name changed) of psycholinguistics, 4 July 2022.

> Prof.: *I do believe one should separate his linguistic beliefs from his political ones.*
>
> Me: *well, yes*
>
> Prof.: *So perhaps making people in a grammar-related presentation dislike Chomsky for his political beliefs is not entirely fair?*
>
> Me: *perhaps even not at all fair*

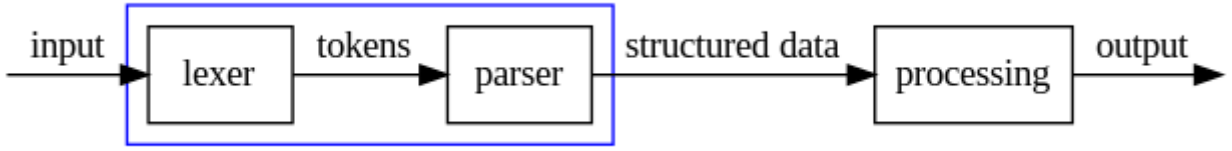
As a reward for your suffering, here's a state machine that consumes bananas.



Shall we write a parser then?

Let's write a parser for arithmetic expressions with `+`, `-`, `*`, `/`, `%` (a context-free grammar) and parentheses (making it recursive). It'll have variables.

Parsing context-free grammars is typically done in two stages.



The first stage, called a "lexer" or a "tokeniser", breaks a stream of characters into tokens. In a programming language these are symbols like `printf`, keywords like `if`, literals like 42 and `"hello, world\n"`, operators like && and +, brackets, commas, etc.

The second stage is a LALR parser that takes a stream of tokens, figures out its structure and delivers the data to the rest of the program for further processing.

- About that LALR thing...

I got 99 problems, but my parser's LALR(1)

Above I was like "let's apply this rule to these tokens, because I said so". But we could try something else and reach a dead end. That's non-deterministic pushdown automation for you, I guess. This is stupid, so it's not how real parsers work.

A parser generator generates a deterministic state machine that builds the tree as it consumes tokens. The kind of parser `yacc` generates is called LALR(1).

A LALR(1) parser is:

- ***L***ook-***A***head: consumes further tokens to resolve ambiguities in text already parsed.
- ***L***eft-to-Right: consumes input linearly with no backtracking.
- ***R***ightmost Derivation (left associativity): `a-b-c` means `(a-b)-c`, not `a-(b-c)`.
- ***1*** token of look-ahead.

Don't ask me about look-ahead, I don't understand it either. There's something called LR(1) which also does it, but allegedly the "LA" is not meaningless.

Stage 0: Parsing

stage 0: grammar

At stage 0 we will write a parser that consumes the text and does nothing.

Let's put our grammar definition in a `yacc` file `parse.y`, starting with some boilerplate.

goyacc generates a Go file. We put the prologue at the top, followed by the definition of the data structure holding the tokens (empty) and the types of tokens we have (numbers and identifiers), and mark the end of the section with %%.

```
%{
package main
%}

%union {
}

%token NUM
%token IDENT

%%
```

stage 0: grammar

The top level is `stmts` (a list of statements).

`stmts` is one of:

- nothing
- `stmts` followed by a semicolon, allowing for empty statements
- `stmts` followed by a statement and a semicolon

```
top:    stmts
stmts:  | stmts ';' | stmts stmt ';'
```

Statements end with semicolons. The lexer will inject fake semicolons at the end of each line.

This is the trick used in the Go lexer, which is why you don't end lines with semicolons in Go code. Except that our lexer will not be as nuanced and will always inject semicolons.

stage 0: grammar

A statement is an assignment or an expression.

```
stmt:   assign | expr
```

An assignment is a variable name, an `'='` and an expression.

```
assign: IDENT '=' expr
```

An `expr` is either a `expr2` or an addition or subtraction with `expr` on the left and `expr2` on the right. This makes rightmost derivation work.

`expr2` is similar, but is separate to give multiplication operators higher precedence.

```
expr:   expr2 | expr '+' expr2 | expr '-' expr2
expr2:  expr3 | expr2 '*' expr3 | expr2 '/' expr3 | expr2 '%' expr3
```

stage 0: grammar

`expr3` can be:

- a number,
- a variable name,
- a negation of `expr3`,
- a top level expression in parentheses.

```
expr3:  NUM | IDENT | '-' expr3 | '(' expr ')'
```

We end this section with %%. Could put some Go code below, but we won't.

```
%%
```

Done.

stage 0: grammar

We saw some single character tokens that weren't explicitly defined, like `'+'` and `';'`.

It's fine.

Now let's see the code.

stage 0: code - token

This magic comment tells `go generate` to run `goyacc`.

```
//go:generate goyacc parse.y
```

I'm lazy so I wrote a `Makefile`.

```
TARGET!=    basename `pwd`
INSTALLDIR=    ../go
GENTARGET=    ${INSTALLDIR}/${TARGET}.go
CLEANFILES=    y.go y.output
```

`package main`, blah blah. Here's a token. It has a type and a string.

```
type token struct {
    typ int
    s   string
}
```

stage 0: code - token

We can print it to see what the lexer does.

```
func (tok token) String() string {
    var tt string
    switch tok.typ {
    case 0:
        tt = "$end"
    case 1:
        tt = "$unk"
    case NUM:
        tt = "NUM"
    case IDENT:
        tt = "IDENT"
    default:
        tt = string(rune(tok.typ))
    }
    return fmt.Sprintf("%s %q", tt, tok.s)
}
```

stage 0: code - lexer: concurrency

Here's the lexer type, name courtesy of goyacc. The Lex function is called by the parser, This API interrupts our flow, so we send the tokens on the channel whenever we feel like it. We also print the tokens as we pass them to the parser.

```
type yyLex struct {
    c chan token
}
```

```
func (yy *yyLex) Lex(yylval *yySymType) int {
    tok := <-yy.c
    fmt.Println("token:", tok)
    return tok.typ
}

func (yy *yyLex) sendToken(tok token) {
    yy.c <- tok
}
```

stage 0: code - lexer: single character tokens

Next we need the tokeniser itself. `nextToken` will cut a token out of a string, and return the token and the rest of the string.

yacc has two predefined token types: 0 or `$end` for end of file, and 1 or `$unk` for unknown. Make the default an `$unk` token of length 1.

```
func (yy *yyLex) nextToken(s string) (token, string) {
    var (
        tok  = token{typ: 1}
        tlen = 1
    )
```

The type of a single character token is the character itself.

```
    const bareTokens = "=+-*/%();"
    switch {
    case strings.Index(bareTokens, s[:1]) != -1:
        tok.typ = int(s[0])
```

stage 0: code - lexer: numbers and identifiers

If the first character is a digit, scan the string until we run out of digits and declare it a number. Same with lowercase letters and identifiers.

```
    case s[0] >= '0' && s[0] <= '9':
        for tlen < len(s) && s[tlen] >= '0' && s[tlen] <= '9' {
            tlen++
        }
        tok.typ = NUM
    case s[0] >= 'a' && s[0] <= 'z':
        for tlen < len(s) && s[tlen] >= 'a' && s[tlen] <= 'z' {
            tlen++
        }
        tok.typ = IDENT
    }
```

No `default` in this `switch`, we're already set up for an unknown token of length 1. Now let's cut the token from the string up to `tlen`, trim space from the rest and return both.

```
    tok.s, s = s[:tlen], s[tlen:]
    return tok, strings.TrimSpace(s)
}
```

stage 0: code - lexer: main loop

Split input into lines. Trim space, then scan and send tokens until the line is empty.

```
func (yy *yyLex) run(in io.Reader) {
    var tok token
    sc := bufio.NewScanner(in)
    for sc.Scan() {
        s := strings.TrimSpace(sc.Text())
        for s != "" {
            tok, s = yy.nextToken(s)
            yy.sendToken(tok)
        }
```

Inject a fake semicolon at the end of each line, and $end at the end.

```
            yy.sendToken(token{typ: ';'})
        }
        if err := sc.Err(); err != nil {
            fmt.Fprintln(os.Stderr, err)
        }
        yy.sendToken(token{})
}
```

stage 0: code - putting it all together

The parser needs the lexer to have an error handler. Let's do it real quick.

```
func (yy *yyLex) Error(s string) {
    fmt.Fprintln(os.Stderr, s)
}
```

`main`: run the lexer in a goroutine and call the parser `yacc` generated for us.

```
func main() {
    yyErrorVerbose = true
    yy := yyLex{
        c: make(chan token),
    }
```

```
    go yy.run(os.Stdin)
    yyParse(&yy)
}
```

*...wait, what was that gap in the middle of `main`? — **NOTHING. THERE IS NO GAP IN `main`.***

Anyway, shall we try it?

stage 0: demo time!

```
func main() {
    yyErrorVerbose = true
    yy := yyLex{
        c: make(chan token),
    }
    if true {
        in := bytes.NewBufferString(`1 + 2 + 3
            a = 4+5/6`)
        go yy.run(in)
        fmt.Println("parser returned", yyParse(&yy))
        return
    }
    go yy.run(os.Stdin)
    yyParse(&yy)
}
```

Run

Stage 1: Printing the parse tree

stage 1: the parse tree

This was the actual next thing I did after writing the parser. Here's what it does.

```
func main() {
    yyErrorVerbose = true
    yy := yyLex{
        c: make(chan token),
    }
    if true {
        in := bytes.NewBufferString(`1 + 2 + 3
            a = 4+5/6`)
        go yy.run(in)
        fmt.Println("parser returned", yyParse(&yy))
        top.print()
        return
    }
    go yy.run(os.Stdin)
    yyParse(&yy)
    top.print()
}
```

Run

stage 1: data structures

Let's define a data structure representing a node in a binary tree. A node can hold a number or a string, and have up to two children. We'll call the node `tree` due to our stupidity.

```
type tree struct {
    typ   int
    n     int
    s     string
    left  *tree
    right *tree
}
```

We'll also have an array of `trees` (for `stmts`).

```
type list []*tree
```

And a variable holding the result.

```
var top list
```

stage 1: data types

We shall define the data types. `num` is for numbers, `word` is for identifiers, `tree` and `list` for the above.

```
%union {
        num  int
        word string
        tree *tree
        list list
}
```

Here are the types of results of parsing rules.

```
%type <num> num
%type <tree> stmt assign expr expr2 expr3 expr4 var
%type <list> top stmts
```

stage 1: trees

Now let's build a tree node for `'+'`.

```
expr:
        expr2
|       expr '+' expr2
        {
                $$ = &tree{
                        left:  $1,
                        typ:   '+',
                        right: $3,
                }
        }
```

This is the rule for `expr` that we saw in stage 0, but broken into lines and with a code block (the `{ ... }` thing) added.

If you know Go, you'll recognise that this code returns a pointer to a `tree` structure with three members initialised. But what's up with the dollars? And why no code block for the first alternative?

stage 1: BIGBUCKS$

$$ refers to the result, and numberbucks to successive parameters. These get substituted by yacc with references to its internal data.

```
  +------------------------------------ $$ (result)
  |       +---------------------------- $1 (1st param)
  |       |      +--------------------- $2 (2nd param)
  |       |      |       +------------- $3 (3rd param)
  v       |      |       |       +----- code block
expr:     v      |       |       |
        expr2    v       v       |
|       expr    '+'    expr2     |
        {                        v
                $$ = something($1, $3)
        }
```

If the code block is not present, the default action is:

```
        {
                $$ = $1
        }
```

stage 1: numbers

Let's define a `num` type for numbers.

```
%token <num> NUM
```

We can optimise the negation by negating the number while parsing, for which we will need the `num` rule.

```
%type <num> num
```

```
num:
        NUM
|       '-' num {
                $$ = -$2
        }
```

stage 1: numbers

Then we can put numbers (and variables) in the tree.

```
expr3:
        num
        {
                $$ = &tree{
                        typ: NUM,
                        n:   $1,
                }
        }
|       expr4
```

```
var:
        IDENT
        {
                $$ = &tree{
                        typ: IDENT,
                        s:   $1,
                }
        }
```

stage 1: lists

The rest of expression rules are the same, so let's deal with `stmts`.

- An empty `stmts` list does nothing. We don't initialise the slice, because Go is magic and append works on uninitialised slices. We do need an empty block, however, because there's no $1.
- A rule adding an empty statement does nothing.
- A rule adding an actual statement appends it to the list.

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

stage 1: lists

The rule for `top` assigns the list to the global variable.

```
top:
        stmts
        {
                top = $1
        }
```

So we're done with `yacc`, now we just need to change the lexer.

stage 1: lexer

We need the `token` to hold a number, ...

```
type token struct {
    typ int
    s   string
    n   int
}
```

... so that we can pass it to the parser. (We'll pass variable names too.)

```
func (yy *yyLex) Lex(yylval *yySymType) int {
    tok := <-yy.c
    yy.last = tok
    switch tok.typ {
    case NUM:
        yylval.num = tok.n
    case IDENT:
        yylval.word = tok.s
    }
    return tok.typ
}
```

stage 1: lexer

We'll add the code for parsing numbers to `nextToken`. In case parsing of the number fails, we'll print an error and return 1 ($unk).

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

stage 1: lexer

There was this line in Lex:

```
    yy.last = tok
```

We're saving the last token passed to the parser, so that if parsing fails, we can print the offending token.

```
type yyLex struct {
    c    chan token
    last token
}
```

```
func (yy *yyLex) Error(s string) {
    fmt.Fprintln(os.Stderr, s)
    fmt.Fprintln(os.Stderr, "last token:", yy.last)
}
```

Now we just need to add some boring code to print trees, which I won't show here.

Let's run it.

stage 1: demo time!

```
func main() {
    yyErrorVerbose = true
    yy := yyLex{
        c: make(chan token),
    }
    if true {
        in := bytes.NewBufferString(`1 + 2 + 3
            a = 4+5/6`)
        go yy.run(in)
        fmt.Println("parser returned", yyParse(&yy))
        top.print()
        return
    }
    go yy.run(os.Stdin)
    yyParse(&yy)
    top.print()
}
```

Run

Stage 2: Interpreter

stage 2: rationale

My s00per s1kr1t project had variables and needed to run code. After I could print the parse tree, I thought about how to write the interpreter, and concluded that the simplest thing to do would to assemble one from closures at parse time.

It will also be fast to run, as it's basically native compiled Go code.

- We're writing a shitty calculator, so a bare expression (without variable assignment) will just print the result.
- We will also not handle missing variables and division by zero gracefully at this stage.

    (That project didn't have the zero division issue. It did handle missing variables, but I don't remember how.)

Let's do it then.

stage 2: variables

We will need variables, so we'll store them in a map from `string` (variable name) to `int` (value). We don't handle errors, so reading a missing variable will return 0.

```
type varMap map[string]int

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

Our global state will have two things now.

```
var runtime = struct {
    top  list
    vars varMap
}{
    vars: make(varMap),
}
```

stage 2: lists

We don't have trees anymore. All we have is a list of functions returning `int`.

```
type list []func() int

func (l list) Run() {
    for _, v := range l {
        v()
    }
}
```

The only added line of code I haven't shown is this one at the end of `main`:

```
    runtime.top.Run()
```

Now that this is done, let's see the parser.

stage 2: parser

We'll need an `import` statement so that we can print division by zero errors.

```
%{
package main

import (
        "fmt"
        "os"
)
%}
```

We'll replace `tree` with a function returning int, which is `fun` (allegedly).

```
%union {
        num  int
        word string
        fun  func() int
        list list
}
```

```
%type <fun> stmt assign expr expr2 expr3 expr4 var
```

stage 2: numbers

Now onto the closures. We'll start with numbers. The recipe for `num` is unchanged, but here's how we turn a `num` into a `fun`:

```
expr3:
        num
        {
                n := $1
                $$ = func() int {
                        return n
                }
        }
|       expr4
```

Seems simple enough, right? We create an anonymous function that returns an `int`. But why not return just `$1` instead?

stage 2: numbers

Here's the code goyacc generates for `return $1`:

```
    case 15:
        yyDollar = yyS[yypt-1 : yypt+1]
        {
            yyVAL.fun = func() int {
                return yyDollar[1].num
            }
        }
```

This closure captures `yyDollar`, a parser's internal variable. Our code creates a variable with the correct value that the closure captures:

```
    case 15:
        yyDollar = yyS[yypt-1 : yypt+1]
        {
            n := yyDollar[1].num
            yyVAL.fun = func() int {
                return n
            }
        }
```

stage 2: variables

Reading a variable is just a matter of calling `Get`, and writing is as simple.

```
var:
        IDENT
        {
                s := $1
                $$ = func() int {
                        return runtime.vars.Get(s)
                }
        }
```

```
assign:
        IDENT '=' expr
        {
                s, a := $1, $3
                $$ = func() int {
                        runtime.vars[s] = a()
                        return 0
                }
        }
```

stage 2: arithmetics

Let's see how we implement addition.

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

Other operators are the same, except division that is more elaborate:

```
                $$ = func() int {
                        x, y := a(), b()
                        if y == 0 {
                                fmt.Fprintln(os.Stderr, "division by zero")
                                return 0
                        }
                        return x / y
                }
```

Shitty error handling.

stage 2: bare expressions

Now let's make bare expressions print their result. We'll put this code in `stmt: expr`.

```
stmt:
        assign
|       expr
        {
                a := $1
                $$ = func() int {
                        fmt.Println(a())
                        return 0
                }
        }
```

Change top to `runtime.top` and we're done.

```
top:
        stmts
        {
                runtime.top = $1
        }
```

Let's run it.

stage 2: demo time!

```
func main() {
    yyErrorVerbose = true
    yy := yyLex{
        c: make(chan token),
    }
    if true {
        in := bytes.NewBufferString(`1 + 2 + 3
            a = 4+5/6
            a`)
        go yy.run(in)
        fmt.Println("parser returned", yyParse(&yy))
        runtime.top.Run()
        return
    }
    go yy.run(os.Stdin)
    yyParse(&yy)
    runtime.top.Run()
}
```

Run

Stage 3: Just for fun

stage 3: for loops

IRL at this stage I was done. But I wanted to show you how easy it is to extend this, and I had an idea.

`for` loops.

Let's turn this shitty non-interactive calculator into a shitty programming language interpreter.

stage 3: token

First, let's introduce another token.

```
%token <word> IDENT FOR
```

Now let's add braces to `bareTokens` and keyword detection to `nextToken`:

```
    const bareTokens = "=+-*/%(){};"
```

```
    case s[0] >= 'a' && s[0] <= 'z':
        for tlen < len(s) && s[tlen] >= 'a' && s[tlen] <= 'z' {
            tlen++
        }
        switch s[:tlen] {
        case "for":
            tok.typ = FOR
        default:
            tok.typ = IDENT
        }
```

Now we're done with *that* file. Back to the parser.

stage 3: block

We'll need blocks. A block is a list of statements surrounded by curly braces.

```
block:
        '{' stmts '}'
```

What it does is run the statements. Hey, we already have a function for that.

```
        {
                a := $2
                $$ = func() int {
                        a.Run()
                        return 0
                }
        }
```

stage 3: statements

We want to have Go-like for loops, which come in three varieties:

```
for stmt; expr; stmt { code } // normal for loop
for expr { code }             // like while loop in C
for { code }                  // infinite loop
```

We don't have `break`, `return` or `exit`, so we're only interested in the first two.

Ok, so a block is a statement. But we don't want it to appear *anywhere* statements can go. In particular, the statements in that `for stmt; expr; stmt` thing should not be blocks (or `for` loops).

So now we have a statement hierarchy. Great.

```
stmt:
        stmt2
|       block
|       forloop

stmt2:
        assign
|       expr
```

stage 3: loop

We don't have booleans, so we'll do it the C way: zero is false, non-zero is true.

```
forloop:
        FOR expr block
        {
                a, b := $2, $3
                $$ = func() int {
                        for a() != 0 {
                                b()
                        }
                        return 0
                }
        }
|       FOR stmt2 ';' expr ';' stmt2 block
        {
                a, b, c, d := $2, $4, $6, $7
                $$ = func() int {
                        for a(); b() != 0; c() {
                                d()
                        }
                        return 0
                }
        }
```

stage 3: parser

Now we only need to add those rules to the `fun` type and we're done.

```
%type <fun> stmt stmt2 block forloop assign expr expr2 expr3 expr4 var
```

We don't have logical operators, we don't even have comparisons. We can still test it though. Let's do it.

stage 3: demo time!

```
func main() {
    yyErrorVerbose = true
    yy := yyLex{
        c: make(chan token),
    }
    if true {
        s := `
        for i = 0; i - 5; i = i + 1 {
            for j = -2; j; j = j + 1 {
                i; j
            }
        }
        `
        go yy.run(bytes.NewBufferString(s))
        fmt.Println("parser returned", yyParse(&yy))
        runtime.top.Run()
        return
    }
    go yy.run(os.Stdin)
    yyParse(&yy)
    runtime.top.Run()
}
```

Run

Things to come

stage 4: Error handling

Error propagation: execution stops on errors.

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

```
        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))
```

Run

stage 5: Interactive calculator

```
        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 5: Fernschreibmaschine mit Telefonanschluss???

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

stage 6: Floats and more operators

Floating point, automatic `int` ⇄ `float64` conversion

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

Shorter operator definitions, reused for assighments

```
    divOp = newDivModOp(
        func(a, b int) int { return a / b },
        func(a, b float64) float64 { return a / b },
    )
```

```
    "/":   {'/', divOp},
```

```
    "/=":  {DIVEQ, divOp},
```

stage 6: Unary ops, integer-only ops, comparison ops, logic ops, etc.

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

```
    "==":  {EQ, Equal},
    "!=":  {NE, Less | Greater},
    "&&":  {LAND, logicalAnd},
```

Easy to use!

```
expr5:
        expr6
|       expr5 op5 expr6         { $$ = $2.NewFun($1, $3) }

op5:    '*' | '/' | '%' | '&' | BIC | LSHIFT | RSHIFT
```

stage 6: Fancy assignments

```
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
```

```
    "++": {INC, newUnOp(
        func(a int) int { return a + 1 },
        func(a float64) float64 { return a + 1 },
    )},
```

stage 6: With the magic of interfaces, for loop is an operator

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

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

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

```
        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
```

Run

Thank you

Vadim Vygonets
August 2022