Writing a parser in Go, the C way

1. Write some grammar rules.

2. Write the rest of the parser.

Tobias Guggenmos
@slrtbtfs

About me

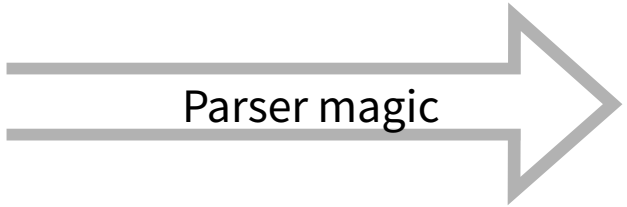
- Software Engineering Intern @ Red Hat
- OpenShift Monitoring Team
- Working on Prometheus and the Prometheus Language Server
- Math student in Ulm

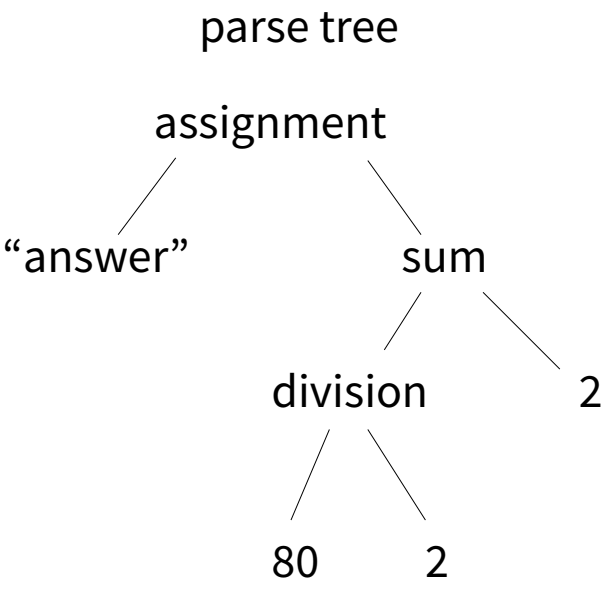
Tobias Guggenmos
@slrtbtfs

What is a parser?

String

```
answer = 80/2 + 2
```





Tobias Guggenmos
@slrtbtfs

What is YACC?

Formal grammar

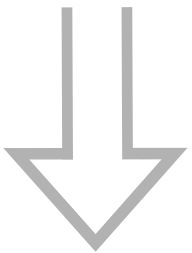
```
start: expr {
        if !yylex.(*interpreter).evaluationFailed{
                fmt.Println($1)
        }}
     | assignment;

expr:
      NUMBER {
        var err error
        $$, err = strconv.ParseFloat($1, 64)
        if err != nil{
                yylex.Error(err.Error())
        }
        }
    | IDENTIFIER {
        var ok bool
        $$, ok = yylex.(*interpreter).vars[$1]
        if !ok {
                yylex.Error(fmt.Sprintf("Variable
undefined: %s\n", $1))
        }
        }
    | expr '+' expr { $$ = $1 + $3 }
    | expr '-' expr { $$ = $1 - $3 }
    | expr '*' expr { $$ = $1 * $3 }
    | expr '/' expr { $$ = $1 / $3 }
    | '(' expr ')'  { $$ = $2 }
    | '-' expr %prec '*' { $$ = -$2 }
    ;

assignment:
          IDENTIFIER '=' expr {
                if !yylex.
(*interpreter).evaluationFailed {
                        yylex.
(*interpreter).vars[$1] = $3
                }};
```

A parser, written in go

Tobias Guggenmos
@slrtbtfs

Step 1: Lexer

```
answer = 80/2 + 2
```



Identifier: answer	=	Number: 80	/	Number: 2	+	Number: 2

Tobias Guggenmos
@slrtbtfs

Step 1: Lexer

```
answer = 80/2 + 2
```

Concept:

```
switch ... {
case <space>:          // ignore.
case <is number>:      return NUMBER
case <is identifer>: return IDENTIFIER
default:               return <next letter>
}
```

Problem:

How are numbers and identifiers recognized?

Tobias Guggenmos
@slrtbtfs

Step 1: Lexer

```
answer = 80/2 + 2
```

Solution: Regular Expressions

```
var tokens = []tokenDef{
    tokenDef{
        regex: regexp.MustCompile(`^[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?`),
        token: NUMBER,
    },
    tokenDef{
        regex: regexp.MustCompile(`^[_a-zA-Z][_a-zA-Z0-9]*`),
        token: IDENTIFIER,
    },
}
```

Tobias Guggenmos
@slrtbtfs

Step 1: Lexer

```
answer = 80/2 + 2
```

Actual lexer code:

```
// Skip spaces.
for ; len(l.input) > 0 && isSpace(l.input[0]); l.input = l.input[1:] {
}

// Check if the input has ended.
if len(l.input) == 0 {
    return EOF
}
```

Tobias Guggenmos
@slrtbtfs

Step 1: Lexer

```
answer = 80/2 + 2
```

Actual lexer code:

```
// Check if one of the regular expressions matches.
for _, tokDef := range tokens {
    str := tokDef.regex.FindString(l.input)
    if str != "" {
        // Pass string content to the parser.
        ...
        return tokDef.token
    }
}


// Otherwise return the next letter.
ret := int(l.input[0])
l.input = l.input[1:]
return ret
```

Tobias Guggenmos
@slrtbtfs

Step 1: Lexer

At the end we have to implement this interface:

```
type yyLexer interface {
    Lex(lval *yySymType) int
    Error(s string)
}
```

yySymType is generated from the following statement in the grammar file:

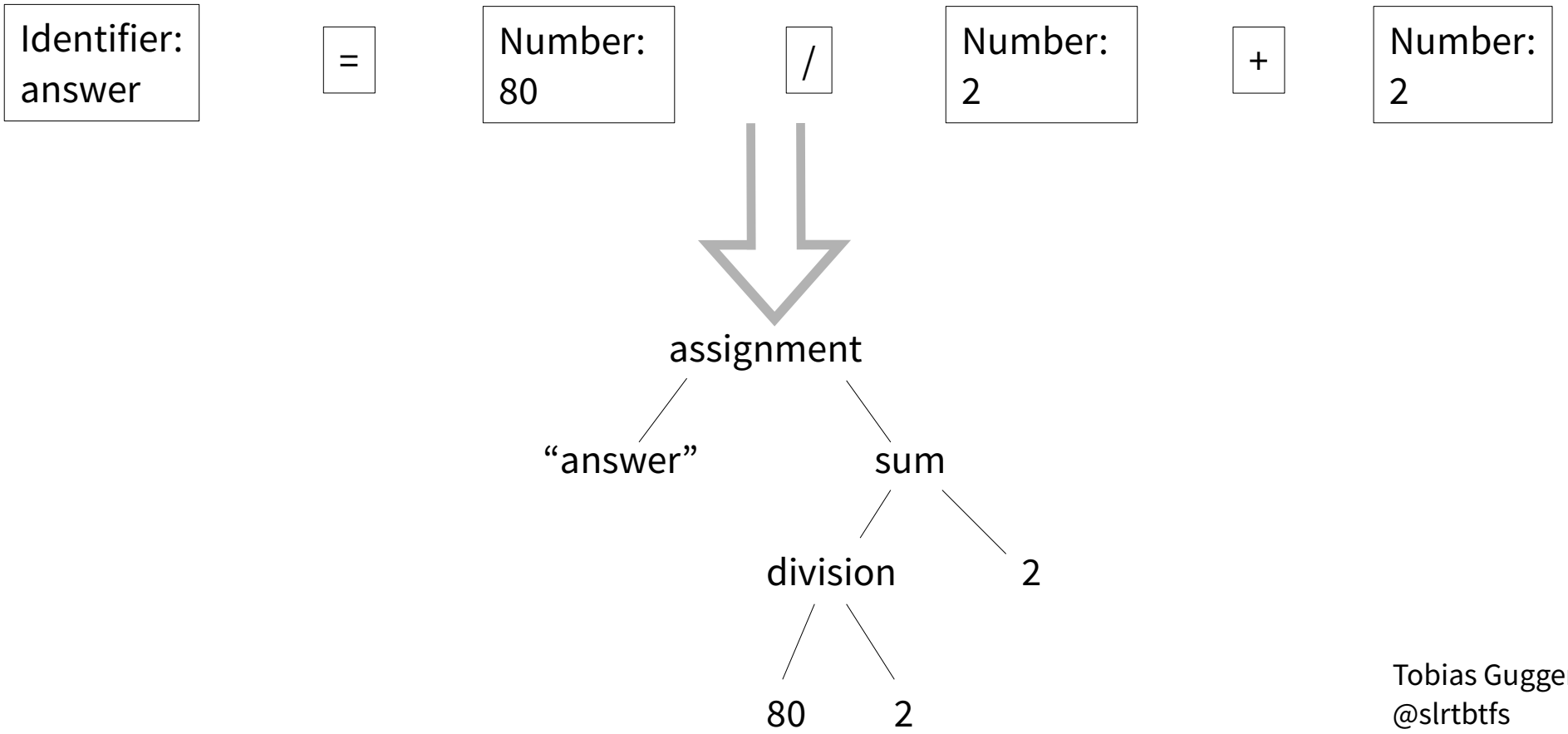
```
%union {
String string
Number float64
}
```

Which is transpiled to:

```
type yySymType struct {
    yys    int
    String string
    Number float64
}
```

Tobias Guggenmos
@slrtbtfs

Step 2: Parsing



Tobias Guggenmos
@slrtbtfs

Step 2: Parsing

2.1. Write a formal EBNF grammar:

```
start: expr
     | assignment;

expr:
      NUMBER
    | IDENTIFIER
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | '(' expr ')'
    | '-' expr
    ;

assignment:
          IDENTIFIER '=' expr;
```

Tobias Guggenmos
@slrtbtfs

Step 2: Parsing

2.2. Add actions to the grammar:

```
%token<String> NUMBER IDENTIFIER
%type <Number> expr
```

```
start: expr          { fmt.Println($1) }
     | assignment;

expr:
      NUMBER         { $$ = strconv.ParseFloat($1)// + error handling}
    | IDENTIFIER     { $$ = // lookup variable }
    | expr '+' expr  { $$ = $1 + $3 }
    | expr '-' expr  { $$ = $1 - $3 }
    | expr '*' expr  { $$ = $1 * $3 }
    | expr '/' expr  { $$ = $1 / $3 }
    | '(' expr ')'   { $$ = $2 }
    | '-' expr       { $$ = -$2 }
    ;

assignment:
          IDENTIFIER '=' expr;
```

Tobias Guggenmos
@slrtbtfs

Step 2: Parsing

2.3. Run the interpreter:

```
$ goyacc grammar.y
22 shift/reduce conflicts

$ go build .

$ ./calulator

> 1 + 2 * 3
7
> 2 * 3 + 1
8
```

Tobias Guggenmos
@slrtbtfs

Step 2: Parsing

2.4. Fix the grammar:

```
%left '+' '-'
%left '*' '/'

...
expr:
...
    | '-' expr %prec '*' { $$ = -$2 }
...
```

Tobias Guggenmos
@slrtbtfs

Step 2: Parsing

2.5. Make variable handling work:

```
| IDENTIFIER {
        var ok bool
        $$, ok = yylex.(*interpreter).vars[$1]
        if !ok {
                yylex.Error(fmt.Sprintf("Variable undefined: %s\n", $1))
        }
        }
...
IDENTIFIER '=' expr {
                if !yylex.(*interpreter).evaluationFailed {
                        yylex.(*interpreter).vars[$1] = $3
                }};
```

Tobias Guggenmos
@slrtbtfs

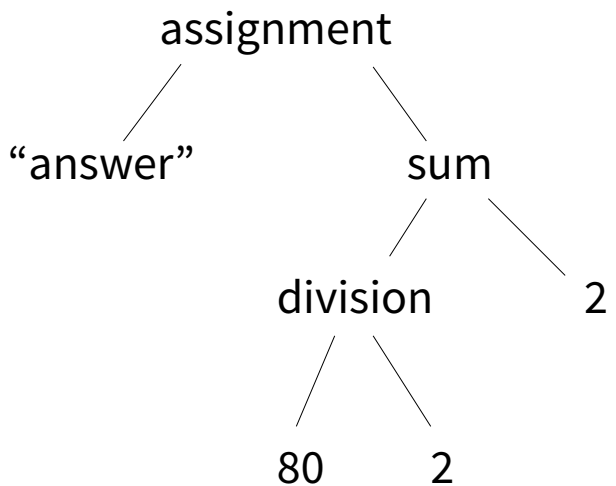
Step 2: Parsing

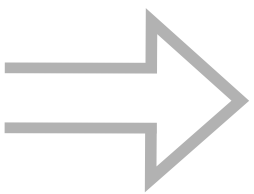
[2.6. Actually use a syntax tree:]

```
start: expr {yylex.(*interpreter).parseResult = &astRoot{$1}}
     | assignment {yylex.(*interpreter).parseResult = $1};

expr:
      NUMBER {$$ = &number{$1} }
    | IDENTIFIER { $$ = &variable{$1}}
    | expr '+' expr { $$ = &binaryExpr{Op: '+', lhs: $1, rhs: $3} }
    | expr '-' expr { $$ = &binaryExpr{Op: '-', lhs: $1, rhs: $3} }
    | expr '*' expr { $$ = &binaryExpr{Op: '*', lhs: $1, rhs: $3} }
    | expr '/' expr { $$ = &binaryExpr{Op: '/', lhs: $1, rhs: $3} }
    | '(' expr ')'  { $$ = &parenExpr{$2}}
    | '-' expr %prec '*' { $$ = &unaryExpr{$2} }

    ;

assignment:
          IDENTIFIER '=' expr {$$ = &assignment{$1, $3}};
```

Tobias Guggenmos
@slrtbtfs

[Step 3: Evaluating]





```
answer = 42
```

Tobias Guggenmos
@slrtbtfs