



Parsing & Go

@quasilyte 2021

ПАРСИТЬ?

Кому вообще это нужно?

Всю жизнь парсил, продолжаю парсить и буду парсить

Database firewall (SQL parsing)	Data Armor
Go компилятор и ассемблер	Intel
NoVerify линтер	VK
KPHP компилятор	VK

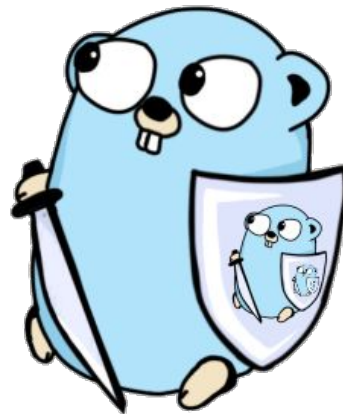
А ещё был open source



[regexp-lint](#)



[go-critic](#)



[go-ruleguard](#)



[phpgrep](#)

NoVerify (Go)

Вручную написанный парсер

Генератор FSM Ragel

Генератор парсеров goyacc

Что из этого мы используем?

NoVerify (Go)

- ✓ Вручную написанный парсер
- ✓ Генератор FSM Ragel
- ✓ Генератор парсеров goyacc

Всё!

КРНР (C++)

Вручную написанный парсер

Генератор парсеров bison

Генератор лексеров lex

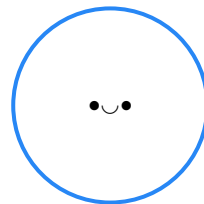
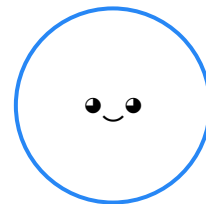
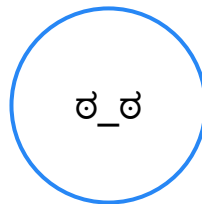
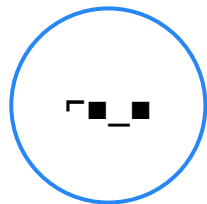
Что из этого мы используем?

КРНР (C++)

- ✓ Вручную написанный парсер
- ✓ Генератор парсеров bison
- ✓ Генератор лексеров lex

Всё!

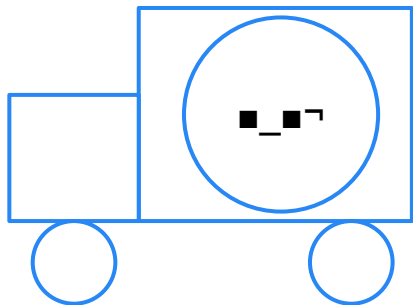
История из нашей команды



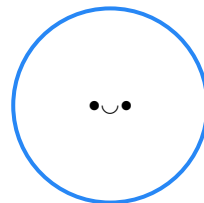
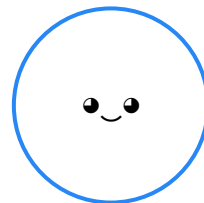
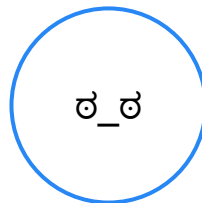
Автор NoVerify

История из нашей команды

Google



Автор NoVerify



История из нашей команды

NoVerify

ಠ_ಠ

😊

😊

Что мы будем делать

- Убедимся, что парсинг - не вымышленная задача
- Разберём несколько способов парсинга
- Рассмотрим типичные проблемы и их решения

Чего здесь не будет

- Введения в лексический анализ и парсинг
- Подробнейшего разбора какого-то из инструментов
- Академической точности

Что мы будем делать



Убедимся, что парсинг - не вымышленная задача

- Разберём несколько способов парсинга
- Рассмотрим типичные проблемы и их решения

Чего здесь не будет

- Введения в лексический анализ и парсинг
- Подробнейшего разбора какого-то из инструментов
- Академической точности

Что мы будем делать



Убедимся, что парсинг - не вымышленная задача

- Разберём несколько способов парсинга

- Рассмотрим типичные проблемы и их решения

Чего здесь не будет

- Введения в лексический анализ

Глубокого разбора какого-то из инструментов

- Академической точности

А ещё будут бенчмарки!



А что будем парсить?

Типы внутри phpdoc комментариев

```
/**  
 * @return ?int|void|string[]  
 */  
function check_rights() {  
    return false;  
}
```


А зачем это парсить?



сложно анализировать

легко анализировать

Выражения типов внутри phpdoc

<code>int, float, null</code>	primitive type
<code>Foo, Foo\Bar</code>	[qualified] type name
<code>?T</code>	nullable type
<code>T?</code>	optional key type
<code>T[]</code>	array type
<code>X Y</code>	union type
<code>X&Y</code>	intersection type

Выражения типов внутри phpdoc

<code>int, float, null</code>	primitive type
<code>Foo, Foo\Bar</code>	[qualified] name
<code>?T</code>	nullable type
<code>T?</code>	optional key type
<code>T[]</code>	array type
<code>X Y</code>	union type
<code>X&Y</code>	intersection type

Сложна



participe

```
parser := participle.MustBuild(&Expr{})
```

Go structs



Inferred parser

```
type Expr struct {  
    Number *float64 `@(Float|Int)`  
    Var    *string   `| @Ident`  
}
```

Быстрая справка (но у нас нет на неё времени)

@<expr>	Парсим expr, кладём в поле
@@	Парсим по типу поля, кладём туда же
Ident, Int, ...	Парсим именованный токен
"foo"	Парсим токен с ровно таким текстом
<expr> <expr>	Парсим альтернативы (or)
<expr>*	Парсим expr 0-n раз (результат - слайс)
<expr>?	Парсим expr 0-1 раз

Парсим простые выражения типов

```
type TypeNameExpr struct {  
    Primitive *string      `@"int"|"float"`  
    Class     *ClassNameExpr `| @@`  
}  
  
type ClassNameExpr struct {  
    Part string      `@Ident`  
    Next *ClassNameExpr `("\\" @@)?`  
}
```






Приоритеты операторов

1. Имена, скобочки (самый высокий)
2. Nullability operator
3. Массивы, optional key operator
4. Intersection
5. Union (самый низкий)

```
X|Y&Z == X|(Y&Z)
```

```
?int[] == (?int)[]
```


Выражаем приоритеты через грамматику

1. Имена, скобочки (самый высокий)  ● PrimaryExpr
2. Nullability operator  ● PrefixExpr
3. Массивы, optional key operator  ● PostfixExpr
4. Intersection  ● IntersectionExpr
5. Union (самый низкий)  ● UnionExpr

```
X|Y&Z == X|(Y&Z)
```

```
?int[] == (?int)[]
```



1. **PrimaryExpr** int float Foo (int)
2. PrefixExpr ?X
3. PostfixExpr X? X[]
4. IntersectionExpr X&Y
5. UnionExpr X|Y

```
type PrefixExpr struct {  
    Ops    []*PrefixOp    `@@*`  
    Right  *PrimaryExpr    `@@`  
}
```



1. PrimaryExpr int float Foo (int)
2. PrefixExpr ?X
3. PostfixExpr X? X[]
4. IntersectionExpr X&Y
5. UnionExpr X|Y

```
type PostfixExpr struct {  
    Left *PrefixExpr `@@`  
    Ops []*PostfixOp `@@*`  
}
```



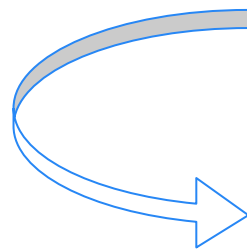
1. PrimaryExpr int float Foo (int)
2. PrefixExpr ?X
3. PostfixExpr X? X[]
4. IntersectionExpr X&Y
5. UnionExpr X|Y

```
type IntersectionExpr struct {  
    Left  *PostfixExpr    `@@`  
    Right *IntersectionExpr `("&" @@)?`  
}
```

1. PrimaryExpr int float Foo (int)
2. PrefixExpr ?X
3. PostfixExpr X? X[]
4. IntersectionExpr X&Y
5. UnionExpr X|Y



```
type UnionExpr struct {  
    Left  *IntersectionExpr `@@`  
    Right *UnionExpr       `("|" @@)?`  
}
```



1. PrimaryExpr int float Foo (int)
2. PrefixExpr ?X
3. PostfixExpr X? X[]
4. IntersectionExpr X&Y
5. UnionExpr X|Y

```
type PrimaryExpr struct {  
    TypeName *TypeNameExpr `@@`  
    Parens   *UnionExpr     `| "(" @@ ")"`  
}
```

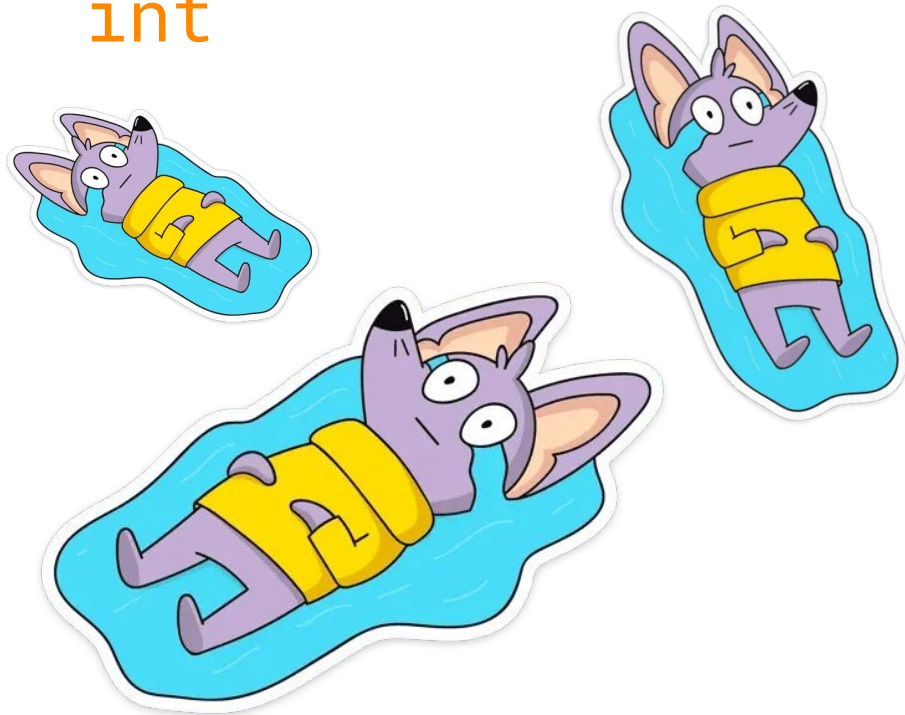
Запустим парсер!

Запустим парсер!

int

Запустим парсер!

int



```
{
  "Left": {
    "Left": {
      "Left": {
        "Ops": null,
        "Right": {
          "TypeName": {
            "Primitive": "int",
            "Class": null
          },
          "Parens": null
        }
      },
      "Ops": null
    },
    "Right": null
  },
  "Right": null
}
```

Плюсы `participle` (+)

- Автоматический мапинг грамматики на AST
- Не нужны внешние утилиты типа уасс
- Легко парсить простые форматы

Минусы `Participle` (-)

- Может получиться не очень удобное AST
- Нет простых способов работы с приоритетами операторов
- Парсер может получиться медленным

Отслеживаем производительность

В исходниках vk.com примерно 400000 phpdoc типов

Метод парсинга	Парсим <code>Foo Bar null</code>	400000 раз
participle	78000 наносек	~31 сек



Преобразуем AST

Будешь использовать сгенерированные под
grps типы в бизнес-логике?

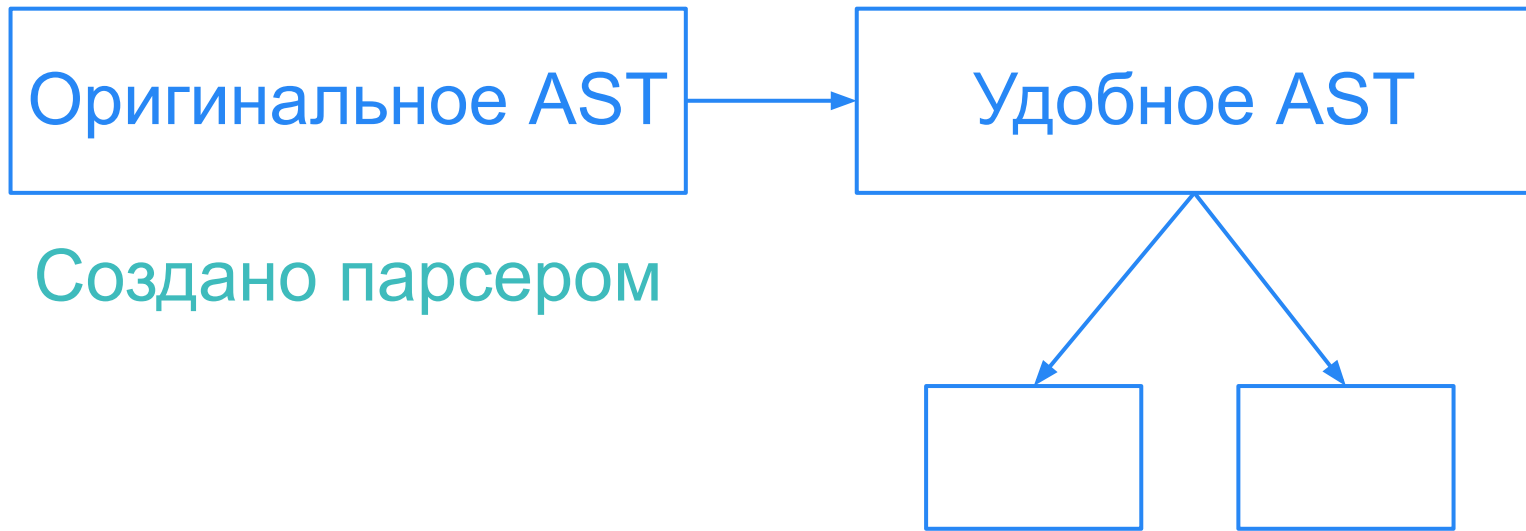
Нет!



Да!

Вводим новые AST-типы

package phpdoc



Новое AST 👍

```
type IntersectionType struct {
    Left  Type
    Right Type
}

type UnionType struct {
    Left  Type
    Right Type
}

type ClassName struct {
    Parts []string
}

type Type interface {
    String() string
}
```

Старое AST 👎

```
type IntersectionExpr struct {
    Left  *PostfixExpr  `@@`
    Right *IntersectionExpr `("&" @@)?`
}

type UnionExpr struct {
    Left  *IntersectionExpr `@@`
    Right *UnionExpr      `("|" @@)?`
}

type ClassNameExpr struct {
    Part string `@Ident`
    Next *ClassNameExpr `("\\" @@)?`
}
```

Пример конвертирования AST

```
// primary ::= type_name | '(' union_expr ')'  
func (conv *converter) convertPrimary(expr *PrimaryExpr) phpdoc.Type {  
    if expr.TypeName != nil {  
        return conv.convertTypeName(expr.TypeName)  
    }  
    if expr.Parens != nil {  
        return conv.convertRoot(expr.Parens)  
    }  
    return nil  
}
```


Пример конвертирования AST

```
// type_name ::= primitive_type_name | class_name
func (conv *converter) convertTypeName(expr *TypeNameExpr) phpdoc.Type {
    if expr.Primitive != nil {
        return &phpdoc.PrimitiveTypeName{Name: *expr.Primitive}
    }
    if expr.Class != nil {
        return conv.convertClassName(expr.Class, nil)
    }
    return nil
}
```

Плюсы (+)

- Проще менять парсер
- Увеличивает удобство работы с AST во всём остальном коде

Минусы (-)

- Очередной IR
- Конвертировать AST не бесплатно - замедляем наш парсинг



Тестируем парсер

Тестовые кейсы

```
tests := []struct {  
    input  string  
    expect string  
}{  
    {`int`,    `int`},  
    {`float`,  `float`},  
    {`A\B\C`,  `A\B\C`},  
    {`(int)`,  `int`},  
    {`?int`,   `?(int)`},  
}
```

Ожидаемый результат
в формате строки,
создаваемой
`Type.String()`

Парсим и сравниваем

```
typ, err := p.Parse(` ` + test.input + ` `)
if err != nil {
    // fail: unexpected error
}
if typ.String() != test.expect {
    // fail: printed form mismatches
}
```

Парсим повторно и сравниваем

```
typ2, err := p.Parse(typ.String())
if err != nil {
    // fail: unexpected error during re-parse
}
if typ.String() != typ2.String() {
    // fail: re-parse result mismatches
}
```



Oleg Kovalov

произнеси слово ФАЗЗИНГ плс)

1:20 PM

ФАЗЗИНГ
(кродеться)



goyacc + text/scanner

Используем уасс

1. Пишем файл грамматики (.y)
2. Генерируем парсер (через запуск `go yacc`)
3. Реализуем лексер под него
4. Соединяем лексер с парсером

Вводим заготовку для лексера

```
// A simple lexer that uses a scanner.Scanner
// to do the lexing. It also stores
// the parse result inside itself.
type yyLex struct {
    s      scanner.Scanner
    result phpdoc.Type
}
```

Вводим заготовку для лексера

```
// A simple lexer that uses a scanner.Scanner  
// to do the lexing. It also stores  
// the parse result inside itself.
```

```
type yyLex struct {  
    s scanner.Scanner  
    result phpdoc.Type  
}
```

yy?!
дефолтный префикс
для yacc

phpdoc.y: определяем “symbol value”

```
// превратится в структуру yySymType
%union{
    tok  rune          // id текущего токена
    text string        // текст для T_NAME токенов
    expr phpdoc.Type   // для правил с типами
}
```

Типы токенов и нетерминальных правил

example

```
: T_NAME ':' type_expr {  
    var text string = $1  
    var expr phpdoc.Type = $3  
    ...  
}
```

phpdoc.y: декларация токенов

```
%token <tok> T_NULL
%token <tok> T_FALSE
%token <tok> T_INT
%token <tok> T_FLOAT
%token <tok> T_STRING
%token <tok> T_BOOL
%token <text> T_NAME

%union{
    tok rune
    text string
    expr phpdoc.Type
}
```

phpdoc.y: декларация токенов

```
%token <tok> T_NULL
%token <tok> T_FALSE
%token <tok> T_INT
%token <tok> T_FLOAT
%token <tok> T_STRING
%token <tok> T_BOOL
%token <text> T_NAME

%union{
    tok rune
    text string
    expr phpdoc.Type
}
```

phpdoc.y: декларация токенов

```
%token <tok> T_NULL
%token <tok> T_FALSE
%token <tok> T_INT
%token <tok> T_FLOAT
%token <tok> T_STRING
%token <tok> T_BOOL
%token <text> T_NAME

%union{
    tok rune
    text string
    expr phpdoc.Type
}
```


phpdoc.y: декларация токенов

```
%token <tok> T_NULL  
%token <tok> T_FALSE  
%token <tok> T_INT  
%token <tok> T_FLOAT  
%token <tok> T_STRING  
%token <tok> T_BOOL  
%token <text> T_NAME
```

Ассоциированные id
токенов

Для них будут
созданы константы

phpdoc.y: приоритеты и ассоциативность

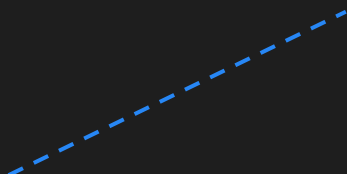
```
%right '|'      // самый низкий приоритет
%right '&'
%left OPTIONAL  // постфиксный '?'
%right '['
%left '?'       // самый высокий приоритет
```

phpdoc.y: описываем синтаксис

```
start
    : type_expr { yylex.(*yyLex).result = $1 }
    ;
```

phpdoc.y: описываем синтаксис

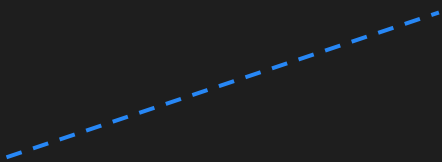
```
                                %type <expr> type_expr  
start  
  : type_expr { yylex.(*yyLex).result = $1 }  
  ;
```



phpdoc.y: описываем синтаксис

```
yyParser.Parse(lexer)

start
: type_expr { yylex.(*yyLex).result = $1 }
;
```



phpdoc.y: описываем синтаксис

```
                                type yyLex struct {  
                                    s      scanner.Scanner  
                                    result phpdoc.Type  
                                }  
start  
    : type_expr { yylex.(*yyLex).result = $1 }  
    ;
```

phpdoc.y: описываем синтаксис

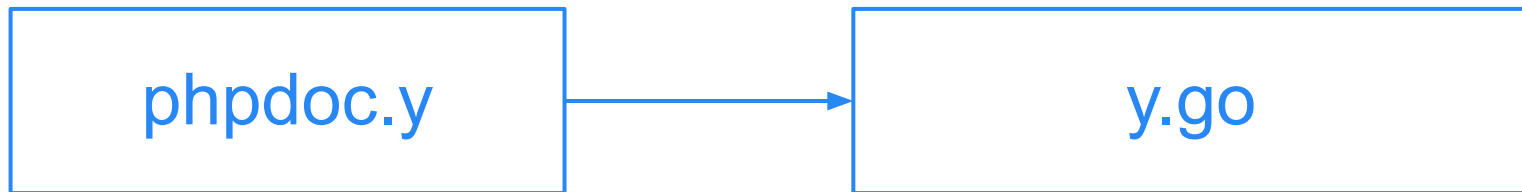
```
primitive_type
    : T_NULL { $$ = &PrimitiveTypeName{Name: "null"} }
    | T_FALSE { $$ = &PrimitiveTypeName{Name: "false"} }
    | T_INT { $$ = &PrimitiveTypeName{Name: "int"} }
    | T_FLOAT { $$ = &PrimitiveTypeName{Name: "float"} }
    | T_STRING { $$ = &PrimitiveTypeName{Name: "string"} }
    | T_BOOL { $$ = &PrimitiveTypeName{Name: "bool"} }
    ;
```

phpdoc.y: описываем синтаксис

```
type_expr
: primitive_type { $$ = $1 }
| T_NAME { $$ = &TypeName{Parts: strings.Split($1, ``)} }
| '(' type_expr ')' { $$ = $2 }
| '?' type_expr { $$ = &NullableType{Elem: $2} }
| type_expr '[' ']' { $$ = &ArrayType{Elem: $1} }
| type_expr '?' %prec OPTIONAL { $$ = &OptionalKeyType{Elem: $1} }
| type_expr '&' type_expr { $$ = &IntersectionType{X: $1, Y: $3} }
| type_expr '|' type_expr { $$ = &UnionType{X: $1, Y: $3} }
;
```


Запускаем goyacc

Сгенерированный файл



goyacc грамматика

```
$ goyacc phpdoc.y
```

Реализуем лексер

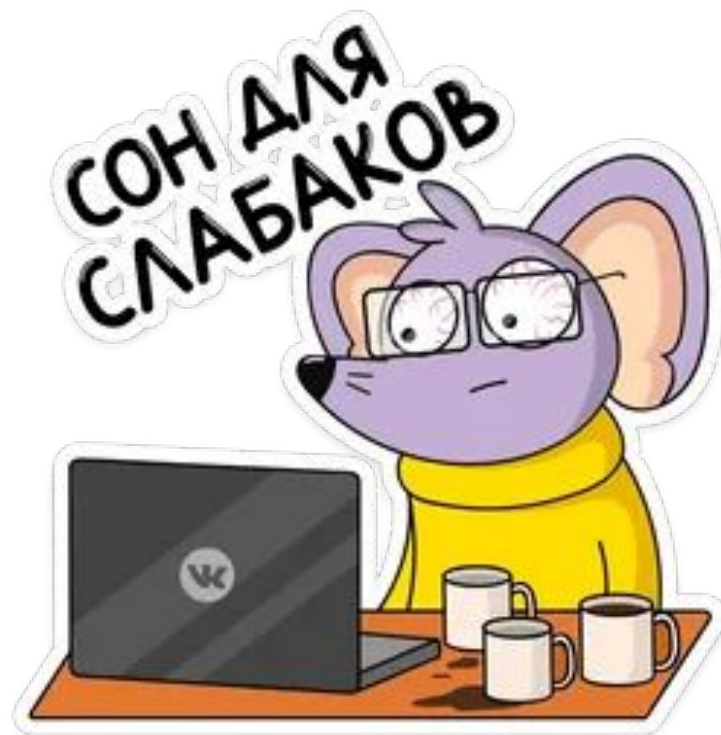
```
func (l *yyLex) Lex(sym *yySymType) int {  
    tok := l.nextToken(sym)  
    sym.tok = tok  
    return int(tok)  
}  
  
%union{  
    tok rune  
    text string  
    expr phpdoc.Type  
}
```

Реализуем лексер

```
func (l *yyLex) nextToken(sym *yySymType) rune {  
    tok := l.s.Scan()  
    if tok == scanner.Ident {  
        text := l.s.TokenText()  
        if tok, ok := nameToToken[text]; ok { return tok }  
        sym.text = text  
        return T_NAME  
    }  
    return tok  
}
```

Реализуем лексер

```
var nameToToken = map[string]rune{
    "int":    T_INT,
    "float":  T_FLOAT,
    "null":   T_NULL,
    "string": T_STRING,
    "false":  T_FALSE,
    "bool":   T_BOOL,
}
```



Связываем воедино и запускаем!

```
input := "int|?float"

lexer := NewLexer()
lexer.s.Init(strings.NewReader(input))
parser := yyNewParser()
parser.Parse(lexer)
result := lexer.result
```

Плюсы goyass (+)

- Генерирует эффективные парсеры
- Можно сразу собирать красивое AST
- Удобно описывать приоритеты и ассоциативность

Минусы goyass (-)

- Требует стороннюю утилиту (goyass)
- Не очень красивая интеграция с Go (но в целом ОК)

Отслеживаем производительность

В исходниках vk.com примерно 400000 phpdoc типов

Метод парсинга	Парсим <code>Foo Bar null</code>	400000 раз
participle	78000 наносек	~31 сек
goyacc + text/scanner	4200 наносек	~1.68 сек



goyacc + ragel

Структура Ragel файлов

```
// обычный код...

%%{
    Ragel сниппет (multi-line)
}%}

%% Ragel директива (single-line);

// обычный код...
```

Реализуем лексер

```
package main

%%machine lexer;
%%write data;

type yyLex struct {
    pos      int
    src      string
    result   phpdoc.Type
}
```

Реализуем лексер

```
package main
```

```
%%machine lexer;
```

```
%%write data;
```

```
type yyLex struct {
```

```
    pos    int
```

```
    src    string
```

```
    result phpdoc.Type
```

```
}
```

Название для
генерируемой FSM

(играет роль префикса)

Реализуем лексер

```
package main
```

```
%%machine_lexer;
```

```
%%write data;
```

```
type yyLex struct {
```

```
    pos    int
```

```
    src    string
```

```
    result phpdoc.Type
```

```
}
```

Вставляем в это место
константы и таблицы
для FSM

Реализуем лексер

```
package main

%%machine lexer;
%%write data;

type yyLex struct {
    pos      int
    src      string
    result    phpdoc.Type
}
```

Теперь мы сами будем
сканировать входную
строку

Реализуем лексер

```
func (l *yyLex) Lex(lval *yySymType) int {  
    tok := 0  
    // TODO: сюда вставим boilerplate  
    %%{  
        // TODO: декларация FSM  
    %%}  
    %%write init;  
    %%write exec;  
    return tok  
}
```

Boilerplate

```
data := l.src    // string или []byte
p := l.pos       // текущее смещение (внутри data)
pe := len(data)  // позиция окончания
eof := pe        // позиция EOF

// ts и te - это начало/конец токена
var cs, ts, te, act int
```


Описываем FSM

```
whitespace = [\t ];
```

```
ident_first = [a-zA-Z_] | (0x0080..0x00FF);
```

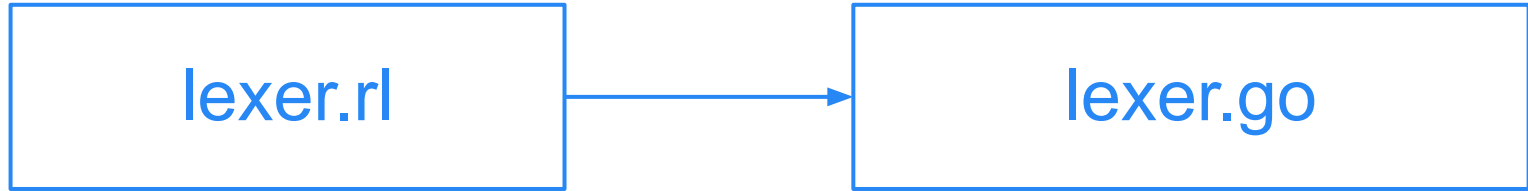
```
ident_rest  = ident_first | [0-9] | [\\];
```

```
ident       = ident_first (ident_rest)*;
```

Описываем FSM

```
main := |*  
    whitespace => {};  
    'int' => { tok = T_INT; fbreak; };  
    'float' => { tok = T_FLOAT; fbreak; };  
    'null' => { tok = T_NULL; fbreak; };  
    'string' => { tok = T_STRING; fbreak; };  
    'false' => { tok = T_FALSE; fbreak; };  
    'bool' => { tok = T_BOOL; fbreak; };  
    ident => { tok = T_NAME; fbreak; };  
    any => { tok = int(data[ts]); fbreak; };  
*|;
```

Сгенерированный файл



Рagel файл

```
$ ragel -Z -G2 lexer.rl -o lexer.go
```

Тип генерируемой FSM,
G0, G1, G2 - goto-based

Внимание: может сгенерировать
десятки тысяч строк кода



```
$ ragen -Z -G2 lexer.rl -o lexer.go
```

Почему именно Ragel, а не golex?

- Создаёт нереально быстрый код (быстрее golex)
- Удобный в использовании (лучше golex)
- Полезен не только для лексеров/парсеров

Рандомный факт: [php-parser](#) когда-то использовал golex вместо ragel

Плюсы ragel (+)

- Можно создать очень эффективный лексер под любой парсер
- Делает ваши волосы более шелковистыми

Минусы ragel (-)

- Требует ещё одной сторонней утилиты (ragel)

Отслеживаем производительность

В исходниках vk.com примерно 400000 phpdoc типов

Метод парсинга	Парсим <code>Foo Bar null</code>	400000 раз
participle	78000 наносек	~31 сек
goyacc + text/scanner	4200 наносек	~1.68 сек
goyacc + ragel	1600 наносек	~0.6 сек



Пишем парсер ручками

Зачем вообще писать парсер руками?

Некоторые из сигналов:

- Для вашего формата нет хорошей грамматики
- Написать грамматику для формата слишком сложно
- Нужно разбирать частично некорректные данные
- Вы feeling lucky

╰_(ツ)_╯

Рекурсивный спуск для людей



Парсеры Пратта

Введение в парсеры Пратта

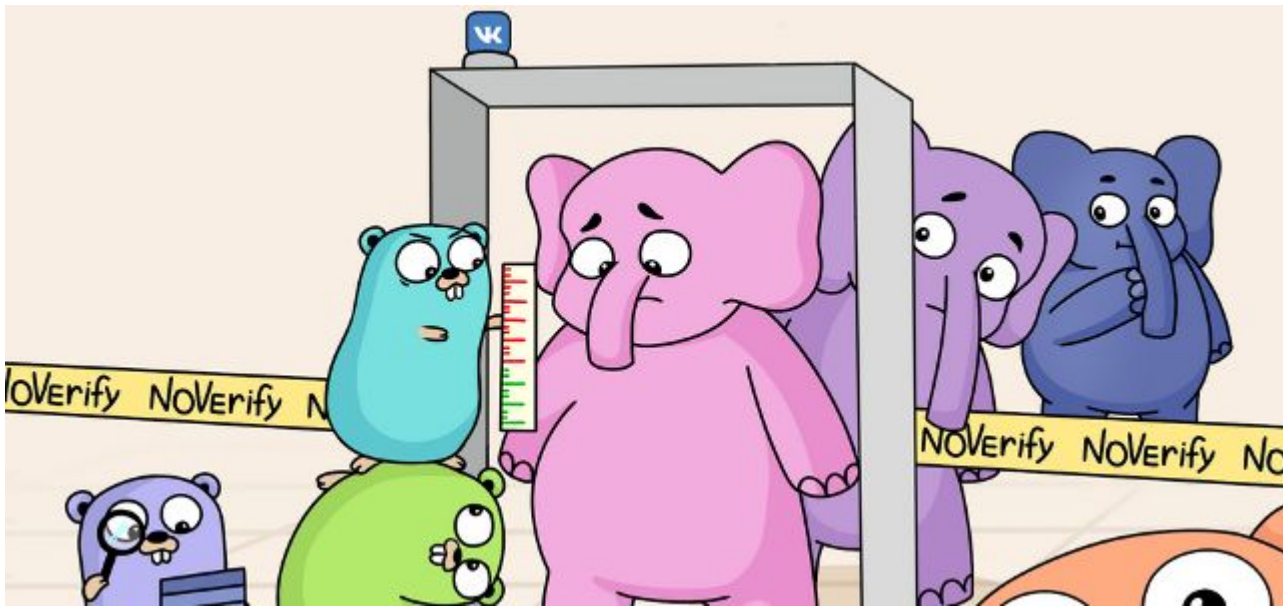
Статьи с примерами на Go:

- [Парсеры Пратта в Go \(ru\)](#)
- [Pratt parsers in Go \(en\)](#)

Оригинал с примерами на Java:

- [Pratt Parsers: Expression Parsing Made Easy](#)

В NoVerify* парсер phpdos типов у нас
написан руками



(*) Линтер для PHP, написанный на Go

Парсер phpdoc типов NoVeirfy

- Умеет разбирать частично некорректные типы
- Во многих случаях не делает аллокаций
- Поддерживает сложные типы вроде генериков

Sources: [noverify/src/phpdoc/type_parser.go](https://noverify.github.io/src/phpdoc/type_parser.go)

Отслеживаем производительность

В исходниках vk.com примерно 400000 phpdoc типов

Метод парсинга	Парсим <code>Foo Bar null</code>	400000 раз
participle	78000 наносек	~31 сек
goyacc + text/scanner	4200 наносек	~1.68 сек
goyacc + ragel	1600 наносек	~0.6 сек
ручной парсер	800 наносек	~0.3 сек

Отслеживаем производительность

В исходниках vk.com примерно 400000 phpdoc типов

Метод парсинга	Парсим <code>Foo Bar null</code>	400000 раз
participle	78000 наносек	~31 сек
goyасс + text/scanner	4200 наносек	~1.68 сек
goyасс + ragel	1600 наносек	~0.6 сек
ручной парсер	800 наносек	~0.3 сек
ручной парсер (no conv)	250 наносек	~0.1 сек



Почти всё

Исходники всех примеров (готовые парсеры)

github.com/quasilyte/parsing-and-go

- Парсер на основе participle
- goyacc+text/scanner
- goyacc+ragel
- Вручную написанный парсер
- Тесты
- Бенчмарки

Что мы не разобрали (домашнее задание)

- Обработка ошибок в парсере и лексере
- Проставление локаций/позиций для AST элементов
- `participle` с Ragel лексером и альтернативной грамматикой
- Пулы объектов для парсеров
- Некоторые другие либы, типа [pigeon](#) (аналог `participle`)
- Как ещё тестировать парсеры (тесты позиций)
- Что делать с комментариями (`freefloating` токены)
- Левая рекурсия и прочие заболевания

...

Полезные штучки

Работа с грамматиками:

- [Resolving common grammar conflicts in parsers](#)
- [Crafting interpreters: parsing expressions](#)

Полезные штучки

Ragel:

- [Ragel: state machine compiler](#)
- [Lexing with Ragel Parsing with Yacc](#)
- [Speeding up regexp matching with ragel](#)
- [Ragel cheat sheet](#)

Полезные штучки

Парсеры Пратта:

- [Pratt Parsers: Expression Parsing Made Easy](#)
- [Pratt parsers in Go \(ru\)](#)
- [Pratt parsers in Go \(en\)](#)



Parsing & Go

@quasilyte 2021