

Обзор разработки языков программирования

Камнев Георгий Павлович

nt.gocha@gmail.com

2020

Мотивация

Субъективные причины влияют на объективные процессы

ЯП — инструменты с набором **относительно** хороших концептов

Объективно

ЯП могут быть многословны

Может не доставать концептов в языке → дублирование кода → Ошибки

Концепты даже в пределах одного языка могут быть спорными

Дилемма

Собственные знания опережают формальные — унифицированные процессы

Общее для всех ЯП и не только

ЯП - Это черный ящик с входом и выходом

Вход — программа - исходный код

Выход — программа/результат вычислений

Синонимы и близкие термины/задачи

Parse (парсинг), синтаксический, лексический анализ, интерпретация, трансляция, компиляция, формальная грамматика

Для ЯП характерны те же концепты, что и для языков вообще

Структура, Семантика(смысл), Символичность, Правила, Эволюция/Деволюция

Исходный код — не только ценный мех 1

Формы исходного кода

Текст согласно определенным правилам

Графические языки (Дракон, Схема-техника, BPMn)

Последовательность байт для виртуальной машины или процессора

Исходный код — не только ценный мех 2

Формы исходного кода

Текст согласно определенным правилам

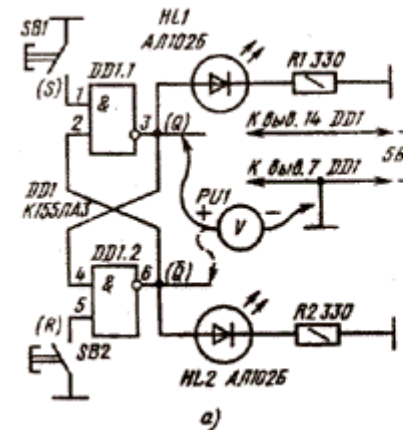
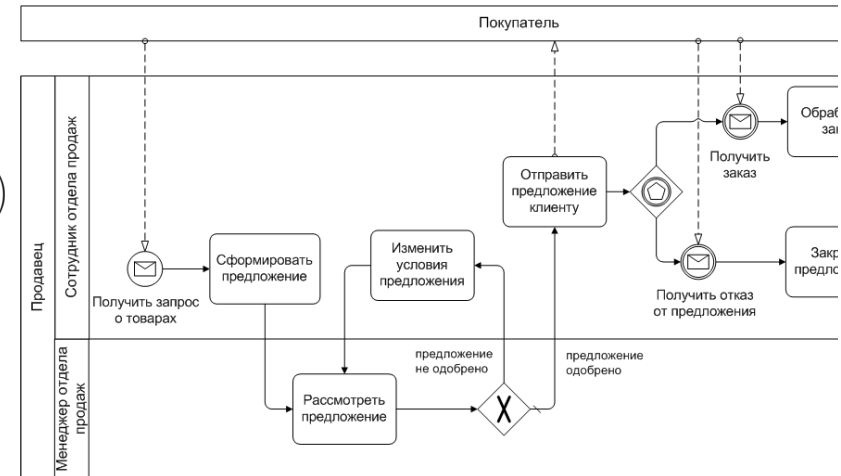
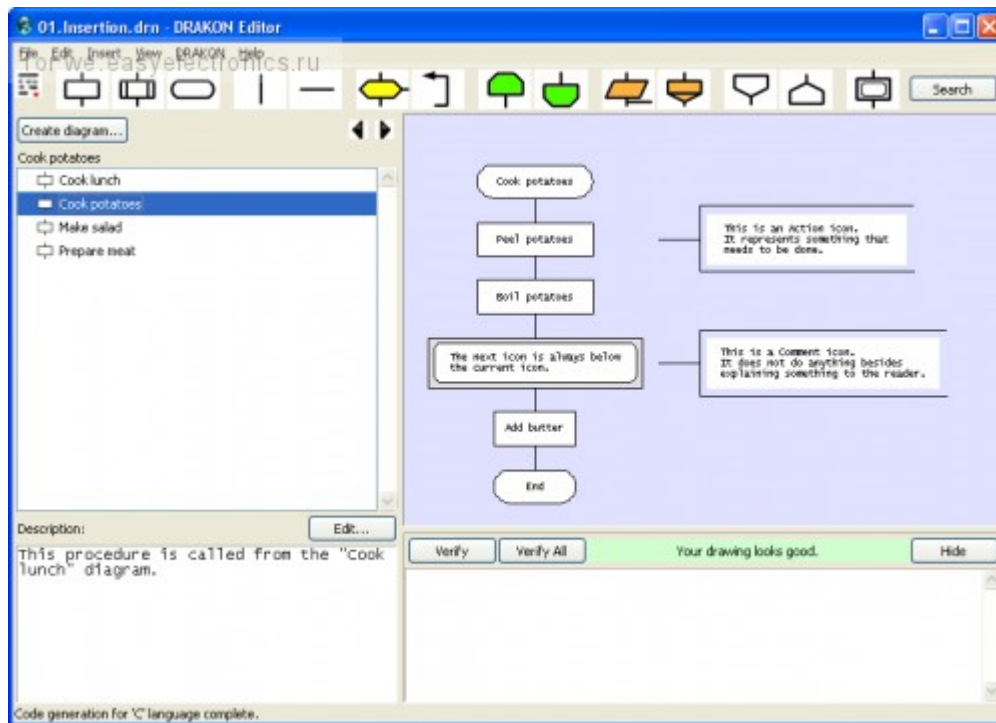
```
public class CreateWebSource
{
    public static void main(String[] args) throws Exception
    {
        System.out.println( "" );

        try
        {
            if ( args == null || args.length != 4 )
            {
                System.out.println(
                    "Usage:\n  CreateWebSource <webServiceURL> <userName> <password>");
            }
            else
            {
                // Get web service URL from command-line arguments
                String webServiceURL = args[0];
                System.out.println( "Using web service URL \"" + webServiceURL
```

Исходный код — не только ценный мех 3

Формы исходного кода

Графические языки (Дракон, Схема-техника, BPMn)



S	R	Q	\bar{Q}
0	0	1	1
0	1	1	0
1	0	0	1
1	1	X	X

б)

«Интерпретатор» и «Компилятор»

Прочитать исходную программу

Составить план действий

Составляется на основе формальной договоренности меж программистом и интерпретатором — формальной грамматике правил языка программирования

Проверить корректность плана

Выполнить

Согласно плану выдать результат

Или результат

Или программа на другом языке

C/C++ ⇒ EXE | WASM

JAVA ⇒ JVM

TS ⇒ JS

ORM ⇒ SQL

$$\begin{array}{l} 2 + 2 \times 2 \\ \downarrow \\ (2 + (2 \times 2)) \\ \downarrow \\ (2 + 4) \\ \downarrow \\ 6 \end{array}$$

AST / «Прочитать исходный код»

AST — Abstract syntax tree

Абстрактное синтаксическое
дерево

Форма представления исходного кода
в виде дерева объектов

- ◆ - Бинарный оператор $\rightarrow \text{fn}(\text{left}, \text{right})$
- - Литеральное значение (константа)
- - Значение зависимое от контекста (переменная)

$1 + 2.5 * a - 4 ^ b(5)$

◆ Вычитание

◆ Сложение

● Число - литерал 1

◆ Умножение

● Число - литерал 2.5

□ Переменная a

◆ Возведение в степень

● Число - литерал 4

□ Результат вызова функции

□ Имя функции b

Аргументы

● Число - литерал 5

Компиляция или же план действий

$$1 + 2.5 * a - 4 ^ b(5)$$

- ◆ Вычитание 19
- ◆ Сложение 4
 - Число - литерал 1 3
 - ◆ Умножение 2
 - Число - литерал 2.5 0
 - Переменная a 1
- ◆ Возведение в степень 10
 - Число - литерал 4 9
 - Результат вызова функции 8
 - Имя функции b 7
 - Аргументы 6
 - Число - литерал 5 5

План действий

Рекурсивный спуск с верху вниз

Вычисление листовых значений

Для интерпретаторов — вычисление

Для компиляторов — генерация инструкций

Подъем вверх и вычисление

И так до самого корня обратно

Фазы трансляции

Анализ

Лексический анализ

разбиение потока символов на простые конструкции: литералы, ключевые слова, и т.д.

Синтаксический анализ

Сборка AST дерева

Вывод типов, семантика

может быть имплицитна фаза

Для узлов AST вычисляется тип результата и как он будет достигнут

Модификации (опционально)

Модификация AST

Макросы

Генерация

Для интерпретаторов — выполнение

Для компиляторов — генерация кода

Лексический анализ

Задача лексического анализа

разбиение потока символов на простые конструкции:

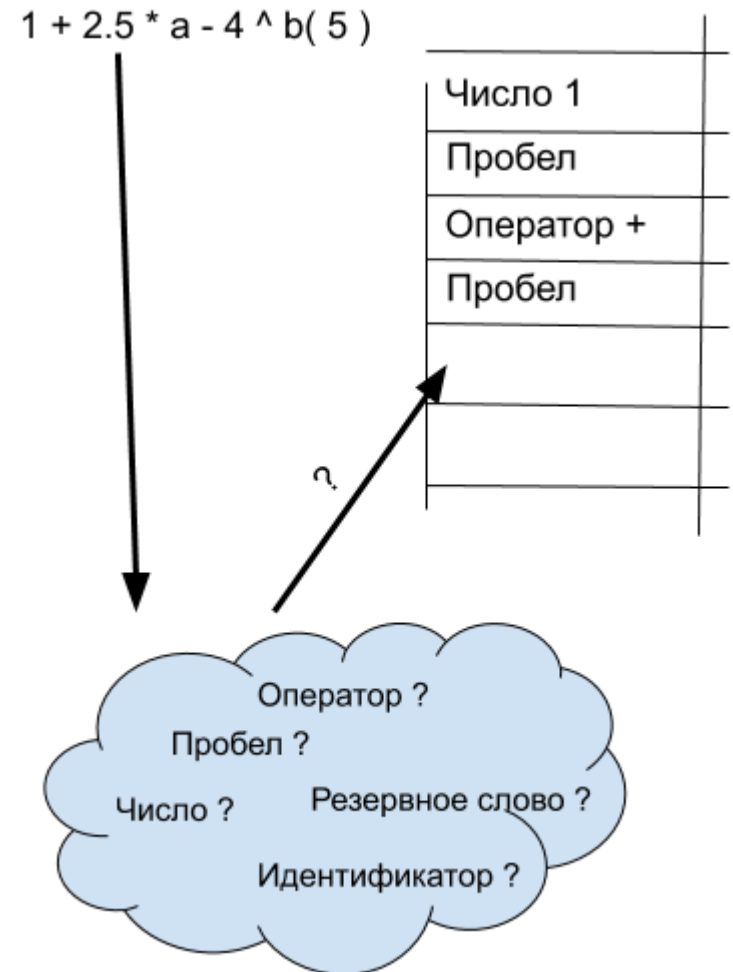
литералы, ключевые слова, и т. д. - **лексемы (Token)**

1 + 2.5 * a - 4 ^ b (5)

На выходе должен получиться массив примерно такой

Индекс (0...) Терминал/Лексема Значение

0	Число	1
1	Оператор	+
2	Число	2.5
3	Оператор	*
4	Идентификатор	a
5	Оператор	-
6	Число	4
7	Оператор	^
8	Идентификатор	b
9	Резервное слово	(
10	Число	5
11	Резервное слово)



Лексический анализатор, псевдокод

lexer(inputSource, from) : List<Token> = {

ptr = from

while(!eof){

hasMatch = false

for(rule in rules){

match : Option[Token] = rule(inputSource, ptr)

if(match.ok){

hasMatch = true

ptr += ...

}

}

if(!hasMatch) error() else continue;

Лексер классы

Лексемы/Терминалы

Token {

begin : SourcePointer

end : SourcePointer

}

SourcePointer {

file : File

lineNumber : Integer

charNumber : Integer

text(to : SourcePointer) : Option[String]

move(charCount : Integer) : SourcePointer

eof() : Boolean

}

Указатель на позицию
в исходнике

NumberToken extends Token {

value : Number

}

StringLiteralToken extends Token {

value : String

}

KeywordToken extends Token {

keyword : Keyword

}

Синтаксический анализ, постановка проблемы

Пример — TASK123

определить относятся ли входной набор символов к числу или нет

тип чисел

Дробное

Описание

три блока проверок

Целая часть - Проверка на совпадение к символа к классу цифр

Может повторяться 0 и более раз

Десятичная точка - Проверка на совпадение к символа к классу д. точек

Обязательно и только один раз

Дробная часть - Проверка на совпадение к символа к классу цифр

Может и должно повторяться 1 и более раз

Пример

234.568

Данные на входе (массив)

Символ	Класс
2	целая часть
3	целая часть
4	целая часть
.	десятичная точка
5	дробная часть
6	дробная часть
8	дробная часть

Синтаксический анализ, псевдокод (TASK123)

```
function parseNumber( text, offset ){  
    const begin = offset  
  
    // целая часть  
    while( true ){  
        if( text[offset] in ['0','1',..., '9'] ){  
            intBuffer += text[offset]  
            offset += 1  
        }else{  
            break  
        }  
    }  
  
    // десятичная точка  
    if( !(text[offset] in [',', ';']) ){  
        return null  
    }  
}
```

```
// дробная часть  
while( true ){  
    if( text[offset] in ['0','1',..., '9'] ){  
        floatBuffer += text[offset]  
        offset += 1  
    }else{  
        break  
    }  
}  
  
if( floatBuffer.length<1 )return null  
  
return {  
    begin: begin,  
    end: offset,  
    number: parseFloat( intBuffer + '.' + floatBuffer ) }  
}
```

Синтаксический анализ, псевдокод критика TASK123

Избыточность

Такой код избыточен по сравнению с описанием числа, при том что описание однозначно и непротиворечиво

Решение

Формализация реализации

Типичные части реализации можно заменить на условный код (псевдокод)

Формализация постановки

При постановке использовать условный код

Правила написания зафиксировать в документе Грамматика TASK123

Типичные части описания грамматики сводятся к более простым и однозначным конструкциям, следуя которым можно написать однозначный парсер

Формальные грамматики, TASK123

Примеры грамматических правил 1

DIGIT

Описание

Множество всех десятичных цифр

Формальное описание

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Псевдокод

```
var DIGIT = ['0','1',..., '9']
```

INTPART

Описание

Целая часть,

Может 0 и более раз содержать цифры

Формальное описание

{ DIGIT }

Псевдокод

```
while( true )
```

```
if( text[offset] in DIGIT ){
```

```
    intBuffer += text[offset]
```

```
    offset += 1
```

```
}else break
```

DOT

Описание

Десятичная точка

Формальное описание

. | ,

Псевдокод

```
if( !(text[offset] in [',', '.']) ) return null
```

Формальные грамматики, TASK123

Примеры грамматических правил 2

FLOATPART

Описание

Дробная часть. Может 1 и более раз содержать цифры

Формальное описание

`DIGIT { DIGIT }`

parseNumber

Описание

Проверка является ли текст дробным числом

Формальное описание

`[INTPART]`

`DOT`

`FLOATPART`

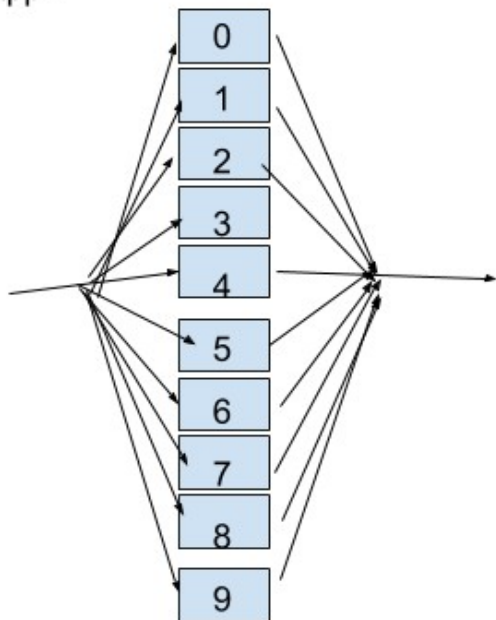
В данной грамматике используются спец синтаксис:

- ◆ Фигурные скобки - означает что содержание может быть повторено 0 и более раз
- ◆ Квадратные скобки - может отсутствовать
- ◆ Последовательное написание см. `parseNumber` - соответ последовательным проверкам, где каждая проверка начинается с того места, на котором завершилась успешно предыдущая проверка, иначе отвергается вся последовательность
- ◆ Вертикальная черта - альтернативные варианты

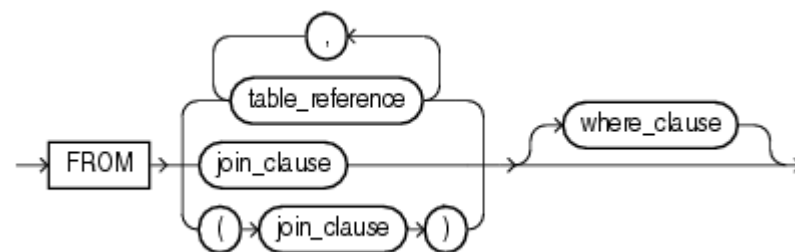
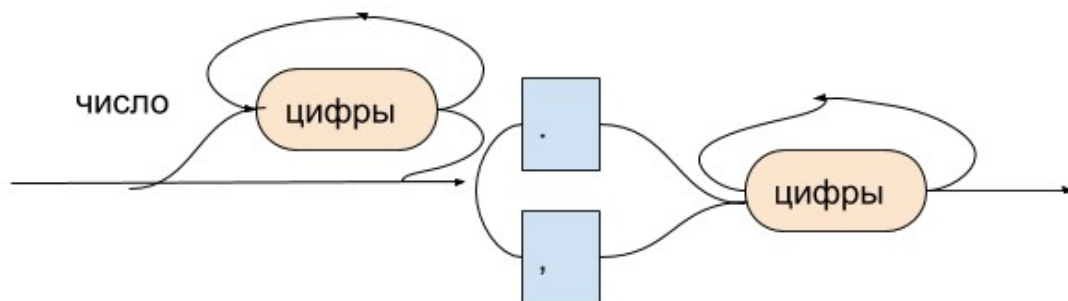
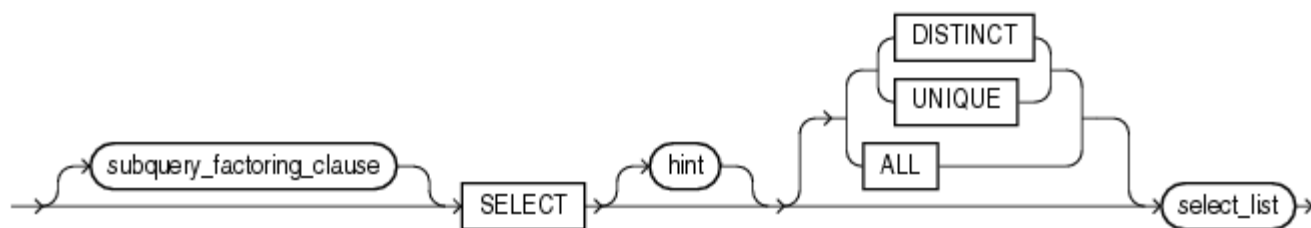
Формальные грамматики, TASK123

Графическая нотация

цифры



Такая нотация используется в документации СУБД ORACLE



Что такое грамматика языка

Грамматика языка X

это некая программа на псевдоязыке

Грамматика описывает набор правил, где правило:

правило - это функция

правило может быть именованное (например `parseNumber`)

и содержит описание на псевдоязыке

Упоминание в теле функции грамматики, другого (или этого же) равно вызову соответ. функции

Задача грамматики

распознать что некая часть текста (или весь) соответствует некому правилу язык

Требование

Грамматика должна быть однозначной

Грамматика должна быть конечной (не уходить в бесконечную рекурсию)

Рабочий пример. BASIC. Фибоначи.

Проект <https://gochaorg.github.io/basic/>

Интерпретатор BASIC написан на typescript

```
01 rem sample fibonachchi
10 let sum = 0
20 let num1 = 0
30 let num2 = 1
32 let cnt = 0
40 let sum = num1 + num2
50 let num1 = num2
55 let num2 = sum
60 print sum
62 let cnt = cnt + 1
70 if cnt < 100 then goto 40
```

```
BASIC> 1
BASIC> 2
BASIC> 3
BASIC> 5
BASIC> 8
BASIC> 13
BASIC> 21
BASIC> 34
BASIC> 55
BASIC> 89
BASIC> 144
BASIC> 233
BASIC> 377
BASIC> 610
BASIC> 987
```

Рабочий пример. BASIC. AST.

```
{
  "sourceLines": [
    {
      "line": 1,
      "statement": {}
    },
    {
      "line": 10,
      "statement": {}
    },
    {
      "line": 20,
      "statement": {}
    }
  ]
}
```

N° src line

40 let sum = num1 + num2

```
{
  "line": 40,
  "statement": {
    "kind": "Let",
    "begin": {},
    "end": {},
    "variable": {
      "begin": 95,
      "end": 98,
      "kind": "ID",
      "id": "sum"
    },
    "value": {
      "operator": {},
      "left": {},
      "right": {},
      "lexems": [],
      "children": [],
      "kind": "BinaryOperator"
    }
  }
}
```

40 let sum = num1 + num2

```
{
  "value": {
    "operator": {
      "begin": 106,
      "end": 107,
      "keyWord": "+",
      "kind": "OperatorLex"
    },
    "left": {
      "id": {
        "begin": 101,
        "end": 105,
        "kind": "ID",
        "id": "num1"
      },
      "lexems": [],
      "children": [],
      "kind": "VarRef"
    },
    "right": {},
    "lexems": [],
    "children": [],
    "kind": "BinaryOperator"
  }
}
```

BASIC. TS. statement()

```
/**
 * statement ::= remStatement
 *             | letStatement
 *             | runStatement
 *             | gotoStatement
 *             | ifStatement
 *             | gosubStatement
 *             | returnStatement
 *             | printStatement
 *             | callStatement
 */
statement(opts?:Options):Statement|null {
  if( !opts ){ opts = this.options }
  this.log('statement() ptr=',this.ptr.gets(3))

  const remStmt = this.remStatement(opts)
  if( remStmt )return remStmt

  const letStmt = this.letStatement(opts)
  if( letStmt )return letStmt

  const runStmt = this.runStatement(opts)
  if( runStmt )return runStmt

  const gotoStmt = this.gotoStatement(opts)
  if( gotoStmt )return gotoStmt
```

```
const ifStmt = this.ifStatement(opts)
if( ifStmt )return ifStmt

const gosubStmt = this.gosubStatement(opts)
if( gosubStmt )return gosubStmt

const returnStmt = this.returnStatement(opts)
if( returnStmt )return returnStmt

const printStmt = this.printStatement(opts)
if( printStmt )return printStmt

const callStmt = this.callStatement(opts)
if( callStmt )return callStmt

return null
}
```

BASIC. TS. letStatement()

```
/**
 * letStatement ::= [ SourceLineBeginLex | NumberLex ]
 *                StatementLex(LET) IDLex OperatorLex(=) expression
 */
letStatement(opts?:Options):LetStatement|null {
  if( !opts ){ opts = this.options }
  if( this.ptr.eof )return null

  const prod = (arg?:{line:number,lex:Lex}) => { ...
  }

  if( opts.tryLineNum ){
    return this.matchLine(prod) || prod()
  }else{
    return prod()
  }
}
```


BASIC. TS. letStatement() continue

```
/**  
 * letStatement ::= [ SourceLineBeginLex | NumberLex ]  
 *                 StatementLex(LET) IDLex OperatorLex(=) expression  
 */
```

```
const lexId = this.ptr.get()  
if( lexId instanceof IDLex ){  
  const lxNext = this.ptr.get(1)  
  if( lxNext instanceof OperatorLex && lxNext.keyWord == '=' ){  
    this.ptr.move(2)  
    const exp = this.expression()  
    if( exp ){  
      const begin = arg ? arg.lex : lexLet  
      let end = exp.rightTreeLex || begin  
      this.ptr.drop()  
      return new LetStatement(begin, end, lexId, exp)  
    }  
  }  
}
```

The diagram illustrates the mapping of grammar symbols to code variables:

- Red line:** Connects `SourceLineBeginLex` to `lexId`.
- Blue line:** Connects `IDLex` to `lexId` and `OperatorLex(=)` to `lxNext`.
- Orange line:** Connects `expression` to `exp`.

BASIC. TS. LetStatement class

```
export class LetStatement extends Statement {
  readonly kind:string
  readonly begin:Lex
  readonly end:Lex
  readonly let?:Lex
  readonly variable:IDLex
  readonly value:Expression
  constructor(begin:Lex, end:Lex, variable:IDLex, value:Expression){
    super()
    this.kind = 'Let'
    this.begin = begin
    this.end = end
    this.variable = variable
    this.value = value
  }
  get varname(){ return this.variable.id }
}
```

```
export abstract class Statement {
  /** ...
  abstract readonly begin:Lex

  /** ...
  abstract readonly end:Lex

  /** ...
  get sourceLine():number|undefined { ...
  }
}
```

Fn^{Syntax}() «функциональные атомы»

Функция синтаксического анализа

```
interface GR<P extends Pointer<?,?,P>, T extends Tok<P>> extends Function<P, Optional<T>>
```

```
/** Токен/Лексема - результат разбора последовательности токенов
```

```
public interface Tok<P extends Pointer<?,?,P>> {
```

```
/** Возвращает начало токена ...*/
```

```
P begin();
```

```
/** Возвращает
```

```
P end();
```

```
}
```

← Лексема

```
/** Указатель на список символов/лексем ...*/
```

```
public interface Pointer<TOK,POS,SELF extends Pointer<TOK,POS,SELF>> extends Comparable<SELF> {
```

```
/** Проверка что указатель находится за границей списка ...*/
```

```
boolean eof();
```

```
/** Получение значения текущего указателя ...*/
```

```
POS position();
```

```
/** Перемещение указателя n позиций вперед/назад ...*/
```

```
SELF move(POS pos);
```

```
/** Предпросмотр n-ой лексемы относительно текущего указателя ...*/
```

```
Optional<TOK> lookup(POS pos);
```

```
/** Выбор минимального указателя ...*/
```

```
public static <TOK,POS,SELF extends Pointer<TOK,POS,SELF>> SELF min( SELF ... ptrs ){...}
```

```
/** Выбор максимальный указателя ...*/
```

```
public static <TOK,POS,SELF extends Pointer<TOK,POS,SELF>> SELF max( SELF ... ptrs ){...}
```

```
}
```

Указатель →
Передается
в функцию

Fn^{Syntax}() «Функциональные операции»

TASK123-v2 — парсинг числа

Определим что у нас будет чем

```
/** Указатель на поток символов строки */  
public class CharPointer implements Pointer<Character,Integer,CharPointer> {
```

← Указателем на символы

```
/* Токен соответствующий последовательности символов */  
public class CToken implements Tok<CharPointer> {  
    private CharPointer begin;  
    private CharPointer end;
```

← Что является лексемой

```
/** Создает грамматическое правило из предиката ...*/  
GR<CharPointer,CToken> test(Predicate<Character> filter){...}
```

← Конвертация из предиката
В грамматическое правило

В нашем словаре есть всего 3 класса символов

Цифра (digit)

Пробел (whitespace)

Десятичная точка (dot) →

```
digit = test(Character::isDigit);  
whitespace = test(Character::isWhitespace);  
GR<CharPointer,CToken> dot = test( c -> c=='.' );
```

Fn^{Syntax}() «Функциональные операции»

TASK123-v2 — парсинг числа продолжение

Лексема — список цифр

```
/** Набор цифр */
public class DigitsToken extends CToken {
    public DigitsToken(CharPointer begin, CharPointer end)
    public DigitsToken(CToken begin, CToken end) { super(b
    public DigitsToken(List<CToken> tokens) { super(tokens
```

```
public static final GR<CharPointer, DigitsToken> digits
    = digit.repeat().map(DigitsToken::new);
```

```
public static final GR<CharPointer, NumberToken> integerNumber
    = digit.repeat().map( digits -> new NumberToken( new DigitsToken(digits) ) );
```

```
public static final GR<CharPointer, NumberToken> floatNumber
    = digits.next(dot).next(digits)
    .map( (intDigits,dot,floatDigits)->new NumberToken(intDigits,floatDigits) );
```

```
public static final GR<CharPointer, NumberToken> number
    = floatNumber.another(integerNumber)
    .map( t->(NumberToken)t );
```

Лексема числа — число из
Одного или двух списков цифр

```
public class NumberToken extends CToken {
    private final DigitsToken integerPart;
    private final DigitsToken floatPart;
```

Формально **digits ::= { digit }**

Иначе цифра (digit) должна быть повторена несколько раз (repeat())
digit.repeat() равноценно { digit }


Формально **floatNumber ::= digits dot digits**

т. е. Сначала идет список цифр (digits),
затем (next) точка (dot), затем (next) список цифр (digits)

Формально **number ::= floatNumber | integerNumber**

Первый вариант floatNumber другой (another) вариант integerNumber

Пример JS выражений



```
# Начальное правило
expression ::= ifOp

# Операторы
ifOp  ::= or '?' or ':' or | or
or    ::= and { '||' and }
and   ::= bitOr { '&&' bitOr }
bitOr ::= bitXor { '|' bitXor }
bitXor ::= bitAnd { '^' bitAnd }
bitAnd ::= equals { '&' equals }
equals ::= compare [ ( '==' | '!=' | '===' | '!== ' ) compare ]
compare ::= bitShift
        [ ( '<' | '<=' | '>' | '>=' | 'in' | 'instanceof' ) bitShift ]
bitShift ::= addSub { ('<<' | '>>' | '>>>') addSub }
addSub  ::= mulDiv { ('+' | '-') mulDiv }
mulDiv  ::= power { ('*' | '/' | '%' ) power }
power   ::= primary { '**' primary }

# Первичные конструкции
primary ::= atom [ postfix ]
postfix ::= { # Доступ к свойству
            '.' idTok
            | # Вызов метода
              '(' [ expression { ',' expression } ] ')'
            | # Доступ к элементу массива
              '[' expression ']'
          }

# Атомарные конструкции
atom ::= parentheses
      | unaryExpression
      | varRef
      | listExpression
      | mapExpression
      | literal

# Группировка операций
parenthes ::= '(' expression ')'
```

Приоритет растёт вниз от expression к parentheses

Правила рекурсивны см. expression

Это даёт возможность писать бесконечно вложенные конструкции:
a + (b — (d + e).toString()).toString().toUpperCase()

Итого

»Что это было«

Форма

Исходный код — не обязательно текст, но AST важно для понимания

Дерево AST

Состоит из терминалов — лексем (листья в AST) и не терминалов — узлы дерева AST

Дерево AST собирается из листьев в дерево согласно набору правил — грамматике (eBNF, PEG, ...)

Грамматика — некий псевдокод из набора функций

Грамматика

Тело функции в своей основе описывается следующими инструкциями

Терминалами/лексемами

Не терминалами — вызов функции

Последовательностью терминалов/не терминалов $\rightarrow a\ b()\ c$

Группировкой инструкций $\rightarrow ()$

Ветвлением инструкций $\rightarrow a\ |\ b$

Повторением $\rightarrow \{ \text{digit} \}$

Опциональными значениями $\rightarrow a\ [\ b]$

Итого

»Что не было«

Не рассмотрено

AST

Грамматики

Не однозначные грамматики

Левая / правая рекурсия

Алгоритмы синтаксического анализа — LL (частично), LR, SLR, GLR

Фазы компиляции

Фаза вывода типов (семантика и т. д.)

Модификация AST — Макросы

Генерация продукта (exe, llvm)

Backend компиляторы и вычислительные устройства

Не рассмотрено

Модели типизации

Статическая/динамическая типизация

Вывод типов

Отладка / DEBUG / Profiling

Инструменты

ANTLR, YACC

DSL / Xtext

Лингвистика

Омонимы/Синонимы

Пределы описания — т. Гёделя

Отличия и Аналогии с естественных языков

Логичность и (не)противоречивость, инструментализм, эволюция

PS Мотивация — или зачем

ЯП — имеют природу Идеального онтологический статус

Так же как и математика они обозначают нечто, что является реальным, но сами таковыми не являются

ЯП — это практики программирования представленные формальными грамматиками

ЯП имеют одновременно несколько концептов, удачная практика закрепляется в виде шаблона, которые если выживают — то закрепляются в языке → эволюция: ASM → C → C++ → Java → Scala → ?

ЯП — явный пример искусственной эволюции

Гены → концепты, есть хорошие и не очень

Критерий выживания → увеличение размера популяции — мы производим их отбор

ЯП — не догма

ЯП — это один из инструментов между проблемой и решением

«Не боги горшки обжигают» - всегда, есть шанс создать свой ЯП