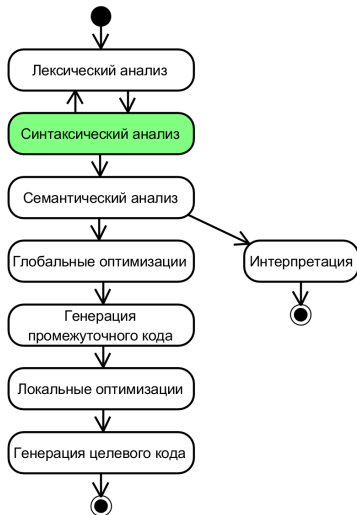


Синтаксический анализ на F#

Юрий Литвинов

17.04.2020г

Фазы компиляции

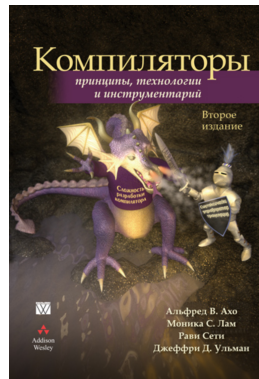


Книжка

Must read

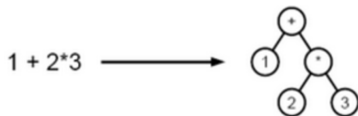
А. Ахо, Р. Сети, Дж. Ульман, М. Лам.
Компиляторы. Принципы, технологии,
инструменты.

- ▶ Так же известна как “Книга дракона” (“Dragonbook”)



Синтаксический анализ

- ▶ Анализ последовательности токенов с целью выяснить синтаксическую структуру
 - ▶ Сопоставление с формальной грамматикой
- ▶ Строит структуру данных, представляющую разобранный по синтаксическим правилам документ
 - ▶ Чаще всего, абстрактное синтаксическое дерево (Abstract Syntax Tree, AST)
 - ▶ Бывает ещё дерево разбора (Parse tree) — содержит все токены из входной строки, в явном виде обычно не строится



Другие задачи синтаксического анализа

- ▶ Диагностика ошибок
- ▶ Восстановление после ошибок
 - ▶ Режим паники
 - ▶ Коррекция
 - ▶ Грамматические правила, обнаруживающие ошибки
 - ▶ “Предсказание ошибок”
- ▶ Привязка — определение для каждой синтаксической конструкции её места в коде

Формальные грамматики

- ▶ Терминал — символ входной строки для синтаксического анализатора (токен)
 - ▶ Для лексического анализа входная строка состоит из букв, для синтаксического — из токенов
- ▶ Нетерминал — объект, представляющий сложную синтаксическую конструкцию
- ▶ Грамматика, формально: (Σ, N, P, S) , где
 - ▶ Σ — множество терминалов
 - ▶ N — множество (алфавит) нетерминальных символов
 - ▶ P — продукции, функции вида «цепочка символов» \rightarrow «цепочка символов», где слева в цепочке есть хотя бы один нетерминал
 - ▶ $P: (\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$
 - ▶ S — стартовый символ, $S \in N$

Пример грамматики

$$E ::= E + E$$
$$| E - E$$
$$| -E$$
$$| (E)$$
$$| \text{NUMBER}$$
$$\text{NUMBER} ::= 1 \mid 2 \mid 3 \mid \dots \mid 9$$

Иерархия Хомского

- ▶ Регулярные языки (языки типа 3) — задаются регулярными выражениями, разбираются конечными автоматами
- ▶ Контекстно-свободные грамматики — грамматики, у которых слева в продукциях может быть только один символ (нетерминал)
 - ▶ Пример с предыдущего слайда — КС-грамматика
 - ▶ Разбираются стековыми автоматами (например, рекурсивным спуском)
- ▶ Контекстно-зависимые грамматики — в левой части может быть нетерминал и “контекст”, нетерминал раскрывается в правой части
 - ▶ Разбираются линейно ограниченными недетерминированными машинами Тьюринга (то есть, всё плохо)
- ▶ Языки типа 0 — грамматики без ограничений на вид продукций
 - ▶ Разбираются машинами Тьюринга (то есть всё очень плохо)

В реальной жизни

- ▶ Регулярные языки — регэкспы, весь лексический анализ
 - ▶ Не умеют считать, поэтому грамматики вида $a^n b^n$ (скобочные последовательности) им не под силу
 - ▶ Не могут в иерархические структуры, никогда не парсите регэкспами HTML
- ▶ Контекстно-свободные грамматики — грамматики большинства современных языков программирования
 - ▶ Не могут в анализ типов
- ▶ Контекстно-зависимые грамматики — грамматика C++ и некоторых неаккуратных мест в других языках
 - ▶ Пример: `A c;` — либо **class** **A**<T> {}, либо **int** A; **int** B; **int** c;
- ▶ Языки типа 0 — естественные языки (да, их тоже анализируют грамматиками, и вообще, Хомский был лингвистом)

Вывод в грамматике

- ▶ Формально, если есть грамматика $G = (\Sigma, N, P, S)$, то вывод, \Rightarrow_G — бинарное отношение на строках
 - ▶ $x \Rightarrow_G y \iff \exists u, v, p, q \in (\Sigma \cup N)^* : (x = upv) \wedge (p \rightarrow q \in P) \wedge (y = uqv)$
 - ▶ Неформально, шаг вывода — применение одной из продукций
- ▶ \Rightarrow_G^* — рефлексивное транзитивное замыкание \Rightarrow_G
 - ▶ $x \Rightarrow_G^* y$ — существует конечная последовательность применений продукций грамматики, которая по x делает y
 - ▶ Говорят, « y выводится из x »
- ▶ *Порождение* — последовательность шагов вывода
- ▶ $L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$ — язык, порождаемый грамматикой G

Пример

Грамматика:

$E ::= E + E$
| $E * E$
| $-E$
| (E)
| id

Входная строка: $-(id + id)$

Порождения:

- ▶ Левое: $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id)$
- ▶ Правое: $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + id) \Rightarrow -(id + id)$

Левая рекурсия

Проблема:

$$A \rightarrow Aa \mid b$$

Пример:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

Решение:

$$A \rightarrow bA'$$

$$A' \rightarrow aA' \mid \epsilon$$

Пример:

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon'$$

$$F \rightarrow (E) \mid id$$

Неоднозначность

Строка: $id + id * id$

Грамматика (как была): $E ::= E + E \mid E * E \mid -E \mid (E) \mid id$

Вывод:

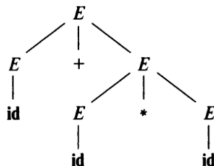
$$E \Rightarrow E + E$$

$$\Rightarrow id + E$$

$$\Rightarrow id + E * E$$

$$\Rightarrow id + id * E$$

$$\Rightarrow id + id * id$$



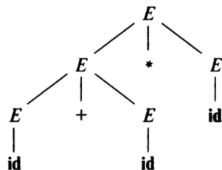
$$E \Rightarrow E * E$$

$$\Rightarrow E + E * E$$

$$\Rightarrow id + E * E$$

$$\Rightarrow id + id * E$$

$$\Rightarrow id + id * id$$

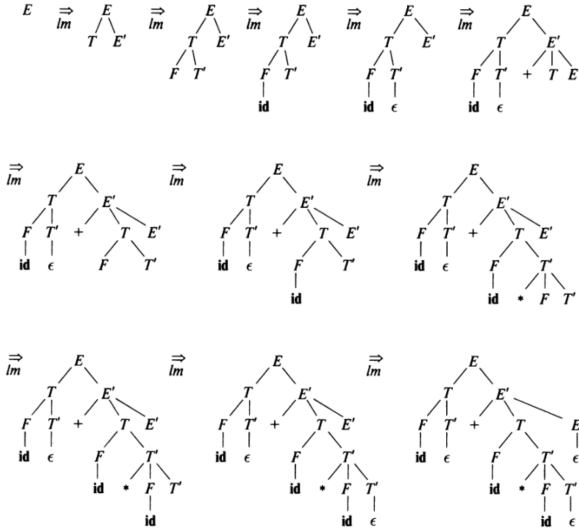


Алгоритмы разбора

- ▶ Нисходящий разбор — начинаем со стартового нетерминала, пытаемся построить входную строку
 - ▶ Рекурсивный спуск
 - ▶ LL-анализ
- ▶ Восходящий разбор — пытаемся найти во входной строке последовательность терминалов и нетерминалов и свернуть её в нетерминал
 - ▶ LR-анализ

Пример

Нисходящий разбор с построением левого порождения



FIRST(α) и FOLLOW(α)

Пусть α — строка из нетерминалов и терминалов

- ▶ FIRST(α) — множество всех терминалов, с которых может начинаться α
 - ▶ Считается рекурсивно, раскрытием нетерминальных символов
- ▶ FOLLOW(α) — множество всех терминалов, которые могут стоять за α в выводе в грамматике G
 - ▶ Считается через FIRST во всех цепочках выводов, в которых может встречаться α
 - ▶ ϵ -продукции требуют особого внимания

Зачем:

- ▶ FIRST позволяет выбрать из альтернативных продукций
- ▶ FOLLOW — чтобы выбрать между ϵ -продукцией и какой-то другой

Рекурсивный спуск

- ▶ По одной функции на нетерминал
- ▶ Просмотр строки слева направо

Выбираем продукцию $A \rightarrow X_1 X_2 \dots X_k$;

for i от 1 до k **do**

if X_i – нетерминал **then**

 Вызов функции $X_i()$;

else if X_i равно текущему символу a **then**

 Переходим к следующему символу;

else

 Обнаружена ошибка;

end

end

BNF

Форма Бэкуса-Наура

- ▶ В угловых скобках — нетерминал (`<literal>`)
- ▶ `::=` — определение (`<brackets> ::= '(' | ')'`)
- ▶ `|` — альтернатива

Пример: `<expr> ::= <term> | <expr> <addop> <term>`

Пример

BNF, записанная в синтаксисе BNF

```
<syntax> ::= <rule> | <rule> <syntax>
<rule> ::= <opt-whitespace> "<" <rule-name> ">" <opt-whitespace>
        "::=" <opt-whitespace> <expression> <line-end>
<opt-whitespace> ::= " " <opt-whitespace> | ""
<expression> ::= <list> | <list> <opt-whitespace> "|" <opt-whitespace> <expression>
<line-end> ::= <opt-whitespace> <EOL> | <line-end> <line-end>
<list> ::= <term> | <term> <opt-whitespace> <list>
<term> ::= <literal> | "<" <rule-name> ">"
<literal> ::= "" <text1> "" | "" <text2> ""
<text1> ::= "" | <character1> <text1>
<text2> ::= " | <character2> <text2>
<character> ::= <letter> | <digit> | <symbol>
<character1> ::= <character> | ""
<character2> ::= <character> | ""
<rule-name> ::= <letter> | <rule-name> <rule-char>
<rule-char> ::= <letter> | <digit> | "-"
```

Расширенная форма Бэкуса-Наура

- ▶ $\{ \}$ — 0 или более повторений
- ▶ $[]$ — 0 или 1 раз (опционально)
- ▶ $()$ — группировка
- ▶ $,$ — конкатенация
- ▶ Вариантов синтаксиса EBNF больше, чем звёзд на небе

Пример

BNF, записанная в синтаксисе BNF

```
character = letter | digit | symbol | "_" ;
identifier = letter , { letter | digit | "_" } ;
terminal = "" , character , { character } , ""
          | "" , character , { character } , "" ;
lhs = identifier ;
rhs = identifier
    | terminal
    | "[" , rhs , "]"
    | "{" , rhs , "}"
    | "(" , rhs , ")"
    | rhs , "|" , rhs
    | rhs , ",", rhs ;
rule = lhs , "=", rhs , ";" ;
grammar = { rule } ;
```

Парсер-комбинаторы

- ▶ Основная идея — а давайте рассматривать парсер как композицию более простых парсеров
 - ▶ Определим примитивные парсеры и комбинаторы, строящие парсеры по парсерам
- ▶ По сути, удобная запись рекурсивного спуска
 - ▶ Не всегда, иногда используются “настоящие” преобразования грамматик
 - ▶ Например, Meerkat
- ▶ Пример — FParsec
 - ▶ Порт известной библиотеки Parsec (OCaml)
 - ▶ Рассмотрим <http://www.quanttec.com/fparsec/tutorial.html>

Пример

open FParsec

[<EntryPoint>]

let main argv =

let result = "1.23" |> (run pfloat)

match result **with**

| Success(result, _, _) -> printfn "%f" result

| Failure(e, _, _) -> printfn "%s" e

0

Комбинаторы конкатенации

```
val (>>.): Parser<'a,'u> -> Parser<'b,'u> -> Parser<'b,'u>
```

```
val (.>>): Parser<'a,'u> -> Parser<'b,'u> -> Parser<'a,'u>
```

```
let str s = pstring s
```

```
let floatBetweenBrackets = str "[" >>. pfloat .>> str "]"
```

```
val pstring: string -> Parser<string, 'u>
```


Что получилось

F# Interactive

```
> test floatBetweenBrackets "[1.0]";;
```

Success: 1.0

```
> test floatBetweenBrackets "[]";;
```

Failure: Error in Ln: 1 Col: 2

```
[]  
^
```

Expecting: floating-point number

```
> test floatBetweenBrackets "[1.0]";;
```

Failure: Error in Ln: 1 Col: 5

```
[1.0  
  ^
```

Note: The error occurred at the end of the input stream.

Expecting: ']'

Свои комбинаторы

```
let betweenStrings s1 s2 p = str s1 >>. p .>> str s2
```

```
let floatBetweenBrackets = pfloat |> betweenStrings "[" "]"
```

```
let floatBetweenDoubleBrackets = pfloat |> betweenStrings "[[" ""]]"
```

или

```
let between pBegin pEnd p = pBegin >>. p .>> pEnd
```

```
let betweenStrings s1 s2 p = p |> between (str s1) (str s2)
```

between — библиотечный комбинатор, его определять не надо

Разбор списков

Грамматика: ("["float "]")*

Парсер:

F# Interactive

```
> test (many floatBetweenBrackets) "";;
```

```
Success: []
```

```
> test (many floatBetweenBrackets) "[1.0]";;
```

```
Success: [1.0]
```

```
> test (many floatBetweenBrackets) "[2][3][4]";;
```

```
Success: [2.0; 3.0; 4.0]
```

```
> test (many floatBetweenBrackets) "[1][2.0E]";;
```

```
Failure: Error in Ln: 1 Col: 9
```

```
[1][2.0E]
```

```
^
```

```
Expecting: decimal digit
```

Один или больше элементов

Грамматика: ("["float "]")+

Парсер:

F# Interactive

```
> test (many1 floatBetweenBrackets) "(1)";;
```

```
Failure: Error in Ln: 1 Col: 1
```

```
(1)
```

```
^
```

```
Expecting: '['
```

Обработка ошибок

F# Interactive

```
> test (many1 (floatBetweenBrackets  
               <?> "float between brackets")) "(1)";;
```

Failure: Error in Ln: 1 Col: 1

(1)

^

Expecting: float between brackets

Ещё пример

Грамматика: `"[" (float ("," float)*)? "]"`

Примеры: `"["`, `"[1.0]"`, `"[2,3,4]"`

Парсер:

```
let floatList = str "[" >>. sepBy pfloat (str ",") .>> str "]"
```

Что получилось:

F# Interactive

```
> test floatList "[";;
```

```
Success: []
```

```
> test floatList "[1.0]";;
```

```
Success: [1.0]
```

```
> test floatList "[4,5,6]";;
```

```
Success: [4.0; 5.0; 6.0]
```

Пробелы

Проблема:

F# Interactive

```
> test floatBetweenBrackets "[1.0, 2.0]";;
```

Failure: Error **in** Ln: 1 Col: 5

```
[1.0, 2.0]
```

^

Expecting: ']'

Решение:

```
let ws = spaces
```

```
let str_ws s = pstring s .>> ws
```

```
let float_ws = pfloat .>> ws
```

```
let numberList = str_ws "[" >>. sepBy float_ws (str_ws ",") .>> str_ws "]"
```

Что получилось

F# Interactive

```
> test numberList @"[ 1 ,  
                2 ]";;
```

Success: [1.0; 2.0]

```
> test numberList @"[ 1,  
                2; 3]";;
```

Failure: Error **in** Ln: 2 Col: 27
2; 3]
 ^

Expecting: ',' or ']'

Парсинг строк

F# Interactive

```
> test (many (str "a" <|> str "b")) "abba";;
```

```
Success: ["a"; "b"; "b"; "a"]
```

```
> test (skipStringCI "<float>" >>. pfloat) "<FLOAT>1.0";;
```

```
Success: 1.0
```

```
let identifier =
```

```
    let isIdentifierFirstChar c = isLetter c || c = '_'
```

```
    let isIdentifierChar c = isLetter c || isDigit c || c = '_'
```

```
    many1Satisfy2L isIdentifierFirstChar isIdentifierChar "identifier"
```

```
    .>> ws // skips trailing whitespace
```

Есть встроенный парсер identifier

Строковые литералы

Грамматика:

stringLiteral: "" (normalChar|escapedChar)* ""

normalChar: any char except '\' and ""

escapedChar: '\' ('\\"'|""'|'n'|'r'|'t')

Парсер:

let stringLiteral =

let normalChar = satisfy (fun c -> c <> '\\' && c <> "")

let unescape c = match c with

| 'n' -> '\n'

| 'r' -> '\r'

| 't' -> '\t'

| c -> c

let escapedChar = pstring "\\" >>. (anyOf "\\nrt\\" |>> unescape)

between (pstring "\"") (pstring "\"")

(manyChars (normalChar <|> escapedChar))

Комбинирование результатов

```
val pipe2: Parser<'a,'u> -> Parser<'b,'u>  
    -> ('a -> b -> 'c) -> Parser<'c,'u>
```

```
let product = pipe2 float_ws (str_ws "*" >>. float_ws)  
    (fun x y -> x * y)
```

Что получилось:

F# Interactive

```
> test product "3 * 5";;  
Success: 15.0
```

Ещё комбинаторы

```
type StringConstant = StringConstant of string * string
```

```
let stringConstant = pipe3 identifier (str_ws "=") stringLiteral  
    (fun id _ str -> StringConstant(id, str))
```

Что получилось:

F# Interactive

```
> test stringConstant "myString = \"stringValue\"";;  
Success: StringConstant ("myString", "stringValue")
```

```
fun tuple2 p1 p2 = pipe2 p1 p2 (fun x1 x2 -> (x1, x2))
```

F# Interactive

```
> test (float_ws .>>. (str_ws "," >>. float_ws)) "123, 456";;  
Success: (123.0, 456.0)
```

Разбор альтернатив

val (<|>): Parser<'a,'u> -> Parser<'a,'u> -> Parser<'a,'u>

```
let boolean = (stringReturn "true" true)  
               <|> (stringReturn "false" false)
```

Что получилось:

F# Interactive

```
> test boolean "false";;  
Success: false  
> test boolean "true";;  
Success: true  
> test boolean "tru";;  
Failure: Error in Ln: 1 Col: 1  
tru  
^  
Expecting: 'false' or 'true'
```

Важные особенности

- ▶ `<|>` применяет правую часть, только если левая пофэйлилась и не использовала ни один символ из входного потока
- ▶ `<|>` не делает никакой предпросмотр
- ▶ `<|>` не ищет самую длинную подходящую строку

Пример:

F# Interactive

```
> run (pstring "a" <|> pstring "ab") "ab";;  
val it : ParserResult<string,unit> = Success: "a"
```

```
> test ((ws >>. str "a") <|> (ws >>. str "b")) " b";;  
Failure: Error in Ln: 1 Col: 2
```

```
b  
^
```

Expecting: 'a'

Как пофиксить

F# Interactive

```
> test (ws >>. (str "a" <|> str "b")) " b";;  
Success: "b"
```

Ещё комбинаторы

```
p1 <|> p2 <|> p3  
choice [p1; p2; p3]
```

val attempt: Parser<'a,'u> -> Parser<'a,'u>

F# Interactive

```
run ((attempt ab) <|> ac) "ac";;  
val it : ParserResult<(string * string),unit> = Success: ("a", "c")
```

Комбинаторы бэктрекинга:

```
val (>>?): Parser<'a,'u> -> Parser<'b,'u> -> Parser<'b,'u>  
(.>>?): Parser<'a,'u> -> Parser<'b,'u> -> Parser<'a,'u>  
val (>>.<?): Parser<'a,'u> -> Parser<'b,'u> -> Parser<('a * 'b),'u>
```


Большой пример: парсер JSON

► Грамматика: <https://www.json.org/json-en.html>

► Пример:

```
{  
  "employee":{ "name":"John", "age":30, "city":"New York" }  
  "array": [ "John", "Anna", "Peter" ]  
}
```

https://www.w3schools.com/js/js_json_datatypes.asp

► AST:

```
type Json = JString of string  
          | JNumber of float  
          | JBool   of bool  
          | JNull  
          | JList   of Json list  
          | JObject of Map<string, Json>
```

Элементарные типы

```
let jnull = stringReturn "null" JNull
let jtrue = stringReturn "true" (JBool true)
let jfalse = stringReturn "false" (JBool false)
let jnumber = pfloat |>> JNumber
```

Строки

```
let str s = pstring s
```

```
let stringLiteral =
    let escape = anyOf "\"\\bfnrt"
    |>> function
        | 'b' -> "\"b"
        | 'f' -> "\"u000C"
        | 'n' -> "\"n"
        | 'r' -> "\"r"
        | 't' -> "\"t"
        | c -> string c // every other char is mapped to itself
```

```
let unicodeEscape =
    ^^1/// converts a hex char ([0-9a-fA-F]) to its integer number (0-15)
    let hex2int c = (int c &&& 15) + (int c >>> 6)*9

    str "u" >>. pipe4 hex hex hex hex (fun h3 h2 h1 h0 ->
        (hex2int h3)*4096 + (hex2int h2)*256 + (hex2int h1)*16 + hex2int h0
        |> char |> string
    )
```

```
let escapedCharSnippet = str "\"\" >>. (escape <|> unicodeEscape)
let normalCharSnippet = manySatisfy (fun c -> c <> "\"" && c <> "\\')
```

```
between (str "\"\"") (str "\"\"")
    (stringsSepBy normalCharSnippet escapedCharSnippet)
```

```
let jstring = stringLiteral |>> JString
```

Рекурсия

```
let jvalue, jvalueRef = createParserForwardedToRef<Json, unit>()
```

- ▶ jvalue — парсер, который просто вызывает парсер из ref-ячейки jvalueRef
- ▶ Изначально там парсер, который ничего не делает
- ▶ Поскольку ref-ячейка мутабельна, присвоим туда настоящий парсер позже

Списки и объекты

```
let ws = spaces
```

```
let listBetweenStrings sOpen sClose pElement f =  
    between (str sOpen) (str sClose)  
        (ws >>. sepBy (pElement .>> ws) (str "," >>. ws) |>> f)
```

```
let jlist = listBetweenStrings "[" "]" jvalue JList
```

```
let keyValue = stringLiteral .>>. (ws >>. str ":" >>. ws >>. jvalue)
```

```
let jobject = listBetweenStrings "{" "}" keyValue (Map.ofList >> JObject)
```

Соберём всё воедино

```
do jvalueRef := choice [jobject
    jlist
    jstring
    jnumber
    jtrue
    jfalse
    jnull]

let json = ws >>. jvalue .>> ws .>> eof
```

Заключение

- ▶ Подробности: <http://www.quanttec.com/fparsec/tutorial.html>
- ▶ Ещё большие подробности:
<http://www.quanttec.com/fparsec/users-guide/>
- ▶ И ещё большие подробности:
<http://www.quanttec.com/fparsec/reference/>
- ▶ Осталось за бортом:
 - ▶ FsLex/FsYacc — неидиоматичный, но более “взрослый” генератор парсеров
 - ▶ <https://fsprojects.github.io/FsLexYacc/>
 - ▶ ANTLR — стандарт де-факто в серьёзном синтаксическом анализе
 - ▶ <https://www.antlr.org/>
 - ▶ Не поддерживает F#, но с C# всё ок
 - ▶ YaccConstructor — мощная библиотека и генератор парсеров, для исследовательских целей
 - ▶ <https://github.com/YaccConstructor/YaccConstructor>
 - ▶ Написан на F#
 - ▶ Разрабатывается на матмехе