

Синтаксический анализ на F#

Часть 2: FParsec vs FsLex/FsYacc

Юрий Литвинов
y.litvinov@spbu.ru

16.04.2026

Арифметический интерпретатор

Задача: разработать интерпретатор арифметических выражений

- ▶ Должны поддерживаться
 - ▶ +, -, *, / (с приоритетами операций)
 - ▶ Унарный минус
 - ▶ Скобки
 - ▶ Целые числа
- ▶ По входной строке надо явно построить AST
- ▶ По построенному AST вычислить выражение

Грамматика

$E ::= E + E$

| $E - E$

| $E * E$

| E / E

| $-E$

| (E)

| NUMBER

NUMBER ::= [0..9]+

Подготовительная работа

- ▶ Создаём проект
- ▶ Добавляем ссылку на FParsec в проект
- ▶ Убеждаемся, что всё работает

open FParsec

```
[<EntryPoint>]
let main argv =
    let result = "1.23" |> (run pfloat)
    printfn "%A" result
    0
```

Представление AST

```
type Expression =
| Plus of Expression * Expression
| Minus of Expression * Expression
| Multiplication of Expression * Expression
| Division of Expression * Expression
| Negation of Expression
| Number of int
```

Начнём с Number

```
let number = digit
```

```
let testInput = "9"  
printfn "%A" (testInput |> run number)
```

```
let testInput = "12"  
printfn "%A" (testInput |> run number)
```

Вывод

Success: '9'

Success: '1'

Позитивное замыкание, правильный разбор числа

```
let number = many1 digit
```

```
let testInput = "9"  
printfn "%A" (testInput |> run number)
```

```
let testInput = "12"  
printfn "%A" (testInput |> run number)
```

Вывод

```
Success: ['9']  
Success: ['1'; '2']
```

Делаем узел дерева

```
let number =
    many1 digit
|>> (List.fold (fun acc x -> acc * 10 + int (x.ToString())) 0) >> Number
```

```
let testInput = "9"
printfn "%A" (testInput |> run number)
```

```
let testInput = "12"
printfn "%A" (testInput |> run number)
```

Вывод

Success: Number 9

Success: Number 12

Рекурсивные правила

```
let expression, expressionRef = createParserForwardedToRef()
```

```
let negation = pchar '-' >> expression |>> Negation
```

```
expressionRef := choice [negation; number]
```

```
let testInput = "-9"
```

```
printfn "%A" (testInput |> run expression)
```

```
let testInput = "--12"
```

```
printfn "%A" (testInput |> run expression)
```

Вывод

Success: Negation (Number 9)

Success: Negation (Negation (Number 12))

Победим пробелы

```
let expression, expressionRef = createParserForwardedToRef()
let negation =
    pchar '-' .>> spaces >>. expression .>> spaces |>> Negation
expressionRef := choice [negation; number]
```

```
let testInput = "- 9"
printfn "%A" (testInput |> run expression)
```

Вывод

Success: Negation (Number 9)

Сложение!

Наивный подход-1

```
let expression, expressionRef = createParserForwardedToRef()
let negation =
```

```
    pchar '-' .>> spaces >>. expression .>> spaces |>> Negation
```

```
let plus = expression .>> pchar '+' .>>. expression |>> Plus
```

```
expressionRef := choice [negation; number; plus]
```

```
let testInput = "1 + 2"
```

```
printfn "%A" (testInput |> run expression)
```

Вывод

Success: Number 1

Сложение!

Наивный подход-2

```
let expression, expressionRef = createParserForwardedToRef()
let negation =
    pchar '-' .>> spaces >>. expression .>> spaces |>> Negation
let plus = expression .>> pchar '+' .>>. expression |>> Plus
```

expressionRef := choice [negation; plus; number]

```
let testInput = "1 + 2"
printfn "%A" (testInput |> run expression)
```



Вывод

Stack overflow.

Факторизуем грамматику

$E ::= \text{PRIMARY } E'$

$E' ::= + \text{PRIMARY } E'$

| - $\text{PRIMARY } E'$

| * $\text{PRIMARY } E'$

| / $\text{PRIMARY } E'$

| e

$\text{PRIMARY} ::= -E$

| (E)

| NUMBER

$\text{NUMBER} ::= [0..9]^+$

Перепишем парсер

```
let expression, expressionRef = createParserForwardedToRef()
let negation =
    pchar '-' .>> spaces >>. expression .>> spaces |>> Negation
let brackets =
    pchar '(' .>> spaces >>. expression .>> spaces .>> pchar ')' .>> spaces

let primary =
    negation
    <|> brackets
    <|> number

let expression', expression'Ref = createParserForwardedToRef()
expression'Ref := pchar '+' >>. primary .>>. expression' |>> ???

AST строить неудобно!
```

Введём промежуточное представление дерева

Parse tree

type Primary =

- | Negation **of** E
- | Brackets **of** E
- | Number **of** int

and E =

- | E **of** Primary * E'

and E' =

- | Plus **of** Primary * E'
- | Minus **of** Primary * E'
- | Multiplication **of** Primary * E'
- | Division **of** Primary * E'
- | Epsilon

Теперь уже перепишем парсер (1)

Чтобы он строил Parse tree

```
let e, eRef = createParserForwardedToRef()
```

```
let negation = pchar '-' .>> spaces >>. e .>> spaces |>> Negation
```

```
let brackets =
```

```
    pchar '(' .>> spaces >>. e .>> spaces .>> pchar ')' .>> spaces  
|>> Brackets
```

```
let primary =  
    negation  
<|> brackets  
<|> number
```

Теперь уже перепишем парсер (2)

E' и всё вместе

```
let e', e'Ref = createParserForwardedToRef()
e'Ref :=
  (pchar '+' >> spaces >> primary .>> spaces .>> e' |>> Plus)
  <|> (pchar '-' >> spaces >> primary .>> spaces .>> e' |>> Minus)
  <|> (pchar '*' >> spaces >> primary .>> spaces .>> e' |>> Multiplication)
  <|> (pchar '/' >> spaces >> primary .>> spaces .>> e' |>> Division)
  <|> preturn Epsilon
```

e'Ref := primary .>> spaces .>> e' |>> E

```
let testInput = "1 + 2"
printfn "%A" (testInput |> run e)
```

Вывод

Success: E (Number 1,Plus (Number 2,Epsilon))

Небольшой рефакторинг

```
let (!) parser = parser .>> spaces
```

```
let e', e'Ref = createParserForwardedToRef()
```

```
e'Ref :=
```

```
  (!!(pchar '+') >>. !primary .>>. !e' |>> Plus)  
<|> (!!(pchar '-') >>. !primary .>>. !e' |>> Minus)  
<|> (!!(pchar '*') >>. !primary .>>. !e' |>> Multiplication)  
<|> (!!(pchar '/') >>. !primary .>>. !e' |>> Division)  
<|> preturn Epsilon
```

```
eRef := !primary .>>. !e' .>> eof |>> E
```

Приоритет операций, проблема

```
let testInput = "1 + 2 * 3"  
printfn "%A" (testInput |> run e)
```

```
let testInput = "1 * 2 + 3"  
printfn "%A" (testInput |> run e)
```

Вывод

```
Success: E (Number 1,Plus (Number 2,Multiplication (Number 3,Epsilon)))  
Success: E (Number 1,Multiplication (Number 2,Plus (Number 3,Epsilon)))
```

Алгоритм сортировочной станции? Нет! У нас есть вся мощь формальных языков и библиотека парсер-комбинаторов

Ещё раз подправим грамматику

$E ::= \text{TERM } E'$

$E' ::= + \text{ TERM } E'$

| - $\text{TERM } E'$

| e

$\text{TERM} ::= \text{FACTOR } \text{TERM}'$

$\text{TERM}' =$

| * $\text{FACTOR } \text{TERM}'$

| / $\text{FACTOR } \text{TERM}'$

| e

$\text{FACTOR} ::= -E$

| (E)

| NUMBER

$\text{NUMBER} ::= [0..9]^+$

Приведём Parse Tree в соответствие

```

type E =
| E of Term * E'
and E' =
| Plus of Term * E'
| Minus of Term * E'
| Epsilon
and Term =
| Term of Factor * Term'
and Term' =
| Multiplication of Factor * Term'
| Division of Factor * Term'
| Epsilon
and Factor =
| Negation of E
| Brackets of E
| Number of int

```

И сам парсер

```
let e, eRef = createParserForwardedToRef()
```

```
let factor = !(pchar '-') >> .le |>> Negation
    <|> (!(pchar '(') >> .le .>> !(pchar ')')) |>> Brackets)
    <|> number
```

```
let term', term'Ref = createParserForwardedToRef()
```

```
term'Ref := !(pchar '*') >> .ifactor .>> .!term' |>> Multiplication
    <|> (!(pchar '/') >> .ifactor .>> .!term' |>> Division)
    <|> preturn Epsilon
```

```
let term = .ifactor .>> .!term' |>> Term
```

```
let e', e'Ref = createParserForwardedToRef()
```

```
e'Ref :=
    !(pchar '+') >> .!term .>> .le' |>> Plus
    <|> !(pchar '-') >> .!term .>> .le' |>> Minus)
    <|> preturn E'.Epsilon
```

```
eRef := .!term .>> .le' |>> E
```

Теперь

```
let testInput = "1 + 2 * 3"  
printfn "%A" (testInput |> run e)
```

```
let testInput = "1 * 2 + 3"  
printfn "%A" (testInput |> run e)
```

Вывод

```
Success: E (Term (Number 1,Epsilon),  
           Plus (Term (Number 2,Multiplication (Number 3,Epsilon)),Epsilon))  
Success: E (Term (Number 1,Multiplication (Number 2,Epsilon)),  
           Plus (Term (Number 3,Epsilon),Epsilon))
```

Сложнее, но тут уже получилась некоторая структура

Построим AST по Parse Tree

Сначала Factor

```
let rec buildAST expr =
    let buildFactor = function
        | Negation(e) -> Expression.Negation(buildAST e)
        | Brackets(e) -> buildAST e
        | Number(x) -> Expression.Number(x)

()
```

Построим AST по Parse Tree

Теперь термы

```
let rec buildTerm' acc = function
```

```
| Multiplication(factor, rest) ->  
    buildTerm' (Expression.Multiplication(acc, buildFactor factor)) rest  
| Division(factor, rest) ->  
    buildTerm' (Expression.Division(acc, buildFactor factor)) rest  
| Epsilon -> acc
```

```
let buildTerm (Term(factor, rest)) = buildTerm' (buildFactor factor) rest
```

Построим AST по Parse Tree

А теперь и всё выражение

```
let rec buildE' acc = function
| Plus(factor, rest) ->
    buildE' (Expression.Plus(acc, buildTerm factor)) rest
| Minus(factor, rest) ->
    buildE' (Expression.Minus(acc, buildTerm factor)) rest
| E'.Epsilon -> acc
```

```
let buildE (E(term, rest)) = buildE' (buildTerm term) rest
```

```
buildE expr
```

Потестим

```
let testInput = "1 * 2 + 3"
```

```
let result = testInput |> run e
printfn "%A" result
```

```
match result with
| Success(result, _, _) -> printfn "%A" <| buildAST result
| _ -> printfn "%A" result
```

Вывод

Success: E (Term (Number 1,Multiplication (Number 2,Epsilon)),
 Plus (Term (Number 3,Epsilon),Epsilon))
 Plus (Multiplication (Number 1,Number 2),Number 3)

Что дальше

- ▶ А считать выражение по такому дереву мы уже умеем
- ▶ Что в итоге получилось: <https://gist.github.com/yurii-litvinov/3b8b9e9328e06ac49d15481ba2cb3684>
- ▶ Что ещё умеет FParsec: <https://www.quanttec.com/fparsec/tutorial.html>
- ▶ Полное описание API библиотеки:
<https://www.quanttec.com/fparsec/reference/>
- ▶ Монады!
<https://www.quanttec.com/fparsec/users-guide/where-is-the-monad.html>
- ▶ Как на самом деле парсить арифметические выражения:
<https://www.quanttec.com/fparsec/reference/operatorprecedenceparser.html>

FsLex/FsYacc, подготовительная работа

- ▶ Создаём проект
- ▶ Добавляем ссылку на nuget-пакет FsLexYacc
- ▶ Создаём новый файл, Lexer.fsl
- ▶ Пишем туда

```
{  
open FSharp.Text.Lexing  
open System  
  
let lexeme = LexBuffer<_>.LexemeString  
}  
  
let digit = ['0'-'9']  
  
rule token = parse  
| digit+ { Int32.Parse(lexeme lexbuf) }
```

FsLex/FsYacc, подготовительная работа (2)

- Добавляем в проектный файл описание лексера

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
</PropertyGroup>

<ItemGroup>
    <FsLex Include="Lexer.fsl">
        <OtherFlags>--module Lexer --unicode</OtherFlags>
    </FsLex>
    <Compile Include="Program.fs" />
</ItemGroup>

<ItemGroup>
    <PackageReference Include="FsLexYacc" Version="10.0.0" />
</ItemGroup>

</Project>
```

FsLex/FsYacc, подготовительная работа (3)

- ▶ Компилируем проект и добавляем в проект появившийся `Lexer.fs`
- ▶ Почему так извращённо — потому что обычно для сборки используют `fake`

```
<ItemGroup>
<Compile Include="Lexer.fs" />
<FsLex Include="Lexer.fsl">
  <OtherFlags>--module Lexer --unicode</OtherFlags>
</FsLex>
<Compile Include="Program.fs" />
</ItemGroup>
```

FsLex/FsYacc, подготовительная работа (4)

- ▶ Пишем точку входа

```
open FSharp.Text.Lexing
```

```
[<EntryPoint>]
```

```
let main argv =
```

```
    let testInput = "9"
```

```
    let lexbuf = LexBuffer<char>.FromString testInput
```

```
    let tokens = Lexer.token lexbuf
```

```
    printfn "%A" tokens
```

```
0
```

Вывод

```
9
```

Теперь то же самое с парсером

- ▶ Создаём Parser.fsy

```
%{  
%}  
}%
```

```
%token <int> INT
```

```
%start start  
%type <int> start
```

```
%%
```

```
start: INT { $1 }
```

Добавляем в проектный файл, руками

До лексера!

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <FsYacc Include="Parser.fsy">
      <OtherFlags>--module Parser</OtherFlags>
    </FsYacc>
    <FsLex Include="Lexer.fsl">
      <OtherFlags>--module Lexer --unicode</OtherFlags>
    </FsLex>
    <Compile Include="Lexer.fs" />
    <Compile Include="Program.fs" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="FsLexYacc" Version="10.0.0" />
  </ItemGroup>
</Project>
```

Собираем и добавляем Parser.fs

Тоже до лексера!

```
<Project Sdk="Microsoft.NET.Sdk">
...
<ItemGroup>
  <FsYacc Include="Parser.fsy">
    <OtherFlags>--module Parser</OtherFlags>
  </FsYacc>
  <Compile Include="Parser.fs" />
  <FsLex Include="Lexer.fsl">
    <OtherFlags>--module Lexer --unicode</OtherFlags>
  </FsLex>
  <Compile Include="Lexer.fs" />
  <Compile Include="Program.fs" />
</ItemGroup>
...
</Project>
```

Правим лексер

Чтобы он генерил лексемы для парсера

```
{  
open FSharp.Text.Lexing  
open System  
open Parser  
  
let lexeme = LexBuffer<_>.LexemeString  
}  
  
let digit = ['0'-'9']  
  
rule token = parse  
| digit+ { INT(Int32.Parse(lexeme lexbuf)) }
```

Пробуем, что получилось

В Program.fs

```
open FSharp.Text.Lexing
```

```
[<EntryPoint>]
let main argv =
    let testInput = "9"

    let lexbuf = LexBuffer<char>.FromString testInput
    let ast = Parser.start Lexer.token lexbuf
    printfn "%A" ast
    0
```

Вывод

9

Теперь наконец можно приступать к делу!

Вспомним грамматику

$E ::= E + E$

| $E - E$

| $E * E$

| E / E

| $-E$

| (E)

| NUMBER

NUMBER ::= [0..9] +

Позаимствуем из FParsec AST

В новый файл Ast.fs, до парсера

module Types

type Expression =

- | Plus **of** Expression * Expression
- | Minus **of** Expression * Expression
- | Multiplication **of** Expression * Expression
- | Division **of** Expression * Expression
- | Negation **of** Expression
- | Number **of** int

Требуемые токены

В парсере

```
%{  
open Types  
%}
```

```
%token <int> INT  
%token PLUS  
%token MINUS  
%token MUL  
%token DIV  
%token LPAR  
%token RPAR  
%token EOF
```

...

Поддержка в лексере

...

```
rule token = parse
| digit+ { INT(Int32.Parse(lexeme lexbuf)) }
| '+' { PLUS }
| '-' { MINUS }
| '*' { MUL }
| '/' { DIV }
| '(' { LPAR }
| ')' { RPAR }
| eof { EOF }
```

Грамматика в парсере

Пока просто потестим, разбирается ли

start: expression { 0 }

expression:

- | INT {}
- | expression PLUS expression {}
- | expression MINUS expression {}
- | expression MUL expression {}
- | expression DIV expression {}
- | MINUS expression {}
- | LPAR expression RPAR {}

Попробуем

```
open FSharp.Text.Lexing
```

```
[<EntryPoint>]
```

```
let main argv =
```

```
    let testInput = "1+2*3"
```

```
    let lexbuf = LexBuffer<char>.FromString testInput
```

```
    let ast = Parser.start Lexer.token lexbuf
```

```
    printfn "%A" ast
```

```
    0
```

Вывод

```
0
```

Пробелы?

```
open FSharp.Text.Lexing
```

```
[<EntryPoint>]
let main argv =
    let testInput = "1 + 2 * 3"

    let lexbuf = LexBuffer<char>.FromString testInput
    let ast = Parser.start Lexer.token lexbuf
    printfn "%A" ast
    0
```

Вывод

Unhandled exception. System.Exception: unrecognized input

Чиним, в лексере

```
let digit = ['0'-'9']
let whitespace = [' ' '\t' '\r' '\n']
```

```
rule token = parse
| digit+ { INT(Int32.Parse(lexeme lexbuf)) }
| '+' { PLUS }
| '-' { MINUS }
| '*' { MUL }
| '/' { DIV }
| '(' { LPAR }
| ')' { RPAR }
| whitespace { token lexbuf }
| eof { EOF }
```

Теперь всё работает

Семантические действия

Плюс выкинем start

```
%start expression
```

```
%type <Expression> expression
```

```
%%
```

expression:

```
| INT { Number($1) }  
| expression PLUS expression { Plus($1, $3) }  
| expression MINUS expression { Minus($1, $3) }  
| expression MUL expression { Multiplication($1, $3) }  
| expression DIV expression { Division($1, $3) }  
| MINUS expression { Negation($2) }  
| LPAR expression RPAR { $2 }
```

Запустим

```
open FSharp.Text.Lexing
```

```
[<EntryPoint>]
```

```
let main argv =
```

```
    let testInput = "1 + 2 * 3"
```

```
    let lexbuf = LexBuffer<char>.FromString testInput
```

```
    let ast = Parser.expression Lexer.token lexbuf
```

```
    printfn "%A" ast
```

```
0
```

Вывод

```
Plus (Number 1,Multiplication (Number 2,Number 3))
```

Магия LALR-разбора!

Приоритет операций?

```
[<EntryPoint>]
let main argv =
    let testInput = "1 * 2 + 3"

    let lexbuf = LexBuffer<char>.FromString testInput
    let ast = Parser.expression Lexer.token lexbuf
    printfn "%A" ast
    0
```

Вывод

Multiplication (Number 1,Plus (Number 2,Number 3))

Не магия, а неоднозначность вывода :(

Подправим грамматику

Прямо в парсере, благо это просто

expression:

```
| term PLUS term { Plus($1, $3) }
| term MINUS term { Minus($1, $3) }
| term { $1 }
```

term:

```
| factor MUL factor { Multiplication($1, $3) }
| factor DIV factor { Division($1, $3) }
| factor { $1 }
```

factor:

```
| INT { Number($1) }
| MINUS expression { Negation($2) }
| LPAR expression RPAR { $2 }
```

Что получилось

Вывод

Plus (Multiplication (Number 1,Number 2),Number 3)

- ▶ Полные исходники
<https://github.com/yurii-litvinov/FsParsersDemo>
- ▶ Некоторое объяснение того, что это было:
<https://github.com/fsprojects/FsLexYacc/blob/master/docs/content/jsonParserExample.md>
- ▶ Ещё пример: https://en.wikibooks.org/wiki/F_Sharp_Programming/Lexing_and_Parsing
- ▶ Внятное описание того, что происходит: D. Syme, A. Granicz, A. Cisternino, Expert F#, 2007
 - ▶ В более свежих изданиях про FsLex/FsYacc нет, такие дела
- ▶ Жалкое подобие документации на домашней странице проекта: <https://fsprojects.github.io/FsLexYacc/>

Выводы

- ▶ FParsec — встроенный DSL для описания грамматик
 - ▶ Легко интегрируется с кодом
 - ▶ Не надо отдельных файлов и странных упражнений с проектными файлами
 - ▶ Нисходящий разбор, боль с грамматикой
 - ▶ И боль с построением AST
- ▶ FsLex/FsYacc — внешний DSL
 - ▶ Странный синтаксис (и, соответственно, не очень с поддержкой в редакторе)
 - ▶ Нужны внешние инструменты
 - ▶ Восходящий разбор, никакой боли с грамматикой
 - ▶ Нам просто повезло, у LR-разбора свои проблемы
 - ▶ Не очень с сообщениями об ошибках