Синтаксический анализ на F#

Часть 2: FParsec vs FsLex/FsYacc

Юрий Литвинов

08.05.2020г

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)
```

F# Interactive

```
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)
```

F# Interactive

```
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)
```

F# Interactive

```
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)
```

F# Interactive

```
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)
```

F# Interactive

```
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)
```

F# Interactive

```
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)
```

F# Interactive

```
Stack overflow.
```

Факторизуем грамматику

```
E ::= PRIMARY E'

E' ::= + PRIMARY E'
    | - PRIMARY E'
    | * PRIMARY E'
    | / PRIMARY E'
    | e

PRIMARY ::= -E
    | (E)
    | NUMBER

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

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

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

F# Interactive

```
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)
```

F# Interactive

```
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 ::= TERM E'

E' ::= + TERM E'
   | - TERM E'
   | e

TERM ::= FACTOR TERM'

TERM' =
   | * FACTOR TERM'
   | / FACTOR TERM'
   | e

FACTOR ::= -E
   | (E)
   | NUMBER

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 '-') >>. !e |>> Negation
        <|> (!(pchar '(') >>. !e .>> !(pchar ')') |>> Brackets)
        <|> number

let term', term'Ref = createParserForwardedToRef()

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

let term = !factor .>>. !term' |>> Term

let e', e'Ref  = createParserForwardedToRef()

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

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

Теперь

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

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

F# Interactive

```
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
```

F# Interactive

```
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