Qualifying Scala project

Interpreter implementation

For this project team should implement interpreter of a functional language, that looks like ML family of languages (Haskell, OCaml, F#). This project is split to steps from the simplest implementation to a more advanced. The goal is to go as far as possible. This project is curated by Alexandr Basov (tg: @basov_ae), feel free to ask questions, all steps could be discussed, if you have own ideas for language implementation, you are welcome.

Motivation and results

By this project we can learn more about functional languages, by implementing one! This is a good start if you want to try language development from scratch. The result language would be not so useful to write real-life programs, but would be a good start for going further. We use lambda-calculus here as a core language, than we improved it to a state where you can easily express some untyped lambda-calculus programs with recursion. From there, types, pattern-matching, literals for common data-structures could be added.

Project roadmap

- Implement lambda-calculus interpreter
  - It should be REPL, that could be compiled and run. Read lambda-term from the console, evaluate it, output the result of evaluation
- Def declaration (C-grade)
  - Used to define functions that later could be used to evaluate terms
- Syntax improvement
  - Make syntax of the language to look like more human-readable
- Support of conditional construction (if-than-else) (B-grade)
- Support recursion via fixed-point combinator (A-grade)

Explanations of each step

From here goes explanation of each step, some theory and approaches how it could be implemented.

1-step lambda calculus

This is tiny language that as powerful as turing machine. Terms could be defined as:

Syntax	**Name**	**Definition**
`x`	Variable	A character or string representing a value
`\x.t`	Abstraction	Function definition (`t` is a term). The variable `x` becomes bound in the expression.
`(t u)`	Application	Applying a function to an argument. `t` and `u` are terms.

And the rules for evaluation are:

Syntax	**Name**	**Definition**
`\x.t{x} -> \y.t{y}`	alpha-conversion	Rename bound occurrence of `x` to `y`
`(\x.t u) -> t{x:=u}`	beta-reduction	Apply left term to other by replacing bound `x` in `t` by `u`

Variables could be free or bounded, this can be defined inductively as:

- The free variables of `x` are just `x`
- The set of free variables of `\x.t` is the set of free variables of `t`, but with `x` removed
- The set of free variables of `(t u)` is the union of the set of free variables of `t` and the set of free variables of `u`
- Other variables are bounded variables

Also to define how term should be evaluated, we introduce Beta-normal forma, that defined as:

- If we can apply beta-reduction to term `t`, `t` is called a redex
- If beta-reduction cannot be applyed, than `t` is in the Beta-normal form

This is small but yet powerful language. This is a good introduction with examples of numbers definition, operations on numbers and recursion

To implement the language, we can follow this steps:

- Write a type that defines grammar of lambda calculus
- Write a function that evaluates expression of lambda-calculus to Beta-normal form
- Write a parser that can take a string and produce lambda-calculus term
- Combine parser with evaluation with REPL

2-step def expressions (C grade)

After implementation of the first step, we can construct lambda terms, but it is not convinient to just write big terms, def adds some sort of abstraction. Now our program would be a set of declarations and a term to evaluate.

```
# this a functions that takes an argument and returns it
def id = \x.x
(id y)
# evaluated by converting (id y) to (\x.x y)
# than we do beta-reduction and receive y as a result
```

3-step syntax improvement

We can improve syntax of our language a bit, to be more simple. Consider def id = \x.x we say that id is a function of one argument that has a body. We can rewrite that function as id x = x, or consider function def add = \x.\y.'some body that adds two numbers', than we can rewrite it as def add x y = ... where ... is body that should be defined.

The next improvement is the (t u) expression, consider application (((t u1) u2) u3), where t for example a function of three arguments: \x.\y.\z.'some body here', brackets here are left-associative, we can improve syntax by saying that: (..(t u1).. un) == t u1 ... un.

We also can improve syntax of lambda-abstraction, consider \x.t firstly we replace . to ->, so our abstraction syntax now is \x -> t. Than we say, that if we have sequence of abstractions: \x.\x1. ... \xn.'body' or with new syntax: \x -> \x1 -> ... \xn, than we can rewrite such expression as: \x x1 ... xn -> 'body'.

4-step conditional construction (B-grade)

In our language we want to have some standard o operators such as if than else. We can define true and false in lambda calculus as (here new syntax from 2 step is used):

```
def true x y = x
def false x y = y
```

So true and false are functions of two arguments. That true returns first argument and false return the second. Now we can define a function that behaves like if-than-else:

```
def ifThanElse bool thanBranch elseBranch = ((bool thanBranch) elseBranch)
# or rewritten as according to improved syntax
def ifThanElse bool thanBranch elseBranch = bool thanBranch elseBranch
```

So the logic is pretty straightforward, we supply true or false with two arguments, true will return the first one, false the second one. This logic could be seen in tuples too:

```
def fst x y = x
def snd x y = y
def getFromTuple getter x y = getter x y
```

Now we can add this ifThanElse logic as embedded construction in the language:

```
def true x y = x
def false x y = y

def smth = 'function that computes something'
def smthElse = 'function that computes something else'

def example x = if x than smth else smthElse

example true
```

Or for example we can define not function as:

```
# bool is either true of false functions
def not bool = if bool than false else true
```

We also can add this syntax for if than else:

```
def not bool =
  if bool
  than false
  else true
```

5-step recursion (A-grade)

To compute something interesting we often want loops, in functional world we do loops with recursion. But here comes the problem, recursion uses name of a function in recursive part, but lambda calculus does not have names for function, we introduced def bindings that acts by substitution original code, it is not actuale 'name'. We can solve this problem by using Y-combinator or more generally fixed point combinator.

This paragraph should explain how it is possible to define recursion. Consider numbers in the Church encoding:

```
def 0 f x = x # looks like false function
def 1 f x = f x # we apply f to x 1 time
def 2 f x = f (f x) # f applied 2 times
...
# defs define succ that will do +1
# this probably takes some time to understand
def plus m n f = m f (n f x)
def succ n f x = f (n f x) # or def succ = plus 1

# than defs define some operations
def mult m n f = m (n f)
def pred n f x = n (\g h -> h (g f)) (\u -> x) (\u -> u)

# because 0 actually is alpha-equivalent to false
def isZero num = if num than false else true
```

Firstly we try to solve problem in the straightforward way:

```
def fuc n =
  if isZero n
  than 1
  else mult n (fuc (pred n))
```

The problem here is that we do the replacement of fuc in the term (some sort of macros), so it will never terminate. But we can introduce additional argument for arbitrary function:

```
# <function> <argument>
# becomes
(\f -> (f argument)) function =>
function argument
```

We can abuse this to do:

```
def fuc1 f n =
  if isZero n
  than 1
  else mult n (f f (pred n)) # we should put two f here to work (try 1 f)

# Than
def fuc = fuc1 fuc1
# (fuc1 fuc1) =>
# \n ->
#   if isZero n
#   than 1
#   else mult n (fuc1 fuc1 (pred n))

# let's try evaluate
fuc 2 =>
mult 2 (if isZero 1 than 1 else mult 1 (fuc1 fuc1 (pred 1))) =>
mult 2 (mult 1 (if isZero 0 than 1 else mult 0 (fuc1 fuc1 (pred 0)))) =>
mult 2 (mult 1 1) => mult 2 1 => 2
```

The problem to solve it that line:

```
...
else mult n (f f (pred n)) # we should put two f here to work (try 1 f)
```

We want a function that takes non-recursive function of the form \f arg -> if cond than terminateCondition else f (recursionStep) For fuc it would be:

```
def fuc1 f n =
  if isZero n
  than 1
  else mult n (f (pred n))
```

We can achieve the goal by introducing:

```
def fix f = (\x -> f (x x)) (\x -> f (x x))
# fix g => g (fix g) => g (g (fix g)) => ...

# lets now try
```

```
def fuc = fix fuc1 =>
def fuc = (\f -> (\x -> f (x x)) (\x -> f (x x))) fuc1 =>
def fuc = (\x -> fuc1 (x x)) (\x -> fuc1 (x x))
def fuc = fuc1 ((\x -> fuc1 (x x)) (\x -> fuc1 (x x))) =>
def fuc n =
  if isZero n
  than 1
  else mult n (fuc1 ((\x -> fuc1 (x x)) (\x -> fuc1 (x x))) (pred n))
```

Now using implemented fix we can introduce rec syntax:

```
def fuc1 f n =
  if isZero n
  than 1
  else mult n (f (pred n))

def fuc = fix fuc1

#Now become
rec fuc n =
  if isZero n
  than 1
  else mult n (fuc (pred n))
```