Language description

My language name is Olol. It's a simple functional language. It provides following features:

- "typedef" allows user to create recursive algebraic types with pattern matching
- operators <,>,==,!= are comparison operators
- simple logical operators: &, ||
- simple arithmetic operators: +,-,*,/,%
- with "def" olol allows user to define different variables
- with "fun" olol allows user to define different functions
- "if" as usual
- it provides creating anonymous functions with "\x -> exp" clause
- its "match with" clause provides a possibility to use pattern matching with different data structures
- it will probably contain simple library with some functionalities (eg. if and List type will probably be made like this)
- all instructions and expressions in programm are separated by ;

Language grammar

I am not using BNFC convention. Therefore I've created rather informal grammar. Here is quick explanation:

- all things in "" are reserved instructions, types, ect..
- (sth)+ means at least one repetition of sth
- (sth)* means at least 0 repetitions of sth
- identifier is a string started with small letter
- Identifier is a string started with a capital letter
- Integer is any proper integer number

```
Programm := ProgElem+

ProgElem := Typedef ";" | Comment | Exp ";"

Exp := "True" | "False" | Integer
   | Exp + Exp | Exp - Exp | Exp * Exp | Exp / Exp | Exp % Exp
   | Exp == Exp | Exp != Exp | Exp > Exp | Exp < Exp
   | Exp || Exp | Exp & Exp
   | "(" Exp ")"
   | identifier    -> variables and functions | Identifier  -> constructors names
   | "if " Exp " then " Exp " else " Exp?
   | Exp Exp
   | "def " identifier "=" Exp | "fun " identifier "=" Exp   -> rules used for defining
variables/functions
   | "\" identifier* "->" Exp     -> rule used for defining anonymous functions
   | "match" Exp "with" ("case" MatchExp "->" Exp)+ "matchend"   -> pattern matching

MatchExp := "True" | "False" | Integer | List?
                    identifier | Identifier (identifier)*

Type := "Int" | "Boolean"
          | Type "->" Type
          | Identifier (Type)*  -> for new types

Typedef := "new type" Identifier identifier* "=" Identifier (Type | identifier)*
                    ("|" Identifier (Type | identifier)*)*

Comment := "/=^.^=" Char* "=^.^=/" -> multiple-line comments
          | "/=^.^=" Char*         -> single-line comments
```

Functionality table

First I plan to create language for 20 points and than probably the one for 25 points.
If I have enough time I would do the one for 30 points too

Na 20 punktów
01 (two types) +
02 (arithmetical operations, comparison) +
03 (if clause) +
04 (functions with multiple arguments, recurrency) +
05 (anonymous functions, higher order functions and partial application) +
06 (run-time errors) +
Lists:
07 (with pattern matching) +
08 (built-in operations) maybe
09 ("syntax sugar") maybe

For 25 points
10 (lists of any type, nested lists and lists of functions) +
11 (simple algebraic types with one level pattern matching) +
12 (static identifier binding) +
13 (static typing) +

For 30 points
  14 (polymorphic and recurrency algebraic types) ? when enough time
  15 (nested pattern matching, complete test) ? when enough time

Bonus
  16 (polymorphic types with types reconstruction algorithm)

Together: 25 (for the ones with +)

Example programm

```
/=^.^= This is single-line comment

/=^.^=
   This is multiple-line comment
=^.^=/


/=^.^= Variables and functions declarations
def k = 2
fun f x y = if x == y then 2 else 4


/=^.^= Arithmetic operators
def a = 2 + 2
de b = 2 - 2 * 3


/=^.^= Comparison operators
2 == 2  /=^.^= True
2 != 2  /=^.^= False


/=^.^= Match usage
func match_func x = match x with
                            case 2 -> True
                            case 3 -> False
                    matchend
```

```
/=^.^= Calling function
def a = match_func 2  /=^.^= True


/=^.^= Anonymous function
(\x -> x * 2) 6


/=^.^= New type usage
new type Tree a = Leaf a | Node a Tree Tree
new type List a = Null | Elem a List
```