Little project on a parser 19/12/2016.

The following context-free grammar generates the programs of a functional toy language that is called Lisp Kit and that, even though simple, can manage higher- order functions, integer values, lists and expressions that contain infix (OPA and OPM) operators as well as prefix operators (OPP). The nonterminals always start with a capital letter, whereas terminals are either punctuation symbols, parentheses or strings of small letters. The terminals **integer** and **var** stand, respectively, for any integer value and any string of alphanumeric characters beginning with a small letter (i.e., any program variable). The symbol ∈ stands for the empty word.

1. Prog::= **let** Bind **in** Exp **end** | **letrec** Bind **in** Exp **end**
2. Bind::= **var =** Exp (**and** Bind | ∈)
3. Exp ::= Prog | **lambda (**Seq_Var**)** Exp | Expa | OPP (Seq_Exp) |
   if Exp **then** Exp **else** Exp
4. Expa::= Term (OPA Expa | ∈)
5. Term ::= Factor (OPM Term | ∈)
6. Factor ::= **var** (Y | ∈) | **integer** | **null** | **(** Expa**)**
7. Y :: = **( )** | **(**Seq_Exp**)**
8. OPA::= **+ | -**
9. OPM::= ***** | /
10. OPP::= **cons** | **head** | **tail** | **eq** | **leq**
11. Seq_Exp::= Exp (**,** Seq_Exp |∈)
12. Seq_Var ::= **var** Seq_var | ∈

Observe that this grammar uses the extended syntax that was introduced in the lectures. An example of this extension is production 4. Expa ::= Term (OPA Expa | ∈) that represents two productions: one having only Term at the right-hand side and one having Term OPA Expa at the right-hand side.
Observe also that the nonterminal Expa has productions similar to those studied during the course, but with a difference: Expa generates expressions that, beside integers, contain also variables and even function invocations. This is done by production 6 and in particular Factor::= **var** (Y | ∈) that, becomes a function invocation when Y is chosen and is just a variable when ∈ is chosen.

A few example programs that can be generated by this grammar follow:
let x=2 **and** y=4 **in** x+y*2 **end**

letrec fact = lambda (n) **if** eq(n,1) **then** 1 **else** n* fact (n-1) **and** x=cons(1, cons(2, null)) **and** f = lambda (l g) **if** eq(l, null) **then null else cons**(g (**head**(l)) , f (g, **tail** (l))) **in** f(x,fact) **end**

The project consists in writing a parser for the given grammar . This parser must compute, for any given input string that contains a program generated by the grammar, a tree representing the structure of the program. The tree is a value of the following data type:

data LKC =VAR String | NUM Int | NULL | ADD LKC LKC |
SUB LKC LKC | MULT LKC LKC | DIV LKC LKC |
EQ LKC LKC | LEQ LKC LKC | H LKC | T LKC | CONS LKC LKC |
IF LKC LKC LKC | LAMBDA [LKC] LKC | CALL LKC [LKC] |
LET LKC [(LKC,LKC)] | LETREC LKC [(LKC, LKC)]
deriving(Show, Eq)

As an example, consider the second program given before. The corresponding LKC value is as follows:
LETREC (CALL (VAR "f") [VAR "x", VAR "fact"])
--next comes the list of pairs representing the three binders. Only the 1st one is given:
[(VAR "fact", LAMBDA (VAR "n") IF (EQ VAR "n" NUM 1) NUM 1 (MULT (VAR "n") (CALL VAR "fact" [(SUB VAR "n" NUM 1)]))),
(,) -- second binder
(,) -- third binder
]
If you look at the first binder, you recognize the left part of the binder (VAR "fact"), and after you find the right part of the binder which is a lambda, thus it starts with LAMBDA, with the list of formal parameters (VAR "n") and then the body that is a conditional and therefore starts with an IF with three parameters describing the condition, the then branch and the else branch. The most complicated part is: (MULT (VAR "n") (CALL VAR "fact" [(SUB VAR "n" NUM 1)])) that represents, n * f (n-1). It may be the case that some extra constructor needs to be added to LKC in order to have the parser work.