Programming Language Description

Julia Bazińska 385134

My language will be a functional one with syntax very similar to Haskell. Features:

- expressions:
  - `int` and `bool` types,
  - standard arithmetic for `int`s: `+ - * / ( )`,
  - standard logic operators for `bool`s: `&&, ||`,
  - standard comparisons for `int`s: `>` $\leq$ $\geq$ `< == !=`,
  - local variable declaration with `let ... in ...` with static binding,
  - `if ... then ... else ...` conditional expression,
- functions:
  - recurrent,
  - with multiple arguments,
  - anonymous,
  - currying,
  - higher order functions,
  - closure,
- lists:
  - of any type (also nested and lists of function),
  - with pattern matching,
  - syntactic sugar for list constants,
- runtime error handling,
- polymorphic and recurrent algebraic data types with pattern matching,
- pattern matching: any level of pattern nesting,
- warnings when defining a partial function for algebraic type with `case ... of ...`,
- static typing with explicit types.

Those are features suggested in the task description for 30 points.
Expected points: 30

Language grammar

```
-- Based on:
-- github.com/BNFC/bnfc/blob/master/examples/haskell-core/Core.cf
-- mimuw.edu.pl/~ben/Zajecia/Mrj2018/Latte/Latte.cf

Program. Prog ::= [Decl];

-- Literals ---------------------------------------------

LitInt.     Lit ::= Integer ;
LitTrue.    Lit ::= "true" ;
LitFalse.   Lit ::= "false" ;
LitList.    Lit ::= "[" [Expr] "]" ;

-- Expressions ------------------------------------------

EVar.     Expr8 ::= Ident ;
ELit.     Expr8 ::= Lit ;
EApp.     Expr7 ::= Expr7 Expr8 ;
Neg.      Expr3 ::= "-" Expr4 ;
Not.      Expr6 ::= "!" Expr7 ;
ECons.    Expr5 ::= Expr6 ":" Expr5 ;
EMul.     Expr4 ::= Expr4 MulOp Expr5 ;
EAdd.     Expr3 ::= Expr3 AddOp Expr4 ;
ERel.     Expr2 ::= Expr2 RelOp Expr3 ;
EAnd.     Expr1 ::= Expr2 "&&" Expr1 ;
EOr.      Expr  ::= Expr1 "||" Expr ;
Lambda.   Expr  ::= "\\" Bind "->" Expr;
Let.      Expr  ::= "let" [Decl] "in" Expr ;
Case.     Expr  ::= "case" Expr2 "of" "{" [EAlt] "}";
If.       Expr  ::= "if" Expr "then" Expr "else" Expr;

coercions Expr 8 ;
separator Expr "," ;

-- Operators --------------------------------------------

Plus.     AddOp ::= "+" ;
Minus.    AddOp ::= "-" ;
Times.    MulOp ::= "*" ;
Div.      MulOp ::= "/" ;
LTH.      RelOp ::= "<" ;
LE.       RelOp ::= "<=" ;
GTH.      RelOp ::= ">" ;
GE.       RelOp ::= ">=" ;
EQU.      RelOp ::= "==" ;
NE.       RelOp ::= "!=" ;
```

```
-- Declarations ------------------------------------------

-- Variable and functions --------------------------------
VDecl. Decl ::= Ident "::" ETy "=" Expr ;
FDecl. Decl ::= Ident [Ident] "::" ETy "=" Expr ;
separator nonempty Ident "" ;

-- Algebraic data types ----------------------------------
DDecl.          Decl ::= "data" Ident "=" [ConstrDef] ;
Constr.     ConstrDef ::= Ident [ConstrArg] ;
ConstrArgDef.ConstrArg ::= Ident ;

separator ConstrArg "" ;
separator nonempty ConstrDef "|" ;

separator Decl ";" ;

-- Types --------------------------------------------------

ETVar.    ETy2  ::= Ident ;
ETList.   ETy1  ::= "List" ETy1 ;
ETApp.    ETy1  ::= ETy2 ETy1 ;
ETBool.   ETy1  ::= "Bool" ;
ETInt.    ETy1  ::= "Int" ;
ETArrow.  ETy   ::= ETy1 "->" ETy ;

coercions ETy 2 ;

-- Comments -----------------------------------------------

comment "//" ;
comment "/*" "*/" ;

-- Binding ------------------------------------------------

BindMulti.    Bind ::= "(" [BindElem] ")";
BindElemT. BindElem ::= Ident "::" ETy;

separator nonempty BindElem "," ;

-- alternatives and pattern matching ----------------------

EAltCase. EAlt ::= ETopPattern "->" Expr;

ETopPatternAt. ETopPattern ::= Ident "@" EPattern ;
ETopPatternNo. ETopPattern ::= EPattern ;
EPatData.         EPattern ::= Ident [EPatConstrArg] ;
EPatLit.          EPattern ::= Lit ;
EPatIdent.        EPattern ::= Ident ;
EPatDefault.      EPattern ::= "_" ;
```

```
EPatHeadIdent.    EPattern ::= Ident ":" EPattern ;
EPatHeadLit.      EPattern ::= Lit ":" EPattern ;

EPatConstrArgDef.EPatConstrArg ::= Ident ;
separator nonempty EPatConstrArg "" ;

separator nonempty EAlt ";" ;
```

Examples

```
// Some basic examples.
f :: Bool = true;
n :: Int = 423;
x :: Int = let f :: Int = 45 in f * 23;
funApply :: Test = fun (1+1) 1;
cons :: List Int = 1:2:3:4:[];
lambda :: Bool -> Int = \ (x :: Int -> Bool -> Int, y::Int) -> x y;
if1 :: Int = if true && false then 123 else let f :: Int = 45 in f + 23;
if2 :: Int = if true && 13 > y then 123 else 11;
someList :: List Int = [1,2+3,3*19,4];
listOfFunctions :: List Int -> Int = [\(r :: Int) -> 3*r, \(r :: Int) ->
    r+r];

// Cases example.
cases :: Int = case x of {
   (y :: Bool) -> 23;
   12:[23] -> 11;
   x:(xs :: List Int) -> 1244;
   a:b:c:d:rest -> 124;
   p@x:xs -> x:p;
   s -> 12
};

cases2 :: Int = case x of {
   Some a -> 1;
   Nothing -> 0
};

// Exponential Fibonacci.
n :: Int = 100;
fibo :: Int -> Int = \(n :: Int) ->
   if n == 1 || n == 2 then 1 else fibo (n-1) + fibo (n-2);
alot :: Int = fibo n;

// Higher order function.
mapInt :: List Int -> (Int -> Int) -> List Int =
```

```
  \(lst :: List Int, fun :: Int -> Int) -> case lst of {
     [] -> [];
     x:xs -> (fun x):(map xs fun)
};

// Double numbers in the list.
doubleTheList :: List Int -> List Int =
  \(lst :: List Int) -> mapInt lst (\(x :: Int) -> x*2);

// Closure
f x :: Int -> Int -> Int =
  let g :: Int -> Int = \(y :: Int) -> y+x in
     g;

// Syntactic sugar for sumOfTwo = \a -> \b -> ...
sumOfTwo a b :: Int -> Int -> Int = a + b;

// Simple Maybe for Ints
data MaybeInt = Nothing | Some Int;

// Parametrized Maybe type
data Maybe a = Nothing | Some a;

// Recursive parametrized type
data Tree a = Empty | Node a Tree Tree;

// Custom parametrized list.
data MyList a = Empty | Nonempty a MyList;
```