HasKDice: Un lenguaje de especificación de tiradas de dados

Informe final
Análisis de Lenguajes de Programación

Realizado por

Zimmermann, Sebastián

Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura

29/07/2020

Índice

1. Panorama general de HasKDice

1.1. Inspiración

Los juegos de mesa, y en particular los juegos de rol, suelen emplear las tiradas de dados para representar el factor aleatorio de las partidas. Sin embargo la forma de utilizarlos varía completamente en cada juego.

En los casos más sencillos, basta con algunas tiradas de dados de igual cantidad de caras, pero en casos más complejos se requiere calcular si hay dobles, quedarse con las k tiradas más altas, y hay juegos que tienen reglas específicas que suelen ser bastante más complejas y que requerirían sumar operaciones nuevas para expresarlas. Si bien existen diversos sistemas online que realizan tiradas de dados, estos suelen usar tiradas simples de dados prefijados y resulta difícil expresar el objetivo deseado sin tener que realizar múltiples operaciones.

El objetivo de este lenguaje es poder especificar de manera concisa y no ambigua cualquier tipo de operación compleja. Esto no solo permite realizar tiradas sino incluso tener un sistema que automatiza todo tipo de tiradas de dados para cualquier juego o sistema que lo necesite.

Este proyecto se inspira en el lenguaje Troll que es un DSL para simular tiradas de dados, y por lo tanto, los objetivos son similares. El lenguaje Troll está armado en ML (específicamente en MoscowML) y si bien es mucho más potente, su código no es abierto. Este lenguaje tiene como un objetivo personal ser un proyecto de Software Libre y en este momento está licenciado con GNU en mi repositorio git.

1.2. En resumen, ¿Para qué hacer un DSL que especifique tiradas de dados?

- Descripciones concisas y no ambiguas que favorecen a la comunicación.
- Tener un lenguaje que automatiza tiradas de dados para cualquier juego o sistema que lo necesite.
- Que dicho sistema sea de Software Libre.

2. Sintaxis y operaciones

A continuación se detalla el funcionamiento de la sintaxis del lenguaje, haciendo énfasis en distintos tipos que posee y su capacidad de expresión. En general, la notación utilizada para expresar tiradas de dados por escrito es estándar, con lo cuál no supondría un problema para un usuario sin conocimientos de programación. A su vez, se emplean palabras reservadas (en inglés) que representan operaciones específicas a realizar sobre un conjunto de tiradas de dados.

2.1. Notación de tiradas

En diversos juegos de rol, se utiliza cierta notación para representar tiradas de dados. Por ejemplo $3d6+4$ representa 3 tiradas de un dado de 6 caras y luego sumarle 4 al resultado. Para representar esto, existe la operación d o D, que justamente realiza tiradas de dados con igual probabilidad para cada tirada. La operación toma dos números (mayores a 0) que se utilizan para representar cuantos dados de cuantas caras serán lanzados.

Por ejemplo, la operación "$5d9$" realizaría 5 tiradas de un dado de 9 caras, es decir, 5 números entre 1 y 9. Existe otra operación que representa dados que tienen caras con 0, esta es Z y se utiliza el caracter z o Z de la misma manera que se usa la d.

Para ambos casos, usar mayúscula o minúscula es indistinto.

Alternativamente, una colección de tiradas de dados puede escribirse utilizando corchetes ('[' , ']') separados por una coma. Por ejemplo [1, 3, 7, 20] son 4 tiradas de dados que "evaluaron" 1, 3, 7, 20 respectivamente, donde por "evaluaron" nos referimos que ya fueron "lanzados" y a partir de este momento, son una colección de números. Deja de importar la cantidad de caras que tenía cada dado, o cuántas tiradas de qué dados fueron inicialmente.

2.2. Operaciones con colecciones de dados

Unas vez creada una colección de tiradas de dados, podemos realizar operaciones sobre esta. Las operaciones sobre colecciones se dividen fundamentalmente en dos tipos, según su tipo de valor de retorno. Existen operaciones que devuelven una nueva colección de tiradas, y otras operaciones que devuelven un valor numérico.

2.2.1. Operaciones que devuelven valores numéricos

La operación *sum C*, devuelve un número con la suma acumulada de todas las tiradas que haya en una colección de tiradas *C*.

Por otro lado, las operaciones *min C* y *max C* retornan el valor mínimo o máximo de una colección de tiradas de dados *C*.

Finalmente la operación *count C* cuenta la cantidad de tiradas que hay en una colección *C*, y retorna ese número.

Por ejemplo, si yo evalúo "*max* 5d6", realiza 5 tiradas de un dado de 6 caras, y se queda con el valor máximo de las 5 tiradas.

2.2.2. Operaciones que devuelven colecciones de tiradas

Existen otras operaciones que devuelven colecciones de tiradas, como *largest n C* y *least n C* que toman los *n* elementos más grandes o chicos de la colección *C*, respectivamente, y los retornan en forma ordenada.

Notar que *largest 1* no es equivalente a *max*, pues *max* retorna el **valor**, mientras que *largest 1* retorna **una colección de una sola tirada de dados**.

Es posible filtrar dentro de una colección de tiradas, basado en operaciones de comparación como <, >, <=, >=, ==, /=, utilizando la operación *filter*. También, para contar la cantidad de resultados, se puede utilizar la operación *count* que cuenta la cantidad de elementos de una colección de tiradas de dados.

Por ejemplo, si se efecutara "*count filter* (<=3) 3*d*6" esto retornaría la cantidad de elementos menores o iguales a 3 en 3 tiradas de un dado de 6 caras.

2.2.3. Operadores entre colecciones

Con esto hemos visto las operaciones básicas con colecciones. Además de estas, existen dos operadores que permiten combinar este tipo de operaciones entre ellas.

El operador @@ concatena dos colecciones de tiradas en una sola colección.

Por ejemplo, si se ejecutara "*sum* (*largest* 5 10*d*9 @@ *least* 3 5*d*6)", inicialmente crearía una colección única con las 5 tiradas más altas de 10 tiradas de un dado de nueve caras, y las 3 tiradas más chicas de 5 tiradas de un dado de seis caras, para luego tomar todos los valores numéricos y sumarlos entre si, dando como resultado un valor numérico. Este es un ejemplo de lo sencillo que resulta especificar una tirada compleja.

Finalmente, existe otro operador # que, de manera similar a el operador *d* utiliza dos argumentos, un valor numérico y una operación que genere colecciones de dados, y la ejecuta esa cantidad de veces concatenando todas en un resultado final. Por ejemplo, si se ejecuta "6 # *largest* 3 10*d*6" esto realizaría 6 veces 10 tiradas de un dado de 6 caras y para cada operación, tomaría los 3 mayores, y me retornaría la colección de las 18 tiradas de un dado de 6 caras resultantes.

2.3. Operadores sobre valores enteros

Como hemos visto, dentro del lenguaje existen varias operaciones sobre colecciones que retornan valores enteros, como *sum*, *count*, *max* o *min*.

Para combinar todas estas operaciones, resulta posible sumar, restar, multiplicar o dividir entre estos valores, utilizando los operadores +, −, ∗, / . Resulta importante aclarar que la división es **división entera**.

También existen los operadores módulo (%), que devuelve el resto de una división y el operador signo (~) que retorna el signo de una operación (1 si es positivo, 0 si es nulo y -1 si es negativo). Además, se puede usar el operador (−) antes de un número para representar valores negativos.

2.4. Operadores sobre valores booleanos

HasKDice soporta valores booleanos. Por empezar tiene definidas las constantes *True* y *False* que son palabras reservadas para representar justamente veradero y falso. Además existen las operaciones menor, mayor, menor igual, mayor igual, igual y distinto, utilizando respectivamente los operadores, <, >, <=, >=, ==, /=, que permiten comparar valores enteros entre si.

Por otro lado, soporta las operaciónes and (&&), or (||) y not (¬), para poder combinar operaciones booleanas entre si.

2.5. Condicionales, variables y operaciones iterativas

HasKDice tiene comandos que permiten realizar diversas operaciones que interactuan entre ellas o con el sistema.

Una de las operaciones más importantes es la operación *let*, que permite asignar valores a funciones dentro de un programa en HasKDice. Al ejecutar un *let* x := 5*d*6 se calculan 5 tiradas de un dado de 6 caras y se la guarda en una nueva variable llamada "x". Esta variable puede volver a utilizarse en todo el programa.

Las variables son tipadas y pueden ser **enteras**, **booleanas**, o **una colección de tiradas de dados**. HasKDice tiene un tipado inferido y chequea los tipos de las operaciones antes de evaluarlas (dando error de tipo en ejecución en caso que exista uno).

Utilizar dos "*lets*" seguidos sobre la misma variable reemplazará el resultado con el último valor.

Una secuencia condicional que permite combinar muchas operaciones es la secuencia *if then else*, que chequea un valor booleano y dependiendo del resultado ejecuta lo que haya después del *then* o del *else*, según sea verdadero o falso respectivamente.

Otra operación sintáctica importante es el operador ";" que separa un comando de otro comando. Esta herramienta resulta importante para componer distintas operaciones. El programa retornará el resultado de la última de todas estas.

Por ejemplo, si efectuo "*let* x := 3*d*6 ; 4*d*7 ; *least* 1 x", el programa lanzará 3 dados de 6 caras, guardará el resultado en x, luego realizará 4 tiradas de un dado de 7 caras (cuyo resultado se perderá) y finalmente tomará el valor más pequeño de x y lo retornará en una colección.

Con estas operaciones resulta mucho más sencillo expresar las reglas de un juego relativamente complejo. Tomemos como ejemplo una tirada de una partida de Backgammon. Se deben tirar dos dados de 6 caras y usar ambos dos valores, pero si saco dobles (ambos dados iguales), puedo usar cada valor dos veces (o sea 4 veces el mismo valor). Esto en HasKDice se escribe de la siguiente forma:

- *let* b := 2*d*6 ; *if* (*max* b) == (*min* b) *then* b@@b *else* b

2.5.1. Operaciones iterativas

Finalmente existen dos operaciones iterativas *repeat* y *accumulate*, ambas dos vienen acompañadas de otra palabra reservada *until* que marca la separación entre los dos argumentos que reciben. Ambas operaciones utilizan solamente variables de tipo **Colección**.

Antes de comenzar, resulta indispensable explicar la convención de qué significa que una expresión evalue **vacía**.

- Si es un valor entero, que evalúe 0.
- Si es un valor booleano, que evalúe True.
- Si es una colección, evalúe vacía (es decir, ninguna tirada).

La primer operación, *repeat*, toma dos argumentos, el primero debe ser una expresión de asignación de una variable que sea de tipo **Colección** y la segunda operación es una operación cualquiera. *repeat* realiza la primer asignación y continúa repitiendo la segunda operación la cantidad de veces que sea necesaria, hasta que esta evalue **vacía**.

Un ejemplo sería " *repeat* x := 2*d*6 *until* (max x) >5 " que repite dos tiradas de dados de 6 caras hasta que alguna de las dos sea un 6. Finalmente retornaría una colección 2*d*6 donde al menos uno de las tiradas sea un 6.

La última operación, *accumulate*, funciona igual que *repeat*, pero retorna una colección de valores donde concatena todas las colecciónes que evalúa el primer término hasta que la segunda operación evalúa vacío.

En otras palabras si realizaramos el mismo ejemplo con un acumulador, " *accumulate* x := 2*d*6 *until* (max x) >5 ", el resultado final sería una colección de pares de tiradas de dados de 6 caras, donde en la última tirada de dos dados de 6 caras, hay al menos un 6 (y en las anteriores no).

3. Organización del código

El lenguaje está modularizado en 6 programas por separado, que van en orden de necesidad.

- El primer programa es **AST.hs**, que es tal cual su nombre describe, el árbol sintáctico abstracto del programa. Ahí están definidas todas las operaciones, los tipos, los errores y fundamentalmente una instancia Show de cada uno, permitiendo que el imprimir en pantalla sea de cómoda lectura.
- El segundo programa es **RandomState.hs**. En este código están definidas las 3 clases de mónadas, una que representa el manejo de generadores de números aleatorios, otra que maneja el estado de variables y una tercera que maneja errores. Luego, se definen 2 mónadas (una para tipos y otra para evaluaciones) y se instancia a cada una en las clases que son necesarias.
- El tercer programa es **TypeEval.hs**. Como su nombre lo indica, es el evaluador de tipos. Básicamente toma cualquier tipo de expresión, chequea recursivamente que todos los tipos sean correctos y devuelve el tipo que retorna la expresión. Esto permite chequear en tiempo real el tipado de las expresiones para evitar errores.
- El cuarto programa es **Eval.hs**. Es el evaluador del lenguaje, se encarga de tomar una expresión cualquiera y evaluar a su resultado final (usualmente en forma recursiva). Está diferenciado en distintos sub-evaluadores según distintos tipos de expresiones. Antes de evaluar, llama al evaluador de tipos definido en el programa anterior.
- El quinto programa es **LexerParser.hs**. Este programa parsea una cadena de caracteres a expresiones definidas en **AST.hs**, permitiendo así que el evaluador pueda evaluarlas.
- El sexto y último programa es **Main.hs**. Este programa se divide en dos partes, una inicial que toma el nombre de un programa que esté en la carpeta "prog" y lo evalúa. Alternativamente, puede llamarse al modo interactivo que permite parsear y evaluar en tiempo real expresiones.

Estos programas se encuentran en el anexo al final del trabajo.

4. Instrucciones de Instalación

Para instalar el software, pueden dirigirse a mi repositorio que explica las instrucciones de instalación.

Es necesario tener Stack y Haskell instalado.

5. Instrucciones de uso: Carga de archivos / Modo interactivo

Una vez instalado, HaskDice cuenta con dos modos. Cargado de archivos y modo interactivo. A modo de ejemplo, voy a utilizar el código del archivo **test.hkd**, presente en el repositorio.

```
let c := 2D6 ;
if max [1,2] < (min [1,2]) then (sum 2D6) else (count 2D6) ;
¬True && False ;
2/3 * 15 % 4 + 3 - 5 + -6 ;
True || True ;
filter (<=4) [1,2,3,4,5,6] ;
accum cd := 2D6 until (count cd > 1 ; count cd > 1) ;
(sum 1D4) # 1D6 ;
least 2 [1,2,3,4] ;
largest 2 [1,2,3,4] ;
filter (== 3) [1,2,3,4] ;
1 + -2 ;
least 2 [1,2,3,4] @@ largest 2 [1,2,3,4] ;
least 6 [1,2,3,4]
```

5.1. Carga de archivo

El cargado de archivos es similar a cualquier lenguaje, simplemente carga el archivo, lo parsea y retorna el resultado final junto con el estado de variables.

```
.> haskdice test.hkd
Result: [1,2,3,4]
Variables: [("c",[1,2])]

Finished sucessfully!
```

5.2. Modo interactivo

El sistema interactivo posee un parser interactivo (junto a un estado de variables que va guardando los resultados de las llamadas de variables) y además permite cargar archivos utilizando el comando ":l" o ":load". También tiene un modo de ayuda usando ":help" o ":?" que explica brevemente el uso de cada comando.

```
.> haskdice -i
  _    _           _  _______   _
 | |  | |         | |/ /  __ \ (_)
 | |__| | __ _ ___| ' /| |  | | _  ___ ___
 |  __  |/ _` / __|  < | |  | || |/ __/ _ \
 | |  | | (_| \__ \ . \| |__| || | (_|  __/
 |_|  |_|\__,_|___/_|\_\_____/ |_|\___\___|


HkD> :l test.hkd
Result: [1,2,3,4]
Variables: [("c",[1,5]),("cd",[6,4])]
```

Vale considerar que al cargar un archivo, actualiza el estado de variables con los resultantes del cargado del mismo, así que de ahí en más podría emplear por ejemplo, el máximo de c

```
HkD> max c
Result: 5
Variables: [("c",[1,5]),("cd",[6,4])]
```

Alternativamente, ante un error de parseo tendríamos un mensaje como este:

```
HkD> 2+
(line 1, column 3):
unexpected end of input
expecting end of "+", "-", "~", identifier, integer, "max", "min", "sum", "count", "least"
, "largest", "filter" or "("


  .   ^
  | <###> /
  |  #*#  /
  :###`###:
  |  ###
  | #####
 _|_.___.__
YOU SHALL NOT
    PARSE
```

6. Anexo: Código

6.1. AST.hs

```haskell
{-# LANGUAGE GADTs #-}

module AST where

import Prelude
import System.Random

---------------------------------------
----- Initial types -------------------
---------------------------------------

-- Collections
type Collection = [Int]

-- Variable identifiers
type Variable = String

-- Possible values that a command can return
data Value = C Collection
           | I Int
           | B Bool
 deriving (Eq, Ord)

instance Show Value where
    show (C c) = show c
    show (I i) = show i
    show (B b) = show b

-- Variable State
type Env = [(Variable,Value)]

-- Typing State of variable (for type eval)
type TypEnv = [(Variable, Type)]

-- Result of an Eval (for instance show purposes)
data EvalResult = ER (Value, Env, StdGen)

instance Show EvalResult where
    show (ER (val,st,stdg)) = "Result: " ++ show val ++ "\nVariables: " ++ show st ++ "\n"


-- Possible errors that the program may have
data Error = TypingError Type Type String
           | VarNotExist String
           | DivByZero String String
           | ModByZero String String
           | PatternMatchError

instance Show Error where
    show (TypingError t1 t2 e) = "\n -- Error: Type error -- \nExpecting type: " ++
        show t1 ++
        " \nbut found type: " ++
        show t2 ++
        " \nin expression " ++
        show e ++ ".\n"
```

```
    show (VarNotExist s) = "\n -- Error: Variable does not exist -- \nThe variable \"" ++ id s ++
    show (DivByZero e1 e2) = "\n -- Error: Division by Zero -- \n" ++
        "A division by zero ocurred when dividing: \n" ++
        show e1 ++
        " \nby: \n" ++
        show e2 ++ ".\n"
    show (ModByZero e1 e2) = "\n -- Error: Mod by Zero -- \n" ++
        "A mod by zero ocurred when calculating mod of\n" ++
        show e1 ++
        " \nby: \n" ++
        show e2 ++ ".\n"
    show (PatternMatchError) = "An error ocurred in the error pattern match, this should never hap

-- The result of an evaluation (type or value)
data Result a = Return a
              | Crash Error

instance Show a => Show (Result a) where
    show (Return r) = show r
    show (Crash e)  = show e


-- Possible Types for the TypeSystem
data Type = TInt
            | TColl
            | TBool
 deriving(Eq, Ord)
        -- ~ | TFun Type Type

instance Show Type where
    show TInt  = "Int"
    show TColl = "Collection"
    show TBool = "Bool"



-- FilOp are different operators that filter can have
data FilOp = Grtth Int
           | Lowth Int
           | GrtEqt Int
           | LowEqt Int
           | Equal Int
           | NEqual Int

instance Show FilOp where
    show (Grtth i)  = "(>" ++ show i ++ ")"
    show (Lowth i)  = "(<" ++ show i ++ ")"
    show (GrtEqt i) = "(>=" ++ show i ++ ")"
    show (LowEqt i) = "(<=" ++ show i ++ ")"
    show (Equal i)  = "(==" ++ show i ++ ")"
    show (NEqual i) = "(/=" ++ show i ++ ")"



-- Expression represents operations with collections.
-- D / Z make a kDn or kZn roll
-- Least / Largt represents lower / greater  K elements of a collection.
-- Filter can filter a collection with a FilOp
```

```haskell
-- Concat concats two collections
-- Const is a constant
-- Max y min calculates the minimum/maximum roll.
-- Sum y count, calculates the sum or count the amount of rolls
-- IsEmpty checks if the value is empty (No dice rolls in a collection, 0 if integer or True if bo
data Expression where
     D       :: Int -> Int -> Expression
     Z       :: Int -> Int -> Expression
     INT     :: Int -> Expression
     COLL    :: Collection -> Expression
     BOOL    :: Bool -> Expression
     Var     :: Variable -> Expression
     Least   :: Int -> Expression -> Expression
     Largt   :: Int -> Expression -> Expression
     Filter  :: FilOp -> Expression -> Expression
     Concat  :: Expression -> Expression -> Expression
     MAX     :: Expression -> Expression
     MIN     :: Expression -> Expression
     SUM     :: Expression -> Expression
     COUNT   :: Expression -> Expression
     ADD     :: Expression -> Expression -> Expression
     MINUS   :: Expression -> Expression -> Expression
     TIMES   :: Expression -> Expression -> Expression
     DIV     :: Expression -> Expression -> Expression
     MOD     :: Expression -> Expression -> Expression
     UMINUS  :: Expression -> Expression
     SGN     :: Expression -> Expression
     INDEP   :: Expression -> Expression -> Expression
     Eq      :: Expression -> Expression -> Expression
     NEq     :: Expression -> Expression -> Expression
     Lt      :: Expression -> Expression -> Expression
     Gt      :: Expression -> Expression -> Expression
     GEt     :: Expression -> Expression -> Expression
     LEt     :: Expression -> Expression -> Expression
     AND     :: Expression -> Expression -> Expression
     OR      :: Expression -> Expression -> Expression
     NOT     :: Expression -> Expression
     IsEmpty :: Value -> Expression

instance Show Expression where
    show (D k n)        = show k ++ "D" ++ show n
    show (Z k n)        = show k ++ "Z" ++ show n
    show (INT i)        = show i
    show (COLL c)       = show c
    show (Var v)        = id v
    show (Least i c)    = "least " ++ show i ++ " " ++ show c
    show (Largt i c)    = "largest " ++ show i ++ " " ++ show c
    show (Filter fop c) = "filter " ++ show fop ++ " " ++ show c
    show (Concat c1 c2) = show c1 ++ " @@ " ++ show c2
    show (MAX e)        = "max " ++ show e
    show (MIN e)        = "min " ++ show e
    show (SUM e)        = "sum " ++ show e
    show (COUNT e)      = "count " ++ show e
    show (ADD e1 e2)    = show e1 ++ " + " ++ show e2
    show (MINUS e1 e2)  = show e1 ++ " - " ++ show e2
    show (TIMES e1 e2)  = show e1 ++ " * " ++ show e2
    show (DIV e1 e2)    = show e1 ++ " / " ++ show e2
    show (MOD e1 e2)    = show e1 ++ " % " ++ show e2
```

```
    show (UMINUS e1)     = " -" ++ show e1
    show (SGN e1)        = "sgn " ++ show e1
    show (INDEP n c)     = show n ++ " # " ++ show c
    show (BOOL b)        = show b
    show (Eq e1 e2)      = show e1 ++ " == " ++ show e2
    show (NEq e1 e2)     = show e1 ++ " /= " ++ show e2
    show (Lt e1 e2)      = show e1 ++ " < " ++ show e2
    show (Gt e1 e2)      = show e1 ++ " > " ++ show e2
    show (GEt e1 e2)     = show e1 ++ " >= " ++ show e2
    show (LEt e1 e2)     = show e1 ++ " <= " ++ show e2
    show (AND e1 e2)     = show e1 ++ " && " ++ show e2
    show (OR e1 e2)      = show e1 ++ " || " ++ show e2
    show (NOT e)         = "¬" ++ show e
    show (IsEmpty v)     = "is empty " ++ show v

-- Commands
-- Expr is any expression
-- Let saves the value of an expression in a variable (if it doesn't exists, creates one).
-- Seq is a sequence of commands
-- If then else is.. that.
-- Accum accummulates the results of the first command until the second is empty
-- Repunt repeats the results of the first command until the second is not empty
data Command where
    Expr       :: Expression -> Command
    Let        :: Variable -> Expression -> Command
    Seq        :: Command -> Command -> Command
    IfThenElse :: Expression -> Command -> Command -> Command
    ACCUM      :: Command -> Command -> Command
    REPUNT     :: Command -> Command -> Command

instance Show Command where
    show (Expr e)             = show e
    show (Let v c)            = id v ++ " := " ++ show c
    show (Seq c1 c2)          = show c1 ++ ";\n" ++ show c2
    show (IfThenElse b c1 c2) = "if " ++ show b ++ " then " ++ show c1 ++ " else " ++ show c2
    show (ACCUM c1 c2)        = "accumulate " ++ show c1 ++ " until " ++ show c2
    show (REPUNT c1 c2)       = "repeat " ++ show c1 ++ " until " ++ show c2
```

6.2. RandomState.hs

```
{-# LANGUAGE MultiParamTypeClasses #-}

module RandomState where

import AST
import Control.Applicative (Applicative(..))
import Control.Monad       (liftM, ap)
import Control.Monad.Fail
import Data.List
import System.IO
import System.Random
import Prelude



---------------------------------------
----- Random State Monad --------------
---------------------------------------
```

```haskell
newtype RandomState a = RS { runRS :: Env -> StdGen -> Result (a, Env, StdGen) }

instance Functor RandomState where
    fmap = liftM

instance Applicative RandomState where
    pure   = return
    (<*>)  = ap

instance Monad RandomState where
    return d = RS (\st sg -> Return (d, st, sg))
    m >>= f  = RS (\st sg -> case (runRS m st sg) of
                        Crash e              -> Crash e
                        Return (d, st', sg') -> runRS (f d) st' sg')


---------------------------------------
----- Type State Monad ----------------
---------------------------------------

newtype TypeState a = TS { runTS :: TypEnv -> Result (a,TypEnv) }

instance Functor TypeState where
    fmap = liftM

instance Applicative TypeState where
    pure   = return
    (<*>)  = ap

instance Monad TypeState where
    return t = TS (\st -> Return (t, st))
    m >>= f  = TS (\st -> case (runTS m st) of
                        Crash e         -> Crash e
                        Return (t, st') -> runTS (f t) st')


---------------------------------------
----- Monad Classes and Instances -----
---------------------------------------



-- Class that represents monads with an enviroment of variables
class Monad m => MonadState m a where
    -- Search a variable value
    lookfor :: Variable -> m a
    -- Updates a variable value
    update :: Variable -> a -> m ()

instance MonadState RandomState Value where
    lookfor v = RS (\st sg -> case (lookfor' v st sg) of
                        Crash (VarNotExist v) -> Crash (VarNotExist v)
                        Return j              -> Return (j, st, sg))
            where lookfor' v []          sg = Crash (VarNotExist v)
                  lookfor' v ((u, j):ss) sg | v == u = Return j
                                            | v /= u = lookfor' v ss sg
    update v val = RS (\st sg -> Return ((), update' v val st, sg))
```

```
                  where update' v i []          = [(v, i)]
                        update' v i ((u, _):ss) | v == u = (v, i):ss
                        update' v i ((u, j):ss) | v /= u = (u, j):(update' v i ss)

instance MonadState TypeState Type where
    lookfor v = TS (\st -> case (lookfor' v st) of
                        Crash (VarNotExist v) -> Crash (VarNotExist v)
                        Return j              -> Return (j, st))
            where lookfor' v []          = Crash (VarNotExist v)
                  lookfor' v ((u, j):ss) | v == u = Return j
                                         | v /= u = lookfor' v ss
    update v val = TS (\st -> Return ((), update' v val st))
                  where update' v i []          = [(v, i)]
                        update' v i ((u, _):ss) | v == u = (v, i):ss
                        update' v i ((u, j):ss) | v /= u = (u, j):(update' v i ss)



-- Class that represent monads that has possible errors
class Monad m => MonadError m where
    throwTypingError  :: Type -> Type -> String -> m a -- The Value thing may change
    throwVarNotExist  :: String -> m a
    throwDivByZero    :: String -> String -> m a
    throwModByZero    :: String -> String -> m a
    throwPatMatch     :: m a

instance MonadError RandomState where
    throwTypingError t1 t2 s = RS (\st sg -> Crash $ TypingError t1 t2 s)
    throwVarNotExist v       = RS (\st sg -> Crash $ VarNotExist v)
    throwDivByZero e1 e2     = RS (\st sg -> Crash $ DivByZero e1 e2)
    throwModByZero e1 e2     = RS (\st sg -> Crash $ ModByZero e1 e2)
    throwPatMatch            = RS (\st sg -> Crash PatternMatchError)

instance MonadError TypeState where
    throwTypingError t1 t2 s = TS (\st -> Crash $ TypingError t1 t2 s)
    throwVarNotExist v       = TS (\st -> Crash $ VarNotExist v)
    throwDivByZero s1 s2     = TS (\st -> Crash $ DivByZero s1 s2)
    throwModByZero s1 s2     = TS (\st -> Crash $ ModByZero s1 s2)
    throwPatMatch            = TS (\st -> Crash PatternMatchError)

-- Class that represent monads that works with randomness
class Monad m => MonadRandom m where
    -- getStd generates a split from the generator to maintain randomness.
    getStd :: m StdGen

instance MonadRandom RandomState where
    getStd = RS (\st sg -> let (sg1,sg2) = split sg in
                           Return (sg1,st,sg2))

instance MonadFail RandomState where
    fail _ = throwPatMatch

instance MonadFail TypeState where
    fail _ = throwPatMatch
```

6.3. TypeEval.hs

```haskell
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleContexts #-}


module TypeEval where

import AST
import RandomState
import System.Random

-- |initState of type enviroment
initStateType ::TypEnv
initStateType = []

-- Takes a variable and transforms and returns the type (with the name of the variable)
varToType :: (Variable,Value) -> (Variable,Type)
varToType (str,(C _)) = (str,TColl)
varToType (str,(I _)) = (str,TInt)
varToType (str,(B _)) = (str,TBool)

-- Mapfunction to map an enviroment of variables and returns the type.
mapingTypeEnv :: Env -> TypEnv
mapingTypeEnv state = map varToType state


-- Checks if two types are equal
equalType :: (MonadState m Type, MonadError m) => Type -> Type -> String -> m ()
equalType t1 t2 e = if (t1 == t2) then return ()
                                  else throwTypingError t1 t2 e

-- Checks if in a binary operator that expects types t1/t2, the actual types are correct
typingBinaryOp :: (MonadState m Type, MonadError m) => Type -> Expression -> Type -> Expression ->
typingBinaryOp t1 e1 t2 e2 optype = do
    actt1 <- typingExp e1
    equalType t1 actt1 (show e1)
    actt2 <- typingExp e2
    equalType t2 actt2 (show e2)
    return optype

-- Checks if a unaryOp that expects type t1 has the actual type t1.
typingUnaryOp :: (MonadState m Type, MonadError m) => Type -> Expression -> Type -> m Type
typingUnaryOp typ exp untype = do
    acttyp <- typingExp exp
    equalType typ acttyp (show exp)
    return untype

-- Checks if a unaryOp that expects type t1 has the actual type t1.
typingUnaryCommand :: (MonadState m Type, MonadError m) => Type -> Command -> Type -> m Type
typingUnaryCommand typ com untype = do
    acttyp <- typingCommand com
    equalType typ acttyp (show com)
    return untype


-- Checks for a Value what type it is.
typingValue :: (MonadState m Type, MonadError m) => Value -> m Type
typingValue (C _) = return TColl
```

```
typingValue (I _) = return TInt
typingValue (B _) = return TBool

-- Checks if type of arguments of an expression are correct, and returns the type of the expressio
typingExp :: (MonadState m Type, MonadError m) => Expression -> m Type
typingExp (D _ _)   = return TColl
typingExp (Z _ _)   = return TColl
typingExp (INT _)   = return TInt
typingExp (COLL _)  = return TColl
typingExp (Least _ c)  = typingUnaryOp TColl c TColl
typingExp (Largt _ c)  = typingUnaryOp TColl c TColl
typingExp (Filter _ c) = typingUnaryOp TColl c TColl
typingExp (Concat c1 c2) = typingBinaryOp TColl c1 TColl c2 TColl
typingExp (MAX c) = typingUnaryOp TColl c TInt
typingExp (MIN c) = typingUnaryOp TColl c TInt
typingExp (SUM c) = typingUnaryOp TColl c TInt
typingExp (COUNT c) = typingUnaryOp TColl c TInt
typingExp (ADD x y) = typingBinaryOp TInt x TInt y TInt
typingExp (MINUS x y) = typingBinaryOp TInt x TInt y TInt
typingExp (TIMES x y) = typingBinaryOp TInt x TInt y TInt
typingExp (DIV x y) = typingBinaryOp TInt x TInt y TInt
typingExp (MOD x y) = typingBinaryOp TInt x TInt y TInt
typingExp (UMINUS x) = typingUnaryOp TInt x TInt
typingExp (SGN x) = typingUnaryOp TInt x TInt
typingExp (INDEP n c) = typingBinaryOp TInt n TColl c TColl
typingExp (BOOL _) = return TBool
typingExp (Eq x y) = typingBinaryOp TInt x TInt y TBool
typingExp (NEq x y) = typingBinaryOp TInt x TInt y TBool
typingExp (Lt x y) = typingBinaryOp TInt x TInt y TBool
typingExp (Gt x y) = typingBinaryOp TInt x TInt y TBool
typingExp (GEt x y) = typingBinaryOp TInt x TInt y TBool
typingExp (LEt x y) = typingBinaryOp TInt x TInt y TBool
typingExp (AND p q) = typingBinaryOp TBool p TBool q TBool
typingExp (OR p q) = typingBinaryOp TBool p TBool q TBool
typingExp (NOT p) = typingUnaryOp TBool p TBool
typingExp (IsEmpty v) = do
    typingValue v
    return TBool
typingExp (Var var) = do
    t <- lookfor var
    return t

-- Takes a command, checks the type of the result and if some command has typing errors.
typingCommand :: (MonadState m Type, MonadError m) => Command -> m Type
typingCommand (Expr e) = typingExp e
typingCommand (Let v e) = do
        t <- typingExp e
        update v t
        return t
typingCommand (Seq c1 c2) = do
        tc1 <- typingCommand c1
        tc2 <- typingCommand c2
        return tc2
typingCommand (IfThenElse b c1 c2) = do
        tb <- typingUnaryOp TBool b TBool
        tc1 <- typingCommand c1
        tc2 <- typingCommand c2
        return tc2
```

```
typingCommand (ACCUM c1 c2) = do
        tc1 <- typingUnaryCommand TColl c1 TColl
        tc2 <- typingCommand c2
        return tc1
typingCommand (REPUNT c1 c2) = do
        tc1 <- typingUnaryCommand TColl c1 TColl
        tc2 <- typingCommand c2
        return tc1

-------------------------------------
--- Typing Evaluator ----------------
-------------------------------------


-- Check if a program is typed correctly
evalType :: Command -> Env -> Result Type
evalType exp st = case (runTS (do { res <- typingCommand exp; return res}) (mapingTypeEnv st)) of
              Crash e      -> Crash e
              Return (t,_) -> Return t
```

6.4. Eval.hs

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleContexts #-}

module Eval where

import AST
import TypeEval
import RandomState
import Control.Applicative (Applicative(..))
import Control.Monad       (liftM, ap)
import Control.Monad.Fail
import Data.List
import System.IO
import System.Random
import Prelude


---------------------------------------
----- Initial State -------------------
---------------------------------------

-- Initial State (Null)
initState :: Env
initState = []

---------------------------------------
----- Evaluator -----------------------
---------------------------------------


-- eval is the first function to be called, to eval the result that the main call upon.
eval :: StdGen -> Env -> Command -> Result EvalResult
eval gen state exp = case evalType exp state of
                          Crash e     -> Crash e
                          Return _    -> case (runRS (do {res <- evalCommand exp; return res}) sta
                                                 Crash e   -> Crash e
```

```
                                                Return eres -> Return (ER eres)


-- evalFiltOp eval an operator to the filter, and returns it in a function.
evalFiltOp :: FilOp -> (Int -> Bool)
evalFiltOp (Grtth n)  = (>n)
evalFiltOp (Lowth n)  = (<n)
evalFiltOp (GrtEqt n) = (>=n)
evalFiltOp (LowEqt n) = (<=n)
evalFiltOp (Equal n)  = (==n)
evalFiltOp (NEqual n) = (/=n)

-- evalExp takes any kind of expression and evaluates that expression, generating a Value.
evalExp :: (MonadState m Value, MonadError m, MonadRandom m, MonadFail m) => Expression -> m Value
evalExp (D k n) = do
    g' <- getStd
    let rolls = take k (randomRs (1 :: Int,n) g')
    return (C rolls)
evalExp (Z k n) = do
    g' <- getStd
    let rolls = take k (randomRs (0 :: Int,n) g')
    return (C rolls)
evalExp (INT n) = return (I n)
evalExp (COLL c) = return (C c)
evalExp (Var v) = lookfor v
evalExp (Least k ce) = do
           (C rolls) <- evalExp ce
           let rolls' = take k (sort rolls)
           return (C rolls')
evalExp (Largt k ce) = do
           (C rolls) <- evalExp ce
           let rolls' = take k (sortBy (flip compare) rolls)
           return (C rolls')
evalExp (Filter fop ce) = do
           (C rolls) <- evalExp ce
           let funcfilt = evalFiltOp fop
           return $ C (filter funcfilt rolls)
evalExp (Concat exp1 exp2) = do
           (C c1) <- evalExp exp1
           (C c2) <- evalExp exp2
           return $ C (c1 ++ c2)
evalExp (MAX ce) = do
            (C rolls) <- evalExp ce
            return (I $ foldr max 0 rolls)
evalExp (MIN ce) = do
            (C rolls) <- evalExp ce
            return (I $ foldr min (maxBound::Int) rolls)
evalExp (SUM ce) = do
            (C rolls) <- evalExp ce
            return (I $ sum rolls)
evalExp (COUNT ce) = do
            (C rolls) <- evalExp ce
            return (I $ length rolls)
evalExp (ADD x y) = do
            (I x') <- evalExp x
            (I y') <- evalExp y
            return $ I (x' + y')
evalExp (MINUS x y) = do
```

```
            (I x') <- evalExp x
            (I y') <- evalExp y
            return $ I (x' - y')
evalExp (TIMES x y) = do
            (I x') <- evalExp x
            (I y') <- evalExp y
            return $ I (x' * y')
evalExp (DIV x y) = do
            (I x') <- evalExp x
            (I y') <- evalExp y
            if (y' == 0) then throwDivByZero (show x) (show y)
                         else return $ I (x' `div` y')
evalExp (MOD x y) = do
            (I x') <- evalExp x
            (I y') <- evalExp y
            if (y' == 0) then throwModByZero (show x) (show y)
                         else return $ I (x' `mod` y')
evalExp (UMINUS x) = do
            (I n) <- evalExp x
            return $ I (n*(-1))
evalExp (SGN x) = do
            (I n) <- evalExp x
            return $ I (signum n)
evalExp (INDEP n c) = do
            (I cant) <- evalExp n
            (C coll) <- evalExp c
            if (cant > 1) then do {(C coll2) <- evalExp (INDEP (INT (cant-1)) c) ; return $ C (col
                          else return (C coll)
evalExp (BOOL b) = return (B b)
evalExp (IsEmpty v) = do
        case v of
            (C coll) -> return $ B (coll == [])
            (B bool) -> return $ B (bool == True)
            (I numb) -> return $ B (numb == 0)
evalExp (Eq e1 e2) = do
        (I n) <- evalExp e1
        (I m) <- evalExp e2
        return $ B (n == m)
evalExp (NEq e1 e2) = do
        (I n) <- evalExp e1
        (I m) <- evalExp e2
        return $ B (n /= m)
evalExp (Lt e1 e2) = do
        (I n) <- evalExp e1
        (I m) <- evalExp e2
        return $ B (n < m)
evalExp (Gt e1 e2) = do
        (I n) <- evalExp e1
        (I m) <- evalExp e2
        return $ B (n > m)
evalExp (LEt e1 e2) = do
        (I n) <- evalExp e1
        (I m) <- evalExp e2
        return $ B (n <= m)
evalExp (GEt e1 e2) = do
        (I n) <- evalExp e1
        (I m) <- evalExp e2
        return $ B (n <= m)
```

```
evalExp (AND b1 b2) = do
        (B x) <- evalExp b1
        (B y) <- evalExp b2
        return $ B (x && y)
evalExp (OR b1 b2) = do
        (B x) <- evalExp b1
        (B y) <- evalExp b2
        return $ B (x || y)
evalExp (NOT b) = do
        (B x) <- evalExp b
        return $ B (not x)


-- evalCommand takes a command and evaluates both the value and the changes in the state.
evalCommand :: (MonadState m Value, MonadError m, MonadRandom m, MonadFail m) => Command -> m Valu
evalCommand (Expr exp) = do
            e <- evalExp exp
            return e
evalCommand (Seq c1 c2) = do
            n <- evalCommand c1
            m <- evalCommand c2
            return m
evalCommand (IfThenElse b c1 c2) = do
            (B bool) <- evalExp b
            if (bool) then (do {res <- evalCommand c1; return res})
                      else (do {res <- evalCommand c2; return res})
evalCommand (Let name e) = do
            res <- evalExp e
            update name res
            return res
evalCommand (REPUNT (Let v c) exp) = do
            (C e1) <- evalCommand (Let v c)
            e <- evalCommand exp;
            res <- evalCommand (IfThenElse (IsEmpty e) (Expr (COLL e1)) (REPUNT (Let v c) exp));
            return res
evalCommand (ACCUM (Let v c) exp) = do
            (C head) <- evalCommand (Let v c)
            e <- evalCommand exp;
            (C tail) <- evalCommand (IfThenElse (IsEmpty e) (Expr (COLL [])) (ACCUM (Let v c) exp)
            (C list) <- evalExp (Concat (COLL head) (COLL tail))
            return (C list)


-- Definí un estandar de IsEmpty : Bool = False, Coll = [], Int = 0


testEval = do
    g <- newStdGen
    let res = eval g [] (Expr (Filter (GrtEqt 3) (Largt 3 (D 5 8)) ))
    let typeres = evalType (Expr (Filter (GrtEqt 3) (Largt 3 (D 5 8)) ))

    let test1 = eval g [] (Let "x" (COLL [1,2,3]))
    let typetest1 = evalType (Let "x" (COLL [1,2,3]))

    let test2 = eval g [] (IfThenElse (IsEmpty (C [])) (Expr (D 1 6)) (Expr (Z 1 8)))
    let typetest2 = evalType (IfThenElse (IsEmpty (C [])) (Expr (D 1 6)) (Expr (Z 1 8)))

    let test3 = eval g [] (Seq (Let "b" (D 2 6)) (IfThenElse (Eq (MAX (Var "b")) (MIN (Var "b")))
```

```
    let typetest3 = evalType (Seq (Let "b" (D 2 6)) (IfThenElse (Eq (MAX (Var "b")) (MIN (Var "b")

    let test4 = eval g [] (Seq (Let "b" (D 2 6)) (IfThenElse (Eq (MAX (INT 2)) (MIN (Var "b"))) (E

    let test5 = eval g [] (ACCUM (Let "b" (D 2 6)) (Expr (Eq (MAX (INT 2)) (MIN (Var "b"))))  )
    let test6 = eval g [] (Seq (Let "a" (D 1 6)) (Seq (Let "b" (D 1 6)) (Seq (Let "c" (D 1 6)) (Le
    let testdivZero = eval g [] (Expr (DIV (SUM (D 1 6)) (INT 0)) )
    let testmodZero = eval g [](Expr (MOD (SUM (D 1 6)) (INT 0)) )
    let testnodVar  = eval g [](Expr (ADD (Var "b") (INT 0)) )
    let testgreater = eval g [](ACCUM (Let "a" (D 1 6)) (Expr (Gt (MAX (Var "a")) (INT 3))))
    let test8 = eval g [](REPUNT (Let "a" (D 1 6)) (Expr (Eq (MAX (Var "a")) (INT 6))))

    print "test greater"
    print testgreater
    print "test 8: "
    print test8

    print testdivZero
    print testmodZero
    print testnodVar
```

6.5. LexerParser.hs

```
module LexerParser where

import Text.ParserCombinators.Parsec
import Text.Parsec.Token
import Text.Parsec.Language (emptyDef)
import Text.Parsec.Error
import AST
import Data.Char
import Data.Maybe

-----------------------------------------------
--- Token analizer and Total Parser ------------
-----------------------------------------------

type ParseResult = Either ParseError Command

-- ~ instance Show ParseResult where
    -- ~ show (Left e) = "YOU SHALL NOT PARSE \n" ++ show e -- Yes, this was the only reason.
    -- ~ show (Right c) = show c


totParser :: Parser a -> Parser a
totParser p = do
                whiteSpace lis
                t <- p
                eof
                return t


-- Token analizer
lis :: TokenParser u
lis = makeTokenParser (emptyDef   { commentStart  = "{-"
                                  , commentEnd    = "-}"
                                  , commentLine   = "--"
                                  , opLetter      = char '='
```

```
                                  , reservedNames = ["True","False","if","then", "least", "largest",
                                                     "max", "min", "sum", "count", "else", "while",
                                  , reservedOpNames = ["-", "+","*", "/", "~", "#", "%", "@@",
                                                       "==", "/=", "&&", "||", "<", ">", "<=", ">=",
                                  , caseSensitive = True
                                  })

-----------------------------------
--- Common Parsers ----------------
-----------------------------------
parenParse  :: Parser p -> Parser p
parenParse p = do symbol lis "("
                  x <- p
                  symbol lis ")"
                  return x

intParse = do
    n <- integer lis
    return (fromInteger n :: Int)


varParse :: Parser Expression
varParse = do name <- identifier lis
              return (Var name)



-------------------------------------
--- IntExps Parser ------------------
-------------------------------------


opParseTerm = do reservedOp lis "+"
                 return (ADD)
            <|> do reservedOp lis "-"
                   return (MINUS)

opParseFactor = do  { reservedOp lis "*" ; return (TIMES) }
              <|> do { reservedOp lis "/" ; return (DIV) }
              <|> do { reservedOp lis "%" ; return (MOD) }

unaryParse = do { reservedOp lis "-"
                ; b <- factorParse
                ; return (UMINUS b) }
             <|> do { reservedOp lis "~"
                ; b <- factorParse
                ; return (SGN b) }

intexp :: Parser Expression
intexp  = do
    f <- integer lis
    return $ INT (fromInteger f :: Int)

filOpParse :: Parser FilOp
filOpParse = try (do symbol lis "("
                     symbol lis ">"
                     n <- intParse
                     symbol lis ")"
```

```
                    return (Grtth n) )
                <|> try (do symbol lis "("
                            symbol lis "<"
                            n <- intParse
                            symbol lis ")"
                            return (Lowth n) )
                <|> try (do symbol lis "("
                            symbol lis "<"
                            symbol lis "="
                            n <- intParse
                            symbol lis ")"
                            return (LowEqt n) )
                <|> try (do symbol lis "("
                            symbol lis ">"
                            symbol lis "="
                            n <- intParse
                            symbol lis ")"
                            return (GrtEqt n) )
                <|> try (do symbol lis "("
                            symbol lis "="
                            symbol lis "="
                            n <- intParse
                            symbol lis ")"
                            return (Equal n) )
                <|> try (do symbol lis "("
                            symbol lis "/"
                            symbol lis "="
                            n <- intParse
                            symbol lis ")"
                            return (NEqual n) )

opCollParse :: Parser Expression
opCollParse = do reserved lis "max"
                 e <- collTermParse
                 return (MAX e)
              <|> do reserved lis "min"
                     e <- collTermParse
                     return (MIN e)
              <|> do reserved lis "sum"
                     e <- collTermParse
                     return (SUM e)
              <|> do reserved lis "count"
                     e <- collTermParse
                     return (COUNT e)
              <|> do reserved lis "least"
                     n <- intParse
                     c <- collTermParse
                     return (Least n c)
              <|> do reserved lis "largest"
                     n <- intParse
                     c <- collTermParse
                     return (Largt n c)
              <|> do reserved lis "filter"
                     fop <- filOpParse
                     c <- collTermParse
                     return (Filter fop c)
```

```
factorParse :: Parser Expression
factorParse  = unaryParse
             <|> try varParse
             <|> try intexp
             <|> try opCollParse
             <|> try (parenParse intExprParse)

termParse :: Parser Expression
termParse  = chainl1 factorParse opParseFactor

intExprParse :: Parser Expression
intExprParse  = chainl1 termParse opParseTerm

-------------------------------------
--- CollExps Parser -----------------
-------------------------------------

diceBinopParse = do { reservedOp lis "D" ; return (D) }
                 <|> do { reservedOp lis "d" ; return (D) }
                 <|> do { reservedOp lis "Z" ; return (Z) }
                 <|> do { reservedOp lis "z" ; return (Z) }

diceParse :: Parser Expression
diceParse = do k <- intParse
               op <- diceBinopParse
               n <- intParse
               return (op k n)
            -- ~ <|> do k <- intParse
                    -- ~ reservedOp lis "Z"
                    -- ~ n <- intParse
                    -- ~ return (Z k n)

collParse :: Parser Expression
collParse = do
    symbol lis "["
    c <- sepBy1 intParse (symbol lis ",")
    symbol lis "]"
    return (COLL c)

-- ~ concatParse =  do reservedOp lis "@@"
                  -- ~ return (Concat)

-- ~ indepParse = do { try (do n <- intExprParse
                           -- ~ reservedOp lis "#"
                           -- ~ c <- collExprParse
                           -- ~ return (INDEP n c)) }

binopCollParse = do reservedOp lis "@@"
                    return (Concat)
                 <|> do reservedOp lis "#"
                        return (INDEP)


collTermParse :: Parser Expression
collTermParse = try diceParse
                <|> try collParse
                <|> try intExprParse
                <|> try (parenParse collExprParse)
```

```
collExprParse :: Parser Expression
collExprParse =  chainl1 collTermParse binopCollParse



-----------------------------------
--- BoolExps Parse ----------------
-----------------------------------

boolPrimitive :: Parser Expression
boolPrimitive  = do { f <- reserved lis "True"; return (BOOL True) }
               <|> do { f <- reserved lis "False"; return (BOOL False) }


compOpParse = do  { reservedOp lis "==" ; return (Eq) }
              <|> do { reservedOp lis "/=" ; return (NEq) }
              <|> do { reservedOp lis "<" ; return (Lt) }
              <|> do { reservedOp lis ">" ; return (Gt) }
              <|> do { reservedOp lis "<=" ; return (LEt) }
              <|> do { reservedOp lis ">=" ; return (GEt) }

boolOpParse = do  { reservedOp lis "&&" ; return (AND) }
              <|> do { reservedOp lis "||" ; return (OR) }

negationParse :: Parser Expression
negationParse  = do { reservedOp lis "¬"
                    ; b <- bTermParse
                    ; return (NOT b) }

comparisonParse :: Parser Expression
comparisonParse  = do { x <- collExprParse
                      ; f <- compOpParse
                      ; y <- collExprParse
                      ; return (f x y)}


boolParse :: Parser Expression
boolParse  = chainl1 bTermParse boolOpParse

bTermParse :: Parser Expression
bTermParse  = negationParse
            <|> try boolPrimitive
            <|> try comparisonParse
            <|> try collExprParse
            <|> try (parenParse boolParse)
            -- ~ <|> try varParse

-------------------------------------
--- General Expressions Parser ------
-------------------------------------

expParse :: Parser Expression
expParse = try boolParse
           -- ~ <|> try collExprParse
           -- ~ <|> try varParse
```

```
-------------------------------------
--- Command Parser ------------------
-------------------------------------

dacParse :: Parser (Command -> Command -> Command)
dacParse  = do  { reservedOp lis ";" ; return (Seq) }

letParse :: Parser Command
letParse  = do reserved lis "let"
               name <- identifier lis
               reservedOp lis ":="
               value <- expParse
               return (Let name value)

exprParse = do e <- expParse
               return (Expr e)

ifElseParse :: Parser Command
ifElseParse  = do reserved lis "if"
                  cond <- boolParse
                  reserved lis "then"
                  thencmd <- commParse
                  reserved lis "else"
                  elsecmd <- commParse
                  return (IfThenElse cond thencmd elsecmd)

recursiveCommParse :: Parser Command
recursiveCommParse = do reserved lis "repeat"
                        name <- identifier lis
                        reserved lis ":="
                        value <- expParse
                        reserved lis "until"
                        c <- commLine
                        return (REPUNT (Let name value) c)
                     <|> do reserved lis "accum"
                            name <- identifier lis
                            reserved lis ":="
                            value <- expParse
                            reserved lis "until"
                            c <- commLine
                            return (ACCUM (Let name value) c)


commLine :: Parser Command
commLine  = ifElseParse
          <|> letParse
          <|> try exprParse
          <|> parenParse commParse
          <|> recursiveCommParse


commParse :: Parser Command
commParse = chainr1 commLine dacParse

-- ~ ifParse :: Parser Command
-- ~ ifParse  = do { reserved lis "if"
             -- ~ ; cond <- exprParse
             -- ~ ; reserved lis "then"
```

```
            -- ~ ; thencmd <- commParse
            -- ~ ; elsecmd <- elseParse
            -- ~ ; return (IfThenElse cond thencmd elsecmd)}

-- ~ elseParse :: Parser Command
-- ~ elseParse  = do { try (do { reserved lis "else"
                        -- ~ ; symbol lis "{"
                        -- ~ ; elsecmd <- comm
                        -- ~ ; symbol lis "}"
                        -- ~ ; return elsecmd}) }



-------------------------------------
-- Función de parseo
-------------------------------------

gandalf :: String


gandalf = "\n\n   .    ^       \n" ++
          "   | <###> |    \n" ++
          "   |  #*#  |    \n" ++
          "   :###`###:    \n" ++
          "   |  ###       \n" ++
          "   | #####      \n" ++
          " _|_.___.__    \n" ++
          "YOU SHALL NOT \n" ++
          "    PARSE     \n\n"

parseFile :: SourceName -> String -> ParseResult
parseFile sn st = case parse (totParser commParse) sn st of
                    (Left e) -> Left $ addErrorMessage (Message gandalf) e
                    c        -> c

parseInt :: String -> ParseResult
parseInt s = case parse (totParser commParse) "" s of
                (Left e) -> Left $ addErrorMessage (Message gandalf) e
                c        -> c
```

6.6. Main.hs

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleContexts #-}

module Main where

import Control.Exception (catch,IOException)
import System.IO.Error
import qualified System.Console.Readline (readline)
import Data.List
import Data.Char

import AST
import TypeEval
import RandomState
import Eval
import LexerParser
```

```
import Prelude
import System.Environment
import System.Random

main :: IO ()
main = do
    args <- getArgs
    case args of
        [] -> putStrLn "Error"
        ("-i":xs) -> interactiveMode xs
        (name:xs) -> executeFile False [] name

executeFile :: Bool -> Env -> String -> IO ()
executeFile inter st name = do
    g <- newStdGen
    file <- readFile name
    case parseFile name file of
        Left error -> print error
        Right t    -> do case eval g [] t of
                           Crash er -> do print er
                                          if inter then readevalprint g st else putStr "Finishe
                           Return reval@(ER (value, state, stdg)) -> do print reval
                                                                        if inter then readevalp
                                                                                 else putStr "F

haskdicelogo :: IO ()
haskdicelogo = do
    putStrLn "  _    _           _  _______   _                  "
    putStrLn " | |  | |         | |/ /  __ \\ (_)                 "
    putStrLn " | |__| | __ _ ___| ' /| |  | | _  ___ ___         "
    putStrLn " |  __  |/ _` / __|  < | |  | || |/ __/ _ \\       "
    putStrLn " | |  | | (_| \\__ \\ . \\| |__| || | (_|  __/     "
    putStrLn " |_|  |_|\\__,_|___/_|\\_\\_____/ |_|\\___\\___|  \n\n"

iprompt :: String
iprompt = "HkD> "

ioExceptionCatcher :: IOException -> IO (Maybe a)
ioExceptionCatcher e = if (isEOFError e) then do {print "Goodbye! See you soon!"; return Nothing}

helpText :: String
helpText = "\n Welcome to HaskDice! Commands available right now: \n\n" ++
           "- :load / :l  <file> loads a file \n" ++
           "- :quit / :q  Quits the interactive mode \n" ++
           "- :help / :?  You are here so.. you know.. helps (?)\n" ++
           "Have a pleasant day! \n\n"

interactiveMode :: [String] -> IO ()
interactiveMode []       = do
        haskdicelogo
        g <- newStdGen
        readevalprint g [] -- "infinite" loop with random number generator + empty state (no varia

readevalprint :: StdGen -> Env -> IO ()
readevalprint g st = do maybeline <- catchIOError (System.Console.Readline.readline iprompt) ioExc
                        case maybeline of
                             Nothing      -> putStr "Unexpected error\n"
                             Just ""      -> readevalprint g st
```

```
                                Just ":q"    -> putStr "Goodbye! See you soon! :D\n"
                                Just ":quit" -> putStr "Goodbye! See you soon! :D\n"
                                Just ":help" -> do {putStr helpText ; readevalprint g st}
                                Just ":?"    -> do {putStr helpText ; readevalprint g st}
                                Just line    -> loadOrInter line g st

loadOrInter :: String -> StdGen -> Env -> IO ()
loadOrInter line g st = if isPrefixOf ":l" line then do let (_,t') =  break isSpace line
                                                            t      =  dropWhile isSpace t'
                                                        executeFile True st t
                                                else case (parseInt line) of
                                                          (Left e) -> do {print e; readevalprint g
                                                          (Right res) -> case eval g st res of
                                                                              Crash e
                                                                              Return reval@(ER (va
```