Pràctica obligatòria de Haskell. Programes imperatius

1 Presentació

Volem representar programes escrits en un llenguatge imperatiu molt senzill

```
INPUT X;
INPUT Y;
IF X > 0 OR X = 0 OR NOT 0 > Y THEN
  Z := 1;
  WHILE X > Y
  DO
    X := X - 1;
    Z := Z * Z
  END
ELSE
  Z := 0;
END
PRINT Z;
```

L'objectiu de la pràctica és fer un interpret per aquests programes i fer un detector senzill de possibles plagis.

Considerem que les variables es representen amb identificadors (`Ident`) que són `Strings`.

Aquest programes s'executen sobre una llista de valors i retornen una llista de valors o un missatge d'error. Cada `INPUT` sobre una variable agafa un valor de la llista d'entrada i cada `PRINT` d'una variable posa un valor a la llista de sortida. L'ordre de la sortida ha de coincidir amb l'ordre en que s'han fet els prints.

2 Generació del l'arbre de sintaxi abstracta (AST)

Feu el que es demana als següents apartats respectant els noms de les classes, els tipus i les funcions que s'indiquen.

1. Definiu un `data` **polimòrfic** (Command a) que permeti tractar programes on les constants puguin ser de qualsevol tipus, per exemple, `Int`, `Integer`, `Double`, etc.

Que permeti representar l'assignació (amb el constructor `Assign`), l'input (amb el constructor `Input`), el print (amb el constructor `Print`), la composició seqüencial (obligatòriament) com a **llista** de Command (amb el constructor `Seq`), el condicional (amb el constructor `Cond`) i la iteració (amb el constructor `Loop`). Per això també cal un data polimòrfic per representar les expressions booleanes i un altre per les expressions numèriques.

En les expressions booleanes podem tenir `AND`, `OR` i `NOT` (pels que usarem les mateixes paraules pels constructors), més els comparadors relacionals `>` (amb el constructor `Gt`) i `=` (amb el constructor `Eq`) entre expressions numèriques.

En les expressions numèriques podem tenir variables (amb el constructor `Var`), constants (amb el constructor `Const`) i els operadors de suma (`+`, amb el constructor `Plus`), resta (`-`, amb el constructor `Minus`), producte (`*`, amb el constructor `Times`) i divisió (`/`, amb el constructor `Div`) entre expressions numèriques.

Per simplificar la lectura considerarem que a les expressions no hi ha parèntesis i que les expressions booleanes s'avaluen d'esquerra a dreta i les numèriques amb les prioritats estàndard. A més, sempre hi haura un blanc abans i després dels operadors `:=` , `>` , `=` , `+` , `-` , `*` , `/` .

2. Definiu una funció `readCommand:: Read a => String -> Command a`. El string de l'entrada només pot incloure blancs o salts de línia a més de les coses pròpies del llenguatge. Podeu assumir que serà sintàcticament correcte.

3. Definiu correctament la funció de mostrar en el tipus `Command` com a instància de la classe `Show`, de manera que el resultat sigui un String, que al fer `putStr` del `show` es mostri el codi indentat (amb dos blancs més en cada nivell) tal com a l'exemple anterior.

3 Interpret

En aquesta part volem implementar un interpret per al nostre llenguatge. Per això, feu el que es demana als següents apartats respectant els noms de les classes, els tipus i les funcions que s'indiquen.

1. Definiu en Haskell una nova classe de tipus anomenada `SymTable` de tipus m que representen l'estat de la memòria, és a dir que ens permet mantenir i consultar els valors que conten les variables. Noteu que m és un contenidor com passa a la class Functor o Monad. Aquesta nova classe tindrà les següents operacions:

   (a) `update` que donada una memòria (genèrica) de elements de tipus `a` (és a dir `m a` , una variable (String) i un valor de tipus `a` retorna la nova memòria actualitzada (de tipus `m a`).

(b) `value` que donada una memòria (de tipus `m a`) i una variable (String) retorna un valor de tipus `a` que és el que té la variable a la memòria.

(c) `exists` que donada una memòria (de tipus `m a`) i una variable (String) retorna un valor de tipus `Bool` que indica si la variable té un valor assignat a la memòria.

(d) `start` que és de tipus `m a` i que representa la memòria inicial (buida).

2. Feu un instance de la classe `SymTable` representant la memòria amb una llista de parells on el primer és un String i el segon el tipus genèric dels elements del programa. Us caldrà definir un nou data.

3. Feu un instance de la classe `SymTable` representant la memòria amb un arbre binari de cerca sobre parells com els de l'apartat anterior, ordenat per la primera component. Us caldrà definir un nou data.

4. Feu una funció `interpretCommand :: (SymTable m, Num a, Ord a) => m a -> [a] -> Command a -> ((Either [a] String),m a, [a])`, que interpreta un AST per una memòria i una entrada donada i retorna una tripleta que conté a la primera component la llista amb totes les impressions o bé un missatge d'error, i a la segona i la tercera component la memòria i l'entrada respectivament després d'executar el codi.

   Per a fer aquesta funció heu de fer una funció que avaluï expressions booleanes i una que avaluï expressions numèriques. L'avaluació d'aquestes darreres expressions dóna error si conté alguna variable sense assignar o si es produeix una divisió per zero. Qualsevol programa o expressió que contingui una subexpressió que avalua a error, també avalua a error. S'ha de comunicar quin ha estat l'error: "undefined variable" or "division by zero". Per a l'avaluació d'expressions numèriques s'ha de fer un tractament especial de la divisió, a més de controlar que no hi hagi divisió per zero. La divisió s'ha d'interpretar com el quocient (és a dir, la part entera), però no podeu usar el `div` perquè això requereix que el tipus sigui de la classe `Integral`. Per això, heu d'implementar la vostra pròpia divisió entera `mydiv :: (Num a, Ord a) => a -> a -> a`, que serà la que usareu per avaluar la divisió en les expressions.

5. Usant la funció anterior feu una funció `interpretProgram:: (Num a,Ord a) => [a] -> Command a -> (Either [a] String)`, que avalua un codi complet per a una entrada donada.

4 Detecció de còpies

Per a realitzar una comprovació de la similitud de dos programes, compararem les seves estructures tenint en compte només les instruccions que impliquen bifurcació, és a dir, els ifthenelse i els while. Per aixo heu de fer el següent.

1. Definiu una funció `expand :: Command a -> Command a`, que elimina les connectives AND i OR de les condicions dels ifthenelse introduint més condicionals i mantenint el mateix comportament.

2. Definiu una funció `simplify :: Command a -> Command a`, que eliminat totes les instruccions sense bifurcació i deixant un `(Seq [])` com a l'únic possible codi que no sigui un condicional o un loop.

3. Considerem que dos programes són iguals si les seves versions expandides i simplificades són estructuralment isomorfes, és a dir que, sense tenir en compte les condicions, són iguals sota permutació de la part del "then" i la part del "else" dels condicionals.

   Definiu `Command` com a instància de la classe `Eq` usant aquesta noció d'igualtat.