Pràctica obligatòria de Haskell. Programes imperatius. Part 1

1 Presentació

Per a la primera part només es poden usar funcions de l'entorn `Prelude`.

Volem representar programes escrits en un llenguatge imperatiu 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
```

o també:

```
INPUT X
EMPTY P
WHILE X > 0 OR X = 0
DO
 INPUT Y
 PUSH P Y
END
S := 0
SIZE P L
WHILE L > 0
DO
  POP P Y
  S := S + Y
  L := L - 1
END
PRINT S
```

L'objectiu de la pràctica és fer un interpret i un tester per aquests programes. La primera part considerarà la lectura i la interpretació de programes. La segona part tractarà la part de testing.

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 expressió numèrica 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. Les operacions amb piles només accepten identificadors excepte al segon paràmetre del push, que pot ser una expressió numèrica.

2 Generació del data AST en Haskell

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`), les operacions amb piles (amb els constructors `Empty`, `Push`, `Pop` i `Size` que només accepten identificadors excepte al segon paràmetre del push) 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, que són un nou data genèric `BExpr`, 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, que són un nou data genèric `NExpr`, 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 (`*` i amb el constructor `Times`) 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.

2. 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 als exemples anteriors.

3 Lectura de programes

Feu una gramàtica per al llenguatge usant PCCTS i adapteu el printAST per a que generi una expressió que sigui del data `Command` de manera que fent que `Command` faci un deriving del Read, poguem llegir el programa d'entrada simplement fent un read de l'expressió generada.

Per a la segona part haureu de fer un petit script que combini aquesta lectura amb l'execució.

4 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 un nou data anomenat `SymTable a` que ens permet mantenir i consultar els valors que conten les variables i saber el seu tipus.

   Noteu que tenim variables de dos tipus diferents, o bé és el tipus general `a` o bé són piles de `a` . Es considera que totes les variables són globals i, per tant, poden tenir un únic tipus.

2. Definiu en Haskell una nova classe de tipus anomenada `Evaluable` de tipus e que representen expressions. Noteu que e és un contenidor com passa a la class Functor o Monad. Aquesta nova classe tindrà les següents operacions:

   (a) `eval :: (Num a, Ord a) => (Ident -> Maybe a) -> (e a) -> (Either String a)`.

   (b) `typeCheck :: (Ident -> String) -> (e a) -> Bool`.

   Feu que tant `NExpr` com `BExpr` siguin instance de la classe `Evaluable`.

   Per això 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 de tipus incorrecte.

3. Feu una funció `interpretCommand :: (Num a, Ord a) => SymTable a -> [a] -> Command a -> ((Either String [a]),SymTable 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.

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

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" o "empty stack" o "type error".