Pràctica de Haskell. Programes de fontaneria. Part 1 i 2

1 Presentació

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

Volem representar programes escrits en una variant del llenguatge de fontaneria que vam usar a la pràctica de compiladors

```
INPUT X
INPUT Y
T1 = TUBE X Y
T2 = TUBE 10 Y
(T3,T4) = SPLIT T2
INPUT Z
T6 = TUBE Z 2
TCOP = T6
C1 = CONNECTOR Y
T5 = MERGE T3 C1 TCOP
T5 = MERGE T2 C1 TCOP
T5 = MERGE T1 C1 T3
TN = TUBE 5 Y

PRINT LENGTH(T4)
PRINT DIAMETER(T5)
(T7,T8) = SPLIT T5
C2 = CONNECTOR Y
C3 = CONNECTOR Y
T9 = TUBE LENGTH(T7) DIAMETER(T8)
DRAW TUBE LENGTH(T7) DIAMETER(T8)
T10 = MERGE MERGE T7 C2 T8 C3 TN
(T1,T2) = SPLIT T10
C4 = CONNECTOR DIAMETER(T1)
INPUT Y
C5 = CONNECTOR Y
IF (DIAMETER(C4) == DIAMETER(C5))
  T11 = MERGE T1 C4 MERGE T4 C5 T2
ELSE
```

```
ENDIF
V = TUBEVECTOR OF 5
WHILE (NOT FULL(V) AND (LENGTH(T11) > 3 AND LENGTH(T11) < 30 OR LENGTH(T11) == 0))
  (X,Y) = SPLIT T11
  DRAW X
  PUSH V X
  T11 = Y
ENDWHILE
FT = TUBE 0 DIAMETER(T11)
WHILE (NOT EMPTY(V) AND LENGTH(FT) < 8)
  POP V TAUX
  CX = CONNECTOR DIAMETER(T11)
  FT = MERGE FT CX TAUX
ENDWHILE
```

L'objectiu de la pràctica és fer un interpret i un tester per aquests programes. La primera part considera la presentació i la interpretació de programes. La segona part tracta de fer una mica 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. També tenim l'operació `DRAW`, que donada una expressió que defineix un tub el mostra, amb una seqüència de valors numèrics, on el primer és el diàmetre i la resta són la longitud de tots els tubs bàsics (no composats per altres tubs) que la formen tal i com s'ha construït. L'ordre de la sortida ha de coincidir amb l'ordre en que s'han fet els prints i els draws.

Aquestes operacions són noves respecte a la pràctica de compiladors. També, com podeu veure a l'exemple, hem afegit condicionals i només es pot usar `LENGTH` i `DIAMETER` dins d'expressions numèriques. Per altra banda podem posar expressions de tub (amb `MERGE` i `TUBE`) o expressions de connector (amb `CONNECTOR`) dins d'altres `MERGE` o dins del `DRAW`. Per a qualsevol dubte sobre el que es pot fer, mireu la gramàtica que apareix a l'arxiu `plumber.g`

Igual que a la pràctica de compiladors només l'assignació entre variables fa una còpia, la resta d'usos de les variables fa perdre el valor (si es completa l'operació a nivell global).

2 Representació del programa 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 representar programes on les constants puguin ser de qualsevol tipus, per exemple, `Int`,

Integer, Double, etc. Per això també cal un data polimòrfic per representar les expressions booleanes, expressions numèriques, expressions de tub i expressions de connector.

El data (Command a) ha de permetre representar la copia del contingut de variables (amb el constructor `Copy`), l'assignació (amb els constructors `TAssign` o `CAssign` segons si s'assignen expressions de tub o de connector), la partició d'un tub en dos (amb el constructor `Split`), l'input (amb el constructor `Input`), el print (amb el constructor `Print`), el draw (amb el constructor `Draw`), 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`). També tenim constructors per a declarar un vector (amb el constructor `DeclareVector`) i per afegir i treure elements (amb els constructors `Push` i `Pop`).

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`), `<` (amb el constructor `Lt`) i `==` (amb el constructor `Eq`) entre expressions numèriques. També podem preguntar si un identificador de vector és buit o ple (amb els constructors `Empty` i `Full`).

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`) i el producte (`*`, amb el constructor `Times`). A més, podem usar la longitud i el diàmetre (amb els constructors `Length` i `Diameter`) d'un identificador.

En les expressions de tub, que són un nou data genèric `TExpr`, podem tenir variables (amb el constructor `TVar`), operacions de mescla (amb el constructor `Merge` que opera amb dos expressions de tub i una de connector) i operacions per crear un tub bàsic (amb el constructor `Tube` que opera amb expressions numèriques).

Finalment, en les expressions de connector, que són un nou data genèric `CExpr`, podem tenir variables (amb el constructor `CVar`) i operacions per crear un connector (amb el constructor `Connector` que opera amb una expressió numèrica).

Per a més detalls (i per si falta algun constructor en aquesta descripció), mireu el resultat d'aplicar el programa generat per l'arxiu `plumber.g` sobre el joc de proves que s'adjunta a `inp.txt`

Noteu que l'expressió Haskell generada per aquest programa el podeu definir de tipus `Command Int` or `Command Double`. De fet, qualsevol programa de tubs que li passeu us el convertirà en el `Command` que el representa.

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 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) d'elements construïts sobre el tipus `a` (és a dir `m a` , una variable (String) i un valor de tipus `(Val 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 `Maybe (Val a)` que és el que té la variable a la memòria, en cas que hi sigui.

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

   Heu de triar com definiu el data polimòrfic `(Val a)` per representar els diferents valors que poden prendre els diferents tipus de variables. Noteu que tenim variables numèriques, de connector, de tub i de vector de tubs. Useu diferents constructors per diferenciar aquests tipus de valors.

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 és `(Val a)`. 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 :: (Num a, Ord a, SymTable m) => m a -> [a] -> Command a -> ((Either String [a]), m a, [a])` , que interpreta un Command (programa) per una memòria i una entrada donada i retorna una tripleta que conté a la primera component la llista amb totes les impressions (amb print o draw) 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 això també heu de fer funcions per avaluar expressions booleanes, numèriques, de tubs i de connectors.

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 "variable content no longer exists" o "unmatched diameter" o "empty vector" o "full vector" o "type error".

5. 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.

4 Test comparatiu

Feu un main Haskell (i les funcions auxiliars que calguin) que ens demani per l'entrada estàndard si els programes treballen amb enters o amb reals (opcions 0 i 1 respectivament) i que a continuació llegeixi dos programes ja amb el format per a poder llegir-lo amb un `read` que estan respectivament a la primera línia dels arxius "programhs1.txt" i "programhs2.txt" (sempre tindran aquests noms). A continuació ens preguntarà si volem usar una memòria representada amb una llista o amb un arbre binari de cerca (opcions 0 i 1 respectivament). Finalment, mostrarà un menú (text) per a poder fer proves amb els programes de dues formes diferents:

1. *Test comparatiu manual.* Si s'escull aquesta opció (identificada amb 0), es demanarà per l'entrada una llista (que serà del tipus que calgui) i mostrarà la llista resultant d'executar els dos programes amb aquesta llista d'entrada llegida (si calen més valors a l'entrada que els que s'han introduït, aquests hauran de generar-se aleatòriament). També indicarà si els resultats són iguals i mostrara les variables que tenen un valor diferent a la memòria final després de les dues execucions (això inclou els que només apareixen en una de les dues). Per a mostrar aquesta diferència entre les memòries, mostreu una llista de tripletes que contenen el nom de la variable, el valor en la primera memòria i el valor en la segona (feu que Val sigui de la classe `Show`).

2. *Test comparatiu automàtic.* Si s'escull aquesta opció (identificada amb 1), es demanarà el nombre de tests diferents que es vol passar. Donat aquest nombre k, es generaran k llistes de test (del tipus que calgui) diferents amb les que s'executaran els dos programes.

   Per a cada llista de test es mostraran els resultats tal com en el test anterior i, a més s'indicarà al final el nombre de test que han donat llistes de sortida diferents (no importa si han tingut la memòria diferent).

Excepte els programes a executar, tota la resta de dades es llegiran per l'entrada estàndard.

Per a poder llegir d'un fitxer caldrà importar el mòdul `System.IO`, posant

```
import System.IO
```

i usar les operacions `openFile` (amb `ReadMode`), `hGetLine` i `hClose`, vistes a classe.

Feu un script que es digui "test" que rep dos arxius per paràmetre i que serveix per connectar l'execució del parser "plumber" (pccts) amb el programa Haskell. Així, passareu els dos programes a interpretar en dos arxius (i posareu la sortida de les dues crides a "plumber" en els arxiu "programhs1.txt" i "programhs2.txt" respectivament). Com que la resta de dades que ens demani el programa Haskell es rebran per l'entrada estàndard, ha de ser possible redirigir l'entrada estàndard a un fitxer on estiguin totes les dades que demana el programa Haskell (a part del programam a executar) separades per salts de línia: codi de tipus de dades (0 o 1), codi de menú (0 o 1) i la resta de dades per a cada opció del menú.

Entregueu un únic fitxer amb el programa Haskell ".hs" i el script "test" que connecta l'execució. Si heu modificat el fitxer del pccts "plumber.g" entregueu-lo també (i indiqueu perquè l'heu modificat en un comentari a l'inici del fitxer).