Pràctiques de Llenguatges, Tecnologies i Paradigmes de programació. Curs 2015-16

Part II Programació Funcional

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Pràctica 3

Índex

1 Objectiu de la Pràctica

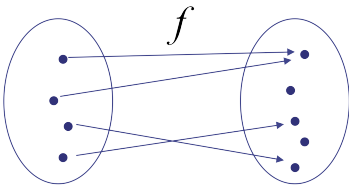
L'objectiu d'aquesta pràctica és introduir el Haskell i presentar les facilitats bàsiques d'un entorn de programació funcional, en concret GHCi. L'entorn GHCi és la versió interactiva del "The Glasgow Haskell Compiler"(GHC), disponible en `http://www.haskell.org/ghc`. GHC és el compilador més avançant per al llenguatge de programació funcional Haskell (veure `http://www.haskell.org`). L'avantatge de GHCi respecte al compilador GHC o a l'intèrpret RunGHC és l'entrada interactiva i millors missatges d'error, com també el depurador d'errors basat en traces.

L'entorn GHCi, i el sistema GHC, admeten totes les característiques, de Haskell i serveix de plataforma de proves per a noves funcionalitats. Actualment, GHC admet totalment l'estàndard "Haskell 2010 language". Per a més informació sobre l'estàndard Haskell 2010, podeu consultar `http://www.haskell.org/onlinereport/haskell2010`.

2 Què és el Haskell

El Haskell és un llenguatge de programació purament funcional. A diferència dels llenguatges de programació imperatius tradicionals (`C`, `C++`, `Java`, `C#`, etc.), en els quals es posa l'accent en el *com* (o siga, que són llenguatges en els que les solucions dels problemes s'expressen com seqüències de tasques que s'han d'executar per resoldre'ls), en els funcionals es fa més èmfasi en el *què*, això és, en la descripció dels elements del problema així com de les relacions entre aquests elements.

En un llenguatge de programació funcional, es defineixen funcions basades en el concepte de funció matemàtica. Cal recordar que les funcions matemàtiques es defineixen entre dos conjunts d'elements, denominats respectivament *domini* i *codomini*.



En els llenguatges purament funcionals, com el Haskell:

- Les funcions no tenen efectes col·laterals, per la qual cosa l'execució de les mateixes no pot alterar elements globals. De fet, davant d'un argument d'entrada, tornen sempre el mateix resultat. D'altra banda, el resultat retornat per una funció només depèn dels seus arguments d'entrada (el que s'anomena *transparència referencial*).

Dues funcions són iguals si i només si tornen el mateix davant els mateixos arguments.

- El domini o el codomini d'una funció pot contenir una altra funció, o el que és el mateix, els arguments o resultat d'una funció poden ser, al seu torn, una altra funció. Idènticament, una expressió pot avaluar a una funció. Per exemple:

```
> let f = (3.5*)   -- f és "multiplicar per 3.5"
> f 5              -- s'aplica f a l'argument 5
> 17.5
```

- L'ordre en la definició de les funcions no és rellevant (tot i que la mateixa funció pot ser especificada, algunes vegades, en una sèrie de casos l'ordre sí que pot ser rellevant).

- La inexistència d'efectes laterals facilita el tractament paral·lel i concurrent.

A més, el Haskell és un llenguatge fortament tipat, que resol l'assignació de tipus estàticament, el que implica que totes les expressions i funcions tenen un tipus assignat durant el procés de compilació o interpretació. Tota funció té la seua *signatura*, perfil, o definició de domini i codomini. Per exemple:

```
exemple :: String -> Int
```

determina que la funció `exemple` está definida de les cadenes (`String`) en els enters (`Int`). Haskell, addicionalment, pot inferir els tipus de les expressions o les signatures de les funcions, a partir de com estan construïdes i dels tipus explícits o implícits de les subexpressions que puguen aparèixer en les mateixes.

Per exemple (com es veurà al llarg d'aquesta pràctica, es pot determinar el tipus d'una expressió utilitzant la comanda `:t`) per a la definició vista abans:

```
> let f = (3.5*)   -- f és "multiplicar per 3.5"
...
> :t f             -- tipus de f?
> Double -> Double
```

on, com es pot veure, el Haskell ha inferit, a partir de la definició de la primera línia, que `f` és una funció de `Double` en `Double`.

Una característica rellevant del Haskell és que té *avaluació peresosa*[1], el que vol dir que retardarà o postergarà els còmputs mentre ho puga fer. Per exemple, l'execució de les dues primeres línies de codi Haskell que segueix té com a resultat el que es mostra en la tercera:

```
> let m7 = [x | x <-[1..], x `mod` 7 == 0]
> take 10 m7
> [7,14,21,28,35,42,49,56,63,70]
```

L'expressió `[1..]` representa en Haskell la llista de tots els enters a partir de l'1. La primera línia associa la llista infinita de tots els múltiples de 7 a l'identificador `m7`.

L'expressió `take n ls`, torna els primers `n` elements de certa llista `ls`. Com es pot veure, el càlcul de la primera línia s'ha demorat, en funció del que expressa la segona, de manera que tan només s'han generat realment els elements inicials necessaris d'una llista infinita.

A més de tot l'anterior, Haskell és un llenguatge molt complet (té, per exemple, un sistema de tipus potent, que inclou genericitat), robust (amb eines de desenvolupament, documentació i ajuda) i usat industrialment, que representa el resultat actual de l'evolució d'altres llenguatges funcionals que es van originar als anys 60. Es dedicarà la resta d'aquest butlletí i les pràctiques a introduir el seu ús.

3 Com s'usa GHCi

L'entorn GHCi consta, bàsicament, d'un intèrpret de tipus text. L'entorn GHCi ve amb el compilador GHC, el qual és de domini públic i es distribueix amb llicència GNU. GHC sol estar disponible com un paquet per a ser instal·lat en la majoria de les distribucions Linux. En els ordinadors del laboratori de pràctiques, GHCi opera sobre Linux, encara que hi han versions públiques per a Windows i Mac, entre altres sistemes. Es pot obtindre esta i altres versions de GHCi via WWW en la següent adreça d'internet:

`http://www.haskell.org/ghc/`

3.1 Per començar

El sistema està instal·lat en els laboratoris de pràctiques i l'aplicació està accessible des de qualsevol directori. L'intèrpret respon escrivint `ghci` des de la terminal de Linux:

[1] Lazy evaluation.

```
$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
```

No hi ha un editor de text associat a GHC però molts editors de text coneguts admeten el format Haskell i acoloreixen i disposen de funcionalitats específiques per a programes Haskell. Per a l'entorn Linux, recomanem EMACS, l'editor de programari lliure per excel·lència (vegeu `http://www.gnu.org/software/emacs`), GEDIT, l'editor oficial de GNOME (vore `https://wiki.gnome.org/Apps/Gedit`) i GEANY, un editor també basat en el GTK2 toolkit (vore `http://www.geany.org`). Per a Windows, recomanem CONTEXT, un editor gratuït (vore `http://www.contexteditor.org`), i també les versions de GEANY i d'EMACS per a aquest sistema.

Quan s'entra en l'entorn, l'intèrpret queda a l'espera de que s'introdueixca un comandament o una expressió a avaluar. La majoria d'ordres en GHCi comencen amb ':', seguit d'un o més caràcters. És important recordar dues ordres d'ús freqüent:

`? :q` Eixir de GHCi
`? :?` Mostrar la llista de totes les ordres disponibles

Alguns dels comandaments més habituals es mostren a continuació. Qualsevol d'ells es pot abreujar amb l'inici del mateix (només cal el primer caràcter), p.ex. `:lo` o `:l` per a `:load`.

```
:load <modulename>   - loads the specified module
:reload              - reloads the current module
:type <expression>   - print type of the expression
:info <function>     - view function in on-line documentation
:quit                - exit the interpreter
:help                - shows available commands
```

Per interrompre el procés en curs, es pot utilitzar, com és habitual, `^C`.

A continuació es descriuen algunes de les característiques de l'entorn GHCi. Per a una descripció detallada, consultar la documentació que es troba 'on line'.

3.2 Usant GHCi

Per avaluar una primera expressió, es pot teclejar el següent exemple en la línia d'entrada de comandes:

```
2 + 3 * 8
```

seguida de la tecla `RET`. GHCi avalua l'expressió i imprimix el seu valor just després de l'ordre, i mostra novament el *prompt* `Prelude` [2] al finalitzar:

```
Prelude> 2 + 3 * 8
26
Prelude>
```

Les funcions aritmètiques com `+` i `*` estan predefinides i s'escriuen en notació infixa. Altres funcions aritmètiques s'escriuen en notació prefixa, com ara `div` o `mod`. No obstant això, les funcions infixes es poden utilitzar en notació prefixa usant parèntesis, i les funcions prefixes es poden utilitzar en notació infixa usant l'accent cap a l'esquerra.

```
> (+) 2 3
5
> div 4 2
2
> 4 `div` 2
2
```

De la mateixa manera, es pot teclejar l'expressió lògica

```
> True && False
```

i la resposta serà

```
False
```

Ara es pot preguntar pel tipus d'esta expressió (comandament `:type`):

```
>:t True && False
```

I l'intèrpret respon:

```
True && False ::  Bool
```

indicant que l'expressió, quan s'avalue, donarà un valor de tipus lògic.

[2] `Prelude` fa referència al mòdul per defecte que ha carregat inicialment el sistema, les definicions del qual es poden utilitzar en la sessió.

3.3 Edició i Càrrega de Programes

En el nivell més alt, un programa en Haskell és un conjunt de mòduls. Cada mòdul conté una col·lecció de declaracions, que inclou el nom del mòdul, el tipus de les funcions contingudes en aquest i la definició d'aquestes funcions.

Per exemple, utilitzant l'editor, escribiu el programa:

```
module Signum where
-- Definició de la funció signum' (signe):
signum' x = if x < 0 then -1 else
            if x == 0 then 0 else 1
```

A continuació, guardeu el fitxer. Per a GHC i GHCi, el nom del fitxer ha de tenir l'extensió `.hs`. Encara que el nom d'un mòdul no té per què coincidir amb el nom del fitxer on s'ha escrit, sol ser molt recomanable fer-ho així. El primer caràcter del nom utilitzat per al mòdul ha d'estar en majúscules.

Cal observar que les línies precedides pels caràcters `--` són comentaris. També ho són les seqüències de codi parentizadas pel símbol d'obertura: `{-` i el de tancament: `-}`.

Tornant a la finestra de l'intèrpret GHCi, es carrega el programa emmagatzemat en el fitxer `Signum.hs` introduint `:load` en la línia d'ordes amb el nom del fitxer i la ruta d'accés.

```
Prelude> :load Signum.hs
[1 of 1] Compiling Signum          ( Signum.hs, interpreted )
Ok, modules loaded: Signum.
*Signum>
```

Com ja s'ha dit, GHCi té una funció d'autocompleció usant la tecla tabulador, com en la majoria d'intèrprets d'ordres dels sistemes operatius. Per exemple, escrivint `:l S` i pulsant el tabulador, GHCi completa la comanda amb el nom complet del fitxer.

El mateix passa quan s'avaluen expressions, on GHCi pot completar el nom d'una funció amb un nom més llarg del comú. A continuació indiquem algunes invocacions senzilles a la funció `signum'`.

```
*Signum> signum'(0)
0
*Signum> signum'(100)
1
*Signum> signum'(-100)
-1
```

Convé recordar que cada ordre `:load` inicialitza la base de dades de l'intèrpret, per això només es poden avaluar expressions que continguen

funcions predefinides o definides en el mòdul actual (en aquest cas, la funció `signum'`) o en algun mòdul importat pel mòdul actual.

3.4 Missatges d'error i alertes

GHCi informa de possibles errors sintàctics i de tipus durant la càrrega d'un fitxer. Per exemple, si s'obre l'editor i s'escriu el programa:

```
module Hello where

hello n = concat (replicate n 'hello ')
```

se salva en fitxer `Hello.hs` i es carrega a continuació en GHCi utilitzant el comandament `:load`, es produeix el següent missatge d'error:

```
Hello.hs:3:32:
    lexical error in string/character literal at character 'e'
Failed, modules loaded: none.
```

Com es veu, l'intèrpret mostra la posició exacta de l'error (línia 2, columna 32). Corregiu l'error, reemplaçant les cometes simples per dobles cometes, és a dir, `"hello"`. Intenteu carregar novament el programa amb l'ordre `:r` [3].

Ara la compilació no produeix errors, però és una bona pràctica escriure explícitament en cada mòdul el perfil de les funcions definides en ell. El perfil de les funcions indica el seu tipus. Si no s'escriu, el compilador intenta deduir o inferir-ho de la definició. Per exemple, es pot consultar el tipus inferit automàticament per l'intèrpret per a la funció `hello` de la manera següent:

```
*Hello> :t hello
hello :: Int -> [Char]
*Hello>
```

el que indica que la funció `hello` rep un `Int` i retorna una llista de `Char`. Afegir el perfil a la definició de la funció és molt aconsellable i ajuda a detectar errors.

Comentari addicional: les llistes d'elements es defineixen en Haskell usant els claudàtors, p. ex. `[Char]` per a una llista de caràcters, `[Int]` per a una llista de nombres enters, etc. A més el tipus de dades `String` en Haskell es manipula com una llista de caràcters, de manera que l'expressió `"hello"` es representa realment com la llista `['h','e','l','l','o']`. Les llistes d'elements s'estudien en detall en la Pràctica 4.

[3] Comandament `:reload`, càrrega de nou l'últim fitxer.

Després de carregar el programa i comprovar que no té errors, l'intèrpret podrà avaluar expressions que contenen crides a les funcions definides en aquest. Per exemple, si s'avalua l'expressió `hello 10`, s'obté la següent cadena de caràcters.

```
*Hello> hello 10
"hello hello hello hello hello hello hello hello hello hello "
```

3.5 Tipus

Haskell és un llenguatge fortament tipificat. La comprovació de tipus es realitza en temps de compilació. GHCi no sols és capaç de detectar errors de tipus, sinó que també pot suggerir possibles formes de resoldre els conflictes. Obriu l'editor, escriviu el programa següent amb nom `Typeerrors.hs`.

```
module Typeerrors where
convert :: (Char, Int) -> String
convert (c,i) = [c] ++ show i

main = convert (0,'a')
```

Comentari addicional: en Haskell es poden definir tuples d'elements de qualsevol longitud simplement fent servir els parèntesis i la coma, p. ex. el tipus de dades `(Char, Int)` i el patró `(c,i)` en la definició de la funció anterior.

Carregant el fitxer anterior en l'intèrpret, es produeix el següent missatge d'error:

```
Typeerrors.hs:6:18:
    Couldn't match expected type `Int' against inferred type `Char'
    In the expression: 'a'
    In the first argument of `convert', namely `(0, 'a')'
    In the expression: convert (0, 'a')
Failed, modules loaded: none.
```

que indica que l'expressió `(0, 'a')` és de tipus `(Int, Char)` quan s'esperava una expressió de tipus `(Char , Int)` d'acord amb el tipus definit en el programa per a la funció `convertir`. Una possible solució a aquest problema consisteix en canviar l'orde dels elements `0` i `'a'`.

3.6 Definició de Funcions

La definició d'una funció `f` consta en general de la declaració del seu perfil o tipus i un nombre d'equacions. Com ja hem comentat, la declaració del tipus és opcional, ja que GHCi infereix els tipus automàticament, encara que és recomanable incloure la declaració en el programa.

En Haskell hi ha els denominats *símbols constructors* i *símbols definits*. Els símbols definits són els que disposen d'alguna equació que definisca la seua avaluació, p. ex. les funcions `signum'`, `hello` i `convert` vistes anteriorment. Els símbols constructors són els símbols que no estan definits per cap equació. Un esquema general per a definir una funció pot ser el següent:

```
f ::  Tipus_1 -> ... -> Tipus_n -> Tipus_f
f (patró_1) ... (patró_n) = expressió
```

Cadascuna de les expressions $\texttt{patró}_i$ representa un argument de la funció i es coneix com a *patró*. Un patró només pot contindre constructors o variables però no pot contindre funcions definides. Els noms de les variables i de les funcions s'escriuen en minúscules. Cal notar la diferència entre escriure $\texttt{add}(1, 2, 3)$, que és la sintaxi habitual en la majoria de llenguatges de programació, i `add 1 2 3`, que és la sintaxi funcional (vore Secció 4).

És possible ometre els parèntesis dels patrons en les definicions de funció, sempre que no hi haja ambigüitat i s'ajuste a la definició del tipus de la funció. Per exemple, es pot introduir la definició següent per a la funció que calcula la llargària d'una llista:

```
module Length where

length' [] = 0
length' (x:t) = 1 + length' t
```

Comentari addicional: una llista es pot veure com una seqüència d'expressions connectades pel símbol `:` i acabades amb `[]` (que representa a la llista buida), p. ex. la cadena de caràcters `"hello"`, en realitat és la llista de caràcters `['h','e','l','l','o']`, en Haskell també es pot escriure com a `'h':'e':'l':'l':'o':[]`. Les llistes d'elements s'estudien en detall en la Pràctica 4.

Ara, una vegada carregat aquest programa en l'intèrpret, es pot demanar que s'avalue l'expressió `length' ([1,2,3])`, que torna el valor `3`. En aquest cas, també es pot escriure la mateixa expressió sense els parèntesis, `length' [1,2,3]`, ja que no hi ha ambigüitat perquè `length'` te un únic argument. D'aquesta manera s'obté la mateixa resposta. A més, la funció `length'` admet altres tipus de llistes d'elements, així, p.ex. l'expressió `length'("hola")` torna `4` i, en canvi, l'expressió `length'(["hola"])` torna `1`.

Es poden definir funcions usant *equacions condicionals*, o *amb guarda*, la seva forma general és:

```
f x1 x2 ...  xn
                  | condicio1 = exp1
                  | condicio2 = exp2
                  ⋮
                  | condicionm = expm
```

Per exemple, per a la funció potència, es pot definir la funció següent:

```
module Power1 where
power1  :: Int -> Int -> Int
power1 _ 0 = 1
power1 n t = n * power1 n (t - 1)
```

on _ representa una variable (com `n` o `t`) però el seu nom és irrellevant. Una versió més eficient fent servir equacions condicionals és:

```
module Power2 where

power2  :: Int -> Int -> Int
power2 _ 0 = 1
power2 n t
       | even t = power2 (n * n) (div t 2)
       | otherwise =  n * power2 (n * n) (div t 2)
```

on `even` i `div` són funcions predefinides i l'expressió `otherwise` sempre s'avalua a `True`.

Comentari addicional: L'orde d'aparició de les equacions en el programa és important ja que Haskell buscarà la primera equació aplicable de dalt a baix en el programa i, una vegada trobada una equació aplicable, no continua provant amb la resta. El mateix ocorre amb les equacions condicionals, on les condicions s'avaluen de dalt a baix i, una vegada una condició s'avalua a `True`, no comprova la resta de condicions. Per exemple, per a la funció `power2`, si el segon argument és un `0`, aplicarà la primera equació i no intentarà aplicar la segona. En canvi si el segon argument és diferent de `0`, la primera equació no és aplicable i comprovarà la condició `even t` de la segona equació, els arguments de la qual no requereixen cap símbol constructor. Si aquesta condició s'avalua a cert, aplicarà la part dreta apropiada i, sinó, aplicarà la part dreta del cas `otherwise`.

Altres formes o expressions que es poden utilitzar en la definició de funcions en Haskell són les següents.

where. S'utilitza la forma `where` quan es vol definir una *associació de valors* local a una funció [4].

```
f x1 x2 ...  xn = exp
     where
        definicioFuncio1
        ...
        definicioFuncionm
```

Exemples d'ús:

```
f x y = (a+1) * (a+2)
      where a = (x + y) / 2

f x y = g (a+1) (a+2)
      where
        a = (x + y) / 2
        g x y = x * y

f x y = g (a+1) (a+2)
      where
        a = (x + y) / 2 ; g x y = x * y
```

let. Les expressions `let` tenen un propòsit semblant a les `where` (permeten associar valors a funcions en una expressió). Però, a diferència de les formes `where`, les `let` són realment expressions avaluables [5]. La seva sintaxi general és la següent:

```
f x1 x2 ...  xn =
     let definicioFuncio1
         ...
         definicioFuncionm
     in exp
```

Exemples d'ús:

```
f1 = let a = 3 + 2
     in a * a * a

f2 = 4 * (let a = 9 in a + 1) + 2
```

[4] Aquesta *associació de valors* té com a àmbit l'equació de la definició d'una funció on aparega.

[5] L'àmbit d'aquesta associació és exclusivament el de l'expressió on va inclosa.

Addicionalment, les expressions `let` es poden utilitzar en GHCi. Amb elles, es pot associar valors a funcions que es poden definir en una sessió, per exemple:

```
Prelude> let x = (2+)
Prelude> let y = x 4 + 3
Prelude> y
9
```

4 Currificació i aplicació parcial

En Haskell hi ha dues alternatives per definir funcions de dues o més arguments. Per exemple, les següents son dues definicions d'una operació d'addició, `add` y `cAdd`:

```
add :: (Int, Int) -> Int
add (x,y) = x + y

cAdd :: Int -> Int -> Int
cAdd x y = x + y
```

Hi ha un isomorfisme entre els dominis d'aquestes dues versions de la funció[6]. La segona versió està currificada i té algunes avantatges respecte de la primera des del punt de vista de la programació:

- és una forma de reduir el nombre de parèntesis d'una expressió, ja que per a invocar a la funció, s'escriu `cAdd x y` en compte de `add(x,y)`.
- facilita l'aplicació parcial d'una funció, que, intuïtivament, consisteix en no proporcionar tots els arguments d'una funció.

En general, l'aplicació parcial suposa la següent manera de funcionament: siga una funció `f` amb dos arguments d'entrada de tipus `a` i `b` respectivament i eixida de tipus `c`:

```
f :: a -> b -> c
```

I ara, suposem que es té una expressió `exp` de tipus `a` (que també es pot escriure com `exp::a`) i també pot vore's com una funció constant de tipus `a`. Llavors, es pot escriure l'expressió "`f exp`" que serà de tipus

```
f exp :: b -> c
```

[6]Un isomorfisme (o homomorfisme bijectiu) és una relació entre estructures algebraiques que preserva l'estructura. Els còmputs realitzats en qualsevol dels dos dominis seran equivalents.

És a dir, "`f exp`" és una funció que prendrà com a argument un element de tipus `b` i tornarà com a resultat un element de tipus `c`.

Vegeu ara el següent exemple amb operadors aritmètics. Assumint el següent operator aritmètic per a la multiplicació (cal notar que un operador definit entre parèntesi indica que aquest operador es pot utilitzar en notació infixa):

```
(*) :: Int -> Int -> Int
```

Ara es pot definir diverses funcions aritmètiques, fent ús de l'orde superior.

```
squarepow :: Int -> Int
squarepow x = x * x

doubleHO :: (Int -> Int) -> Int -> Int
doubleHO f x = f (f x)

fourthpow :: Int -> Int
fourthpow = doubleHO squarepow
```

La funció `dobleHO` és d'orde superior ja que un dels seus arguments és una funció en lloc d'una dada, en concret, el paràmetre `f`. És important remarcar que la funció `fourthpow` espera un argument de tipus `Int` encara que la seua definició no incloga cap paràmetre formal, és a dir, no apareix definit com `fourthpow x`. Això es deu a l'aplicació parcial de la funció d'orde superior `dobleHO` que rep com a argument la funció `squarepow`.

L'operador " `->` " és associatiu per la dreta, és a dir, "`a -> b -> c`" equival a "`a -> (b -> c)`" i difereix de "`(a -> b) -> c`". L'operador d'aplicació funcional és associatiu per l'esquerra, és a dir, "`f a b`" equival a "`(f a) b`" i difereix de "`f (a b)`".

5 Tipus de dades simples

Hi ha una col·lecció de tipus de dades, funcions i operadors que poden usar-se en qualsevol programa. Aquests són els elements predefinits del llenguatge, que es troben, en el cas del Haskell, en el mòdul `Prelude` del sistema. Totes les funcions predefinides (a excepció de les aritmètiques) són, en un principi, prefixes, encara que, tal com es vorà, és possible usar-les en notació infixa. També és possible ometre els parèntesis dels arguments quan no hi haja confusió (per exemple, una funció com `f (a) (b)` també es pot escriure com `f a b`).

1. **El tipus `Bool`**

   Els valors d'aquest tipus representen expressions lògiques, el resultat de les quals pot ser verdader o fals. Només hi ha dos valors constants per a aquest tipus: `True` i `False` (escrits d'aquesta manera), que representen els dos resultats possibles.

   Funcions i Operadors

   Els següents operadors i funcions predefinits operen amb valors booleans:

   - `(&&) :: Bool -> Bool -> Bool`. És la conjunció lògica
   - `(||) :: Bool -> Bool -> Bool`. És la disjunció lògica
   - `not :: Bool -> Bool`. És la negació lògica
   - `(==) :: Bool -> Bool -> Bool`. Torna `True` si el primer argument és igual al segon, i `False` si són distints.
   - `(/=) :: Bool -> Bool -> Bool`. Torna `True` si el primer argument no és igual al segon argument, i `False` si són iguals.

2. **El tipus `Int`**

   Els valors d'aquest tipus són nombres enters de precisió limitada[7]. Els valors constants d'aquest tipus s'escriuen amb la notació habitual: $0, 1, -1, 2, -2, \ldots$

   Funcions i Operadors

   Algunes de les funcions i operadors definits per a aquest tipus són:

   - `(+)`, `(-)`, `(*) :: Int -> Int -> Int`. Són la suma, resta i producte de enters respectivament.
   - `(^) :: Int -> Int -> Int`. És l'operador potència. L'exponent haurà de ser un natural.
   - `div`, `mod :: Int -> Int -> Int`. Són el quocient i el residu de dividir dos enters respectivament.
   - `abs :: Int -> Int`. És el valor absolut
   - `signum :: Int -> Int`. Torna 1, -1 o 0 respectivament, segons siga el signe de l'argument enter.
   - `even`, `odd :: Int -> Bool`. Comproven la paritat (parell o senar) d'un nombre enter.
   - `(==) :: Int -> Int -> Bool`. Torna `True` si el primer argument és igual al segon, i `False` si són distints.

[7] En Haskell existeix també el tipus `Integer`, amb precisió no limitada, amb les mateixes operacions que el `Int`. Tots dos tipus són compatibles entre si.

- `(/=) :: Int -> Int -> Bool`. Torna `True` si el primer argument no és igual al segon argument, i `False` si són iguals.

3. **El tipus `Float`**

   Els valors d'aquest tipus representen nombres reals amb precisió limitada a 7 o 8 dígits decimals[8]. Hi ha dues maneres d'escriure valors reals:

   - notació habitual: per exemple `1.35`, `-1.0`, o `1`.
   - notació científica: per exemple `1.5e3` (que denota el valor 1.5×10^3).

   Funcions i operadors

   Algunes de les funcions i operadors definits per a este tipus són els següents.

   - `(+)`, `(-)`, `(*)`, `(/) :: Float -> Float -> Float`. Són la suma, resta, producte i divisió de reals respectivament.
   - `(==) :: Float -> Float -> Bool`. Torna `True` si el primer argument és igual al segon i `False` si no ho és.
   - `(/=) :: Float -> Float -> Bool`. Torna `True` si el primer argument és diferent del segon, i `False` si són iguals.
   - `sqrt :: Float -> Float`. Torna l'arrel quadrada d'un real.
   - `(^) :: Float -> Int -> Float`. Torna la potència de base real i exponent enter.
   - `(**) :: Float -> Float -> Float`. Torna la potència amb base i exponent reals.
   - `truncate :: Float -> Int`. Torna la part entera d'un real.
   - `signumFloat :: Float -> Int`. Torna `1`, `-1` o `0` segons el signe del número real.

4. **El tipus `Char`**

   Un valor de tipus `Char` representa un caràcter (lletra, dígit, ...). Un valor constant de tipus caràcter s'escriu entre cometes simples (per exemple `'a'`, `'9'`, ...). Per utilitzar aquest tipus de dades, cal incloure `Import Data.Char`, bé en el mòdul que s'estiga definint, bé en la sessió GHCi en la que es desitje actuar (comandament `:module +`). La importació de mòduls s'estudiarà en una altra pràctica més endavant.

[8] En Haskell també existeix el tipus `Double` de precisió doble. No obstant això, ambdós tipus, `Float` i `Double`, no són compatibles entre si, pel que poden ser necessàries operacions explícites de transformació entre ells.

Funcions i operadors

Algunes de les funcions definides per a este tipus són:

- `ord :: Char -> Int`. Torna el codi ASCII/Unicode corresponent al caràcter del argument.
- `chr :: Int -> Char`. És la funció inversa a `ord`.
- `isUpper, isLower, isDigit, isAlpha :: Char -> Bool`. Comproven si el caràcter del argument és una lletra majúscula, minúscula, un dígit o una lletra, respectivament.
- `toUpper, toLower :: Char -> Char`. Converteixen la lletra que prenen com a argument en majúscula o minúscula, respectivament.
- `(==) :: Char -> Char -> Bool`. Torna `True` si el primer argument és igual al segon, i `False` si són distints.
- `(/=) :: Char -> Char -> Bool`. Torna `True` si el primer argument no és igual al segon argument, i `False` si són iguals.

6 Exercicis

6.1 Exemples ja resolts

1. Escribiu una funció `nextchar` que prenga com a argument una lletra de l'alfabet i torne la lletra que el segueix.

```
module Nextchar where
 import Data.Char
 nextchar :: Char -> Char
 nextchar c = chr ((ord c) + 1)
```

I la seua execució

```
*Nextchar> nextchar 'a'
'b'
```

2. Definiu una funció `fact` per a calcular el factorial d'un nombre enter no negatiu.

```
module Factorial where

fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n - 1)
```

I la seua execució

```
*Factorial> fact 3
6
```

6.2 Exercicis a resoldre

1. Escriviu una funció `numCbetw2` que retorne quants caràcters hi ha entre dos caràcters donats (sense incloure-los). Per exemple:

```
numCbetw2 'a' 'c'
1
numCbetw2 'e' 'a'
3
numCbetw2 'a' 'b'
0
numCbetw2 'x' 'x'
0
```

2. Escriviu una funció recursiva que torne el sumatori des d'un valor enter fins a un altre (incloent ambdós).

3. Definiu una funció binària (amb dues arguments) `max'` que torne el major dels seus dos arguments.

4. Escriviu una funció `leapyear` que determine si un any és o no bixest. Un any és bixest si és múltiple de 4. No obstant això, no ho són els múltiples de 100, a excepció dels que són múltiples de 400 que sí que ho són. Per exemple, 1800 no va ser bixest mentre que l'any 2000 si que ho va ser.

5. Escriviu una funció `daysAmonth` que calcule el nombre de dies d'un mes, donats els valors numèrics de mes i any. Considereu els anys bixests per a febrer.

6. Escriviu una funció `remainder` que torne el residu de la divisió de dos nombres enters no negatius, divisor diferent de zero, usant sustraccions.

7. Usant la definició prèvia de la funció factorial, escriviu una definició de la funció `sumFacts` tal que torne la suma dels factorials fins a un número n, és a dir, `sumFacts` n `= fact 0 + fact 1 +...+ fact` n.

7 Avaluació

L'assistència a les sessions de pràctiques és obligatòria per a aprovar l'assignatura. L'avaluació d'aquesta segona part de pràctiques es realitzarà mitjançant un examen individual al laboratori.