Laboratorio de Programación Multiparadigma (LPM)

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

---

Práctica 5: El entorno de programación PAKCS/Curry

Santiago Escobar

Julia Sapiña

Índice

1. Introducción

En esta práctica vamos a describir el lenguaje multi paradigma Curry. A continuación, mostraremos una breve introducción a la sintaxis del lenguaje Curry y los ejercicios que hay que realizar.

El lenguaje de programación Curry integra características de los paradigmas de programación más importantes:

programación funcional: expresiones con funciones anidadas, evaluación perezosa, funciones de orden superior;

programación lógica: variables lógicas, estructuras de datos parciales, facilidades de búsqueda;

programación concurrente: evaluación de restricciones concurrente con sincronización sobre las variables lógicas.

El mecanismo de ejecución de los programas Curry combina los principios operacionales de *narrowing* y *residuación* (ver la sección 3).

El desarrollo del lenguaje Curry parte de una iniciativa internacional cuyo objetivo consiste en diseñar una plataforma común para la investigación, docencia y aplicación de los lenguajes declarativos. Podéis encontrar una información más detallada en:

`http://curry-language.org`

A continuación resumimos las principales características del lenguaje Curry. El documento que describe con detalle el lenguaje Curry está disponible en dicha web bajo el título de "Curry Report".

2. Sintaxis de un programa en Curry

2.1. Declaraciones de Tipos

Una declaración de tipo tiene la forma

$$\texttt{data T}\ \alpha_1\ \ldots\ \alpha_n = \texttt{C}_1\ \tau_{11}\ \ldots\ \tau_{1n_1} \mid \ldots \mid \texttt{C}_k\ \tau_{k1}\ \ldots\ \tau_{kn_k}$$

e introduce un nuevo tipo `T` de aridad n, cuyos constructores son $\texttt{C}_1, \ldots, \texttt{C}_k$ y cada uno de los τ_{ij} es una *expresión de tipo* construida a partir de las *variables de tipo* y algunos constructores. Curry dispone de algunos tipos predefinidos, como `Bool`, `Int`, listas, tuplas, etc. Observad que los tipos se escriben en notación *currificada* (ver el apartado 2.2), ya que Curry es un lenguaje de orden superior.

Por ejemplo, las declaraciones de tipo:

```
data Bool = True | False
data List a = [] | a : List a
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

introducen el tipo `Bool` con dos constantes `True` y `False`, y los tipos polimórficos `List a` (listas cuyos elementos son de tipo `a`) y `Tree a` (árboles cuyos elementos son de tipo `a`). Las listas se expresan usando el operador binario infijo ":", por ejemplo, `0:(1:(2:[]))` es una lista (aunque Curry permite también la notación Prolog: `[0,1,2]`).

2.2. Declaración de Funciones

Las funciones se definen mediante una declaración de tipo, seguida de una lista de ecuaciones. La declaración de tipo (para una función de aridad n) tiene la forma:

$$\texttt{f} :: \tau_1 -> \tau_2 -> \ldots -> \tau_n -> \tau$$

donde $\tau_1, \ldots, \tau_n, \tau$ son tipos. Declarar el tipo de las funciones no es obligatorio en Curry (dispone de un sistema de inferencia de tipos automático), aunque resulta muy aconsejable. En particular, puede servir para detectar fácilmente errores en el programa, cuando se definen funciones que no se ajustan al tipo declarado. Observad que escribimos el tipo en notación currificada, es decir, escribiremos

```
suma :: Int -> Int -> Int
```

en lugar de

```
suma :: Int x Int -> Int
```

Esta notación es habitual cuando se trabaja con lenguajes de orden superior. De la misma forma, las expresiones del programa se deben escribir en notación currificada. Por ejemplo, la expresión `suma(6,producto(3,5))` se escribe como `suma 6 (producto 3 5)`. En particular, los argumentos de una función se escriben a continuación del nombre de la función separados por un espacio en blanco (agrupándose entre paréntesis aquellos argumentos que sean a su vez funciones con argumentos). Respecto a las ecuaciones que definen las funciones, éstas pueden ser de la forma

$$\texttt{f t}_1 \ \ldots \ \texttt{t}_n = \texttt{e}$$

donde $\texttt{t}_1, \ldots, \texttt{t}_n$ son términos constructores (es decir, sin símbolos de función definidos en el programa) y `e` es una expresión arbitraria. La parte izquierda de la ecuación $\texttt{f t}_1 \ \ldots \ \texttt{t}_n$ no puede contener ocurrencias repetidas de una misma variable. Las funciones se pueden definir también mediante ecuaciones condicionales

$$\texttt{f t}_1 \ \ldots \ \texttt{t}_n \mid \texttt{c} \ = \texttt{e}$$

donde la condición `c` es una restricción (es decir, de tipo `Constraint`). Las ecuaciones condicionales sólo se pueden aplicar si la condición tiene éxito. Si aparecen varias ecuaciones condicionales con la misma parte izquierda, podemos abreviarlas así:

$$\begin{array}{ll} \texttt{f t}_1 \ \ldots \ \texttt{t}_n & \mid \texttt{c}_1 \ = \texttt{e}_1 \\ & \vdots \\ & \mid \texttt{c}_k \ = \texttt{e}_k \end{array}$$

Su significado será exactamente el mismo que

$$\begin{array}{ll} \texttt{f t}_1 \ \ldots \ \texttt{t}_n & \mid \texttt{c}_1 \ = \texttt{e}_1 \\ \vdots & \vdots \\ \texttt{f t}_1 \ \ldots \ \texttt{t}_n & \mid \texttt{c}_k \ = \texttt{e}_k \end{array}$$

También es posible definir una ecuación condicional en la que las guardas tengan tipo `Bool` (en lugar de `Constraint`). Por ejemplo, podemos escribir

```
fac n | n==0       = 1
      | otherwise  = fac (n-1)*n
```

donde la igualdad '==' se evalúa a `True` si ambas expresiones son iguales y constructoras (sin permitir la instanciación de variables) y la constante predefinida `otherwise` tiene el valor `True`. En este caso, se puede usar también la instrucción condicional `if_then_else`:

```
fac n = if n==0 then 1 else fac (n-1)*n
```

Sólo existe una restriccion con respecto a las ecuaciones condicionales: todas las guardas deben ser de tipo `Constraint` o bien de tipo `Bool`, pero no se pueden mezclar.

2.3. Declaraciones Locales

Existen dos formas de declarar funciones auxiliares que no son visibles globalmente en el programa. La primera, consiste en utilizar la construcción `let` *decls* `in` *exp*. La lista de declaraciones *decls* puede definir funciones o constantes. Los valores introducidos en *decls* sólo son visibles en las partes derechas de las propias declaraciones y en la expresión *exp*. Por ejemplo, la expresión

```
let a=3*b
    b=6
in 4*a
```

se reduce al valor `72`. Cuando lo que pretendemos es definir una función auxiliar, resulta más habitual emplear la cláusula `where`. Por ejemplo, la siguiente función

```
exp b n = if n==0
             then 1
             else if even n then square (exp b (n `div` 2))
                            else b * (exp b (n-1))
          where even n = n `mod` 2 == 0
                square n = n*n
```

implementa el exponente de un número (b^n), de forma que las funciones `even` y `square` sólo son visibles en el cuerpo de la regla anterior, pero no en el resto del programa. Hay que tener cuidado porque el espaciado en Curry es muy importante, como se muestra en el ejemplo anterior. Es decir, la posición en la que empieza una nueva línea de texto en un programa es determinante para saber si es continuación de una definición anterior o no.

2.3.1. Variables Libres

Al igual que en Prolog, es posible introducir variables libres en el cuerpo de una función. Por ejemplo, dada la siguiente definición para la función Booleana `mother`:

```
mother John   = Christine
mother Alice  = Christine
mother Andrew = Alice
```

podríamos averiguar los hijos de `Alice` mediante la ecuación

```
mother x =:= Alice
```

donde `x` es una variable libre (el significado de '`=:=`' lo veremos en la siguiente sección). La única restricción en Curry es que, a diferencia de Prolog, las variables libres deben *declararse* usando la notación `x free` (dentro de una construcción `let` o `where`). Por ejemplo, podemos escribir

```
isgrandmother x | mother (mother y) =:= x  = True where y free
```

para definir la función `isgrandmother`, la cual comprueba si su argumento es abuela de alguien. Veamos otro ejemplo. Dada la definición de `append`:

```
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

podríamos definir la función `last`, que computa el último elemento de una lista, como sigue

```
last l | append xs [x] =:= l  = x
         where x,xs free
```

2.4. Restricciones e Igualdad

Como hemos visto, las condiciones en las ecuaciones condicionales pueden ser *restricciones*. Las restricciones ecuacionales tiene la forma `e1 =:= e2` y tienen éxito si ambas expresiones pueden reducirse a dos expresiones *constructoras* (es decir, en las que no aparecen símbolos de función definidos en el programa) que unifiquen. Por ejemplo, la restricción `append x [1] =:= [y]` tiene éxito ya que puede reducirse a `[1] =:= [y]` y ambas expresiones unifican y son constructoras.

Observad que una restricción no es equivalente a una expresión Booleana. Concretamente, en el caso de las restricciones, sólo evaluamos su *satisfacibilidad*, es decir, el caso en el que se reducen a `True`. Las expresiones Booleanas, por el contrario, pueden reducirse a `True` o `False`. Así, por ejemplo, si interpretásemos `[1] =:= [y]` como una expresión Booleana, existirían infinitas soluciones para el caso en que su evaluación devuelve el valor `False`.

Las restricciones en Curry pueden ejecutarse concurrentemente mediante el uso de la conjunción: `c1 & c2`. Para resolver `c1 & c2`, se intenta resolver primero `c1`; si ésta *suspende* (debido a las anotaciones `rigid`, ver la sección 3), entonces pasamos a resolver `c2` hasta que sea posible seguir con la ejecución de `c1` (debido a la instanciación de variables provocada por la ejecución de `c2`).

3. Semántica Operacional del lenguaje Curry

La semántica operacional de Curry es una combinación de los mecanismos de *narrowing* y *residuación*. Las *expresiones de respuesta* se denotan por σ [] `e`, donde σ es la sustitución computada hasta el momento y `e` es la expresión a reducir. Decimos que una expresión de respuesta σ [] `e` está "resuelta" si e es una expresión constructora. Los objetivos se denotan mediante una *disyunción* de expresiones de respuesta, de la forma:

$$\{\texttt{x} = 0,\ \texttt{y} = 2\}\ []\ 2\ \vee\ \{\texttt{x} = 1,\ \texttt{y} = 2\}\ []\ 3$$

o, simplemente:

```
{x=0, y=2} 2 | {x=1, y=2} 3
```

Por ejemplo, dada la función

```
f 0 = 2
f 1 = 3
```

el resultado de evaluar `f 1` es `3`, mientras que la evaluación de `f x` devuelve la expresión disyuntiva:

```
{x=0} 2 | {x=1} 3
```

Cuando se produce una llamada a función cuyos argumentos no estan lo suficientemente instanciados, podemos proceder de dos formas:

- instanciando las variables de forma que la llamada a función se pueda resolver usando las distintas reglas que definen la función (es decir, empleando *narrowing*), lo que dara lugar a una nueva disyunción (como en el caso de `f x`);
- o bien *suspendiendo* la expresión hasta que sus variables esten lo suficientemente instanciadas (es decir, usando *residuación*).

La forma de proceder vendrá determinada por las *anotaciones de evaluación* asociadas a las funciones. Así, una función puede ser anotada como `rigid` (y se resolverán sus llamadas mediante residuación) o como `flex` (y se empleará narrowing). Si una función no aparece anotada, se considerará `rigid` por defecto.

Por ejemplo, consideremos de nuevo la definición de `f`, en este caso anotada como `flex`:

```
f eval flex
f 0 = 2
f 1 = 3
```

Ahora, una llamada como `f x` procede a instanciar la variable `x` a `0` y `1` de forma que se puedan aplicar ambas reglas, lo que produce la solución:

```
{x=0} 2 | {x=1} 3
```

Sin embargo, si declaramos la función `rigid`:

```
f eval rigid
f 0 = 2
f 1 = 3
```

entonces una llamada como `f x` no puede resolverse, es decir, el objetivo *suspende*. Para poder resolver la llamada, sería necesaria una expresión como, por ejemplo

```
f x =:= y  &  x =:= 1
```

la cual se resuelve como sigue

```
f x =:= y  &  x =:= 1
--> {x=1} f 1 =:= y
--> {x=1} 3 =:= y
--> {x=1, y=3}
```

4. Tipos de datos disponibles en Curry

Los siguientes tipos están predefinidos en Curry:

Booleanos: Están predefinidos mediante la declaración

```
data Bool = True | False
```

La conjunción secuencial `&&` (es decir, la conjunción *lógica*, pero no concurrente) se define así:

```
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && x = False
```

También disponemos de la disyunción (||) y la negación (`not`). Asimismo, la constante `otherwise` está predefinida al valor `True`.

Las expresiones Booleanas se emplean principalmente en las condicionales, que se definen como sigue

```
if_then_else :: Bool -> a -> a -> a
if True  then x else y = x
if False then x else y = y
```

Las funciones cuyo resultado es de tipo `Bool` suelen llamarse simplemente *predicados*. Respecto a la igualdad `==`, ésta se define de forma similar a `=:=`, es decir, ambas expresiones deben reducirse a términos constructores, pero no permite la instanciacion de variables (para evitar evaluaciones infinitas).

Restricciones: Las restricciones que aparecen en las ecuaciones condicionales tienen asignado el tipo `Constraint`. Asimismo, una función cuyo resultado sea de tipo `Constraint` se suele llamar una *restricción*. La restricción trivial es la palabra `success`, que siempre se satisface. Las restricciones se pueden combinar mediante el operador de conjunción concurrente: `c1 & c2 & ... & cn`. Además, es posible resolverlas de forma secuencial (de izquierda a derecha) usando el operador `&>`. Así, una restricción `c1 &> c2` procederá a resolver completamente `c1` y luego `c2`. Si `c1` suspende, entonces la expresión completa suspende.

Enteros: El tipo `Int` se emplea para denotar los números enteros. La operadores habituales, como $*$, $+$, etc., están predefinidos y exigen que sus argumentos estén completamente instanciados (es decir, son `rigid`).

Listas: El tipo `[t]` denota el dominio de las listas cuyos elementos son de tipo `t`. El tipo lista puede ser considerado como predefinido por la declaración

```
data [a] = [] | a : [a]
```

donde `[]` denota la lista vacía y `x : xs` es una lista no vacía cuyo primer elemento es `x` y cuyo resto es `xs`. La notación Prolog `[e1,e2,...,en]` también está permitida y es equivalente a `e1:e2:...:en:[]`.

Como operadores predefinidos tenemos, entre otros, la concatenación de listas (`++`) y la aplicación de una función a todos los elementos de una lista (`map`), cuyas definiciones son

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys

map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Caracteres: Valores como 'a' ó '0' denotan constantes del tipo `Char`. Existen dos funciones de conversión predefinidas entre caracteres y sus valores ASCII:

```
ord :: Char -> Int
chr :: Int -> Char
```

Cadenas: El tipo `String` es simplemente una abreviatura para [`Char`], es decir, las cadenas de caracteres son simplemente *listas* de caracteres. Al igual que en Prolog, la cadena `"Hola"` se puede escribir también como `['H','o','l','a']`. Un término se puede convertir en una cadena mediante la función

```
show :: a -> String
```

Por ejemplo, el resultado de `show`(`42`) es la lista de caracteres `['4','2']`.

4.1. El Sistema de Inferencia de Tipos

El lenguaje Curry incorpora un sistema automático de inferencia de tipos. Así, no es necesario declarar explícitamente el tipo de todas las funciones del programa. A menudo, es suficiente con declarar los tipos de datos (es decir, las declaraciones `data`) que vamos a usar, aunque si no es posible inferir el tipo de alguna de las funciones del programa, se producirá un error durante la compilación, y será necesario escribir su declaración de tipo de forma explícita.

5. Implementaciones de Curry

Existen varias implementaciones del lenguaje Curry:

- PAKCS, disponible en `http://www.informatik.uni-kiel.de/~pakcs`.

  La herramienta "Portland Aachen Kiel Curry System (PAKCS)" es una implementación de Curry desarrollada conjuntamente por la "Portland State University" de EE.UU. y la "Christian-Albrechts-Universität zu Kiel" de Alemania. PAKCS proporciona una interfaz interactiva para cargar programas y ejecutar expresiones.

- MCC, disponible en `http://danae.uni-muenster.de/curry/`.

  La herramienta "Münster Curry Compiler (MCC)" es un compilador de Curry para sistemas UNIX, que codifica un programa Curry al lenguaje C y luego utiliza el compilador GNU C.

- KiCS2, disponible en
  `http://www-ps.informatik.uni-kiel.de/kics2/`

  La herramienta "Kiel Curry System (KiCS)" es un compilador de Curry a Haskell desarrollado por la "Christian-Albrechts-Universität zu Kiel" de Alemania.

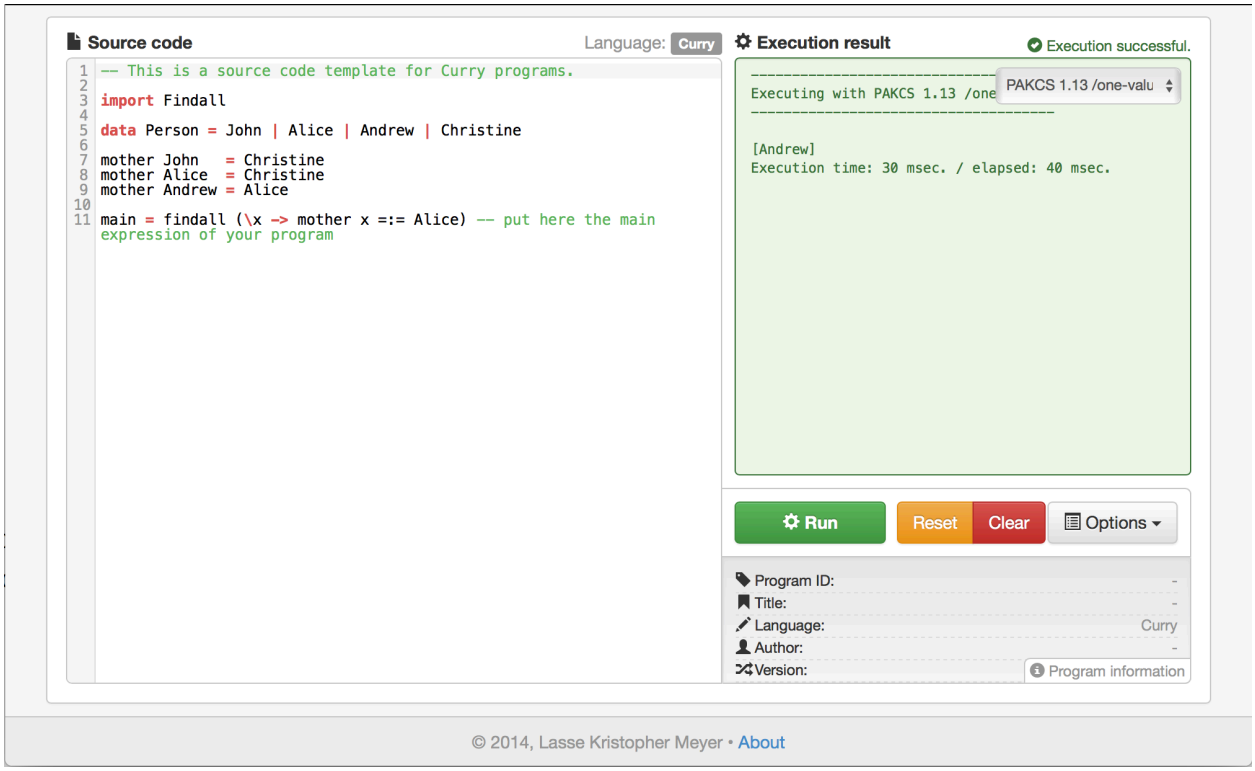
- UPV-Curry, disponible en
  `http://www.dsic.upv.es/grupos/elp/upv-curry/upv-curry.html`.

  Desarrollamos una versión de Curry en el grupo de métodos formales en los años 2000. Esta herramienta estaba desarrollada en Prolog pero actualmente no se mantiene y ya no funciona correctamente.

6. Entorno de Trabajo: el sistema PAKCS disponible online

Para la realización de estas prácticas nos vamos a restringir a la implementación PAKCS y el entorno de programación Smap creado para PAKCS, disponible en `http://www-ps.informatik.uni-kiel.de/smap/smap.cgi`.

Por ejemplo, si introducimos el programa anterior sobre la relación madre, y pulsamos en el botón "Run", obtendremos el resultado de ejecutar la expresión quién es el hijo de "Alice", véase el resultado en la siguiente captura.



Hay que prestar atención al uso de la librería `Findall` y la función `findall`, que nos permite realizar una búsqueda encapsulada de *todas* las soluciones a una consulta. En el caso de que haya un número infinito de soluciones o que la evaluación de una expresión no termine, es mejor utilizar la función `findfirst`, que devuelve la *primera* solución a una consulta.

7. Objetivo de la práctica

El objetivo de esta práctica consiste en codificar los ejercicios adjuntos en Curry y responder a las preguntas.

7.1. Monty Python Example

Dado el famoso ejemplo de los Monty Python sobre las brujas, disponible en `https://youtu.be/yp_l5ntikaU`. El razonamiento lógico que utilizan puede describirse en programación lógica de la siguiente manera:

```
bruja(X) :- arde(X), mujer(X) .
```

```
arde(X) :- madera(X) .
madera(X) :- flota(X) .
flota(X) :- mismopeso(pato, X) .
mujer(lola) .
mismopeso(pato,jamon) .
```

Una forma de codificarlo en Curry es la siguiente:

```
import Findall

data A = pato | lola | jamon

bruja X = arde X & mujer X
arde X = madera X
madera X = flota X
flota X = mismopeso pato X
mujer lola = success
mismopeso pato jamon = success
```

Y las preguntas que hay que responder son

1. ¿Qué devuelve la ejecución de la expresión `findall (\x -> bruja x)`?
2. ¿Qué devuelve la ejecución de la expresión `bruja lola =:= success`?
3. ¿Qué devuelve la ejecución de la expresión `bruja lola =:= success` si añadimos una ecuación de `mismopeso` para que compare `pato` y `lola`?

7.2. Listas: longitud

Dada la siguiente definición de la función `length` para calcular la longitud de una lista,

```
length [] = 0
length (x:xs) = 1 + length xs
```

responde a las siguientes preguntas:

1. ¿Qué devuelve la ejecución de la expresión `length []`?
2. ¿Qué devuelve la ejecución de la expresión `length [1,2,3,4]`?
3. ¿Qué devuelve la ejecución de la expresión `findfirst (\xs -> length xs =:= 0)`?
4. ¿Qué devuelve la ejecución de la expresión `findfirst (\xs -> length xs =:= 10)`?

7.3. Listas: eliminar un elemento

¿Cómo especificarías en Curry una función que elimina un elemento de una lista?

1. ¿Qué devuelve la ejecución de la expresión `remove 1 []`?
2. ¿Qué devuelve la ejecución de la expresión `remove 1 [1,2,3,4]`?
3. ¿Qué devuelve la ejecución de la expresión
   `findall (\(x,y) -> remove x [1,2,3,4] =:= y)`?

7.4. Suma y comparación

Dada la siguiente definición de la función `sum` para calcular la suma de dos números naturales y la función `leq` para compara dos números naturales,

```
import Findall

data Nat = z | s Nat

sum z y = y
sum (s x) y = s (sum x y)

leq z n = True
leq (s m) (s n) = leq m n

all1 = findall (\(a,b) -> (leq (sum a b) (s z)) =:= True)
all2 = findall (\b -> (leq (sum z z) (s b)) =:= True)
all3 = findall (\(a,b) -> (leq (sum a b) (s a)) =:= True)
```

responde a las siguientes preguntas:

1. ¿Qué devuelve la ejecución de la expresión `all1`?
2. ¿Qué devuelve la ejecución de la expresión `all2`?
3. ¿Qué devuelve la ejecución de la expresión `all3`?
4. ¿Y si cambiamos el `findall` de la expresión `all3` por un `findfirst`?

7.5. Resta

¿Cómo especificarías en Curry una función que resta dos números naturales?

1. ¿Qué devuelve la ejecución de la expresión `substract (s z) (s s z)`?
2. ¿Qué devuelve la ejecución de la expresión `substract (s s z) (s z)`?
3. ¿Qué devuelve la ejecución de la expresión
   `findall (\(x,y) -> substract x (s s z) =:= y)`?