Apunte Laboratorio ALPI [*]

Pablo Speciale

[*] Análisis de Lenguaje de Programación I - Universidad Nacional de Rosario

Índice

1. Conceptos Importantes

En esta sección se explica que es un lenguaje funcional puro. Además, se dan algunas definiciones que son la esencia de Haskell.

1.1. Valores

Entidades matemáticas abstractas con ciertas propiedades. Una expresión denota un valor. Entre las clases de valores que una expresión puede denotar, se incluyen: números de varios tipos (Int, Integer, Float, etc.), caracteres, funciones y listas. Observar que, en Haskell, las funciones también son valores.

1.2. Expresiones

Cadenas de símbolos utilizados para denotar valores

- Expresiones atómicas: llamadas también *formas normales*. Por abuso de notación, les decimos valores.
- Expresiones compuestas: se "arman" combinado subexpresiones. Por abuso de notación, sólo les decimos expresiones.

Notas

1. No confundir entre valores y su representación (mediante expresiones). Pueden existir muchas formas de representar un mismo valor. Por ejemplo, 5 ó 101 (en binario) ó V (en números Romanos).
2. Una expresión está "bien formada" si cumple con las reglas sintácticas y reglas de asignación de tipo.
3. Algunos valores no tienen representación canónica, por ejemplo, los valores funcionales.

1.3. Funciones

- Visión denotacional: una función es un valor matemático que relaciona cada elemento de un conjunto (de partida) con un único elemento de otro conjunto (de llegada).
- Visión operacional: una función es un mecanismo que dado un elemento del conjunto de partida, calcula el elemento correspondiente del conjunto de llegada.

Extensionalidad

Este principio recibe el nombre de *principio de extensionalidad*.

$$f = g \qquad \Leftrightarrow \qquad f\ x = g\ x$$

1.4. Ecuaciones orientadas

$$\begin{aligned} Expresion - a - definir &= Expresion - definida \\ e_1 &= e_2 \end{aligned}$$

- Visión denotacional: se define que el valor denotado por e_1 es el mismo que el denotado por e_2.
- Visión operacional: para calcular el valor de una expresión que contiene a e_1, se puede reemplazar e_1 por e_2.

Nota

Dada una expresión bien formada, determinamos el valor que denota mediante ecuaciones. Y calculamos el valor de la misma, reemplazando subexpresiones, de acuerdo con las reglas dadas por las ecuaciones, a ésto se lo llama reducción (ver sección 6).

1.5. Algunas definiciones

Función de alto orden: una función que recibe otra función como argumento, o la retorna como resultado.

Transparencia Referencial: el valor de una expresión depende sólo de los elementos que la constituyen. Una definición más informal, cuando dos cosas son iguales que sean iguales en todas partes. Por ejemplo, en C, uno puede hacer lo siguiente:

```
x = x + 1;
```

Se le asigna a x el valor de $x+1$. Este ejemplo muestra un lenguaje que no posee la importante propiedad de "Transparencia Referencial", pues x no vale lo mismo antes y después de la asignación. Un lenguaje así se dice que posee "Efectos colaterales".
La Transparencia Referencial implica:

- Abstracción de detalles de ejecución.
- Posibilidad de demostrar propiedades usando las propiedades de las subexpresiones y métodos de deducción lógicos.

Lenguaje funcional puro: lenguaje de expresiones con transparencia referencial y funciones de alto orden, cuyo modelo de cómputo es la reducción realizada mediante el reemplazo de igualdad por igualdad. **Haskell** es un lenguaje funcional puro.

1.6. Tipos

Conjunto de valores con propiedades comunes. Los tipos denotan conjuntos de valores.

Tipado Fuerte (strong typing): toda expresión debe tener un tipo para ser válida.

Notación: $e :: A$ se lee "la expresión e tiene tipo A".

Significa el valor denotado por e pertenece al conjunto de valores denotado por A.

Inferencia de tipos:

Dada un expresión e, determinar si tiene tipo o no según las reglas, y cuál es ese tipo [1].

Algunas reglas de inferencia de tipo:

$$\begin{array}{lrcl} \text{si} & e_1 :: A \; y \; e_2 :: B & \Rightarrow & (e_1, e_2) :: (A, B) \\ \text{si} & m, n :: Int & \Rightarrow & m + n :: Int \\ \text{si} & d = e \; y \; e :: A & \Rightarrow & d :: A \end{array}$$

Chequeo de tipos: $(e, A) \rightarrow Bool$

Dada una expresión e y un tipo A, determinar si $e :: A$ según las reglas.

[1] Devuelve el tipo más general, es decir, el tipo con más variables de tipos

2. Sintaxis de Haskell

En esta sección se muestra lo esencial de la sintaxis de Haskell.

2.1. Case sensitive

La sintaxis de Haskell diferencia entre mayúsculas y minúsculas. Se dice que es *Case sensitive*.

2.2. Comentarios

Los comentarios empiezan con "`--`" (dos menos). Por ejemplo:

```
-- Este es un comentario, cuando empieza con "--"
-- Haskell (más precisamente Hugs o ghc) lo ignora
```

2.3. Definiciones de Variables

La definición de una *variable* tendrá la forma

```
nombre_variable :: Tipo
nombre_variable = expresion
```

como en el ejemplo

```
size :: Integer
size = 12 + 13
```

2.4. Funciones

Existen dos formas de nombrar una función, mediante un identificador (por ejemplo, `sum`, `product` y `fact`), o mediante un símbolo de operador (por ejemplo, $\star$ y +)

2.4.1. Identificadores

Los identificadores de variables o funciones deben comenzar con una letra minúscula, seguido, opcionalmente por una secuencia de caracteres, cada uno de los cuales es: una letra, un dígito, un apóstrofe (`'`) o un guión bajo (_). Los siguientes son ejemplos posibles de identificadores:

```
sum     f     f''     fintSum     nombre_con_guiones
```

Los siguientes identificadores son palabras reservadas y no pueden utilizarse como nombres de funciones o variables:

```
case   of     where    let    in      if
then   else   data     type   infix   infixl
infixr class  instance primitive
```

2.4.2. Operadores

Algunas funciones se escriben entre sus (dos) argumentos en lugar de precederlos. A una función escrita usando la notación infija se le llama un *operador*. Por ejemplo,

```
3 <= 4            en lugar de            menorIgual 3 4
```

Un símbolo de operador es escrito utilizando uno o más de los siguientes caracteres:

```
:   !   #   $    %  &    *   +   .  /   <   =
>   ?   @    \   ^   |   -
```

Leer sección 1.4.3 del libro.

2.4.3. Relación entre identificadores y operadores

Se proporcionan dos mecanismos simples para utilizar un identificador como un símbolo de operador o un símbolo de operador como un identificador:

- Cualquier identificador será tratado como un símbolo de operador si está encerrado entre comillas inversas (`` ` ``). Por ejemplo,

```
x `mod` y        es equivalente a        mod x y
```

- Cualquier símbolo de operador puede ser tratado como un identificador encerrándolo en paréntesis. Por ejemplo,

```
x + y          es equivalente a        (+) x y
```

2.4.4. Definiciones de Funciones con Pattern Matching

La declaración de una función f está formada por un conjunto de ecuaciones con el formato:

```
f <pat1> <pat2> . . . <patn> = <expresion>
```

Donde cada una de las expresiones $< pat_1 >< pat_2 > ... < pat_n >$ representa un *argumento*[2] de la función y es denominado un patrón. El número n de argumentos se denomina *aridad*. Si f fuese definida por más de una ecuación, cada una debe tener la misma aridad.

Si f fuese un operador sería:

```
<pat1> f <pat2> = <expresion>
```

Por ejemplo,

```
minimo       :: (Integer, Integer) -> Integer
minimo (x,y) =  if  x <= y  then  x  else  y
```

La condición "`x <= y`" se evalúa a un valor de tipo `Bool`, es decir, `True` o `False`.

[2]No es del todo cierto que una función tome n argumentos, como se verá en la sección *Currificación*.

2.4.5. Definiciones con Guardas

Otro modo de expresar esencialmente la misma definición de `minimo` es escribir:

```
minimo       :: (Integer, Integer) -> Integer
minimo (x,y) | x <= y    = x
             | otherwise = y
```

Esta forma de definición utiliza *ecuaciones con guardas*. Cada cláusula consiste en una condición, o *guarda*, y en una expresión separada de la guarda por un signo de igualdad (=). En general una ecuación con guardas toma la forma:

```
f x1 x2 ... xn | condicion1 = e1
               | condicion2 = e2
               .
               .
               .
               | condicionm = em
```

Leer sección 1.5 del libro.

2.4.6. Definiciones recursivas

Las definiciones pueden ser también *recursivas*. He aquí un ejemplo:

```
fact   :: Integer -> Integer
fact n =  if  n==0  then  1  else  n * fact(n-1)
```

Otro modo de escribir la misma función pero con *ajuste de patrones (**Pattern Matching**)*.

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

Observar que se usó `0` y `n+1` para que los patrones sean disjuntos. Al a ser los patrones disjuntos, el orden en que aparecen las ecuaciones no importa; en caso contrario, sí es importante el orden.

Leer sección 1.5.1 del libro.

2.4.7. Definiciones locales

Considérese la siguiente función que calcula el número de raíces diferentes de una ecuación cuadrática de la forma $a\,x^2 + b\,x + c = 0$

```
numeroDeRaices :: Int -> Int -> Int -> Int
numeroDeRaices a b c
```

```
    | discr >  0     = 2
    | discr == 0     = 1
    | discr <  0     = 0
    where discr = b*b - 4*a*c
```

Las definiciones locales pueden también ser introducidas por *let* de la forma:

```
let <decls> in <expr>
```

Leer sección 1.5.2 del libro.

2.4.8. Expresiones case

Una expresión case puede ser utilizada para evaluar una expresión y, dependiendo del resultado, devolver uno de los posibles valores.

```
paridad :: Int -> String
paridad x = case (x `mod` 2) of
            0 -> "par"
            1 -> "impar"
```

2.4.9. Aplicación de Funciones

Notar que en la aplicación de una función, los argumentos no necesitan ser encerrados entre paréntesis (cómo usualmente se hace en matemáticas). Supóngase la siguientes

```
promedio :: Float -> Float -> Float
promedio a b = (a+b)/2
```

Por ejemplo, si queremos calcular el promedio entre 5 y 47, podemos hacer en hugs (ver sección 3):

```
Hugs> promedio 5 47
26.0
```

Los paréntesis van a ser necesarios cuando queramos "romper" el orden de precedencia estándar de los operadores. Como la aplicación de función tiene precedencia más alta que todos los demás operadores, si queremos calcular el promedio ente 6 y 9+9, tenemos que hacer

```
Hugs> promedio 6 (9+9)
12.0
```

Pues,

```
promedio 6 9 + 9        equivale a        (promedio 6 9) + 9
```

2.5. Secciones

Se puede encerrar también un argumento junto con el operador.

$$\begin{aligned}(x \oplus)\, y &= x \oplus y \\ (\oplus\, y)\, x &= x \oplus y\end{aligned}$$

Una excepción: $(-x)$ se interpreta como la aplicación del operador unario de negación.
Leer sección 1.4.4 del libro.

2.6. Tabla de precedencia/asociatividad de operadores

```
infixl     9     !!
infixr     9     .
infixr     8     ^
infixl     7     *
infix      7     /,      `div`,      `rem`,      `mod`
infixl     6     +,      -
infix      5     \\
infixr     5     ++,      :
infix      4     ==,      /=,      <,      <=,      >=,      >
infix      4     `elem`,      `notElem`
infixr     3     &&
infixr     2     ||
```

Observaciones

- La aplicación de funciones tiene la mayor precedencia.
- $A \to B \to C$ significa $A \to (B \to C)$

Leer sección 1.4.5 y 1.4.6 del libro.

2.7. Disposición del código

Cómo es posible evitar la utilización de separadores que marquen el final de una ecuación, una declaración, etc. Por ejemplo, dada la siguiente expresión:

```
ejemplo x y z = a + b
      where a = f x y
            b = g z
```

Cómo sabe el sistema Haskell que no debe analizarla como:

```
ejemplo x y z   = a + b
      where a   = f x
            y b = g z
```

La respuesta es que el Haskell utiliza una sintaxis bidimensional denominada espaciado (layout) que se basa esencialmente en que las declaraciones están alineadas por columnas. En el ejemplo anterior, obsérvese que a y b comenzaban en la misma columna. Las reglas del espaciado son bastante intuitivas y podrían resumirse en:

1. El siguiente carácter de cualquiera de las palabras clave `where`, `let` u `of` es el que determina la columna de comienzo de declaraciones en las expresiones `where`, `let` o `case` correspondientes. Por tanto podemos comenzar las declaraciones en la misma línea que la palabra clave, en la siguiente o siguientes.

2. Es necesario asegurarse que la columna de comienzo dentro de una declaración está más a la derecha que la columna de comienzo de la siguiente cláusula. En caso contrario, habría ambigüedad, ya que el final de una declaración ocurre cuando se encuentra algo a la izquierda de la columna de comienzo.

El espaciado es una forma sencilla de agrupamiento que puede resultar bastante útil. Por ejemplo, la declaración anterior sería equivalente a:

```
ejemplo x y z = a + b
    where { a = f x y ;
            b = g z }
```

3. Cosas sobre el Hugs

Existen varias implementaciones de Haskell; nosotros usaremos Hugs, ya que es libre, está disponibles para distintas plataformas (MS-Windows, Mac OS X, Unix), presenta una interfaz de usuario flexible y es bastante eficiente.

3.1. Prelude.hs

El *prelude* es un fichero de nombre `Prelude.hs` que es cargado automáticamente al arrancar Hugs y contiene la definición de un conjunto de funciones que podemos usar cuando las necesitemos. Algunos ejemplos: `div`, `mod`, `sqrt`, `id`, `fst`, `snd`.

3.2. Iniciando Hugs

Al iniciar una sesión en Hugs, nos aparecerá una ventana como ésta:

```
__   __ __  __  ____   ___     ________________________________________________
||   || ||  || ||  || ||__     Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||    Copyright (c) 1994-2005
||---||         ___||          World Wide Web: http://haskell.org/hugs
||   ||                        Report bugs to: hugs-bugs@haskell.org
||   || Version: May 2006      ________________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs>
```

Hugs evaluará cualquier expresión, sintácticamente correcta, que se ingrese en el *prompt*

```
Hugs>  <expresion>
<resultado>
```

Por ejemplo, si ingresemos $(7 + 4) * 3$, la evaluará y nos devolverá 33

```
Hugs> (7+4)*3
33
```

Si ingresamos $sqrt\ 2$, nos devolverá una aproximación de $\sqrt{2}$

```
Hugs> sqrt 2
1.4142135623731
```

3.3. Comandos de Hugs

Si uno escribe `:?` y presiona `ENTER`:

```
LIST OF COMMANDS:  Any command may be abbreviated to :c where
c is the first character in the full name.

:load <filenames>   load modules from specified files
:load               clear all files except prelude
:also <filenames>   read additional modules
:reload             repeat last load command
:project <filename> use project file
:edit <filename>    edit file
:edit               edit last module
:module <module>    set module for evaluating expressions
<expr>              evaluate expression
:type <expr>        print type of expression
:?                  display this list of commands
:set <options>      set command line options
:set                help on command line options
:names [pat]        list names currently in scope
:info <names>       describe named objects
:browse <modules>   browse names defined in <modules>
:find <name>        edit module containing definition of name
:!command           shell escape
:cd dir             change directory
:gc                 force garbage collection
:version            print Hugs version
:quit               exit Hugs interpreter
```

La primer línea dice que se puede usar como abreviatura sólo la primera letra de cada comando. Observar estos comandos equivalentes:

```
Hugs> :type id
id :: a -> a
Hugs> :t id
id :: a -> a
Hugs>
```

El comando `:t` imprime el tipo de una expresión.

3.4. Un programa en Haskell

Por ahora hemos evaluado expresiones en el prompt de Hugs y escrito algunas definiciones. Pero, ¿dónde "guardamos" esas definiciones? ¿cómo hacemos para usarlas? En esta sección responderemos todas estas cuestiones.

A una lista de definiciones se la denomina *script*. Los pasos a seguir para resolver un problema serían los siguientes:

1. Escribimos un script con la definición de todas las funciones que necesitemos para resolver el problema. Para ello podemos utilizar cualquier editor de texto, aunque es recomendable usar alguno que resalte automáticamente la sintaxis de Haskell. Guardamos el archivo con una extensión *.hs*.

2. Cargamos el script en Hugs. Para ello utilizamos el comando *:load* seguido del nombre del archivo. Ahora en el prompt de Hugs ya podemos evaluar cualquier expresión que haga referencia a funciones definidas en el script.

Al escribir el script debemos recordar que Haskell distingue entre mayúsculas y minúsculas (se dice que es *Case Sensitive*. Por ejemplo, `mi_funcion` y `mi_Funcion` resultan identificadores distintos.

Una buena práctica de programación es documentar siempre el código fuente. Un *comentario* en un script es información de valor para el lector humano, más que para el computador (no interviene para nada al momento de evaluar una expresión). El símbolo "`--`" inicia un comentario, que ocupa la parte de la linea hacia la derecha del símbolo.

4. Tipos de Haskell

Se introduce algunos tipos de datos básicos (Booleanos, Char, Int) y tipos de datos compuestos (tuplas, listas, funciones).

¿Para qué sirven los tipos?

- Detección de errores comunes
- Documentación
- Especificación rudimentaria
- Oportunidad de optimización en compilación

Importante: es una buena práctica en programación empezar dando el tipo del programa que se quiere escribir.

Leer el capítulo 2 del libro

4.1. Booleanos

Se representan por el tipo `Bool` y contienen dos valores: "False" y "True". El tipo de datos `Bool` se puede definir con una declaración de tipos de datos:

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

El prelude incluye varias funciones para manipular valores booleanos:

Sintaxis de Haskell	Sintaxis estándar en lógica
`x && y`	$x \wedge y$
`x || y`	$x \vee y$
`not x`	$\neg x$
`x == y`	$x \equiv y$
`x /= y`	$x \not\equiv y$ o $x \; xor \; y$

Haskell tambien incluye una construcción que permite seleccionar entre dos alternativas dependiendo de un valor booleano.

`if` *condicion* `then` *expresion1* `else` *expresion2*

funciona de la siguiente manera: se evalúa *condicion*; si reduce a `True` (i.e. la condición es verdadera) se evalua *expresion1* y se devuelve su valor, y si reduce a *False* (i.e. la condición es falsa), se evalua y devuelve el valor de `expresion2`.

Obsérvese que una expresión de ese tipo sólo es aceptable si `condicion` es de tipo `Bool` y si *expresion1* y *expresion2* son del mismo tipo.

Leer sección 2.1 del libro.

4.2. Números enteros

Existen dos tipos básicos para representar enteros: `Int` e `Integer`. Ambos incluyen los enteros negativos, el cero y los enteros positivos. La única diferencia es que `Int` representa enteros de rango acotado, mientras que `Integer` representa enteros de rango arbitrario.

Esto significa que operar con valores de tipo `Int` puede provocar un *desbordamiento*, y el resultado puede no ser el correcto. En cambio, la aritmética `Integer` no presenta este inconveniente. Notar que este beneficio se gana a costo de algo de pérdida de performance: la aritmética `Integer` es más lenta que la aritmética `Int`.

En el prelude se incluye un amplio conjunto de operadores y funciones que manipulan enteros:

```
(+)         suma.
(*)         multiplicación.
(-)         substracción.
(^)         potenciación.
negate      menos unario (la expresión "-x" se toma como "negate x")
div         división entera
rem/mod     resto de la división entera. Siguiendo la ley:
            (x `div` y)*y + (x `rem` y) == x
            módulo, como rem sólo que el resultado tiene el mismo
            signo que el divisor.
odd         devuelve True si el argumento es impar
even        devuelve True si el argumento es par.
abs         valor absoluto
signum      devuelve -1, 0 o 1 si el argumento es negativo, cero ó
            positivo, respectivamente.
```

Ejemplos

```
3^4 == 81, 7 `div` 3 == 2, even 23 == False
7 `rem` 3 == 1, -7 `rem` 3 == -1, 7 `rem` -3 == 1
```

4.3. Número en punto flotantes

En ciencias de la computación, a los números con parte fraccionaria, se los denomina *números en punto flotante*. En Haskell, éstos estan provistos por los tipos básicos `Float` y `Double`.

Tener en cuenta que estos valores son aproximaciones que se representan con un número fijo de dígitos, y que bajo ciertas circunstancias esto puede provocar errores de redondeo.

Los números pueden describirse con la notación decimal estándar, o utilizando notación científica; por ejemplo, 1.0e3 equivale a 1000.0, mientras que 5.0e-2 equivale a 0.05.

Otro uso corriente de los números en punto flotante es para representar enteros que son demasiado grandes en valor absoluto para declararse como `Int`, pero que tienen pocos dígitos significativos, por ejemplo 56.23e12.

La diferencia entre `Float` y `Double` reside en que los primeros representan números en punto flotante de precisión simple, mientras que los segundos, números en punto flotante de precisión doble.

El prelude incluye también múltiples funciones de manipulación de flotantes:

`(+), (-), (*), (/)`	`Suma, resta, multiplicación y división fraccionaria.`
`(^)`	`Exponenciación con exponente entero.`
`(**)`	`Exponenciación con exponente fraccionario.`
`sin, cos, tan`	`Seno, coseno y tangente.`
`asin, acos, atan`	`Arcoseno, arcocoseno y arcotangente.`
`ceiling`	`Convierte un número fraccionario en un entero redondeando hacia el infinito.`
`floor`	`Convierte un número fraccionario en un entero redondeando hacia el menos infinito.`
`truncate`	`Convierte un número fraccionario en un entero redondeando hacia el cero (trunca la parte decimal).`
`round`	`Convierte un número fraccionario en un entero redondeando hacia el entero más cercano.`
`fromIntegral`	`Convierte un entero en un número en punto flotante.`
`log`	`Logaritmo en base e.`
`sqrt`	`Raíz cuadrada (positiva).`

4.4. Caracteres

Representados por el tipo `Char`, los elementos de este tipo representan caracteres individuales como los que se pueden introducir por teclado. Los valores de tipo carácter se escriben encerrando el valor entre comillas simples, por ejemplo

```
'a', '0', '.', ' ' y 'Z'
```

Notar que `'0'` y `0` son dos valores distintos. El primero es de tipo caracter y el segundo de tipo entero.

Algunos caracteres especiales deben ser introducidos utilizando un código de escape; cada uno de éstos comienza con el carácter de barra invertida (\) , seguido de uno o más caracteres que seleccionan el carácter requerido. Algunos de los más comunes códigos de escape son:

```
'\\'    barra invertida
'\''    comilla simple
'\"'    comilla doble
'\n'    salto de línea
```

En contraste con algunos lenguajes comunes (como el C, por ejemplo), los valores de tipo Char son completamente distintos de los enteros. Sin embargo, el prelude proporciona dos funciones primitivas, `ord` y `chr`, que permiten realizar la conversión.

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

Por ejemplo, la siguiente función convierte mayúsculas en minúsculas.

```
mayusc :: Char -> Char
mayusc c = chr (ord c - ord 'a' + ord 'A')
```

Leer sección 2.2 del libro.

4.5. Tipos de Funciones

Si `a` y `b` son dos tipos, entonces `a` $\rightarrow$ `b` es el tipo de una función que toma como argumento un elemento de tipo `a` y devuelve un valor de tipo `b`. Las funciones en Haskell son objetos de primera clase. Pueden ser argumentos o resultados de otras funciones o ser componentes de estructuras de datos. Esto permite simular mediante funciones de un único argumento, funciones con múltiples argumentos. Considérese, por ejemplo, la función de suma (+). En matemáticas se toma la suma como una función que toma una pareja de enteros y devuelve un entero. Sin embargo, en Haskell, la función suma tiene el tipo:

```
(+) :: Int -> (Int -> Int)
```

(+) es una función de un argumento de tipo `Int` que devuelve una función de tipo `Int` $\rightarrow$ `Int`. De hecho "(+) 5" denota una función que toma un entero y devuelve dicho entero más 5. Este proceso se denomina currificación y permite reducir el número de paréntesis necesarios para escribir expresiones. De hecho, no es necesario escribir `f(x)` para denotar la aplicación de la función `f` al argumento `x`, sino simplemente `f x`. Ver más sobre currificación en la sección 5.2.

Leer sobre currificación en sección 1.4.2 del libro.

- ¿Cuál es la operación básica de una función?

  La *Aplicación* a un elemento de su partida. La *Aplicación* es la operación con mayor precedencia.

- ¿Qué expresiones denotan funciones?

  * Nombres definidos como funciones: Ej.: cuadrado
  * Funciones Anónimas[3] (lambda abstracciones). Ej.: ($\backslash\, x \rightarrow x + x$)
  * Resultado de usar otras funciones. Ej.: cuadrado . cuadrado

4.6. Listas

Si `a` es un tipo cualquiera, entonces `[a]` representa el tipo de listas cuyos elementos son valores de tipo `a`. Por ejemplo,

[3] Ver más sobre Funciones Anónimas en 5.1

```
xs :: [Int]
xs = [1, 8, 2, 4]
```

El prelude incluye un amplio conjunto de funciones de manejo de listas, por ejemplo:

```
length xs   devuelve el número de elementos de xs
xs ++ ys    devuelve la lista resultante de concatenar
            xs e ys
concat xss  devuelve la lista resultante de concatenar
            las listas de xss
map f xs    devuelve la lista de valores obtenidos al aplicar
            la función f a cada uno de los elementos de la
            lista xs.
```

Convenio: para representar una lista se usa el convenio `xs`, para listas de listas se usa `xss`, y así sucesivamente.

Luego, volveremos con las Listas.

4.7. Cadenas (String)

Una cadena es tratada como una lista de caracteres y el tipo `String` se toma como una abreviación de "[Char]". El tipo `String` es un renombramiento:

```
type String = [Char]
```

Las cadenas pueden ser escritas como secuencias de caracteres encerradas entre comillas dobles. Todos los códigos de escape utilizados para los caracteres, pueden utilizarse para las cadenas.

```
Hugs> "hola"
"hola"

Hugs> ['h','o','l','a']
"hola"

Hugs>
```

Puesto que las cadenas son representadas como listas de caracteres, todas las funciones para listas pueden ser utilizadas también con cadenas:

```
Hugs> length "Hola"
4

Hugs> "Hola, " ++ "amigo"
"Hola, amigo"
```

```
Hugs> concat ["Esto"," ","es"," ","una"," ", "cadena"]
"Esto es una cadena"

Hugs> map ord "Hola"
[104, 111, 108, 97]

Hugs>
```

La diferencia entre 'a' y "a" es que lo primero es un carácter, mientras que lo segundo es una lista de caracteres que contiene un único elemento.

Leer sección 2.7 del libro.

4.8. Enumeraciones

Un modo de definir un nuevo tipo de datos es enumerar explícitamente sus valores. Por ejemplo,

```
data Dia = Dom | Lun | Mar | Mie | Jue | Vie | Sab
```

Los nombres de las constructoras se distinguen de otras clases de nombres haciéndolos comenzar por una letra mayúscula. El nombre de un tipo declarado, también comienza por una letra mayúscula.

El tipo `Dia` define un conjunto de siete valores distintos. Cada uno de estos valores se dice qie es un *contructores* del tipo `Dia`.

Por defecto, esta definición no me permite operar con valores de tipo `Dia` en la manera que esperaria: no puedo comparar dos valores por igualdad, no puedo imprimir sus valores en pantalla, etc.

Para poder hacer esto, debo agregarle a la defínción una opción más:

```
data Dia = Dom | Lun | Mar | Mie | Jue | Vie | Sab
           deriving(Eq, Ord, Enum, Show)
```

que hace que se genere automáticamente las declaraciones de las instancias de las clases de tipos nombradas.

Si `a` es el tipo siendo definido,

- `Eq`: permite usar los operadores == y `/=` con valores de tipo `a`,
- `Ord`: permite usar los operadores <, <=, > y >= con valores de tipo `a`. La relación "<" está dada por el orden en que se enumeran los constructores en la definición del tipo. Así, `Lun < Vie` y `Dom < Sab`,
- `Enum`: define las funciones `fromEnum :: a -> Int` y `toEnum :: Int -> a`. La función `fromEnum` asocia un natural a cada uno de los valores de `a`, de acuerdo al orden en la declaración y comenzando con 0. `toEnum` es la inversa a izquierda de `fromEnum`. Por ejemplo `fromEnum Dom` reduce a `0`, `fromEnum Mie` reduce a `3` y `toEnum 2 == Mar`[4] reduce a `True`.

[4]No se puede inferir el tipo de `toEnum 2`.

- `Show`: permite imprimir los valores del tipo `a` en pantalla.

Leer sección 2.3 del libro.

4.9. Tuplas

Un modo de combinar tipos para crear tipos nuevos es construir pares. El tipo `(A,B)` corresponde a la operación de producto cartesiano de la teoría de conjuntos.

Contamos con los destructores `fst` y `snd` que devuelve la primer y segunda componente de un par respectivamente; por ejemplo `fst (2,'b')` reduce a `2` y `snd (2,'b')` reduce a `'b'`.

Más general, si $T_1, T_2, ..., T_n$ son tipos y $n \geq 2$, entonces hay un tipo de n-tuplas escrito $(T_1, T_2, ..., T_n)$ cuyos elementos pueden ser escritos como $(x_1, x_2, ..., x_n)$ donde cada $x_1, x_2, ..., x_n$ tiene tipos $T_1, T_2, ..., T_n$ respectivamente.

Ejemplo

```
(1, [2], 3)   :: (Int, [Int], Int)
('a', False)  :: (Char, Bool)
((1,2),(3,4)) :: ((Int, Int), (Int, Int))
```

Obsérvese que, a diferencia de las listas, los elementos de una tupla pueden tener tipos diferentes. Sin embargo, el tamaño de una tupla es fijo.

Por completitud, Haskell también proporciona una tupla *vacía*. Por definición, la expresión (), leída `unit`, tiene tipo (). El tipo () tiene dos elementos, $\perp$ y (). Un uso posible de () es convertir las constantes en funciones; por ejemplo,

```
pifun   :: () -> Float
pifun   =  3.14159
```

Al elevar las constantes al nivel de las funciones, se puede ir más allá en el estilo de programación no aplicativo. Por ejemplo, se podría escribir

`cuadrado . cuadrado . pifun` en lugar de `(cuadrado . cuadrado) pi`

Leer sección 2.4 del libro.

4.10. Polimorfismo

Algunas funciones y operadores trabajan con muchos tipos. Por ejemplo,

```
id :: a -> a
id x = x
```

Se lee "`id` es una función que dado un elemento de algún tipo `a`, retorna un elemento de ese mismo tipo". Aquí `a` denota *variables de tipo*. Un tipo que contenga variables de tipo se denomina *tipo polimórfico*.

Entonces, la *identidad* es una función *polimórfica*. El tipo de su argumento puede ser instanciado. Por ejemplo,

```
(id 3)     :: Int        y aquí      id :: Int  -> Int
(id True)  :: Bool       y aquí      id :: Bool -> Bool
```

Leer sección 1.6.1 del libro.

5. Más sobre funciones

Aquí se explica que es currificación, y se da diferentes formas de construir funciones.

5.1. Funciones anónimas (lambda abstracciones)

Una expresión que denota una función son las lambda abstracciones:

$$\begin{array}{rcl} \lambda\, x & . & x + x \\ \lambda\, var & . & expresion \end{array}$$

var: es el *binder*. La variable de cuantificación o ligador.

expresion: es el *scope*. El alcance de la variable de cuantificación.

En Haskell se escriben así

```
\ x -> x + x
```

El significado es el siguiente:

$$(\backslash\, x \to E)\, x_0 \quad \equiv \quad E\,[x := x_0]$$

Por ejemplo,

```
    (\x -> x + x)  2
=                          < Por regla anterior >
    (x + x) [ x := 2 ]
=                          < Sustitución >
    2 + 2
=                          < Aritmética >
    4
```

Veamos dos formas equivalentes de definir una misma función:

$$\begin{array}{lcl} doble\ x & = & x + x \\ doble & = & \backslash\, x \to x + x \end{array}$$

5.2. Currificación

Un artificio útil para reducir el número de paréntesis de una expresión consiste en reemplazar un argumento estructurado por una secuencia de argumentos más simples. Para ilustrar esta idea, consideremos la función *suma*.

```
suma :: (Integer,Integer) -> Integer
suma (x,y) = x + y
```

Otra definición, esencialmente equivalente, de la misma función es:

```
sumac :: Integer -> Integer -> Integer
sumac x y = x + y
```

Veamos antes algo sobre funciones anónimas.

```
    (\y -> y + 1)  8
=                        < def. de lambda expresiones >
    (y + 1) [ y := 8 ]
=                        < Sustitución >
    8 + 1
=                        < Aritmética >
    9
```

Esto es la función sucesor, definida comúnmente como:

```
succ :: Integer -> Integer
succ y = y + 1
```

O sea,

```
succ = \y -> y + 1
```

Y la expresión anterior queda

```
    (\y -> y + 1)  8
=                        < def. de succ >
    succ 8
=                        < def. de succ >
    9
```

Si quisiéramos generalizar aún más la función anterior y en vez de sumar 1 se suma un x genérico. Para el ejemplo siguiente, la función (anónima) recibe dos argumentos (uno después del otro); en este caso, 12 y luego 89, y se intentará sumarlos.

```
    (\x -> (\y -> x + y)) 12 89
=                        < def. de lambda expresiones >
    ((\y -> x + y) 89) [x := 12]
=                        < Sustitución >
    (\y -> 12 + y) 89
=                        < def. de lambda expresiones >
    (12 + y) [ y := 89 ]
=                        < Sustitución >
    12 + 89
=                        < Aritmética >
    101
```

Observe aquí que la función anónima se comporta como lo haría $sumac$. Veamos qué es lo que sucede:

$$\begin{array}{lcl} sumac & = & \backslash\, x \rightarrow (\backslash\, y \rightarrow x + y) \\ sumac\ x & = & \backslash\, y \rightarrow x + y \\ sumac\ x\ y & = & x + y \end{array}$$

Para que la currificación funcione de un modo consistente, necesitamos que la operación de aplicación funcional asocie a la izquierda en las expresiones. Así,

```
sumac x y                 significa       (suma x) y
cuadrado cuadrado x       significa       (cuadrado cuadrado) x
```

Observar que la última es incorrecta. También observar que el operador $\rightarrow$ asocia a la derecha. Esto es:

$$A \rightarrow B \rightarrow C \qquad \text{significa} \qquad A \rightarrow (B \rightarrow C)$$

Ver en la tabla de asociatividad en la sección 2.6.

La currificación conlleva dos ventajas. En primer lugar, puede ayudar a reducir el número de paréntesis que han de escribirse en las expresiones. En segundo lugar, las funciones currificadas pueden ser aplicadas a un solo argumento, dando como resultado otra función que puede ser útil por sí misma. Por ejemplo,

```
suma 1 = succ
```

Demostraremos ésto, primero recordar

```
    sumac x
=                   < Una de las def. de sumac >
    \y -> x + y
```

Instanciando el x a 1

```
    sumac 1
=                   < Def. de sumac >
    \y -> 1 + y
=                   < Conmunt. >
    \y -> y + 1
=                   < Def. de succ >
    succ
```

Si lo deseamos, siempre podemos convertir una función no currificada en otra currificada. La función *curry* toma un función no currificada y devuelve un versión currificada de la misma función; su definición es

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f x y  =  f (x,y)
```

La función *uncurry* hace lo opuesto y convierte una función currificada en otra que no lo es.

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y)  =  f x y
```

Leer sección 1.4.2 del libro.

5.2.1. Forma práctica de ver la Currificación

Para fines práctico, sólo es necesario pensar que la funciones toman una secuencia de argumentos y devuelve un resultado. Por ejemplo,

```
suma4 :: Int -> Int -> Int -> Int -> Int
suma4 x1 x2 x3 x4  =  x1 + x2 + x3 + x4
```

Aunque nos quedamos con ésta idea intuitiva de currificación, es didáctico ver cómo queda poniendo todos los paréntesis.

```
suma4 :: Int -> (Int -> (Int -> (Int -> Int)))
(((suma4 x1) x2) x3) x4  =  x1 + x2 + x3 + x4
```

Nos quedamos con la primera definición de `suma4`.

5.3. Composición de Funciones

La composición de dos funciones `f` y `g` se define mediante la ecuación

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g)  x  =  f (g x)
```

La función `(f . g)` aplicada a `x` se define como el resultado de aplicar primero `g` a `x`, y luego aplicar al resultado `f`.

Recordar la "Relación entre identificadores y operadores" (ver 2.4.3). Por dicha relación, una definición alternativa sería,

```
(.) :: (a -> b) -> (c -> a) -> c -> b
(.) f g x = f (g x)
```

Pero nos quedaremos con la primera.

La composición funcional es una operación asociativa.

```
(f . g) . h  =  f . (g . h)
```

para todas las funciones `f`, `g` y `h` de tipos apropiados. En consecuencia, no hay necesidad de poner paréntesis al escribir secuencias de composiciones.

Leer sección 1.4.7 del libro.

5.4. Funciones totales y parciales.

Decimos que una función es *total*, si está definida para todo elemento de su dominio. En otras palabras, `f :: A -> B` es total si y sólo si `f x` $\neq \perp$ para todo `x::A`. En caso contrario, decimos que `f` es *parcial*. Ver la definición de Botton ($\perp$) en la sección 6.1.

Por ejemplo, si definimos

```
double :: Integer -> Integer
double x = 2 * x

fact :: Integer -> Integer
fact n = if n==0 then 1 else n * fact(n-1)

reciproco :: Float -> Float
reciproco x = 1/x
```

`double` resulta una función total, mientras que `fact` y `reciproco` resultan parciales. Esto se debe a que `fact` aplicado a un entero negativo denota una computación que no termina, y `reciproco` no está definido en cero.

Notar que en programación funcional tenemos una definición diferente de *dominio* de una función a la que tenemos usualmente en matemáticas. En matemáticas escribimos $f : A \to B$ para denotar que f es una función definida para todo x en A, mientras que en Haskell, `f :: A -> B` significa que `f` está definida (i.e. no evalúa a $\perp$) para eventualmente alguno de los valores en `A`.

6. Evaluación

6.1. Reducción

Definición: cero o más contracciones (reemplazo de un redex).

Redex (reducible expresión): subexpresión que coincide con una instancia del lado izquierdo de un ecuación. Un redex más interno es aquel que no contiene otro redex. Un redex más externo es aquel que no está contenido en otro redex. Otra definición posible, subexpresión que se puede reemplazar por otra.

Forma normal: expresión que no contiene redexes (o sea, expresión que no se puede reducir). No toda expresión tiene forma normal. La reducción pretende obtener la formal normal.

¿Cuál es el valor de *infinito*? Definido así:

```
infinito = infinito + 1
```

- Visión denotacional:
  - Botton ($\perp$): valor teórico que representa a un error o a una computación que no termina.
- Visión operacional:
  - expresión cuyas computaciones no terminan (infinitos).
  - expresión que no están definidas ($1/0$).

Notas

- No se puede manejar de manera operacional.
- No se puede preguntar si algo es $\perp$ sin obtener $\perp$. Comparación con $\perp$ da $\perp$.

Función estricta: $f \perp = \perp$

Si la función necesita el valor para dar el resultado entonces es estricta

Función no estricta: $f \perp \neq \perp$

Orden de evaluación: algoritmo para la elección del redex a reducir.

- **Reducción más interna u orden aplicativo:** primero los redexes más internos. También llamada evaluación impaciente.
- **Reducción más externa u orden normal:** primero los redexes más externos.

Nota: en ambos casos, si hay más de un redex al mismo nivel, elige el de más a la izquierda.

Propiedad 1. *La reducción más externa tiene la propiedad importante de que si una expresión tiene forma normal, entonces ésta la encontrará.*

Cuál es el resultado de evaluar en orden aplicativo la siguiente expresión:

```
fst (2, infinito)  =  fst (2, infinito + 1)
                   =  fst (2, infinito + 1 + 1)
                   =  fst (2, infinito + 1 + 1 + 1)
                   =  fst (2, infinito + 1 + 1 + 1 + 1)
                   .
                   .
                   .
```

Y cuál es el resultado de evaluar la misma expresión en orden normal:

```
fst (2, infinito)  =  2
```

Propiedad 2. *Orden aplicativo* $\Rightarrow$ *TODAS las funciones son estrictas*
Orden normal $\Rightarrow$ *hay funciones estrictas y no estrictas*

Nota: la reducción más externa a veces puede necesitar más pasos que la reducción más interna. El problema surge con cualquier función cuya definición contenga apariciones repetidas de un argumento. Por ejemplo, el resultado de evaluar en orden aplicativo la siguiente expresión:

```
    cuadrado (3 + 4)
=                          < def. de + >
    cuadrado 7
=                          < def. de cuadrado >
    7 * 7
=                          < def. de * >
    49
```

Y es el resultado de evaluar la misma expresión en orden normal:

```
    cuadrado (3 + 4)
=                          < def. de cuadrado >
    (3 + 4) * (3 + 4)
=                          < def. de + >
    7 * (3 + 4)
=                          < def. de + >
    7 * 7
=                          < def. de * >
    49
```

Obsérvese que se necesitó un paso más porque en la definición de `cuadrado` aparece repetido su argumento. Esto es:

```
cuadrado x = x * x
```

6.2. Evaluación perezosa (o *Lazy*)

Es una evaluación en orden normal, con las siguientes características adicionales:

- El argumento de una función sólo se evalúa cuando es necesario para el cómputo.
- Un argumento no es necesariamente evaluado por completo; sólo se evalúan aquellas partes que constituyen efectivamente al cómputo.
- Si un argumento se evalúa, tal evaluación se realiza sólo una vez.

Ventajas

- Termina siempre que cualquier otro orden de reducción termine.
- No requiere más (sino posiblemente menos) pasos que la evaluación impaciente.
- Manipulación de estructura de datos infinitas.
- Manipulación de computaciones infinitas.

Desventajas

- Es difícil calcular el costo de ejecución.

Adoptaremos la evaluación lazy como nuestro modelo de cálculo porque tiene las propiedades deseables anteriormente mencionadas.

Leer sección 7.1 del libro.