Paradigma Funcional

Cátedra de Paradigmas de Programación
Facultad Regional Buenos Aires
Universidad Tecnológica Nacional

Ing. Lucas Spigariol

Buenos Aires – 2008

Presentación

¿Por qué Funcional?

El ámbito de aplicación de las ciencias informáticas constituye cada vez más un amplio y creciente abanico de alternativas en el que el profesional de sistemas debe posicionarse para efectuar su labor. Como parte de la compleja dinámica del actual mundo de sistemas, junto con el crecimiento sostenido de desarrollos informáticos para los sectores comerciales en general, con desafíos y requerimientos siempre cambiantes, y la notable diversificación de soluciones computacionales que en los últimos años ha ido impregnando todo tipo de organizaciones e instituciones sociales, uno de los ámbitos que ha hecho de la informática una herramienta más que indispensable para conseguir sus propósitos es el de la investigación científica.

El paradigma de la programación funcional, a medida que fue evolucionando técnicamente en el tiempo, por sus propias características esenciales y condicionado por los vaivenes y tendencias del mercado mundial de sistemas en el que se halla, se ha mantenido como un paradigma vigente, con lenguajes y herramientas concretas que permite desarrollar aplicaciones de propósito general, basándose en principios sólidos y formulaciones de carácter declarativo que constituyen en sí mismas todo un mundo específico dentro del universo de la programación.

El objetivo de este trabajo es presentar de una manera simple y clara las principales características del paradigma, como modesta introducción de un modelo que aún tiene mucho por aportar a las ciencias de la computación. A quienes estén muy acostumbrados a trabajar siguiendo la lógica de los paradigmas imperativos, o inclusive de objetos, o tal vez les resultarán extraños algunas de los conceptos y estrategias que se utilizan, pero una vez entendida su fundamentación teórica y su filosofía de funcionamiento, propia de los paradigmas declarativos, descubrirán lo sencillo y consistente del modelo y se sorprenderán de cómo es posible que con un conjunto relativamente reducido de herramientas y basándose en criterios matemáticos se puedan desarrollar un sinnúmero de aplicaciones. Quienes desde hace años trabajamos con lenguajes funcionales podemos afirmar la utilidad de este paradigma no sólo como instrumento para

desarrollar aplicaciones concretas, sino también como ejercicio mental y de desarrollo de formas alternativas de razonamiento y planteo de soluciones, cuya utilidad y aplicación se extiende a numerosas problemáticas informáticas más allá de los límites del propio paradigma. Precisamente, lo más importante del paradigma es la posibilidad de pensar, abordar y crear abstracciones para un problema de una forma diferente que enriquece la tarea de programar en cualquier lenguajes de programación, y al decir "programar" también se puede incluir el análisis y diseño de un sistema.

Para ilustrar y ejemplificar las explicaciones hemos elegido el Haskell, un lenguaje puramente funcional de propósito general. Un esquema sólido y consistente de tipos de datos y un mecanismo simple de evaluación de expresiones, junto con una implementación eficiente de la evaluación diferida, de las funciones de orden superior, del "garbage collector", del manejo de expresiones lambda y otras características del paradigma funcional, hacen del Haskell una herramienta sencilla de utilizar y a la vez lo suficientemente potente como para desarrollar soluciones de los más variados y complejos problemas de aplicación.

En los primeros capítulos se desarrollan los principales conceptos del paradigma funcional, con ejemplos prácticos.

Luego, se presenta un anexo con las características del lenguaje Haskell, con una recorrida por la sintaxis y las principales funciones predefinidas.

El presente material es una actualización de otros textos que anteriormente había elaborado, con nuevos ejemplos, ampliaciones conceptuales y ejemplos renovados, elaborado en base a mi experiencia de 15 años como docente en la carrera de Ingeniería en Sistemas de Información, en la Facultad Regional Buenos Aires de la Universidad Tecnológica Nacional, y en particular, en la cátedra de Paradigmas de Programación, desde su creación en el marco del plan de estudios de 1995.

El objetivo principal de este trabajo, es que en sus manos se convierta en una herramienta de utilidad para su formación permanente y su ejercicio profesional.

Ing. Lucas Spigariol

Capítulo 1

Conceptos generales

- **El paradigma Funcional**
- **Principales características**
  - Caso práctico: Planilla de cálculo
  - Caso práctico: QuickSort
- **Campo de aplicación**
- **Historia y lenguajes**
- **Lenguaje Haskell**

El paradigma Funcional

El paradigma funcional está basado en el **modelo matemático de composición funcional**. Utiliza funciones matemáticas puras **sin asignaciones destructivas** y, por tanto, sin efecto de lado.

Un programa desarrollado en un lenguaje del paradigma funcional consiste en un conjunto ordenado de **declaraciones**, en las que no existe el concepto de asignación, como tampoco algoritmos, sentencias, comandos ni estructuras de control de naturaleza imperativa. La importancia del concepto de declaratividad en este paradigma, permite encuadrarlo dentro de los paradigmas **declarativos**[1]**.** La clave para que funcione es la existencia de un mecanismo interno del sistema (llamado comúnmente "motor") que evalúa las expresiones de una manera acorde, mediante una serie de pasos intermedios de transformación de código que son transparentes para el programador a la hora de desarrollar el software.

A diferencia de otros paradigmas donde un programa describe en detalle los pasos que la máquina debe ejecutar para realizar el ordenamiento, con la mayoría del código ocupándose de los detalles de bajo nivel de la manipulación de datos, un programa funcional resuelve el problema de ordenamiento en un **más alto nivel**, con una **mejora significativa en la brevedad y claridad**. Los programas son más fáciles de diseñar, de escribir y de mantener, pero dan al programador menos posibilidades de control sobre la máquina.

[1] El concepto de declaratividad y su vinculación con la organización de los paradigmas se puede ampliar en SPIGARIOL, Lucas. *Conceptos fundamentales de los Paradigmas de Programación.* Apunte CEIT. 2008.

Las **funciones**, basadas en el concepto matemático del término, son expresiones que establecen una relación de correspondencia o asociación entre miembros de un conjunto (el dominio) y miembros de otro conjunto (la imagen), en un determinado sentido. De esta manera, para cada elemento del conjunto dominio corresponde un determinado elemento del conjunto imagen, y no existen elementos del conjunto dominio que no tengan su correspondiente imagen. Generalizando, las funciones tienen su dominio conformado por varios conjuntos y existe un elemento en la imagen para cada elemento del producto cartesiano de todos ellos.

La definición de una función debe contemplar, ya sea con expresiones para casos puntualmente o genéricas, todos las entradas posibles y asociar cada una de ellas con otra expresión que corresponde al resultado. Cuando la función es invocada, los argumentos recibidos se unifican con las expresiones codificadas y se retorna el resultado de su evaluación.

En este modelo, el resultado de un cálculo es la entrada del siguiente, y así sucesivamente hasta que una composición produce el resultado deseado. Así, un programa es un conjunto de funciones que cooperan entre ellas para el logro de un objetivo común. Toda expresión es evaluada mediante un proceso reducción, que básicamente consiste en combinar un mecanismo de unificación (*pattern matching*) y sustitución. En este proceso se utilizan las definiciones de función realizadas hasta obtener un valor no reducible.

La escritura de sistemas de software grandes que el trabajo es difícil y cara, y su mantenimiento lo es aún más. Lenguajes de programación funcionales, pueden hacerlo más fácil y más barato.

Buena parte del tiempo de vida de un producto de software se utiliza en la especificación, el diseño y el mantenimiento, y no en la programación. Los lenguajes funcionales son magníficos para escribir especificaciones que pueden ser ejecutadas realmente (y por lo tanto, probado y eliminado errores). Tal especificación entonces es el primer prototipo del programa final.

Los programas funcionales son también relativamente fáciles de mantener, porque el código es más corto, más claro, y el control riguroso del efecto de lado elimina una enorme cantidad de interacciones imprevistas. [2]

Principales características

La **transparencia referencial** permite que el valor que devuelve una función esté únicamente determinado por el valor de sus argumentos consiguiendo que una misma expresión tenga siempre el mismo valor.

Permite aumentar su flexibilidad y realizar unidades de software genéricas mediante la utilización **de tipos de datos genéricos**, que es una forma de implementar el **polimorfismo** en el paradigma funcional.

[2] Sitio oficial de Haskell (www.haskell.org)

La **evaluación perezosa o diferida** (*lazy evaluation*) en una característica de gran potencia, por la que se evalúa del programa solamente tanto como se requiere para conseguir la respuesta. En la invocación de funciones con sus correspondientes argumentos, la evaluación de las expresiones intervinientes se posterga hasta el momento en que realmente sean utilizadas.

Una herramienta para la formulación de algunas soluciones, principalmente en las más de bajo nivel, es la **recursividad,** basada en el principio matemático de inducción, que se ve expresada en el uso de tipos de datos recursivos, como las listas, y funciones recursivas que las operan.

Una característica fundamental es la posibilidad de tratar a las funciones como datos mediante la definición de **funciones de orden superior**, que permiten un gran nivel de abstracción y genericidad en las soluciones. el uso correcto de las funciones de orden superior puede mejorar substancialmente la estructura y la modularidad de muchos programas.

Entre las estructuras de datos que utiliza se destacan las **listas**, como tipo de dato compuesto que, junto con las tuplas, permite organizar conjuntos de valores.

En el modelo funcional es más sencillo demostrar la corrección de los programas ya que se cumplen propiedades matemáticas tradicionales como la propiedad conmutativa, asociativa, etc.

Los lenguajes funcionales relevan al programa de la compleja tarea de gestión de memoria. El almacenamiento se asigna y se inicializa implícitamente, y es recuperado automáticamente por un recolector de la basura (*garbage collector*). La tecnología para la asignación de memoria y de la recolección de la basura están actualmente bien desarrollados por lo que su costo de funcionamiento es leve.

Caso práctico: Planilla de cálculo

Cualquier persona que ha utilizado una planilla de cálculo tiene experiencia de la programación funcional. En una hoja de balance, uno especifica el valor de cada celda en términos de valores de otras celdas. El foco está en qué debe ser computado, no cómo debe serlo. Por ejemplo:

- **No se especifica el orden** en la cual las celdas deben ser calculadas, ya que es la planilla de cálculo quien computa las celdas en un orden que respete sus dependencias.
- **No se le dice** a la planilla de cálculo **cómo asignar su memoria**, se espera que presente un panorama al parecer infinito de celdas, y que asigne memoria solamente a esas celdas cuando realmente las utilice.
- En general, se especifica el valor de una celda **mediante una expresión cuyas piezas se puedan evaluar en cualquier orden**, **antes que por una secuencia de comandos que compute su valor.**

Una consecuencia interesante de la no especificación de orden de la planilla de cálculo, es que la noción de la asignación no es muy útil, ya que si no se sabe exactamente cuándo sucederá una asignación, no se puede hacer mucho uso de ella. Esto, lo une al paradigma con el que comparte la centralidad de la declaratividad y lo pone un contraste fuerte con programas en los lenguajes de paradigmas procedurales, que consisten esencialmente en una secuencia cuidadosamente especificada de asignaciones, o de objetos, en la cual el ordenar de las llamadas del método es crucial para el significado de un programa.

Este foco en el de alto nivel, de especificar el "qué" antes que en el bajo nivel del "cómo" es una característica el distinguir de lenguajes de programación funcionales.

Otro lenguaje con fuertes rasgos funcionales bien conocido es el lenguaje de consulta estándar de base de datos SQL. Una consulta del SQL es una expresión que implica las proyecciones, selecciones, ensambles y así sucesivamente. La pregunta dice qué relación debe ser computada, sin decir cómo debe ser computada. De hecho, la pregunta se puede evaluar en cualquier orden conveniente. Las puestas en práctica del SQL realizan a menudo la optimización extensa de la pregunta que (entre otras cosas) calcula hacia fuera la mejor orden para evaluar la expresión.

Caso práctico: QuickSort

Las planillas de cálculo y el SQL son ejemplos bastante especializadas. Los lenguajes de programación funcionales toman las mismas ideas y las mueven en el reino de la programación de uso general. Para tener una idea de como es un programa funcional, y la expresividad de sus lenguajes, se pueden ver los siguientes programas que realizan un ordenamiento rápido (quicksort).

El primer programa se escribe en un lenguaje del paradigma funcional, Haskell, y el otro en un lenguaje imperativo, C. [3]

Mientras que el programa en C describe en detalle los pasos que la máquina debe ejecutar para realizar el ordenamiento -con la mayoría del código ocupándose de los detalles de bajo nivel de la manipulación de datos- el programa de Haskell resuelve el problema de ordenamiento en un más alto nivel, con una mejora significativa en la brevedad y claridad.

Ejemplo de Quicksort en Haskell

```
qsort [ ]    = [ ]
qsort (x:xs) = qsort (filter (< x) xs) ++ [x] ++ qsort (filter (>= x) xs)
```

[3] Ejemplos de implementación extraídos del sitio oficial de Haskell (www.haskell.org)

Ejemplo de Quicksort en C

```
void qsort(int a[], int lo, int hi) {
{
  int h, l, p, t;
  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];
    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);
    t = a[l];
    a[l] = a[hi];
    a[hi] = t;
    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

Los programas funcionales tienden a ser mucho más concisos que las contrapartes imperativas. El quicksort es un caso extremo, pero en general los programas son más cortos (en una proporción de 2 a 10).

Los programas funcionales son a menudo más fáciles de entender. Se debe poder entender el programa sin mucho conocimiento anterior del Haskell o del quicksort. Lo mismo ciertamente no pueden ser dichos del programa de C. Toma un tiempo entender, y aún cuando se lo logra, es muy fácil cometer un pequeño error y terminar con un programa incorrecto.

Una explicación detallada del quicksort de Haskell:

> La primera línea se lee: "El resultado de ordenar una lista vacía ([ ]) es otra lista vacía". La segunda: "El resultado de ordenar una lista, cuyo primer elemento se denomina *x* y el resto se denomina *xs*, es la concatenación (++) de las listas que se obtienen de ordenar

> los elementos de *xs* que son menores que *x* y de ordenar los elementos de *xs* que son mayores o iguales a *x*, con *x* intercalado en el centro."

En Haskell, el programa no sólo ordenará listas de números enteros, sino también listas de números de punto flotante, listas de caracteres, listas de listas; de hecho, ordenará listas de cualquier cosa para las cuales tenga sentido tener las operaciones "menor" y "mayor". En cambio, la versión de C puede ordenar solamente un conjunto de números enteros. Se ve cómo el polimorfismo permite reutilizar el código.

El quicksort de C utiliza una técnica ingeniosa, por la que ordena el conjunto sin usar ningún almacenamiento adicional. Consecuentemente, funciona rápidamente y en una cantidad pequeña de memoria. En cambio, el programa de Haskell asigna bastante memoria adicional detrás de lo que se ve y funciona algo más lento que el otro programa.

En un lenguaje imperativo, el quicksort asume esta complejidad algorítmica para lograr una reducción en el tiempo de ejecución. En los usos donde se requiere la mejor performance a cualquier costo, o cuando la meta es ajustar minuciosamente un algoritmo de bajo nivel, un lenguaje imperativo como C sería probablemente una opción mejor que Haskell, precisamente porque proporciona un control más íntimo sobre la manera exacta de la cual se realiza el cómputo.

Por otra parte, son pocos los programas que requieren performance a cualquier costo. Después de todo, hace tiempo que se ha dejado de escribir programas de lenguaje assembler, excepto quizás para ciertas aplicaciones de más bajo nivel, porque las ventajas del tener un modelo de programación de soporte compensan con creces el costo de un tiempo de ejecución más modesto.

Con el paradigma funcional sucede algo similar: Los lenguajes funcionales tienden hacia un modelo de programación de alto nivel, con una ganancia en el diseño, desarrollo y mantenimiento y una pérdida relativa de control sobre la máquina, pero que para muchos programas es un precio perfectamente aceptable para el buen resultado que se obtiene.

Campo de aplicación

Desde sus inicios, los lenguajes funcionales fueron pensados como lenguajes de uso universal para el procesamiento de datos en todo tipo de aplicaciones. Uno de los principales ámbitos de su utilización está relacionado principalmente con la **investigación científica** y **aplicaciones matemáticas**.

Lo más importante, es que aún no estando en una situación donde se pueda usar un lenguaje funciona para desarrollar una determinada aplicación, el aprendizaje de la dinámica, el enfoque y los conceptos del paradigma facilitan la realización de un mejor programa en cualquier lenguaje.

Historia y lenguajes

Los orígenes teóricos del modelo funcional se remontan a los años 30 en los cuales Church propuso un nuevo modelo de estudio de la computabilidad mediante el **cálculo lambda**. Este modelo permitía trabajar con funciones como objetos de primera clase. En esa misma época, Shönfinkel y Curry construían los fundamentos de la lógica combinatoria que tendrá gran importancia para la implementación de los lenguajes funcionales.[4]

La arquitectura del computador ejerce una influencia estableciendo restricciones a los métodos de diseño de soluciones, tendiendo a limitar el lenguaje para que éste se pueda implementar eficientemente en las máquinas existentes. Durante años los lenguajes han estado restringidos por las ideas de Von Neumann, ya que la mayoría de las computadoras tienen una estructura muy similar a la original de éste.

Hacia 1950, John McCarthy diseñó el lenguaje **LISP** (List Processing) que utilizaba las listas como tipo básico y admitía funciones de orden superior. Este lenguaje se ha convertido en uno de los lenguajes más populares en el campo de la Inteligencia Artificial. Sin embargo, para que el lenguaje fuese práctico, fue necesario incluir características propias de los lenguajes imperativos como la asignación destructiva y los efectos laterales que lo alejaron del paradigma funcional.

A principios de la década de los setenta aparecieron los primeros síntomas de lo que se ha denominado **crisis del software**. Los programadores que se enfrentan a la construcción de grandes sistemas de software observan que sus productos no son fiables. La alta tasa de errores conocidos (*bugs)* o por conocer pone en peligro la confianza que los usuarios depositan en sus sistemas. La raíz del problema radica en la dificultad de **demostrar que el sistema cumple los requisitos** especificados. La verificación formal de programas es una técnica costosa que en raras ocasiones se aplica.

Una posible solución fue proponer un **modelo de computación diferente** al modelo imperativo tradicional**.** Se basa en la idea de que los problemas detectados son inherentes al modelo computacional utilizado y su solución no se encontrará a menos que se utilice un modelo diferente. Así nace la **programación funcional** o aplicativa, cuyo objetivo es describir los problemas mediante funciones matemáticas puras sin efectos laterales.

En 1978 **J. Backus** (uno de los diseñadores de FORTRAN y ALGOL) consiguió que la comunidad informática prestara mayor atención a la programación funcional con su artículo *"Can Programming be liberated from the Von Neumann style?"* en el que criticaba las bases de la programación imperativa tradicional mostrando las ventajas del modelo funcional. Además Backus diseñó el lenguaje funcional **FP** (*Functional Programming)* con la filosofía de definir nuevas funciones combinando otras funciones.

[4] Se puede ver un desarrollo histórico más completo en LABRA G, José E., *Introducción al lenguaje Haskell*

A mediados de los 70, **Gordon** trabajaba en un sistema generador de demostraciones denominado LCF que incluía el lenguaje de programación **ML** (*Metalenguaje).* Aunque el sistema LCF era interesante, se observó que el lenguaje ML podía utilizarse como un lenguaje de propósito general eficiente. ML optaba por una solución de compromiso entre el modelo funcional y el imperativo ya que, aunque contiene asignaciones destructivas y Entrada/Salida con efectos laterales, fomenta un estilo de programación claramente funcional. Esa solución permite que los sistemas ML compitan en eficiencia con los lenguajes imperativos.

A mediados de los ochenta se realizó un esfuerzo de estandarización que culminó con la definición de **SML** *(Stándar ML).* Este lenguaje es fuertemente tipado con resolución estática de tipos, definición de funciones polimórficas y tipos abstractos. Al mismo tiempo que se desarrollaban FP y ML, **David Turner** (primero en la Universidad de St. Andrews y posteriormente en la Universidad de Kent) trabajaba en un nuevo estilo de lenguajes funcionales con evaluación perezosa y definición de funciones mediante encaje de patrones. El desarrollo de los lenguajes SASL (*St. Andrews Static Language*), KRC *(Kent Recursive Calculator*) y **Miranda** tenía como objetivo facilitar la tarea del programador incorporando facilidades sintácticas como las guardas, el encaje de patrones, las listas por comprensión y las secciones.

A comienzos de los ochenta surgió una **gran cantidad de lenguajes funcionales** debido a los avances en las técnicas de implementación. Entre éstos, se podrían destacar Hope, LML, Orwell, Erlang, FEL, Alfl, etc. Esta gran cantidad de lenguajes perjudicaba el desarrollo del paradigma funcional. En 1987, se celebró una conferencia en la que se discutieron los problemas que creaba esta proliferación. Se decidió formar un comité internacional que diseñase un nuevo lenguaje puramente funcional de propósito general denominado Haskell.

Sin embargo, se observa que a pesar de los años transcurridos, los lenguajes de programación funcional han tenido éxito a la hora de reemplazar a los lenguajes convencionales en ciertas áreas de aplicación, pero no han logrado ubicarse masivamente en aplicaciones de uso general.

En 1996 apareció la versión 1.3 del lenguaje Haskell que incorporaba, entre otras características, mónadas para Entrada/Salida, registros para nombrar componentes de tipos de datos, clases de constructores de tipos y diversas librerías de propósito general. Posteriormente surgen nuevas versiones.

En 1998 se decidió proporcionar una versión estable del lenguaje, que se denominará **Haskell98** a la vez que se continúa la investigación de nuevas características.[5]

[5] La más completa información sobre el lenguaje Haskell puede consultarse en el sitio: www.haskell.org

Lenguaje Haskell

Haskell es un **lenguaje funcional puro**, de propósito general, que incluye muchas de las últimas innovaciones en el desarrollo de los lenguajes de programación funcional, como son las funciones de orden superior, evaluación perezosa (*lazy evaluation*), tipado polimórficos, tipos definidos por el usuario, encaje de patrones (*pattern matching*), y definiciones de listas por comprensión.

Su nombre proviene de **Haskell Brooks Curry**, quien trabajó en la lógica matemática que sirve como fundamento de los lenguajes funcionales. **Haskell** se basa en el ***cálculo lambda***, por lo tanto el signo de lambda (**λ**) es utilizado como ícono.

El lenguaje **Haskell** se creó con el objetivo de unificar las características más importantes de los lenguajes funcionales. Al diseñar el lenguaje se observó que no existía un tratamiento sistemático para implementar el polimorfismo con lo cual se construyó una solución conocida como las clases de tipos. Incorpora, además, otras características interesantes como la facilidad en la definición de tipos abstractos de datos, el sistema de entrada/salida puramente funcional y la posibilidad de utilización de módulos.

Hay un amplio número de compiladores y de intérpretes disponibles, la mayoría libres. Uno de ellos, de simple manejo y amplitud de prestaciones es el intérprete **Hugs** (basado en el **Gofer**, un sistema experimental de programación funcional, presentado en 1991), que ofrece una gran compatibilidad con el estándar **Haskell**. En realidad, el nombre **Hugs** fue en principio escogido como un mnemónico de "*Haskell Users' Gofer Sistem*"

En este trabajo, los ejemplos se desarrollan utilizando la versión que se denomina *WinHugs 98*.[6]

[6] Se puede ampliar y descargar el software gratuitamente en www.haskell.org/hugs

Capítulo 2

Las funciones

- **Transparencia referencial**
- **Fundamento matemático**
- **Definición de funciones**
- **Mecanismo de evaluación**
  - Funciones de varios argumentos
  - Uso de patrones
  - Expresiones condicionales
- **Recursividad**
  - Inducción
  - Construcción de funciones recursivas
  - Formulación declarativa
  - Demostración

Transparencia referencial

Toda funcionalidad que desarrolla un programa en el paradigma funcional, se describe mediante **funciones** que implementan la transparencia referencial, consiste en que el valor de una expresión depende únicamente del valor de sus componentes, de manera que siempre que se evalúa el mismo bloque de software con los mismos parámetros, se obtiene el mismo resultado, sin importar el comportamiento interno de dicho bloque.

No tiene tienen asignación destructiva ni efecto de lado. La evaluación de un bloque de código no produce un cambio en el estado de información del sistema que pueda afectar una posterior evaluación del mismo u otro bloque.

Las variables son utilizadas para hacer referencia a valores intermedios y parámetros de las funciones, como resultados de cálculos anteriores y entradas a subsiguientes cálculos. Si bien internamente puedan utilizar alguna porción de memoria, no son "celdas" en las que se vayan realizando sucesivas "asignaciones".

Teniendo en cuenta que no existen algoritmos, sentencias, comandos ni estructuras de control imperativas, un programa es un **conjunto de**

declaraciones de funciones, que mediante un mecanismo interno de evaluación se combinan de manera tal de responder a las consultas realizadas.

Fundamento matemático

Las funciones del paradigma funcional se basan en las **nociones matemáticas del concepto de función**.

En una primera aproximación, se entiende por función una relación entre dos conjuntos por la cual a cada elemento de un conjunto le corresponde le corresponde un elemento del otro conjunto. Al primer conjunto se lo denomina "**dominio**" y al segundo "**imagen**". A todo elemento del dominio le corresponde uno y sólo uno de los elementos de la imagen. También en consistencia con la definición matemática de función, no hay ningún impedimento para que diversos elementos del dominio les corresponda el mismo elemento de la imagen.

Las funciones se pueden definir mediante la enumeración de todos los casos posibles o mediante expresiones genéricas.

Ejemplo:

Una función llamada "calcular" que relaciona algunos números con otros se expresa con la siguiente tabla:

función "Calcular"	
Dominio	Imagen
0	1
1	1
2	4
3	10

Una función f, definida de los números reales en reales, que calcula el número siguiente de un número.

$f: \Re \rightarrow \Re$

$f(x) = x + 1$

Las funciones implican una **transformación** que se establece siempre de la misma manera, es decir independientemente del lugar o momento en que se evalúa. La utilización de las funciones consiste en realizar una invocación de la función, en la que se especifica el argumento, y obtener un resultado. Para devolver dicho resultado, la función realizó algún tipo de operación interna de acuerdo a cómo haya sido definida.

Ejemplo:

En el ejemplo anterior, si se invoca a la función "calcular" con el valor 2 como argumento se obtiene como resultado el valor 4. Esto se logra buscando en la primer columna de la tabla el primer valor y devolviendo el valor que se encuentre en la misma fila pero en la otra columna.

En el segundo caso, ante la invocación f(2) se obtiene como resultado un 3. Para ello, en primer lugar la variable independiente x asume el valor 2 y es utilizado en la expresión contigua, en la que se realiza la suma entre dicho valor y la constante 1.

Definición de funciones

Los lenguajes del paradigma funcional desarrollan un esquema similar para la definición de sus funciones. Sus elementos básicos son

- Una función se identifica con un **nombre** único.
- El alcance de los conjuntos **dominio** e **imagen** de la función se representan mediante la declaración del "**prototipo**" de la función donde se especifican los tipos de datos correspondientes.[7]
- En una función se enumera **una serie de ecuaciones** con todos los casos a contemplar. Cada uno de ellas, además de repetir el nombre de la función, tiene dos partes, unidas por una igualdad, donde sus términos especifican a qué elemento del conjunto dominio le corresponde qué otro elemento del conjunto imagen:
    - Una expresión que representa los elementos posibles del dominio, llamada "**patrón**" (pattern).
    - Una **expresión** que representa el elemento de la imagen a retornar como **resultado**.
- Cada una de las ecuaciones que forman parte de la definición de una función se pueden definir haciendo referencia a un elemento en particular (con una constante), o mediante expresiones genéricas (con variables) que representen a un conjunto de elementos.

Ejemplo:

Una función definida por dos ecuaciones

```
f :: UnTipo -> OtroTipo

f expresion1 = expresion2

f expresion3 = expresion4
```

Expresion1 y **expresión3** representan a los elementos del dominio y **expresión2** y **expresión4** a los elemento de la imagen que se corresponden con los anteriores, respectivamente.

[7] No es obligatorio incluir la información de tipo, ya que éste puede ser inferido por el sistema (Ver capítulo 4 "Tipos de datos genéricos"). Sin embargo, es una buena práctica, ya que aporta claridad al código y permite detectar errores de inconsistencia de tipos de datos.

Ejemplo:

Las mismas funciones matemáticas anteriores pueden escribirse como funciones del paradigma de la siguiente manera:

```
calcular:: Int -> Int
calcular 0 = 1
calcular 1 = 1
calcular 2 = 4
calcular 3 = 10
```

Se interpreta como que **calcular** es una función que recibe un valor de tipo **Int** (número entero) y devuelve otro valor de tipo **Int**, de acuerdo a las especificaciones detalladas. La siguiente invocación:

```
calcular 2
4
```

El valor **2** del argumento se unifica con el patrón constante **2** en la tercera ecuación, devolviendo **4** como resultado

```
f :: Float –> Float
f  x  =  x + 1
```

Se interpreta como que **f** es una función que recibe un valor de tipo **Float** (número real de punto flotante) y devuelve otro valor de tipo **Float**, resultante de sumarle **1**. La invocación:

```
f 5
6
```

El valor **5** del argumento se unifica con la variable **x**, que se sustituye a la derecha de la ecuación, se evalúa la suma y se devuelve el **6**.

En una función con varias ecuaciones, es frecuente que en algunas de ellas se utilicen constantes y en otras variables.

Ejemplo:

Una función que informa si un carácter es o no una vocal:

```
esVocal:: Char -> Bool
esVocal 'a' = True
esVocal 'e' = True
esVocal 'i' = True
esVocal 'o' = True
esVocal 'u' = True
esVocal x = False
```

Cuando se invoca

esVocal 'i'

True

Se unifica la tercera ecuación y el resultado es **True**. Cuando se invoca

esVocal 'p'

False

Al no coincider el argumento con ninguna de las constantes, se unifica el valor **'p'** con la variable **x** de la última ecuación y el resultado es **False**.

Mecanismo de evaluación

El mecanismo de evaluación es un proceso que consiste en tomar una expresión e **ir transformándola** aplicando las definiciones de funciones **hasta que no pueda transformarse más**. La expresión resultante se denomina **representación** "**canónica**" y es retornada como valor final.

De esta manera, ante la invocación de una función con su nombre y argumentos, el mecanismo de evaluación, llamado "**encaje de patrones**" (*pattern matching*), sigue básicamente los siguientes pasos:

- **Identificar** entre todas las funciones definidas, la que corresponda a **la función invocada**, utilizando su nombre.
- **Chequear la consistencia de los tipos de datos** de los argumentos con la definición de los tipos de datos del dominio de la función.
- **Recorrer secuencialmente** las distintas ecuaciones contempladas, **intentando unificar** los valores de argumentos con los de cada patrón.
- Cuando esta unificación es satisfactoria, **se evalúa la expresión** a la derecha de la igualdad **sustituyendo los valores unificados** y se retorna su resultado.

En los patrones, una expresión constante se unifica con otra expresión constante cuando representa el mismo valor. Cualquier expresión constante se unifica con una variable.

A la expresión a retornar se la evalúa para encontrar una expresión canónica que se pueda retornar. Si la expresión tiene valores constante, estos son retornados directamente. Si se incluyen variables, se asumen para su evaluación los valores unificados, sustituyendo cada aparición de la variable por su correspondiente valor. Si la expresión incluye invocaciones a otras funciones, se implementa nuevamente el mecanismo de evaluación hasta encontrar una expresión canónica.

Ejemplo:

Ante la siguiente invocación:

calcular 2

4

Luego de ubicar la definición de la función **calcular** y chequear que el **2** es un valor válido para el tipo de dato **Int**, se analizan las ecuaciones. En la primera, el **2** del argumento no coincide con el **0** del patrón. En la siguiente, el **2** tampoco se puede unificar con el **1**. En la tercera, el argumento y patrón coinciden en el valor 2**.** Entonces se evalúa la expresión que está del otro lado de la igualdad, que tiene un valor constante **4**, y se lo retorna.

Ante la siguiente invocación:

f 2

3

Luego de ubicar la definición de la función **f** y chequear que el **2** es un valor válido para el tipo de dato **Float**, se evalúan las ecuaciones. En la primera (y única, en este caso) el **2** del argumento se unifica con la variable **x** del patrón. Entonces se evalúa la expresión **x + 1** sustituyendo la variable **x** por su valor **2** y se realiza la suma obteniendo como resultado parcial un **3** (En realidad se dispara una nueva invocación, en esta caso a la función **"+"** con argumentos **2** y **1**, con su consiguiente proceso de evaluación, que en definitiva retorna el valor 3**)** y, tratándose de una expresión canónica que no puede seguir reduciéndose, se retorna este mismo valor como resultado final de la función.

Funciones de varios argumentos

Las funciones pueden tener más de una variable independiente y por lo tanto ser definidas para recibir varios parámetros. En este caso, la expresión a la derecha de la igualdad, el patrón, está formado por tantos términos como variables independientes.

La cantidad de argumentos se denomina **aridad.** Si una función fuese definida por más de una ecuación, todas ellas deben tener la misma aridad.

Ante una consulta, el mecanismo de **pattern matching** intenta unificar todos los argumentos a la vez para la misma ecuación, cada uno con su correspondiente expresión. Cuando logra unificarlos a todos, como se explica anteriormente, se evalúa y retorna la expresión del otro lado de la igualdad.

En forma análoga, la especificación del prototipo de la función debe incluir a un tipo de dato para cada argumento.

Ejemplo:

Una función f cualquiera de 2 argumentos:

```
f :: UnTipo -> OtroTipo -> OtroTipoMas
f expresion1 expresion2 = expresion3
f expresion4 expresion5 = expresion6
```

UnTipo y **OtroTipo** son los tipos de dato correspondientes a los argumentos del dominio y **OtroTipoMas** es el tipo de dato de la imagen. **Expresión1** y **expresión4** representan a diferentes casos para el primer argumento y **expresion2** y **expresion5** a los valores posibles del segundo argumento.

Ante una invocación:

f valor1 valor2

Si **expresión1** se logra unifica con **valor1** y simultáneamente **expresión2** con **valor2**, se retorna **expresión3**. De igual manera, el resultado será **expresion6** si se unifica **expresión4** con **valor1** y **expresión5** con **valor2**.

Ejemplo:

Una función llamada y, que opera con valores booleanos y resuelve una conjunción lógica.

```
y:: Bool -> Bool -> Bool
y True True = True
y a b = False
```

Ante una invocación:

```
y  True False
False
```

En la primera ecuación aunque se unifique el primer argumento no lo hace el, por lo que en definitiva no logra unificar la ecuación. En cambio, en la segunda ecuación, al ser dos variables unifica sin dificultad y retorna **False**.

Otra invocación:

```
y  True True
True
```

Unifica en la primera ecuación

En general, con funciones de "n" argumentos, para cada ecuación va a haber n elementos a la izquierda de la igualdad y uno a la derecha, y el prototipo va a estar definido por n+1 tipos de datos.

Ejemplo:

```
f ::  Tipo1 -> Tipo2 -> … -> TipoN> -> TipoN+1
f expresion1 expresion2 … expresionN = expresionN+1
```

Cada una de las declaraciones **Tipo1**… **TipoN** representa cada tipo de dato del domino y el elemento **TipoN+1** representa la imagen de la función. La **expresion1** … **expresionN** representa el patrón que se corresponde uno con cada argumento de la función, siendo **expresionN+1** el resultado.

El nombre de una variable no puede utilizarse más de una vez en la parte izquierda de cada ecuación en una definición de función.

Ejemplo:

Esta definición **no** será aceptada por el sistema:

```
iguales x x = True
iguales x y = False
```

Uso de patrones

Los patrones no sólo pueden contener variables o constantes, sino también otras expresiones:

Variable anónima: Se representan por el carácter "_" y encajan con cualquier valor, pero no es posible referirse posteriormente a dicho valor, por lo que se utiliza en casos donde no interesa de qué valor se trata. Usar una variable común en vez de una anónima no altera el funcionamiento, pero se pierde claridad y simplicidad.

Ejemplo:

Una función muy simple que indica si un número es cero

```
esCero :: Int -> Bool
esCero 0 = True
esCero x = False
```

Al invocar a la función con un argumento que no sea **0**, la unificación se satisface asumiendo **x** un valor que no vuelve a utilizar, ya que no se lo sustituye del otro lado de la igualdad. La definición puede ser modificada, reemplazando la última línea por

```
esCero _ = False
```

En este caso, al invocarse también con un argumento que no sea **0**, la presencia de la variable anónima satisface el pattern matching y el comportamiento es el mismo.

Constructores: De acuerdo al tipo de dato de un patrón, existen determinadas expresiones, llamadas constructores, que permiten realizar la unificación de una manera particular. En el caso de los números enteros, existe el constructor +, con el que se pueden establecer patrones de la forma x+k, donde x es una variable y k una constante entera. De esta manera, la unificación con la variable x se produce deduciendo (o "despejando") **cuál sería el valor de la variable para que coincida el patrón con el argumento**.

Ejemplo:

Definiendo una función como:

```
cuadradoDelAnterior (x+1) = x * x
```

Dada la invocación:

```
cuadradoDelAnterior 5
16
```

Para responder, se intenta unificar el valor **5** con la expresión **x + 1**, y en consecuencia la variable **x** asume el valor **4**. El mecanismo de evaluación continúa su proceso realizando la multiplicación sustituyendo **x** por **4** y devolviendo el valor **16**.

Expresiones condicionales

Cuando los patrones no son suficientes para contemplar y diferenciar todos los casos a contemplar en las ecuaciones, se puede complementar su uso

estableciendo **condiciones adicionales que se deben satisfacer para que la unificación sea satisfactoria**.

Así, cualquiera de las ecuaciones de la definición de una función podría subdividirse con "**guardas" ("|")** donde se indican las diferentes condiciones (expresión que al evaluarse retorna un valor de verdad) sobre los valores de los argumentos que se requiere que se cumplan para cada caso. Cada condición tiene asociado su correspondiente expresión que establece el resultado a devolver.

Su paralelo en las matemáticas son las funciones definidas por partes, donde según los diferentes rangos de valores de la variable independiente, varía la expresión que define la función.

Una ecuación con guardas puede tener dos o más condiciones, que son evaluadas secuencialmente hasta encontrar una que se unifique. En general, aunque no es obligatorio, para garantizar que alguna se cumpla, se añade la expresión "**otherwise"** como última condición que siempre es verdadera.

Ejemplo:

```
funcion    expresionX expresionY    | condicion1 = expresion1
                                    | condicion2 = expresion2
                                    ...
                                    | condicionN-1 = expresionN-1
                                    | otherwise = expresionN
```

Habiéndose unificado **expresionX** y **expresionY**, se evalúa **condicion1** y si es verdadera, se devuelve **expresion1**, si no, se evalúa **condicion2** y así sucesivamente. Si ninguna de las condiciones **1** a **N-1** se cumplió, **otherwise**, que está definida con un valor **True**, siempre unifica y retorna la **expresionN**

Ejemplo:

Una función que devuelve el signo que le corresponde a un número.

```
signo :: Int -> String
signo 0 = "cero"
signo x | x > 0 = "positivo"
        | x < 0 = "negativo"
```

Para el caso en que el argumento es el valor constante **0**, unifica en la primera ecuación y retorna "**cero**". Cuando el argumento es otro valor, se unifica con la variable **x**. Allí hay dos alternativas, una para cada rango de valores posibles de **x**. Si es mayor a **0** verifica la condición que se encuentra a continuación de la guarda, en la segunda ecuación y retorna "**positivo**". Si no, si es menor a **0**, unifica en la última ecuación y retorna "**negativo**".

Ejemplo:

Los dos resultados posibles a devolver en una función mínimo que pretende obtener el menor número entre dos números dados, no pueden ser identificados mediante el uso

de patrones presentado hasta el momento, sino que es necesario discriminar ambos casos mediante la evaluación de una condición que compare a los dos argumentos:

```
minimo :: Int -> Int -> Int
minimo x y | x < y = x
           | otherwise = y
```

Para cualquier valor de **x** e **y**, cuando **x** es menor que **y**, **x** es el mínimo. En otro caso contrario, el mínimo es **y**.

Recursividad

Una característica, lejos de ser exclusiva del paradigma pero que tiene gran importancia en la formulación de algunas soluciones, es la recursividad, que se ve expresada en el uso de **funciones recursivas**. La recursividad, entendida como iteración con asignación no destructiva, está relacionada con el principio de **inducción** y surge de la definición axiomática de los números naturales[8]. Una función recursiva se define con al menos un término recursivo, en el que se vuelve a invocar la función que se está definiendo, y algún término no recursivo como caso base para que la recursividad se detenga y la invocación no entre en un ciclo infinito (excepto que se quiera iterar infinitas veces, en cuyo caso la función puede no tener un caso base). [9]

Inducción

El **principio de inducción** sostiene que para demostrar que una determinada propiedad se cumple para todos los números naturales, es suficiente con demostrar que se cumplen dos condiciones.

- La propiedad P se cumple para **1**.
- **Si** una propiedad P se cumple para el número natural **n entonces** se cumple para **n+1**.

Construcción de funciones recursivas

Aplicándolo a las funciones del paradigma funcional, una expresión recursiva permite resolver numerosos problemas. La solución de un problema se plantea con dos casos:

- Un "**caso base**" de nivel "1", no recursivo, generalmente de simple resolución, a veces, inclusive, de naturaleza trivial.
- Un "**caso genérico**" de nivel "n + 1"donde la recursividad consiste en invocar a la misma función para un caso de nivel "n". Asumiendo como

[8] Teorema planteado por Giuseppe Peano en su trabajo de sistematización de las matemáticas con una notación funcional

[9] El paradigma funcional también permite la definición de funciones recursivas sin caso base para generar, por ejemplo, listas infinitas. Ver capítulo 5 "Evaluación diferida".

cierto que la función para el caso "n" devuelve un resultado correcto, se lo utiliza dentro de una expresión que represente la resolución de caso de nivel "n + 1".

Al invocar la función en un caso cualquiera, ésta se irá invocando a sí misma tantas veces como sea necesario, evaluándose cada vez la expresión del caso genérico, y se evaluará una vez la expresión del caso base, que evitará una recursividad infinita.

Cuando una función para realizar su cometido necesita invocar a otra función, no existe ninguna restricción que impida que la función invocada sea la misma función. De esta manera, la evaluación de una función recursividad es considerada naturalmente como **un caso particular de invocación** y se implementa el mismo mecanismo de evaluación que para todas las funciones.

Ejemplo:

Se define una función llamada prod, que realiza el producto entre dos números enteros como sumas sucesivas, de la siguiente manera:

```
prod :: Int -> Int -> Int

prod  0 n  = 0

prod m n  = n + prod (m – 1)  n
```

Desde algún contexto cualquiera donde se quiera calcular el producto de dos números se invoca a la función **prod** enviando como parámetros los valores correspondientes. Suponiendo que se quiere calcular el producto entre **2** y **3** la invocación sería:

```
prod 2 3
```

Entre todas las funciones definidas, se busca mediante el nombre la definición de la función **prod**. Se encuentra así la primer línea con el prototipo de la función donde además de coincidir el nombre de la función se chequea que haya una correspondencia entre el tipo de dato de los parámetros enviados con los tipos de datos definidos en la función. El primer argumento, el **2**, es un valor entero que se corresponde con el tipo de dato **Int** indicado en el prototipo, y lo mismo sucede con el segundo argumento, el **3**. Así, el chequeo de datos es satisfactorio.

En la primer ecuación se intenta unificar el valor constante **2** con el valor constante **0** y el valor constante **3** con la variable **n**. Como en el primer caso no se puede unificar dos valores constantes que son diferentes, por más que en el segundo caso hubiera podido realizarse, la unificación no se produce.

En la segunda ecuación se intenta unificar el valor **2** con la variable **m** y el valor **3** con la variable **n**. Lo hace satisfactoriamente, por lo que **m** asume el valor **2** y **n** el valor **3**. Comienza entonces la evaluación de la expresión a la derecha del "**=**" para obtener el valor a devolver como resultado final. Sustituyendo las variables por sus valores, la expresión unificada resulta:

```
prod 2  3  = 3 + prod (2 – 1) 3
```

Las reglas de precedencia en la evaluación de la expresión y el uso de los paréntesis nos indican que en este caso tiene mayor precedencia la función **"-"**, luego la función **prod** y por último la función **"+"**

La subexpresión **(2 – 1)** se resuelve de acuerdo a la implementación de la función **"-"** (que realiza la resta de números enteros) y será reemplazada por su resultado. Aquí nos abstraemos de la forma en que internamente se resuelva la resta y del momento en que

realmente se ejecuta[10], pero en definitiva se obtiene como resultado en valor entero **1**. La expresión resultante es:

prod 2 3 = 3 + prod 1 3

La subexpresión **prod 1 3** se resuelve mediante una nueva invocación (recursiva) a la misma función **prod**, siguiendo una serie de pasos similares a estos, pero lo hará en forma independiente, es decir, con otro lote de variables con valores que no se mezclan ni confunden con los de esta invocación. En forma similar al caso anterior, nos abstraemos por el momento de cómo se realiza, pero sabemos que la función **prod** resuelve el producto entre dos números enteros por lo que podemos confiar en que el resultado será el esperado, en este caso, el valor entero **3**. La expresión resultante es

prod 2 3 = 3 + 3

Finalmente, la expresión **3 + 3** se resuelve mediante la invocación a la función **"+"** que realizará la operación de la suma de ambos números de acuerdo a la forma en que esté definida internamente, y devolverá el resultado **6**. La expresión resultante queda como

prod 2 3 = 6

Por lo tanto, no habiendo más expresiones que evaluar, es devuelta la expresión canónica **6** como resultado al contexto donde se produjo la invocación.

Cuando se invoque **prod** con otros parámetros (excepto cuando el primero sea **0**) la forma de resolución será la misma, como va a suceder por ejemplo cuando se lo haga con **1** y **3**, ejecutándose siempre el caso genérico e invocando recursivamente la función para resolver un caso "menor".

Dado el caso en que precisamente la invocación sea por ejemplo:

prod 0 3

El mecanismo de evaluación, habiendo encontrado la definición de la función y chequeado la consistencia de los tipos de datos, comienza el mecanismo de encaje de patrones. A diferencia de las otras invocaciones, en este caso se unifica en la primer ecuación el argumento **0** con el patrón **0** y el argumento **3** con la variable **n**, por lo que se evalúa la expresión contigua y se retorna su valor, en este caso, el **0**. La expresión unificada sería:

prod 0 3 = 0

Para esta invocación la segunda ecuación no se llega nunca a ejecutar, por lo que el llamado recursivo no se produce.

Demostración

Para demostrar que la función realiza su **funcionalidad correctamente** (y como estrategia para desarrollar su solución) **no es necesario hacer un seguimiento** de todas las invocaciones sucesivas que se hacen, sino garantizar que tanto el caso genérico como el caso base sean correctos independientemente.

Generalizando, puede haber funciones recursivas con más de dos ecuaciones en las que haya varias expresiones recursivas y varios casos bases que contemplen el abanico de casos posibles y deberá demostrarse que todos sean correctos, pero basta con probar cada uno una sola vez.

[10] Para mayores precisiones acerca del momento en que se evalúa realmente cada expresión, ver el capítulo 5 "Evaluación diferida".

Formulación declarativa

Haciendo presente la naturaleza declarativa del paradigma funcional, por la que la **formulación correcta del problema** y la **descripción de la respuesta esperada** constituyen en sí mismas prácticamente la codificación de la función que soluciona el problema, el uso de la recursividad requiere el ejercicio de abstracción suficiente para encontrar una definición del problema y su respuesta en términos recursivos. En general, todo procesamiento de datos de carácter **iterativo** puede replantearse en forma **recursiva**.

Ejemplo:

El cálculo del factorial de un número natural es un ejemplo típico de un problema que se puede definir fácilmente en términos recursivos y, en consecuencia, resolverlo mediante una función recursiva. En otras palabras, la solución se obtiene "traduciendo" la formulación del problema hecha en un lenguaje natural a un lenguaje de programación funcional.

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

Capítulo 3

Listas y tuplas

- **Tipos de datos compuestos**
- **Listas**
  - Construcción
  - Descomposición
  - Cadenas
- **Tuplas**
- **Listas como secuencias**
- **Listas por comprensión**

Tipos de datos compuestos

Los tipos de datos compuestos representan **estructuras de datos** que contienen simultáneamente un conjunto de valores y tienen la particularidad de poder ser tratados en conjunto **como una unidad** o **distinguiendo por separado sus componentes** y utilizados independientemente.

Tienen definidos operadores especiales, llamados **constructores** que permiten tanto construir un dato compuesto a partir de un conjunto de datos simples o descomponerlo en sus partes, según la manera y el lugar de la función donde se utilicen

Los tipos de datos compuestos más utilizados son las **listas** y las **tuplas**.

Tuplas

Una tupla representa una estructura de datos que contiene una **cantidad fija** de elementos, en la que se pueden combinar **distintos tipos de datos**, cada uno en una posición fija de la estructura.

Para definir el tipo de dato en el prototipo de una función, una tupla se representa encerrando entre paréntesis y separando por comas a cada uno de los tipos de dato de los elementos que contiene, en el orden deseado. Para representar cada tupla, se presentan los valores de manera análoga, respetando las posiciones indicadas por los correspondientes tipos de datos en la definición.

Ejemplo:

Si el tipo de dato de la tupla es:

(Char, Int, Bool)

Algunos posibles valores son

('a', 7, True)

('x', 0, False)

('5', -10, True)

Tanto para construir una tupla en una expresión a retornar de una función, como para descomponerla en un patrón, se enumeran las componentes con comas y se encierran entre paréntesis.

Ejemplo:

Dada una función que suma las componentes de una tupla que representa un par ordenado de enteros:

```
sumaPar :: ( Int, Int ) -> Int
sumaPar ( x , z ) = x + z
```

Ante la invocación

```
sumaPar (5, 8)
13
```

Al unificarse, la tupla se descompone en sus partes, asumiendo la variable **x** el valor **5** de la primera componente y la variable **z** el valor **8** de la segunda componente.

Ejemplo:

Dada una función que determina si un par ordenado formado por un entero y un carácter tiene un 0

```
origen :: (Int, Char) -> Bool
origen (0, x)  = True
origen n       = False
```

En la primera ecuación se descompone la tupla recibida como argumento y sólo se unifica si su primer componente coincide con la constante **0** de la primera componente del patrón. En la segunda ecuación, no es necesario descomponer la tupla, por lo que la variable **n** hace referencia a la tupla completa.

Ejemplo:

Dada una función que recibe 2 enteros y devuelva una tupla donde la primer componente sea el mayor de los dos y la segunda el menor.

```
maxmin :: Int -> Int -> (Int, Int)
maxmin x y | x>=y = (x, y)
           | otherwise = (y, x)
```

Una vez resuelta la unificación por el caso que corresponda, en la expresión que indica el valor a retornar se construye la tupla utilizando los valores que hayan asumido las variables **x** e **y**.

Listas

Una lista representa una estructura de datos que contiene una **cantidad variable** de elementos, siempre del **mismo tipo**, que están dispuestos secuencialmente dentro de la estructura. El tipo de dato de las listas está definido en forma **recursiva**, con dos posibilidades:

- Una lista es una estructura **vacía**
- Una lista es **un elemento** concatenado con **otra lista**

En el prototipo de una función una lista se representa encerrando entre corchetes (**[ ]**) el tipo de dato de los elementos que contiene.

Ejemplo:

[Int]	Lista de enteros
[Char]	Lista de caracteres

Construcción

Una lista vacía se representa con [ ].

Una lista con elementos se representa enunciado el elemento ubicado en la primera posición (llamado comúnmente cabeza de la lista) a continuación el constructor, que es el signo dos puntos (:), y por último la otra lista (llamada comúnmente cola de la lista).

Ejemplo:

1:[ ]

Una lista con el entero **1** como primer elemento y la lista vacía como cola, por lo tanto una lista con un único elemento, el **1**.

'a':('b':[ ])

Una lista con el carácter **'a'** como cabeza y una cola que es una lista formada a su vez por una cabeza **'b'** y una cola vacía. En definitiva, una lista con dos elementos, los caracteres **'a'** y **'b'**

Las listas no vacías pueden ser construidas enunciando explícitamente sus elementos, separándolos con comas (,) y encerrando la expresión completa entre corchetes. Esta notación es equivalente a la anterior para construir una lista con elementos constantes.

Ejemplo:

1:[ ]	es equivalente a	**[1]**
'a':('b':[ ])	es equivalente a	**['a', 'b']**

Descomposición

En el patrón de una función, se puede tratar a **toda la lista como un conjunto** o **descomponerla en su cabeza y su cola** para poder unificarlas por separado y luego ser utilizados sus valores en forma independiente, utilizando el constructor. En ambos casos, se pueden utilizar variables o constantes.

Por una cuestión de nomenclatura, se debe encerrar toda la expresión del patrón entre paréntesis.

Ejemplo:

Dada la función long que calcula la longitud de una cadena de enteros:

```
long :: [ Int ] -> Int
long    [ ]       = 0
long (n:ns)    =  1 + long ns
```

El patrón de la primera ecuación presenta la constante de la lista vacía, que sólo se unificará cuando el argumento sea también una lista vacía, caso en que retornará el valor entero **0**.

En la segunda ecuación, el patrón es una lista descompuesta en cabeza y cola, con las variables **n** y **ns** respectivamente, por lo que se unificará con cualquier lista que contenga al menos un elemento.

Ante la invocación:

```
long [ 3, 4, 6]
3
```

No se unifica en la primera ecuación por no ser una lista vacía. En la segunda ecuación se unifica el valor **3** con la variable **n** y la lista formada por los valores **4** y **6** con la variable **ns**. A continuación, se evalúa la expresión a retornar con los siguientes valores sustituidos:

```
long ( 3 : [ 4, 6 ] ) = 1 + long [ 4, 6 ]
```

Al evaluarse, la invocación a long retornará **2** (porque se asume que **long** funciona correctamente para un caso genérico menor que el que se está evaluando) y al sumarle **1** se obtiene como resultado final **3** (también se asume que la función **+** suma correctamente)

Ejemplo:

Dada una función similar a la anterior, pero que cuenta sólo la cantidad de números 7 que haya en la lista de enteros:

```
long7 :: [ Int ] -> Int
long7    [ ]              = 0
long7 (7:ns)  =  1 + long7 ns
long7 (n:ns)  =  long7 ns
```

Ante una invocación, la unificación se produce en forma similar al ejercicio anterior, con la diferencia que en la segunda ecuación, al tener como cabeza un valor constate, sólo se unifica cuando la lista recibida como argumento coincide con dicho valor (en ese caso

se incrementa en uno respecto de la cantidad de números 7 que haya en al cola). Si la cabeza de la lista no coincide, se pasa a evaluar el patrón de la ecuación siguiente, donde al tener variables tanto en la cabeza como en la cola, se unifica con cualquier lista no vacía.

En ambos ejemplos, también pudo haberse utilizado una variable anónima en el patrón en vez de la variable **n**, ya que ésta no es utilizada luego.

```
long7 (_:ns)  =  long7 ns
long7 (_:ns)  =  1 + long7 ns
```

Cadenas

Una cadena es tratada como un caso particular de una **lista**, **formada por caracteres**. Su tipo de dato se puede expresar tanto como [Char] como con su sinónimo "String". Al expresarlas por extensión, se puede enumerar los caracteres sin separarlos por comas y encerrar todo entre comillas dobles (“”).

Ejemplo:

“”	es equivalente a	**[ ]**
[‘a’, ‘b’]	es equivalente a	**“ab”**

Ejemplo

Una función que dadas dos palabras, devuelva los caracteres comunes a ambas palabras, sin repetirse.

```
comunes :: String -> String -> String
comunes   [ ]  _     = [ ]
comunes (x:xs) ys    | esta x ys      = x : comunes xs (sacaRepe x ys)
                     | otherwise     = comunes xs ys
```

La función principal **comunes,** cuando no es vacía, considera dos posibles casos: que el primer elemento de la lista, **x,** pertenezca o no a la otra lista, **ys** (tarea que realiza la función auxiliar **esta**). En caso afirmativo, la lista solución está formada por el elemento **x**, concatenado con los elementos comunes entre la cola de la primer lista, **xs**, y la lista **ys** sin las ocurrencias del mencionado elemento (que se obtiene invocando a la función **sacaRepe)**

```
sacaRepe :: Char -> String -> String
sacaRepe _   [ ]      = [ ]
sacaRepe x (y:ys)     | x == y        = sacaRepe x ys
                      | otherwise     = y : sacaRepe x ys
```

La función **sacaRepe**, dado un carácter y una cadena, devuelve la cadena recibida sin las ocurrencias del carácter.

```
esta :: Char -> String -> Bool
esta x [ ]            = False
esta x (y:ys)         | x /= y        = esta x ys
                      |otherwise      = True
```

La función **esta** devuelve True cuando un determinado carácter forma parte de una cadena.

Una invocación general puede ser:

comunes "paradigmas" "programacion"

"parigm"

Las listas y tuplas pueden combinarse entre sí de multiples maneras, formando listas de tuplas, tuplas de listas, etc.

Ejemplo:

Si el tipo de dato es:

[(Char, [Int])]

Algunos posibles valores son

[('a', [2,4,6]) , (e, [1,2])]

[ ]

[('@', [123,176,345]), ('&', [1223]), ('%', [ ]), ('#', [-1,-2,-6,0])]

Se interpreta como una lista de tuplas formadas por un carácter, y una lista de números.

Listas como secuencias

La notación de las secuencias aritméticas permite generar una gran cantidad de listas útiles.

Existen cuatro formas de expresar secuencias aritméticas:

[m..] Produce una lista (potencialmente infinita) de valores que comienzan con m y se incrementan en pasos simples.

Ejemplo:

[1..]	es equivalente a	**[1, 2, 3, 4, 5, 6, 7, 8, 9, etc...**

[m..n] Produce la lista de elementos desde m hasta n, con incrementos en pasos simples. Si m es menor que n devuelve la lista vacía.

Ejemplo:

[-3..3]	es equivalente a	**[-3, -2, -1, 0, 1, 2, 3]**
[1..1]	es equivalente a	**[1]**
[9..0]	es equivalente a	**[ ]**

[m, m'..] Produce la lista (potencialmente infinita) de valores cuyos dos primeros elementos son m y m'. Si m es menor que m' entonces los siguientes

elementos de la lista se incrementan en los mismos pasos. Si m es mayor que m' entonces los incrementos serán negativos. Si son iguales se obtendrá una lista infinita cuyos elementos serán todos iguales.

Ejemplo:

[1, 3..]	es equivalente a	**[1, 3, 5, 7, 9, 11, 13, etc...**
[0, 0..]	es equivalente a	**[0, 0, 0, 0, 0, 0, 0, etc...**
[5, 4..]	es equivalente a	**[5, 4, 3, 2, 1, 0, -1, etc...**

[m, m'..n] Produce la lista de elementos [m,m',..] hasta el valor límite n. Los incrementos vienen marcados por la diferencia entre m y m'.

Ejemplo:

[1, 3..12]	es equivalente a	**[1, 3, 5, 7, 9, 11]**
[0, 0..10]	es equivalente a	**[0, 0, 0, 0, 0, 0, 0, etc...**
[5, 4..1]	es equivalente a	**[5, 4, 3, 2, 1]**

Las secuencias no están únicamente restringidas a enteros, pueden emplearse con elementos enumerados como caracteres, flotantes, etc.

Ejemplo:

['0'..'5'] ++ ['A'..'E'] es equivalente a **"012345ABCDE"**

[1.2 , 1.35 .. 2.00] es equivalente a **[1.2, 1.35, 1.5, 1.65, 1.8, 1.95]**

Las secuencias pueden ser utilizadas en cualquier contexto donde se espere una lista.

Ejemplo

Una función que dada una lista de números y dos posiciones devuelve la sublista compuesta por los elementos de la lista que estén comprendidos entre las posiciones dadas, inclusive:

```
intervalo:: [ Int ] -> Int -> Int -> [ Int ]
intervalo   [ ]  _ _   =  [ ]
intervalo (x:xs) 1 1   =  [ x ]
intervalo (x:xs) 1 n   =  x : intervalo xs 1 (n - 1)
intervalo (x:xs) m n   =  intervalo xs (m - 1) (n - 1)
```

Una invocación posible es:

```
intervalo [ 9, 18 .. 90 ] 3 6
[ 36, 45, 54, 63]
```

Listas por comprensión

La notación de listas por comprensión permite declarar de forma concisa una gran cantidad de iteraciones sobre listas. Esta notación está adaptada de la teoría de conjuntos. Sin embargo, se trabaja con listas, no con conjuntos. El formato básico de la definición de una lista por comprensión es una expresión genérica, que define a cada elemento de la lista, y una serie de cualificadores, que determinan las características, posibilidades y restricciones para cada uno de ellos.

Ejemplo:

[expresion | cualificador1, cualificador2 … cualificadorN]

Los cualificadores son de dos tipos:

- **Generadores:** Un cualificador de la forma **patron <- lista** es utilizado para extraer de la **lista,** en el mismo orden en que están, cada elemento que unifique con el **patrón**.

Ejemplo:

[x*x | x <- [1..10]]

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

El cualificador generador **x <- [1..10]** hace que la variable **x** asuma cada uno de los valores de la lista **[1..10]**. La expresión **x * x** indica que el contenido de la lista va a ser el resultado de evaluar la expresión para cada uno de los valores **x** cualificados.

- **Filtros:** Es una expresión de tipo booleana que se utiliza combinada con los generadores, que actúa como restricción sobre los valores generados, de manera que sólo van a ser tenidos en cuenta para integrar la lista aquellos que satisfagan la expresión.

Ejemplo:

[x*x | x <- [1..10] , even x]

[4, 16, 36, 64, 100]

Se comporta como en el ejemplo anterior, con la diferencia que la expresión **x*x** solo se evalúa sobre los elementos generados que además verifiquen el cualificador **even x**, es decir, aquellos que sean pares.

Cuando aparecen varios cualificadores hay que tener en cuenta que las variables generadas por los cualificadores posteriores varían más rápidamente que las generadas por los cualificadores anteriores

Ejemplo

[(x, y) | x<- [1..3], y <- [1..2]]

[(1,1), (1,2), (2,1), (2,2), (3,1), (3,2)]

Representa una lista de tuplas de dos de la forma **(x,y)**, donde **x** varía entre **1** y **3** e **y** lo hace entre **1** y **2**. En otras palabras , genera todas las combinaciones.

Por otra parte, los cualificadores posteriores pueden utilizar los valores generados por los anteriores

Ejemplo

[(x, y) | x <- [1..3], y <- [1..x]]

[(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]

Representa una lista de tuplas de dos de la forma **(x,y)**, donde **x** varía entre **1** y **3** e **y** lo hace entre **1** y el valor de **x**. Por lo tanto, para **x** = **1**, **y** vale sólo **1;** para **x** = 2, **y** asume los valores **1** y **2;** y por último, para **x** = **3**, **y** vale **1**, **2** y **3**.

El orden en que se indican los cualificadores es importante. Puede tener efecto directo en la eficiencia. Las variables definidas en cualificadores posteriores ocultan las variables definidas por cualificadores anteriores. No es una buena costumbre reutilizar los nombres de las variables, ya que dificulta la comprensión de los programas, aunque las expresiones sean válidas.

Ejemplo

[x | x <- [[1,2], [3,4]], x <- x]

[1, 2, 3, 4]

Capítulo 4

Tipos de datos genéricos

- **Inferencia de tipos**
- **Polimorfismo**
- **Variables de tipos de datos**
- **Restricciones de contexto**

Inferencia de Tipos

El sistema de inferencia de tipos consiste en que el sistema contiene un mecanismo que infiere el tipo de dato de las expresiones, por lo que **no es obligatoria la declaración el prototipo de las funciones**.

Cuando el programador declara el tipo de alguna expresión, el sistema chequea que el tipo declarado coincide con el tipo inferido. De esta manera, la explicitación de los prototipos **permite una mayor seguridad** evitando errores de tipo en tiempo de ejecución y una mayor eficiencia, evitando realizar comprobaciones de tipos en tiempo de ejecución.

Ejemplo:

Declarada la siguiente función, que traduce un valor de verdad al idioma español:

```
traduce True = "verdadero"
traduce False = "falso"
```

El sistema infiere automáticamente que el tipo es

```
traduce :: Bool -> String
```

Si en el programa se hubiese declarado que tiene un tipo diferente, el sistema daría un error de tipos en tiempo de compilación

Polimorfismo

Remitiendo a las ideas básicas del polimorfimo[11], que consiste en construir piezas de software genéricas que trabajen indistintamente con diferentes tipos

[11] El concepto de polimorfismo se puede ampliar en SPIGARIOL, Lucas. *Conceptos Fundamentales de los Paradigmas de Programación.* Apunte CEIT. 2008.

de entidades y que las puedan considerar como intercambiables, se puede afirmar que dicha situación se aplica en el paradigma funcional, al definir funciones cuyos argumentos pueden ser indistintamente, de uno u otro tipo de dato.

Los sistemas de inferencia de tipos aumentan su flexibilidad mediante la utilización de polimorfismo, que permite que el tipo de una función dependa de un parámetro. El polimorfismo tiene como gran beneficio una mayor reutilización de código ya que no es necesario repetir algoritmos para estructuras similares.

Ejemplo:

La misma función del capítulo anterior que calcula la longitud de una lista de números enteros, si se la define sin el prototipo, resulta:

```
long    [ ]      = 0
long (n:ns)   =  1 + long ns
```

El sistema de inferencia de tipos, en vez de devolver

```
long :: [Int] -> Int
```

Va a devolver

```
long :: [a] -> Int
```

Ya que no hay nada en la definición de la función que indique (y a la vez que restrinja) que el contenido de la lista son números enteros. La variable **a** representa la posibilidad que el tipo de dato sea cualquiera y que por lo tanto, no sea necesario definir una función **long** para listas de otro tipo de contenido, sino que para todas ellas se puede utilizar la misma definición de **long**.

Variables de tipos de datos

Hay muchas funciones en las que las ecuaciones y expresiones que determinan su funcionalidad son las mismas independientemente del tipo de dato que reciben o devuelven. En estos casos, la utilización **de tipos de datos genéricos**, mediante variables de tipos de datos, permite implementar el polimorfismo para aumentar la flexibilidad y genericidad de las funciones, definiéndolas una sola vez y utilizándolas para diferentes tipos de datos.

En la declaración del prototipo de una función, en vez de especificar un tipo de dato en particular, se utiliza **una variable de tipo de dato**. En el momento de la invocación, esta variable infiere su valor a partir del tipo de dato de los argumentos. La utilización de ese argumento que se haga en el interior de la función debe ser lo suficientemente genérica para que funcione correctamente en todos los casos, ya que puede ser en algunas ocasiones de un tipo de dato y en otras ocasiones de otro.

Cuando hay varios tipos de datos genéricos en una función, éstos pueden coincidir o no entre sí. Para indicar que deban ser del mismo tipo, se utiliza la misma variable y para permitir que puedan sean diferentes, se utilizan nombres de variables distintos.

Ejemplo:

Una función que concatena dos listas de números enteros, definida de la siguiente manera:

```
concatenacion:: [ Int ] -> [ Int ] -> [ Int ]
concatenacion [ ]    ys    = ys
concatenacion (x:xs) ys = x : concatenacion xs ys
```

Excepto por su prototipo, no tiene nada en particular, que la diferencie de una función que concatene dos listas de caracteres o de cualquier otro tipo de dato.

Para hacerla genérica, se pude reemplazar el prototipo por el siguiente:

```
concatenacion:: [a] -> [a] -> [a]
```

Donde **a** es el nombre de la variable de tipo de dato y siendo todos los argumentos del mismo tipo. Al invocarla:

```
concatenacion [1,3,4] [2,4]
[1,3,4,2,4]
```

La variable **a** asume como valor el tipo de dato **Int**. Si se la invoca:

```
concatenacion “argen” “tina”
“argentina”
```

La variable **a** asume como valor el tipo de dato **Char**.

Ejemplo:

Dada una función que retorna la segunda componente de una tupla de dos elementos, su definición puede ser polimórfica.

```
segundaComponente :: ( a, b ) -> b
segundaComponente    ( _ , x ) = x
```

Como los tipos de datos de ambas componentes no tienen la obligatoriedad de ser los mismos, se utilizan dos variables diferentes, **a** y **b**, pero como el tipo de dato del valor de retorno de la función debe coincidir siempre con el tipo de dato de la segunda componente de la tupla, se usa la misma variable **b**.

Restricciones de contexto

Hay casos donde si bien se quiere generalizar una función para permitir recibir valores de diferentes tipos de datos, dadas las características particulares de la función (dada por las funciones auxiliares que invoca internamente), **no se pueden recibir valores de cualquiera de todos los tipos de datos existentes, sino solamente de algunos de ellos**.

Para permitir esta generalización pero a la vez restringirla a un determinado subconjunto de tipos de datos, en la definición del **prototipo de una función** se establece una **restricción** a cada variable utilizada.

Existen varias restricciones ya predefinidas y cada una está asociada a una serie de tipos de datos que coinciden en tener definidas cierto grupo de

funciones necesariamente. Se habla de "clases" de tipos de datos, siendo cada tipo de dato una "instancia" de ella. [12]

La restricción de contexto se especifica anteponiendo el prefijo correspondiente a la variable en el prototipo de la función. Puede haber varias restricciones para la misma o diferentes variables.

Eq: Representa una restricción de **equivalencia**. Incluye a los tipos de datos que tienen definido un criterio de igualdad propio y en consecuencia pueden ser comparados mediante las funciones **==** y **/=** (distinto). Entre ellos se encuentran **Char**, **Int**, **Float**, **Bool** y las listas y tuplas formadas por ellos.

Ord: Representa una restricción de **orden**. Incluye a los tipos de datos que tienen definido un criterio de ordenamiento propio y en consecuencia pueden ser comparados mediante las funciones de los tipos de dato **Eq**, y además las funciones **<**, **>**, **<=** y **>=**. Entre los tipos de datos incluidos están todos los mencionados para **Eq**. **Ord** incluye a **Eq**, por lo tanto, si una variable tiene una utilización que ameritaría ser definida tanto **Eq** como **Ord**, con especificar **Ord** es suficiente.

Num: Representa una restricción **numérica**. Incluye a los tipos de datos numéricos en general, que tienen definidas las operaciones básicas más frecuentes, como la multiplicación (*), la suma (+) y la resta (-), no así la división. Entre los tipos de datos incluidos más comunes están **Float** e **Int**.

Ejemplo:

Dada una función que informa si un determinado elemento pertenece o no a una lista:

```
pertenece :: (Eq a) => a -> [a] -> Bool
pertenece y   [ ]       =  False
pertenece y (x:xs)    | y == x          = True
                      | otherwise       = pertenece y xs
```

Como en la definición se invoca a la función **"=="** con los valores de las variables **y,** que unifica con el primer argumento, y la variable **x**, que unifica con el primer elemento de la lista que se reciben como segundo argumento, ambos deben ser de un tipo genérico **a** que tenga la restricción de contexto **Eq**.

Ejemplo:

Dada una función que devuelve el mayor elemento de una lista:

```
mayor :: (Ord a) => [a] -> a
mayor [ x ]    =  x
mayor (x:xs) |  x > mayor xs     = x
             | otherwise         = mayor xs
```

El valor que es devuelto es un elemento de la lista, por lo que el tipo de dato del contenido de la lista y el de la imagen deben ser necesariamente el mismo. Como en la definición se invoca a la función **">"** con los valores de las variables **x** y el resultado de la

[12] Internamente las restricciones de contexto se organizan mediante un sistema que incluye el uso de "clases", "instancias", "herencia" y otros términos típicos del paradigma de objetos.

invocación a **mayor,** ambos deben ser de un tipo genérico **a** que tenga la restricción de contexto **Ord**.

Ejemplo:

Sea una función que calcula la sumatoria de elementos de una lista:

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

Como en la definición se invoca a la función **"+"** con dos argumentos, un elemento **x** de la lista y el resultado de **sum**, ambos deben ser de un tipo genérico **a** que tenga la restricción de contexto **Num**. (Esta definición con contempla el caso de lista vacía)

Capítulo 5

Evaluación diferida

- **Evaluación diferida**
- **Listas infinitas**

Evaluación diferida

Para la evaluación de los argumentos de las funciones se utiliza el sistema de **evaluación diferida** (*lazy evaluation*), **evaluación no estricta** o **evaluación perezosa**[13], por lo que la evaluación de **las expresiones invocadas se posterga hasta el momento en que realmente sean utilizadas**.

En la invocación de una función, si se envía como argumento una expresión que por ejemplo incluye la invocación de otra función, lo que provoca la evaluación diferida es que la expresión que se envía como argumento no se evalúa antes de ser enviada, sino que se envía la expresión sin evaluar y es la función invocada la que va a evaluarla en el momento en que lo requiera.

Ejemplo:

Dada la definición de una función que calcula el cuadrado de un número:

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

Ante la siguiente invocación:

```
cuadrado (3+4)
```

La evaluación diferida hace que se reciba el argumento sin evaluar y se unifique la variable x con la expresión completa 3+4. Por lo tanto la expresión unificada sería:

```
cuadrado (3+4) = (3+4) * (3+4)
```

Y recién en este momento se evalúa la suma para poder retornar el resultado final. En este caso, el resultado final y los cálculos realizados no difieren respecto de haber utilizado otro tipo de evaluación.

[13] En oposición a la evaluación diferida, existe la evaluación ansiosa (*eager evaluation*), que es mayormente utilizada en los lenguajes de programación, que evalúa todos los argumentos antes de la invocación de la función y envía los resultados de dichas evaluaciones.

Ejemplo:

Tomando dos funciones predefinidas que se utilizan de la siguiente manera:

“argentina” !! 3

‘e’

En general, **!!** devuelve el elemento de la enésima posición (empezando en 0) de una lista cualquiera.

“argen” ++ “tina”

“argentina”

En general, **++** concatena dos listas cualesquiera.

Si se realiza la invocación

(“argen” ++ “tina”) !! 3

‘e’

El resultado es el esperado, más allá del modo de evaluación. Lo interesante es que gracias al a evaluación diferida, no se llegó a completar la evaluación de la concatenación de las dos listas, debido a que **!!** no lo necesitaba para completar su tarea. Para **!!**, con sólo tener una lista de **4** elementos fue suficiente para devolver lo solicitado, por lo tanto, no le requirió a **++** que realizara toda la concatenación. La lista del segundo argumento de **++** pudo haber sido tan extensa como se desee, sin que su extensión afecte la eficiencia de la consulta, ya que no es evaluada.

El **diferimiento de la evaluación** de una expresión, permite que de acuerdo a cómo esté definida **no haya que evaluarla en su totalidad para llegar al resultado esperado**. En consecuencia las subexpresiones que no fue necesario evaluar podrían haber contenido expresiones infinitas, haber generado bucles interminables o inclusive provocar errores sin que ello interrumpa o afecte la ejecución del programa. Aún sin llegar a esos casos más extremos, el hecho de no requerir evaluar toda la expresión ahorra ejecutar código innecesario con el consiguiente aumento de **eficiencia** y **optimización de recursos**.

Es una propiedad del paradigma que permite utilizarse como herramienta para resolver ciertos problemas, entre los que se destacan las listas infinitas. También permite reutilizar programas, o partes de programas, mucho más fácilmente, sin tener que hacer ajustes que serían imprescindibles en otros lenguajes, y facilita el desarrollo de programas más modulares.

Listas infinitas

Uno de los beneficios de la evaluación diferida consiste en la posibilidad de manipular listas infinitas. Evidentemente, no es posible construir o almacenar un objeto infinito en su totalidad. Sin embargo, se puede construir objetos **potencialmente infinitos,** pieza a pieza, según las necesidades de evaluación.

De esta manera, y teniendo en cuenta que la definición del tipo de dato de una lista no requiere predeterminar un tamaño, las listas pueden ser infinitas, es decir, contener tantos elementos como se requiera sin establecer un máximo. Su

construcción se suele realizar con **funciones recursivas** en las que no se definen casos bases que corten la recursividad y permitan así una invocación recursiva infinita.

Ejemplo:

Considerando la siguiente función:

```
desde :: Int -> [ Int ]
desde n = n : desde ( n+1 )
```

Ante una invocación genera una lista de todos los enteros a partir del argumento enviado, constituyendo una lista infinita. En el caso

```
desde 5
```

se obtiene la lista

```
[ 5, 6, 7, 8.. ]
```

que crece infinitamente, sin terminar nunca.

Pero dependiendo de dónde esté utilizada esta expresión, puede que la lista infinita se genere sólo parcialmente o incluso no se evalúe nunca.

Dada una función que retorna los **n** primeros elementos de una lista:

```
primeros :: Int -> [ Int ] -> [ Int ]
primeros 0    _        = [ ]
primeros n (x:xs)      = x : primeros (n – 1)  xs
```

Si se la invoca como:

```
primeros 1 (desde 5)
[ 5 ]
```

La función **primeros** recibe inicialmente el argumento sin evaluar. Cuando intenta unificarlo con la expresión **x:xs,** recién se la evalúa por primera vez. De esta manera, el argumento

```
desde 5
```

de acuerdo a la definición de la función, y por el mismo mecanismo de la evaluación diferida, se evalúa sólo lo necesario para resolver la unificación, por lo que es reemplazado por

```
5: (desde 6)
```

En la función **primeros** se unifica en la segunda ecuación

```
primeros 1 (5:(desde 6)) = 5: primeros 0 (desde 6)
```

La invocación recursiva de la función **primeros**, al ser evaluada, se unifica por la primera ecuación, que retorna una lista vacía sin llegar a evaluar el argumento que contiene a la lista potencialmente infinita.

Por lo tanto, la respuesta final es la lista formada por un solo elemento, el **5**.

Ejemplo

Una función que devuelve una lista con todos los años bisiestos después de Cristo.

```
bisiestos :: [ Int ]
bisiestos = obteneranios 1
```

Como caso particular, **bisiestos** se trata de una función constante que no recibe argumentos sino que devuelve siempre la misma lista.

```
obteneranios :: Int -> [ Int ]
obteneranios n      | esBisiesto n  = n : obteneranios ( n+4 )
                    | otherwise     = obteneranios ( n+1 )
```

Un año es bisiesto cuando es divisible por cuatro, excepto si también es divisible por 100. Si además es divisible por 400 vuelve a ser bisiesto. (el 1900 no, el 1996 y el 2000 sí)

```
esBisiesto :: Int -> Bool
esBisiesto n        | esDivisible n 400     = True
                    | esDivisible n 100     = False
                    | esDivisible n 4       = True
                    | otherwise             = False
```

La función **esDivisible** invoca a la función predefinida **mod**, que calcula el resto de la división entera.

```
esDivisible :: Int -> Int -> Bool
esDivisible x y  = x `mod` y == 0
```

Ante la invocación:

```
bisiestos
[ 4, 8,.. 1896, 1904,.. 1996, 2000, 2004.. ]
```

Capítulo 6

Funciones de orden superior

- **Funciones de orden superior**
- **Aplicación parcial**
- **Composición de funciones**
- **Expresiones lambda**

Funciones de orden superior

La programación funcional incorpora el concepto de función como **objeto de primera clase**, lo que significa que **las funciones son tratadas como datos** y en consecuencia pueden ser pasadas como parámetros, calculadas, devueltas como resultados, incluidas en una estructura de datos o utilizadas en cálculos más complejos con otros datos.

Los lenguajes funcionales ofrecen generalmente maneras potentes de encapsular abstracciones. Las abstracciones son la clave de la construcción modular y de los programas mantenibles. Un mecanismo poderoso de abstracción disponible en los lenguajes funcionales son las funciones de orden superior.

Las funciones que reciben a otras funciones como parámetros se llaman funciones de orden superior. En otras palabras **una función es de orden superior cuando existe al menos uno de sus argumentos cuyo tipo de dato es función**.

El **tipo de dato de una función** esta determinado por los **tipos de datos de su dominio e imagen**. Si a y b son dos tipos de datos, entonces a->b es el tipo de una función que toma como argumento un elemento de tipo a y devuelve un valor de tipo b.

La utilización de funciones de orden superior proporciona una mayor **flexibilidad** al programador, permite **reutilizar código** y brinda un **mayor nivel de abstracción**. Es una de las características más sobresalientes de los lenguajes funcionales y en algunos ámbitos se considera que es la propiedad que distingue un lenguaje funcional de otro. El uso correcto de las funciones de orden superior puede mejorar substancialmente la estructura y la modularidad de muchos programas.

Ejemplo:

La función **map** permite operar fácilmente con listas. Recibe una función que aplica a cada elemento de la lista, obteniendo una nueva lista con los resultados de dichas evaluaciones.

Al invocarla, se envía como primer argumento una función, en este caso una función que calcula el factorial de un entero y como segundo argumento una lista de enteros

```
map factorial [ 2, 3, 4 ]
```

El resultado obtenido es la lista con los respectivos factoriales.

```
[2, 6, 24 ]
```

La definición de la función es:

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

El primer argumento que recibe es de tipo función, en particular, de una función que va de un tipo genérico a otro tipo genérico **(a->b)**. De esta manera, **map** puede recibir cualquier otra función que respete ese prototipo.

Ejemplo

Una función que dada una función y una lista, devuelve el resultado de aplicar sucesivamente la función sobre todos los elementos:

```
multitarea :: ( a -> a -> a ) -> [ a ] -> a
multitarea f  [ ]        = error “lista vacia”
multitarea f  [ x ]      = x
multitarea f  (x:xs)     = f x ( multitarea f xs )
```

La primera ecuación, especifica el mensaje de error que es desplegado ante la invocación con una lista vacía.

Invocando de la siguiente manera

```
multitarea  (+) [ 2, 4, 1 ]
7
```

Calcula la sumatoria de todos los números de la lista.

Cambiando el tipo de dato de la función y de la lista:

```
multitarea (++) [ “hola”, “que”, “tal” ]
“holaquetal”
```

Permite concatenar todos los elementos de la lista

Una variante, que cambia el orden de evaluación de los elementos de la lista se obtiene reemplazando la última línea por:

```
multitarea f  (x:xs)     = f ( multitarea f xs ) x
```

En este caso, ante la invocación de la sumatoria respondería lo mismo, ya que la suma es conmutativa, mientras que la concatenación resultaría al revés:

multitarea (++) [“hola”, “que”, “tal”]

“talquehola”

Aplicación Parcial

Todas las funciones **devuelven otra función,** cuando son **evaluadas con una menor cantidad de argumentos** de los que tiene definidos.

Los tipos de datos en el prototipo de una función están implícitamente asociados del final hacia el principio.

Ejemplo:

Una función de definida de la siguiente manera

funcion :: Tipo1 -> Tipo2 -> ... -> TipoN-1 -> TipoN

Es interpretada por defecto como si tuviese paréntesis:

funcion :: Tipo1 -> (Tipo2 -> (... -> (TipoN-1 -> TipoN)))

En una función de **n** argumentos, al invocarse con los argumentos **1** al **k**, la función que se retorna tiene su prototipo formado por los tipos de datos **k +1** hasta **n**. En general, la función que devuelve tiene los argumentos que no hayan sido invocados.

Ejemplo:

Al invocar a la función anterior con menos argumentos de los que se supone que recibe:

funcion argumento1

Se obtiene como resultado una nueva función, cuyo tipo de dato es

Tipo2 -> ... -> TipoN-1 -> TipoN

La función resultante se comporta como función en el lugar donde se la ubique. Para que tenga sentido, será evaluada en un contexto donde se le completen los demás argumentos para que devuelva un valor que no sea otra función.

La utilidad principal de este proceso de **evaluación parcial de funciones es** permite crear nuevas funciones a partir de las funciones existentes.

Ejemplo:

La función de suma (**+**) recibe dos valores enteros como argumento y devuelve un entero. Su tipo de dato es:

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

Explicitando la precedencia con paréntesis es equivalente a

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

Por lo tanto, se puede invocar:

5 +

Se obtiene una función cuyo tipo es **Int->Int**, o sea, que recibe un entero y devuelve otro entero. De hecho, esta expresión denota una nueva función, que en caso de aplicarse sobre un entero devuelve dicho entero más **5**

(5 +) 3

8

El mismo caso deja de ser trivial al utilizarlo en un nuevo contexto, como siendo argumento de **map**. La invocación:

map (5 +) [1, 4, 6]

[6, 9, 11]

Ejemplos:

(1+) es la función sucesor que devuelve su argumento más 1.

(1.0/) es la función inverso.

(/2) es la función mitad.

(:[]) es la función que convierte un elemento simple en una lista con un único elemento que lo contiene

Existe un caso especial importante. Una expresión de la forma:

(-e) es interpretada como **negate e**, no como la aplicación parcial de **"-"** que resta el valor de **e** de su argumento.

Su uso está relacionado con el concepto de **currificación**[14], que plantea la utilización de argumentos individuales en vez de estructuras de tipo tuplas en el prototipo de la función. De esta manera, es condición necesaria para que se pueda dar la aplicación parcial. Solo puede evaluarse parcialmente una función de varios parámetros si está currificada.

Ejemplo

Una definición sencilla de una función

multiplicar :: (Int, Int) -> Int

multiplicar (x, y) = x * y

Así definida, **multiplicar** no esta currificada, ya que recibe un sólo argumento, que es una tupla. La versión currificada de la misma función es la siguiente, con dos argumentos simples:

multiplicar :: Int -> Int -> Int

multiplicar x y = x * y

La aplicación parcial de **multiplicar** sólo es posible con la segunda definición.

Invocando

multiplicar 2

Se obtiene una nueva función que calcula el doble de un número entero

[14] En honor a Haskell Curry. (Ver capítulo 1 "Conceptos generales)

Ejemplo

Una función que genera una lista infinita con la sucesión numérica de acuerdo a la función que recibe como argumento y un valor inicial:

```
sucesion :: Num a => ( a -> a ) -> a -> [ a ]
sucesion f  x = x : sucesion f ( f x )
```

La función **sucesion**, que es de orden superior, se puede consultar con un argumento que sea una función parcialmente invocada:

```
sucesion ( multiplicar 2 ) 2
[ 2, 4, 8, 16.. ]
```

La función **multiplicar** completa sus argumentos al ser utilizada dentro del desarrollo de **sucesion**.

Dada la simplicidad de la definición de la función **multiplicar**, la invocación anterior podría haberse realizado, con el mismo resultado, de la forma:

```
sucesion (2* ) 2
```

Composición de funciones

Las funciones se componen en forma similar a las funciones matemáticas. Para ello, existe una función de orden superior que se representa con el signo "punto" (.) y que tiene como argumentos las dos funciones a componer, retornando una nueva función.

Ejemplo

El prototipo de la función está definido como:

```
( . ) :: (c -> b) -> (a -> c) -> a -> b
```

Siendo **f** y **g** dos funciones que se reciben como argumento, la composición **(f.g) x** es equivalente a **f (g x)**

Ejemplo:

Dada una función que permite obtener el último elemento de una cadena de enteros:

```
ultimo :: [ Int ] -> Int
ultimo [ x ]  = x
ultimo (x:xs) = ultimo xs
```

Sea una función que devuelve si un entero es par:

```
esPar :: Int -> Bool
esPar 0 = True
esPar 1 = False
esPar n = esPar ( n-2 )
```

Para calcular si el último elemento de una cadena de enteros es par, se puede componer **esPar** con **ultimo**:

esPar . ultimo

Representa una función cuyo prototipo es

[Int] -> Bool

Al evaluarla con la siguiente lista como argumento

(esPar . ultimo) [28, 3, 5]

Se obtiene como resultado

False

Expresiones lambda

Además de las definiciones de función con nombre, es posible definir y utilizar funciones sin necesidad de darles un nombre explícitamente, mediante las expresiones lambda.

Una expresión lambda tiene una estructura similar a la de una función. De hecho, las ecuaciones típicas que componen una función son formas abreviadas de las expresiones lambda, y todo el desarrollo matemático denominada cálculo lambda está en los fundamentos mismos del paradigma funcional.

En general, una expresión lambda está constituida por una serie de variables, que hace las veces del dominio de la función, y una expresión, que equivale a la imagen de la función.

Ejemplo

Una función lambda tiene la siguiente forma

\ variable1 variable2 .. variableN -> expresion1

La expresión denota una función que toma un número de parámetros, uno por cada variable, produciendo el resultado especificado por expresion1.

Ejemplo

Una función como

inc x = x+1

es equivalente a la expresión lambda

\x -> x+1

Al hacer la invocación

(\x -> x+1) 6

El resultado es

7

Ejemplo

\x y -> x + y

Toma dos argumentos enteros y devuelve su suma. Es una expresión equivalente al operador (+):

```
(\x y -> x + y ) 2 3
5
```

Las expresiones lambda son utilizadas frecuentemente en conjunto con las funciones de orden superior, actuando como argumentos.

Ejemplo

Si se quiere obtener una lista con el cuadrado de los primeros números naturales, una posibilidad, invocando a la función de orden superior, es:

```
map  cuadrado [ 1..5 ]
[ 1, 4, 9, 16, 25 ]
```

Esta solución requiere de la definición de la función cuadrado, que puede estar definida así:

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

Para obtener la misma funcionalidad, la alternativa utilizando una expresión lambda es invocar:

```
map  ( \x -> x * x ) [ 1..5 ]
```

Anexo

El lenguaje Haskell

- **Características generales**
- **Nombres de función**
  - Operadores
  - Identificadores
  - Intercambio de notación
- **Orden de evaluación**
  - Precedencia
  - Asociatividad
  - Valores por defecto
- **Tipos de datos más comunes**
  - Booleanos
  - Enteros
  - Flotantes
  - Caracteres
- **Tipos definidos por el usuario**
  - Sinónimos de tipo
  - Definiciones de tipos de datos
  - Tipos Enumerados
  - Tipos Recursivos
- **Funciones de uso frecuente**

Características generales

En el lenguaje Haskell existe un conjunto de especificaciones y consideraciones que permiten desarrollar las funciones. También son numerosas las funciones estándar predefinidas[15] y los tipos de datos disponibles.

[15] Las funciones básicas están definidas en un archivo denominado *prelude.*

Nombres de función

Existen dos formas de nombrar una función, mediante un identificador (por ejemplo, **sum**, **product** y **fact**) y mediante un símbolo de operador (por ejemplo, * y **+**) El sistema distingue entre los dos tipos según la forma en que estén escritos.

Las funciones tienen notación prefija, es decir que se las declara y se las invoca indicando primero el nombre de función y luego los argumentos, mientras que los símbolos de operadores se utilizan con notación infija, es decir, escribiendo el operador entre ambos argumentos, tanto en la invocación como en la declaración.

Identificadores

Un **identificador** comienza con una letra del alfabeto seguida, opcionalmente, por una secuencia de caracteres, cada uno de los cuales es una letra, un dígito, un apóstrofe (') o un subrayado (_).

Los identificadores que representan funciones o variables deben comenzar por letra minúscula. Con mayúsculas comienzan los nombres de los tipos de datos (Int, Char, etc), las restricciones de contexto (Ord, Eq, etc.) y los identificadores de las funciones constructoras (por ejemplo, las expresiones True y False).

Ejemplo:

Los siguientes son posibles identificadores:

sumatoria f f'' unaFuncion elemento_x

Existen identificadores que son palabras reservadas y no pueden utilizarse como nombres de funciones o variables.

Ejemplo:

Las palabras reservadas son:

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

Operadores

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

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

Además, el carácter (~) también se permite, aunque sólo en la primera posición del nombre. Los nombres de operador que comienzan con (:) son

utilizados para funciones constructoras como los identificadores que comienzan por mayúscula mencionados anteriormente.

Los siguientes símbolos tienen usos especiales:

:: = .. @ \ | <- -> ~ =>

Todos los otros símbolos de operador se pueden utilizar como variables o nombre de función, incluyendo los siguientes:

+ ++ && || <= == /= // . ==> $ @@ -*- \/ /\ ... ?

Intercambio de notación

Se proporcionan dos mecanismos simples para utilizar un identificador de función 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 (`).

Ejemplo:

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

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

Ejemplo:

x + y es equivalente a **(+) x y.**

Orden de evaluación

Cuando se trabajan con símbolos de operador es necesario tener en cuenta la asociatividad y la precedencia:

Precedencia

La expresión **2 * 3 + 4** podría interpretarse como **(2 * 3) + 4** o como **2 * (3 + 4)**. Para resolver la ambigüedad cada operador tiene asignado un valor de precedencia (un entero entre 0 y 9). En una situación como la anterior se comparan los valores de precedencia y se utiliza primero el operador con mayor precedencia (en el *standar prelude* el **+** y el ***** tienen asignados 6 y 7, respectivamente, por lo cual se realiza primero la multiplicación).

Asociatividad

La regla anterior resuelve ambigüedades cuando los símbolos de operador tienen distintos valores de precedencia, sin embargo, la expresión **1 - 2 - 3** puede ser tratada como **(1 - 2) - 3** resultando **-4** o como **1 - (2 - 3)** resultando **2**. Para resolverlo, a cada operador se le puede definir una regla de asociatividad.

Para un símbolo de operador cualquiera, por ejemplo **&**, se podría decir que es:

- Asociativo a la izquierda: si la expresión **x & y & z** se interpreta como si fuera **(x & y) & z**
- Asociativo a la derecha: si la expresión **x & y & z** se interpreta como si fuera **x & (y & z)**
- No asociativo: Si la expresión **x & y & z** se rechaza como un error sintáctico.

En el *standar prelude* el (-) está definido como asociativo a la izquierda, por lo que la expresión **1 - 2 - 3** se trata como **(1 - 2) - 3**.

Valores por defecto

Por defecto, todo símbolo de operador se toma como no-asociativo y con precedencia 9. Estos valores pueden ser modificados mediante una declaración con los siguientes formatos:

infixl digito ops	Para declarar operadores asociativos a la izquierda
infixr digito ops	Para declarar operadores asociativos a la derecha
infix digito ops	Para declarar operadores no asociativos

ops representa una lista de uno o más símbolos de operador separados por comas y **digito** es un entero entre 0 y 9 que asigna una precedencia a cada uno de los operadores de la lista. Si el dígito de precedencia se omite se toma 9 por defecto.

En el *standar prelude* se utilizan las siguientes declaraciones:

```
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 ||

Algunas expresiones y las formas equivalentes siguiendo las declaraciones del *standar prelude:*

Expresión:	Equivalente a:	Motivos
1+2-3	(1 + 2) - 3	(+) y (-) tienen la misma precedencia y son asociativos a la izquierda.
x : ys ++ zs	x : (ys ++ zs)	(:) y (++) tienen la misma precedencia y son asociativos a la derecha.
x == y || z	(x == y) || z	(==) tiene más precedencia que (||)
3 + 4 * 5	3 + (4 * 5)	(*) tiene más precedencia que (+)
y `elem` z:zs	y `elem` (z:zs)	(:) tiene más precedencia que `elem`
12 / 6 / 3	error sintáctico	(/) no es asociativo

La aplicación de funciones tiene más precedencia que cualquier símbolo de operador.

Ejemplo:

f x + g y equivale a **(f x) + (g y)**

f x + 1 es tratada como **(f x) + 1** en lugar de **f (x+1)**

Tipos de datos más comunes

Una parte importante del lenguaje Haskell lo forma el sistema de tipos que es utilizado para detectar errores en expresiones y definiciones de función.

Los principales tipos predefinidos se pueden clasificar en: **tipos simples,** cuyos valores se toman como primitivos, por ejemplo, Enteros, Flotantes, Caracteres y Booleanos; y **tipos compuestos**, cuyos valores se construyen utilizando otros tipos, como listas, tuplas y funciones

El universo de valores es particionado en colecciones organizadas, denominadas **tipos**. Cada tipo tiene asociadas un conjunto de operaciones que tiene el mismo significado para los elementos del mismo tipo.

Ejemplo:

La función **+** se puede aplicar entre enteros pero no entre caracteres o funciones.

Una propiedad importante es que es posible asociar un único tipo a toda expresión bien formada. Esta propiedad hace que el Haskell sea un lenguaje

fuertemente tipado. Como consecuencia, cualquier expresión a la que no se le pueda asociar un tipo es rechazada como incorrecta antes de la evaluación.

Ejemplo:

```
f x = 'A'
g x = x + f x
```

La expresión **'A'** denota el carácter A. Para cualquier valor de **x**, el valor de **f x** es igual al carácter **'A'**, por tanto es de tipo **Char**. Puesto que el **+** es la operación suma entre números, la parte derecha de la definición de **g** no está bien formada, ya que no es posible aplicar **+** sobre un carácter.

El análisis de los escritos puede dividirse en dos fases: **análisis sintáctico**, para chequear la corrección sintáctica de las expresiones y **análisis de tipo**, para chequear que todas las expresiones tienen un tipo correcto.

Booleanos

Se representan por el tipo **Bool** y contienen dos valores: **True** y **False**. El *standard prelude* incluye varias funciones para manipular valores booleanos.

Ejemplo:

x && y	es **True** si y sólo si **x** e **y** son **True**
x || y	es **True** si y sólo si **x** o **y** o ambos son **True**
not x	es el valor booleano opuesto de **x**

Enteros

Representados por el tipo **Int**, se incluyen los enteros positivos y negativos tales como el **-273**, el **0** ó el **383**. Como en muchos sistemas, el rango de los enteros utilizables está restringido. También se puede utilizar el tipo **Integer** que denota enteros sin límites superior ni inferior.

En el *standard prelude* se incluye un amplio conjunto de operadores y funciones que manipulan enteros.

Ejemplo:

+	suma.
*****	multiplicación.
-	substracción.
^	potenciación.
negate	menos unario (la expresión "**-x**" se toma como "**negate x**")
div	división entera
rem	resto de la división entera.

mod	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.
gcd	máximo común divisor.
lcm	mínimo común múltiplo.
abs	valor absoluto
signum	devuelve **-1**, **0** o **1** si el argumento es negativo, cero ó positivo.

Algunas invocaciones y sus resultados:

3 ^ 4	**81**
7 `div` 3	**2**
even 23	**False**
(x `div` y) * y + (x `rem` y)	**x**
7 `rem` 3	**1**
-7 `rem` 3	**-1**
7 `rem` -3	**1**
7 `mod` 3	**1**
-7 `mod` 3	**2**
7 `mod` -3	**-2**
gcd 32 12	**4**
abs (-2)	**2**
signum 12	**1**

Flotantes

Representados por el tipo **Float**, los elementos de este tipo pueden ser utilizados para representar fraccionarios así como cantidades muy largas o muy pequeñas. Sin embargo, tales valores son sólo aproximaciones a un número fijo de dígitos y pueden aparecer errores de redondeo en algunos cálculos que empleen operaciones en punto flotante. Un valor numérico se toma como un flotante cuando incluye un punto en su representación o cuando es demasiado grande para ser representado por un entero.

También se puede utilizar notación científica; por ejemplo **1.0e3** equivale a **1000.0**, mientras que **5.0e-2** equivale a **0.05**.

El *Standard prelude* incluye también múltiples funciones de manipulación de flotantes:

pi, exp, log, sqrt, sin, cos, tan, asin, acos, atan, etc.

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'**. 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.

Ejemplo:

Algunos de los más comunes códigos de escape son:

'\\'	barra invertida
'\''	comilla simple
'\"'	comilla doble
'\n'	salto de línea
'\b' or '\BS'	backspace (espacio atrás)
'\DEL'	borrado
'\t' or '\HT'	tabulador
'\a' or '\BEL'	alarma (campana)
'\f' or '\FF'	alimentación de papel

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 *standard prelude* proporciona las funciones toEnum y fromEnum que permiten realizar la conversión.

Ejemplo:

La siguiente función convierte mayúsculas en minúsculas.

```
mayusc :: Char -> Char
mayusc c = toEnum (fromEnum c - fromEnum 'a' + fromEnum 'A')
```

Tipos definidos por el usuario

Sinónimos de tipo

Los sinónimos de tipo se utilizan para proporcionar abreviaciones para expresiones de tipo aumentando la legibilidad de los programas. Un sinónimo de tipo se define con la palabra reservada **type** y una ecuación con el nuevo nombre y su equivalencia.

Ejemplo:

```
type Nombre = String
type Edad = Integer
type String = [Char]
type Persona = (Nombre, Edad)
```

Esto tipos de datos se pueden utilizar como los propios del lenguaje

```
tocayos :: Persona -> Persona -> Bool
tocayos (nombre2, _ ) (nombre2,_) = nombre1 == nombre2
```

Definiciones de tipos de datos

Aparte del amplio rango de tipos predefinidos, en **Haskell** también se permite definir nuevos tipos de datos mediante la sentencia **data**. La definición de nuevos tipos de datos aumenta la seguridad de los programas ya que el sistema de inferencia de tipos distingue entre los tipos definidos por el usuario y los tipos predefinidos. Se utilizan para construir un nuevo tipo de datos formado a partir de otros.

Ejemplo:

```
data Persona = Pers Nombre Edad
juan::Persona
juan = Pers "Juan Lopez" 23
```

Pers es el contructor del tipo de dato y debe especificarse cuando se desdea descomponer el valor compuesto en sus partes.

Una función que manejen dicho tipos de datos:

```
esJoven:: Persona -> Bool
esJoven (Pers n e) = e < 25
```

También se pueden dar nombres a los campos de un tipo de datos producto:

Ejemplo:

```
data Persona = Pers { nombre::Nombre, edad::Edad }
```

Los nombres de dichos campos sirven como funciones selectoras del valor correspondiente.

```
tocayos:: Persona -> Persona -> Bool
tocayos p1 p2 = nombre p1 == nombre p2
```

Tipos enumerados

Se puede introducir un nuevo tipo de datos enumerando los posibles valores, que luego pueden ser usados para definir funciones simples.

Ejemplo:

```
data Color = Rojo | Verde | Azul
data Temperatura = Frio | Caliente
data Estacion = Primavera | Verano | Otonio | Invierno

tiempo :: Estacion -> Temperatura
tiempo Primavera = Caliente
tiempo Verano = Caliente
tiempo _ = Frio
```

Las diferentes alternativas de casos para un tipo enumerado, pueden contener a su vez otros tipos de datos.

Ejemplo:

```
data Forma = Circulo Float | Rectangulo Float Float
```

El tipo de dato Forma puede ser tanto un círculo, con una única variable de tipo Float para el radio, como un rectángulo, con dos variables, una para la base y otra para la altura. Para calcular el área, se utiliza la siguiente función, que comprende los dos casos posibles.

```
area :: Forma -> Float
area (Circulo radio) = pi * r * r
area (Rectangulo base altura) = base * altura
```

pi es una función constante definida en el *prelude* que retorna el correspondiente valor.

Tipos recursivos

Los tipos de datos pueden autorreferenciarse consiguiendo valores recursivos y también pueden definirse tipos de datos polimórficos.

Ejemplo:

```
data Expr = Literal Integer | Suma Expr Expr | Resta Expr Expr
```

Expr es un nuevo tipo que puede asumir tres valores: un valor llamado **Literal**, con un entero como argumento, un valor llamado **Suma** que incluye a otras dos elementos de tipo **Expr** y el tercero, denominado **Resta**, que también incluye a otras dos valores **Expr**, siendo de esta manera, recursivo.

```
evaluar :: Expr -> Integer
evaluar (Literal n) = n
evaluar (Suma e1 e2) = evaluar e1 + evaluar e2
evaluar (Resta e1 e2) = evaluar e1 - evaluar e2
```

La función **evaluar**, contempla los tres casos posibles y se llama recursivamente para devolver un valor entero con las operaciones matemáticas correspondientes.

Ejemplo:

Se define un tipo que representa árboles binarios:

```
data Arbol a = Hoja a | Rama (Arbol a) (Arbol a)
```

La expresión:

```
a1 = Rama (Rama (Hoja 12) (Rama (Hoja 23) (Hoja 13)) (Hoja 10)
```

es del tipo

```
Arbol Integer
```

Otra posibilidad es que el árbol tuviera en las hojas caracteres, listas de enteros o cualquier otro tipo, incluso árboles.

```
a2 ::Arbol [Integer]
a2 = Rama (Hoja [1,2,3] Hoja [4,5,6])
a3 :: Arbol Char
a3 = Rama (Hoja 'a') (Hoja 'b')
```

Una función que obtiene la lista de nodos hoja de un árbol binario:

```
hojas :: Arbol a -> [a]
hojas (Hoja h) = [h]
hojas (Rama izq der) = hojas izq ++ hojas der
```

Algunas invocaciones:

```
hojas a1
[12, 23, 13, 10]
hojas a2
[[1,2,3], [4,5,6]]
hojas a3
"ab"
```

Funciones de uso frecuente

El *standard prelude* incluye un amplio conjunto de funciones:

Ejemplo:

```
length :: [ a ] -> Int
```

Devuelve el número de elementos de una lista

```
length [1,3,10]
3
```

```
(++) :: [ a ] -> [ a ] -> [ a ]
```

Devuelve la lista resultante de concatenar dos listas

```
[1,3,10] ++ [2,6,5,7]
 [1, 3, 10, 2, 6, 5, 7]
```

concat :: [[a]] -> [a]

Devuelve la lista resultante de concatenar todas las listas que hay en una lista

concat [[1], [2, 3], [ ], [4, 5, 6]]

[1, 2, 3, 4, 5, 6]

map :: (a -> b) -> [a] -> [b]

Devuelve la lista resultante de aplicar una función a todos los elementos de una lista

map even [1, 2, 3, 4]

[False, True, False, False]

filter:: (a -> Bool) -> [a] -> [a]

Devuelve la lista con los elementos de otra lista para los cuales la función que se recibe como argumento devuelve **True**.

filter even [1, 2, 3 , 4]

[2, 4]

(!!):: [a] -> Int -> a

Devuelve el elemento de la posición enésima de una lista, asumiendo que las listas se enumeran desde 0.

“abcde” !! 2

‘c’

take:: Int -> [a] -> [a]

Devuelve una lista con los n primeros elementos de una lista

take 3 [1, 2, 3, 4, 5]

[1, 2, 3]

drop :: Int -> [a] -> [a]

Devuelve una lista sin los n primeros elementos de una lista

drop 3 [1, 2, 3, 4, 5]

[4 , 5]

reverse :: [a] -> [a]

Devuelve una lista invertida

reverse “hola”

“aloh”

elem :: Eq a => a -> [a] -> Bool

Devuelve **True** si un elemento está en una lista, **False** en caso contrario

elem 1 [4,1,2]

True

sum :: Num a => [a] -> a

Devuelve la sumatoria de los números de una lista

sum [4, 1, 2]

7

minimum :: Ord a => [a] -> a

Devuelve el mínimo elemento de una lista

minimum [4, 1, 2]

1

maximum :: Ord a => [a] -> a

Devuelve el máximo elemento de una lista

maximum “hola”

‘o’

Bibliografía

- **ALONSO AMO, F y SEGOVIA PEREZ, F**. *Entornos y Metodologías de Programación*. Paraninfo.
- **GHEZZI, Carlo y JAZAYERI, Mehdi**. *Conceptos de Lenguajes de Programación*. Díaz de los Santos.
- **LABRA G, José E.,** *Introducción al lenguaje Haskell*. Universidad de Oviedo
- **RAVI y SETHI.** *Lenguajes de programación - Conceptos y constructores.* Addison Wesley.
- **SPIGARIOL, Lucas**. *Conceptos Fundamentales de los Paradigmas de Programación.* Apunte CEIT. 2008
- **WATT, David**. *Programming Languajes Concepts and Paradigms*. Prentice Hall.

Fuentes en Internet

- **Sitio oficial de Haskell:**

  www.haskell.org

Índice