Paradigmas de Programación

Unidad 1. Introducción.

- 1.1 paradigmas de programación
- 1.2 clasificación de los paradigmas

Unidad 2. El paradigma orientada a objetos.

- 2.1 datos y lógica
- 2.2 clases y objetos
- 2.3 abstracción y encapsulación
- 2.4 herencia
- 2.5 polimorfismo

Unidad 3. El paradigma funcional

- 3.1 expresiones e instrucciones
- 3.2 funciones puras
- 3.3 inmutabilidad
- 3.4 funciones recursivas
- 3.5 procesamiento de datos y listas

Unidad 4. El paradigma lógico

4.1 expresiones y predicados

4.2 variables y predicados

4.3 argumentos y predicados

4.4 consultas y predicados

4.5 procesamiento de datos y listas

Bibliografía

- Programación en C++ Denzel Pearson
- Functional programming principles Russell Prentce Hall
- Lage programming in depth Cowsell Pearson

Paradigmas.pdf

ANTECEDENTES

Paradigma de programación se refiere a cómo consideramos que un programa debe ser hecho y cuál debería ser su significado. Y así existen dos estilos de paradigmas (a modo muy general).

1. **Imperativo** Un programa está hecho de instrucciones (la lógica del programa) y esas instrucciones deben manipular los datos del programa. Es decir, en el paradigma imperativo existe una clara separación entre instrucciones y datos, las primeras manipulan a las segundas.

Nota Por instrucción entendemos una orden o comando al procesador para que haga algo.

- **Estructurado**

  Secuencia de instrucciones, la primera llamada punto de entrada y hasta una última instrucción. Mover datos, instrucciones matemáticas, lógicas y para el control de flujo, etc.

  Se hace una clara separación entre instrucciones y datos.

- **Procedural**

  No es lo mismo que funcional, aquí el programa de funciones o procedimientos que se combinan entre sí para darle lógica al programa. Aquí existe:

  - Una función principal que contiene el punto de entrada y le da sentido a la lógica general del programa.
  - Y una o más funciones auxiliares que llevan a cabo una pequeña parte del propósito general del programa, y que son invocadas o llamadas por la función principal.

  Normalmente se sigue haciendo esa clara distinción entre instrucciones y datos, pues estos últimos ahora existen como:

  - Variables globales y locales
  - Parámetros de funciones y valores de retorno en funciones

- **Orientado a objetos**

  Está hecho de objetos que interactúan entre sí. Un objeto es un dato, pero es un dato que:

  - Contiene su propia lógica (instrucciones)
  - Y sus propios datos}

  Ya no separa tan claramente las instrucciones y a los datos, sino que ahora las instrucciones son parte de los datos.

2. **Declarativo** No está hecho de instrucciones, sino de expresiones que sirven para manipular información.

   Nota Una instrucción es algo que al ejecutarse no necesariamente regresa un resultado, una expresión en cambio, siempre regresa un resultado.

   Las expresiones son un concepto matemático.

   Son asuntos de alto nivel de abstracción. Por esa razón la programación declarativa casi siempre tiene más sentido matemático y menos de procedimientos.

- Funcional Los programas están hechos como la composición de funciones puras.

- Lógico Los paradigmas tienen forma de argumento construido con:
  - Hechos, reglas y consultas.

ORIENTADO A OBJETOS

En esta segunda unidad estudiamos el paradigma orientado a objetos. El estilo de programación que seguimos bajo este paradigma es el de programación orientada a objetos, o POO por sus siglas. En esta unidad estudiamos los conceptos clave de la POO, que a grandes rasgos son los siguientes:

1. Clases
2. Encapsulamiento
3. Objetos.
4. Herencia.

5. Polimorfismo.

Clases y Objetos

Si bien el enfoque del paradigma POO son los objetos como las entidades primordiales de trabajo, un objeto, al final, sólo es la instancia de una clase. Así que para en verdad comprender la naturaleza de un objeto, primero hay que analizar de qué clase viene y cómo es esa clase.

Clase: la categoría general a la que un conjunto de objetos pertenecen o son instancias.

Un objeto al ser una instancia de una clase, recibe de esta, todo lo relacionado a su aspecto y todo lo relacionado a su comportamiento. Ese aspecto y ese comportamiento es para con el mundo exterior al objeto.

Atributos y métodos

Eso de un objeto se refiere a los atributos en él, algunos se pueden ver exteriormente y otros no. Así que para indicar qué se puede ver y que no en un objeto, definimos atributos de la clase. Y eso del comportamiento se refiere a cómo se interactúa, desde el exterior con el objeto.

Algunas cosas se pueden hacer con un objeto y otras no. En la clase definimos métodos para dejar claro cómo interactuar con el objeto.

Métodos, atributos y diagramas UML

Una buena forma de representar gráficamente a una clase (sin entrar en detalle) es mediante diagramas UML de clase. Un diagrama de clase UML hace algo así:

Nombre de la clase
Atributos
Métodos()

Una versión más detallada de un diagrama de clase lista cada atributo y cada

Nombre de la clase
atributo1 atributon

método de la siguiente manera:

```
método1()
métodon()
```

Supongamos que queremos representar a una clase que acumule a los puntos en un espacio R^2 (bidimensional).

Es claro que P2 = (-6, -3) y P1 = (3, 5). Supongamos que necesitamos ser capaces de desplazar un punto sumándole o restándole algo a cada coordenada. Al menos necesitamos que nuestra clase (a la que llamamos Punto 2D):

- Un atributo x, la coordenada en x
- Un atributo y, la coordenada en y
- Un método al que llamemos desplazar() con dos parámetros “deltaX” y “deltaY” que son los desplazamientos en x e y, respectivamente.

Y el diagrama UML que representa esta situación es el siguiente:

Punto2D
- x -y
- desplazar(deltaX, deltaY)

Y así, P1 tendría el atributo en x = 3, el atributo en y = 5. P2 tiene el atributo x = -6 y el atributo en y = 3. Y luego, si debemos desplazar a P a una posición P1 = (-5, 6) habría que sumar `deltaX` - 6 a la coordenada en x, y el `deltaY` = 1a la coordenada en y.

Clases, atributos, métodos y objetos en Ruby

Todo lo anterior está muy bonito, pero son puras palabras y dibujitos, ahora pongámoslo en práctica usando a Ruby como nuestro `lenguaje` de referencia.

- Primero debemos definir la clase, que en Ruby se hace:

```
class Punto2D
    #Métodos
```

```
end
```

En Ruby, las clases tienen un nombre tal que su primer letra es mayúscula.

Y todo lo que existe entre `class` Punto2D y `end` son los métodos de la clase.

- Un método se define así:

```
def nombreMétodo(param1, param2, paramn)
    #instrucciones
    return algo
end
```

Así que nuestra clase Punto2D en Ruby, una primera definición es:

```
class Punto2D
    def desplazar(deltaX, deltaY)
        @x += deltaX #atributo de instancia
        @y += deltaY
    end
end
```

Todo está muy bien, pero ¿Qué es eso de @x y @y? Y en todo caso ¿En dónde quedaron los atributos x e y de la clase? Debemos conocer que en Ruby una clase puede definir dos tipos de atributos para un objeto que se instancie de ella.

- Atributos de clase
- Atributos de instancia

Los `atributos de clase` son más bien como las variables globales que toda instancia de la clase puede ver (y por "ver" nos referimos a leer y/o escribir). Estos atributos son únicos para todas las instancias. Los atributos de clase en realidad no nos interesa.

Los atributos de instancia, en cambio, sí nos interesan. Estos atributos son únicos para cada instancia, es decir, cada instancia tiene su propia versión de un atributo de instancia.

En Ruby un atributo de clase se define como `@@nombreAtributo`. Mientras que los atributos de instancia se definen como `@nombreAtributo`.

Y en Ruby ¿Dónde definimos un atributo? donde se nos dé la gana.

Aquel método que de pronto utilice un atributo por primera vez en la clase la está definiendo implícitamente.

Métodos Constructores

Aunque es verdad que en Ruby a un atributo de clase le basta con aparecer en alguna instrucción en algún método para quedar finalmente definido, no es la mejor idea. La razón de esto, tiene que ver con su valor inicial. Por ejemplo, en el método desplazar() de la clase Punto2D.

```
def desplazar(deltaX, deltaY)
    @x += deltaX
    @y += deltaY
end
```

Exactamente ¿A qué valor previo de x :y a que valor, pero de y se suma deltaX y deltaY respectiva/e? La respuesta es que no existe un valor previo para ninguna de las dos, o no uno que nosotros controlemos. Pero para eso llegan los métodos constructores.

Un método constructor es un método que es invocado o llamado por Ruby cuando un objeto se instancia.

Instanciar: crear, dar lugar a un nuevo objeto.

Un método constructor sólo es llamado implícitamente al instanciar un objeto, no puede llamarse explícitamente.

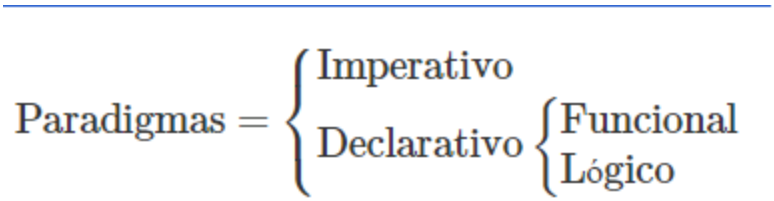
¿Para que nos servirá un método constructor?

Para asegurarnos de que al instanciar un objeto, sus atributos posean, ahora sí, un valor inicial correcto. Y en Ruby los constructores siempre se llaman initialize(), y pueden o no, tomar parámetros de entrada.

```
def initialize(param1, param2, paramn)
    #inicializaciones aquí
end
```

PROGRAMACIÓN FUNCIONAL

Ya se mencionó que existen dos paradigmas básicos de programación, el imperativo, el mas utilizado y posiblemente el primer tipo de paradigma de programación con el que nos hemos encontrado, y el declarativo, uno no muy común aunque con muchos detalles y aspectos que luego son adoptados en la programación imperativa. El siguiente cuadro sinóptico ilustra la categorización de los paradigmas de programación.



El paradigma `imperativo` en resumen, establece que un programa está hecho de instrucciones que sirven primordialmente para modificar el estado de un programa.

Luego tenemos al paradigma `declarativo` que establece que un programa está construido con expresiones, no con instrucciones.

Y luego tenemos el *paradigma de programación* `funcional`, el primero de dos paradigmas de programación declarativos que estudiamos en este curso.

Pero exactamente **¿Qué propone el paradigma de programación funcional?** Propone lo siguiente:

- Un programa está construido como una combinación de funciones puras.

Es decir que el paradigma de programación funcional utiliza funciones puras a manera de expresiones. Este es el principio básico de la programación funcional, sin embargo existen muchos mas detalles detrás de la programación funcional que debemos revisar con detalle. Las siguientes secciones nos presentan estos detalles con finura.

El concepto de estado en un programa

Un concepto fundamental para distinguir entre la programación funcional en comparación con la programación imperativa es el del *estado de un programa*. ¿A que le llamamos ***estado en un programa*** en la programación imperativa? El estado de un programa es la descripción explícita de como van las cosas hasta el momento y en cierto contexto.

De acuerdo con lo anterior la forma mas común de mantener el estado de un programa es mediante variables, ya sean locales a una función o globales en todo el programa, y a estas variable las llamamos ***variables de estado***. Por otro lado, una variable en general puede ser de dos tipos:

1. Variable ***mutable***: puede cambiar su valor tantas veces como sea necesario a lo largo del programa.
2. Variable ***inmutable***: una vez asignado su valor inicial, no puede cambiar a otro valor, mantiene su valor inicial a lo largo del programa.

En la programación imperativa el estado de un programa normalmente se mantiene en una variable mutable, pues es más sencillo así. La razón de esto es que una única variable es todo lo que basta para que cualquier proceso o hijo en el programa verifique el estado actual de las cosas. Pero esto no es verdad para todos, porque a la programación funcional esto no le agrada.

A la programación funcional no le agradan las variables mutables

Las variables `mutables` como método para mantener el estado de un programa tiene definitivamente sus ventajas, la principal de ellas es que es mas simple que

todo el estado de un programa sea accesible desde una única variable. Pero también tiene su desventaja:

1. El mantenimiento y seguimiento del programa se hace mas complejo entre mas variables mutables son introducidas.
2. Es fácil caer en errores o al menos en un programa con un comportamiento extraño cuando no se tiene cuidado en el momento en que un proceso altera el valor en alguna variable mutable.
3. Y todo esto es aún peor cuando el programa se ejecuta en un contexto de paralelismo o multiprocesamiento.

Por supuesto que todos estas particularidades se pueden controlar, por ejemplo usando candados, semáforos, etc. Sin embargo todo esto al final se resume en un programa que implica cada vez mas complejidad, complejidad que debe ser atendida por el mismo programador al diseñar su programa. En eso llega la programación funcional con una idea muy atractiva:

- La programación funcional considera que toda esta complejidad puede ser evitada o al menos minimizada si en primer lugar evitamos que el programa utilice variables mutables.

Vaya que esto se escucha extremo: un programa no debe utilizar variables mutables ¿como es esto posible? Resulta que es totalmente posible, pero para ello debemos cambiar nuestro punto de vista imperativo de las cosas.

Funciones puras

Entremos en detalles de programación funcional. La programación funcional es una forma de programación declarativa con la particularidad de que en la programación funcional las expresiones con las que se construye un programa son `funciones puras`. Así que ¿qué es una función pura? La siguiente definición es desde un punto de vista de un programador:

Una función pura es una función que:

1. Toma una cantidad específica de parámetros de entrada.
2. Produce un único resultado que solo depende de esos argumentos de entrada.

Esta definición de lo que es una función pura tiene algunas implicaciones de las que se debe estar consciente al momento de implementar programas que se digan funcionales. Vale la pena repasar algunas de estas implicaciones con detalle.

Las funciones puras no mantienen estados

Luego, para evitar mantener estado en un programa debemos evitar:

- Usar variables globales definitivamente.
- Usar ***variables mutables*** a lo largo de una función.

Pero ¿Qué es una variable mutable? Digamos que en un programa imperativo existen dos tipos de variables:

1. Variables mutables, que son variables a las que se les puede reasignar su valor tantas veces como sea necesario.
2. Variables inmutables, que son variables que, una vez que se les asigne un valor inicial, ya no será posible reasignar tal valor. Mantienen su valor inicial a lo largo del programa.

El estado de un programa imperativo siempre existirá en variables mutables, pues el estado de un programa cambia conforme el proceso evoluciona, por lo que si queremos evitar que nuestro programa tenga estado lo primero que debemos evitar es el uso de variables mutables.

Las funciones puras no utilizan variables globales

Por otro lado tenemos las variables globales, variables que de existir en un programa funcional deberían ser inmutables, claro está, pero **¿Para qué una variable global en un programa hecho de funciones puras que no la usarían?** Recordemos que una función pura produce un resultado que solo depende de sus parámetros de entrada y de nada mas, lo que deja fuera la posibilidad de utilizar una variable global como parte del procesamiento que lleva a cabo la función.

Las funciones puras no afectan sus parámetros

En realidad este detalle es orto caso mas de que una función pura no tiene efectos secundarios, pero vale la pena analizarle con detalle. Esto se debe a las siguientes razones:

- De una función pura esperamos un resultado.
- Ese resultado lo produce la función usando solo sus parámetros de entrada.
- Así que los parámetros de entrada son solo de entrada y nunca de salida, no se deben utilizar como depósitos para resultados extras.
- No se supone que cambiemos el valor de un parámetro porque eso sería como un objeto mutable, algo indeseable para la programación funcional.
- Y es que al final un resultado extra en un parámetro sería un efecto secundario, algo totalmente indeseable en programación funcional.

Un programa funcional no utiliza IF-THEN sin ELSE

Primero que nada regresemos a aquello de que un programa funcional se compone de expresiones. Sabemos que los lenguajes de programación, sea el lenguaje que sea, ofrecen al programador un conjunto de instrucciones, (remarquemos aquello de instrucciones, que no expresiones) para que desarrolle el programa que desea.

Esto nos haría pensar que por lo tanto nunca podremos desarrollar programas funcionales porque con lo único que contamos es con instrucciones y no con expresiones de esas que necesitamos en un programa funcional. Esto es verdad hasta cierto punto, sin embargo sepamos que:

- Toda instrucción que regrese un resultado se puede considerar une expresión.

Así que mientras utilicemos instrucciones que regresan resultados estaremos mas que bien. Y afortunadamente para nosotros la mayoría de las instrucciones más básicas en casi cualquier lenguaje de programación regresan un resultado, y lo que es mejor, la mayoría de ellas no tienen efectos secundarios. Así que estamos en terreno seguro con ellas. Existen, sin embargo, algunas instrucciones que no necesariamente cumplen con esto.

Un programa funcional no utiliza instrucciones de ciclo

De todos los efectos que tiene la programación funcional sobre la psique de un desarrollador demasiado cómodo con la programación imperativa, el mas notorio es aquel debido al hecho de que ***en la programación funcional no se deben utilizar instrucciones de ciclo***, ni para ciclos FOR, WHILE o DO-WHILE. La razón de esto es sencilla:

- Toda instrucción de ciclo requiere (al menos) una variable de estado para controlarse a si misma.

Por ejemplo un ciclo FOR necesariamente requiere un contador de iteraciones, mientras que un ciclo WHILE o un DO-WHILE requieren una variable para comparar una condición de paro para el ciclo. En ambos casos, el contador de iteraciones y la variable para comparar la condición de paro tenemos una variable mutable, una variable de de estado. Y como ya lo conocemos, este tipo de variables son indeseables en un programa funcional.

Un programa funcional utiliza funciones recursivas en lugar de ciclos

Las instrucciones de ciclos son elementos de programación imperativa demasiado importantes como para simplemente no recurrir a ellos, pues siempre que el programa requiera repetición en la ejecución de instrucciones lo primero que se viene a la mente de todo desarrollador son los ciclos, las instrucciones de ciclo. Pero las instrucciones de ciclo no son la única manera de lograr repetición de instrucciones en un programa, existe otro elementos no demasiado popular en la programación imperativa: la recursividad. **Y ¿qué es la recursividad?** Es algo que definiremos de la siguiente manera:

> La recursividad es una técnica para resolver un problema a partir de dividirlo una y otra vez en casos mas simples hasta

> que, eventualmente el problema no pueda ser simplificado mas, pues se llegó al caso mas simple del mismo.

La recursividad trasera en detalle

Debemos remarcar que en la recursividad trasera toda operación que la función recursiva debe llevar acabo debe ocurrir antes del llamado recursivo, y que, de hecho el resultado de esas operaciones (al que llamaremos *resultado acumulativo*) es un parámetro de la misma función recursiva. Esto nos conduce a concluir lo siguiente:

1. El caso base, que es el que termina la cadena de llamadas recursivos debe arrojar el resultado final de la función, resultado que ya viene en ese parámetro acumulativo que mencionamos antes.
2. El caso recursivo por lo tanto solo se encarga de acumular resultados para formar el nuevo parámetro acumulativo que pasa al llamado recursivo.
3. Y nunca olvidemos que el resultado del llamado recursivo es ya el resultado que la misma función arrojará al final.

Estos tres puntos nos ofrecen todo lo que debemos conocer para desarrollar una función recursiva por atrás, o con recursividad trasera.

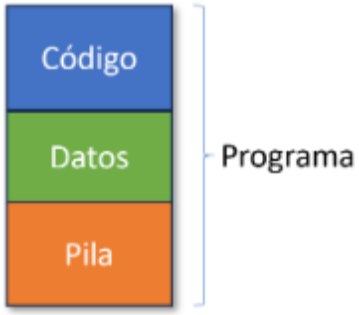
La importancia de las funciones con recursividad trasera

Recordemos que a muy bajo nivel, un programa que se ejecuta se carga en memoria, y al hacerlo utiliza la menos tres secciones:

1. El ***código*** del programa, es decir las instrucciones.
2. Los ***datos globales*** del programa, por ejemplo las variables globales que pueden estar inicializadas o no.
3. Y la ***pila*** del programa, la parte del programa usada para manipular todas las funciones en las que se divide el programa.

La siguiente figura ilustra esta segmentación de un programa al cargarse en memoria.

> Existe una cuarta sección o parte de un programa conocido como heap, una zona del programa en donde se almacena exclusivamente las variables y datos dinámicos del programa. Normalmente el heap de un programa se aloja en la misma pila del mismo.



Un programa, por lo regular, utiliza una *función principal* que lleva a cabo el control del proceso completo, y una o mas *funciones auxiliares* que son llamadas por la función principal para que lleven a cabo una parte del proceso completo. Ahora, siempre que invocamos una función, será necesario asignarle un espacio en la pila del programa conocido como ***marco de pila*** (*stack frame*, en inglés). El marco de pila es un espacio en la pila del programa en donde se almacena toda la información que la función requiere para ejecutarse (copias de los parámetros, variables locales) además de toda la información que la función requiere para regresar al programa que le mandó "llamar" originalmente (la dirección de retorno), una vez que termine de ejecutarse

> El marco de pila de una función se crea en la pila del programa al llamar a la función y se elimina de la pila cuando la función termina su ejecución.

Un desbordamiento de pila es fatal para nuestro programa en ejecución, pues normalmente el S.O termina su ejecución con un resultado de error. Pero ¿que

tiene todo esto que ver con aquello de las funciones recursivas? Una función recursiva es una que llama a otra función, solo que en ambos casos se trata de la misma función. El efecto final es el mismo, estaremos apilando marcos de pila en la pila del programa. Entonces, si una función recursiva se llama a si misma n veces aún tendremos una profundidad de anidamiento de n marcos de pila y aún corremos riesgo de desbordar nuestra pila, Y a menos que n sea mas bien pequeño, definitivamente corremos riesgo de saturar la preciada pila.

Pero en eso llegan los lenguajes de mas alto nivel con un detalle especial: muchos lenguajes de alto nivel utilizan un ***optimizador de manejo de pila***, este componente es capaz de ***optimizar el manejo de pila de una función con recursividad trasera para que nunca desborde la pila, aún si la profundidad de anidamiento es muy grande***.

Nótese el *de mas alto nivel* cuando hablamos de lenguajes de programación en la afirmación anterior. Por ejemplo Scala, Clojure y la mayoría de los Lisp modernos, Elixir, Elm, entro otros contienen esta característica. En estos lenguajes, ***cuando se implementa una función con recursividad trasera es posible pedirle al lenguaje que produzca una versión de esta misma función que nunca caerá en desbordamientos de pila***, lo que resulta genial. Pero nótese que:

> Solo se puede optimizar una función recursiva cuando esta implementa recursividad trasera

Esto implica que una función recursiva que ha sido implementada usando recursividad no trasera no puede ser optimizada, solo aquellas que usan recursividad trasera. He allí la razón por la que se recomienda el uso de recursividad trasera antes que el de recursividad no trasera, todo se debe a una posible optimización en el uso de la pila que el mismo lenguaje puede llevar a cabo por nosotros.

Las funciones son objetos de orden superior

Ahora pasemos a otro aspecto clave detrás de la programación funcional: en un programa funcional ***las funciones son tanto instrucciones como datos***. Con esto

nos referimos al hecho de que si hacemos una equiparación con la programación imperativa en donde los programas están hechos de instrucciones y los datos sobre las que estas operan, en un programa funcional las funciones son tanto equivalentes a las instrucciones como a también a los datos. Es decir que las funciones (puras) lo son todo en un programa funcional. Mas específicamente afirmamos lo siguiente:

> Una función es un objeto de orden superior cuando una función puede hacer lo siguiente :
>
> 1. Una función puede ser asignada como valor a una variable.
> 2. Pasar como argumento de otra función.
> 3. Regresar como valor de retorno de otra función.

Sintaxis básica de Haskell

Hola mundo:

```
main :: IO ()
main = putStrLn "Hola, mundo!"
```

En Haskell, `main` es la función principal que se ejecuta cuando se ejecuta el programa. En este ejemplo, `main` utiliza `putStrLn` para imprimir "Hola, mundo!" en la consola. La expresión `:: IO ()` indica que `main` es una acción de entrada/salida que no devuelve ningún valor.

Función simple:

```
haskellCopy code
-- Definición de una función para sumar dos números
suma :: Int -> Int -> Int
suma x y = x + y
```

Listas:

```
haskellCopy code
-- Definición de una lista de números
numeros :: [Int]
numeros = [1, 2, 3, 4, 5]

-- Accediendo al primer elemento de la lista
primerNumero :: Int
primerNumero = head numeros
```

Funciones de orden superior:

```
haskellCopy code
-- Función de orden superior que suma 1 a cada elemento de un
a lista
sumarUnoATodos :: [Int] -> [Int]
sumarUnoATodos lista = map (\x -> x + 1) lista
```

Funciones anónimas:

```
haskellCopy code
-- Función anónima para sumar 1 a un número
sumarUno :: Int -> Int
sumarUno = \x -> x + 1
```

Aplicación parcial:

```
haskellCopy code
-- Función de suma parcial que suma 5 a un número
sumarCinco :: Int -> Int
sumarCinco = suma 5
```

Funciones monádicas:

```
haskellCopy code
-- Impresión de un mensaje en la consola
imprimirMensaje :: IO ()
imprimirMensaje = putStrLn "Hola, Haskell!"
```

Estos ejemplos muestran la sintaxis básica de Haskell y cómo se pueden utilizar diferentes conceptos como funciones, listas, funciones de orden superior, funciones anónimas, aplicación parcial y funciones monádicas en el código Haskell.

Funciones como valor de una variable:

En Haskell, las funciones pueden tratarse como valores asignables a variables. Esto significa que puedes definir una función y luego asignarla a otra variable, y esa variable se comportará de la misma manera que la función original.

```
haskellCopy code
-- Funciones como valor de una variable
add :: Int -> Int -> Int
add x y = x + y

addFunction :: Int -> Int -> Int
addFunction = add
```

Funciones como parámetro de otra función:

En Haskell, las funciones pueden tomar otras funciones como argumentos. Esto permite crear funciones de orden superior, que son funciones que pueden operar sobre otras funciones.

```
haskellCopy code
-- Funciones como parámetro de otra función
```

```
applyOperation :: (Int -> Int -> Int) -> Int -> Int -> Int
applyOperation operation x y = operation x y
```

Funciones que regresan otras funciones:

En Haskell, las funciones pueden devolver otras funciones como resultado. Esto es útil para crear funciones más específicas y reutilizables, o para aplicar técnicas como el currying.

```
haskellCopy code
-- Funciones que regresan otras funciones
multiplyBy :: Int -> Int -> Int
multiplyBy factor = \x -> x * factor
```

Funciones aplicadas parcialmente:

La aplicación parcial de funciones en Haskell implica proporcionar solo algunos de los argumentos a una función en lugar de todos. Esto crea una nueva función que espera los argumentos restantes.

```
haskellCopy code
-- Funciones aplicadas parcialmente
addOne :: Int -> Int
addOne = add 1
```

Funciones anónimas:

En Haskell, las funciones anónimas se definen utilizando la sintaxis `\argumentos -> cuerpo`. Estas funciones no necesitan un nombre y son útiles cuando solo se necesitan localmente.

```
haskellCopy code
-- Funciones anónimas
```

```
greet :: String -> IO ()
greet = \name -> putStrLn $ "Hello, " ++ name ++ "!"
```