**Tarea 1**

**¿Qué es un paradigma?**

Como paradigma denominamos **todo aquel modelo, patrón o ejemplo que debe seguirse en determinada situación**. La palabra, como tal, proviene del griego παράδειγμα (parádeigma).

En un sentido amplio, se refiere a una teoría o conjunto de teorías que sirve de modelo a seguir para resolver problemas o situaciones determinadas que se planteen.

Un paradigma de programación se refiere a la forma en que un programador o un conjunto de programadores dan solución a uno o varios problemas claramente definidos. En este sentido, representa una manera particular de ofrecer soluciones.

**¿Cuáles son los Paradigmas de los Lenguajes de Programación?**

El **paradigma por procedimientos**, es tal vez el más conocido y utilizado en el proceso de programación, donde los programas se desarrollan a través de procedimientos. Pascal C y BASIC son tres de los lenguajes imperativos más importantes. La palabra latina imperare significa "dar instrucciones". El paradigma se inició al principio del año 1950 cuando los diseñadores reconocieron que las variables y los comandos o instrucciones de asignación constituían una simple pero útil abstracción del acceso a memoria y actualización del conjunto de instrucciones máquina. Debido a la estrecha relación con la arquitectura de la máquina, los lenguajes de programación imperativa pueden ser implementados muy eficientemente, al menos en principio.  
  
El **paradigma imperativo** aún tiene cierto dominio en la actualidad. Una buena parte del software actual ha sido desarrollado y escrito en lenguajes imperativos. La gran mayoría de programadores profesionales son principalmente o exclusivamente programadores imperativos (Hay que añadir que los paradigmas de la programación concurrente y orientada al objeto son en realidad sub-paradigmas de la programación imperativa, así que sus adeptos también son programadores imperativos).  
  
El **paradigma declarativo o paradigma de programación lógica** se basa en el hecho que un programa implementa una relación antes que una correspondencia. Debido a que las relaciones son más generales que las correspondencias (identificador - dirección de memoria), la programación lógica es potencialmente de más alto nivel que la programación funcional o la imperativa. El lenguaje más popular enmarcado dentro de este paradigma es el lenguaje PROLOG. El auge del paradigma declarativo se debe a que el área de la lógica formal de las matemáticas ofrece un sencillo algoritmo de resolución de problemas adecuado para, usarse en un sistema de programación declarativo de propósito general.  
  
Si la programación imperativa se caracteriza por el uso de variables, comandos y procedimientos, la programación funcional se caracteriza por el uso de expresiones y funciones. Un programa dentro del **paradigma funcional**, es una función o un grupo de funciones compuestas por funciones más simples estableciéndose que una función puede llamar a otra, o el resultado de una función puede ser usado como argumento de otra función. El lenguaje por excelencia ubicado dentro de este paradigma es el LISP. Por ejemplo si se desea obtener la nota promedio de un alumno podría construirse una función promedio la cual se obtendría a partir de otras funciones más simples: una (sumar) la cual obtiene la suma de las entradas de la lista, otra (contar) la cual cuenta el número de entradas de la lista y la tercera (dividir) que obtiene el cociente de los valores anteriores, su sintaxis será:  
  
(dividir (sumar notas) (contar notas))  
  
Obsérvese que la estructura anidada refleja el hecho de que la función dividir actúa sobre los resultados de suma y contar.  
  
**El paradigma orientado a objetos**, se basa en los conceptos de objetos y clases de objetos. Un objeto es una variable equipada con un conjunto de operaciones que le pertenecen o están definidas para ellos. El paradigma orientado a objetos actualmente es el paradigma más popular y día a día los programadores, estudiantes y profesionales tratan de tomar algún curso que tenga que ver con este paradigma, podría decirse, que programar orientado a objetos está de moda.

Alrededor de 1970 David Parnas planteó el ocultamiento de la información como una solución al problema de gerenciar grandes proyectos software. Su idea fue encapsular cada variable global en un módulo con un grupo de operaciones (al igual que los procedimientos y las funciones) que permitan tener un acceso directo a la variable. Otros módulos pueden acceder a la variable sólo indirectamente, llamando a estas operaciones. Hoy se usa el término objeto para tales módulos o variables encapsuladas a sí mismas. Lenguajes imperativos como Pascal Y C han sido modificados (o añadidos) para que soporten el paradigma orientado a objetos para dar Delphi en el caso de Pascal y C++ en el caso de C.

Una de las bondades importantes de los lenguajes orientados a objetos es que las definiciones de los objetos pueden usarse una y otra vez para construir múltiples objetos con las mismas propiedades o modificarse para construir nuevos objetos con propiedades similares, pero no exactamente iguales.

Pero que es exactamente un lenguaje orientado a objetos- Los siguientes conceptos señalan las características generalmente aceptadas acerca de los lenguajes orientados a objetos.

* Objetos y clases son obviamente los conceptos fundamentales. Una clase es un conjunto de objetos que comparten las mismas operaciones.
* Objetos (o al menos referencia a objetos) deben ser valores de la clase base. Así, cualquier operación puede tomar un objeto como un argumento y puede devolver un objeto como resultado. De esta manera el concepto de clase de objetos está relacionado con el concepto de tipo de dato.
* Herencia es también vista como un concepto clave dentro del mundo de los objetos. En este contexto, la herencia es la habilidad para organizar las clases de objetos en una jerarquía de subclases y superclases y las operaciones dadas para una clase se pueden aplicar a los objetos de la subclase.

**Tarea 2**

**Definición de: lenguaje de programación**

Un lenguaje de programación consiste en un conjunto de órdenes o comandos que describen el proceso deseado. Cada lenguaje tiene sus instrucciones y enunciados verbales propios, que se combinan para formar los programas de cómputo. Los lenguajes de programación no son aplicaciones, sino herramientas que permiten construir y adecuar aplicaciones. Le permiten a un programador poder expresar el procesamiento de datos y sus estructuras en la computadora.

**Clasificación de los lenguajes de alto nivel**

* Lenguajes imperativos o procedurales. Estos lenguajes se fundamentan en el uso de variables para almacenar valores y el uso de instrucciones que indican las operaciones a realizar sobre los datos. La mayoría de los lenguajes de alto nivel son de este tipo.
* Lenguajes declarativos. En este caso, el proceso por el cual se ejecuta el programa no aparece de forma explícita. Los programas se construyen mediante la definición de funciones (lenguajes funcionales como Lisp) o expresiones lógicas que indican las relaciones entre determinadas estructuras de datos (lenguajes deprogramación lógica como Prolog).
* Lenguajes orientados a objetos. El diseño de los programas se centra más en los datos y su estructura. Los programas consisten en descripciones de unidades denominadas objetos que encapsulan los datos y las operaciones que actúan sobre ellos. Uno de los lenguajes más usados en esta filosofía es el C++.
* Lenguajes orientados al problema. Este tipo de lenguajes están diseñados para problemas específicos, principalmente de gestión. Los programas están formados por sentencias que indican qué se quiere hacer. Generalmente, suelen ser generadores de aplicaciones que permiten cierta automatización de la tarea de desarrollo de software de gestión.

**Ejemplos de lenguajes:**

* Interpretados: Los lenguajes interpretados más famosos en la actualidad son PHP, ASP, Perl, Python, TCL y Bash.
* Compilados: Pascal, C, C++.
* Orientados a objetos: Java, Ruby, Visual Basic.
* Funcionales: Los lenguajes funcionales híbridos más conocidos son Scala, Lisp, Clojure, Scheme, Ocaml, SAP y Standard ML.

**Tarea 3: Prolog**

**Simbolos**

* **X is Y** %unificación con evaluación.
* **X = Y** %unificación simbólica
* **X=:=Y** %comparación con evaluación
* **X == Y** %comparación simbólica.

|  |  |
| --- | --- |
| Simbolo | significado |
| + | suma |
| - | Resta |
| \* | Multiplication |
| \*\* | potencia |
| / | Division real |
| // | Division entera |
| mod | Resto de division entera |
| abs | Valor absoluto |
| ^ | potencia |
| max | Valor maximo de dos |
| min | Valor minimo de dos |

* **Sintaxis:**

Un programa Prolog es un conjunto de cláusulas de Horn (hechos y reglas) donde:

* + - Todas las clausulas acaban con “.”
    - El símbolo <-- se escribe como “:-“ que se lee como si (condicional). En los hechos no es necesario (basta poner el ´átomo y “.”).
    - Los nombres de predicado y función siguen las pautas habituales de los identificadores, pero comienzan con “minúscula”.
    - Los nombres de variable también siguen esas pautas, pero comienzan por mayúscula.
    - Los atomos en el cuerpo de las reglas se separan mediante comas “,” que se leen como “y”.
    - Ejecutar un programa no tiene sentido como tal (en principio). Lo que se hace es plantear objetivos a resolver.
* **Esqueleto de los programas:**

Existen dos tipos de cláusulas: Hechos y Reglas. Una regla es del tipo:

Cabeza :- Cuerpo.

y se lee como "La cabeza es verdad si el cuerpo es verdad". El cuerpo de una regla consiste en llamadas a predicados, que son llamados los **objetivos** de las reglas. El predicado ,/2 (es decir, un operador de paridad 2 (que recibe 2 argumentos) y de nombre , ) denota conjunción de objetivos, y el operador ;/2 denota disyunción. Conjunciones y disyunciones pueden sólo aparecer en el cuerpo, no en la cabeza de la regla.

Las cláusulas sin cuerpo (es decir, antecedente) son llamados **hechos** porque siempre son ciertos. Un ejemplo de un hecho es:

gato(tom).

* **Aplicación a descargar para editar y ejecutar:**

Programa para editar: SWI-Prolog editor

Para crear un ejecutable existe el predicado qsave\_program/2. Este predicado graba el estado actual del Prolog en el fichero que le indiquemos. La forma de crear el ejecutable será: en primer lugar, compilar el programa en Prolog y, en segundo lugar, llamar al predicado qsave\_program/2. El ejecutable creado se deberá acompañar siempre de las librerías adecuadas. En concreto, para Windows, cuando se lleve el ejecutable a otro ordenador, habrá que copiar también todos los ficheros con extensión dll que se encuentran en el directorio bin del SWI-Prolog.

* **a qué tipo de paradigma pertenece:**

Prolog se enmarca en el paradigma de los lenguajes lógicos y declarativos.

Se basa en la definición de reglas lógicas para luego, a través de un motor de inferencias lógicas, responder preguntas planteadas al sistema y así resolver los problemas. Este paradigma está dentro del paradigma de [programación declarativa](https://es.wikipedia.org/wiki/Programaci%C3%B3n_declarativa) junto con el funcional.  
Sus campos de aplicación principales son el área de Sistemas Expertos, e Inteligencia Artificial. La mayoría de los lenguajes de programación lógica se basan en la teoría lógica de primer orden, aunque también incorporan algunos comportamientos de orden superior como la lógica difusa.  
Por su esencia declarativa, un programa lógico no tiene un algoritmo que indique los pasos que detallen la manera de llegar a un resultado, sino que es el sistema internamente el que proporciona la secuencia de control. No existe el concepto de asignación de variables, sino el de unificación.

* **Un ejemplo de un programa escrito en ese lenguaje**

*% La sintaxis es fibonacci(N, F) -> Término N de la sucesión (el resultado se guarda en F).*

fibonacci(0, 0) :-!.

fibonacci(1, 1) :-!.

fibonacci(N ,F) :-N1 is N - 1, fibonacci(N1, F1),N2 is N - 2, fibonacci(N2, F2), F is F1 + F2.

*%el fibonacci se llama recursivamente dejando el resultado en F.*

**Tarea 4: Python**

**Símbolos**

|  |  |
| --- | --- |
| Simbolo | significado |
| + | suma |
| - | Resta |
| - | negación |
| \* | Multiplication |
| \*\* | potencia |
| / | Division real |
| // | Division entera |
| % | Resto de division entera(modulo) |
| == | Es igual que |
| != | Es distinto que |
| < | Es menor que |
| <= | Es menor o igual que |
| > | Es mayor que |
| >= | Es mayor o igual que |

**Sintaxis**

* Comentarios: detrás de #
* Asignación: se usa=

Ej:

>>>a=3+2J

>>> # Estoesun comentario

>>>B=7#asignación

* Definición de funciones:

>>>def nombrefunción(arg1,arg2..)

:...instrucción1

...instrucción2

.....................

....instrucciónN

...

La indentación delimita qué instrucciones pertenecen a la función.

A no ser que se use return, la función devuelve None.

* If:

>>>if condicion:

...instrucciones...

elif condicion:

...instrucciones...

else:

...instrucciones

...

* For:

>>>for variable in variable\_lista:

...instrucciones

...

Range: usado de la forma range(i) genera la lista[0,1,...,i].

Usado de la forma range(i,j) genera la lista[i,i+1,...,j-1]

* filter(función, lista\_arg) devuelve una lista con los elementos x de lista\_arg que cumplen f(x) = true.

>>>def f(x): return x%2 != 0 and x%3 != 0

...

>>>filter(f, range(2,25))

[5,7,11,13,17,19,23]

* map(función, lista\_arg) devuelve la lista:

[f(lista\_arg[0]),f(lista\_arg[1]),...,f(lista\_arg[n])]

>>>def cubo(x): return x\*x\*x

...

>>>map(cubo, range(1,5))[1,8,27,64]

* List Comprehensions: operaciones muy intuitivas con listas

>>>vec= [2,4,6]

>>>[3 \* x for x in vec] [6,12,18]

>>>[3 \* x for x in vecif x > 3] [12,18]

>>>[[x,x\*\*2] for x in vec][[2,4],[4,16],[6,36]]

* Definición y uso de Clases:

>>>class NombreDeClase:

...sentencia1

...................

...sentenciaN

...

>>>variable = NombreDeClase()

* Las variables de instancia no necesitan declararse.

>>>x=MiClase()

>>>x.numero=7

x.f() equivalea MiClase.f(x)

>>>MiClase.f(x)

“holamundo”

* Para definir una subclase se escribe:

class NombreDeClase(NombreSubclase)

Se heredan funciones y variables de clase.

* Es muy fácil dotar de herencia múltiple, basta con incluir más clases como argumentos en la definición:

>>>class MiSubClase(Superclase1, Superclase2,...)

A la hora de resolver un mensajea un objeto de MiSubClase se buscará el método en la propia clase, sino se encuentra se busca en Superclase1 (y en sus súper clases), luego en Superclase2, etc.

**Esqueleto**

la versión que recomendaríamos como esqueleto básico de programa ejecutable sería:

#!/bin/env python

print "Hola mundo"

Este fichero puede ser hecho ejecutable y ser ejecutado sin tener que escribir python dado que el propio fichero indica cómo buscar el comando python en el sistema.

$ chmod u+x hola\_mundo.py

$ ./hola\_mundo.py

Hola mundo

$ python hola\_mundo.py

Hola mundo

Cuando hemos ejecutado el fichero directamente lo hemos hecho dando la ruta relativa completa al fichero (./hola\_mundo.py). Hemos de hacerlo así porque normalmente el directorio de trabajo no se encuentra en el $PATH que el shell utiliza para buscar los ficheros ejecutables. Si hubiésemos intentado ejecutarlo directamente habríamos obtenido un error:

$ hola\_mundo.py

hola\_mundo.py: orden no encontrada

**aplicación a descargar para editar y ejecutar**

El paquete de instalación de Python, disponible para sistemas de 32 y 64 bits,

dispone de ejemplos y documentación, así como de IDLE, un sencillo entorno

de desarrollo para los scripts.

Para seguir la tradición informática lo primero que vamos a hacer es escribir un programa que imprima “Hola mundo”. Para ello vamos a abrir un intérprete de comandos de Python escribiendo python en la línea de comandos:

$ python

Python 2.6.6 (r266:84292, Sep 15 2010, 16:22:56)

[GCC 4.4.5] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>> print('hola mundo')

hola mundo

>>>

Al ejecutar el comando python sin ningún argumento extra se abre un interprete en el que podemos ejecutar comandos en Python. En el caso del ejemplo nos indica que estamos utilizando la versión 2.6.6 del lenguaje. Python tiene distintas versiones, la más moderna es la 3.3, pero las más utilizadas son todavía las de la serie 2.x. Este curso está escrito para las versiones 2.x porque las 3.x todavía no son usadas ampliamente. De todos modos las diferencias entre ambas versiones son mínimas y lo que vamos a ver en el curso es aplicable a ambas.

Este es un prompt similar al del shell al que ya estamos acostumbrados. Python nos indica que está listo para recibir órdenes. Si escribimos una orden como print(‘hola mundo’), Python la ejecutará y nos mostrará el resultado:

>>> print('Hola mundo!')

Hola mundo!

Una vez cumplida nuestra primera misión podemos salir del interprete de Python escribiendo quit() o pulsando las teclas control y d.

python es el intérprete que se distribuye con el lenguaje, pero como interprete interactivo [IPython](http://es.wikipedia.org/wiki/IPython) nos puede resultar más útil. IPython colorea la sintaxis y completa los comandos por nosotros pulsando la tecla tabulador.

Aunque el intérprete de python es muy útil, normalmente los programas no los vamos a escribir utilizándolo. Python tiene otro modo de ejecutar los comandos. Lo que haremos será escribir los comandos uno tras otro en un fichero de texto y después le pediremos a python que los ejecute secuencialmente. Por ejemplo, podemos crear el siguiente fichero utilizando un editor de textos:

hola\_mundo.py:

print("Hola mundo")

Una vez escrito y guardado el fichero podemos ejecutarlo escribiendo lo siguiente en la línea de comandos:

$ python hola\_mundo.py

Hola mundo

La ventaja de este sistema frente al del interprete es que el programa queda escrito en un fichero y lo podemos ejecutar tantas veces como queramos sin tener que reescribirlo.

Normalmente haremos que los ficheros que contienen código Python tengan la extensión .py. Esto no es un requisito obligatorio, pero nuestro trabajo será más sencillo si lo hacemos así.

**a qué tipo de paradigma pertenece**

**Python** es un lenguaje de programación multiparadigma. Esto significa que más que forzar a los programadores a adoptar un estilo particular de programación, permite varios estilos: **programación orientada a objetos**, **programación imperativa** y **programación funcional**. Otros paradigmas están soportados mediante el uso de extensiones.

**Python** usa tipado dinámico y conteo de referencias para la administración de memoria. Una característica importante de **Python** es la resolución dinámica de nombres; es decir, lo que enlaza un método y un nombre de variable durante la ejecución del programa (también llamado enlace dinámico de métodos).

Otro objetivo del diseño del lenguaje es la facilidad de extensión. Se pueden escribir nuevos módulos fácilmente en **C** o **C++**. \*\*Python \*\*puede incluirse en aplicaciones que necesitan una interfaz programable.

Aunque la programación en **Python** podría considerarse en algunas situaciones hostil a la programación funcional tradicional del Lisp, existen bastantes analogías entre **Python** y los lenguajes minimalistas de la familia Lisp como puede ser Scheme.

**Un ejemplo de un programa escrito en ese lenguaje**

1. ###Programa que imprima los 25 primeros numeros naturales

2. n = 1

3. while n <= 25:

4. print n,

5. n += 1

**Tarea 5: Scheme**

**Símbolos**

**+** = Suma cada uno de los argumentos. Ejemplo: (+ 1 2 3) = 6

**\*** = Multiplica cada uno de los argumentos. Ejemplo: (\* 1 2 3) = 6

**-** = Primer argumento menos los restantes o negación del número. Ejemplo: (- 5.3 2) = 3.3 **/** = Primer argumento entre cada uno de los restantes.Ejemplo: (/ 6 3) = 2

**abs** = Valor absoluto del argumento. Ejemplo: (abs -4) = 4

**exp** = Exponencial. Ejemplo: (expt 2 3) = 8

**max** = Máximo de sus argumentos. Ejemplo: (max 1 3 4 2 3) = 4

**min** = Mínimo de sus argumentos. Ejemplo: (min 1 3 4 2 3) = 1

**Sintaxis**

### Comentarios

Para agregar un comentario en Scheme se inicia con un punto y coma (;) y continúan hasta el final de la línea.

### Variables

Las variables son dinámicamente tipadas. Para asociarlas a un valor concreto, podemos usar define, una expresión let, o alguna de sus variantes. Las variables asignadas en el primer nivel usando define están en ámbito global (es decir, son visibles en el resto de programa).

(define var1 value)

Las variables asignadas mediante let ven su ámbito reducido al cuerpo de dicho let:

(let ((var1 value))

...

ámbito de var1

...)

### Procedimientos

Las funciones o procedimientos son objetos de primera clase en Scheme. Pueden ser asignados a variables. Por ejemplo, una función de dos argumentos *arg1* y *arg2* puede definirse como:

(define fun

(lambda (arg1 arg2)

...))

o en la forma abreviada equivalente:

(define (fun arg1 arg2)

...)

Las llamadas a función tienen la sintaxis siguiente:

(fun value1 value2)

Como vemos, la función invocada se encuentra en primer lugar, seguida de los argumentos de la llamada, formando una lista. Podemos también utilizar el procedimiento apply, que toma dos argumentos: el primero es el procedimiento que queremos invocar, mientras que el segundo es la lista de argumentos. Así, la anterior llamada a función puede escribirse, de forma equivalente, como:

(apply fun (list value1 value2))

### Listas

Scheme usa listas enlazadas de forma análoga a otros dialectos de Lisp.

Téngase en cuenta que la utilización de listas es mucho más sencilla que en otros lenguajes de programación tales como C, C++ y Pascal.

### Tipos de datos

Otros tipos de datos en Scheme son los enteros, racionales, reales, complejo, símbolos, cadenas, y puertos, listas asociativas, tablas hash, vectores, array y estructuras.

La mayoría de implementaciones proporciona lo que se conoce como una torre numéric completa, así como aritmética exacta e inexacta.

Los símbolos pueden ser definidos de varias maneras, siendo

'symbol

(string->symbol "symbol")

las más comunes.

### Igualdad

Scheme tiene tres tipos diferentes de igualdad:

eq?

Devuelve #t si los dos objetos son exactamente el mismo objeto, comprobando incluso dónde están guardados físicamente.

eqv?

Normalmente igual que eq?, pero trata algunos objetos (por ejemplo, caracteres y números) de forma especial para que los números que sean iguales sean eqv? incluso si no son eq?.

equal?

Compara el contenido de las estructuras de datos tales como listas, vectores y cadenas para determinar si son iguales.

string=?

Compara dos cadenas

char=?

Compara dos caracteres

=

Compara números

### Estructuras de control

#### Evaluación condicional

(cond (prueba1 expr1)

(prueba2 expr2)

...

(else exprn))

La primera expresión para la que la prueba resulte ser cierto (cualquier cosa salvo #f cuenta como cierto) será evaluada. Si todas las pruebas resultan ser #f, se evalúa la cláusula else.

Una variante de la cláusula cond es:

(cond...

(test => expr)

...)

En este caso, expr debe resultar en una función que toma un solo argumento. Si test resulta ser cierto, se llama a la función anterior con el valor devuelto por test.

Scheme también tiene:

(if test then-expr else-expr)

pero se usa mucho menos porque cond es más general y normalmente resulta más legible.

#### Bucles

Los bucles en Scheme suelen tomar la forma de una recursión final o *tail recursion* en inglés. Este tipo de recursión es preferido porque dispensa la acumulación de tramas en la pila de llamadas y su subsecuente desbordamiento. El estándar exige a las implementaciones optimizar llamadas en posición de recursión final para generar código equivalente a un ciclo en lenguajes imperativos. Un ejemplo clásico es la función factorial, que puede definirse sin recursión final como:

(define (factorial n)

(cond ((= n 0) 1)

(else (\* n (factorial (- n 1))))))

(factorial 5)

;; => 120

O con recursión final usando un procedimiento extra:

(define (factorial n)

(fact-iter 1 n))

(define (fact-iter product n)

(if (< n 2)

product

(fact-iter (\* product n)

(- n 1))))

(factorial 5)

;; => 120

### Entrada/salida

Scheme tiene el concepto de *puertos* de donde leer o a los que escribir. Scheme define tres puertos por defecto, accesibles con las funciones current-input-port, current-output-port y current-error-port.

**Esqueleto**

(define(NOMBRE DE LA FUNCION) ; Se define el nombre de la función o proyecto.

(INSTRUCCIONES O PASOS A REALIZAR))

(NOMBRE DE LA FUNCION)

EJECUCION DEL PROGRAMA. ; Se especifican los pasos o procedimientos a realizar.

**A qué tipo de Paradigma pertenece**

Al ser de carácter multiparadigmatico, podemos decir que pertenece al paradigma Funcional y el Imperativo.

Los programas escritos en un lenguaje funcional están constituidos únicamente por definiciones de funciones, entendiendo estas no como subprogramas clásicos de un lenguaje imperativo, sino como funciones puramente matemáticas, en las que se verifican ciertas propiedades como la *transparencia referencial* (el significado de una expresión depende únicamente del significado de sus sobrexpresiones), y por tanto, la carencia total de *efectos colaterales*.

Otras características propias de estos lenguajes son la no existencia de asignaciones de variables y la falta de construcciones estructuradas como la secuencia o la iteración (lo que obliga en la práctica a que todas las repeticiones de instrucciones se lleven a cabo por medio de funciones recursivas).

La programación imperativa se rige por dos conceptos básicos para la construcción de programas: la estructura y el modulo. De ahí que se hable de programación estructurada y de programación modular.

La programación en el paradigma imperativo consiste en determinar qué datos son requeridos para el cálculo, asociar a estas direcciones de memoria y efectuar, paso a paso, una secuencia de transformaciones en los datos almacenados de forma tal que el estado final represente el resultado correcto.

**Aplicación a descargar para editar y ejecutar**:

Existen diversos programas que pueden ayudar a compilar dicho lenguaje

* [Bigloo](http://www-sop.inria.fr/mimosa/fp/Bigloo/) es un sistema cuyo objetivo es permitir programar en Scheme aplicaciones que generalmente requieren C. Produce ejecutables pequeños y rápidos y se puede comunicar con C y Java. Principalmente para Unix, hay también versiones Win32 (con Cygwin) y MacOS X (con xnu-201.5). Licencia GPL.
* CHICKEN es un traductor de Scheme a C que se ejecuta en MacOS X, Windows y muchas versiones de Unix.
* Stalin es un traductor de Scheme a C muy optimizado.
* PLT-Scheme es un nombre que agrupa varias implementaciones de Scheme, siendo la principal DrScheme. DrScheme es un entorno gráfico interactivo de programación para Windows, MacOS X y Unix/X.
* SISC es un intérprete de Scheme en Java que implementa por completo el estándar R5RS.

**Ejemplo de programa escrito en ese lenguaje:**

; función que suma dos números agregados como argumento

(define (suma num1 num2)

; implementación del let anterior, aunque no es estrictamente necesario

(let

;variables de la operación

((numero1 num1)

(numero2 num2))

; instrucción que suma los números, es decir, la función

(+ numero1 numero2))

**Tarea 6:Haskell**

**Símbolos**

En Haskell, y en lo siguiente,

"o :: t" quiere decir que el objeto "o" es miembro del tipo "t", y

"t -> s" es un tipo, específicamente una función, que consume algo de tipo "t" y produce algo de tipo "s".

El operador (->) se nida por el derecho, ya que "t -> s -> r" quiere decir "t -> (s -> r)".

**Sintaxis**

### El tipo *Bool*

Los valores con este tipo representan expresiones lógicas cuyo resultado puede ser *True* o *False*.

#### Funciones y operadores

* (&&) :: Bool -> Bool -> Bool. Conjunción lógica.
* (||) :: Bool -> Bool -> Bool. Disyunción lógica.
* not :: Bool -> Bool. Negación lógica.
* otherwise :: Bool. Función constante que devuelve el valor *True*.

### El tipo *Int*

Los valores de este tipo son números enteros de precisión limitada que cubren al menos el intervalo [-2^29, 2^29 - 1] ([minBound, maxBound]).

### El tipo *Integer*

Los valores de este tipo son números enteros de precisión ilimitada que tienen las mismas funciones y operadores del tipo Int.

### El tipo *Float*

Los valores de este tipo son números reales. ( 2010, 23.4, 5.7 )

#### Funciones y operadores

* (+), (-), (\*), (/), (^) :: Float -> Float -> Float. Suma, resta, producto, división real y potencia de exponente entero.
* abs, signum, negate :: Int -> Int. Valor absoluto, signo y negación.
* (\*\*) :: Float -> Float. Potencia de exponente real

### El tipo *Double*

Los valores de este tipo son números reales, de mayor rango y con aproximaciones más precisas que los de tipo Float.

### El tipo *Char*

Los valores de este tipo son caracteres que se encuentran en una masa de alta complejidad de en una suma de caracteres dados con su alta definición.

Antes de utilizar esta función en hugs debemos utilizar IMPORT CHAR antes de nuestro algoritmo.

### Tuplas

Los elementos que forman una [tupla](https://es.wikipedia.org/wiki/Tupla) pueden ser del mismo o de distintos tipos. Es un conjunto de componentes relacionados. Por ejemplo: ('a', True,3)

### Listas

Los valores de este tipo son una colección de elementos del mismo tipo. Existen dos constructores para listas:

* [Elementos\_separados\_por\_comas], **por ejemplo:** [1,2,3,4]
* (primer\_elemento:resto\_de\_la\_lista), **por ejemplo:** (1:(2:(3:(4:[]))))

**Esqueleto**

binOpApp binOp = op (EJEMPLO) 🡨 --- Librería a utilizar, en este caso “binOpApp binOp = op”, nos permite realizar operaciones matemáticas sencillas.

main =

……………………..[Instrucciones y procesos a realizar]……………..

………………………………………………………………………………………………

……………………………………………………………………………………………….

-- Aquí se escriben las operaciones y procedimientos a realizar por el programa.

**Aplicación**

Un editor de texto y un compilador de Haskell. Probablemente ya tienes instalado tu editor de texto favorito así que no vamos a perder el tiempo con esto. Ahora mismo, los dos principales compiladores de Haskell son GHC (Glasgow Haskell Compiler) y Hugs. Para los propósitos de esta guía usaremos GHC. No voy a cubrir muchos detalles de la instalación. En Windows es cuestión de descargarse el instalador, pulsar “siguiente” un par de veces y luego reiniciar el ordenador.

GHC toma un script de Haskell (normalmente tienen la extensión .hs) y lo compila, pero también tiene un modo interactivo el cual nos permite interactuar con dichos scripts. Podemos llamar a las funciones de los scripts que hayamos cargado y los resultados serán mostrados de forma inmediata. Para aprender es mucho más fácil y rápido en lugar de tener que compilar y ejecutar los programas una y otra vez. El modo interactivo se ejecuta tecleando ghci desde tu terminal. Si hemos definido algunas funciones en un fichero llamado, digamos, misFunciones.hs, podemos cargar esas funciones tecleando :l misFunciones, siempre y cuando misFunciones.hs esté en el mismo directorio en el que fue invocado ghci. Si modificamos el script .hs y queremos observar los cambios tenemos que volver a ejecutar :l misFunciones o ejecutar :r que es equivalente ya que recarga el script actual. Trabajaremos definiendo algunas funciones en un fichero .hs, las cargamos y pasamos el rato jugando con ellas, luego modificaremos el fichero .hs volviendo a cargarlo y así sucesivamente. Seguiremos este proceso durante toda la guía.

**A qué tipo de paradigma pertenece**

Haskelles un lenguaje de programación funcional puro, de propósito general, que incluye muchas de las últimas innovaciones en el desarrollo de los lenguajes de programación funcional.

Incluye muchas de las últimas innovaciones en el desarrollo de los lenguajes de programación funcional, como son las funciones de orden superior, evaluación perezosa, tipos polimórficos estáticos, tipos definidos por el usuario, encaje por patrones, y definiciones de listas.

Incorpora, además, otras características interesantes como el tratamiento sistemático de la sobrecarga, la facilidad en la definición de tipos abstractos de datos, el sistema de entrada/salida puramente funcional y la posibilidad de utilización de módulos.

**Ejemplo**

binOpApp binOp = binOp

main =

let add = (+)

sub = (-)

in do

putStrLn (show (binOpApp add 1 2))

putStrLn (show (binOpApp sub 1 2))

TAREA 7: R

**Simbolos**

**+ =** adición.

- = substracción.

\* = multiplicación.

/ = división.

^ = potencia.

% % = módulo o residuo.

%/% = división de enteros.

< : Menor que.

<= : Menor o igual que.

> : Mayor que.

>= : Mayor o igual que.

! x NO lógico. Negación

x & y Y lógico Evalúa solo la primera

x && y id.. Evalúa ambas expresiones

x | y O lógico. Evalúa solo la primera

x || y id. Evalúa ambas expresiones

xor(x, y) O exclusivo, EXCLUSIVIDAD

**Sintaxis**

El lenguaje de programación R al tener una sintaxis adoptada de otros lenguajes con sintaxis bastante intuitiva, lo hace uno de los más fáciles y sencillos de entender, debido a que su estructura a la hora de codificar no demanda de algún uso de palabras ya predefinidas o algo por el estilo, salvo los operadores lógicos.

* INSTRUCCIÓN O PROCESO A REALIZAR.

Solo se coloca el carácter “>” al principio de cada proceso o operación que se desee realizar.

**Palabras ya reservadas:**

abs(x): Valor absoluto de x

ceiling(x): Devuelve el primer entero mayor a x

floor(x): Devuelve el primer entero más pequeño de x trunc(x): Devuelve la parte entera de x eliminando los decimales de un número

**Al ser un lenguaje de programación enfocado a la estadística y graficacion de datos. Contamos con las siguientes funciones:**

* Para calcular la **media aritmética** o **promedio** del vector usamos la función:

**mean(vector)**

que devuelve como resultado en este caso 7.

* Para obtener la **mediana** del vector podemos usar la función:

**median(vector)**

y tendremos como resultado 7.

* Podemos sumar los valores que tiene el vector con:

**sum(vector)**

y obtendremos como resultado 35.

* Con la siguente función obtenemos el valor máximo del vector teniendo como resultado en este caso 9.

**max(vector)**

**ESQUELETO**

> CUERPO DEL METODO………..………..… En esta parte se aplican las funciones y/o operadores aritméticos que deseemos aplicar

> [1] IMPRESIÓN DE LOS RESULTADOS REALIZADOS POR EL METODO

Y en la parte final de la estructura se realiza la impresión de los resultados obtenidos.

**Aplicación a descargar para editar y ejecutar**

RStudio es un entorno de desarrollo integrado (IDE) para R. Es software libre con licencia GPLv3 y se puede ejecutar sobre distintas plataformas (Windows, Mac, or Linux) o incluso desde la web usando RStudio Server.

Vamos a comenzar escribiendo el típico “Hola Mundo” de todos los inicios de programación con un lenguaje nuevo. Para ello vamos a Console y escribimos:

> hola.mundo <- function() {cat("¡Hola Mundo!\n")}

> hola.mundo()

¡Hola Mundo! Resultado

Con esto (la primera línea) lo que hacemos es crear un nuevo objeto **hola.mundo** que es de tipo función y al que le asignamos una tarea (lo que hay ente llaves) y al ejecutarlo (segunda línea) se le dice a R que realice la tarea asociada a dicha función, es decir, que escriba ¡Hola Mundo!

**A qué tipo de Paradigma pertenece**

Al ser de carácter multiparadigmatico, podemos decir que pertenece al paradigma Funcional, Imperativo y Orientado a Objetos.

Incluye muchas de las últimas innovaciones en el desarrollo de los lenguajes de programación funcional, como son las funciones de orden superior, evaluación perezosa, tipos polimórficos estáticos, tipos definidos por el usuario, encaje por patrones, y definiciones de listas.

La programación en el paradigma imperativo consiste en determinar qué datos son requeridos para el cálculo, asociar a estas direcciones de memoria y efectuar, paso a paso, una secuencia de transformaciones en los datos almacenados de forma tal que el estado final represente el resultado correcto.

La Programación Orientada a Objetos es un paradigma de programación que viene a innovar la forma de obtener resultados. Los objetos manipulan los datos de entrada para la obtención de datos de salida específicos, donde cada objeto ofrece una funcionalidad especial.

**Ejemplo de programa escrito en ese lenguaje**

USAR R COMO UNA CALCULADORA BASICA

> a=1

> b=2

> c=3; d=4

> a+b+c+d

[1] 10

> a-d

[1] -3

> c/d

[1] 0.75

> a

[1] 1

> b

[1] 2

> 2 \* pi

[1] 6.283185