Lenguajes y Paradigmas de Programación

Teoría y las prácticas asociadas a los distintos temas

Universitat d'Alacant
Universidad de Alicante

Resumen

Este documento contiene un temario que se puede aplicar a toda la programación en general pero se dividirá la forma en la que se imparte. Por un lado, veremos el temario apoyándonos sobre el lenguaje de Scheme y por otra parte, con Swift.

Se hará una breve introducción con respecto a la historia de los lenguajes de la programación para comprender las bases de los distintos tipos, se darán los conceptos de cada tema en un ámbito más general y se añadirá al finalizar un bloque de contenido las funciones sobre el lenguaje que se usa en dicho bloque.

Eduardo Espuch

Curso 2019-2020

Índice

1. Historia y conceptos de los lenguajes de programación **2**
1.1. Historia de los lenguajes de programación . . . 2
1.2. Elementos de los lenguajes de programación . . . 4
1.3. Abstracción . . . 5
1.4. Paradigmas de programación . . . 7
1.5. Compiladores e interpretes . . . 9

2. Programación funcional **11**
2.1. El paradigma de Programación funcional . . . 11
2.2. Scheme como lenguaje de programación funcional . . . 14
2.3. Tipos de datos compuestos en Scheme . . . 16
2.4. Listas en Scheme . . . 17
2.5. Funciones como tipos de datos de primera clase . . . 18
2.6. Prácticas relacionadas con este temario . . . 20

3. Procedimientos recursivos **21**
3.1. El coste de la recursión . . . 21
3.2. Soluciones al coste de la recursión: procesos iterativos . . . 22
3.3. Soluciones al coste de la recursión: memoization . . . 22
3.4. Recursión y gráficos de tortuga . . . 23
3.5. Prácticas relacionadas con este temario . . . 25

4. Estructuras de datos recursivas **25**
4.1. Listas estructuradas . . . 25
4.2. Árboles . . . 27
4.3. Arboles binarios . . . 30
4.4. Prácticas relacionadas con este temario . . . 31

5. Programacion funcional en Swift **32**
5.1. Practicas relacionadas con este temario . . . 32

6. Programacion OO en Swift **36**
6.1. Practicas relacionadas con este temario . . . 36

7. Anexos **39**
7.1. Seminario de Scheme aplicado a su bloque de contenido . . . 39
7.2. Conceptos básicos . . . 39
7.3. Soluciones al coste de la recursion . . . 42
7.4. Graficos de tortuga en Racket . . . 43
7.5. Listas estructuradas como pseudoarboles . . . 43
7.6. Listas estructuradas como arboles . . . 44
7.7. Listas estructuradas como arboles binarios . . . 44
7.8. Seminario de Swift aplicado a su bloque de contenido . . . 44

Bloque introductorio

En este breve bloque se introducirá la historia de la programación (yo la ignoraría y si no esta hecha es porque he puesto prioridad en otros conceptos) y los conceptos mas generales sobre los lenguajes de programación.

1. Historia y conceptos de los lenguajes de programación

Primero veremos sucesos históricos que permitieron a los lenguajes de programación ser la herramienta que es en la actualidad y después nos introduciremos mas en la definición de estos, los tipos y propiedades.

1.1. Historia de los lenguajes de programación

Los distintos sucesos historicos que fomentaron la aparicion de los actuales lenguajes de programacion. **NOTA:** esta en WIP y no se si lo hare ya que lo veo mas como conceptos de curiosidad que de necesidad.

De las maquinas de calcular a los computadores programables

WIP

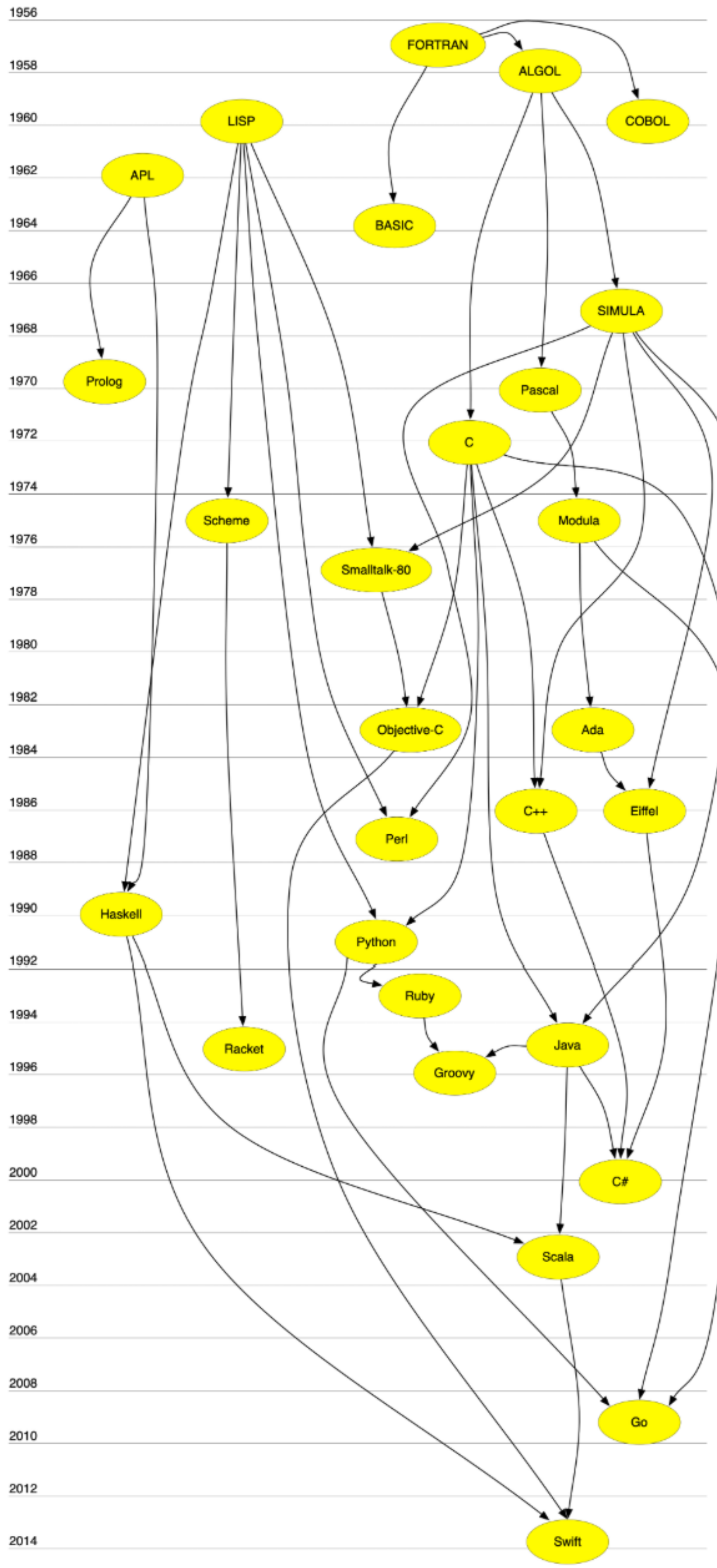
Los primeros lenguajes de programación

WIP

El nacimiento de los lenguajes de programación

WIP

Los primeros lenguajes de alto nivel



•Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje
Informacion Lenguaje

La explosión de los lenguajes de programación

WIP

Lenguajes de programación en la actualidad

WIP

1.2. Elementos de los lenguajes de programación

En este apartado trataremos con conceptos y propiedades de los lenguajes de programacion, definiendolos y categorizandolos. **NOTA:** estara brevemente explicado ya que unicamente pondre lo que considere necesario para futuras explicaciones.

NOTA 2: hay muchos conceptos repetidos a mi parecer o que son innecesarios en esta version, cuando se disponga de mas tiempo, se resumirán pero, considerando que es un tema que los profesores decidieron que lo viésemos por nuestra cuenta... me limitare a indicar al finalizar este bloque a indicar que conceptos son mas importantes de conocer hasta que el resumen este hecho.

Definición de la Encyclopedia of Computer Science

Un lenguaje de programación, según la **Encyclopedia of Computer Science**, es un conjunto de caracteres, reglas para combinarlos y reglas para especificar sus efectos cuando sean ejecutados por un computador, teniendo las siguientes características:

- No requiere ningún conocimiento de código maquina por parte del usuario.
- Tiene independencia de la máquina.
- Se traduce a código máquina.
- Utiliza una notación que es más cercana al problema especifico que se esta resolviendo que al código maquina.

Definición de Abelson y Sussman

Los **procesos computacionales** son seres abstractos que habitan los computadores. Cuando están en marcha, los procesos manipulan otras cosas abstractas denominadas datos. La evolución de un proceso está dirigida por un patrón de reglas denominado un programa

Un **lenguaje de programación potente** es más que sólo una forma de pedir a un computador que realice tareas. El lenguaje también sirve como un marco dentro del que organizamos nuestras ideas acerca de los procesos. Así, cuando describimos un lenguaje, deberíamos prestar atención particular a los medios que proporciona el lenguaje para combinar ideas simples para formar ideas más complejas.

Características de un lenguaje de programación

Recordemos las características de un lenguaje de programación, pero con mas detalle:

- Define un proceso que se ejecuta en un computador.
- Es de alto nivel, cercano a los problemas que se quiere resolver (abstracción).
- Permite construir nuevas abstracciones que se adapten al dominio que se programa.

Elementos de un lenguaje de programación

Para Abelson y Sussman, todos los LPs permiten combinar ideas simples en ideas mas complejas mediante los siguientes mecanismos:

- **Expresiones primitivas** que representan las entidades mas simples del lenguaje.
- **Mecanismos de combinación** con los que se construyen elementos compuestos a partir de elementos mas simples.
- **Mecanismos de abstracción** con los que dar nombre a los elementos compuestos y manipularlos como unidades.

Sintaxis y semántica

Sintaxis: conjunto de reglas que definen qué expresiones de texto son correctas. Por ejemplo, en C todas las sentencias deben terminar en ';'.

Semántica: conjunto de reglas que define cuál será el resultado de la ejecución de un programa en el computador.

Los lenguajes son para las personas

Los lenguajes de programación deben ser precisos, deben poder traducirse sin ambigüedad en lenguaje máquina para que sean ejecutados por computadores. Pero deben ser utilizados (leídos, comentados, probados, etc.) por personas.

La programación es una actividad colaborativa y debe basarse en la comunicación.

Importancia del aprendizaje de técnicas de lenguajes de programación

Es importante conocer cómo funciona "por dentroün lenguaje de programación y sus características comparadas.

- Mejora el uso del lenguaje de programación
- Incrementa el vocabulario de los elementos de programación
- Permite una mejor elección del lenguaje de programación
- Mejora la habilidad para desarrollar programas efectivos y eficientes
- Facilita el aprendizaje de un nuevo lenguaje de programación
- Facilita el diseño de nuevos lenguajes de programación

1.3. Abstracción

Hablamos de **abstracción** cuando queremos aislar un elemento de su contexto o del resto de los elementos que lo acompañan, es decir, hacer más énfasis en el '¿qué hace?' que en el '¿cómo lo hace?'.

Los LPs tienen como misión fundamental el proporcionar las herramientas necesarias para permitir las construcción de abstracciones, como podría serlo el denotar por nombres a una entidad del lenguaje (variables, funciones, clases,...) aislando a dicha entidad del resto.

Sera importante escoger un buen nombre para estos elementos ya que afectara a la hora de leer y reutilizar el código en el que se usen.

Modelar como una actividad fundamental

El **modelado de datos** es el proceso de documentar un diseño de sistema de software complejo como un diagrama de fácil comprensión, usando texto y símbolos para representar la forma en que los datos necesitan fluir.

- Para escribir un programa que preste unos servicios es fundamental modelar el dominio sobre el que va a trabajar
- Es necesario definir distintas abstracciones (tanto Application Programming Interfaces (APIs), como datos) que nos permitan tratar sus elementos y comunicarnos correctamente con los usuarios que van a utilizar el programa.
- Las abstracciones que vamos construyendo van apoyándose unas en otras y permiten hacer compresible y comunicable un problema complejo

WIP añadir ejemplo de modelado

Abstracciones computacionales

Existen abstracciones propias de la informática (computer science), que se utilizan en múltiples dominios. Por ejemplo, abstracciones de datos como:

- Listas
- Árboles
- Grafos
- Tablas hash

También existen abstracciones que nos permiten tratar con dispositivos y ordenadores externos:

- Fichero
- Raster gráfico
- Protocolo TCP/IP

Construcción de abstracciones

Uno de los trabajos principales de un informático es la construcción de abstracciones que permitan ahorrar tiempo y esfuerzo a la hora de tratar con la complejidad del mundo real.

TCP es lo que los científicos de computación llaman una abstracción: una simplificación de algo mucho más complicado que va por debajo de la cubierta. Resulta que una gran parte de la programación de computadores consiste en construir abstracciones. ¿Qué es una librería de cadenas? Es una forma de hacer creer que los computadores pueden manipular cadenas tan fácilmente como manipulan los números. ¿Qué es un sistema de ficheros? Es una forma de hacer creer que un disco duro no es en realidad un montón de platos magnéticos en rotación que pueden almacenar bits en ciertas posiciones, sino en su lugar, un sistema jerárquico de carpetas-dentro-de-carpetas que contiene ficheros individuales. Blog de Joel Spolsky.

Distintos aspectos de los lenguajes de programación

La programación es una disciplina compleja, que tiene que tener en cuenta múltiples aspectos de los lenguajes de programación y las API:

- Programas como procesos *runtime* que se ejecutan en un computador. Tenemos que entender qué pasa cuando se crea un objeto, cuánto tiempo permanece en memoria, cuál es el ámbito de una variable, etc.

  Herramientas: depuradores, analizadores de rendimiento.

- Programas como declaraciones estáticas. Hay que considerar un programa desde el punto de vista de una declaración de nuevos tipos, nuevos métodos, tipos genéricos, herencia entre clases, etc.

  Herramientas: entornos de programación con autocompletado de código, detección de errores sintácticos.

- Programas como comunicación y actividad social. Tenemos que tener en cuenta que un programa va a ser usado por otras personas, leído, extendido, mantenido, modificado. Los programas siempre se van a modificar.

  Herramientas: sistemas de control de versiones (Git, Mercurial, Github, Bitbucket), de gestión incidencias (Jira) , tests que evitan errores de regresión, ...

1.4. Paradigmas de programación

Este apartado contiene definiciones mas importantes, las cuales se recomienda conocer.

¿Qué es un paradigma de programación?

Un paradigma define un conjunto de características, patrones y estilos de programación basados en alguna idea fundamental. Es conveniente ver un paradigma como un estilo de programación que puede usarse en distintos lenguajes de programación y expresarse con distintas sintaxis.

Normalmente todos los lenguajes tienen características de más de un paradigma. Por motivos prácticos los lenguajes más populares no se limitan de forma estricta o pura a un único paradigma de programación.

Existen lenguajes que refuerzan y promueven la expresión de código en más de un paradigma de programación. Y lo hacen no por necesidad o accidente, sino con el intento explícito de fusionar más de un paradigma en una forma única de programar. Estos lenguajes se denominan lenguajes **multi-paradigma**.

NOTA: breve listado de los tipos de programacion:

- **Programación imperativa o por procedimientos:** se basa en dar instrucciones al ordenador de como hacer las cosas en forma de algoritmos, en lugar de describir el problema o la solución.*Ejemplo: C, BASIC o Pascal.*

- Programación orientada a objetos: está basada en el imperativo, pero encapsula elementos denominados objetos que incluyen tanto variables como funciones.*Ejemplo: C++, C#, Java o Python .*

- Programación dinámica: está definida como el proceso de romper problemas en partes pequeñas para analizarlos y resolverlos de forma lo más cercana al óptimo, busca resolver

problemas en O(n) sin usar por tanto métodos recursivos. Este paradigma está más basado en el modo de realizar los algoritmos, por lo que se puede usar con cualquier lenguaje imperativo.*Ejemplo: SQL.*

- Programación dirigida por eventos: la programación dirigida por eventos es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen.*Ejemplo: SQL.*

- **Programación declarativa**: está basada en describir el problema declarando propiedades y reglas que deben cumplirse, en lugar de instrucciones. Hay lenguajes para la programación funcional, la programación lógica, o la combinación lógico-funcional. La solución es obtenida mediante mecanismos internos de control, sin especificar exactamente cómo encontrarla (tan solo se le indica a la computadora qué es lo que se desea obtener o qué es lo que se está buscando). No existen asignaciones destructivas, y las variables son utilizadas con transparencia referencial. Los lenguajes declarativos tienen la ventaja de ser razonados matemáticamente, lo que permite el uso de mecanismos matemáticos para optimizar el rendimiento de los programas.

  - Programación funcional: basada en la definición los predicados y es de corte más matemático.*Ejemplo: Scheme (una variante de Lisp) o Haskell. Python también representa este paradigma.*
  - Programación lógica: basado en la definición de relaciones lógicas.*Ejemplo: Prolog.*
  - Programación con restricciones: similar a la lógica usando ecuaciones. *Ejemplo: variantes del Prolog.*

- **Programación multiparadigma:** es el uso de dos o más paradigmas dentro de un programa. El objetivo en el diseño de estos lenguajes es permitir a los programadores utilizar el mejor paradigma para cada trabajo, admitiendo que ninguno resuelve todos los problemas de la forma más fácil y eficiente posible.*Ejemplo: Scheme de paradigma funcional o Prolog (paradigma lógico), que cuentan con estructuras repetitivas, propias del paradigma imperativo.*

- **Programación reactiva:** este paradigma se basa en la declaración de una serie de objetos emisores de eventos asíncronos y otra serie de objetos que se "suscriben.ª los primeros (es decir, quedan a la escucha de la emisión de eventos de estos) y *reaccionan* a los valores que reciben.*Ejemplo: Rx de Microsoft (Acrónimo de Reactive Extensions).*

- **Lenguaje específico del dominio o DSL:** se denomina así a los lenguajes desarrollados para resolver un problema específico, pudiendo entrar dentro de cualquier grupo anterior. *Ejemplo: SQL.*

Veamos a continuación los paradigmas más importantes:

Paradigma funcional

- La computación se realiza mediante la evaluación de expresiones
- Definición de funciones
- Funciones como datos primitivos
- Valores sin efectos laterales, no existen referencias a celdas de memoria en las que se guarda un estado modificable

- Programación declarativa (en la programación funcional pura)

Lenguajes: Lisp, Scheme, Haskell, Scala, Clojure.

Paradigma lógico

- Definición de reglas
- Unificación como elemento de computación
- Programación declarativa

Lenguajes: Prolog, Mercury, Oz.

Paradigma imperativo

Los lenguajes de programación que cumplen el paradigma imperativo se caracterizan por tener un estado implícito que es modificado mediante instrucciones o comandos del lenguaje. Como resultado, estos lenguajes tienen una noción de secuenciación de los comandos para permitir un control preciso y determinista del estado.

- Definición de procedimientos
- Definición de tipos de datos
- Chequeo de tipos en tiempo de compilación
- Cambio de estado de variables
- Pasos de ejecución de un proceso

Lenguajes: Pascal.

Paradigma orientado a objetos

- Definición de clases y herencia
- Objetos como abstracción de datos y procedimientos
- Polimorfismo y chequeo de tipos en tiempo de ejecución

Lenguajes: Java.

1.5. Compiladores e interpretes

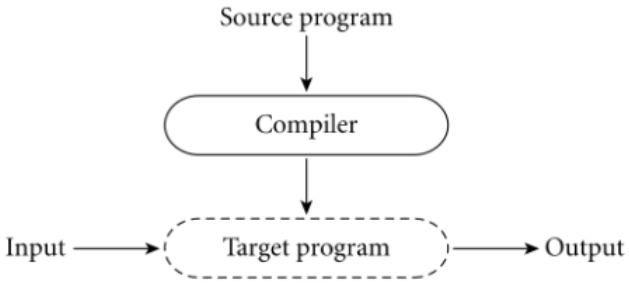
En el nivel de abstracción más bajo, la ejecución de un programa en un computador consiste en la ejecución de un conjunto de instrucciones del código máquina del procesador. Podemos tener un programa en ensamblador que, dependiendo del tipo de lenguaje en el que haya sido escrito, el codigo maquina que se ejecutara sera:

- el resultado de la compilación del programa original (en el caso de un lenguaje compilado)
- el código de un programa (intérprete) que realiza la interpretación del programa original (en el caso de un lenguaje interpretado)

Compilación

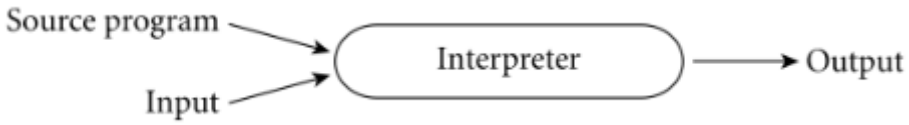
El proceso de compilación de un programa consiste en la traducción del código fuente original en el lenguaje de alto nivel al código máquina específico del procesador en el que va a ejecutarse el programa. El código máquina resultante sólo corre en el procesador para el que se ha generado.

La siguiente figura (tomada, como las demás de este apartado del Programming Language Pragmatics) muestra el proceso de generación y ejecución de un programa compilado.



- Ejemplos: C, C++
- Diferentes momentos en la vida de un programa: tiempo de compilación y tiempo de ejecución
- Mayor eficiencia

Interpretación



- Ejemplos: BASIC, Lisp, Scheme, Python, Ruby
- No hay diferencia entre el tiempo de compilación y el tiempo de ejecución
- Mayor flexibilidad: el código se puede construir y ejecutar .ºn the fly"(funciones lambda o clousures)

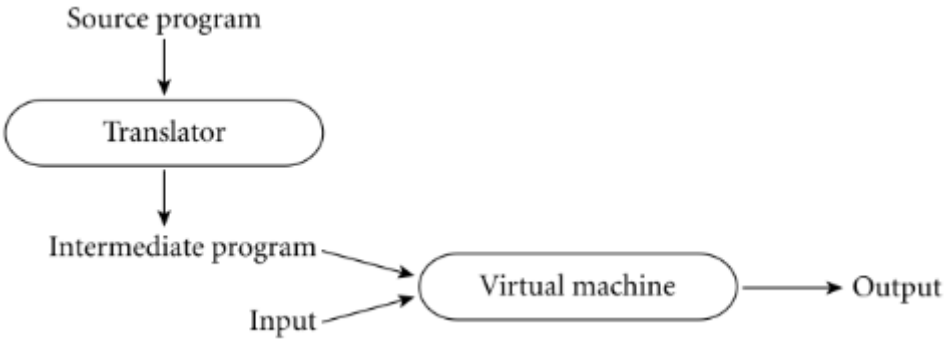
Los lenguajes interpretados suelen proporcionar un shell o intérprete. Se trata de un entorno interactivo en el que podemos definir y evaluar expresiones. Este entorno se denomina en los círculos de programación funcional un REPL (Read, Eval, Print, Loop) y ya se utilizó en los primeros años de implementación del Lisp. El uso del REPL promueve una programación interactiva en la que continuamente evaluamos y comprobamos el código que desarrollamos.

Enfoques mixtos

Existen también enfoques mixtos, como el usado por el lenguaje de programación Java, en el que se realizan ambos procesos.

En una primera fase el compilador de Java (javac) realiza una traducción del código fuente original a un código intermedio binario independiente del procesado, denominado bytecode. Este código binario es multiplataforma.

El código intermedio es interpretado después por un el intérprete (java) que ya sí que es dependiente de la plataforma. En la figura el intérprete se denomina Virtual machine (no confundir con el concepto de máquina virtual que permite emular un sistema operativo, por ejemplo VirtualBox).



- Ejemplos: Java, Scala

Bloque de Scheme

Se recomienda al lector que, antes de comenzar la lectura de este bloque, se disponga a leer el anexo del Seminario de Scheme, ya que en este bloque, aunque se vaya a tratar de generalizar el contenido, se pondrán ejemplo aplicados sobre éste.

2. Programación funcional

Este temario veremos conceptos básicos sobre el paradigma de Programación funcional aplicandolos a Scheme.

2.1. El paradigma de Programación funcional

Veamos conceptos sobre la Programación funcional en un ámbito mas general.

Pasado y presente del paradigma funcional

Definiciones y características

La programación funcional define un programa como un conjunto de funciones matemáticas que convierten unas entradas en unas salidas, sin ningún estado interno y ningún efecto lateral. Definamos las características principales:

- Definiciones de funciones matemáticas puras, sin estado interno ni efectos laterales, en este sentido, se podría considerar como un tipo concreto de programación declarativa.
- Valores inmutables
- Uso profuso de la recursión en la definición de las funciones
- Uso de listas como estructuras de datos fundamentales
- Funciones como tipos de datos primitivos: expresiones lambda y funciones de orden superior

Historia y características de Lisp WIP

Definiciones y características WIP

Lenguajes de programación funcional WIP

Aplicaciones prácticas de la programación funcional WIP

PROGRAMACIÓN DE SISTEMAS CONCURRENTES WIP

DEFINICIÓN Y COMPOSICIÓN DE OPERACIONES SOBRE STREAMS WIP

PROGRAMACIÓN EVOLUTIVA WIP

Evaluación de expresiones y definición de funciones

En esta parte comenzaremos a ver conceptos básicos que no se hayan comentado en el seminario, por lo tanto se irán añadiendo.

- En programación funcional, se usa 'evaluar una expresión', en lugar de 'ejecutarla' con la finalidad de reforzar la idea de que tratamos con expresiones matemáticas las cuales devuelve un unico resultado.
- El numero de argumentos de una función se denomina **aridad de la función**.
- Las funciones puras o funciones matemáticas puras son aquellas que cumplan las siguientes condiciones:
  - No modifican los parámetros que se les pasa
  - Devuelven un único resultado
  - No tienen estado local ni el resultado depende de un estado exterior mutable, es decir, devolverán siempre el mismo valor cuando se le pasan los mismos parámetros.
- Composición de funciones: idea fundamental de la programación funcional, en la cual transforman unos datos de entrada en otros de salida.

Programación declarativa vs. imperativa

Programación imperativa

Se trata de un conjunto de instrucciones que se ejecutan una tras otra (pasos de ejecución) de forma secuencial. En la ejecución de estas instrucciones se van cambiando los valores de las variables y, dependiendo de estos valores, se modifica el flujo de control de la ejecución del programa.

Para entender el funcionamiento de un programa imperativo debemos imaginar toda la evolución del programa, los pasos que se ejecutan y cuál es el flujo de control en función de los cambios de los valores en las variables.

Algunas características propias de la programación imperativa que no existen en la programación funcional, las cuales conocemos por lenguajes como C, C++, son:

- Pasos de ejecución. ¿Hace falta decir algo?
- Mutación o asignación destructiva se da cuando se modifica un valor de la variable con los pasos del programa.
- Efectos laterales o referencias, varios identificador referencia al mismo valor.

- Estado local mutable en las funciones, es decir, atributos mutables de un objeto modificable por el uso de funciones, alterando el estado interno del objeto. Si en declarativa no existe mutación, imagínate esto. Es mas, en programación declarativa,se tiene una transparencia referencial, es decir, es posible sustituir cualquier expresión por su resultado sin que cambie el resultado final del programa(para un mismo argumento, una función devolverá el mismo resultado).

Estas son características que no usaremos de ahora en adelante.

Resumen:

- Variable, nombre de una zona de memoria
- Existe asignación
- Referencias
- Pasos de ejecucion

Programación declarativa

Hablamos de programación declarativa para referirnos a lenguajes de programación (o sentencias de código) en los que se declaran los valores, objetivos o características de los elementos del programa y en cuya ejecución no existe mutación (modificación de valores de variables) ni secuencias de pasos de ejecución.

De esta forma, la ejecución de un programa declarativo tiene que ver más con algún modelo formal o matemático que con un programa tradicional imperativo. Define un conjunto de reglas y definiciones de estilo matemático.

Resumen:

- Variable, nombre dado a un valor(es una declaración)
- En lugar de pasos, se utiliza composición de funciones
- No existe asignación ni cambio de estado
- No existe mutación, se cumple la transferencia referencial: dentro de un mismo ámbito todas las ocurrencias de una variable y las llamadas a funciones devuelven el mismo valor

Modelo de computación de sustitución

Un modelo computacional es un formalismo (conjunto de reglas) que definen el funcionamiento de un programa. En el caso de los lenguajes funcionales basados en la evaluación de expresiones, el modelo computacional define cuál será el resultado de evaluar una expresión.

El **modelo de sustitución** es un modelo muy sencillo que permite definir la semántica de la evaluación de expresiones en lenguajes funcionales como Scheme. Se basa en una versión simplificada de la regla de reducción del cálculo lambda.

Es un modelo basado en la reescritura de unos términos por otros. Aunque se trata de un modelo abstracto, sería posible escribir un intérprete que, basándose en este modelo, evalúe expresiones funcionales.

El modelo de sustitución define cuatro reglas sencillas para evaluar una expresión. Llamemos a la expresión e. Las reglas son las siguientes:

1. Si e es un valor primitivo (por ejemplo, un número), devolvemos ese mismo valor.

2. Si e es un identificador, devolvemos su valor asociado con un define (se lanzará un error si no existe ese valor).

3. Si e es una expresión del tipo (f arg1 ... argn), donde f es el nombre de una función primitiva (+, -, ...), evaluamos uno a uno los argumentos arg1 ... argn (con estas mismas reglas) y evaluamos la función primitiva con los resultados.

4.a Si e es una expresión del tipo (f arg1 ... argn), donde f es el nombre de una función definida con un define, tenemos que evaluar primero los argumentos arg1 ... argn y después sustituir f por su cuerpo, reemplazando cada parámetro formal de la función por el correspondiente **argumento evaluado**. Después evaluaremos la expresión resultante usando estas mismas reglas.

4.n Si e es una expresión del tipo (f arg1 ... argn), donde f es el nombre de una función definida con un define, tenemos que sustituir f por su cuerpo, reemplazando cada parámetro formal de la función por el correspondiente **argumento sin evaluar**. Después evaluar la expresión resultante usando estas mismas reglas.

Para **orden aplicativo** se realiza la **4.a**, las evaluaciones antes de realizar las sustituciones, lo que define una evaluación de *dentro a fuera* de los paréntesis. Nos referiremos a esto cuando digamos **modelo de sustitución aplicativo**. **Es el que Scheme usa.**

Por otro lado, con el orden normal, donde se realiza la 4.n, donde se realizan todas las sustituciones primero para acabar con una larga expresión formada por expresiones primitivas. Nos referiremos a el como **modelo de sustitución normal**.

2.2. Scheme como lenguaje de programación funcional

Vamos a centrarnos mas como características funcionales de Scheme, en particular, nos centraremos en este apartado de:

- Símbolos y la primitiva *quote*.
- Uso de listas
- Definición de funciones recursivas en Scheme

Funciones y formas especiales

Funciones , primitivas que se evalúan usando el modelo de sustitución aplicativo, evaluándose las expresiones de dentro a fuera respecto a los paréntesis.

Formas especiales, expresiones primitivas de Scheme que tienen una forma de evaluarse propia, distinta de las funciones. Esta forma de evaluarse puede evitar problemas por errores de ejecución, recordad que con el orden aplicativo comprobaríamos los argumentos y luego se evaluaría la función completa pero con las formas especiales no tiene por que ser así, devolviendo un resultado antes de alcanzar lo que provoca un error.

Formas especiales en Scheme

Vamos a citar y añadir información de ampliación sobre las distintas formas especiales que Scheme tiene definidas, podemos encontrar la sintaxis en el seminario.

- *define* para asociar una expresión (que se evalúa obteniendo un valor) a un identificador.

- *define* para definir funciones. En la evaluación toma parte otra forma especial, *lambda*, la cual se usa para definir funciones en tiempo de ejecución o procedimientos. La evaluación del *define* lo que hace realmente es usar *lambda* para luego asignar a esa función un identificador. Mas adelante veremos la idea mas desarrollada.

- Condicional *if*, evalúa la condición y según el resultado realizara una de las expresiones aportadas.

- Condicional *cond*, se van evaluando las expresiones condicionales de forma ordenada hasta que una devuelva *#t*, si una lo hace, se evalúa la expresión asociada a esta condición, en el caso contrario se devolverá la expresión asociada al *else*, que seria la salida por defecto.

- Expresión lógica *and*, se evalúan los argumentos de 1 en 1, de izquierda a derecha. El proceso de evaluar todos los argumentos acaba en cuanto uno devuelve *#f* (recordad que cualquier valor distinto a *#f* se interpreta como cierto). Se devolverá el valor obtenido de evaluar el ultimo argumento, sin importar que dato es.

- Expresión lógica *or*, se evalúan los argumentos de 1 en 1, de izquierda a derecha. El proceso de evaluar todos los argumentos acaba en cuanto uno devuelve algo distinto a *#f* siendo ese valor el que se devuelva.

- *quote* para símbolos, usando el operando *quote* y un único argumento, convertimos a este argumento en un simbolo, NO ES UN CARACTER. Los simbolos son lo que hemos llamado identificadores y en memoria, a diferencia de los caracteres o cadenas de caracteres, se representan con el codigo *hash* de este. Son un tipo de dado atomico. Otra forma de inicializarlos es usando la comilla.

- *quote* para expresiones, dada una expresión correcta, la forma especial *quote* (o *'*) devolverá una lista o una pareja definida por la expresión con los elementos sin evaluar, a diferencia de usando las funciones *cons* y *list*, que usa el modelo de sustitución aplicativo.

Listas

La utilización de listas es otra de las características fundamentales del paradigma funcional, en Scheme estas pueden contener cualquier valor, incluyendo otras listas.

Como ya hemos comentado, con la función list los argumentos se evalúan mientra que con el uso de la forma especial quote se interpretarían como símbolos.

Con las funciones *car* y *cdr* obtenemos el primer elemento o el resto, ademas de que se puede usar composición de funciones para anidar o usarlo directamente. Por ejemplo, (*car* (*cdr* lista)) y (*cadr* lista) devolverían el mismo elemento.

Con las funciones *cons* y *append* podemos hacer composición de listas. *append* es mas general y concatena todos los argumentos que reciba mientras que estos sean listas y *cons*, por otro lado, añade el primer elemento al principio de la lista que se ha introducido como segundo argumento.

Recursión

Podemos hacer que una función se llame a si misma, las funciones recursivas tienen la siguiente estructura:

```
(define (<func> <args>)
   (<expresion-logica>
      (<expresion-base>);caso base: devuelve un valor
      (<expresion-general> (<func> <args-1>));caso general: evalua a la funcion
   )
)
```

- Caso base: es cuando la función considera que el argumento introducido seria el primer caso.
- Caso general: puede variar como se interprete, pueden haber varios casos pero todos deberán hacer llamada recursiva.

Objetos de primera clase

Diremos que un objeto es de primera clase cuando puede:

- Asignarse a variables
- Pasarse como argumento
- Devolverse por una función
- Guardarse en una estructura de datos mayor

Recursión y listas

Combinando listas con las funciones *car*, *cdr* y *null?*, podemos trabajar con este tipo de estructuras secuenciales

2.3. Tipos de datos compuestos en Scheme

Veamos algunos datos compuestos.

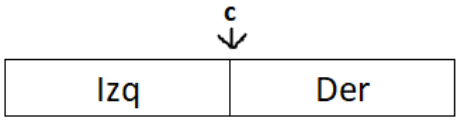
El tipo de dato pareja

Usando *cons* y dos elementos de tipo simple o el *quote* junto con el símbolo '.' separando los dos elementos de tipo simple podemos definir una pareja. Cuando usamos *car* y *cdr* sobre estas, nos devolverán el valor izquierdo o derecho, respectivamente.

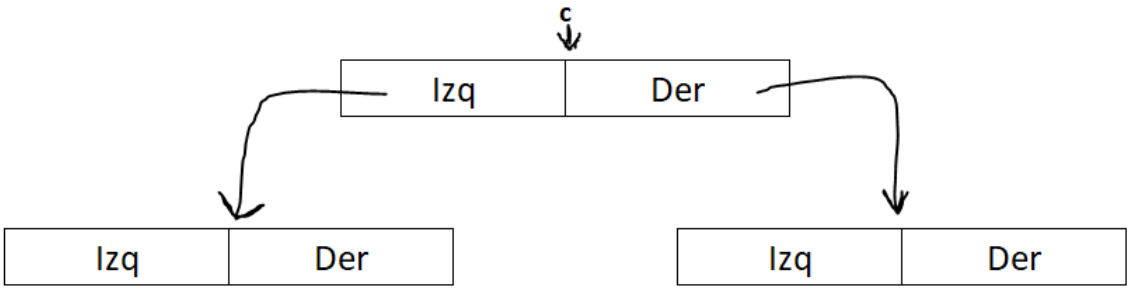
Son estructuras de datos inmutables, es decir, no podrán ser modificadas una vez creadas. Las parejas son objetos de primera clase.

Diagramas caja-y-puntero

Las parejas, en un diagrama Box-n-pointer, se denota de la siguiente forma:



Como podemos definir parejas con sus elementos siendo otras parejas, veremos que tomara la siguiente aparencia:

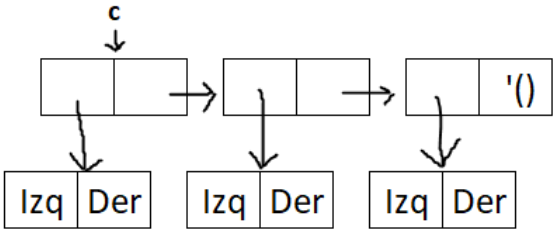


2.4. Listas en Scheme

Una ampliación de listas que, mas adelante, desaparecerá para agruparse con todo el contenido.

Implementación de listas en Scheme

- Una pareja formado por dos datos simples, no es una lista.
- Una lista vacía no sera una pareja pero una lista con al menos un elemento si lo sera.
- Una pareja cuyo segundo elemento es una lista vacía, sera una lista y una pareja. Esto para diagramas box-n-pointers indicaría que el segundo elemento, si es una lista vacía '() y tendríamos una pareja que también es una lista.



  En este ejemplo, **c** es tanto una lista como una pareja donde tendremos 3 parejas como los elementos.

- De igual forma, podemos añadir una lista a una lista y mostrarlo en en los diagramas.

Funciones con número variable de argumentos

Si a la hora de definir una funcion, usamos la siguiente sintaxis (define (<funcion> . <args>) <cuerpo>), permitira una funcion de aridad indefinida, puede incluso no recibir argumentos (por ejemplo, la funcion (*list*) actua asi. Cabe decir que se puede exigir una cantidad de argumentos iniciales poniendolos a la izquierda de '.' siendo la siguiente sintaxis (define (<funcion> <arg1>...<argn> . <lista-args>) <cuerpo>) la apropiada.

- Lo que se reciba a continuacion de '.', se trata como una lista de tamaño variable, por lo general, las funciones de este tipo se comprueba un caso general donde se comprueba si la lista es vacia y otro caso en el que manipula la lista de alguna forma.
- Es complicado aplicar recursividad en estas funciones sin hacer uso de la funcion *apply*, la cual tiene esta sintaxis: (apply <funcion-args-indefinidos> <lista>).
- Estas funciones suelen actuar a la par con funciones que reciben como argumentos una lista que manipulara los argumentos mediante recursividad. Es la unica forma de usar recursividad sin usar *apply*.

Función apply

Como se ha comentado anteriormente, la funcion *apply*, con la sintaxis (apply <funcion-args-indefinidos> <lista>) aplica la funcion indicada usando como argumentos los elementos de la lista. Es una forma de poder usar una lista como un conjunto de argumentos y que estos no tenga porque ser tratados necesariamente como miembros de una lista.

apply es la primera función de orden superior de las que veremos.

2.5. Funciones como tipos de datos de primera clase

Una de las características fundamentales de la programación funcional es considerar a las funciones como objetos de primera clase. Recordemos que un tipo de primera clase es aquel que:

- Puede ser asignado a una variable
- Puede ser pasado como argumento a una función
- Puede ser devuelto como resultado de una invocación a una función
- Puede ser parte de un tipo mayor

Vamos a ver que las funciones son ejemplos de todos los casos anteriores: vamos a poder crear funciones sin nombre y asignarlas a variables, pasarlas como parámetro de otras funciones, devolverlas como resultado de invocar a otra función y guardarlas en tipos de datos compuestos como listas.

Forma especial lambda

Sabemos que lambda es una forma especial y que es una funcion en tiempo de ejecucion a la que podemos asignarle un identificador. Esto permite que cuando usamos el define para crear funciones, lo que realmente se hace es asignar un identificador a un lambda, con los argumentos requeridos y el cuerpo indicado.

Una forma lambda sin evaluar se considera como una procedure (comprobar con la funcion *procedure?* al igual que lo harian las funciones '+' o '-' sin evaluar.

Funciones argumentos de otras funciones

Gracias a que Scheme evalua las funciones y devuelve un resultado, podemos interpretar funciones como argumentos. Ademas, usando comoposicion de funciones, podemos anidar resultados de diversas evaluaciones.

Generalización

En algunos casos, encotraremos funciones con estructura parecida. En estos casos, podemos generalizar la estructura en una funcion que reciba una funcion como argumentos y reducir a lo mas basico el resto de funciones, haciendo que la funcion generalizada las aplique siguiendo la estructura que estas esperarian.

Funciones que devuelven funciones

Podemos definir funciones (que llamaremos funciones constructoras) las cuales devolvera otra funcion (llamadas clausulas).

Esta practica se puede aplicar a:

- Composicion de funciones, la funcion constructora recibe como argumentos otras funciones mientras que la clausura unicamente trataria con los datos.
- Comprueba de valores: funcion constructora recibe una condicion y puede que una funcion y la clausura solo los datos, la clausura ira evaluando los datos con respecto a la condicion pasada previamente.
- Combinando todo: si sabemos que podemos pasar condiciones, funciones y datos como argumentos, y ademas permitir que una funcion tenga una lista de argumentos indeterminada, si aplicamos estos conceptos a la funcion constructora y a la clausura podemos hacer funciones mas complejas.

Funciones en estructuras de datos

La última característica de los tipos de primera clase es que pueden formar parte de tipos de datos compuestos, como listas. Dicho de otra manera, una lista(o pareja) puede estar compuesta por funciones o procedimientos y si seleccionamos uno de los elementos de esta lista lo interpretara como un procedimiento, con lo cual, usando los argumentos validos en una expresion, se evaluara como corresponde.

Funciones de orden superior

Llamamos funciones de orden superior (higher order functions en inglés) a las funciones que toman otras como parámetro o devuelven otra función. Permiten generalizar soluciones con un alto grado de abstracción.

Estas funciones permiten definir operaciones sobre las listas de una forma muy concisa y compacta. Son muy usadas, porque también se pueden utilizar sobre streams de datos obtenidos en operaciones de entrada/salida (por ejemplo, datos JSON resultantes de una petición HTTP).

Tendremos las siguientes funciones:

- map con un argumento lista: con la sintaxis (map <func> <lista>) devolvera una lista en la que se ha aplicado la funcion <func> a cada elemento de la lista. Cabe decir que la funcion <func> actua con un elemento que se espera que sea del mismo tipo que este compuesta la <lista>.
- map con mas argumentos listas: con la sintaxis (map <func> <lista1>...<listan>) devolvera una lista donde la funcion <func> necesitara de n argumentos que los tomara de las listas, devolviendo un unico valor como resultado, razon por la que unicamente la evaluacion devuelve una unica lista.
- filter: con la sintaxis (filter <condicion> <lista>) que devolvera una lista en la cual solo se mostraran los elementos que cumplan la condicion.
- exists?: con la sintaxis (ormap <condicion> <lista>) devolvera un bool *#t* si al menos un elemento de la lista cumple la condicion y *#f* si ninguna lo hace. La funcion *exists?* esta definida en Scheme como *ormap*.

- for-all?: con la sintaxis (andmap <condicion> <lista>) devolvera un bool *#f* si al menos un elemento de la lista no cumple la condicion o *#t* si todos lo cumplen. La funcion *for-all?* esta definida en Scheme como *andmap*.

- foldr: con la sintaxis (foldr <func-combina> <dato-base> <lista>) que devuelve un valor. La idea es que se va usando la funcion de plegado (<func-combina>) con la <lista>, que se lee de derecha a izquierda, y <dato-base>. El proceso de evaluacion consiste en:

  1. Devuelve la base cuando lista es vacia.
  2. Se aplica la funcion de plegado sobre el elementos de la derecha de la lista y resultado previo.

  # El resultado final es el resultado de haber hecho (func elem_izq (func ... (func elem_der base)...)

  La implementacion seria:

```
(define (Foldr func base lista)
    (if (null? lista)
        base
        (func (car  lista) (Foldr func base  (cdr  lista)))
        )
  )
```

- foldl: con la sintaxis (foldl <func-combina> <dato-base> <lista>) que devuelve un valor. Se aplica la funcion de plegado de izquierda a derecha, siendo los primeros elementos tratados la base y el elemento mas a la izquierda.

  1. Devuelve la base cuando lista es vacia.
  2. Se aplica la funcion de plegado sobre el elementos de la izquierda de la lista y resultado previo.

  # El resultado final es el resultado de haber hecho (func elem_der (func ... (func elem_izq base)...)

  La implementacion seria:

```
(define (Foldl func base lista)
    (if (null? lista)
        base
        (func (car (reverse lista)) (Foldl func base (reverse (cdr (reverse lista)))))
        )
  )
```

2.6. Prácticas relacionadas con este temario

texto

Práctica 1

texto

Práctica 2

texto

Práctica 3

texto

Práctica 4

texto

3. Procedimientos recursivos

En este tema veremos algunos aspectos negativos de la recursión: su coste espacial y temporal. Veremos que hay soluciones a estos problemas, cambiando el estilo de la recursión y generando procesos iterativos o usando un enfoque automático llamado memoization en el que se guardan los resultados de cada llamada recursiva. Por último, veremos un último ejemplo curioso e interesante de la recursión para realizar figuras fractales con gráficos de tortuga.

3.1. El coste de la recursión

Hasta ahora hemos estudiado el diseño de funciones recursivas. Vamos a tratar por primera vez su coste. Veremos que hay casos en los que es prohibitivo utilizar la recursión tal y como la hemos visto. Y veremos también que existen soluciones para esos casos.

La pila de la recursión

Cada llamada a la recursión deja una función en espera de ser evaluada cuando la recursión devuelva un valor, estas llamadas en espera, junto con sus argumentos, se almacenan en la **pila de la recursión**.

Cuando la recursión devuelve un valor, los valores se recuperan de la pila, se realiza la llamada y se devuelve el valor a la anterior llamada en espera.

Si la recursión está mal hecha y nunca termina se genera un stack overflow porque la memoria que se almacena en la pila sobrepasa la memoria reservada para el intérprete DrRacket.

Coste espacial de la recursión

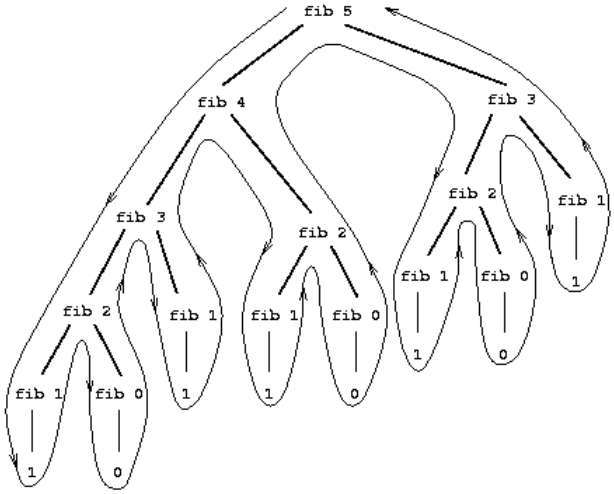
El coste espacial de un programa es una función que relaciona la memoria consumida por una llamada para resolver un problema con alguna variable que determina el tamaño del problema a resolver.

El coste depende del número de llamadas a la recursión

No hace falta decir mas... pero para mayor comodidad veamos el ejemplo con la famosa funcion de Fibonacci que tiene la siguiente estructura:

```
1  Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)
2  Fibonacci(0) = 0
3  Fibonacci(1) = 1
```

Cada llamada a la recursion produce otras dos , acabando con un total de 2^n llamada siendo n el dato introducido en la primera llamada.



Esto implica que la funcion requiere de $O(2^n)$ de espacio de memoria por los valores y por otro lado, tendra un coste temporal de $O(2^n)$.

3.2. Soluciones al coste de la recursión: procesos iterativos

Diferenciamos entre procedimientos y procesos: un procedimiento es un algoritmo y un proceso es la ejecución de ese algoritmo.

Es posible definir *procedimientos recursivos* que generen procesos iterativos (como los bucles en programación imperativa) en los que no se dejen llamadas recursivas en espera ni se incremente la pila de la recursión. Para ello construimos la recursión de forma que en cada llamada se haga un cálculo parcial y en el caso base se pueda devolver directamente el resultado obtenido.

Este estilo de recursión se denomina *recursión por la cola* (tail recursion, en inglés).

Se puede realizar una implementación eficiente de la ejecución del proceso, eliminando la pila de la recursión.

Siento decirlo... pero la unica diferencia que veo en una funcion recursiva y un proceso iterativo es que el proceso iterativo es una funcion que es el unica expresion de otra funcion y procede a hacer una recursion. Tambien es que en el proceso, uno de los argumentos es el resultado o un contador.

Procesos iterativos

- La recursión resultante es menos elegante.
- Se necesita una parámetro adicional en el que se van acumulando los resultados parciales.
- La última llamada a la recursión devuelve el valor acumulado.
- El proceso resultante de la recursión es iterativo en el sentido de que no deja llamadas en espera ni incurre en coste espacial.

3.3. Soluciones al coste de la recursión: memoization

Una alternativa que mantiene la elegancia de los procesos recursivos y la eficiencia de los iterativos es la memoization. La idea de la memoization es guardar el valor devuelto por la cada llamada en alguna estructura (una lista de asociación, por ejemplo) y no volver a realizar la llamada a la recursión las siguientes veces.

Para ello introduciremos una nueva herramienta, un diccionario *clave-valor* con los métodos imperativos *put* y *get*.

- Creación del diccionario: primero debemos tener la implementanción definida:

```
(define (crea-diccionario)
  (mcons '*diccionario* '()))

(define (busca key dic)
  (cond
    ((null? dic) #f)
    ((equal? key (mcar (mcar dic)))
     (mcar dic))
    (else (busca key (mcdr dic)))))

(define (get key dic)
  (define record (busca key (mcdr dic)))
  (if (not record)
      #f
      (mcdr record)))

(define (put key value dic)
  (define record (busca key (mcdr dic)))
  (if (not record)
      (set-mcdr! dic
                 (mcons (mcons key value)
                        (mcdr dic)))
      (set-mcdr! record value))
  value)
```

Con esto tendremos definido las 3 funciones que usaremos.

- Asigna un identificador a un dato tipo diccionario.

  (define <identificador> (crea-diccionario))

- La función (put key value <identificador>) asocia un valor a una clave, la guarda en el diccionario (con mutación) y devuelve el valor.

- La función (get key <identificador>) devuelve el valor del diccionario asociado a una clave. En el caso en que no exista ningún valor se devuelve *#f*.

Con esto, podemos tener un dato que almacena valores en posiciones, y pueden darse casos en los que una funcion puede comprobar si hay almacenado valores, evitando tener que volverlos a calcular.

3.4. Recursión y gráficos de tortuga

texto

Gráficos de tortuga en Racket

La libreria se añade con **(require graphics/turtles)**
Las funciones basicas son:

- *(turtles #t)*: abre una ventana y coloca la tortuga en el centro, mirando hacia el eje X (derecha)

- *(clear)*: borra la ventana y coloca la tortuga en el centro

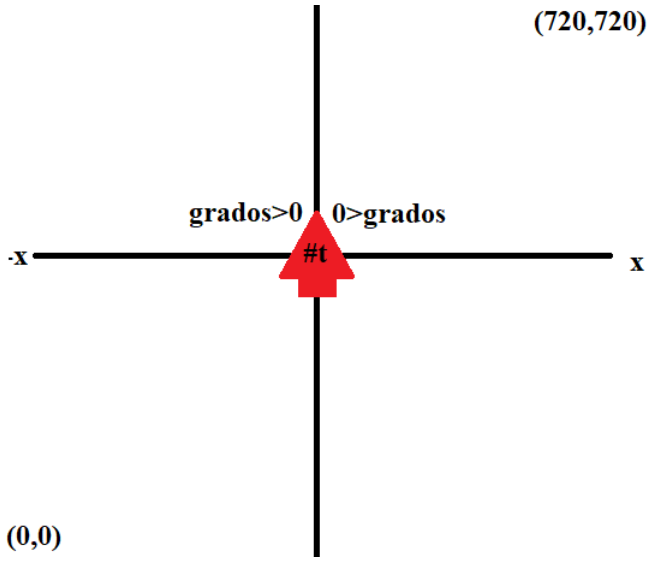
- *(draw d)*: avanza la tortuga dibujando d píxeles
- *(move d)*: mueve la tortuga d píxeles hacia adelante (sin dibujar)
- *(turn g)*: gira la tortuga g grados (positivos: en el sentido contrario a las agujas del reloj)
- *(begin <proceso1>...<proceson>)*: permite realizar una secuencias de ejecuciones, permitiendo realizar dibujos completos en una evaluacion.

Se puede usar recursividad con funciones de dibujo, pero es importante tener claro como debe de ser el algoritmo que debe de seguir y ademas sabiendo que debemos de desplazar la tortuga hasta la posicion inicial esperada, es importante tener claro el desplazamiento que esta realiza en todo momento.

Vamos a proponer a continuacion una de las posibles formas de tratar problemas de dibujar si se hacen de forma recursiva:

1. Realizar un pseudocodigo indicando los casos.
2. Funcion que dibuja la estructura mas basica.
3. Funcion mas compleja que usa las basicas para hacer estructuras mas complicadas, en el caso de que sea necesario. Se recomienda que la tortuga se quede en la misma posicion en el estado inicial que en el final.
4. Funcion recursiva de dibuja, se aplica la recursividad (puede darse recursividad mutua, las curvas de Hillbert son un ejemplo) en la que se aplica el pseudocodigo realizado, como con las funciones previas esperamos que la tortuga acabe en la misma posicion inicial, y que en el caso base se dibuja la estructura basica, el codigo se desplazara con esto en mente.
5. Funcion de inicio, recibe el parametro para la funcion recursiva e inicializa la ventana desplazando la tortuga a una posicion comoda para que ocupe el espacio que necesite.

Como referencia, se deja esta imagen tan elaborada entendiendo como se mueve la tortuga, considerar que la tortuga se inicializa en direccion al eje x pero en este ejemplo apunta al y. El tamaño usado es en pixels en la ventana estandar que a mi me sale.



Recursión mutua

Recursion mutua es el termino usado para un set de funciones que se van llamando entre ellas cumpliendo una condicion hasta que se da un caso base en una de ellas.

3.5. Prácticas relacionadas con este temario

texto

Práctica 5

texto

4. Estructuras de datos recursivas

Con este tema haremos hincapie en recursos que ya hemos conocido y ampliaremos la informacion que conocemos sobre estos y como usarlos con otras funcionalidades.

4.1. Listas estructuradas

Hemos visto que las listas en Scheme se implementan como un estructura de datos recursiva, formada por una pareja que enlaza en su parte derecha el resto de la lista y que termina con una parte derecha en la que hay una lista vacía.

Definición y ejemplos

Llamaremos **lista estructurada** a una lista que contiene otras sublistas. Lo contrario de lista estructurada es una **lista plana**, una lista formada por elementos que no son listas. Llamaremos **hojas** a los elementos de una lista que no son sublistas.

A las listas estructuradas cuyas hojas son símbolos se les denomina en el contexto de la programación funcional expresiones-S

Tengamos en cuenta los siguientes casos:

- Una lista de parejas, es una lista plana ya que las parejas no se consideran listas.
- Una lista vacia no es una hoja.
- Podemos afirmar que una lista es plana si y solo si el primer elemento es una hoja y el resto es una lista plana. Puede implementarse con una **FOS** tipo *for-all?* con la condicion siendo comprobar que sea una hoja.
- Una lista vacia es una lista plana.
- Una lista estructurada es una lista con al menos un elemento siendo una lista. Puede implementarse usando una **FOS** tipo *exists?* con la condicion siendo comprobar que sea una lista.

Ya hemos visto un tipo de diagramas, los Box'n'Pointer, ahora veremos los diagramas con los pseudoarboles. OJO, decimos pseudoarboles porque las listas estructuradas **no son arboles** en realidad, mas adelante veremos como tratar con estos.

Las listas estructuradas definen una estructura de niveles, donde la lista inicial representa el primer nivel, y cada sublista representa un nivel inferior. Los datos de las listas representan las hojas. Vamos a enumerar algunos conceptos antes de comenzar.

1. El simbolo '*' representa una lista, por lo general sera la raiz del arbol.
2. Una lista puede contener datos simples, otras listas e incluso procedimientos. Con este planteamiento, podemos interpretar una expresion como un pseudoarbol.
3. Los datos estan en las hojas, a diferencia de los arboles que hay datos en todos los nodos.
4. El **planteamiento en general** de todas las funciones que se definen es aplicar un algoritmo para el primer hijo y el resto de hijos sucesivamente hasta que el conjunto de restos de hijos sea vacio o sea una hoja, segun que funcion.
5. Muchas funciones pueden ser implementadas usando FOS, en especial, las funciones de plegado y mapeado.

Vamos a considerar las siguientes funciones definidas para los pseudoarboles:

- (hoja? <elem>):bool, comprueba si el elemento es una lista. Si no es un dato tipo lista, entonces es una hoja, devolviendo *#t*.
- (plana? <elem>):bool, partiendo del caso base de que **una lista vacia es plana**, comprobara si el hijo de la izquierda es una hoja asumiendo que el resto de hijos estan en una lista plana. Implementable con una (andmap hoja? <lista>) si usamos una FOS o con el **planteamiento general** y los casos considerados.
- (estructurada? <elem>):bool, si un solo elemento no es una hoja, devolvera *#t*. Puede implementarse partiendo de que si una list no es plana, entonces es estructurada (not (plana? <lista>)) o con un (ormap list? <lista>)
- (num-hojas <elem>):int, partimos con dos casos bases, una lista vacia y una lista con una sola hoja. Cuando una lista es vacia, no se suman hojas al total, cuando solo tiene una hoja, se suma esta.

  Ahora bien, cuando hay mas elementos, seguimos el **planetamiento general**, sumamos el numero de hojas del primer hijo con el numero de hojas del resto de hijos. Este planteamiento se realizara sucesivamente hasta que tengamos que el primer hijo es una unica hoja o una lista vacia y el resto es una lista vacia.

  Es aplicable de diversas formas, con el **planteamiento general** si no usamos FOS, o combinado foldr junto con map, usando una lambda con llamada recursiva.
- (altura? <elem>):int, tomara el nivel mas alto como la altura de un pseudoarbol. Recordad que la lista inicial representa el primer nivel, ademas de que una lista vacia y un elemento independiente (hoja no en un arbol) estaran en el nivel 0. Con esto tenemos el caso base.

  Si consideremos que todas las sublistas de una lista estan en el nivel inferior a esta, y aplicamos el **planteamiento general** a esto, tenemos el caso general, donde comprobara el maximo entre el primer hijo y el resto de hijos de una lista sucesivamente hasta obtener que el resto de hijos es vacio o una hoja.

  La implementacion es similar a num-hojas, añadiendo una funcion max en el caso general, tanto con la FOS como sin ellas.
- (aplana <elem>):lista, concatena todos los elementos de una lista que sean hojas. Implementable usando una funcion append y el plantemiento general, tambien usando las FOS foldr y map.

- (pertenece-lista? <dato> <lista>):bool, buscas de entre todos las hojas si existe un dato. La implementacion en la teoria considera unicamente un dato como si fuese del tipo hoja, por lo tanto, la forma mas comoda para implementarlo es una funcion de busqueda secuencial con respecto a la lista obtenida de usar la funcion aplana sobre ella.

- (nivel-hoja? <dato> <lista>):int, recorrera toda la lista hasta encontrar el nivel maximo del dato pasado. Para esta funcion se propone una implementacion alternativa.

  En la implementacion que se propone haremos una lista que contenga los niveles de todos los elementos que coincidan con el elemento pero se va a realizar de forma iterativa.

```
(define (hoja? elem)
   (not (list? elem)))

(define (nivel-hoja-iter nivel dato lista)
  (cond
    ((null? lista) (list))
    ((hoja? lista)
     (if (equal? dato lista)
         (list (- nivel 1))
         (list)))
     (else (append (nivel-hoja-iter (+ nivel 1) dato (car lista)) (nivel-hoja-iter nivel dato (cdr lista))))))

(define (nivel-hoja-max dato lista)
  (cond
    ((null? lista) -1)
    ((hoja? lista) (if (equal? dato lista)
          0
          -1))
    (else (foldr max -1 (nivel-hoja-iter 1 dato lista)))))
```

  Lo unico confuso que tiene este codigo es que cuando se encuentra un elemento, se añade restando uno al nivel actual. Eso se hace para contrarestar el hecho de que para poder acceder a un elemento del primer hijo debemos suponer que accedemos a una sublista, de ahi que aumentemos el nivel, pero si se da que esta elemento es una hoja y no una lista, volvemos al nivel anterior que es el que corresponde. Por ultimo, tendremos una funcion que es la que el usuario utilizara en la que se comprueba si tratamos con una hoja o una lista vacia y, si se que es asi, solo podra devolver -1 o 0 segun si es equivalente al dato. En otro caso, usara una FOS del tipo foldr para comprobar el maximo de todos los niveles que coincidan con el dato.

- Y no pongo mas por pereza.

4.2. Árboles

Ya hemos trabajado con arboles en PED pero vamos a hacer un repaso.

Definición de árboles en Scheme

Un árbol es una estructura de datos definida por un valor raíz, que es el padre de toda la estructura, del que salen otros subárboles hijos.

Un árbol se puede definir recursivamente de la siguiente forma:

- Una colección de un dato (el valor de la raíz del árbol) y una lista de hijos que también son árboles.

- Una hoja será un árbol sin hijos (un dato con una lista de hijos vacía).

Para representar un arbol con n hijos, usaremos listas de n+1, el primer elemento sera la raiz y los n restantes los hijos contenidos en listas. Trataremos una lista vacia como un arbol vacio,

y un hijo sera una hoja si solo tiene arboles vacios (si hacemos un cdr a una lista, obtenemos una lista vacia, teniendo en car la raiz). Revisad el seminario para una imagen mas clara de la representacion.

Vamos a definir la *barrera de abstraccion*, es decir, las funciones basicas para manejar el tipo de dato arbol, todas ellas con el sufijo *-arbol*. Distinguiremos entre selectores, obtienen un elemento del arbol, y constructores, construyen el arbol... duh.

Selectores

- (dato-arbol arbol):dato devuelve la raiz de un arbol, se implementa usando la funcion car.
- (hijos-arbol arbol):lista devuelve una lista con los hijos de un arbol que, a su vez, son todos arboles. Se implementa con la funcion cdr.
- (hoja-arbol? arbol):bool comprueba si un arbol es un nodo hoja comprobando si la lista de los hijos es vacia. Y la implementacion es tal cual la he dicho.
- El desplazamiento simple entre los hijos de un arbol es usando las funciones car para acceder al hijo y cdr para avanzar al siguiente grupo de hijos, pudiendo usar las funciones c???r.

Constructores

- (nuevo-arbol raiz lista-arboles):arbol crea un arbol concatenando la raiz a la lista arboles usando un cons.

Cabe decir que la barrera de asbtraccion es importante definirla para un tipo de dato.

Funciones recursivas sobre árboles

Vamos a dejar definidas las implementaciones de diversas funciones. Todas estas recursivas, siguiendo un patron de recusrividad mutua junto con una funcion auxiliar.

El patron que seguiran sera el siguiente:

1. Obtener el dato del arbol y la lista de hijos (a la que llamaremos bosque). A la funcion que se aplica sobre el arbol la denotaremos con *-arbol* y la que se aplica sobre el bosque sera la funcion auxiliar.
2. Evaluara sobre la raiz y el resultado aplicar la funcion sobre el bosque, en este caso, se hace una llamada a la funcion auxiliar.
3. La llamada auxiliar comprobara si el bosque esta vacio, en el caso de que no lo este, evaluara sobre el resultado de aplicar la funcion *-arbol* al primer elemento del bosque (un arbol) y la funcion auxiliar sobre el resto de elementos, que es a su vez un bosque.

Este patron se puede realizar usando recursion mutua o en algunos casos, combinacion entre las FOS map y foldr principalmente.

Veamos las funciones propuestas:

- (suma-datos-arbol arbol): devuelve la suma de todos los nodos.

Version con recursion mutua

```
(define (suma-datos-arbol arbol)
   (+ (dato-arbol arbol)
      (suma-datos-bosque (hijos-arbol arbol))))
```

```
(define (suma-datos-bosque bosque)
   (if (null? bosque)
       0
       (+ (suma-datos-arbol (car bosque)) (suma-datos-bosque (cdr bosque)))))
```

Version con FOS

```
(define (suma-datos-arbol-fos arbol)
   (foldr +
          (dato-arbol arbol)
          (map suma-datos-arbol-fos (hijos-arbol arbol))))
```

- (to-list-arbol arbol): devuelve una lista con los datos del árbol.

Version con recursion mutua

```
(define (to-list-arbol arbol)
   (cons (dato-arbol arbol)
         (to-list-bosque (hijos-arbol arbol))))
```

```
(define (to-list-bosque bosque)
   (if (null? bosque)
       '()
       (append (to-list-arbol (car bosque))
               (to-list-bosque (cdr bosque)))))
```

Version con FOS

```
(define (to-list-arbol-fos arbol)
    (cons (dato-arbol arbol)
          (foldr append '() (map to-list-arbol-fos (hijos-arbol arbol)))))
```

- (cuadrado-arbol arbol): eleva al cuadrado todos los datos de un árbol manteniendo la estructura del árbol original.

Version con recursion mutua

```
(define (cuadrado-arbol arbol)
   (nuevo-arbol (cuadrado (dato-arbol arbol))
                (cuadrado-bosque (hijos-arbol arbol))))
```

```
(define (cuadrado-bosque bosque)
   (if (null? bosque)
       '()
       (cons (cuadrado-arbol (car bosque))
             (cuadrado-bosque (cdr bosque)))))
```

Version con FOS

```
(define (cuadrado-arbol-fos arbol)
   (nuevo-arbol (cuadrado (dato-arbol arbol))
                (map cuadrado-arbol-fos (hijos-arbol arbol))))
```

- (map-arbol f arbol): devuelve un árbol con la estructura del árbol original aplicando la función f a subdatos.

Version con recursion mutua

```
(define (map-arbol f arbol)
   (nuevo-arbol (f (dato-arbol arbol))
                (map-bosque f (hijos-arbol arbol))))
```

```
(define (map-bosque f bosque)
   (if (null? bosque)
       '()
       (cons (map-arbol f (car bosque))
             (map-bosque f (cdr bosque)))))
```

Version con FOS

```
(define (map-arbol-fos f arbol)
  (nuevo-arbol (f (dato-arbol arbol))
               (map (lambda (x)
                      (map-arbol-fos f x)) (hijos-arbol arbol))))
```

- (altura-arbol arbol): devuelve la altura de un árbol. Antes de implementarla, recordemos los siguientes conceptos:
  - Longitud de un camino entre dos nodos: número de aristas.

- Altura de un nodo: longitud del camino más largo del nodo a una hoja.
- Profundidad de un nodo: longitud del camino de la raíz al nodo.
- Profundidad de un árbol: profundidad del nodo más profundo.
- Nivel de un nodo: número de predecesores.
- Altura de árbol: altura de la raíz.

Version con recursion mutua

```
(define (altura-arbol arbol)
   (if (hoja-arbol? arbol)
       0
       (+ 1 (altura-bosque (hijos-arbol arbol)))))
```

```
(define (altura-bosque bosque)
    (if (null? bosque)
        0
        (max (altura-arbol (car bosque))
             (altura-bosque (cdr bosque)))))
```

Version con FOS

```
(define (altura-arbol-fos arbol)
   (if (hoja-arbol? arbol)
       0
      (+ 1  (foldr max
                   0
                   (map altura-arbol-fos (hijos-arbol arbol))))))
```

4.3. Arboles binarios

Partiendo de los conceptos que ya conocemos para los arboles, vamos a ver el caso en el que tendremos un arbol con dos hijos, por lo tanto nuestra lista tendra 3 elementos.

Definición de arboles binarios en Scheme

Este arbol es el llamado arbol binario, compuesto por tres elementos. Estos son:

- Raiz: contiene el dato del arbol
- Arbol izquierdo: una lista que representa al arbol izquierdo, esta puede ser una lista vacia o no.
- Arbol derecho: una lista que representa al arbol derecho, esta puede ser una lista vacia o no.

Tanto el arbol izquierdo como el derecho son binarios. Llamaremos arbol vacio a una lista vacia y, por lo tanto, un nodo hoja sera aquel cuyos hijos sean arboles vacios.

De igual manera que hicimos con los arboles, vamos a definir la barrera de abstraccion para los arboles binarios. Terminamos todos los nombres de las funciones con el sufijo *-arbolb* (árbol binario).

Selectores

- (dato-arbolb arbol):dato devuelve la raiz de un arbol, se implementa usando la funcion car.
- (hijo-izq-arbolb arbol):lista devuelve el arbol izquierdo. Se implementa con la funcion cadr.
- (hijo-der-arbolb arbol):lista devuelve el arbol derecho. Se implementa con la funcion caddr.
- arbolb-vacio:(list) un identificador a una lista vacia.

- (vacio-arbolb? arbol):bool comprueba si un arbol es vacio comparandolo con una lista vacia.
- (hoja-arbolb? arbol):bool comprueba si un arbol es un nodo hoja comprobando si tanto el arbol izquierdo como el derecho son vacios.

Constructores

- (nuevo-arbolb raiz hijo-izq hijo-der):arbol crea un arbol usando una list sobre los tres elementos en orden.

Funciones recursivas sobre arboles binarios

- (suma-datos-arbolb arbol): devuelve la suma de todos los nodos.

```
(define (suma-datos-arbolb arbol)
   (if (vacio-arbolb? arbol)
      0
      (+ (dato-arbolb arbol)
         (suma-datos-arbolb (hijo-izq-arbolb arbol))
         (suma-datos-arbolb (hijo-der-arbolb arbol)))))
```

- (to-list-arbolb arbol): devuelve una lista con los datos del árbol.

```
(define (to-list-arbolb arbol)
   (if (vacio-arbolb? arbol)
      '()
      (cons (dato-arbolb arbol)
            (append (to-list-arbolb (hijo-izq-arbolb arbol))
                    (to-list-arbolb (hijo-der-arbolb arbol))))))
```

- (cuadrado-arbolb arbol): eleva al cuadrado todos los datos de un árbol manteniendo la estructura del árbol original.

```
(define (cuadrado-arbolb arbol)
   (if (vacio-arbolb? arbol)
      arbolb-vacio
      (nuevo-arbolb (cuadrado (dato-arbolb arbol))
                    (cuadrado-arbolb (hijo-izq-arbolb arbol))
                    (cuadrado-arbolb (hijo-der-arbolb arbol)))))
```

4.4. Prácticas relacionadas con este temario

texto

Práctica 6

texto

Práctica 7

texto

Bloque de Swift

te

5. Programacion funcional en Swift

te

5.1. Practicas relacionadas con este temario

te

Practica 8

Ejercicio 1 - Practica 8

a) Implementa en Swift la función recursiva *prefijos(prefijo:palabras:)* que recibe una cadena y un array de palabras. Devuelve un array de Bool con los booleanos resultantes de comprobar si la cadena es prefijo de cada una de las palabras de la lista.

b) Implementa en Swift la función recursiva *parejaMayorParImpar(numeros:)* que recibe un array de enteros positivos y devuelve una pareja con dos enteros: el primero es el mayor número impar y el segundo el mayor número par. Si no hay ningún número par o impar se devolverá un 0.

Ejercicio 2 - Practica 8

a) Implementa en Swift la función recursiva compruebaParejas(_:funcion:) con el siguiente perfil:

([Int], (Int) -> Int) -> [(Int, Int)]

La función recibe dos parámetros: un Array de enteros y una función que recibe un entero y devuelve un entero. La función devolverá un array de tuplas que contiene las tuplas formadas por aquellos números contiguos del primer array que cumplan que el número es el resultado de aplicar la función al número situado en la posición anterior.

b) Implementa en Swift la función recursiva coinciden(parejas: [(Int,Int)], funcion: (Int)->Int) que devuelve un array de booleanos que indica si el resultado de aplicar la función al primer número de cada pareja coincide con el segundo.

Ejercicio 3 - Practica 8

Supongamos que estamos escribiendo un programa que debe tratar movimientos de cuentas bancarias. Define un enumerado Movimiento con valores asociados con el que podamos representar:

- Depósito (valor asociado: (Double))
- Cargo de un recibo (valor asociado: (String, Double))
- Cajero (valor asociado: (Double))

Y define la función aplica(movimientos:[Movimiento]) que reciba un array de movimientos y devuelva una pareja con el dinero resultante de acumular todos los movimientos y un array de Strings con todos los cargos realizados.

Ejercicio 4 - Practica 8

Implementa en Swift un tipo enumerado recursivo que permita construir árboles binarios de enteros. El enumerado debe tener

- un caso en el que guardar tres valores: un Int y dos árboles binarios (el hijo izquierdo y el hijo derecho)
- otro caso constante: un árbol binario vacío

Llamaremos al tipo *ArbolBinario* y a los casos *nodo* y *vacio*.

Impleméntalo de forma que el siguiente ejemplo funcione correctamente:

let arbol: ArbolBinario = .nodo(8, .nodo(2, .vacio, .vacio), .nodo(12, .vacio, .vacio))

Implementa también la función *suma*(arbolb:) que reciba una instancia de árbol binario y devuelva la suma de todos sus nodos

Ejercicio 5 - Practica 8

Implementa en Swift un tipo enumerado recursivo que permita construir árboles de enteros usando el mismo enfoque que en Scheme: un nodo está formado por un dato (un Int) y una colección de árboles hijos. Llamaremos al tipo Arbol.

Impleméntalo de forma que el siguiente ejemplo funcione correctamente:

```
1 let arbol1 = Arbol.nodo(1, [])
2 let arbol3 = Arbol.nodo(3, [arbol1])
3 let arbol5 = Arbol.nodo(5, [])
4 let arbol8 = Arbol.nodo(8, [])
5 let arbol10 = Arbol.nodo(10, [arbol3, arbol5, arbol8])
```

Ejercicio 6 - Practica 8

a) Define la función maxOpt(_ x: Int?, _ y: Int?) -> Int? que devuelve el máximo de dos enteros opcionales. En el caso en que ambos sean nil se devolverá nil. En el caso en que uno sea nil y el otro no se devolverá el entero que no es nil. En el caso en que ningún parámetro sea nil se devolverá el mayor.

b1) Escribe una nueva versión del ejercicio 1b) que permita recibir números negativos y que devuelva una pareja de (Int?, Int?) con nil en la parte izquierda y/o derecha si no hay número impares o pares.

b2) Escribe la función sumaMaxParesImpares(numeros: [Int]) -> Int que llama a la función anterior y devuelve la suma del máximo de los pares y el máximo de los impares. El array de números tendrá como mínimo un elemento, por lo que el valor devuelto por la función será un Int (no será Int?).

Practica 9

Ejercicio 1 - Practica 9

a) Indica qué devuelven las siguientes expresiones:

a.1)

```
let nums = [1,2,3,4,5,6,7,8,9,10]
nums.filter{$0 % 3 == 0}.count
```

a.2)

```
let nums2 = [1,2,3,4,5,6,7,8,9,10]
nums2.map{$0+100}.filter{$0 % 5 == 0}.reduce(0,+)
```

a.3)

```
let cadenas = ["En", "un", "lugar", "de", "La", "Mancha"]
cadenas.sorted{$0.count < $1.count}.map{$0.count}
```

a.4)

```
let cadenas2 = ["En", "un", "lugar", "de", "La", "Mancha"]
cadenas2.reduce([]) {
    (res: [(String, Int)], c: String) -> [(String, Int)] in
        res + [(c, c.count)]}.sorted(by: {$0.1 < $1.1})
```

b) Explica qué hacen las siguientes funciones y pon un ejemplo de su funcionamiento:

b.1)

```
func f(nums: [Int], n: Int) -> Int {
    return nums.filter{$0 == n}.count
}
```

b.2)

```
func g(nums: [Int]) -> [Int] {
    return nums.reduce([], {
        (res: [Int], n: Int) -> [Int] in
            if !res.contains(n) {
                return res + [n]
            } else {
                return res
            }
    })
}
```

b.3)

```
func h(nums: [Int], n: Int) -> ([Int], [Int]) {
   return nums.reduce(([],[]), {
       (res: ([Int],[Int]), num: Int ) -> ([Int],[Int]) in
           if (num >= n) {
               return (res.0, res.1 + [num])
           } else {
               return ((res.0 + [num], res.1))
           }
   })
}
```

c) Implementa las siguientes funciones con funciones de orden superior.

c.1) Función `suma(palabras:contienen:)`:

```
suma(palabras: [String], contienen: Character) -> Int
```

que recibe una array de cadenas y devuelve la suma de las longitudes de las cadenas que contiene el carácter que se pasa como parámetro.

c.2) Función `sumaMenoresMayores(nums:pivote:)`:

```
sumaMenoresMayores(nums: [Int], pivote: Int) -> (Int, Int)
```

que recibe un array de números y un número pivote y devuelve una tupla con la suma de los números menores y mayores o iguales que el pivote.

Ejercicio 2 - Practica 9

Define un tipo enumerado con un árbol genérico, tal y como hicimos en el último ejercicio de la práctica anterior, que tenga como genérico el tipo de dato que contiene.

En el siguiente ejemplo vemos cómo debería poderse definir con el mismo tipo genérico un árbol de enteros y un árbol de cadenas:

```
let arbolInt: Arbol = .nodo(53,
                            [.nodo(13, []),
                             .nodo(32, []),
                             .nodo(41,
                                   [.nodo(36, []),
                                    .nodo(39, [])
                                   ])
                            ])
let arbolString: Arbol = .nodo("Zamora",
                               [.nodo("Buendía",
                                      [.nodo("Albeza", []),
                                       .nodo("Berenguer", []),
                                       .nodo("Bolardo", [])
                                      ]),
                                .nodo("Galván", [])
                               ])
```

Define las funciones genéricas *toArray* y *toArrayFOS* que devuelvan un array con todos los componentes del árbol usando un recorrido preorden (primero la raíz y después los hijos). La primera la debes implementar con recursión mutua y la segunda usando funciones de orden superior.

Ejercicio 3 - Practica 9

Implementa en Swift la función imprimirListadosNotas(alumnos:) que recibe un array de tuplas, en donde cada tupla contiene información de la evaluación de un alumno de LPP (nombreAlumno, notaParcial1, notaParcial2, notaParcial3, añosMatriculacion) y que debe imprimir por pantalla los siguientes listados:

- listado 1: array ordenado por nombre del alumno (orden alfabético creciente)
- listado 2: array ordenado por la nota del parcial 1 (orden decreciente de nota)
- listado 3: array ordenado por la nota del parcial 2 (orden creciente de nota)
- listado 4: array ordenado por año de matriculación y nota del parcial 3 (orden decreciente de año y nota)
- listado 5: array ordenado por nota final (media de los tres parciales, ponderados en: 0,34, 0,33, 0,33) (orden decreciente de nota final)

Las ordenaciones hay que realizarlas usando la función sorted.

NOTA: Para que los listados se muestren formateados con espacios, puedes usar la siguiente función (para ello también debes incluir el import que se indica)

```
import Foundation

func imprimirListadoAlumnos(_ alumnos: [(String, Double, Double, Double, Int)]) {
    print("Alumno   Parcial1   Parcial2   Parcial3  Años")
    for alu in alumnos {
        alu.0.withCString {
            print(String(format:"%-10s %5.2f      %5.2f    %5.2f  %3d", $0, alu.1,alu.2,alu.3,alu.4))
        }
    }
}
```

Ejercicio 4 - Practica 9

Dado el array listaAlumnos del ejercicio anterior, utiliza funciones de orden superior para obtener los datos requeridos en cada caso.

A) Número de alumnos que han aprobado primer parcial y suspendido el segundo

B) Alumnos que han aprobado la asignatura (tienen una nota final >= 5)

C) Nota media de todos los alumnos en forma de tupla (media_p1, media_p2, media_p3)

Ejercicio 5 - Practica 9

Implementa la función construye con el siguiente perfil:

func construye(operador: Character) -> (Int, Int) -> Int

La función recibe un operador que puede ser uno de los siguientes caracteres: +, -, *, / y debe devolver una clausura que reciba dos argumentos y realice la operación indicada sobre ellos.

6. Programacion OO en Swift

te

6.1. Practicas relacionadas con este temario

te

Practica 10

Ejercicio 1 - Practica 10

Un ejercicio de repaso del apartado de clausuras del tema anterior. En este caso, se hace énfasis en las clausuras con estado local mutable.

a) ¿Qué se imprime al ejecutar el siguiente programa? Reflexiona sobre el funcionamiento del código, compruébalo con el compilador y experimenta haciendo cambios y comprobando el resultado.

```
var x = 0

func construye() -> () -> Int {
  var x = 10
  return {
    x = x + 5
    return x
  }
}


func usa(funcion: () -> Int) {
  var x = 20
  print(funcion())
}

let g = construye()
usa(funcion: g)
usa(funcion: g)
```

b) Completa el siguiente código para que compile y funcione correctamente e imprima lo indicado. El hueco puede contener más de una línea de código.

```
var array : [() -> Int] = []

func foo() -> Void {
    var x = 0
    array.append
}

foo()
foo()
print(array[0]()) // Imprime 10
print(array[0]()) // Imprime 20
print(array[1]()) // Imprime 10
```

Ejercicio 2 - Practica 10

a) El siguiente código usa observadores de propiedades y una variable del tipo (estática).

¿Qué se imprime al final de su ejecución? Reflexiona sobre el funcionamiento del código, compruébalo con el compilador y experimenta haciendo cambios y comprobando el resultado.

```
struct Valor {
    var valor: Int = 0 {
        willSet {
            Valor.z += newValue
        }
        didSet {
            if valor > 10 {
                valor = 10
            }
        }
    }
    static var z = 0
}

var c1 = Valor()
var c2 = Valor()
c1.valor = 20
c2.valor = 8
print(c1.valor + c2.valor + Valor.z)
```

b) Escribe un ejemplo de código en el que definas una relación de herencia entre una clase base y una clase derivada. Comprueba en el código que un objeto de la clase derivada hereda las propiedades y métodos de la clase base.

Investiga sobre el funcionamiento de la herencia en Swift. Escribe ejemplos en donde compruebes este funcionamiento. Algunos ejemplos de preguntas que puedes investigar (puedes añadir tú más preguntas):

- ¿Se puede sobreescribir el valor de una propiedad almacenada? ¿Y calculada?
- ¿Se puede añadir un observador a una propiedad de la clase base en una clase derivada?
- ¿Hereda la clase derivada propiedades y métodos estáticos de la clase base?
- ¿Cómo se puede llamar a la implementación de un método de la clase base en una sobreescritura de ese mismo método en la clase derivada?

Ejercicio 3 - Practica 10

Tenemos que escribir un programa que permita definir resultados de partidos de fútbol y calcular la puntuación de un conjunto de equipos una vez que se han jugado esos partidos.

Escribe código en Swift que permita resolver el problema, utilizando structs.

Ejercicio 4 - Practica 10

En este ejercicio vamos a trabajar con figuras geométricas usando estructuras y clases.

En el ejercicio deberás usar la función para calcular la raíz cuadrada y el valor de la constante matemática pi.

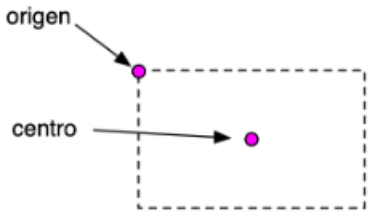
Para usar la función *sqrt* debes importar la librería *Foundation* (import Foundation)

El valor de la constante matemática pi lo puedes obtener con la propiedad Double.pi.

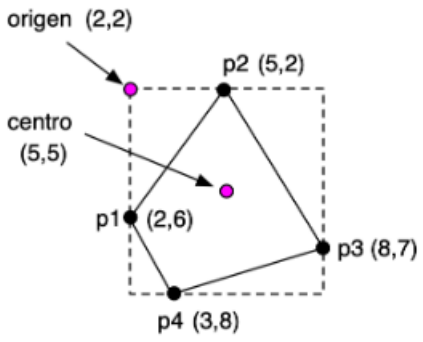
Suponemos que estamos trabajando con coordenadas de pantalla, en las que la coordenada (0,0) representa la coordenada de la esquina superior izquierda de la pantalla. La coordenada Y crece hacia abajo y la coordenada X crece hacia la derecha. Las coordenadas se definirán con números decimales (Double).

Vamos a definir las siguientes estructuras y clases:

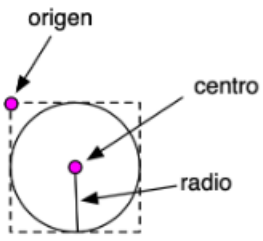
Estructuras: Punto, Tamaño Clases: Figura (clase padre), Cuadrilátero y Circulo (clases derivadas).



Figura



Cuadrilátero



Círculo

Vamos a definir propiedades almacenadas y propiedades calculadas para todas las figuras geométricas.

- Estructuras Punto y Tamaño

  Las debes declarar tal y como aparecen en los apuntes.

- Clase padre Figura:

  Constructor: Figura(origen: Punto, tamaño: Tamaño)

  Propiedades de instancia almacenadas: **origen** (Punto) que define las coordenadas de la esquina superior izquierda del rectángulo que define la figura y **tamaño** (Tamaño) que define el alto y el ancho del rectángulo que define la figura.

  Propiedades de instancia calculadas: **area** (Double, solo lectura) que devuelve el área del rectángulo que engloba la figura y **centro** (Punto, propiedad de lectura y escritura). Es el centro del rectángulo que define la figura. Si modificamos el centro se modifica la posición del origen de la figura.

- Clase derivada Cuadrilatero

  Un cuadrilátero se define por cuatro puntos. La figura padre representa el rectángulo que engloba los cuatro puntos del cuadrilátero (ver imagen arriba).

  Constructor: Cuadrilatero(p1: Punto, p2: Punto, p3: Punto, p4: Punto). Los puntos se dan en el orden definido por el sentido de las agujas del reloj, aunque no siempre se empezará por el punto que está situado más a la derecha. Al crear el cuadrilátero deberemos actualizar las propiedades origen y tamaño de la figura. Para calcular estas propiedades deberás obtener las coordenadas x e y mínimas y máximas de todos los puntos.

  Propiedades de instancia almacenadas propias: Los puntos del cuadrilátero p1, p2, p3 y p4.

  Propiedades de instancia calculadas: **centro** (Punto, de lectura y escritura), heredada de la clase padre. El setter modifica la posición de los puntos del cuadrilátero y del origen de la figura, desplazándolos los mismos incrementos en los que ha sido desplazado el centro de la figura y **area** (Double, sólo lectura) que devuelve el área del cuadrilátero.

- Clase derivada Circulo

  Un círculo se define por un centro y un radio. La figura padre representa el cuadrado más pequeño en el que está inscrito el círculo (ver imagen arriba).

  Constructor: Circulo(centro: Punto, radio: Double). Al crear el círculo deberemos actualizar las propiedades origen y tamaño de la figura.

  Propiedades de instancia almacenadas: radio (Double) que contiene la longitud del radio.

  Propiedades de instancia calculadas: **centro** (Punto, de lectura y escritura), heredada de la clase padre y **area** (Double, de lectura y escritura) que devuelve el área del círculo. El setter modifica el tamaño del círculo (su radio), manteniendo el centro en la misma posición.

- Estructura AlmacenFiguras

  Propiedades almacenadas: figuras: array de figuras.

  Propiedades calculadas: **numFiguras** (Int) que devuelve el número total de figuras añadidas y **areaTotal** (Double) que devuelve la suma total de las áreas de todas las figuras añadidas.

Método: **añade**(figura:) que añade una figura al array y **desplaza**(incX: Double, incY: Double): desplaza todas las figuras las dimensiones especificadas incX (incremento en la coordenada X) e incY (incremento en la coordenada Y). Se deberán mover los centros de todas las figuras en estas magnitudes.

Implementa las estructuras anteriores y escribe algún ejemplo de código en el que se creen al menos un cuadrilátero y un círculo, se prueben sus propiedades, se añadan al almacén de figuras y se prueben sus métodos.

7. Anexos

texto

7.1. Seminario de Scheme aplicado a su bloque de contenido

En este apartado agruparemos todas las funciones que hemos visto en Scheme, considerando que lo hemos visto trabajando en Racket (el entorno de programación DrRacket), se entenderá que el lector conoce tanto que es Scheme como DrRacket, por lo tanto se procederá directamente a tratar las funciones de Scheme.

7.2. Conceptos básicos

Propiedades generales

- Las expresiones en Scheme se dan en el formato de notación prefija ($<$*función*$>$ $<arg_1>$ $\ldots$ $<arg_n>$). Se pueden interpretar los *paréntesis abiertos* '(' como evaluadores o lanzadores de la función que se describiría a continuación. La funcion se llamara **operador** y los argumentos **operandos**.

- Scheme evalúa una expresión primero evaluando los argumentos y a continuación aplica la función de dicha expresión a los valores obtenidos previamente. Una expresión puede ser el argumento de otra, por lo tanto este proceso se va anidando.

- Los términos función y procedimiento significan lo mismo y se usan de forma intercambiable, devolviendo siempre un valor salvo que se produzca un error que detenga la evaluación.

- Se puede definir variables y funciones.

- Es un lenguaje débilmente tipado, es decir, las variables, los argumentos y las funciones no tienen un tipo declarado, permitiendo usar valores de distintos tipos de datos para asignar sucesivamente a una misma variable o para pasar como parámetro a una misma función.

  Esto no significa que las funciones puedan tratar todo tipo de datos, con lo cual se podría considerar el uso de controladores de datos de entrada.

- Las primitivas de Scheme consisten en un conjunto de tipos de datos, formas especiales y funciones incluidas en el lenguaje.

- Los distintos tipos de datos simples son:

  - Booleanos: verdadero (interpretado con #t) y falso interpretado con #f). En muchas operaciones se considera que cualquier valor distinto a #f es verdadero.

  - Números: La cantidad de tipos numéricos que soporta Scheme es grande, incluyendo enteros de diferente precisión, números racionales, complejos e inexactos.
  - Caracteres: Se soportan caracteres internacionales y se codifican en UTF-8. Se denotará con $\#\backslash char$.

- Los distintos tipos de datos compuestos son:
  - Cadenas: Las cadenas son secuencias finitas de caracteres.
  - Parejas: Es un tipo compuesto formado por dos elementos (no necesariamente del mismo tipo). Se mostrara con la sintaxis '(*elem_iz* . *elem_de*)', expresando las expresiones de dentro a fuera y pueden anidarse parejas.

    NOTA: me he dado cuenta que, en un dato que contenga el elemento '.' se considerara pareja y, por lo tanto, este no seria un elemento, sino un indicador de que se trata de una pareja y no una lista.
  - Listas: Uno de los elementos fundamentales de Scheme, y de Lisp, son las listas. Es un tipo compuesto formado por un conjunto finito de elementos (no necesariamente del mismo tipo). Se pueden construir con *list*, *quote* o usando *(cons <elem> <list>)*. Cabe decir que los *quote* permitirían construir parejas usando el elemento '.' mientras que los *list* no.

- Distinguimos dos estructuras de control, en las cuales se permite seleccionar qué parte de una expresión se evaluara en función a una expresión condicional. Estas estructuras son un if (*if*) y un switch (*cond*)
- En Racket se comentan con ';'.
- Se puede hacer pruebas unitarias incluyendo con (require rackunit), usando las siguientes funciones:
  - (check-true <expresion-cond>)
  - (check-false <expresion-cond>)
  - (check-equal? <expresion> <resultado-esperado>)
- MAS

Tabla de funciones

Definiciones de variables y funciones, sintaxis básica	
(define <var> <valor>) > asigna un valor	(define (<func> <args>) <cuerpo-func>) > implementa una funcion, asignandole un nombre

Condicionales básicos		
(< <arg1> ... <argn>)	(and <cond1> ... <condn>)	(equal? <arg1> <arg2>)
(> <arg1> ... <argn>)	(or <cond1> ... <condn>)	v
(= <arg1> ... <argn>)	(not <cond>)	igualdad de tipo

Primitivas sobre numeros, funciones de redondeo y predicados		
(max <arg1> ... <argn>)	(quotient <arg1> <arg2>) > cociente entero	
(min <arg1> ... <argn>)	(remainder <arg1> <arg2>) > resto entero	
(abs <arg>)	(<sin,cos,tan,asin,acos,ata> <arg>)	
(floor <arg>) entero mayor por debajo		
(ceiling <arg>) entero menor por encima		
(truncate <arg>) entero mas cercano con abs no mayor		
(round <arg>) entero mas cercano		
(positive? <arg>)	(negative? <arg>)	(zero? <arg>)
(even? <arg>)	(odd? <arg>)	(number? <arg>)
(integer? <arg>)	(real? <arg>)	

Operaciones sobre caracteres
(char<? <arg1> ... <argn>)
(char-numeric? <arg>)
(char-alphabetic? <arg>)
(char-whitespace? <arg>)
(char-upper-case? <arg>)
(char-lower-case? <arg>)
(char-upcase <arg>)
(char-downcase <arg>)
(char->integer <arg>)
(integer->char <arg>)

Constructores y operaciones
"cadena" (make-string <cantidad> <car>) (string <car1> ... <carn>) (string? <arg>) (substring <arg> <posi> <posf>) de 0 a n-1 (string-list <arg>) (string-length <arg>) (string-ref <arg> <pos>) (string<? <arg1> ... <argn>) se considera <=,=,>,>=

Operaciones con parejas y listas
(cons <elem1> <elem2>) (car <arg>) (cdr <arg>) (list <elem1> ... <elemn>) (pair? <arg>) (list? <arg>) (null? <arg>) (append <list1>...<listn>) Los *quote o* 'comillas' ('), pueden provocar que se construya una pareja o una lista, si se usa correctamente '.'

Condicional *if*
(if <cond> <expresion if *#t*> <expresion if *#f*>)
En las expresiones se pueden anidar mas expresiones *if*

Condicional *cond*
(cond (<cond1> <expresion if *#t*>) ... (<condn> <expresion if *#t*>) (else <expresion if all *#f*>))

Forma especial Lambda
(lambda (<arg1> ... <argn>) <cuerpo> > #<procedure>, es una funcion anonima en tiempo de ejecucion)
Actuaria como un operando, por lo tanto, la forma correcta de usarlo seria anidado entre parentesis seguido de los operandos necesarios, indicado por los argumentos
(lambda (<arg1> ... <argn>) <cuerpo>) <arg1>...<argn> **v** **(<operador>** <operando1>...<operandon>) > <resultado>

Quote para simbolos
(quote <identificador>) **'<identificador>**
Devuelve un tipo de dato atomico, **symbol**, que son como se tratan a los identificadores de variables y funciones. Los simbolos son identificadores sin evaluar.
(symbol? <arg>)

Quote para expresiones
(quote <expresion>) **'<expresion>**
Las expresiones estan entre parentesis
Parejas: si la expresion es (<elemI> . <elemD>)
Listas: si la expresion es (<elem1> ...<elemn>)
Si los elem son caracteres , sin la sintaxis del tipo carácter, se trataran como simbolos (salvo los numeros)

Funciones de argumentos variables
(define (<func> <arg-nec1>...<arg-necn> . <lista-args>) ;caso base, <lista-args> vacia ;caso general, trata la lista <lista-args>)
-Recursividad no es del todo viable, tendra n argumentos atomicos requeridos y el resto que recibe -En el caso general se puede usar la funcion apply para usar recursividad

Funcion apply
(apply <func> <lista-args>)
-Toma los argumentos en la lista como datos atomicos, es quivalente a hacer (<func> <args1>...<argsn>)

Funciones que devuelven funciones
(define (<func> <arg1-func>...<argn-func>) (lambda(<arg1-lambda>...<argn-lambda>) ;manipula <args-lambda> y <args-func>))
-func es una funcion constructora y si se evalua con los <args-func> devolvera una funcion, la clausura, que espera los <args-lambda> para poder ser evaluada
(define f (<func> <arg1-func>... <argn-func>))***>#<procedure>*** (f <arg1-lambda>...<argn-lambda>)>resultado

FOS map con una lista
(map <func-trans> <lista>) > lista
-aplica la funcion a cada elemento de la lista y las almacena en una lista

FOS map con varias listas
(map <func-trans> <lista1>...<listan>) > lista
-la funcion requiere de n argumentos que los recibe de las n listas (todas del mismo tamaño) y almacena los resultados en una lista

FOS filter
(filter <condicion> <lista>) > lista
-Devolvera una lista con los elementos que cumplan la condicion

FOS exists?
(ormap <condicion> <lista>) > bool

FOS for-all?
(andmap <condicion> <lista>) > bool

Ejemplo de la FOS foldl:

```
(define (Foldl func base lista)
    (if (null? lista)
        base
        (func (car (reverse lista)) (Foldl func base (reverse (cdr (reverse lista)))))
        )
  )
```

(foldl + 0 '(1 2 3)) => (+ 3 (+ 2 (+ 1 (0))))

Ejemplo de la FOS foldr:

```
(define (Foldr func base lista)
    (if (null? lista)
        base
        (func (car  lista) (Foldr func base  (cdr  lista)))
        )
  )
```

(foldr + 0 '(1 2 3)) => (+ 1 (+ 2 (+ 3 (0))))

7.3. Soluciones al coste de la recursion

- Proceso iterativo: con apariencia de funciones recursivas, realmente es una forma de usar esta estructura para avanzar, depositando el resultado en un argumento y usando otro como condicional de parada.
- Memoization: haciendo uso de mayor espacio de memoria, se reduce el costo temporal. Se comprueba si se ha realizado una operacion previamente y se obtiene el resultado de este.

Tabla de funciones

Proceso iterativo	Memoization
(define (<proceso-iter> <arg1>...<argn> <arg-cond>) (<cond con arg-cond> <valor-base>;caso base <expresion con proceso-iter> ;caso general, usa los <args> y el <arg-cond> para acumular el resultado. Parece ser recursividad pero el proceso avanza en lugar de esperar un resultado.))	*implementacion de (crea-diccionario) considerada, no necesita hacerse. (define <identificador> (crea-diccionario)) (put key value <identificador>) ;añade el value con la referencia key (get key <identificador>) ;el value asignado si lo encuentra o #f si no

7.4. Graficos de tortuga en Racket

Requiere la libreria graphics turtles

La libreria se añade con **(require graphics/turtles)**
Las funciones basicas son:

- *(turtles #t)*: abre una ventana y coloca la tortuga en el centro, mirando hacia el eje X (derecha)
- *(clear)*: borra la ventana y coloca la tortuga en el centro
- *(draw d)*: avanza la tortuga dibujando d píxeles

- *(move d)*: mueve la tortuga d píxeles hacia adelante (sin dibujar)
- *(turn g)*: gira la tortuga g grados (positivos: en el sentido contrario a las agujas del reloj)
- *(begin <proceso1>...<proceson>)*: permite realizar una secuencias de ejecuciones, permitiendo realizar dibujos completos en una evaluacion.

7.5. Listas estructuradas como pseudoarboles

Interpretacion de una lista estructura como un pseudoarbol

(list (- 2 1) ('hola))

*** hace referencia a una lista**

una lista puede contener otras listas, procedimientos y datos como elementos

el primer hijo estara en el car y el resto en el cdr

Para mayor comprension, diremos que el nivel es el numero de antecesores

Funciones para pseudoarboles

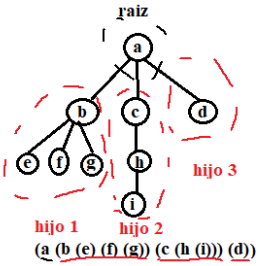
(hoja? <elem>)=> bool
(plana? <elem>)=>bool
(estructurada? <elem>)=>bool
(num-hojas <elem>)=>int
(altura <elem>)=>int
(aplana <lista>)=>lista
(pertenece-lista? <dato> <lista>)=>bool
(nivel-hoja <dato> <lista>)=>int

Recordatorios

hoja: elementos de una lista que no son del tipo lista
lista plana: lista cuyos elementos son todos hojas.
lista estructurada: lista con al menos un elemento siendo del tipo lista

7.6. Listas estructuradas como arboles

Intepretacion de una lista como un arbol



-cada hijo se considera otro arbol
-una lista tendra n+1 elementos para representar un arbol de n hijos
-un nodo hoja tambien es considerado arbol por definicion de hijo
-una lista de arboles de denota como un bosque

Barrera de abstraccion:

Selectores

(**dato-arbol** árbol) ⟶ dato

(**hijos-arbol** árbol) ⟶ lista de árboles

(**hoja-arbol?** árbol) es #t cuando el árbol no tiene hijos

Constructores

(**nuevo-arbol** raiz lista-arboles) ⟶ **arbol**

Funciones recursivas sobre arboles:

-(**suma-datos-arbol** arbol):**int** devuelve la suma de todos los nodos, solo aplicable a arboles con numeros

-(**to-list-arbol** arbol):**list** devuelve una lista con los datos del arbol

-(**cuadrado-arbol** arbol):**arbol** eleva al cuadrado todos los datos de un arbol manteniendo su estructura

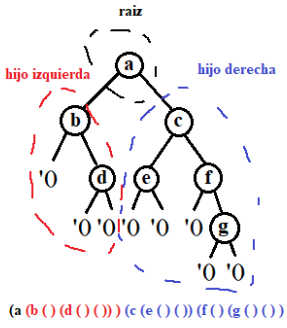
-(**map-arbol** f arbol):**arbol** devuelve un arbol con la misma estructura donde todos los datos has sindo transformados por la funcion f

-(**altura-arbol** arbol):**arbol** devuelve la altura de un arbol

Todas siguen un patron similar de recursion mutua, defininiendo una segunda funcion con el sufijo bosque

7.7. Listas estructuradas como arboles binarios

Intepretacion de una lista como un arbol binario



Barrera de abstraccion:

Selectores

(**dato-arbolb** arbol) ⟶ raiz
(**hijo-izq-arbolb** arbol) ⟶ arbol
(**hijo-der-arbolb** arbol) ⟶ arbol
(**vacio-arbolb?** arbol) ⟶ bool
(**hoja-arbolb?** arbol) ⟶ bool
arbolb-vacio ⟶ (list)

Constructores

(**nuevo-arbolb** dato hijo-izq hijo-der) ⟶ arbol

Funciones recursivas sobre arboles:

-(**suma-datos-arbolb** arbol):**int** devuelve la suma de todos los nodos, solo aplicable a arboles con numeros

-(**to-list-arbolb** arbol):**list** devuelve una lista con los datos del arbol

-(**cuadrado-arbolb** arbol):**arbol** eleva al cuadrado todos los datos de un arbol manteniendo su estructura

Todas siguen un patron similar de recursion mutua, defininiendo una segunda funcion con el sufijo bosque

7.8. Seminario de Swift aplicado a su bloque de contenido

Con Swift, donde puedes probar comandos *aqui*, se trata de un lenguaje con las siguientes caracteristicas:

- Compilado
- De proposito general
- Multi-paradigma (veremos funcional y Orientado a Objetos)
- Fuertemente tipado con la tecnica de inferencia de tipos

Referencias

[1] *Tema 1: Historia y conceptos de los lenguajes de programación*

[2] *Tema 2: Programación funcional*

[3] *Tema 3: Procedimientos recursivos*

[4] *Tema 4: Estructuras recursivas*

[5] *Seminario de Scheme*

[6] *Seminario de Swift*