Unidad 3

Clase 1:

Paradigma funcional: paradigma declarativo basado en un modelo de composición de funciones matemáticas. Un cálculo es la entrada del siguiente.

Características:

- **Expresa QUÉ** hay que calcular, sin especificar tanto el CÓMO.
- **No hay** celda de **memoria asignable** o modificable.

  Existen **valores intermedios** resultados de computos intermedios, útiles para cálculos siguientes.

- **Transparencia referencial:** el resultado devuelto por una función sólo depende de los argumentos que se le pasan en la llamada.
- Los valores resultantes son inmutables, **no hay cambio de estado**.
- No hay valor de variable, la **repetición** se hace con recursividad.
- **Utiliza tipos de datos genéricos**, permite hacerlo más flexible y hacer software genérica, es una forma de hacer **polimorfismo** en funcional.
- **Recursividad:** basada en la inducción, se ve expresada en las listas y sus operaciones.
- **Posibilidad de tratar a las funciones como datos**: gracias a la definición de funciones de orden superior, mejora la abstracción, estructura, modularidad y generalización de las soluciones.
- **Relevan la gestión de memoria**. El almacenamiento se asigna e inicializa implícitamente y se recupera por un recolector de basura con un costo operativo leve.
- **Las abstracciones se hacen con funciones**

Ventajas:

1. Fácil de formular matemáticamente
2. Gestión automática de memoria
3. Código simple
4. Codificación rápida

Desventajas:

1. No es fácil de escalar
2. Difícil de integrar con otras aplicaciones
3. No recomendable para modelar lógica de negocios o tareas transaccionales.

Aplicaciones:

1. **Demostraciones automáticas de teoremas**: por su naturaleza funcional es útil, se puede especificar claramente los problemas matemáticos
2. **Creación de compiladores o analizadores sintácticos**: al ser recursivo es fácil modelarlos.
3. **Problemas de inducción.**
4. **Problemas matemáticos complejos.**
5. **Centros de investigaciones y universidades**.

Lenguajes: Lisp, Haskell, Scheme.

Función: regla de asociación que relaciona 2 o más conjuntos entre sí (Dominio y Codominio).

Una función de A en B es una correspondencia que a cada elemento de A le corresponde un único elemento de B

Dominio: todos los valores que puede tomar el dominio y corresponder con la imagen.

Codominio: todos los valores que puede tomar la función.

Abstracción funcional: expresión por la cual se define una función. Ejemplo: convertir 3*5 en (x) 3 * x o en (x,y) x * y, que también se puede expresar como (x) (y) <expresión>

Función=expresión

Sintaxis de expresión:

<expresión> ::=

<variable> |
<constante> |
(<variable>, ... ,<variable>)
<expresión> | <expresión>
(<expresión>, ... ,<expresión>)

Todas las expresiones denotan un valor

Una expresión puede no permitir argumentos porque ya está evaluada

A. **expresiones atómicas**, que pueden ser <variable> y <constante>
B. **expresiones compuestas**
abstracciones funcionales: (<variable>, . . .,<variable>) <expresión>
aplicaciones funcionales: <expresión> (<expresión>, …. ,<expresión>)

Función de orden superior: uno de sus argumentos devuelve una función.

Permiten abstracción, son más útiles porque parte del comportamiento se especifica al usarlas.

Ej: **(f ,x) f (f x) ó (f) (x) f (f (x))**

Acá f es una expresión/función, x una variable, (x) f (f (x)) una abstracción funcional que posee a x y a la expresión f. f (f (x)) es una abstracción funcional que define la expresión F. Y f (x) es una **aplicación funcional**, define el cuerpo de la abstracción funcional anterior

Esta expresión indica que la imagen de la función sobre la variable f es una función sobre la variable x que aplica dos veces f a x.

Clase 2:

Lambda cálculo:

- sistema formal diseñado para inverstigar la definición de función, la aplicación de funciones y la recursión.
- El más pequeño lenguaje universal de programación.

- Consiste en una regla de transformación simple (sustitución de variables) y un esquema simple para definir funciones.
- Es universal porque cualquier función computable puede ser expresada y evaluada a través de él.

Lambda-expresión: forma compacta de escribir funciones. Ej: f(x) 2x + 1 es x. 2x+1

Evaluación ansiosa o estricta: se evalúan todos los argumentos antes de saber si se van a usar. Lenguajes: Lisp, Scheme

Evaluación perezosa: no se evalúa un argumento de una función hasta que no se necesita. Lenguajes: Haskell, Miranda

Ventajas de la evaluación perezosa: se pueden construir objetos potencialmente infinitos según las necesidades de evaluación,

Datos atómicos: bloque de construcción básico de un lenguaje de programación, tipos de datos elementales. Números y cadenas de texto.

Racket no distingue entre mayúsculas y minúsculas.

Acá termina la teoría como tal de la unidad porque no hay más powers, voy a poner algunas funciones a ver si sirve de algo

Let: (let ((var1 val) [(var2 val)...]) exp1 exp2 ...)

Ej:

- (+ (* 4 4) (* 4 4)) ⇒ 32
- (let ((a (* 4 4)))
  (+ a a)) ⇒ 32
- (let ((list1 '(a b c)) (list2 '(d e f)))
  (cons (cons (car list1) (car list2))
  (cons (car (cdr list1)) (car (cdr list2)))))
  ⇒ ((a . d) b . e)

Se pueden anidar. Ej:

- (let ((a 4) (b -3))
  (let ((a-squared (* a a))
  (b-squared (* b b)))
  (+ a-squared b-squared))) → 25

Lambda: para crear procedimientos: (lambda (var ...) exp1 exp2 ...)

Ej:

- ((lambda (x) (+ x x)) (* 3 4)) ⇒ 24

Se puede usar con let. Ej:

```
(let ((double (lambda (x) (+ x x))))
     (list (double (* 3 4))
           (double (/ 99 11))
           (double (- 2 7)))) ⇒ (24 18 -10)
```

Let*: permite realizar asignaciones donde la definición de las variables internas pueden ver a las variables externas. Ej:

```
(let* ((x (* 5.0 5.0))
       (y (- x (* 4.0 4.0))))
  (sqrt y)) ⇒ 3.0
```

La **diferencia entre let y let*** es que en let las asignaciones se hacen cuando terminan de leerse todas las asignaciones, en let* apenas se lee una asignación ya se hace.

letrec: igual que el let, pero este admite recursividad en su cuerpo, let y let* no.

```
EJ:
(letrec ((sum (lambda (ls)
                (if (null? ls)
                0
                (+ (car ls) (sum (cdr ls)))))))
  (sum '(1 2 3 4 5)))
```

Definiciones de alto nivel: (define cosa). Puede ser una función lambda, una constante, lo que sea. Ej:

(define sandwich "milanesa-tomate-y-lechuga")

sandwich ⇒ "milanesa-tomate-y-lechuga"

(define double-any (lambda (f x) (f x x)))

(double-any + 10) ⇒ 20

(**car lista**) da el primer elem de la lista

(cdr lista) da la lista resultante de sacar el primer elem de la lista

Quote (') le dice a racket que trate como un símbolo al identificador

(cons elemento lista): construye lista, añade el primero al comienzo del segundo

Lista propia: lista vacía o cuyo cdr es una lista vacía.

Lista impropia: se marca la separación de elementos por puntos.

(cdr '(a . b)) → b

(cdr '(a b)) → (b)

Predicados finalizan en ?: retornan #t o #f

Ejemplo: eq?, zero?, string=?

If:

```
(if (<test>)
    (<verdad>)
    (<falso>)
)
```

null=? Devuelve si es una lista vacía

diferencia entre eqv?, eq?, = y equal?

El **=** mira si 2 números son iguales.

El **eq?** mira si 2 elementos representan el mismo objeto en memoria

El **eqv?** Es el eq?, pero funciona siempre, el eq? puede tirar cortes. Este no funciona para comparar listas o vectores

El **equal?** Es exactamente lo mismo que el eqv?, pero este sirve para listas, pares y vectores

Otros predicados son:

- **pair?**
- **symbol?**
- **number?**
- **string?**

cond:permite realizar múltiples test/acciones (es como el switch): (cond (test exp) ... (else exp))

```
Ej: (define sign
      (lambda (n)
        (cond
          ((< n 0) -1)
          ((> n 0) +1)
          (else 0))))
```

Vectores: (vector objetos), acá el # actúa como el ' en listas.

(make-vector n) hace un vector de n pos

(**make-vector n obj)** hace un vector de n pos con todas valiendo obj.

(vector-lenght vector): da la cant de elementos.

(vector-ref vector n) retorna el elemento en la pos n del vector

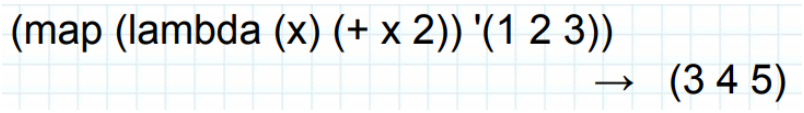
(vector-set! vector n obj) le da el valor obj a la pos n del vector

(vector-fill! Vector obj) reemplaza cada elemento del vector por obj.

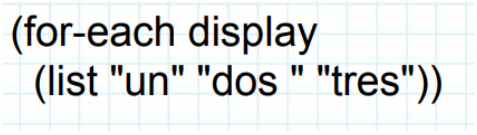
(vector->list vector) convierte en lista al vector, existe el opuesto para lista.

Map: aplica un procedimiento a todos los elementos de la lista y devuelve la lista modificada (puede tener varios argumentos).

(map (procedimiento) (lista))



For-each: igual que map, pero devuelve void. (for-each proceso lista)



Define-struct: crea un struct. Tiene varias operaciones.

Un constructor: make-<nombredelstruct> miembros

Métodos selectores: <nomdelstruct>-<campo> struct. Esto me devuelve esos campos

Métodos accesores: set-<nomstruc>-<campo>! Struct valornuevo Esto me cambia ese campo con el valor que yo le ingreso.

```
;un struct puede definirse, crearse, mostrar sus miembros y modificar sus miembros

(define-struct punto (posx posy))     ;creo el struct. (define-struct nombre (miembros))

;(make-punto 1 2)                      ;creo un punto, retorna #<punto>   (make-nombre miembros)

;(punto-posx (make-punto 1 2))         ;retorno las coords x e y del punto con coords 1 2     (nombrestruct-miembro1 elemento)
;(punto-posy (make-punto 1 2))

;(define punto (make-punto 1 2))       ;defino un punto con ese valor

;(set-punto-posx! punto 22)            ;cambio los valores del punto         (set-nombre-miembro! nombre valornuevo)
;(set-punto-posy! punto 44)
;(punto-posx punto)                    ;muestro los nuevos valores
;(punto-posy punto)
```

Read-char: permite leer un carácter desde la consola

Read-line: lee una línea de caracteres y devuelve un string

Write-char: escribe un carácter en la consola

Write: escribe la expresión en la consola en formato máquina Ej: los strings con comillas dobles y los caracteres precedidos con #\

Display: muestra el resultado de una expresión

Open-input-file: abre un archive y devuelve un puerto de lectura

Open-output-file: abre un archive y devolve un puerto de escritura

close-input-port / **close-output-port**: cierran los puertos.

Unidad 4

Clase 1

Programación lógica:

- Enfocada en el razonamiento lógico y la inferencia automática.

Campos de aplicación:

- Sistemas expertos: emulan a un humano experto en un dominio de conocimiento
- Demostración automática de teoremas
- Inteligencia artificial: sirve para representar conocimiento, razonar y llegar a conclusiones lógicas.
- Cualquier otro campo de acción donde se pueda expresar el problema como un cuerpo de predicados lógicos.

Programa: compuesto por conocimientos (axiomas lógicos). Para ejecutar un programa se provee un axioma nuevo y el programa lo intenta probar con los axiomas existentes.

Términos: denotan objetos dentro del problema. Pueden ser:

- Constantes: empiezan con minúscula.
- Variables: empiezan con _ o con mayúscula.
- Functores: listas, clausulas y eso.

Fórmulas atómicas: afirmaciones simples o relaciones entre entidades del dominio. p(t1,…tn), donde p es un predicado y ti términos.

Inferencia lógica: proceso de deducir conclusiones lógicas a partir de premisas o proposiciones dadas.

Ejemplo:

Premisas: Madre(María) $\wedge$ Hijo(Juan, María)

Conclusion: Ama(María, Juan)

Cláusulas definidas: define relaciones. Formada por cabecera(cuerpo). EJ: padre(tomás, juan).

Objetivos definidos: consultas que el usuario formula al programa para obtener información específica sobre el dominio del problema.

Elementos de un programa PROLOG:

- Base de conocimiento: conjunto de afirmaciones (hechos y reglas) que representan los conocimientos que se poseen sobre el dominio de campo.
- Motor de inferencia: se encarga de la ejecución del programa. Comprueba teoremas usando reglas de inferencia.
- Resolución (intérprete de comandos): permite responder preguntas

Lógica + Control = Programa.

Para programar en prolog: Declarar hechos sobre los objetos y relaciones. Definir reglas sobre los objetos y relaciones. Hacer preguntas sobre los objetos y relaciones.

Hechos: expresan relaciones entre objetos. Ej: tiene(coche,ruedas)

Reglas: hecho que depende de más hechos. p->q. Consiste en cabeza:-cuerpo. La cabeza es un único hecho, el cuerpo pueden ser varios separados por , o ; Ej: sabe(persona):-estudia(persona).

Consultas: herramienta para recuperar información. Si da true, es verdadero, si da false, no se pudo demostrar (podría ser verdadero, solo que prolog no la pudo sacar con la info que le dí). Busca en la base de conocimiento los hechos que coincidan con la pregunta.

Regla de inferencia o principio de resolución: algoritmo que intenta llegar a un absurdo a partir de la negación de la pregunta y los hechos y reglas del programa.

Unificación: mecanismo de prolog para que las variables tomen un valor. Una vez ligada una variable, no se puede cambiar su valor. Es distinto de la asignación en los lenguajes imperativos, acá representa la igualdad lógica.

Variable libre: no unificada, sin valor.

Variable ligada: unificada.

2 términos se unifican si existe una posible asignación de valor de las variables en la que ambos términos son idénticos sustituyendo las variables por dichos valores. Ej: a(X,3) y a(4,Z) unifican con X=4 y Z=3

Si algún término no unifica, ninguna variable queda ligada.

Backtraking: recorrer sistemáticamente todos los caminos posibles. Cuando un camino no conduce a la solución se retrocede al paso anterior para buscar un nuevo camino.

Características:

- Cada solución es el resultado de una secuencia de decisiones.
- Las decisiones pueden deshacerse ya sea porque no lleven a una solución o porque se quieran explorar todas las soluciones (para obtener la solución óptima)
- Existe una función objetivo que debe ser satisfecha u optimizada por cada selección
- Las etapas por las que pasa el algoritmo se pueden representar mediante un árbol de expansión.

Funcionamiento:

- Cuando se va ejecutar un objetivo, Prolog sabe de antemano cuántas soluciones alternativas puede tener.Cada una de las alternativas se denomina punto de elección. Dichos puntos de elección se anotan internamente y de forma ordenada
- Se escoge el primer punto de elección e ejecuta el objetivo eliminando punto de elección en el proceso.
- Si el objetivo tiene éxito se continúa con el siguiente objetivo aplicando estas mismas normas.

- Si el objetivo falla, Prolog da marcha atrás recorriendo los objetivos que anteriormente sí tuvieron éxito (en orden inverso) y deshaciendo las ligaduras de sus variables
- Cuando uno de esos objetivos tiene un punto de elección anotado, se detiene el backtracking y se ejecuta de nuevo dicho objetivo usando la solución alternativa. Las variables se ligan a la nueva solución y la ejecución continúa de nuevo hacia adelante. El punto de elección se elimina en el proceso
- El proceso se repite mientras haya objetivos y puntos de elección anotados

Se puede controlar mediante el corte (!) y el fail.

Corte (!): predicado sin argumentos que siempre se cumple, genera verdadero en la primera ejecución y falla en el backtracking impidiendo el retroceso. Elimina los puntos de elección del predicado que lo contiene.

Motivos para usar el corte:

- Optimizar ejecución: no se explora al pedo con el backtracking puntos de elección que seguro no sirven.
- Facilita la legibilidad y compresión del algoritmo.
- Implementar algoritmos diferentes según la combinación de argumentos de entrada. Algo tipo un switch.
- Conseguir que un predicado tenga una sola solución.

Fallo (fail): predicado sin argumentos que siempre falla, implica la realización del retroceso en el backtracking para generar nuevas soluciones. Sirve por si quiero seguir buscando soluciones aún cuando ya encontré una, forzando el backtracking a seguir.