.UBAfiuba
FACULTAD DE INGENIERÍA

Apuntes
Lenguajes Formales

[75.14/95.48]
2C 2021

Grassano, Bruno
bgrassano@fi.uba.ar

Índice

1. Introducción

El presente archivo contiene los apuntes que fueron tomados a lo largo de la cursada de la materia Lenguajes Formales (75.14/95.48).

La materia es bastante tranquila y muy recomendable para aprender programación funcional. Durante la cursada se tiene un parcial (virtual es oral de media hora donde se cubren todos los temas), y un TP más cerca del final del cuatrimestre. En el TP se implementa un interprete de algún lenguaje, conviene empezarlo con tiempo. (Este cuatrimestre toco de Scheme) Pasada la cursada el final consiste en agregarle alguna característica al interprete realizado en el TP.

Primera y segunda semana

2. Calculo lambda

- Surge en la década del 30, estaban interesados en saber cuando una función es computable.
- Todo es una función en calculo λ , incluso un numero.

Ejemplo: Función sucesor

$$succ(x) = x + 1$$

- succ es el nombre de la función
- x es el parámetro (formal) en la definición de la función
- x+1 es el cuerpo de la función

Si llamo a succ con el numero 4, 4 seria el argumento (efectivo). Argumento se dice en la invocación

Las funciones pueden entonces no tener un nombre, se indican con una flecha

$$x \to x + 1$$

El calculo lambda no tiene entrada salida de forma directa.

Los programas en programación funcional son muchas funciones. En calculo lambda se llaman expresiones.

Programas como expresiones

Un programa funcional consiste en una expresión E que representa tanto al algoritmo como los datos de entrada. Para ejecutarla se le aplican a E reglas de conversión.

$$E[P] \to E[P']$$

- La reducción consiste en remplazar una parte de P de E por otra expresión P'. $P \to P'$ debe estar de acuerdo con las reglas.
- La reducción se repite hasta que la expresión resultante no tenga mas partes que puedan convertirse. Se llama forma normal E* de la expresión E consiste en la salida del programa funcional dado.
- Llamamos combinadores a funciones que combinan algunas reglas de conversión.
- Los sistemas de reducción satisfacen la propiedad de Church-Rosser. Esto establece que la forma normal obtenida es independiente del orden de reducción de los subterminos.
- Tenemos que tener alguna forma de manejar la situación si alguna operación no se puede realizar

Sintaxis del calculo lambda

La sintaxis de una expresión lambda se puede expresar como BNF (Backus-Naur-Form)

$$\begin{aligned} < expresion\lambda > ::= & < variable > | \\ & (\lambda < variable > . < expresion\lambda >) | \\ & (< expresion\lambda >< expresion\lambda >) \end{aligned}$$

Es una de las siguientes 3 opciones

- Una variable (primer termino)
- Una abstracción: con una variable (o parámetro) y un cuerpo que también es una expresión lambda (segundo termino)
- Una aplicación: que tiene un operador (función) y un operando (argumento) que son también ambos expresiones lambda. Se diferencia de antes, en que no va entre paréntesis el argumento, todo va entre paréntesis (tercer termino)

La posición determina que es, a la izquierda el operador (la función), a la derecha el operando (los argumentos) El BNF aporta los <>, ::= , | Entre paréntesis se ponen cosas evaluables. En calculo lambda solo se puede pasar un solo argumento, para pasar varios, se van anidando de a pares.

Convenciones

Aplicando las siguientes reglas se obtiene una expresión equivalente a la original.

$$(((\lambda x.(\lambda y.(yx)))a)b)$$

1. Se omiten paréntesis externos.

$$((\lambda x.(\lambda y.(yx)))a)b$$

2. Se asume que las aplicaciones se asocian a la izquierda.

$$(\lambda x.(\lambda y.(yx)))ab$$

3. Se asume que el cuerpo de las abstracciones se extiende hasta que se cierra un paréntesis o se alcanza el final de la expresión.

$$(\lambda x.\lambda y.yx)ab$$

4. Opcionalmente se pueden contraer múltiples abstracciones lambda (se pueden sacar los lambda intermedios)

$$(\lambda xy.yx)ab$$

Currificacion 'curryng': técnica para invocar una función con menos parámetros de los que esperaría

Variables libres y ligadas

La notación es que se le hacen arcos a los correspondientes parámetros. Se le puede poner arriba la inicial de que tipo es (en español es tienen ambas L, pero en ingles F(ree) y B(ound))

En las variables ligadas es en donde se hace el remplazo del argumento.

Sean ***x***, ***y***, ***z*** variables y ***M***, ***N***, ***P*** expresiones lambda cualesquiera:

La variable ***x*** ocurre *ligada* en la expresión ***N*** si y solo si:

1) $N \equiv \lambda z.M$ siendo $x \equiv z$ o cuando ***x*** ocurre ligada en ***M***
2) $N \equiv M\,P$ donde ***x*** ocurre ligada en ***M*** y/o en ***P***

La variable ***x*** ocurre *libre* en la expresión ***N*** si y solo si:

1) $N \equiv x$
2) $N \equiv \lambda z.M$ siendo $x \neq z$ y donde ***x*** ocurre libre en ***M***
3) $N \equiv M\,P$ donde ***x*** ocurre libre en ***M*** y en ***P***

Ejemplos:

- **λz.x y** — **x** libre, **y** libre
- **λx.x y** — **x** ligada, **y** libre
- **λy.x y** — **x** libre, **y** ligada
- **λx y.x y** — **x** ligada, **y** ligada
- **(λz.z x y) (λx.x)** — **x** libre (operador), **x** ligada (operando), **y** libre, **z** ligada

Reglas de conversión

Son reglas que transforman una expresión en otra.

Alfa

Es un renombre de la variable en una abstracción que tiene la forma $\lambda x.M$ Si la variable y no ocurre libre en M, es posible sustituir por y todas las ocurrencias libres de x en M.

$$\lambda x.M =_\alpha \lambda y.M[y/x]$$

Esta regla solo se puede aplicar con las variables ligadas. La convención de Barendregt dice que no se recomienda que estén de ambos lados una variable con el mismo nombre (la x)

Ejemplos:

1) **λx.x** sustituyendo queda **λy.y**
2) **λx.y x** no se puede aplicar la regla α porque se ligaría la variable **y** que es libre.
   Con cualquier otra variable que no ocurra libre en **M**, sí se podría usar la regla α: **λz.y z**
3) **λx.z x (λu x.x u) v x** sustituyendo queda **λy.z y (λu x.x u) v y**
4) **λx y.x z y** debe convertirse primero en: **λx u.x z u** y luego en: **λy u.y z u**
5) De la expresión **x (λx.x y) (λy.z y)** pueden obtenerse:
   - **x (λt.t y) (λy.z y)**
   - **x (λx.x y) (λu.z u)**
   - **x (λt.t y) (λu.z u)**
   - **x (λx.x y) (λx.z x)** Obs: Evitar (no sigue la convención de Barendregt)

   Pero no puede obtenerse:
   - **z (λz.z y) (λy.z y)**

Beta

Se identifican $\beta - redex$ (expresión reducible β), que son una aplicación cuyo operador es una abstracción (tiene la forma $(\lambda x.M)N$). La regla beta la puedo usar solo para aplicaciones.

Ejemplos de β-redex.

- `(λx.(λu.u) (λv.x v)) ((λt.t t) w)` Una β-*redex* que contiene otras dos
- `(λx.x x) (λy.y y)` Una β-*redex*
- `(λx.x) (λy.y y) z` Una β-*redex*
- `x (λy.y y)` Ninguna β-*redex*

Una expresión lambda que no contiene ninguna β-redex está en forma normal. Mientras no se llegue a la forma normal, puede seguir aplicándose la regla de conversión Beta, que consiste en sustituir por N todas las ocurrencias libres de x en M. Es decir, la regla dice que hay que remplazar en el cuerpo, los parámetros por los argumentos.

$$(\lambda x.M)N =_\beta M[N/x]$$

Aplicaciones:

- `(λu.u z u) a`
  `=β a z a`
- `(λx y.y x) y a`
  `=α (λx z.z x) y a` Obs: Es obligatorio para evitar que `y` se ligue al entrar
  `=β (λz.z y) a`
  `=β a y`
- `(λx.(λu.u) (λv.x v)) ((λt.t t) w)`
  `=β (λu.u) (λv.(λt.t t) w v)`
  `=β λv.(λt.t t) w v`
  `=β λv.w w v`
- `(λt.z) ((λx.x x) (λy.y y))`
  `=β z`
- `(λx.x x) (λy.y y)` Obs: No termina nunca. No tiene forma normal.

Eta

Esta casi no se usa.

Consiste en realizar la reducción de una η-redex que es una abstracción con la forma 'λv.M v' en el cual 'v' **no ocurre libre en M**. (Viendo solo M, independientemente del exterior)

$$\lambda v.Mv =_\eta M$$

- $(\lambda x\ v.x\ v)\ ((\lambda t.t\ t)\ w) =_\beta \lambda v.((\lambda t.t\ t)\ w)\ v =_\beta \lambda v.w\ w\ v =_\eta w\ w$
- $(\lambda v.w\ x\ y\ v)\ z =_\eta w\ x\ y\ z$ Obs: También $(\lambda v.w\ x\ y\ v)\ z =_\beta w\ x\ y\ z$
- $\lambda x.x\ t\ x$ Obs: $\lambda x.M\ x$ no es una η-*redex* con $M \equiv x\ t$ porque x es libre en M

Estrategias de reducción

Call by name

consiste en ir reduciendo siempre la β-redex **mas externa desde la izquierda** y que no este ubicada dentro de una abstracción lamba (cuerpo de una función), hasta llegar a una expresión en forma normal de cabecera (**head normal form**) Puede no coincidir con la forma normal. (Se termina cuando a la izquierda no tengo β-redex)

- $(\lambda u.u\ (\lambda t.t)\ ((\lambda y.y)\ u))\ ((\lambda z.z)\ x)$
  $=_\beta (\lambda z.z)\ x\ (\lambda t.t)\ ((\lambda y.y)\ ((\lambda z.z)\ x))$
  $=_\beta x\ (\lambda t.t)\ ((\lambda y.y)\ ((\lambda z.z)\ x))$

Orden normal

Consiste en ir reduciendo siempre la β-redex mas externa desde la izquierda.

- $(\lambda u.u\ (\lambda t.t)\ ((\lambda y.y)\ u))\ ((\lambda z.z)\ x)$
  $=_\beta (\lambda z.z)\ x\ (\lambda t.t)\ ((\lambda y.y)\ ((\lambda z.z)\ x))$
  $=_\beta x\ (\lambda t.t)\ ((\lambda y.y)\ ((\lambda z.z)\ x))$
  $=_\beta x\ (\lambda t.t)\ ((\lambda z.z)\ x)$
  $=_\beta x\ (\lambda t.t)\ x$

Call by value

Consiste en ir reduciendo siempre la β-redex mas interna desde la izquierda y que no este ubicada dentro de una abstracción lambda.

- $(\lambda u.u\ (\lambda t.t)\ ((\lambda y.y)\ u))\ ((\lambda z.z)\ x)$
  $=_\beta (\lambda u.u\ (\lambda t.t)\ ((\lambda y.y)\ u))\ x$
  $=_\beta x\ (\lambda t.t)\ ((\lambda y.y)\ x)$
  $=_\beta x\ (\lambda t.t)\ x$

Orden aplicativo

Consiste en ir reduciendo siempre la β-redex mas interna desde la izquierda.

```
➢ (λu.u (λt.t) ((λy.y) u)) ((λz.z) x)
  =β (λu.u (λt.t) u) ((λz.z) x)
  =β (λu.u (λt.t) u) x
  =β x (λt.t) x
```

Representación de valores de verdad, funciones lógicas

- If = λp . λq.λr. p q r
- True = λx. λy.x
- False = λx. λy.y
- **If True b c** devuelve b
- **If False b c** devuelve c
- Not = λp.p False True = λp.p (λx. λy.y) (λx. λy.x)
- And = λp.λq.p q False
- Or = λp.λq. p True q
- Xor = λp.λq.p (q False True) q

En base a la definición de funciones, y las reglas de reducción, se llega a formar un lenguaje de programación.

Representación de números, funciones numéricas y relacionales

Se llaman numerales (representaciones de números)

- 0 = λf.λx.x
- 1 = λf.λx.f x
- 2 = λf.λx.f (f x)
- ...

Si contamos la cantidad de f en el cuerpo sabemos a que numero estamos haciendo referencia.

- Succ = λn.λf.λx.f (n f x)
- Pred = λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)
- Add = λm.λn.λf.λx.m f (n f x)
- Sub = λm.λn.n Pred m
- Mul = λm.λn.λf.λx.m (n f) x
- Pow = λm.λn.λf.λx.n m f x

- Fibo = λn.n (λf.λa.λb.f b (Add a b)) (λx.λy.x) (λf.λx.x) (λf.λx.f x)
- IsZero = λn.n (λz.(λx.λy.y)) (λx.λy.x)

Basándonos en el IsZero, podemos encontrar la forma de representar las otras Gte, Lte, Lt, Gt, Eq

- Lte = λx.λy.Iszero (Sub x y)
- Gte = λx.λy.Lte y x
- Lt = λx.λy.Not (Gte x y)
- Gt = λx.λy.Not (Lte x y)
- Eq = λx.λy.And (Lte x y) (Gte x y)
- Ne = λx.λy.Not (Eq x y)

Combinadores

- No tienen variables libres, los usamos para representar ciclos. (El Y y el ω?)
- SKI es un tipo de logica combinatoria. Todo calculo lambda puede reformularse a SKI.
- SKI tiene solo una operación básica, la aplicación.
- Fusión: S = λx.λy.λz.x z (y z)
- Constancia: K = λx.λy.x
- Identidad: K = λx.λy.x

Pares y listas

- Se tienen estructuras y tipos de datos, pares ordenados y listas en calculo lambda.
- Un par ordenado esta compuesto por dos elementos, el primero y el segundo.

$$(a.b) = \lambda s.s\ a\ b$$

- (a.b) es como se veria de forma externa.
- La función Pair se usa para construir pares ordenados.

$$Pair = \lambda x.\lambda y.\lambda s.s\ x\ y$$

Podemos definir la función First y Second.

- First = λp.p True
- Second = λp.p False

La lista vacía se define de la forma:

- Nil = Pair True True o Nil = λz.z

Construimos nodos con CONS. (CONS a (CONS b (CONS c Nil)))
Para verificar si la lista esta vacía, me fijo con Null = First.
Para saber donde esta el final, recorro la lista buscando donde esta el Nil. (Tail)
Al comienzo, si no esta vacia y hacemos First, devuelve False. (se forma en la estructura)

Tercera y cuarta semana

3. APL

- Principios de la década del 60.
- Primeros lenguajes funcionales. Empezó como una notación.
- Tienen 7 características en común como mínimo los lenguajes funcionales.
- Tiene caracteres especiales que representan las funcionalidades. Se pueden resolver problemas de matemática con muy poco código.
- Se puede probar en TryAPL.

Elementos

La gramática tiene 6 elementos (como mínimo). Un solo símbolo ya cambia completamente el significado del programa.

- **Sustantivos**: es un arreglo de rango 0 (escalar o item), de rango 1 (vector o lista), o mayor o igual a 2 (matriz tabla). Son los datos básicamente. Para las matrices le digo de cuanto por cuanto es la matriz.
- **Verbos**: Un verbo es aplicado a un sustantivo, es una función que actúa sobre uno (monódica) o dos sustantivos(diádica) (llamados argumentos). Produce un nuevo sustantivo. Se identifica con un símbolo. *Ejemplos con argumentos: 1)* -4 *cambia signo ;2)* $2-4$ *realiza una resta ;3)* $\iota 3$ (Un argumento a la izquierda, el otro a la derecha)
- **Adverbios**: es un operador que cambia el significado del verbo, los vuelve verbos derivados. *Por ejemplo: / reduce, +/ suma todo lo que venga a la derecha.*
- **Conjunciones**: Es un operador que actúa sobre dos verbos (funciones) y produce un nuevo verbo derivado (función de orden superior).
- **Ligamientos** (**copulas**): Se denota con $\leftarrow$. Permite asignar un elemento gramatical (sustantivo, verbo, etc) en una variable. Entre llaves va una función, es una función anónima. Quiere decir que sus argumentos pueden ser dos como mucho, alfa α entra por la izquierda, y omega ω por la derecha. Con una copula las asignamos para usarlas varias veces.
- **Puntuación**: La única puntuación son los paréntesis. Están para forzar un orden de evaluación distinto del predeterminado (las expresiones de APL se leen de derecha a izquierda va aplicando los verbos sobre los sustantivos).

Particularidades

- Los tipos no se declaran, se asocian a lo mas cercano que tienen.
- El orden de evaluación es de derecha a izquierda. Todos los verbos tienen igual prioridad.
- Resultado de un ligamento, es el valor de su parte derecha
- La selección de items de un arreglo se realiza con [y] dentro de los cuales se colocan los arreglos de posiciones correspondientes a cada dimensión del arreglo (separados por ;). *'ABC'* [*3 2 1*] *resulta 'CBA'*

- Los tipos de los argumentos de las funciones diádicas pueden ser iguales o distintos. *Si son dos escalares, el resultado es otro escalar. Si son un escalar y un arreglo, el resultado es un arreglo con los resultados de aplicar la función diádica entre el escalar y cada uno de los ítems del arreglo. Si son dos arreglos de igual dimensión, el resultado es un arreglo con los resultados de aplicar la función diádica entre los elementos homólogos de ambos arreglos. Si son dos arreglos de distintas dimensiones, el resultado puede ser indefinido o no, dependiendo de la función.*
- Los vectores empiezan con 1, no con 0.
- Los identificadores identifican una sola cosa a la vez, se van pisando.

4. Programación Funcional

Tipo de programación declarativa, mas énfasis en la evaluación de expresiones.
Tiene 7 características principales.

- **Programas como funciones**: Un conjunto de funciones constituyen un programa, y la ejecución consiste en aplicar los argumentos a las funciones.
- **Funciones Puras**: Responsabilidad del programador, uso del APL. Ante los mismos argumentos siempre retorna el mismo valor, conlleva la transparencia referencial (la función puede reemplazarse por su retorno). Además, no debe de tener efectos colaterales, no hay efectos visibles sobre el ambiente desde la cual se invoca (*Con entrada salida ya no es pura*).
- **Datos inmutables**: Las funciones reciben y devuelven cosas, no modifican lo que reciben. (Son persistentes, esto facilita la programación concurrente). En FP no hay variables, por lo que no podemos verlo.
- **Funciones de primera clase**: Tiene que ser tratada como otro tipo de dato. Tienen los mismos derechos que las demás entidades del lenguaje. Todas las funciones son de este tipo. *Si a un numero lo puedo guardar en una variable, a una función también.*

  En FP pueden ser recibidas y retornadas en determinados contextos.
- Funciones de orden superior: Son capaces de recibir a otras funciones o retornar otra función. Solo algunas son de este tipo. (*Las típicas son: map, reduce, bind*)

  Paper recomendado: Revenge of the nerds - Lisp - Paul Graham
- Composición de funciones: Armar funciones mas complejas a partir de mas simples. En FP esta el circulo para componer.
- Recursividad: No es propia exclusiva de la programación funcional. *Si bien se puede resolver de primera de esta forma en la materia, hay que pensar la forma de refactorizar para llevarlo a funciones de orden superior.* Fijarse de si es posible la recursividad de cola.

Currificacion - Currying

- APL no lo tiene.
- Curry, convierte una función que recibe varios argumentos en una que recibe menos argumentos, la cual puede invocarse con menos, y si fuera necesario devuelve otra función que espera los argumentos faltantes. *Función curry*
- Es posible realizarlo directamente en la definición.

Quinta y sexta semana

5. FP John Backus

Es un lenguaje funcional propuesto por Backus.
Consiste en 5 partes

- Conjunto O de Objetos
- Una operación, la Aplicación
- Un conjunto F de funciones f que convierten objetos en otros objetos
- Un conjunto FF de formas funcionales que combina funciones u objetos en nuevas funciones en F.
- Un conjunto D de definiciones de funciones de F.

Las funciones se pueden definir en cualquier orden, mientras que estén cargadas en el ambiente funcionan.

Todas las funciones son de aridad 1, pero este argumento puede ser un conjunto de datos.

Backus quería paralelismo, que no este establecido como es el orden del resultado final, que pueda realizar cosas a la vez. (Liberarnos del estilo de Von Neumann) (El id se utiliza mucho para aprovechar esto.)

Objetos

Un objeto puede ser:

- un átomo, una cadena no nula de caracteres (letras, dígitos), excluyendo los símbolos de la notación FP.
- una secuencia $< x_1, \ldots, x_n >$ donde cada elemento x_i es un objeto.
- un indefinido ($\perp$), representan una situación no computable. *Ej. 1 dividido 0 El interprete usado en la materia no lo tiene.*

Un átomo $\emptyset$ representa la secuencia vacía, y es el único objeto que es a la vez átomo y secuencia.
Los átomos T y F se usan para representar verdadero y falso.
Si una secuencia contiene el indefinido esta indefinida. $< 1, 6, A, \perp, 8 >= \perp$
Van en cursiva.

Aplicación

- Si f es una función y x es un objeto, entonces f : x es una aplicación y representa el objeto que resulta cuando se le aplica f a x. f es el operador y x el operando.
- Cualquier función aplicada a indefinido da indefinido.

Funciones

- Preservan el indefinido al recibirlo.
- Todas las funciones tienen aridad 1.

Algunas funciones son:

- Para acceder a posiciones usar el numero. *Ej. 2 : <A,B,C>* da B. Si queremos desde la derecha agregamos una r a la derecha del numero.

- tl: tail left, devuelve la secuencia sin el primero
- tlr da la cola desde la derecha
- id es la función identidad (devuelve lo recibido)
- atom pregunta si un objeto es un átomo.
- eq para igualdad.
- null para saber si es vació.
- Se pueden realizar las operaciones aritméticas y lógicas (estas ultimas escritas *or, and, not*)
- length da la longitud.
- reverse invierte una secuencia
- trans transpone la secuencia de secuencias
- distl o distr distribuye desde uno de los lados al resto
- apndl o apendr para concatenar
- rotr o rotl para rotar una secuencia

Formas funcionales

Con estas podemos construir funciones y operaciones mas complejas.

- Podemos componer funciones con o
- Construir varias funciones que se van a aplicar a losa objetos entre $[f1, f2, fn]$.
- Podemos establecer condiciones: *Ej. (not o atom → 1; id) : <A, B, C>resulta en A*
- Si queremos usar constantes les agregamos una barra arriba.
- Si queremos insertar una función entre objetos de una secuencia usamos la barra (/) a la izquierda de la función. (actúa como en APL)
- Para aplicar una función a todos los elementos de una secuencia se utiliza el α (en el interprete de la materia es @)

Definición de funciones

Es el conjunto de funciones que vamos definiendo. Algunos ejemplos son iota, factorial, producto interno, etc.

Séptima semana (en adelante)

6. Clojure

- Lenguaje del 2007.
- Derivado del Lisp.
- Es homoiconico, los programas se escriben usando las propias estructuras de datos del lenguaje. (podemos escribir código que interprete otro programa, por eso es bueno para interpretes, Clojure esta escrito en Clojure)
- Es un compilador en tiempo real, da la impresión de ser un interprete, pero en realidad compila y después ejecuta.
- Se traduce a byte code de java, es quien ejecuta realmente. Es parte de ese universo.
- Es muy integrable y expresivo a proyectos grandes. Con poco código puedo obtener muchos resultados.
- Leiningen, es un gestor de paquetes para Clojure.
- En Clojure las variables son temporales.
- Le da preferencia a la recursividad y funciones de orden superior (funciones que reciben y/o devuelven otras funciones).
- Permite representar conjuntos potencialmente infinitos.
- Ofrece una consola de evaluación (REPL: read eval print loop)
- Los predicados en Clojure tienen nombres que terminan en signo de pregunta (?) *Ej. even?* Se pueden ver con

  ```
  (apropos "?")
  ```

- Importamos archivos REPL con

  ```
  (load-file "nombre")
  ```

  después ejecutamos el programa llamando a la primer función (main). Si queremos usar un proyecto es diferente. (Es un conjunto de funciones metidas en un archivo)
- La primera posición la toma como una función. (deriva del concepto de aplicación de calculo lambda)

Estudiamos 6 partes en el curso:

Datos

Un dato puede ser un escalar, una colección cuyos elementos son datos, o una secuencia (abstracción que representa una vista secuencial de una colección)

Escalares

- Los símbolos se refieren a un único elemento, representan el nombre de algo (es Case Sensitive). No puedo tener nombres que empiezan con números. Si queremos evitar que evalué un símbolo incluimos un apostrofe a la izquierda del símbolo. Ejemplo:

```
'a
```

- Los valores literales pueden ser números (enteros como long o BigInt si tiene N como sufijo, punto flotante como double o BigDecimmal si tiene el sufijo M, racionales para fracciones). Los caracteres se representan con la barra invertida al lado del caracter (\a). Hay cadenas de caracteres con comillas dobles ("hola mundo")
- El nulo es el valor Ni, significa nada y representa una referencia nula.
- Los valores de verdad true y false.
- Tenemos constantes simbólicas como ##Inf, ##-Inf y ##NaN (depende la version)
- Hay palabras clave para indexar valores de los mapas, deben empezar con dos puntos (:primero), no se les permite iniciar con un . a la palabra clave.

Colecciones

- **Listas** se usan para representar código, van entre paréntesis. () es la lista vacía. Se usan para acceder secuencialmente a los datos, mantener el orden a medida que se insertan, mantener elementos repetidos, y para agregar y quitar datos rápidamente del frente (LIFO).
- **Vectores** se usan para representar datos, van entre corchetes. [] es el vector vació. Se pueden usar comas para separar los datos.
- Son útiles para acceder por índice aleatoriamente, mantener datos repetidos, mantener orden, y agregar y quitar rápidamente datos del final (LIFO)
- **Colas** no tienen codificación literal. Se deben construir a partir de la cola vacía. Sirve para acceder secuencialmente, mantener repetidos, mantener orden a medida que se insertan, y agregar datos al final y quitar del frente rápidamente (FIFO)
- **Conjuntos**, tenemos el hash-set y el sorted-set. Van entre llaves empezando con el numeral. (#) Sirven para mantener datos sin repeticiones, agregar o eliminar un dato dado, verificar si existe un dato en el conjunto.
- **Mapas** van con llaves sin el numeral. :x 10,:y 15. Acceso aleatorio a los datos por clave, agregar asociaciones de claves únicas y valores, eliminar pares, y verificar si existe una clave en el mapa. Existen hash-map y sorted-map

Secuencias

- Mediante la interfaz ISeq tenemos funciones de propósito general para trabajar con secuencias. Una seq es una abstracción representando una vista secuencial de una colección, osea una lista lógica que proporciona un acceso estable a una secuencia de valores.
- Las listas concretas implementan ISeq, por lo que son secuencias.

Evaluación de expresiones

Todo es una expresión que al ser evaluadas devuelven un valor.

Formas especiales

Las formas especiales tienen reglas de evaluación que difieren de las reglas estándar. Su listado completo se obtiene evaluando:

```
(pprint (keys (. clojure.lang.Compiler specials)))
```

.

- Un ejemplo es el if, no toma todos los argumentos, toma solo a medida que se va evaluando. (lazy evaluation)
- Algunas funciones que tienen efectos colaterales (ej print) además de imprimir retornan nil. (Todo tiene que devolver algo)
- let es para variables. Se liga el valor al identificador declarado. Esta permitido usarlo, es una especie de 'syntactic sugar'.

  ```
  (let [a [1 2 3], b 4]
  ```

  liga a al vector [1 2 3] y b a 4 . No deja efectos colaterales, si veo cuanto vale a, me va a dar indefinido (afuera de la función). Tratar de programar sin let provoca código largo y repetido.
- Tenemos try-catch-finally
- Las funciones que empiezan con un punto son un atajo a java. Es una forma especial. Si el primer operando es un símbolo que se resuelve como un nombre de clase, se considera el acceso a un miembro estático de la clase nombrada. Si el segundo operando es un símbolo y no se proporciona ningún argumento, se considera que es el acceso a un atributo.

Funciones predefinidas

Se tienen varias funciones predefinidas, se mencionan algunas.

- symbol devuelve un símbolo con el nombre indicado
- Si quiero agregar un paréntesis o algún carácter especial de Clojure usaría symbol para que me de ese símbolo en especial. ej symbol ")"
- number? es mas fácil ir preguntando con esta función que directamente con el tipo (ej integer?)
- mod y rem son distintos. La diferencia se nota en los negativos.
- str para convertir algo a una cadena. Si se le pasa mas de una cosa las concatena.
- replace, si lo ponemos solo busca el que esta en core, pero le podemos especificar que queremos que use el de clojure.string
- prn imprime tal cual la cadena, print analiza la cadena (ej los \n)
- Los números son todos true (truthfy) (incluido el 0), los falsos son nil y el propio false.
- some? para preguntar si algo es distinto de nil
- keyword construyo una palabra clave.
- hash-map (no se cuantos argumentos va a tener, solo que la cantidad de argumentos va a ser par, sino se queja) y zip-map, la diferencia esta en que les paso para construir el mapa

- contains? tiene diferentes implementaciones dependiendo de si se le pasa un vector, hash-set, o mapa
- pop para quitar un dato de la colección, tener cuidado que la colección no se modifica.
- get devuelve el elemento, se justifica respecto de (V pos) porque le podemos pasar un valor alternativo (get V 1 "me pase")
- into es mejor con vectores, no los desordena
- No olvidarse de que son inmutables.
- frequencies esta buena para funciones estadísticas, te devuelve cuantas veces aparece algo en un mapa
- (doc funcion) nos da la documentación de una función.
- Con (seq coleccion) nos crea una secuencia a partir de distintas entradas.
- Con (repeat argumento) devuelve una secuencia infinita repitiendo el argumento. Muy potente esta función.
- Con (range) devuelve una secuencia monotona creciente de numeros a partir del 0. Se le puede indicar un rango tambien. Si queremos mas potencia todavia le podemos dar el step.
- (rest coleccion) (next coleccion) para ir moviendose en una coleccion. (Devuelven la coleccion sin el primer elemento) (Difieren en lo que devuelven al final.)
- Si queremos pedir el primero del primero, podemos usar (ffirst '(colecciones)) en lugar de (first (first '(colecciones))) Mirar Operaciones dobles para más detalles.
- (drop) para remover multiples datos.
- (re-seq) devuelve en una secuencia las subcadenas de una cadena que coinciden con una expresion regular. *Muy importante para el TP.*
- (count) para el tamaño de una secuencia.
- (concar) para concatenar secuencias.
- (flatten) para planchar una secuencia.

Funciones de orden superior

Son funciones que reciben a otras funciones. Usar estas funciones es más idiomática.
Poniendo el numeral adelante es una funcion anonima.

- (apply funcion): invoca una funcion de aridad n con los n datos contenidos en el segundo. Podemos crear funciones de orden superior usando estas primitivas.
- (map funcion): Devuelve la secuencia formada por el resultado de aplicar una funcion.
- (mapv funcion): Igual que map solo que devuelve un vector.
- (partial funcion): Devuelve una funcion de menor aridad que la indicada por el primer argumento, fijamos argumentos.
- (reduce funcion): Aplica una funcion de aridad 2 al primer y segundo dato, luego el resultado se devuelve y es usado por el tercer dato, y así sucesivamente. No necesitamos agregar un 'tachito' como en FP para ir guardando los resultados.

- (filter funcion): permite seleccionar multiples datos. Devuelve una secuencia.
- (take-while funcion): Toma mientras se cumpla la condición
- (remove funcion): Devuelve una secuencia eliminando los datos del segundo argumento que cumplan con el predicado. (Lo opuesto de filter, se queda con los que no cumplen)
- (drop-while funcion): Opuesto a take-while
- (every? funcion): Devuelve si se cumple la condicion para todos los argumentos.
- (not-every? funcion): Devuelve si no todos cumplen la condicion.
- (not-any? funcion): Si hay por lo menos uno o no.
- (comp funcion): Devuelve una función que es la composición de los argumentos recibidos.
- (iterate funcion): Devuelve una secuencia infinita formada por el segundo argumento, seguido del primer argumento aplicado sobre el segundo. No se usa mucho. Hay que usar take para que termine.
- (sort funcion): Se le pasa el criterio de ordenamiento.
- (sort-by funcion): Ordena una secuencia con los datos del ultimo argumento, ordenados según los resultados de aplicar una función indicada. *Ej. una secuencia con mapas.*
- (partition-by funcion) Devuelve una secuencia con los valores del segundo argumento particionados según el primero. Va de forma iterativa, crea particiones. No los desordena.
- (group-by funcion): Agrupa segun el argumento. Crea los grupos.

Macros predefinidas

No los puedo usar con map. Si los quiero usar con map debo envolverlos en una funcion.

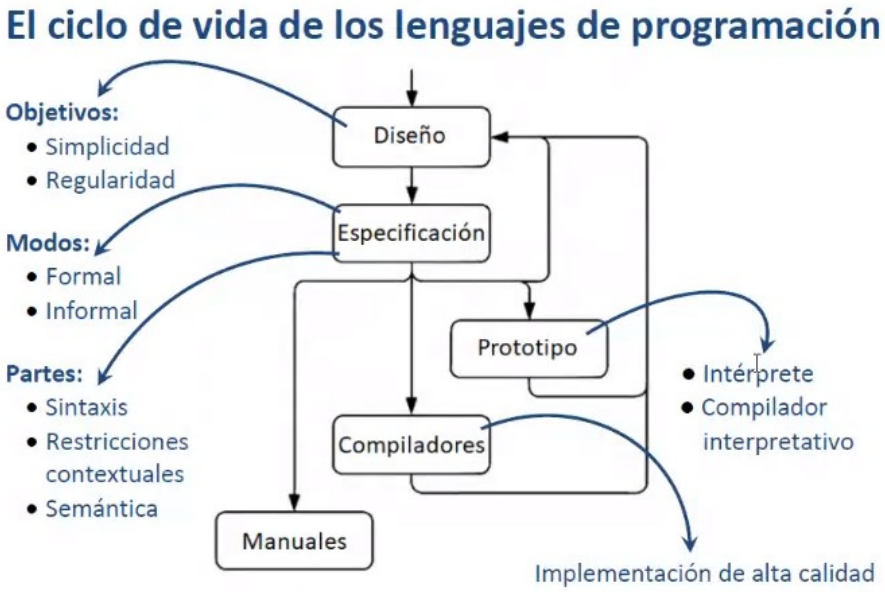
- (cond condiciones y expresiones): Es mejor que el if, más cómoda. La ultima condición tiene que ser algo que sea truthfy (true, :else, :default) puede ser cualquier cosa. Se detiene en la primera que cumple la condición.
- (case n): Es como un switch. No tiene condiciones, segun cada caso devuelve lo que corresponda
- (and condiciones)
- (or condiciones)
- (for): Es una sintaxis para expresar una lista por comprension.
- Hay tres tipos de flecha. $->$, $->>$, $->>>$, cada una aplica una serie de transformaciones sobre un conjunto . La ventaja es que lo puedo leer de forma mas clara.
- (with-out-str) Es bueno para los tests. Evalúa expresiones y devuelve una cadena con la concatenación de sus salidas.
- (declare) es igual a def, la diferencia esta que permite indicar varias funciones a la vez. Se usa para forward declarations.

Programación funcional - Clojure

- Bastante funcional debido a que cumple con las 7 características
- Programas como funciones. El programa consiste en la lista de definiciones de funciones, y la ejecución es aplicarlas.
- Funciones puras. Determinismo, ante los mismos argumentos devuelve el mismo valor (random no es pura), y ausencia de efectos colaterales.
- Datos inmutables, no modifica los valores, cambia las copias.
- Funciones de primera clase, se pueden pasar como argumento.
- Funciones de orden superior, toman funciones como argumento o las retornan.
- Composición de funciones, clojure permite componer funciones de diversas maneras. comp, partial, flechas
- Recursividad, clojure permite con pila y sin pila (recur para que la use si esta disponible).

7. Interpretes

- Podemos hacer un interprete del lenguaje que queramos.
- Siempre hay necesidad de nuevos lenguajes. Esto se conoce como abstracción metalingüística.
- El interprete es otro programa, estamos usando un lenguaje para crear un programa que interpreta otros programas.
- Scheme es un lenguaje funcional, esta pensado para evaluar expresiones.
- Segun el significado del operador en la primera posicion le hago hacer algo.



- En Scheme el valor booleano true es #t, y el false #f.

- Procesadores de lenguaje, cualquier sistema que manipule programas expresados en algún lenguaje. Sirven para ejecutar programas o prepararlos para ser ejecutados.
- Hay problemas prácticos de la interpretación, el rendimiento y el requerimiento de memoria.
- Compilación interpretativa, un compilador genera código de maquina y un interprete emula su ejecución. (Ej. Java)

8. Practica

8.1. Calculo Lambda

Ejercicio 1

Escribir las siguientes expresiones con el menor número de paréntesis posible:

a) (λx.(λy.(λz.((x z) (y z)))))
λx. (λy.(λz.((x z) (y z)))) - Los externos se omiten (1)
λx.λy. (λz.((x z) (y z))) - Cuerpo de las abstracciones hasta el final o paréntesis (3)
λx.λy.λz. ((x z) (y z)) - Cuerpo de las abstracciones hasta el final o paréntesis (3)
λx.λy.λz. (x z) (y z) - Cuerpo de las abstracciones hasta el final o paréntesis (3)
λx.λy.λz. x z (y z) - Aplicaciones se asocian a la izquierda (2)?
λx. y. z. (x z) (y z) - Opcional, se contraen las abstracciones lambda (4)

b) (((a b) (c d)) ((e f) (g h)))
((a b) (c d)) ((e f) (g h)) - Los externos se omiten (1)
(a b) (c d) ((e f) (g h)) - Aplicaciones a la izquierda (2)
a b (c d) ((e f) (g h)) - Aplicaciones a la izquierda (2)
a b (c d) (e f (g h)) - Aplicaciones a la izquierda (2)

c) (λx. ((λy. (y x)) (λv. v) z) u) (λw.w)
(λx. ((λy. y x) (λv. v) z) u) (λw.w) - Aplicaciones a la izquierda (2)
(λx. (λy. y x) (λv. v) z u) (λw.w) - Aplicaciones a la izquierda (2)

Ejercicio 2

Restaurar todos los paréntesis descartados en las siguientes expresiones:

a) x x x x
(x x x x) - Los externos se omiten (1)
((x x x) x) - Aplicaciones se asocian a la izquierda (2)
(((x x) x) x) - Aplicaciones se asocian a la izquierda (2)

b) λx. x λy. y
(λx. x λy. y) - Los externos se omiten (1)
(λx. (x λy. y)) - Cuerpo de las abstracciones hasta el final o paréntesis (3)
(λx. (x (λy. y))) - Aplicaciones se asocian a la izquierda (2)

c) λx.(x λy.y x x) x
(λx.(x λy.y x x) x) - Los externos se omiten (1)
((λx.(x λy.y x x)) x) - Aplicaciones se asocian a la izquierda (2)
((λx.((x λy.y x) x)) x) - Aplicaciones se asocian a la izquierda (2)
((λx.((x λy.y x) x)) x) - Aplicaciones se asocian a la izquierda (2)
((λx.(((x λy.y) x) x)) x) - Aplicaciones se asocian a la izquierda (2)
((λx.(((x (λy.y)) x) x)) x) - Aplicaciones se asocian a la izquierda (2)

Ejercicio 3

Para las siguientes expresiones lambda:

a) Identificar las ocurrencias de variables libres y ligadas.

1. (λx.((λy.y) x)) z - x ligada, y ligada, z libre

   (λx.((λy.y) x)) z - Beta con z (orden normal)

   (λy.y) z) - Beta con z

   z (λx.((λy.y) x)) z

   (λx.x) z - Beta con x (orden aplicativo)

   z - Beta con z

2. (λx.λy.x y) (z y) - x ligada, y ligada (operador), y libre (operando) , z libre

   (λx.λy.x y) (z y) - Beta con x y alfa con la y (si no estaría ligada) (igual con orden normal y aplicativo)

   λu. (z y) u

3. (λx.λy.x) x y - x ligada en la funcion, x e y libres

4. (λx.((λz.z x) (λx.x))) y - x ligada en ambos casos, y libre, z ligada

5. (λx.((λy.x y) z)) (λx.x y)

   x ligada en ambos casos, y ligada (operador de la abstracción interna de la función), y libre (operando), z libre

6. ((λy.(λx.((λx.λy.x) x)) y) M) N

   (λy.(λx.((λx.λy.x) x)) y) M N

   - x interna ligada, x externa ligada, y ligada

7. (λx.λy.λx.x y z) (λx.λy.y) M N -

   x interno ligado en la funcion y ligado en la funcion y en el parametro x libre en el parametro x externo libre en la funcion z libre

8. ((λx.(λy.λz.z) x) ((λx.x x x) (λx.x x x))) x

   (λx.(λy.λz.z) x) ((λx.x x x) (λx.x x x)) x

   aplicación en el parámetro mas externo: x libre en la función: tengo otra aplicación aplicación izquierda

   x ligada, y libre, z ligada aplicación derecha, sus dos abstracciones con x están ligadas (cada x a su lambda)

Notas de las practicas

- Jamas puedo hacer un cambio de variable en una variable libre.
- Si no tengo variables libres en el parámetro no es necesario un cambio de variable
- No hacer cambios de variables de mas, es para ver si entendimos el concepto.
- Orden normal, si el parámetro se puede llegar a reducir, lo meto sin reducirlo.
- Orden aplicativo (lazy evaluation), el parámetro lo puedo reducir? Si, lo reduzco - Puedo reducir el cuerpo de mi función? Si, reduzco cuerpo y parámetro y vuelvo a escribir.

- No ver los ordenes como *la mas interna etc*, usar lo mencionado arriba para evitar equivocaciones.
- Siempre empezar identificando variables libres y ligadas, es para ver donde hago el remplazo.
- Si ambas formas llegan a un resultado, tiene que ser el mismo.
- Evaluo de izquierda a derecha.
- Si hay mas de un paréntesis a la izquierda, el resto esta de mas. Se borran.
- Solo se puede hacer cambio de variable a las variables ligadas al lambda. Nunca cambiar lo que esta entrando.

8.2. APL

Notas tomadas de las practicas.

- Sustantivos todos los parámetros que puede tener una función.
- Todos los verbos llevan uno o dos parámetros como mucho.
- Adverbio es un operador de orden superior, actúa sobre un verbo. Lo modifica. Ejemplo la barra.
- Conjunción es un operador de orden superior que actúa sobre dos verbos generando un verbo derivado. Ejemplo el producto interno.
- APL se lee de derecha a izquierda.
- Los parámetros son inmutables, nunca se modifican los resultados de los parámetros.
- El barra hace la comprensión de todas las columnas, inserta la función entre medio de todas las columnas (en realidad estoy comprimiendo las filas)
- Con un vector es lo mismo poner la barra y la barra cruzada, tenemos un solo elemento.

8.3. FP Backus

Notas generales:

- No hay variables!
- Hay un solo objeto o ambiente, puede ser una lista, o un átomo, o un indefinido.
- Al indefinido, aplicada cualquier función da indefinido.
- La idea es ir aplicando funciones que transformen el ambiente en el buscado (otro ambiente).
- Para eso tenemos funciones primitivas, formas funcionales (ej. la barra /), o aquellas definidas por el usuario.
- El selector se aplica solo a listas (secuencias)
- id devuelve el mismo ambiente.
- Un predicado siempre devuelve un valor de verdad. (con indefinido devuelve indefinido)

Sobre algunas funciones:

- eq necesita una lista con 2 atomos.
- null se aplica a una secuencia, nos dice si esta vacia.
- not toma un atomo que es verdadero o falso.
- distl necesito una lista con 2 elementos, el segundo tiene que ser una secuencia. Junta el primer elemento con cada uno de la secuencia. (distr es a la inversa)
- apndl necesita una lista también.
- rotl rota la lista, mueve en un lugar cada cosa (lista circular)
- Binario y ciclo no los toma
- La composición es la forma de unir funciones.

  $1\ o\ tl :< 1\ 2\ 3\ 4 >$
  $(< 2\ 3\ 4 >)$
  2

- La construcción junta funciones que quiero aplicar en un mismo ambiente, nos da una secuencia con tantos elementos como grupos de funciones tenga.

  $[1, 5, 4] :< 1\ 2\ 3\ 4\ 5 >$
  $<< 1 >< 5 >< 4 >>$

- La condición, a la izquierda si o si necesito un predicado
- La barra / funciona como una comprensión como en APL, la va metiendo entre cada uno de los elementos de la lista. Simula una especie de recursividad. De derecha a izquierda.

  $/+ :< 1\ 2\ 3\ 4\ 5 >$
  $+ :< 4\ 5 >$
  9
  $+ :< 3\ 9 >$
  12
  $+ :< 2\ 12 >$
  14
  $+ :< 1\ 14 >$
  15

- El alfa es la aplicación de una función a cada elemento.

  @+ :<< 5 6 >< 7 8 >< 9 10 >>
  < 11 15 19 >

Algunos ejercicios

1 d) El elemento mínimo entre los máximos por fila de una matriz (minimax).

```
Def minmax = minsec o @maxsec
Def maxsec = /max
Def max = > -> 1; 2
Def minsec = /min
Def min = > -> 2; 1
```

2 a) La pertenencia de un elemento a una secuencia.

/or o @eq o distl :< a < b a c >>
Otra solución

```
def pertenece2 = null o 2# -> ~F; /or o alpha eq o distl
```

Devuelve falso si es la secuencia vacía

3 c) La diferencia de ambas subsecuencias.

```
< <1 2 4 5> <3 4 6 > >
< 1 2 5>


Def pertenece = null o 2 -> ~F; /or o @eq o distl
Def elemento = pertenece -> ~<>;1
Def eliminarVacios = null o 1 -> 2; apndl
Def ev = /eliminarVacios o apndr o [id, ~<>]
Def diferencia = ev o @elemento o distr

<1 2 5 <> 6 <> 7>
< <1 2 5 <> 6 <> 7> <> >
<1 2 5 <> 6 <> 7 <> >
<7 <>>
<7>
< <> <7> >
<7>
<6 <7> >
<6 7>
...
```

Hay un solo ambiente que se va modificando.

3 a) La unión de ambas subsecuencias.

```
< <1 2 3 4> <4 5 6> >
<1 2 3 4 5 6>

Def unir = pertenece -> 2;apndr
/unir o apndr o [id, ~<>]
```

```
Def union= /auxunion o apndr
Def auxunion = pertenece -> 2; apnl
< 1 2 3 4 <4 5 6> >
<4 <4 5 6> >
<4 5 6>
<3 <4 5 6>>
<3 4 5 6>
```

Dos secuencias sin repetición. Si hay repetidos podemos usar un eliminar repetido al final que es similar al eliminar vació (Def union= er o/auxunion o apndr)

3 b) La intersección de ambas subsecuencias.

```
< <1 2 3> <2 3 4> >
< 2 3>

Def pertenece = null o 2 -> ~F; /or o @eq o distl
Def elementoInterseccion = pertenece -> 1;~<>
Def esVacio = null o 1 -> 2; apndl
Def eliminarVacios = /esVacio o apndr o [id, ~<>]
Def interseccion = eliminarVacios o @elementoInterseccion o distr
```

14 a)Los equipos invictos.

- Usar la diferencia, los del 3
- Transponer
- «IN RO><IN BO»
- «....Ganadores><perdedore»
- Devolver los que ganaron y nunca perdieron
- Si transpongo tengo una lista con todos los ganadores y todos los perdedores. Le saco una diferencia y tengo los que ganaron y nunca perdieron.

```
Def pertenece = null o 2 -> ~F; /or o @eq o distl
Def elemento = pertenece -> ~<>;1
Def esVacio = null o 1 -> 2; apndl
Def eliminarVacios = /esVacio o apndr o [id, ~<>]
Def diferencia = eliminarVacios o @elemento o distr
Def esIgual = pertenece -> 2; apndl
Def eliminarRepetidos = /esIgual o apndr o [id, ~<>]
Def invictos = eliminarRepetidos o diferencia o trans
```

14 b) Los que siempre perdieron

```
Def pertenece = null o 2 -> ~F; /or o @eq o distl
Def elemento = pertenece -> ~<>;1
Def esVacio = null o 1 -> 2; apndl
Def eliminarVacios = /esVacio o apndr o [id, ~<>]
Def diferencia = eliminarVacios o @elemento o distr
Def invictos = eliminarRepetidos o diferencia o reverse o trans

otra opcion
Def perdedores = er o diferencia o [2, 1] o trans
```

```
Def agregar = apndr o [id, ~<>]
Def elimrepetidos = /(pertenece -> 2; apndl) o agregar
Def fdif = (@ (pertenece -> ~<>; 1)) o distr
Def diff = elimrepetidos o eliminarVacios o fdif
```

8) Definir el producto de un escalar por una matriz

Queremos:

```
< 4 < <1 2 3> <4 5 6> <7 8 9> > >
<< 4 8 12> <16 20 24> < 28 32 36>>

< 4 < <1 2 3> <4 5 6> <7 8 9> > >
<  < 4 <1 2 3>> <4 <4 5 6>> <4 <7 8 9>> >
<  < <4 1> <4 2> <4 3>> < <4 4> <4 5> <4 6>> < <4 7> <4 8> <4 9>> >
<< 4 8 12> <16 20 24> < 28 32 36>>

Def multilist = (@ *) o distl
Def productoEscalar = @multilist o distl

productoEscalar : <2, <<1 2 3> <4 5 6> <7 8 9>>>
```

Ej Dada una secuencia donde el primero es un numero y el segundo la lista, usar el numero como selector. (Si se pasa devolver vació)

```
<3 < a b d f>>
d

Def iota = auxiota o [~1, id, ~<>]
Def auxiota = (> o [1, 2] -> 3; auxiota o [+ o [1, ~1], 2, apndr o [3, 1]])

Def selector = 2 o 1r o trans o [iota o 1, 2]

[iota o 1, 2] da <<1 2 3> <a b c d>>
```

Al transponer quedan los pares, agarramos el ultimo desde la derecha (una lista, por ejemplo $< 3\ d >$) y de ahí agarramos el elemento.

Nota: al transponer si no son del mismo tamaño es indefinido. (depende de la implementación, en el interprete funciona)

Otra forma es con [1 trans o [iota o len, 2]] agregando un pertenece.

```
<3 << 1 a > <2 b > <3 d> <4 c> >
< 3 <1 a>> <3 <2 b>> <3 <3 d> ....>
distl
@pertenece
< <> <> d <> >
/eliminarVacios
```

Ej Dados dos elementos devolver la cantidad de veces que se encuentra el primer átomo en la segunda lista.

```
< a  < a  g h a j a> >
3
```

```
Def transformar = eq -> ~1; ~0
Def contar = /+ o @transformar o distl

distl
< <a  a> <a g> <a h> <a a> <a j> <a a> >
@transformar
< 1 0 0 1 0 1 >
```

14 c) Los que ganaron más veces de las que perdieron

```
ganaronMas : < <IN BO> <IN RI> <BO RI> <BO SA> >
<IN BO>
```

Queremos una lista con $<$ $<$ *Equipo Ganados Perdidos* $>$ ... $>$ para poder resolver lo pedido.

```
Def union= /auxunion o apndr
Def auxunion = pertenece -> 2; apndl

Def ganadores = @ 1
Def perdedores = @ 2
Def listaEquipos = eliminarRepetidos o union o [ganadores, perdedores]

Def contarResultados = @[1, contar o [1,1 o 2], contar o [1,2 o 2]] o distl o [listaEquipos, tran

Def comparar = eq [2, 3] -> 1 ; ~<>

Def filtrar = > o [2, 3] ->  1;~<>

Def ganaronMas = eliminarVacios o @filtrar o contarResultados
```

8.4. Clojure

Notas

- (list) creamos la lista, nos devuelve ()
- Todas las funciones de creación son de aridad n, es decir, puedo tener n parámetros al llamar a esa funcion. (list 'a 'b 'c)
- (pop)
- (peek)
- (nth)
- (defn) para definir funciones
- La suma y producto tienen aridad n.
- Podemos definir funciones con distinto tipo de aridad.
- map, con un parametro aplica la funcion a cada uno de los elementos de ese unico parametro (alfa de FP).
- apply desempaqueta el parametro. Llama a la funcion con todos los elementos de la secuencia como parametro.

1) Definir la función tercer-angulo que reciba los valores de dos de los ángulos interiores de un triángulo y devuelva el valor del restante.

Mostrar error si alguno es negativo o alguno es mayor a 180

```
(defn tercer-angulo [alfa, beta]
 (cond
    (< 180 alfa) ( println "Error" )
    (< 180 beta) ( println "Error" )
    (neg? alfa) ( println "Error" )
    (neg? beta) ( println "Error" )
    :else ( - 180  (+ alfa beta))
 )
)
```

2) Definir la función segundos que reciba los cuatro valores (días, horas, minutos y segundos) del tiempo que dura un evento y devuelva el valor de ese tiempo expresado solamente en segundos.

```
(defn a-segundos [dias, horas, minutos, segundos]

      (cond
        (neg? dias) ( println "Error" )
        (neg? horas) ( println "Error" )
        (neg? minutos) ( println "Error" )
        (neg? segundos) ( println "Error" )
        :else (+ (* dias 86400) (* horas 3600) (* minutos 60) segundos)
```

```
        )
      )
```

3) Definir la función sig-mul-10 que reciba un número entero y devuelva el primer múltiplo de 10 que lo supere.

```
(defn sig-mul-10 [n]
      (cond
        (= 0 (rem n 1)) ( println "No es entero" )
        (neg? n) (* 10 (quot n 10) )
        (= n 0) 10
        :else (* 10 (+ (quot n 10) 1))
      )
  )
```

5) Definir la función capicua? que reciba un número entero no negativo de hasta 5 dígitos y devuelva true si el número es capicúa; si no, false.

```
(defn capicua? [n]
    (let [unidades (rem n 10),
          decenas (rem (quot n 10) 10),
          centenas (rem (quot n 100) 10),
          unidades-mil (rem (quot n 1000) 10),
          decenas-mil (rem (quot n 10000) 10)]

         (cond
           (= 0 (rem n 1)) ( println "No es entero" )
           (neg? n) (println "Es negativo")
           (< n 10) true
           (< n 100) (= unidades decenas)
           (< n 1000) (= unidades centenas)
           (< n 10000) (and (= unidades unidades-mil) (= decenas centenas))
           :else (and (= unidades decenas-mil) (= decenas unidades-mil))
           )
     )
)
```

```
(defn capicua? [n]
  (cond
    (not (integer? n)) (println "El numero tiene que ser un entero")
    (neg? n) (println "El numero tiene que ser un negativo")
  :else
    (= (reverse (str n)) (seq (str n)))
  )
)

(println (capicua? 12334578))
```

8) Definir la función nth-fibo que reciba un número entero no negativo y devuelva el correspondiente término de la sucesión de Fibonacci.

```
(defn nth-fib [n]
      (cond
        (not (integer? n)) (println "No es entero")
        (neg? n) (println "Es negativo")
        (zero? n) 0
        :else (fib-rec (- n 1) 0 1)
        )
    )

(defn fib-rec [n, anterior, acumulado]
      (cond
        (zero? n) acumulado
        :else (fib-rec (- n 1) acumulado (+ anterior acumulado))
        )
      )
```

Con múltiple aridad:

```
(defn nth-fibo
    ([n]
        (cond
                (neg? n) nil
                (zero? n) 0
                (= n 1) 1
                :else(nth-fibo 0 1 2 n)
            )
        )
    ([f1 f2 cont n]
        (if (= cont n)
            (+ f1 f2)
            (nth-fibo f2 (+ f1 f2) (inc cont) n)
        )
    )
)
```

9) Definir la función cant-dig que reciba un número entero y devuelva la cantidad de dígitos que este tiene.

```
(defn cant-dig [n]
    (cond
      (not (integer? n)) (println "No es entero")
      :else (count (seq (str n)))
      )
)
```

De forma recursiva:

```
(defn contar-dig [n] (if (zero? n) 0 (inc (contar-dig (quot n 10)))))
```

11) Definir la función digs que reciba un número y devuelva una lista con sus dígitos.

El map ya nos da la lista.

```
(defn abs [n]
    (cond
        (< n 0) (* -1 n)
        :else n
    )
)

(defn digs [N]
    (println (map {\0 0 \1 1 \2 2 \3 3 \4 4 \5 5 \6 6 \7 7 \8 8 \9 9} (str (abs n))))
)
```

12) Definir la función repartir que, llamada sin argumentos, devuelva la cadena Üno para vos, uno para mí". De lo contrario, se devolverá una lista, en la que habrá una cadena Üno para X, uno para mí"por cada argumento X

```
(defn uno-para [s]
  (str "Uno para " s ", uno para mí")
)

(defn repartir
  ([] (uno-para "vos"))
  ([& more] (map uno-para more))
)
```

13) Definir una función para producir una lista con los elementos en las posiciones pares de dos listas dadas.

```
(defn pares [unaLista otraLista]
    (map second (concat (partition 2 unaLista) (partition 2 otraLista) ) )
)
```

(partition 2 '(1 2 3 4 5 6 7)) ((1 2) (3 4) (5 6))

17) Definir una función para obtener el elemento central de una lista.

```
(defn central [lista]
    (nth lista (/ (count lista) 2))
)
```

Modificado:

```
(defn obtenerElementoCentral[lista]
    (cond
```

```
        (odd? (count lista)) (nth lista (/ (count lista) 2))
        :else (
            list (nth lista (- (/ (count lista) 2) 1) ) (nth lista (/ (count lista) 2) )
            )
    )
)
```

23) Definir una función para transponer una lista de listas

Devuelve una función de menor aridad que la indicada como primer argumento, ya que los demás argumentos se utilizan ("se fijan") en la función recibida.

```
(defn transponer [listas]
    (apply map list listas)
)

(println (transponer '((1 2 3) (4 5 6))))
(println (transponer '((a b c d e) (1 2 3 4 5) (z x y u v))))
```

Aplica map list a cada una de las listas de la función, lo transforma como una función Apply los manda como parámetros independientes map list ejecuta list entre cada uno de los elementos que tenemos (primeros, segundos, etc)

21) Definir una función para obtener la matriz triangular superior (incluyendo la diagonal principal) de una matriz cuadrada que está representada como una lista de listas.

```
(defn poner-ceros [listas actual]
    (concat
        listas
        (list (into (nthnext actual (count listas) ) (repeat (count listas) 0)))
    )
)

(defn triangular-superior [listas]
    (cond
        (empty? listas) (println "No se tienen listas")
        :else (reduce poner-ceros '() listas)
    )
)
```

```
(defn poner-ceros ([lista cant-ceros]
        (poner-ceros lista cant-ceros 0)
    )
    ([lista cant-ceros pos-actual]
        (cond
            (= pos-actual cant-ceros) lista
            :else
                (recur
```

```
                    (apply list (assoc (vec lista) pos-actual 0))
                    cant-ceros
                    (inc pos-actual)
                )
            )
        )
    )

    (defn triangular-rec [listas actual acumulado]
        (cond
            (= actual (count listas)) acumulado
            (zero? actual) (triangular-rec listas 1 (list (nth listas actual) )  )
            :else (recur listas (inc actual)
                  (concat acumulado (list (poner-ceros (nth listas actual) actual))) )
        )
    )

    (defn triangular-superior [listas]
        (cond
            (empty? listas) (println "No se tienen listas")
            :else (triangular-rec listas 0 '() )
        )
    )
```

22) Definir una función para obtener la diagonal principal de una matriz cuadrada que está representada como una lista de listas.

```
    (defn poner-ceros [listas actual]
        (concat
            listas
            (list (into (nthnext actual (count listas) ) (repeat (count listas) 0)))
        )
    )

    (defn triangular-superior [listas]
        (cond
            (empty? listas) (println "No se tienen listas")
            :else (reduce poner-ceros '() listas)
        )
    )
```

```
    (defn diagonal-rec [listas actual acumulado]
        (cond
            (= actual (count listas))  (reverse acumulado)
            :else (recur listas
                         (inc actual)
```

```
                    (conj acumulado (nth (nth listas actual) actual) )
                )
    )
)

(defn diagonal [listas]
    (cond
        (empty? listas) (println "No se tienen listas")
        :else (diagonal-rec listas 0 '() )
    )
)
```

36) Definir la función sublist que devuelva la sublista correspondiente a una lista, una posición inicial y una longitud dadas.

Por ejemplo: (sublist '(A B C D E F G) 3 2) ->(C D)

```
(defn sublist [lista pos-inicial longitud]
    (take longitud (drop pos-inicial lista))
)
```

35) Definir las funciones filas-max-V y mas-V-o-F que, aplicadas a una matriz de V y F (una lista de listas con los valores V y F), devuelvan, respectivamente:

a) El/los número/s de la/s fila/s en la/s que la cantidad de V es máxima, por ejemplo: (filas-max-V '((V F V V F)(V V F V V)(F F F V F)(V V V F V))) –>(2 4)

```
(defn contar-V [index lista]
    (cond
        (> (get (frequencies lista) 'V) (get (frequencies lista) 'F)) index
        :else nil
    )
)

(defn filas-mas-V [listas]
    (remove nil? (map-indexed contar-V listas))
)
```

```
(defn contar-V [maximo lista]
    (cond
        (> (get (frequencies lista) 'V) (get (frequencies maximo) 'V)) lista
        :else maximo
    )
)

(defn filas-max-V [listas]
    (reduce contar-V listas )
)
```

```
(defn contar-Vs [fila] (reduce + (replace '{V 1, F 0} fila)))

(defn hallar-max [lista] (reduce max lista))

(defn cantidad-de-Vs [m] (map contar-Vs m))

(defn indices-de-max [lista maximo]
    (map * (map #(if (= maximo %) 1 0) lista) (range (count lista))))

(defn filas-max-V [m]
    (map inc
        (filter pos?
            (indices-de-max
                (cantidad-de-Vs m)
                (hallar-max (cantidad-de-Vs m))
            )
        )
    )
)
```

9. Bibliografía

Apuntes otorgados por la cátedra, los mismos se pueden encontrar aquí.