Fundamentos de programación

Datos complejos I: Recursión numérica, listas arbitrariamente largas y estructuras recursivas

Facultad de Ingeniería. Universidad del Valle

Octubre de 2018

Contenido

Contenido

Recursión numérica

$0 \in \mathbb{N}$

Base

(add1 0) → 1
(add1 1) → 2
(add1 2) → 3

Definición

Los números naturales se pueden definir de la siguiente forma:

1. 0 es un natural
2. Si *n* es un numero natural (add1 n) también lo es

(add1 $\underbrace{\text{(add1 } \underbrace{\text{(add1 0)}}_{1}\text{)}}_{2}$)

(cons 5 (cons 4 (cons ... (... (empty)

$5!$ $\quad 5 \times 4 \times 3 \times 2 \times 1$

$3!$ $\quad 3 \times 2 \times 1$

Caso

Base $\quad 1!$ $\quad 1$

$n! = n(n-1)!$

$1! = 1$

$2! = 2 \times 1!$

$3! = 3 \times 2!$

$4! = 4 \times 3!$

$5! = 5 \times 4!$

$100! = 100 \times 99!$

$99! = 99 \times 98!$

$\vdots$

$1! = 1$

$$(\text{Fact } n) = \begin{cases} 1 & n = 1 \\ (* \; n \; (\text{Fact } (- \; n \; 1))) \end{cases}$$

Recursión numérica

Definición

Los números naturales se pueden definir de la siguiente forma:

1. 0 es es un natural
2. Si $n = 0$ (add1 0) $= 1$ es también natural
3. Si $n = 1$ (add1 1) $= 2$ es también natural
4. Si $n = 2$ (add1 2) $= 3$ es también natural

Recursión numérica

Definición

Los números naturales se pueden definir de la siguiente forma:

1. 0 es el primer número
2. Si $n = 0$ (add1 0) $= 1$
3. Si $n = 0$ (add1 (add1 0)) $= 2$
4. Si $n = 0$ (add1 (add1 (add1 0))) $= 3$

Recursión numérica

Definición

Si se observa un número natural se puede definir haciendo un llamado varias veces de la función **add1** a esto lo vamos a conocer como **definición recursiva**

Recursión numérica

Definición

Esto también aplica para casos de funciones, por ejemplo el **factorial** que se define de la siguiente forma:

$$fact(n) = n * (n-1) * (n-2) * ... * 1, fact(0) = 1 \quad (1)$$

Si observa es una secuencia de multiplicaciones.

Recursión numérica

Definición

Si observamos la forma podemos definir una función para calcular el factorial ¿Como sería?

```
;;Contrato factorial: numero -> numero
(define (factorial n)
        ....
)
```

Definición

Empezamos a analizar, si $n = 1$ el factorial es 1

```
;;Contrato factorial: numero -> numero
(define (factorial n)
    (cond
        [(= n 1) 1]  <- Caso base
        ...
)
```

```
(define (factorial n)
   (cond
       [ (= n 1) 1]
       [else (* n (factorial (- n 1)))]
   )
)
```

(factorial 5))

(* 5 (factorial 4)))

(* 5 (* 4 (factorial 3)))

(* 5 (* 4 (* 3 (factorial 2))))

(* 5 (* 4 (* 3 (* 2 (factorial 1)))))

(* 5 (* 4 (* 3 (* 2 1))))

1!

(* 5 (* 4 (* 3 2)))

2!

(* 5 (* 4 6))

3!

(* 5 24)

4!

120

Resumen

1) Recursión.

a) Consiste en dos casos
- Caso base: Trivial
- Caso recursivo: Depende de casos anteriores

b) Diseño de funciones recursivas
- Primero: Debe ir la condición base
- Segundo: Llamado a la misma función: este nos debe de llevar poco a poco hacia CASO BASE

Recursión numérica

Definición

Empezamos a analizar, si $n = 1$ el factorial es 1*1, y si $n = 2$ entonces 1*1*2, **empezamos a ver un patrón**

```
;;Contrato factorial: numero -> numero
(define (factorial n)
    (cond
        [(= n 1) 1]
        [else
            (* n (factorial (- n 1)))
        ]
    )
)
```

Recursión numérica

Definición

Estas funciones que se llaman así mismas son llamadas **funciones recursivas** debe tener en cuenta:

1. Una condición de parada, para que no se llame infinitamente. Es el caso inicial.
2. La función debe siempre retornar el mismo tipo de dato
3. Un llamado a la misma función, utilizando alguna función para unir las salidas (una operación matemática)

Recursión numérica

Ejemplo

1. Diseñe una función **multiplicación**, la cual recibe dos números (a y b) esta retorna el resultado de sumar b veces a
2. Diseñe una función **elevar**, la cual recibe dos números (a y b), esta retorna el resultado de multiplicar b veces a

Recursión numérica

(+ a b) $\underbrace{a + a + a + a}_{b\ veces}$

Ejemplo

Diseñe una función **multiplicación**, la cual recibe dos números (a y b) esta retorna el resultado de sumar b veces a

```
;;Contrato multiplicacion: numero, numero -> numero
(define (multiplicacion a b)
    (cond
        [(= b 1) a] [(= b 0) 0]
        [else
            (+ a (multiplicacion a (- b 1) ))
        ]
    )   (mul 5 3)
)
```

(+ 5 (+ 5 5))

Recursión numérica

Ejemplo

Diseñe una función **elevar**, la cual recibe dos números (a y b), esta retorna el resultado de multiplicar b veces a

```
;;Contrato elevar: numero, numero -> numero
(define (elevar a b)
    (cond
        [(= b 1) a]
        [else
            (* a (elevar a (- b 1)))
        ]
    )
)
```

Fundamentos de programación

Contenido

Listas arbitrariamente largas

Definición

Hasta el momento hemos trabajado con listas de un tamaño dado, pero que sucede si trabajamos con listas de diferente tamaño. Por ejemplo una función que recibe listas de símbolos y se desea encontrar uno, podríamos diseñar una función asi.

```
;;Contrato buscar-simbolo: lista-de-simbolos , simbolo ->
    booleano
(define (buscar-simbolo lista nombre)
     (cond
        [(eqv? (first lista) nombre) #t]
        [else ... ]
     )
)
```

Aquí miramos si el primer elemento de la lista es lo que buscamos, sin embargo, ¿Como verificamos el segundo, y los otros elementos?

Listas arbitrariamente largas

Definición

Una idea sería analizar el resto de la lista (que es una lista que contiene los otros elementos)

```
;;Contrato buscar-simbolo: lista-de-simbolos, simbolo ->
    booleano
(define (buscar-simbolo lista nombre)
    (cond
        [(eqv? (first lista) nombre) #t]
        [else ... (rest lista) ...]
    )
)
```

¿Observan algo en el contrato? ¿El resto de la lista que es?

Listas arbitrariamente largas

Definición

Podríamos enviar el resto de la lista a la misma función (para que siga buscando) y mirar si el símbolo está:

```
;;Contrato buscar-simbolo: lista-de-simbolos, simbolo ->
    booleano
(define (buscar-simbolo lista nombre)
    (cond
        [(eqv? (first lista) nombre) #t]
        [else (buscar-simbolo (rest lista) nombre)]
    )
)
```

Pero, hay algo que está mal ¿Que pasa si el elemento no está en la lista?

Definición

El problema es que si seguimos buscando, ¿Que hacemos cuando llegamos al final de la lista (empty)?

```
;;Contrato buscar-simbolo: lista-de-simbolos , simbolo ->
    booleano
(define (buscar-simbolo lista nombre)
    (cond
        [(empty? lista) #f]
        [(eqv? (first lista) nombre) #t]
        [else (buscar-simbolo (rest lista) nombre)]
    )
)
```

Debemos verificar que si llega al final de la lista

(first listo)
(first (rest listo))

Listas arbitrariamente largas

Definición

Para el diseño de funciones que trabajan sobre listas arbitrariamente grandes debe tener en cuenta:

1. Analizar el primer elemento de la lista: Verificación.
2. Tener en cuenta que la lista termina cuando esta es **empty**. Condición de parada
3. Analizar el resto de la lista, llamando la misma función. **Condición de llamado recursivo**

Listas arbitrariamente largas

Ejemplo

1. Diseñe una función **buscar-numero** que recibe un número y una lista de números. Esta función indica que el número está en la lista de números
2. Diseñe una función **buscar-persona-nombre** que recibe una lista de estructuras persona que tiene tres atributos: nombre, edad y cargo; y recibe un nombre. Esta función indica si hay alguna persona con el nombre indicado

Listas arbitrariamente largas

Definición

Diseñe una función **buscar-numero** que recibe un número y una lista de números. Esta función indica que el número está en la lista de números

```
;;Contrato buscar-numero: lista-de-numeros, numero ->
    booleano
(define (buscar-numero numero listan)
    (cond
        [(empty? listan) #f]
        [(eqv? (first listan) numero) #t]
        [else (buscar-numero numero (rest listan))]
    )
)
```

Definición

Diseñe una función **buscar-persona-nombre** que recibe una lista de estructuras persona que tiene tres atributos: nombre, edad y cargo; y recibe un nombre. Esta función indica si hay alguna persona con el nombre indicado

```
;;Contrato buscar-numero: lista-de-personas, simbolo ->
    booleano
(define-struct persona (nombre edad cargo))
(define (buscar-persona-nombre nombre listaPersonas)
    (cond
        [(empty? listaPersonas) #f]
        [(eqv? (persona-nombre (first listaPersonas))
            nombre) #t]
        [else (buscar-persona-nombre nombre (rest
            listaPersonas))]
    )
)
```

Fundamentos de programación

Recursión numérica

Listas arbitrariamente largas

Estructuras recursivas (Árboles)

Definición

También podemos ir más allá, por ejemplo podemos realiza la suma de elementos en una lista de números observe:

```
;;Contrato sumar-lista: lista-de-numeros -> numero
(define (sumar-lista listan)
    (cond
       [(empty? listan) 0]
       [else (+ (first listan) (sumar-lista (rest listan)
           ))]
    )
)
```

¿Que puede decir el comportamiento de esta función? Analicemos el caso **(cons 1 (cons 2 (cons 3 empty)))**

Listas arbitrariamente largas

Definición

Cuando llamamos la función con la lista **(cons 1 (cons 2 (cons 3 empty)))**.

```
;;Contrato sumar-lista: lista-de-numeros -> numero
(define (sumar-lista listan)
    (cond
       [(empty? listan) 0]
       [else (+ (first listan) (sumar-lista (rest listan)
           ))]
    )
)
```

La lista no está vacía por ende, se ejecuta la clausula **else** y queda lo siguiente:

(+ 1 (sumar-lista (cons 2 (cons 3 empty))))

Listas arbitrariamente largas

Definición

En el siguiente llamado se tiene **(cons 2 (cons 3 empty))**.

```
;;Contrato sumar-lista: lista-de-numeros -> numero
(define (sumar-lista listan)
    (cond
       [(empty? listan) 0]
       [else (+ (first listan) (sumar-lista (rest listan)
           ))]
    )
)
```

La lista no está vacía por ende, se ejecuta la clausula **else** y queda lo siguiente:

(+ 1 (+ 2 sumar-lista (cons 3 empty))))

Listas arbitrariamente largas

Definición

En el siguiente llamado se tiene **(cons 3 empty)**.

```
;;Contrato sumar-lista: lista-de-numeros -> numero
(define (sumar-lista listan)
    (cond
        [(empty? listan) 0]
        [else (+ (first listan) (sumar-lista (rest listan)
            ))]
    )
)
```

La lista no está vacía por ende, se ejecuta la clausula **else** y queda lo siguiente:

(+ 1 (+ 2 (+ 3 (sumar-lista empty))))

Listas arbitrariamente largas

Definición

En el siguiente llamado se tiene **empty**.

```
;;Contrato sumar-lista: lista-de-numeros -> numero
(define (sumar-lista listan)
    (cond
       [(empty? listan) 0]
       [else (+ (first listan) (sumar-lista (rest listan))
            )]
    )
)
```

La lista está vacía por ende, retorna 0 y se tiene **(+ 1 (+ 2 (+ 3 0)))** y se obtiene 6.

Definición

También podemos generar listas, observe:

```
;;Contrato doble-lista: lista-de-numeros ->
    lista-de-numeros
(define (doble-lista listan)
    (cond
        [(empty? listan) empty]
        [else (cons (* (first listan) 2) (doble-lista (
            rest listan)))]
    )
)
```

Listas arbitrariamente largas

Ejemplo

1. Diseñe una función **multiplicar-lista** que recibe una lista de números. Esta función retorna los números de la lista multiplicados entre sí.
2. Diseñe una función **suma-lista-dobles** que recibe una lista de números y un número, esta retorna la suma de la multiplicación de cada uno de los elementos de la lista por el número.
3. Diseñe una función **elevar-cuadrado-lista** recibe una lista de números y esta retorna esa misma lista pero con los elementos elevados al cuadrado

Contenido

Estructuras recursivas (Árboles)

Definición

Existen estructuras cuyos campos pueden definirse con una estructura, un buen ejemplo de ello es una **muñeca rusa**

Definición

¿Como definiríamos una estructura que sea una muñeca rusa:

```
(define-struct rusa-doll
        (doll-interna)
 )
```

Estructuras recursivas (Árboles)

Fundamentos de programación

Recursión numérica

Listas arbitrariamente largas

Estructuras recursivas (Árboles)

Definición

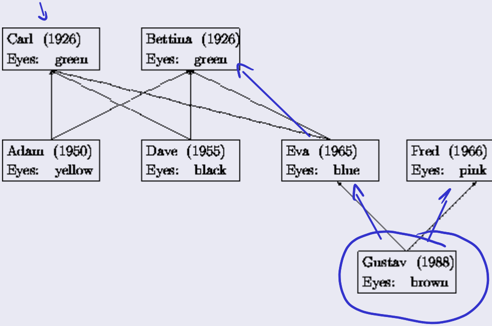
Si queremos una muñeca que contenga otras dos adentro sería.

```
(make-rusa-doll
    (make-rusa-doll
       (make-rusa-doll empty)))
```

(define-struct child (name birthday color-eyes father mother))

Definición

Un caso más aplicado, son los árboles genealógico donde podemos relacionar los parentescos, por ejemplo:



Estructuras recursivas (Árboles)

Definición

Podemos definir un hijo (child) de la siguiente forma:

```
(define-struct child
        (padre madre nombre fecha ojos)
 )
```

Estructuras recursivas (Árboles)

Definición

Por ejemplo podemos definir a alguien en la cima del árbol:

```
(make-child empty empty 'Carl 1926 'green)
```

Estructuras recursivas (Árboles)

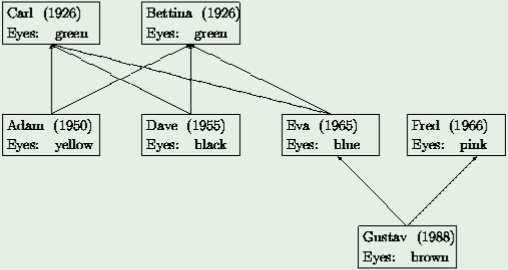
Definición

Pero que pasa con un hijo:

```
(make-child
     (make-child empty empty 'Carl 1926 'green)
     (make-child empty empty 'Bettina 1926 'green)
     'Adam
     'yellow
     )
```

Ejercicio

Defina las estructuras para el resto de la familia (en hoja de papel)



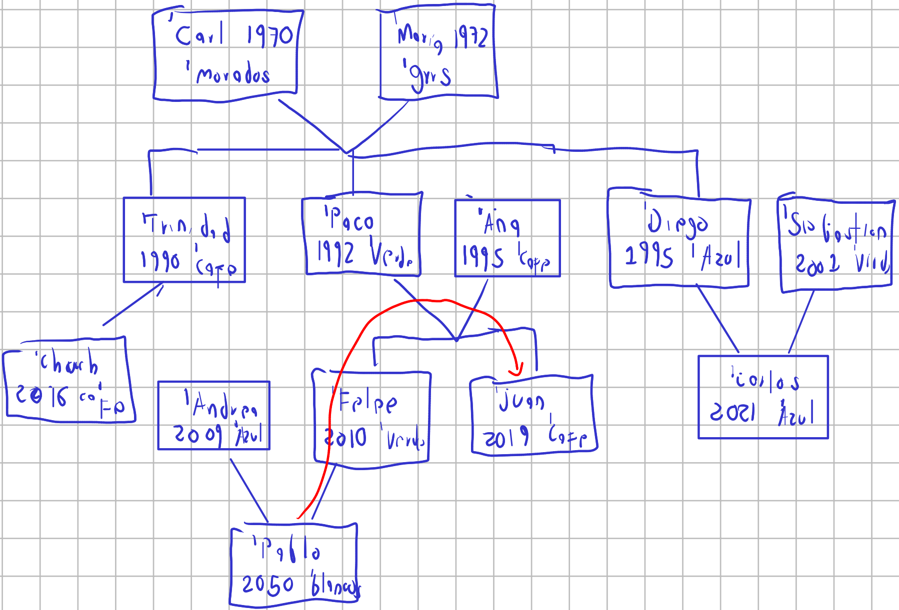
Estructuras recursivas (Árboles)

Ejercicio

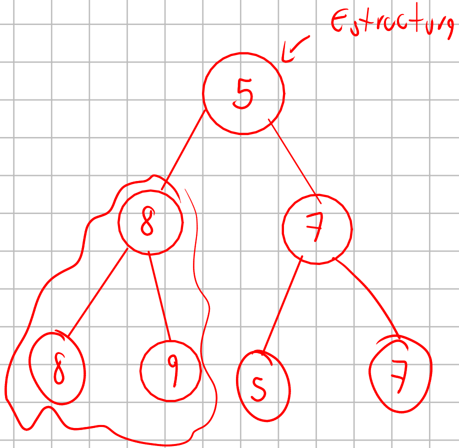
```
;; Primera Generación:
(define Carl (make-child empty empty 'Carl 1926 'green))
(define Bettina (make-child empty empty 'Bettina 1926 '
    green))

;; Segunda Generación:
(define Adam (make-child Carl Bettina 'Adam 1950 'yellow)
    )
(define Dave (make-child Carl Bettina 'Dave 1955 'black))
(define Eva (make-child Carl Bettina 'Eva 1965 'blue))
(define Fred (make-child empty empty 'Fred 1966 'pink))

;; Tercera Generación:
(define Gustav (make-child Fred Eva 'Gustav 1988 'brown))
```



'Carl 1970
'morados
'Maria 1972
'gris
Trinidad
1990 'cafe
'Paco
1992 'Verde
'Ana
1995 'cafe
'Diego
1995 'Azul
'Sebastian
2002 'Verde
'charli
2016 'cafe
'Andrea
2009 'Azul
'Felipe
2010 'Verde
'Juan
2019 'cafe
'carlos
2021 'Azul
'Pablo
2050 'blanco

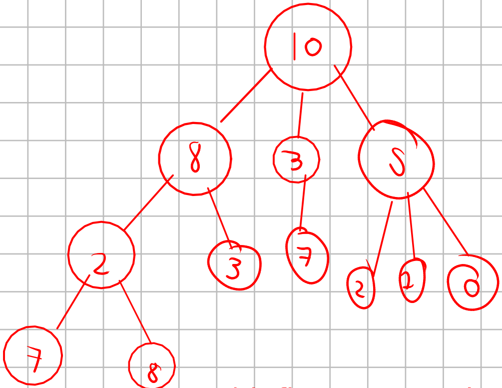


binarios

ternario

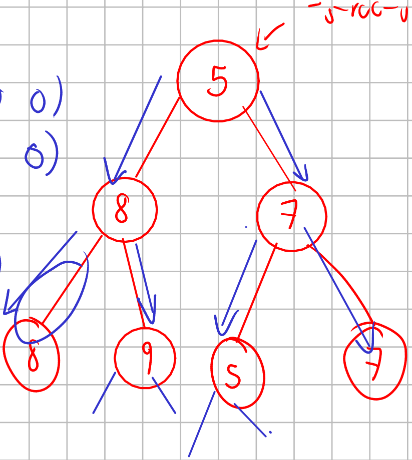
n-arios

(define-struct arbol (valor hizq hder))



(define-struct arbol-t (valor hi1 hi2 hi3))

t(5
t(8
t(8 0 0)
t(9 0 0)
t(7
t(5 0 0)
t(7 0 0)

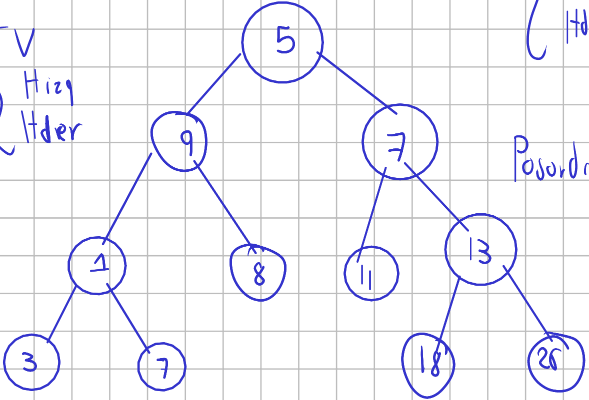


t(valor
•(___ hizq)
[(_ hder)

```
(define-struct arbol
    (valor hizq hder)
)
```

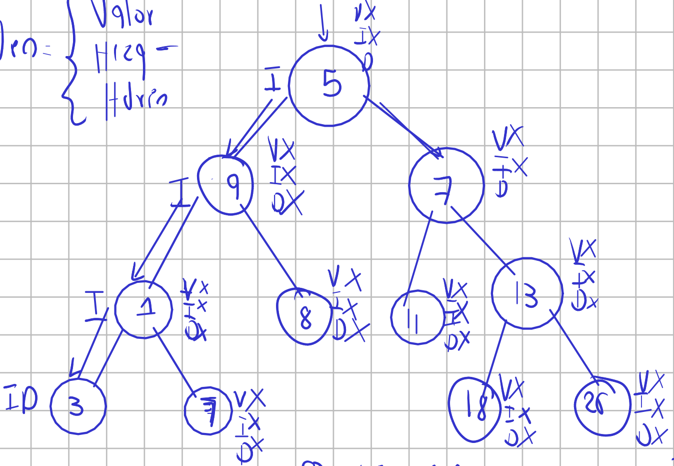
Inorden {Hizq Valor Hder}

Preorden {V Hizq Hder}

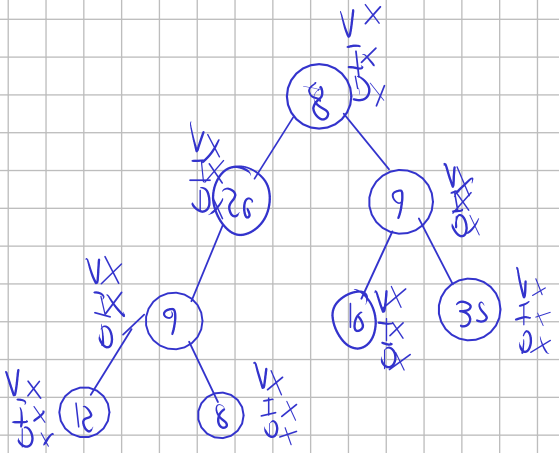


Posorden {Hizq Hder Valor}

Preorden = { Valor, Hizq, Hdrch }



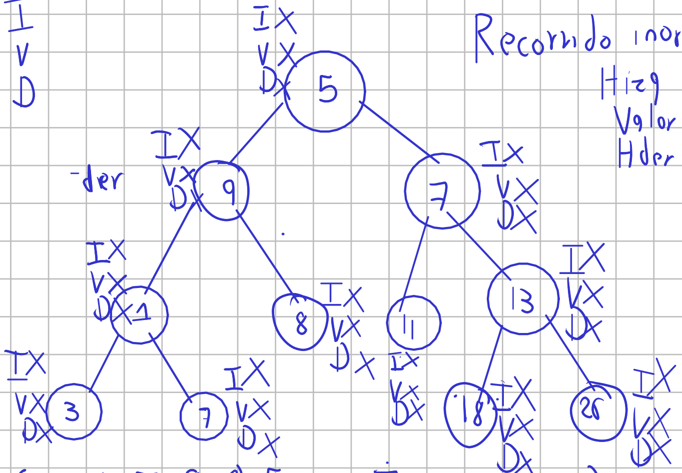
(5 9 1 3 7 8 7 11 13 18 26)



(8 26 9 12 8 9 10 35)

I
V
D

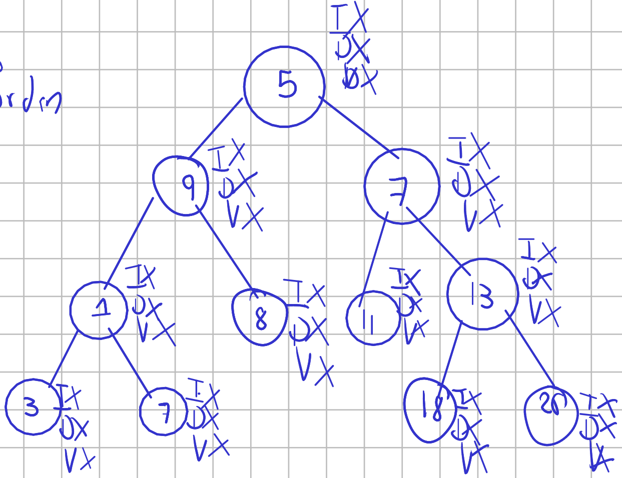
Recorrido inorden
Hizq
Valor
Hder



(3 1 7 9 8 5 11 7 18 13 26)

Recorrido
posorden

Hizq
Hder
Valor



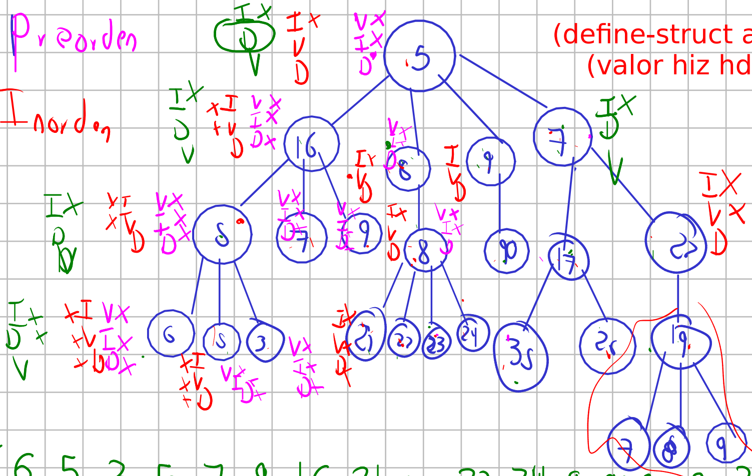
(3 7 1 8 9 11 18 26 13 7 5)

Preorden

Inorden

(define-struct arbol-m
(valor hiz hd1 hd2 hd3))

Posorden

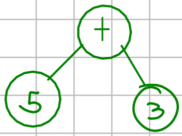


(6 5 3 5 7 9 16 21 22 23 24 8 8 90 9 35 26 17
7 8 9 19 22 7 5)

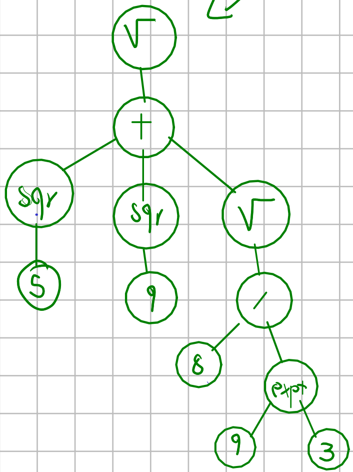
(6 5 5 3 16 7 9 5 21 8 22 23 24 8 90 9 35 17 26 7
7 19 8 9 22)

(5 16 5 6 5 3 7 9 8 8 21 22 23 24 9 90 7 17 35 26 22
19 7 8 9)

$5+3$



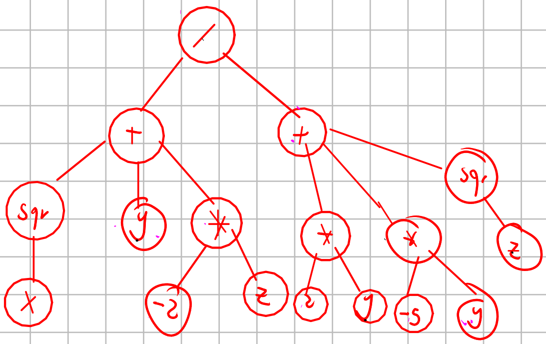
$$\sqrt{5^2+9^2+\sqrt{\frac{8}{9^3}}}$$



Preorden V I D

(√ (+ (sqr 5) (sqr 9) (√ (/ 8 (expt 9 3))))))

$$\frac{x^2 + y - 2z}{2y - 5y + z^2}$$



Inorden I R

$$\frac{(x^2) + y - 2 * z}{2 * y + -5 * y + z^2}$$

Posorden I D V

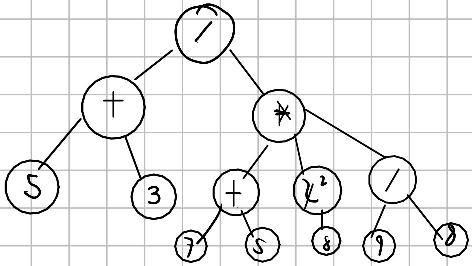
(((x sqr) y (-2 z *) +)
((2 y *) (-5 y *) (z sqr) +)
/)

Preorden

(/ (+ (sqr x) y (* -2 z))
(+ (* 2 y) (* -5 y) (sqr z))
)

(/ (+ 5 3)
(* (+ 7 5) (sqr 8) (/ 9 8)))
)

Prefijo, infijo
posfijo



$$(5+3) \, / \, \left((7+5) * 8^2 * \frac{9}{8}\right)$$

¿Preguntas?

Fundamentos de programación

Recursión numérica

Listas arbitrariamente largas

Estructuras recursivas (Árboles)