

Fundamentos de lenguajes de programación

Robinson Duque, M.Eng, Ph.D

Universidad del Valle

robinson.duque@correounalvalle.edu.co

Programa de Ingeniería de Sistemas
Escuela de Ingeniería de Sistemas y Computación



Este documento es una adaptación del material original de los profesores
Carlos Andres Delgado y Carlos Alberto Ramírez

- 1 Especificación recursiva de datos
 - Especificación inductiva
 - Especificación mediante gramáticas en forma BNF
- 2 Especificación recursiva de programas
 - Ejemplos
- 3 Los conceptos de Alcance y Ligadura de una variable

Especificación Recursiva de datos

- Cuando se escribe un procedimiento, se debe definir que clase de valores se espera como entrada y como salida.
- Ejemplo, la función suma tiene como entrada dos números naturales y tiene como salida un número natural.
- Los datos en las funciones recursivas, pueden tener también definiciones recursivas que faciliten la programación.

Especificación Recursiva de datos

Técnicas

Existe dos técnicas para la definición recursiva de datos:

- ① Especificación inductiva
- ② Especificación mediante gramáticas.

Especificación inductiva

La *especificación inductiva* es un método para especificar un conjunto de valores:

Definición

Se define un conjunto S , el cual es el conjunto más pequeño que satisface las siguientes dos propiedades:

- ① Algunos valores específicos que deben estar en S .
- ② Si algunos valores están en S , entonces otros valores también están en S .

Especificación inductiva

(Top-Down definition) Números múltiplos de 3

Un *número natural* n está en S si y sólo si:

- ① $n = 0$, o
- ② $n - 3 \in S$

Sabemos que $0 \in S$, entonces $3 \in S$ dado que $(3 - 3) = 0$ y $0 \in S$. Igualmente $6 \in S$ dado que $(6 - 3) = 3$ y $3 \in S$...Podemos concluir que S es el conjunto de números naturales que son múltiplos de 3.

¿Está 1 en S ? sabemos que $1 \neq 0$, entonces la primer condición no se satisface, adicionalmente $(1 - 3) = -2$, lo cual no es un número natural y por consiguiente no es un número de S , así la segunda condición no se satisface.

Especificación inductiva

(Top-Down definition) Números múltiplos de 3

Un *número natural* n está en S si y sólo si:

① $\underline{n = 0}$, o

$n = 2, 0$

② $n - 3 \in S$

$n - 3 \in S$

Podemos utilizar esta definición para escribir un procedimiento (*predicado*) que decida si un número natural n está en S :

```
;in-S? : N → Bool  
;usage: (in-S? n) = #t if n is in S, #f otherwise
```

```
(define in-S?  
  (lambda (n)  
    (if (zero? n) #t  
        (if (≥ (- n 3) 0)  
            (in-S? (- n 3))  
            #f))))
```

($\text{if } \underline{\quad ? \quad}$
 $\underline{\#T}$
 $\times \underline{\#F} \quad)$)

Especificación inductiva

(Bottom-up definition) Números múltiplos de 3

Definición del conjunto más pequeño de S contenido en los números naturales N :

① $n \in 0, y$

② si $n \in S$, entonces $n + 3 \in S$.

$0 \in S$
 $3 \in S$
 $6 \in S$

$9 \in S$

(Rules of inference definition) Números múltiplos de 3

Definición utilizando reglas de inferencia. Cada entrada es una regla; la línea horizontal se lee como “si - entonces”; la parte superior es la hipótesis (o antecedente); la parte inferior es la conclusión (o consecuente):

$\overline{0 \in S} \leftarrow \text{axiom}$

$\frac{}{\rightarrow n \in S}$
 $\frac{}{(n + 3) \in S}$

Especificación inductiva

Números pares

- ① Si $n = 2$ entonces n es par
- ② Si n es par, entonces $n + 2$ también es par.

Lista de números

- ① *empty* es una lista de números
- ② Si n es un número y l es una lista entonces $(\text{cons } n \ l)$ es una lista de números

Especificación inductiva

Especificación mediante reglas de inferencia:

Números pares

- ① $2 \in S$
- ② $\frac{n \in S}{(n+2) \in S}$

Lista de números

- ① $\text{empty} \in S$
- ② $\frac{l \in S, n \in N}{(\text{cons } n \ l) \in S}$

$L = (\text{cons } q \ \text{empty})$
 $(\text{cons } s \ L)$

Lista de números (otra posible especificación)

- ① $() \in \text{List-of-Int}$
- ② $\frac{l \in \text{List-of-Int}, n \in \text{Int}}{(n \ . \ l) \in \text{List-of-Int}}$

Especificación inductiva

Especificación formal

Ahora realicemos la especificación inductiva de:

- ① Una lista de número pares
- ② Múltiplos de 5

Especificación inductiva

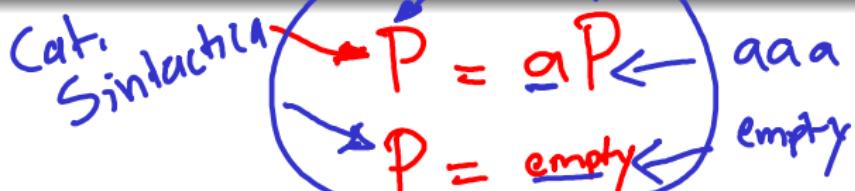
Lista de números pares

- ① $\text{empty} \in S$
- ②
$$\frac{l \in S, n \in N}{(\text{cons } 2n \ l) \in S}$$

Múltiplos de 5

- ① $5 \in S$
- ②
$$\frac{n \in S}{(n+5) \in S}$$

Especificación mediante gramáticas



- Una forma sencilla de especificar datos recursivos es con gramáticas regulares en forma Backus-Naur.
- Las gramáticas se componen de:
 - Símbolos no terminales, que son aquellos que se componen de otros símbolos, son conocidos como categorías sintácticas
 - Símbolos terminales: corresponden a elementos del alfabeto
 - Reglas de producción

Especificación mediante gramáticas

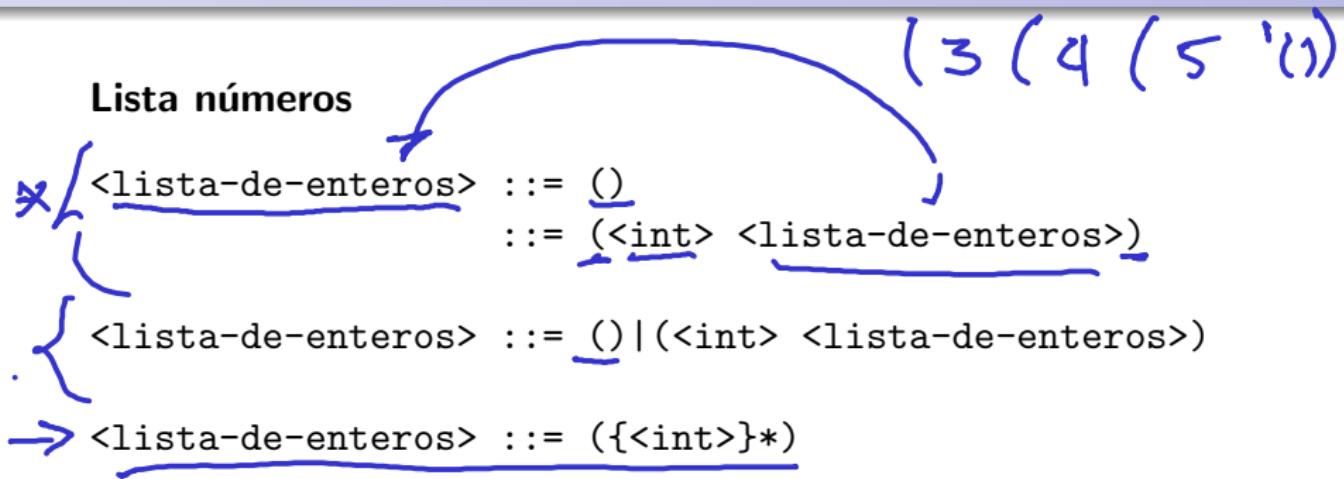
$$P = aP$$

- Alfabeto: Conjunto de símbolos, ejemplo $\sum\{a; b, c\}$
- Reglas de producción: Construcción del lenguaje:
 -  Cerradura de Kleene: $\{a\}^* = \{\epsilon, \{a\}, \{a, a\}, \{a, a, a\} \dots\}$
 - Cerradura positiva: $\{b\}^+ = \{\{b\}, \{b, b\}, \{b, b, b\} \dots\}$

$$\{b\}^+ = \{a\}^* \cup a$$

Especificación mediante gramáticas

Lista números



¿cuáles de las siguientes listas son listas válidas según la gramática?

- (2 3 t 6 7) ✗
- (2 3 5 6 7) ✓
- (2 3 (5 6) 7) ✗

Especificación mediante gramáticas

(a b c d ()(b) ())

Listas de símbolos y listas

<S-list> ::= ({<S-exp>}*)

<S-exp> ::= <Symbol> | <S-list>

Scheme-value

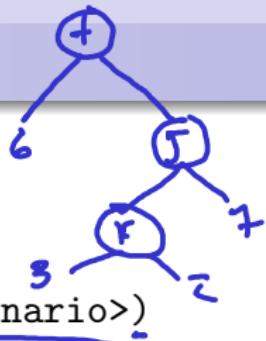
¿cuáles de las siguientes listas son válidas según la gramática?

- (a b c) ✓
- (an (((s-list)) (with () lots) ((of) nesting))) ✓
- (a 3 (b c) 4) ✗

Especificación mediante gramáticas

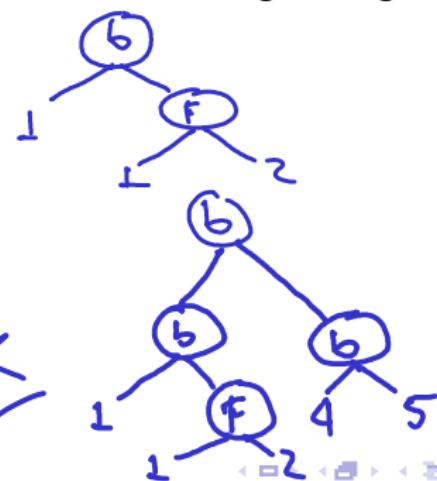
Árbol Binario

$\langle \text{arbol-binario} \rangle ::= \langle \text{int} \rangle$
 $::= (\langle \text{simbolo} \rangle \langle \text{arbol-binario} \rangle \langle \text{arbol-binario} \rangle)$



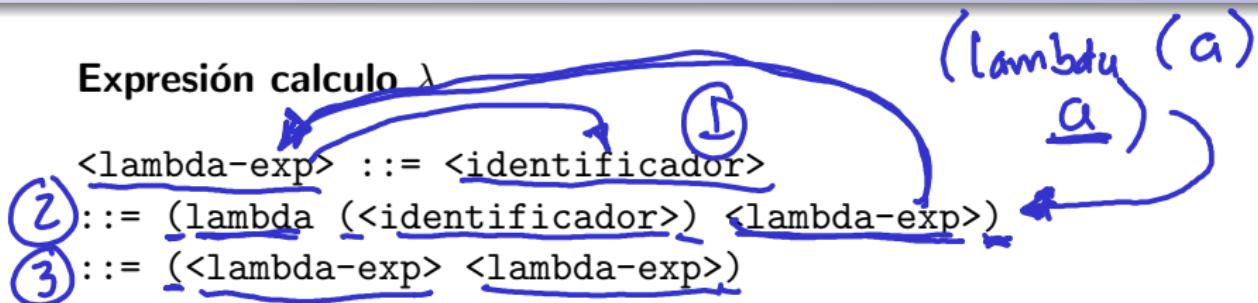
¿cuáles de los siguientes árboles son válidos según la gramática?

- 2 ✓
- (f 1 2) ✓
- (f 1 2)X ✗
- (b 1 (f 1 2)) ✓
- (b x (f 1 2)) ✗
- (b (bX(f 1 2)) (b 4 5)) ✗
- (b (b 1 (f 1 2)) (b 4 5)) ✓



Especificación mediante gramáticas

Expresión calculo λ



El **cálculo lambda** es un **lenguaje simple** que se utiliza frecuentemente para estudiar la teoría de lenguajes de programación. Consiste sólo de referencias a variables, procedimientos que toman un único argumento, y llamados a procedimientos.

Especificación mediante gramáticas

Expresión calculo λ

```

<lambda-exp> ::= <identificador>
 ::= (lambda (<identificador>) <lambda-exp>)
 ::= (λ<lambda-exp> <lambda-exp>)
  
```

¿Cuáles de las siguientes expresiones lambda son válidas según la gramática?

- x ✓
- (s y) ✓
- (s 1) ✗
- (lambda (y) x) ✓
- ((lambda (y) x) z) ✓
- ((lambda (y) x) (lambda (u) u)) ✓
- (lambda (x) ((lambda (y) y) p)) ✗
- (lambda (x) ((lambda (y) y))) ✗

Especificación mediante gramáticas

Expresión calculo λ

Expresiones

En el cálculo λ las funciones son ciudadanos de primera clase

```
(lambda (x) (+ x 1)) ;Como función
((lambda (x y) (* x y)) 1 2) ;Como valor
```

Funciones pueden pasarse como parámetros y retornarse al igual que los valores.

Especificación recursiva de programas

- La definición inductiva o mediante gramáticas de los conjuntos de datos sirve de guía para desarrollar procedimientos que operan sobre dichos datos
- La estructura de los programas debe seguir la estructura de los datos

¡Sigue la gramática!

Especificación recursiva de programas

Gramática: Listas en Scheme

```
<List> ::= () ←
          ::= (<Scheme-Value> <List>) ↗
```

Función que calcula la longitud de una lista

```
; list-length : List → Int
; usage: (list-length l) = the length of l
(define list-length
  (lambda (lst)
    (if (null? lst)
        0
        (+ 1 (list-length (cdr lst))))))
```

; Cómputo de list-length en ejecución:

```
(list-length '(a (b c) d))
= (+ 1 (list-length '((b c) d)))
= (+ 1 (+ 1 (list-length '(d))))
= (+ 1 (+ 1 (+ 1 (list-length '()))))
= (+ 1 (+ 1 (+ 1 0)))
= 3
```

Especificación recursiva de programas

Gramática: Listas en Scheme

```
<List> ::= ()
          ::= (<Scheme-Value> <List>)
```

Función que retorna el elemento en la posición n

```
; nth-element : List x Int -> SchemeVal
; usage: (nth-element lst n) = the n-th element of lst

(define nth-element
  (lambda (lst n)
    (if (null? lst)
        (report-list-too-short n)
        (if (zero? n)
              (car lst)
              (nth-element (cdr lst) (- n 1))))))

(define report-list-too-short
  (lambda (n)
    (eopl:error 'nth-element "List too short by ~s elements.~%" (+ n 1)))))
```

Especificación Recursiva de programas

Un árbol binario

Recordando la definición de los árboles vista anteriormente:

```
<arbol-binario> ::= <int>  
 ::= (<simbolo> <arbol-binario> <arbol-binario>)
```

Este árbol será representado con una lista, debido a la definición recursiva. Ejemplo:

```
(define arbol '(k (h 5 3) (t (s 10 11) 12)))  
(define otroArbol 2)
```

Especificación Recursiva de programas

Gramática: árbol binario

```
<arbol-binario> ::= <int>
 ::= (<simbolo> <arbol-binario> <arbol-binario>)
    · car      cadr      c
```

Función que suma los elementos de un árbol binario

```
;sum-arbol : arbol-binario -> Int
;usage: (sum-arbol arbol) = suma de las hojas del árbol

(define sum-arbol
  (lambda (arbol)
    (if (number? arbol)
        arbol
        (+ (sum-arbol (cadr arbol))
           (sum-arbol (caddr arbol)))
        )
    )
)
```

Especificación Recursiva de programas

Gramática: árbol binario

```
<arbol-binario> ::= <int>  
 ::= (<símbolo> <arbol-binario> <arbol-binario>)
```

Ejercicios

- Crear una función que cuente la cantidad de números enteros en un árbol binario, según la gramática dada.
- Crear una función que cuente la cantidad de símbolos en un árbol binario, según la gramática dada.
- Crear una función que retorne una lista con los símbolos de un árbol binario, según la gramática dada.
- Crear una función que retorne una lista con los números de un árbol binario, según la gramática dada.
- Crear una función que retorne una lista con los números de un árbol binario que satisfagan un predicado P.

Especificación Recursiva de programas

Gramática: lista de números

```
<lista-numeros> ::= '() | (<int> <lista-numeros>)
```

Función que suma los elementos de una Lista números

```
;sum-lista-numeros : lista-numeros -> Int
;;usage: (sum-lista-numeros lst) = suma los elementos de la lista

(define suma-lista-numeros
  (lambda (lst)
    (if (eqv? lst empty)
        0
        (+ (car lst) (suma-lista-numeros (cdr lst)))))
```

Los conceptos de Alcance y Ligadura de una variable

- El concepto de variable es fundamental en los lenguajes de programación
- Una variable puede ser declarada o referenciada

- Declaración:

```
(lambda (x) ...)  
(let ((x ...)) ...)
```

- Referencia:

```
(f x y)
```

Los conceptos de Alcance y Ligadura de una variable

- Una variable está ligada al lugar donde se declara
- El valor referenciado por la variable es su denotación
- Cada lenguaje de programación tiene asociadas unas reglas de ligadura que determinan a qué declaración hace referencia cada variable
- Dependiendo del momento de aplicación de las reglas (antes o durante la ejecución), los lenguajes se denominan de alcance estático o alcance dinámico

Los conceptos de Alcance y Ligadura de una variable

- Una variable x ocurre libre en una expresión E si y solo si existe algún uso de x en E el cual no está ligado a ninguna declaración de x en E

```
((lambda (x) x) y)
```

Los conceptos de Alcance y Ligadura de una variable

Gramática de una expresión λ

```
<lambda-exp> ::= <identificador>  
 ::= (lambda <identificador> <lambda-exp>) cadr caddr ←  
 ::= (<lambda-exp> <lambda-exp>) ←  
 cadr caddr (Lambda (x) y)
```

Determinar si una variable ocurre libre

Así mismo podemos definir un procedimiento para determinar si una variable ocurre libre

```
(define occurs-free?  
  (lambda (var exp)  
    (cond  
      ((symbol? exp) (eqv? var exp)) ←  
      ((eqv? (car exp) 'lambda) lambda) ←  
      (and (not (eqv? (caadr exp) var))  
            (occurs-free? var (caddr exp))))  
      (else (or (occurs-free? var (car exp))  
                (occurs-free? var (cadr exp)))))))
```

Los conceptos de Alcance y Ligadura de una variable

Se define como el alcance de una variable como la región dentro del programa en el cual ocurren todas las referencias a dicha variable.

```
(define x
  (lambda (x)
    (map
      (lambda (x)
        (+ x 1))
      x)))
(x '(1 2 3))
```

; Variable x1
; Variable x2
; Variable x3
; Ref x3
; Ref x2
; Ref x1

Los conceptos de Alcance y Ligadura de una variable

Ejemplo:

```
(lambda (z)
  ((lambda (a b c)
    (a (lambda (a)
          (+ a c)))
   b))
  (lambda (f x)
    (f (z x)))))
```

```
(lambda (z)
  ((lambda (a b c)
    (a (lambda (a)
          (+ a c)))
   b))
  (lambda (f x)
    (f (z x)))))
```

Los conceptos de Alcance y Ligadura de una variable

Cual es el valor de la siguiente expresión:

```
(let ((x 3) (y 4))
      (+ (let ((x (+ y 5)))
           (* x y))
          x)
     )
```

R/= 39

Los conceptos de Alcance y Ligadura de una variable

Cual es el valor de la siguiente expresión:

```
(let ((x 6)
      (y 7))
  (+ (let ((x (- y 6)))
       (* x y))
     x))
```

R/= 13

Los conceptos de Alcance y Ligadura de una variable

Cual es el valor de la siguiente expresión:

```
(let ((x 6)(y 7))
  (*
    (let ((y 8))
      (+
        (let ((x 6) (y x))
          (+ x
              (let ((y 3) (x y)) (+ x (+ 2 y))))
            )
          y)
        )
      (let ((x 4)) (- y x))
    )
  )
```

R/= 75

Próxima sesión

Abstracción de datos (Capítulo 2 EOPL)