La Relación entre Inducción y Programación

Profesor: Juan Francisco Diaz
jdiaz@eisc.univalle.edu.co

Asistente de Docencia: Gerardo M. Sarria M.
gsarria@eisc.univalle.edu.co

March 3, 2002

Especificación Recursiva de Datos

Cuando se escribe un procedimiento, se debe conocer con precisión los valores que pueden ocurrir como argumentos al procedimiento, y los valores que puede retornar dicho procedimiento.

Las técnicas para especificar el conjunto de valores anteriormente nombrado son las siguientes:

1. Especificación Inductiva

Se define un conjunto S, el cual es el conjunto más pequeño que satisface dos propiedades de la siguiente forma:

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

Ejemplo 1:

Definición 1 *El conjunto* lista-de-números *es el conjunto más pequeño de valores que satisface las siguientes dos propiedades:*

1. *La lista vacía es una lista-de-números, y*
2. *Si l es una lista-de-números y n es un número, entonces la pareja $(n.l)$ es una lista-de-números.*

2. Backus-Naur Form (BNF)

Se define un conjunto de *reglas de producción* (llamado gramática), el cual tiene un lado izquierdo donde se encuentran los símbolos no terminales y un lado derecho que consiste en símbolos terminales y no terminales. Ambos lados están separados por el símbolo ::=, que se lee *es* o *puede ser*.

Los símbolos no terminales son nombres de conjuntos[a] que están siendo definidos, y se escriben entre '⟨' y '⟩'. Los símbolos terminales son los caracteres en la representación externa.

Ejemplo 2:

$$
\begin{array}{lcl}
\langle \text{lista-de-números} \rangle & ::= & () \\
\langle \text{lista-de-números} \rangle & ::= & (\langle \text{número} \rangle . \langle \text{lista-de-números} \rangle)
\end{array}
$$

[a]Algunas veces llamados *categorías sintácticas*

2. BNF (continuación)

BNF ofrece también alternativas de notación.

- Cuando se tiene una sola categoría sintáctica para un conjunto de reglas de producción, simplemente se escribe el lado izquierdo y '::=' una sola vez, seguido de todos los lados derechos separados por el símbolo especial '|' (la barra vertical que se lee *ó*).

  Ejemplo 3:

  $$\begin{aligned}\langle\text{árbol-binario}\rangle ::= \;& \langle\text{número}\rangle \mid \\ & (\langle\text{símbolo}\rangle\ \langle\text{árbol-binario}\rangle\ \langle\text{árbol-binario}\rangle)\end{aligned}$$

- Se puede omitir el lado izquierdo de la regla de producción cuando éste es el mismo de la regla predecesora.

  Ejemplo 4:

  $$\begin{aligned}\langle\text{expresión}\rangle \;& ::= \langle\text{identificador}\rangle \\ & ::= (\texttt{lambda}\ (\langle\text{identificador}\rangle)\ \langle\text{expresión}\rangle) \\ & ::= (\langle\text{expresión}\rangle\ \langle\text{expresión}\rangle)\end{aligned}$$

- La notación $\{\ldots\}^*$ o *Kleene star* indica la secuencia de cero o más instancias de lo que esté dentro de las llaves. La notación $\{\ldots\}^+$ o *Kleene plus* indica la secuencia de uno o más instancias.

  Ejemplo 5:

  $$\begin{aligned}\langle\text{lista-s}\rangle \;& ::= \; (\{\langle\text{expresión-símbolo}\rangle\}^*) \\ \langle\text{expresión-símbolo}\rangle \;& ::= \; \langle\text{símbolo}\rangle \mid \langle\text{lista-s}\rangle\end{aligned}$$

2. BNF (continuación)

Si un conjunto es especificado usando BNF, se puede probar que un valor es un miembro del conjunto usando la *derivación sintáctica*. Ésta comienza con el símbolo no terminal del conjunto, luego en cada paso (indicado por $\Rightarrow$), cada símbolo no terminal es reemplazado por el lado derecho correspondiente a la regla.

Ejemplo 6:

Se demuestra que (14 . ()) es una lista de números.

$$
\begin{array}{ll}
 & \langle \text{lista-de-números} \rangle \\
\Rightarrow & (\langle \text{número} \rangle . \langle \text{lista-de-números} \rangle) \\
\Rightarrow & (14 \ . \ \langle \text{lista-de-números} \rangle) \\
\Rightarrow & (14 \ . \ ())
\end{array}
$$

3. Inducción

Se usan definiciones inductivas para probar teoremas de miembros de conjuntos y escribir programas que los manipulan.

Ejemplo 7:

Teorema 1 *Sea* $s \in \langle$*árbol-binario*$\rangle$*,*
donde $\langle$*árbol-binario*$\rangle$ *esta definido como*

$$\begin{array}{rcl} \langle \textit{árbol-binario} \rangle & ::= & \langle \textit{número} \rangle \mid \\ & & (\langle \textit{símbolo} \rangle\ \langle \textit{árbol-binario} \rangle\ \langle \textit{árbol-binario} \rangle) \end{array}$$

Entonces s contiene una número impar de nodos.

Especificación Recursiva de Programas

Una BNF para los tipos de datos sirve de guía para encontrar donde deben usarse los llamados recursivos y cuales son los casos básicos a ser manejados.

Se necesitará un procedimiento por cada categoría sintáctica en la gramática; para cada símbolo no terminal que aparezca en el lado derecho se necesitará un llamado recursivo al procedimiento.

Ejemplo 8:

Construya un procedimiento en Scheme `list-of-numbers?` que toma una lista y determina si ella pertenece a la categoría sintáctica ⟨lista-de-números⟩. Esto es,

```
> (list-of-numbers? '(1 2 3))
#t
> (list-of-numbers? '(1 two 3))
#f
> (list-of-numbers? '(1 (2) 3))
#f
```

Ejemplo 8 (continuación)

Primero se recuerda la definición de $\langle$lista-de-números$\rangle$:

$$\langle \text{lista-de-números} \rangle ::= () \mid (\langle \text{número} \rangle . \langle \text{lista-de-números} \rangle)$$

De allí, se comienza a escribir el procedimiento con el comportamiento más simple: qué sucede si el argumento es una lista vacía.

```
(define list-of-numbers?
  (lambda (lst)
    (if (null? lst)
      ...
      ...)))
```

Ejemplo 8 (continuación)

De la regla de producción, vemos que una lista vacía es una ⟨lista-de-números⟩, luego la respuesta debe ser `#t` (verdadero o *true*).

```
 (define list-of-numbers?
   (lambda (lst)
     (if (null? lst)
|        #t
         ...)))
```

Ejemplo 8 (continuación)

Ahora, de la segunda parte del lado derecho de la regla de producción, podemos ver que una ⟨lista-de-números⟩ es una lista donde el primer elemento es un ⟨número⟩ y el resto es una ⟨lista-de-números⟩. Por esto, si la lista argumento no es vacía, se debe hacer un llamado recursivo así:

```
 (define list-of-numbers?
   (lambda (lst)
     (if (null? lst)
       #t
|      (and
|        (number? (car lst))
|        (list-of-numbers? (cdr lst))))))
```

Y queda escrito el procedimiento.

Ejemplo 9:

Construya el procedimiento `remove-first` en Scheme, que toma como argumentos un símbolo s y una lista de símbolos los, y que retorna una lista con los mismos elementos y en el mismo orden que los, exceptuando la primera ocurrencia de s. Esto es,

```
> (remove-first 'a '(a b c))
(b c)
> (remove-first 'b '(e f g))
(e f g)
> (remove-first 'a4 '(c1 a4 c1 a4))
(c1 c1 a4)
> (remove-first 'x '())
()
```

Ejemplo 9 (continuación)

Antes de construir el procedimiento, se define el conjunto ⟨lista-de-símbolos⟩:

$$\langle\text{lista-de-símbolos}\rangle ::= () \mid (\langle\text{símbolo}\rangle.\langle\text{lista-de-símbolos}\rangle)$$

Ahora, si la lista argumento es vacía, no hay ocurrencias de s para ser removidas, luego la respuesta es la lista vacía:

```
(define remove-first
  (lambda (s los)
    (if (null? los)
      '()
      ...)))
```

Ejemplo 9 (continuación)

Si *los* no es vacía, se tienen dos posibilidades. La primera es que *s* sea el primer elemento de *los*, que conduciría simplemente a retornar la cola de la lista:

```
 (define remove-first
   (lambda (s los)
     (if (null? los)
       '()
|      (if (eqv? (car los) s)
|        (cdr los)
|        ...))))
```

Ejemplo 9 (continuación)

La segunda posibilidad, es que s no sea el primer elemento.
Por la regla de producción, la segunda parte del lado derecho muestra que una ⟨lista-de-símbolos⟩ tiene un ⟨símbolo⟩ seguido de una ⟨lista-de-símbolos⟩; esto significa que s podría ser el primer elemento del resto de *los*, lo que lleva a retornar una lista constituida por el primer elemento de *los* y lo que retorne el llamado recursivo al procedimiento con la cola de *los* como argumento:

```
(define remove-first
  (lambda (s los)
    (if (null? los)
      '()
      (if (eqv? (car los) s)
        (cdr los)
|       (cons (car los) (remove-first s (cdr los)))))))
```

Y queda escrito el procedimiento.

Ejemplo 10:

El siguiente ejemplo es la contrucción en Scheme del procedimiento **subst**, el cual toma tres argumentos: dos símbolos, **new** y **old**, y una lista-s, **slist**, y retorna una lista similar a **slist** pero con todas las ocurrencias de **old** reemplazadas por instancias de **new**.

```
> (subst 'a 'b '((b c) (b () d)))
((a c) (a () d))
```

Ejemplo 10 (continuación)

Como `subst` se construye sobre listas-s, se recuerda la definición

$$
\begin{array}{lcl}
\langle\text{lista-s}\rangle & ::= & (\{\langle\text{expresión-símbolo}\rangle\}^*) \\
\langle\text{expresión-símbolo}\rangle & ::= & \langle\text{símbolo}\rangle \mid \langle\text{lista-s}\rangle
\end{array}
$$

Primero volvemos a escribir la gramática para eliminar el uso de *Kleene-star*:

$$
\begin{array}{lcl}
\langle\text{lista-s}\rangle & ::= & () \\
 & ::= & (\langle\text{expresión-símbolo}\rangle.\langle\text{lista-s}\rangle) \\
\langle\text{expresión-símbolo}\rangle & ::= & \langle\text{símbolo}\rangle \mid \langle\text{lista-s}\rangle
\end{array}
$$

Ejemplo 10 (continuación)

Ahora, como la gramática contiene dos símbolos no terminales, se deben tener dos procedimientos, uno para ⟨lista-s⟩ y otro para ⟨expresión-símbolo⟩:

```
(define subst
  (lambda (new old slist)
    ...))


(define subst-in-symbol-expression
  (lambda (new old se)
    ...))
```

Ejemplo 10 (continuación)

En `subst`, si la lista es vacía, no hay ocurrencias de `old` a reemplazar.

```
 (define subst
   (lambda (new old slist)
|    (if (null? slist)
|      '()
|      ...)))
```

De lo contrario, el primer elemento de `slist` es una ⟨expresión-símbolo⟩ y su cola es otra ⟨lista-s⟩. Luego, el procedimiento debe retornar una lista en la cual, el primer elemento es el resultado de cambiar `old` por `new` en el primer elemento de `slist`, y la cola es el resultado de cambiar `old` por `new` en la cola de `slist`. Pero como el primer elemento de `slist` es una ⟨expresión-símbolo⟩, se usa el procedimiento `subst-in-symbol-expression`:

```
 (define subst
   (lambda (new old slist)
     (if (null? slist)
       '()
|      (cons
|        (subst-in-symbol-expression new old (car slist))
|        (subst new old (cdr slist))))))
```

Ejemplo 10 (continuación)

En `subst-in-symbol-expression`, la gramática dice que el parámetro `se` es un símbolo o una list-s. Si es un símbolo, se pregunta si dicho símbolo es igual a `old`. Si la respuesta es afirmativa, se retorna `new`, de lo contrario se retorna el mismo símbolo `se`. Por otro lado, si `se` no es un símbolo, es una lista-s y debe llamarse al procedimiento `subst` para resolverlo.

```
 (define subst-in-symbol-expression
   (lambda (new old se)
|    (if (symbol? se)
|      (if (eqv? se old) new se)
|      (subst new old se))))
```

Alcance y Ligadura de una Variable

Conceptos Relacionados:

El Concepto de *variable* es fundamental en todos los lenguajes de programación.
Una variable puede ser *declarada* o *referenciada*.

Ejemplo 11:
Declaracion

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

Ejemplo 12:
Referencia

```
(f x y)
```

Conceptos Relacionados de Alcance y Ligadura (continuación)

Una variable esta *ligada* al lugar donde se declara. El valor referenciado por la variable es su *denotación*.

Cada lenguaje de programación tiene las denominadas *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*.

En (`lambda`(⟨identificador⟩)⟨expresión⟩), la ocurrencia de ⟨identificador⟩ es una declaración que liga todas las ocurrencias de esa variable en ⟨expresión⟩ a menos que ocurra una declaración interna a ⟨expresión⟩ de una variable con el mismo nombre.

Variables Libres y Ligadas

Una variable x *ocurre libre* en E si y solo si existe algún uso de x en E el cual no está ligado a ninguna declaración de x en E.
Una variable x *ocurre ligada* en una expresión E si y solo si existe algún uso de x en E el cual está ligado a una declaración de x en E.
Ejemplo 11:

En

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

x ocurre ligada, ya que la segunda ocurrencia de x es una referencia ligada por la primera ocurrencia de x (una declaración). Así mismo, y ocurre libre debido a que su ocurrencia en esta expresión no esta ligada a ninguna declaración.
Sin embargo si agregamos una línea

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

la variable y ocurriría ligada por la declaración del parámetro y en la línea agregada.

Variables Libres y Ligadas (continuación)

Una variable x ocurre libre en una expresión del cálculo lambda E si y solo si

1. E es una referencia a una variable y E es igual a x; o
2. E es de la forma (`lambda` (y) E'), donde y es diferente a x y x ocurre libre en E'; o
3. E es de la forma $(E_1\ E_2)$ y x ocurre libre en E_1 o E_2.

Una variable x ocurre ligada en una expresión del cálculo lambda E si y solo si

1. E es de la forma (`lambda` (y) E'), donde x ocurre ligada en E' o x y y son la misma variable y y ocurre libre en E'; o
2. E es de la forma $(E_1\ E_2)$ y x ocurre ligada en E_1 o E_2.

Alcance de una Variable

Se define el alcance de una variable como la región de texto dentro de la cual ocurren todas las referencias a dicha variable asociadas con su declaración, excluyendo las regiones internas asociadas con declaraciones que usan el mismo nombre de variable.

Ejemplo 12:

```
> (define x                                ; llamemos a esta x1
    (lambda (x)                            ; llamemos a esta x2
      (map
        (lambda (x)                        ; llamemos a esta x3
          (+ x 1))                         ; se refiere a x3
        x)))                               ; se refiere a x2
> (x '(1 2 3))                             ; se refiere a x1
(2 3 4)
```

Alcance de una Variable (continuación)

Algunas veces es de gran ayuda dibujar los bordes de las regiones de texto que delimitan el alcance de una variable. Esos bordes son llamados *contornos*.

Ejemplo 13:

```
(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)) ))    )
```