

**Taller 2: Semántica de lenguajes de programación Fundamentos de Interpretación y  
Compilación de Lenguajes de Programación**

Sofia Castillo Giraldo

Merly Velásquez Cortez

Faculta de Ingeniera, Universidad del Valle

Fundamentos de Interpretación y Compilación de Lenguajes de Programación

Carlos Andrés Delgado S.

30 de noviembre del 2024

## 1. Modificaciones a la gramática

`<expresion> ::= cons ( <expresion> <expresion> )`  
List-exp (valor lista)

`::= empty`  
list-empty-exp

`::= leight ( <expresión> )`  
list-length-exp (valor)

`::= first ( <expresión> )`  
list-first-exp (valor)

`::= rest ( <expresión> )`  
list-rest-exp (lista)

`::= nth ( <expresión> <expresión> )`  
list-nth-exp (lista valor)

Se agregaron nuevas expresiones en la gramática original, respetando la estructura y en consecuencia se adicionaron en la especificación gramatical, como se puede observar:

```
;; Expresiones de listas
(expression ("empty") list-empty-exp)
(expression ("cons" "(" expression expression ")") list-cons-exp)
(expression ("length" "(" expression ")") list-length-exp)
(expression ("first" "(" expression ")") list-first-exp)
(expression ("rest" "(" expression ")") list-rest-exp)
(expression ("nth" "(" expression expression ")") list-nth-exp)
```

Además, se agregó en la especificación gramatical la sentencia:

`<expresión> ::= cond {expression ==> expression} * else ==> expression end`

```
;; Expresión condicional
(expression ("cond" (arbno expression "==">" expression) "else" "==">" expression "end") cond-exp)
```

***Gramática final:***

`<expression> := cons ( <expression> <expression> )`  
List-exp (valor lista)

`:= empty`  
list-empty-exp

`:= leight ( <expresión> )`  
list-length-exp (valor)

`:= first ( <expresión> )`  
list-first-exp (valor)

`:= rest ( <expresión> )`  
list-rest-exp (lista)

`:= nth ( <expresión> <expresión> )`  
list-nth-exp (lista valor)

`:= cond ( <expresión> == > <expresión> ) * else == > <expresión> end`

Y por último, la asignación (Begin, set)

```
;;Asignación
(expression ("begin" expression (arbno ";" expression) "end") begin-exp)
(expression ("set" identificador "=" expression) set-exp)
```

## **2. Modificaciones a la función de evaluación-expresion**

La función `evalucion-expresion` se encarga de manejar los tipos de expresiones que podemos encontrar en la gramática mediante sus funciones, que tiene un procesamiento diferente dependiendo de lo que se ingrese en interpretador.

En esta función se agregaron las siguientes funciones: `empty`, `list-cons-exp`, `list-length-exp`, `list-first-exp`, `list-rest-exp`, `list-nth-exp` y `cond-exp`. En cada una hacemos verificación de la estructura de las listas y retornamos una expresión o lista dependiendo de la función invocada en el interpretador. También se agregaron las funciones `begin-exp` y `set-exp`, donde se evalúa bloques y se cambian valores ya asignados.

Con estos cambios el interpretador puede manejar más casos y poder trabajar mejor en la construcción del lenguaje.

### 3. Funciones creadas

#### a. Función *empty*

```
(list-empty-exp () '())
```

El propósito de esta función es crear una lista vacía, cuando en el interpretador ingresamos *empty*. Es útil a la hora de crear listas, dado de que podemos utilizar la operación *cons* para agregarle elementos a la lista vacía. Esta lista vacía es importante porque sirve como caso base.

#### b. Función *list-cons-exp*

```
(list-cons-exp (head tail)
  (let ((head-val (evaluar-expresion head amb))
        (tail-val (evaluar-expresion tail amb)))
    (if (list? tail-val)
        (cons head-val tail-val)
        (eopl:error "Error:" tail-val "no es una lista"))))
```

El propósito de esta función es permitir crear listas de tamaño *n*. Evalúa *head* y *tail*, verifica si *tail* es una lista, si es una lista agrega *head* al inicio de *tail* sino lanza error.

##### Ejemplo:

```
cons (1 cons (2 empty))
```

➤ (1 2)

```
cons(1 5)
```

➤ Error: 5 no es una lista

Como se menciona anteriormente, junto con *empty* podemos representar diferentes estructuras de listas.

#### c. Función *list-length-exp*

```
(list-length-exp (lst)
  (let ((lst-val (evaluar-expresion lst amb)))
    (if (list? lst-val)
        (length lst-val)
        (eopl:error "Argumento de length no es una lista" lst-val))))
```

El propósito de esta función es conocer el tamaño de cualquier lista ingresada en el intérprete, verifica si lo ingresado corresponde a una lista y posteriormente calcula su tamaño.

Esta función llega a ser útil en la implementación, debido a que al conocer su tamaño podemos tener en cuenta cómo manejar la estructura.

#### d. Función list-first-exp

```
(list-first-exp (lst)
  (let ((lst-val (evaluar-expresion lst amb)))
    (if (and (list? lst-val) (not (null? lst-val)))
        (car lst-val)
        (eopl:error "Argumento de first no es una lista no vacía" lst-val))))
```

El propósito de esta función es retorna el primer elemento de la lista (car).

La función espera una lista y retorna su car, si no es una lista o es una lista vacía genera un error. Con este tipo de verificación podemos manipular fácilmente las estructuras para poder darle un manejo adecuado y evitando errores.

#### e. Función list-rest-exp

```
(list-rest-exp (lst)
  (let ((lst-val (evaluar-expresion lst amb)))
    (if (and (list? lst-val) (not (null? lst-val)))
        (cdr lst-val)
        (eopl:error "Argumento de rest no es una lista no vacía" lst-val))))
```

El propósito de esta función retornar el resto de la lista (tail o cdr). La función espera una lista, y si no es una lista o es una lista vacía genera un error. Es útil para la manipulación de estructuras de listas igualmente ayuda a la iteración de estas.

#### f. Función list-nth-exp

```
(list-nth-exp (lst n)
  (let ((lst-val (evaluar-expresion lst amb))
        (n-val (evaluar-expresion n amb)))
    (if (and (list? lst-val) (integer? n-val) (>= n-val 0) (< n-val (length lst-val)))
        (list-ref lst-val n-val)
        (eopl:error "Argumentos inválidos para nth: lista o índice" lst-val n-val))))
```

El propósito de esta función es retorna el elemento en la posición n de la lista, evalúa una lista y un valor n, si la entrada es un a lista y n es un entero mayor o igual que 0 y menor que el tamaño de la lista, devuelve el elemento ubicado en el índice n.

Es útil para poder manipular cualquier elemento en cualquier posición y hacer una verificación correspondiente.

### g. Función cond-exp

```
(cond-exp (pairs else-exp end-exp)
  (letrec ((evaluar-cond (lambda (pairs)
    (cond
      [(null? pairs) (evaluar-expresion else-exp amb)
        (evaluar-expresion end-exp amb)]
      [(evaluar-expresion (car (car pairs)) amb)
        (evaluar-expresion (cdr (car pairs)) amb)]
      [else (evaluar-cond (cdr pairs))])))
    (evaluar-cond pairs)))
```

Esta función evalúa condiciones y retorna el primer valor que tome como verdadero y sin ninguna es verdadera, evalúa else-exp. Es útil para estructuras condicionales complejas.

### h. Función begin-exp

```
;;begin
(begin-exp (exp lexp)
  (if
    (null? lexp)
    (evaluar-expresion exp amb)
    (begin
      (evaluar-expresion exp amb)
      (letrec
        ((evaluar-begin (lambda (lexp)
          (cond
            [(null? (cdr lexp)) (evaluar-expresion (car lexp) amb)]
            [else
              (begin
                (evaluar-expresion (car lexp) amb)
                (evaluar-begin (cdr lexp)))])))
          (evaluar-begin lexp)))))
```

El propósito de esta función es retornar el último valor evaluado de la variable. Agrupa en un solo bloque, en este caso en el *begin*, evalúa en orden y devuelve el resultado de la última expresión.

Es útil porque nos ayuda a evaluar varias expresiones agrupadas facilitando la manipulación en el intérprete y además lo hace de forma recursiva.

### i. Función set-exp

```
;;set
(set-exp (id exp)
  (begin
    (setref!
      (apply-env-ref amb id)
      (evaluar-expresion exp amb))
    1))
```

El propósito de esta función es asignarle/modificar el valor a la variable por un nuevo valor, lo actualiza de cierta manera. Es útil porque nos permite modificar el valor en cualquier momento.




#### 4. Ejecución y resultados

```
-->empty  
( )
```

```
-->cons(1 cons(2 cons(3 empty)))  
(1 2 3)
```

```
-->cons(let x = 3 in x cons(3 empty))  
(3 3)
```

```
-->cons(1 3)  
 Error: 3 "no es una lista"
```

```
-->cons(  
  let x = 3 y = 2  
  in  
  y  
  cons(x cons(y empty))  
)  
(2 4 2)
```

```
-->let lst = cons(1 cons(2 cons(3 empty))) in  
  length (lst)  
3
```

```
-->begin set x = 5; set y = 10; +(x,y) end  
15
```

---

1 bound occurrence

```
-->let f = proc (x)
  cond
    (x 0) ==> 0
    (x 1) ==> 1
    else ==> *(x, (f(-(x,1))))
  end
in
  2
2
-->let x = cons(1 cons(2 cons(3 empty))) in
  first(x)
1
```