

# Programa 4.1 funciones en lenguaje de programación funcional

## Teoría de la computación

Royce Richmond Ramirez Morales

rramirezm2021@cic.ipn.mx

### Objetivo

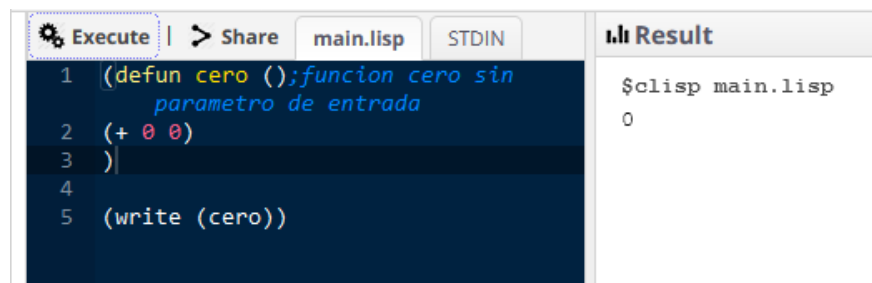
Implementar funciones en lenguaje de programación funcional de las operaciones suma, multiplicación, exponente, predecesor, monus, igualdad, cociente, impar, fibonacci, factorial, división y función de Ackerman

#### Solución:

Primero se definen la funciones elementales, la función cero, sucesor y proyección, en el bloque de código **1** se encuentra la definiciones de estas funciones, en la Imagen **1**, **2** y **3** se muestran las pruebas de ejecución de estas funciones.

```
1 (defun cero ());funcion cero sin parametro de entrada
2 (+ 0 0)
3 )
4
5 (defun suce(a);funcion sucesor
6 (+ a 1)
7 )
8
9 (defun proj (m n);funcion proyeccion
10 (nth (- m 1) n)
11 )
```

**Bloque de código 1:** Definición de las funciones elementales en Lisp.



**Imagen 1:** función cero

The screenshot shows a web-based Lisp interpreter with a dark-themed editor. The editor contains five lines of code: a function definition for 'suce' that takes a parameter 'a' and returns 'a + 1', followed by a call to 'write' to print the result of 'suce 2'. The interface includes tabs for 'main.lisp' and 'STDIN', and a 'Result' panel on the right showing the command '\$clisp main.lisp' and the output '3'.

```
1 (defun suce(a);funcion sucesor
2   (+ a 1)
3 )
4
5 (write (suce 2))
```

Result: \$clisp main.lisp  
3

Imagen 2: función sucesor

The screenshot shows the same web-based Lisp interpreter. The editor contains five lines of code: a function definition for 'proj' that takes parameters 'm' and 'n' and returns the 'nth' element of a list starting from 'm' in a list of length 'n', followed by a call to 'write' to print the result of 'proj 2' applied to the list '(4 3 5 5)'. The 'Result' panel shows the command '\$clisp main.lisp' and the output '3'.

```
1 (defun proj (m n)
2   (nth (- m 1) n)
3 )
4
5 (write (proj 2 (list 4 3 5 5)))
```

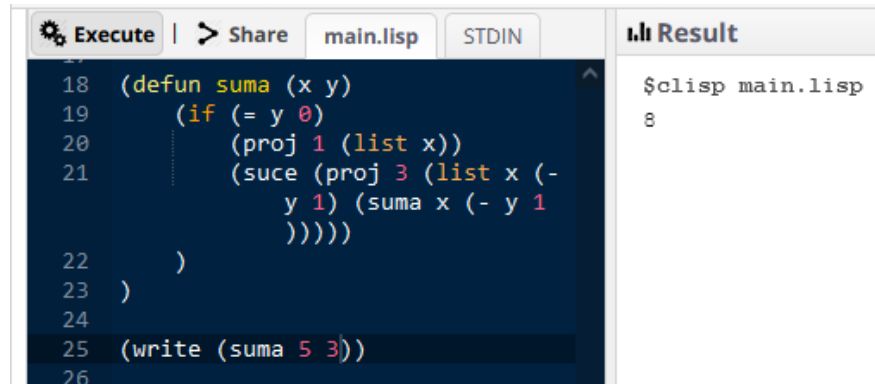
Result: \$clisp main.lisp  
3

Imagen 3: proyección

En el bloque de código 2 se encuentra la definición de la operación suma, multiplicación y exponente, en la Imagen 4, 5 y 6 se muestran las pruebas de ejecución de estas funciones.

```
1 (defun suma (x y)
2   (if (= y 0)
3       (proj 1 (list x))
4       (suce (proj 3 (list x (- y 1) (suma x (- y 1))))))
5 )
6 )
7 (defun mult (x y)
8   (if (= y 0)
9       (proj 1 (list 0))
10      (suma x (mult x (- y 1))))
11 )
12 )
13 (defun expo (x y)
14   (if (= y 0)
15       (proj 1 (list 1))
16       (mult x (expo x (- y 1))))
17 )
18 )
```

Bloque de código 2: Definición de las funciones suma , multiplicación y exponente en Lisp.

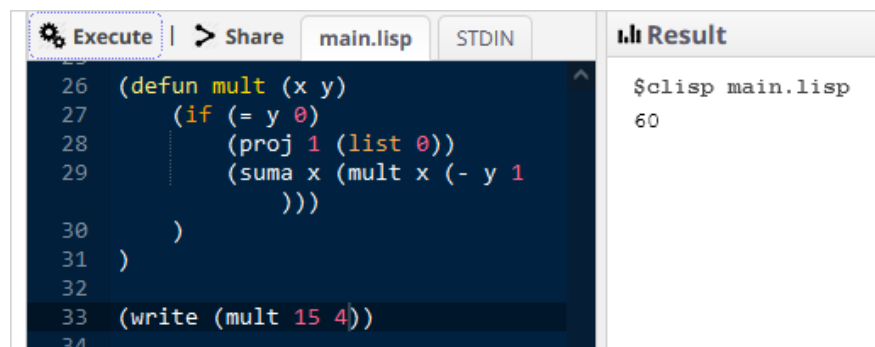


```
17
18 (defun suma (x y)
19   (if (= y 0)
20       (proj 1 (list x))
21       (suce (proj 3 (list x (-
22               y 1) (suma x (- y 1
23               )))))
24   )
25 )
26 (write (suma 5 3))
```

Result

```
$clisp main.lisp
8
```

Imagen 4: función suma

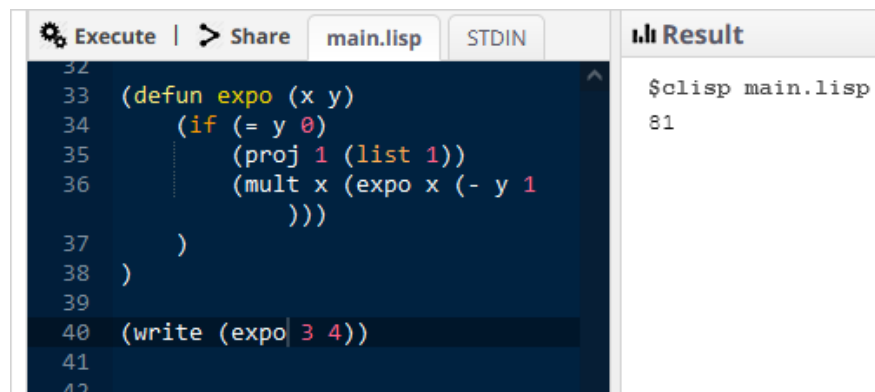


```
26 (defun mult (x y)
27   (if (= y 0)
28       (proj 1 (list 0))
29       (suma x (mult x (- y 1
30               )))
31   )
32 )
33 (write (mult 15 4))
34
```

Result

```
$clisp main.lisp
60
```

Imagen 5: función multiplicación



```
32
33 (defun expo (x y)
34   (if (= y 0)
35       (proj 1 (list 1))
36       (mult x (expo x (- y 1
37               )))
38   )
39 )
40 (write (expo 3 4))
41
42
```

Result

```
$clisp main.lisp
81
```

Imagen 6: función exponente

En el bloque de código 3 se encuentra la definición de la operación predecesor, monus (resta con tope a cero) e igualdad, en la Imagen 7, 8, 9 y 10 se muestran las pruebas de ejecución de estas funciones.

```
1 (defun pred (y)
2   (if (= y 0)
3       (cero )
```

```

4      (proj 1 (list (- y 1) (pred (- y 1))))
5    )
6  )
7
8  (defun monus (x y)
9    (if (= y 0)
10      (proj 1 (list x))
11      (pred (monus x (- y 1))))
12  )
13 )
14
15 (defun equ (x y)
16   (monus 1 (suma (monus y x) (monus x y)))
17 )

```

**Bloque de código 3:** Definición de las funciones predecesor, monus e igualdad en Lisp.

The screenshot shows a web-based Lisp interpreter with tabs for 'Execute', 'Share', 'main.lisp', and 'STDIN'. The code in the editor is as follows:

```

39
40 (defun pred (y)
41   (if (= y 0)
42       (cero )
43       (proj 1 (list (- y 1)
44                     (pred (- y 1)))))
45 )
46
47 (write (pred 45))
48

```

The 'Result' panel on the right shows the command '\$clisp main.lisp' and the output '44'.

**Imagen 7:** función predecesor

The screenshot shows the same web-based Lisp interpreter. The code in the editor is as follows:

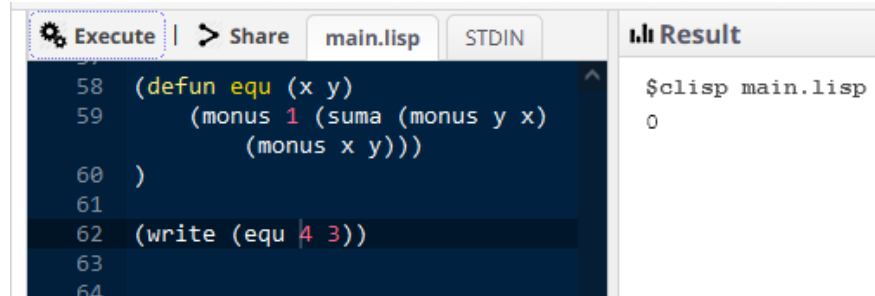
```

49 (defun monus (x y)
50   (if (= y 0)
51       (proj 1 (list x))
52       (pred (monus x (- y 1)
53               )))
54 )
55
56 (write (monus 14 3))
57

```

The 'Result' panel on the right shows the command '\$clisp main.lisp' and the output '11'.

**Imagen 8:** función monus

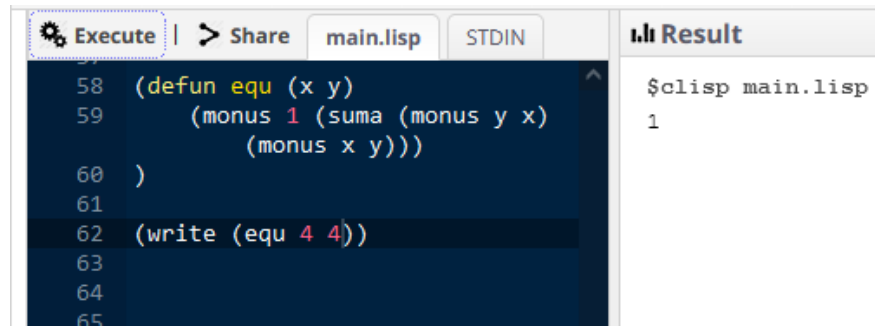


The screenshot shows a code editor with a dark background. The code is as follows:

```
58 (defun equ (x y)
59     (monus 1 (suma (monus y x)
60                     (monus x y)))
61 )
62 (write (equ 4 3))
63
64
```

The 'Execute' button is highlighted with a dashed blue border. The 'Result' panel on the right shows the command '\$clisp main.lisp' and the output '0'.

Imagen 9: función igualdad resultado cero



The screenshot shows a code editor with a dark background. The code is as follows:

```
58 (defun equ (x y)
59     (monus 1 (suma (monus y x)
60                     (monus x y)))
61 )
62 (write (equ 4 4))
63
64
65
```

The 'Execute' button is highlighted with a dashed blue border. The 'Result' panel on the right shows the command '\$clisp main.lisp' and the output '1'.

Imagen 10: función igualdad resultado uno

En el bloque de código 4 se encuentra la definición de la operación cociente, fibonacci e impar, en la Imagen 11, 12, 13 y 14 se muestran las pruebas de ejecución de estas funciones.

```
1 (defun coci (x y)
2     (if (or (zerop y) (zerop x))
3         (proj 1 (list 0))
4         (suma (coci (- x 1) y) (equ x (suma y (mult (coci (- x 1) y) y))))
5     )
6 )
7
8
9 (defun fib (x)
10     (if (= x 1)
11         (+ 0 0)
12         (if (= x 2)
13             (+ 1 0)
14             (if (> x 2)
15                 (suma (fib (- x 1)) (fib (- x 2)))
16             )
17         )
18     )
19 )
```

```

20
21 (defun impa (x)
22   (if (= x 0)
23       (proj 1 (list 0))
24       (monus x (mult 2 (coci x 2))))
25   )
26 )

```

**Bloque de código 4:** Definición de las funciones cociente, fibonacci e impar en Lisp.

The screenshot shows a Lisp IDE with a dark theme. The editor displays the definition of the 'coci' function. The 'Execute' button is highlighted. The 'Result' panel on the right shows the output of the execution.

```

62
63 (defun coci (x y)
64   (if (or (zerop y) (zerop x)
65         (proj 1 (list 0))
66         (suma (coci (- x 1) y)
67               (equ x (suma y
68                     (mult (coci (- x 1)
69                               y) y))))))
67   )
68 )
69
70 (write (coci 7 2))
71

```

Result: \$clisp main.lisp  
3

**Imagen 11:** función cociente

The screenshot shows a Lisp IDE with a dark theme. The editor displays the definition of the 'fib' function. The 'Execute' button is highlighted. The 'Result' panel on the right shows the output of the execution.

```

72 (defun fib (x)
73   (if (= x 1)
74       (+ 0 0)
75       (if (= x 2)
76           (+ 1 0)
77           (if (> x 2)
78               (suma (fib (- x
79                           1)) (fib (- x 2
80                                   )))
81               )
82           )
83   )
84 (write (fib 12))
85

```

Result: \$clisp main.lisp  
89

**Imagen 12:** función fibonacci retornando el numero 12 de la serie, siendo el 89

```
84
85
86 (defun impa (x)
87   (if (= x 0)
88       (proj 1 (list 0))
89       (monus x (mult 2 (coci
90                     x 2)))
91   )
92 )
93 (write (impa 5))
94
```

\$clisp main.lisp  
1

Imagen 13: función impar con resultado uno

```
84
85
86 (defun impa (x)
87   (if (= x 0)
88       (proj 1 (list 0))
89       (monus x (mult 2 (coci
90                     x 2)))
91   )
92 )
93 (write (impa 4))
94
```

Execute | > Share | main.lisp | STDIN | Result

\$clisp main.lisp  
0

Imagen 14: función impar con resultado cero

En el bloque de código 5 se encuentra la definición de la operación factorial, división (empleando minimización) y función de Ackermann, en la Imagen 15, 16, 17 y 18 se muestran las pruebas de ejecución de estas funciones.

```
1 (defun fact (x)
2   (if (= x 0)
3       (proj 1 (list 1))
4       (mult x (fact (- x 1)))
5   )
6 )
7
8 (defun div (x y)
9   (setq g 10)
10  (setq a 0)
11  (loop
12    (setq g (monus (suma x 1) (suma (mult a y) y) ))
13    (when (= g 0) (return a))
14    (setq a (suce a))
15  )
16 )
```

```

17
18 (defun A (x y)
19   (if (= x 0)
20     (+ y 1)
21     (if (and (> x 0) (zerop y))
22       (A (- x 1) 1)
23       (if (and (> x 0) (> y 0))
24         (A (- x 1) (A x (- y 1)))
25       )
26   )
27 )
28
29 )

```

**Bloque de código 5:** Definición de las funciones factorial, división y función de Ackermann en Lisp.

The screenshot shows a web-based Lisp interpreter with tabs for 'Execute', 'Share', 'main.lisp', and 'STDIN'. The code in the editor is as follows:

```

94 (defun fact (x)
95   (if (= x 0)
96     (proj 1 (list 1))
97     (mult x (fact (- x 1)))
98   )
99 )
100
101 (write (fact 4))
102

```

The 'Result' panel on the right shows the command '\$clisp main.lisp' and the output '24'.

**Imagen 15:** función factorial

The screenshot shows the same web-based Lisp interpreter. The code in the editor is as follows:

```

103 (defun div (x y)
104   (setq g 10)
105   (setq a 0)
106   (loop
107     (setq g (monus (suma
108                     x 1) (suma
109                     (mult a y) y )
110               ))
111     (when (= g 0)
112       (return a))
113     (setq a (suce a))
114   )
115 )
116
117 (write (div 4 5))
118

```

The 'Result' panel on the right shows the command '\$clisp main.lisp' and the output '0'.

**Imagen 16:** función división definida



```
103 (defun div (x y)
104     (setq g 10)
105     (setq a 0)
106     (loop
107         (setq g (monus (suma
108                         x 1) (suma
109                             (mult a y) y )
110                     ))
111         (when (= g 0)
112             (return a))
113         (setq a (suce a))
114     )
115 )
116 (write (div 4 0))
```

Result

```
$clisp main.lisp
Signal 15 while exiting on a signal; cleanup may be incomplete
```

Imagen 17: función división no definida

```
115 (defun A (x y)
116     (if (= x 0)
117         (+ y 1)
118         (if (and (> x 0) (zerop
119                     y))
120             (A (- x 1) 1)
121             (if (and (> x 0) (>
122                             y 0))
123                 (A (- x 1) (A x
124                             (- y 1)))
125                 )
126             )
127     )
128 )
129 (write (A 3 3))
```

Result

```
$clisp main.lisp
61
```

Imagen 18: función de Ackermann

La división no definida se encuentra cuando el divisor es cero, en este caso se cicla de manera infinita dando una operación no definida (ya que no se aproxima a ningún valor)