



UNIVERSIDAD DEL VALLE

FACULTAD DE INGENIERÍA

Proyecto final - Informe

AUTORES:

CEBALLOS YATE, Miguel (2259619)

GARCÍA CASTAÑEDA, Alex (2259517)

GÓMEZ AGUDELO, Sebastián (2259474)

HENAO ARICAPA, Stiven (2259603)

DOCENTE:

SAAVEDRA DELGADO, Carlos Andrés

CURSO:

Fundamentos de Lenguajes Programación

Semestre: Quinto - V

Tuluá, Valle del Cauca

Junio de 2024

2. ¡Señor profesor, he aquí tu solución!

Nos complace informar que hemos culminado con éxito el desarrollo del nuevo lenguaje de programación solicitado por la Universidad del Valle, seccional Tuluá. Este lenguaje ha sido diseñado específicamente para cumplir con los requerimientos del **super curso Meta programación modalidad bootcamp**, integrando fundamentos de programación imperativa, orientada a objetos, orientada a eventos, funcional, fundamentos de lenguajes de programación e infraestructuras paralelas. Utilizando el lenguaje Racket como base, hemos creado una herramienta robusta y versátil que permite a los estudiantes alcanzar más de 20 resultados de aprendizaje en un semestre, resolviendo así el problema de la disponibilidad de salas y asegurando una formación integral y avanzada para los futuros profesionales en tecnología y sistemas.

2.1 Especificación del lenguaje

1. Números:

```
(numero-exp (digitoDecimal) decimal-num)
(numero-exp (digitoBinario) bin-num)
(numero-exp (digitoOctal) octal-num)
(numero-exp (digitoHexadecimal) hex-num)
(numero-exp (flotante) flotante-num)
```

2. Texto:

```
(expresion ("\" identificador (arbo identificador) "\") cadena-exp)
```

3. Listas:

```
(expresion ("list" "(" (separated-list expresion "," ")") lista-exp)
(expresion ("cons" "(" expresion expresion ")") cons-exp)
(expresion ("empty") empty-list-exp)
(expresion (primitivaListas "(" expresion ")") prim-list-exp)
(primitivaListas ("first") first-primList)
(primitivaListas ("rest") rest-primList)
(primitivaListas ("empty?") empty-primList)
```

4. Void:

```
'void-exp
```

2.4 Estructuras léxicas

■ Identificadores:

(identificador (letter (arbno (or letter digit "?")))) symbol)

■ Comentarios:

(comment ("% (arbno (not #\newline))) skip)

■ Números enteros:

■ Dígitos binarios:

(digitoBinario ("b" (or "0" "1") (arbno (or "0" "1")))) string)

(digitoBinario ("- "b" (or "0" "1") (arbno (or "0" "1")))) string)

■ Dígitos decimales:

(digitoDecimal (digit (arbno digit)) number)

(digitoDecimal ("- "digit (arbno digit)) number)

■ Dígitos octales:

(digitoOctal ("0x" (or "0" "1" "2" "3" "4" "5" "6" "7"))(arbno (or "0" "1" "2" "3" "4" "5" "6" "7"))) string)

(digitoOctal ("- "0x" (or "0" "1" "2" "3" "4" "5" "6" "7") (arbno (or "0" "1" "2" "3" "4" "5" "6" "7"))) string)

■ Números hexadecimales:

(digitoHexadecimal ("hx" (or "0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "A" "B" "C" "D" "E" "F") (arbno (or "0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "A" "B" "C" "D" "E" "F")))) string)

(digitoHexadecimal ("- "hx" (or "0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "A" "B" "C" "D" "E" "F") (arbno (or "0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "A" "B" "C" "D" "E" "F")))) string)

■ Números Flotantes:

(flotante(digit (arbno digit) "." digit (arbno digit)) number)

(flotante ("- "digit (arbno digit) "." digit (arbno digit)) number)

■ Cadenas:

(expresion ("\" identificador (arbno identificador) "\") cadena-exp)

2.5 Datos

■ Binarios

```
> (scan&parse "b10101010")  
#(struct:a-program () #(struct:num-exp #(struct:bin-num "b10101010")))
```

```
> (scan&parse "-b01010101")  
#(struct:a-program () #(struct:num-exp #(struct:bin-num "-b01010101")))
```

■ Decimales

```
> (scan&parse "23213")  
#(struct:a-program () #(struct:num-exp #(struct:decimal-num 23213)))
```

```
> (scan&parse "-12312 ")  
#(struct:a-program () #(struct:num-exp #(struct:decimal-num -12312)))
```

■ Octales

```
> (scan&parse "0x213345")  
#(struct:a-program () #(struct:num-exp #(struct:octal-num "0x213345")))
```

```
> (scan&parse "-0x23123 ")  
#(struct:a-program () #(struct:num-exp #(struct:octal-num "-0x23123")))
```

■ Hexadecimales

```
> (scan&parse "hxFAB123")  
#(struct:a-program () #(struct:num-exp #(struct:hex-num "hxFAB123")))
```

```
> (scan&parse "-hx99EA")  
#(struct:a-program () #(struct:num-exp #(struct:hex-num "-hx99EA")))
```

■ Flotantes

```
> (scan&parse "412312.2312")  
#(struct:a-program () #(struct:num-exp #(struct:flotante-num 412312.2312)))
```

```
> (scan&parse "-23123.2312")  
#(struct:a-program () #(struct:num-exp #(struct:flotante-num -23123.2312)))
```

■ Booleanos

```
> (scan&parse "true")  
#(struct:a-program () #(struct:true-exp))
```

```
> (scan&parse "false")  
#(struct:a-program () #(struct:false-exp))
```

■ Cadenas de texto

```
> (scan&parse "\"hola mundo\"")  
#(struct:a-program () #(struct:cadena-exp hola (mundo)))
```

```
> (scan&parse "\"hola que tal\"")  
#(struct:a-program () #(struct:cadena-exp hola (que tal)))
```

2.6 Primitivas numéricas

Para las primitivas numéricas primero, en la función eval-expression en el siguiente segmento de código, se evalúan los números y la primitiva a evaluar. Ahora, una vez evaluados los términos, en la función apply-primitive, se desglosan las primitivas numéricas que son posibles de aplicar en el programa.

La función parse-number permite retornar una representación de la operación realizada según la base. Ahora, se aplica una operación según su base, la cual hace las operaciones y retorna un resultado correspondiente a lo aplicado:

Funciones a considerar para el funcionamiento de estas operaciones:

- **remove-char:** Elimina todas las ocurrencias del carácter ch de la cadena str.
- **eliminar_caracter:** Toma una cadena stri y elimina todas las ocurrencias del carácter ch usando remove-char.
- **cambiar-char:** Reemplaza la primera ocurrencia del carácter ch en la cadena str con el carácter re_ch.
- **remplazar_caracter:** Toma una cadena stri y reemplaza todas las ocurrencias del carácter ch con el carácter re_ch usando cambiar-char.

Ejemplos y salidas

```
--> (b1000 + b10)
"b1010"
```

```
--> (0x77 + 0x3)
"0x102"
```

```
--> (hx100 - hx2)
"hxfe"
```

```
--> (123.2 - 1.2)
122.0
```

2.7 Primitivas booleanas

De la misma forma que el procedimiento anterior, la función eval-expression es la que evalúa la expresión y, en combinación con la función apply-primitive, maneja la evaluación de las primitivas booleanas. Posteriormente la función aplicar_segun_base se encarga de aplicar la operación específica de comparación y usando la función retorno_tal_cual retornará el tipo de resultado esperado a la operación, y para evaluar expresiones con operadores lógicos (and, or, xor, not) se usan las siguientes funciones:

- evaluar_lista
- and-func
- or-func
- xor-func
- not-func
- evaluar_booleano

Ejemplos y salidas

```
--> and ( ( 1 > 2) , false)
#f
```

```
--> or ((8 <= 9), true)
#t
```

2.8 Listas

```
(expresion (primitivaListas "(" expresion ")") prim-list-exp)
```

2.9 Primitivas de Listas

first: Primero de lista (car)

```
(primitivaListas ("first") first-primList)
```

rest: Cola de lista (cdr)

```
(primitivaListas ("rest") rest-primList)
```

empty?: True si es vacía la lista (null?)

```
(primitivaListas ("empty?") empty-primList)
```

Ejemplo y salida

```
--> let l = cons (1 cons(2 cons(3 empty))) in list(first(l), rest(l), empty? (rest (l)))
(1 (2 3) #f)
```

2.10 Arrays

```
(expresion ("array" "(" (separated-list expresion ",") ")") array-exp)
```

Ejemplo:

```
--> array(1,2,3,4,5)
#(1 2 3 4 5)
```

2.11 Primitivas de Arrays

Para evaluar las primitivas, se recibe la primitiva a aplicar y el Array para evaluar cada elemento (pasando el Array por la función eval-rands); luego, se pasa a la función primitiva-array la cual permite realizar operaciones como:

length: Tamaño de array (vector-length)

```
(primitivaArray ("length") length-primArr)
```

Ejemplo:

```
--> let t = array(1, 2, 3, 4, 5) in length (t)
5
```

index: Acceder a elemento (vector-ref)

```
(primitivaArray ("index") index-primArr)
```

Ejemplo:

```
--> let t = array (1, 2, 3, 4, 5) in index (t, 2)
3
```

slice: Acceder a sub-arreglo en el rango de posiciones dadas. (list-vector (subvector))

```
(primitivaArray ("slice") slice-primArr)
```

Ejemplo:

```
--> let k = array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) in slice(k, 2, 5)
#(3 4 5 6)
```

setlist: Reemplaza el valor del elemento ubicado en la posición brindada. (vector-set!)

```
(primitivaArray ("setlist") setlist-primArr)
```

Ejemplo:

```
--> let t = array(1, 2, 3, 4, 5) in setlist(t, 2, 10)
#(1 2 10 4 5)
```

2.12 Primitivas de cadenas

Para las primitivas de cadena se recibe la primitiva a aplicar y luego por medio de eval-rands se evalúa los elementos de la cadenas, y luego se invoca la función y por medio de cases se elige la primitiva a ejecutar desglosando la cadena recibida para ello.

2.12.1 length: Retorna el tamaño de la cadena. (string-length)

```
(primitivaCadena ("string-length") length-primCad)
```

Ejemplo:

```
--> let s = "hola mundo cruel" in string-length(s)  
16
```

2.12.2 elementAt: Retorna el elemento que se encuentra en la posición brindada.

(string(string-ref))

```
(primitivaCadena ("elementAt") index-primCad)
```

Ejemplo:

```
--> let s = "hola mundo cruel" in elementAt(s,5)  
"m"
```

2.12.3 concat: Une listas secuencialmente. (concat(string-append))

```
(primitivaCadena ("concat") concat-primCad)
```

Ejemplo:

```
--> let  
    a = " hola "  
    b = " mundo "  
    c = " cruel "  
    in concat(a, b, c)  
"holamundocruel"
```

2.13 Entorno de ligaduras y variables

Funcionamiento let:

1. **Evaluar Argumentos:** Primero, evalúa cada expresión en rands para obtener los valores que se asignarán a las variables locales.
2. **Crear Asignaciones Locales:** Luego, crea un conjunto de pares variable-valor, donde cada identificador en ids se asocia con el valor correspondiente obtenido anteriormente.
3. **Extender Entorno:** Se extiende el entorno actual env con estas nuevas asignaciones locales.
4. **Evaluar Cuerpo:** Finalmente, evalúa la expresión body dentro de este nuevo entorno extendido.

Esta función permite definir variables locales temporales y usarlas dentro de una expresión más grande, manteniendo el código organizado y modular.

let

```
(expresion ("let" (arbo identificador "=" expresion) "in" expresion) let-exp)
```

Funcionamiento var:

1. **Evaluar Argumentos:** Evalúa las expresiones en rands en el entorno actual.
2. **Crear Vector de Argumentos:** Convierte la lista de valores evaluados en un vector.
3. **Extender Entorno Modificado:** Extiende el entorno con los identificadores ids y el vector de argumentos.
4. **Evaluar Cuerpo con Nuevo Entorno:** Evalúa la expresión body en este nuevo entorno modificado.

Esta función permite trabajar con un entorno que ha sido modificado para incluir variables locales como vectores, útil para aplicaciones que requieren estructuras de datos más complejas.


var

```
(expresion ("var" (arbo identificador "=" expresion) "in" expresion) lvar-exp)
```

Ejemplos

```
--> var x = 10
in begin set x = 20; x end
20
```

```
--> let x = 10
in begin set x = 20; x end
```



variable no encontrada en un ambiente valido

```
--> var x = x in x
```

.X. Ciclo infinito

```
--> var f = func(x) if (x==0) {0 else (x + call f((x - 1)))} in call f (10)
55
```

2.14 Condicionales

La gramática define cómo se estructura una expresión if en el lenguaje.

```
(expresion ("if" expresion "{" expresion "else" expresion "}") if-exp)
```

En el eval-expresion el if-exp tiene tres componentes: test-exp (la condición), true-exp (la expresión si la condición es verdadera) y false-exp (la expresión si la condición es falsa), La expresión if-exp se descompone en sus partes: test-exp, true-exp y false-exp, la condición (test-exp) se evalúa en el entorno dado (env) utilizando eval-expresion, Si el resultado de test-exp es verdadero (#t), entonces se evalúa y devuelve true-exp.

Ejemplo y salida

```
--> if ( 4 > 3) { 1 else 2 }
1
--> if (1 == 1) { true else false }
#t
--> if and((2 > 1), true) {1 else 0}
1
```

2.15 Estructuras de control

2.15.1 For

Se define una expresión for que tiene una variable de iteración, un valor inicial, un límite, un paso y una expresión de cuerpo.

```
;;Iteradores
(expresion ("for" identificador "from" expresion "until" expresion "by" expresion "do" expresion) for-exp)
```

El for-exp contiene la variable de iteración, el valor inicial, el valor final, el paso y la expresión de cuerpo.

Se evalúa las expresiones de inicio, fin y paso, Inicia un bucle desde el valor inicial hasta el valor final incrementando por el paso. en cada iteración, extiende el entorno con la variable de iteración y evalúa el cuerpo del bucle.

Ejemplo y salida

```
--> var
  x = 0
  in
    begin
      for i from 0 until 10 by 1 do set x = ( x + i ) ;
      x
    end
45
```

2.15.2 While

Esto define una expresión while con una condición y un cuerpo.

```
(expresion ("while" expresion "{" expresion "}") while-exp)
```

El while-exp contiene una expresión de prueba y una expresión de cuerpo.

Se evalúa la condición (test-exp), si es verdadera, evalúa el cuerpo (body-exp) y repite el bucle, si es falsa, termina el bucle.

Ejemplo y salida

```
--> var
    x = 0
    in
        begin
            while ( x < 10) { set x = ( x + 1) };
            x
        end
10
```

2.15.3 Switch

Esto define una expresión Switch con una condición y un cuerpo.

```
;;Switch
(expresion ("switch" "(" expresion ")" "{" (arbno "case" expresion ":" expresion)
"default" ":" expresion "}") switch-exp)
```

En el eval-expresión el switch-exp contiene una expresión de prueba, una lista de casos y una expresión por defecto, se evalúa la expresión de prueba (test-exp), compara el valor con cada caso, si encuentra un caso coincidente, evalúa y devuelve la expresión correspondiente. si no encuentra coincidencia, evalúa y devuelve la expresión por defecto.

Ejemplo y salida

```
--> var
    x = 0
    in
        begin
            switch ( x ) {
                case 1 : 1
                case 2 : 2
                default : 3
            }
        end
3
```

2.16. Begin y set

■ **Begin:** Se define como una expresión begin que contiene una secuencia de expresiones terminadas por end.

```
(expresion ("begin" expresion (arbno ";" expresion) "end")
  begin-exp)
```

Luego, begin-exp toma una lista de expresiones (exps), La función eval-seq evalúa cada expresión en la secuencia, si es la última expresión, la evalúa y devuelve su valor, si no es la última, la evalúa y continúa con la siguiente expresión en la lista.

Ejemplo y salida

```
--> begin 1; 2; 3 end
3
```

■ **Set:** Se define una expresión set que cambia el valor de una variable a una nueva expresión, el set-exp contiene una variable (var) y una expresión de valor (val-exp).

```
(expresion ("set" identificador "=" expresion)
  set-exp)
```

set-exp toma una variable (var) y una expresión de valor (val-exp), evalúa val-exp en el entorno (env), actualiza el entorno con el nuevo valor para la variable y devuelve una expresión vacía (void-exp).

Ejemplo y salida

```
--> var
    x = 0
    in
        set x = 10
void-exp
```

2.17. Funciones

■ **Func:** Esto define una expresión `func` que toma un parámetro (una variable) y un cuerpo de expresión, el `func-exp` contiene un parámetro (`param`) y una expresión de cuerpo (`body-exp`).

```
(expresion ("func" "(" (separated-list identificador ",") ")") expresion) func-exp)
```

En `eval-expresion` `func-exp` toma un parámetro y una expresión de cuerpo.

Devuelve una clausura (closure) que incluye el parámetro, el cuerpo y el entorno actual.

■ **Call:** Esto define una expresión `call` que aplica una función a un argumento, el `call-exp` contiene una expresión de función (`func-exp`) y una expresión de argumento (`arg-exp`).

```
(expresion ("call" expresion "(" (separated-list expresion ",") ")") app-exp)
```

En el `eval-expresion` pasará lo siguiente:

- `call-exp` toma una expresión de función y una expresión de argumento.
- Evalúa la expresión de función y la de argumento.
- La función evaluada debe ser una clausura.
- Extiende el entorno de la clausura con el parámetro de la función y el valor del argumento.
- Evalúa el cuerpo de la función en el entorno extendido.

Ejemplo y salidas

```
--> var
    f = func (x) x
    in
        call f(10)

10
```

2.18. Estructuras de datos:

Estas definiciones permiten declarar una estructura, instanciarla, obtener el valor de un campo y modificar el valor de un campo.

Estos tipos de datos representan la definición de una estructura, la instanciación de una nueva estructura, la obtención de un campo y la modificación de un campo.

```
;; estructuras
(struct-decl ("struct" identificador "{" (arbno identificador) "}") struct-exp)
;;Instanciación y uso de estructuras
(expression ("new" identificador "(" (separated-list expresion ",") ")") new-struct-exp)
(expression ("get" expresion "." identificador) get-struct-exp)
(expression ("set-struct" expresion "." identificador "=" expresion) set-struct-exp)
```

En el eval-struct-def struct-def toma un nombre y una lista de campos y actualiza el entorno para incluir la nueva estructura.

En eval-expresion pasará lo siguiente:

- new-exp toma un nombre de estructura y una lista de argumentos.
- Busca los campos de la estructura en el entorno.
- Evalúa los argumentos y crea una instancia de la estructura con los campos y valores correspondientes.
- get-exp toma una expresión de estructura y un campo.
- Evalúa la expresión de la estructura y obtiene el valor del campo correspondiente.
- set-struct-exp toma una expresión de estructura, un campo y una nueva expresión de valor.
- Evalúa la expresión de la estructura y la nueva expresión de valor.
- Modifica el campo de la estructura con el nuevo valor.
- Devuelve void-exp.

Ejemplos y salidas

```
--> struct perro {nombre edad color}
let
  t = new perro ( "lucas" ,10 , "verde" )
  in get t.nombre

"lucas"
```

```
--> struct perro {nombre edad color}
let
t = new perro("lucas",10,"verde")
in
begin
set-struct t.nombre = "pepe";
get t.nombre
end

"pepe"
```

```
--> struct perro {nombre edad color}
let
t = new perro("lucas", 10, "Verde")
in
set-struct t.nombre = "Pepé"

void-exp
```

3. Reconocimiento de patrones

Esto define una expresión match que toma una expresión y una serie de patrones con sus correspondientes acciones.

El match-exp contiene una expresión a evaluar y una lista de casos (patrones y sus correspondientes acciones). Los diferentes tipos de patrones están definidos para reconocer listas, números, cadenas, booleanos, arrays y listas vacías.

```
;;Reconocimiento de patrones
(expression ("match" expresion "{" (arbno regular-exp "=>" expresion) "}") match-exp)
```

En el eval-expresion pasa lo siguiente:

- match-exp toma una expresión y una lista de casos.
- Evalúa la expresión principal.
- Itera sobre los casos, intentando hacer coincidir el valor evaluado con cada patrón.
- Si un patrón coincide, evalúa el cuerpo del caso en un entorno extendido con las variables capturadas.
- Si ningún patrón coincide, evalúa la expresión asociada a default.

Ejemplos y salidas

```
--> let
x = list ( 1 , 2 , 3 , 4 , 5 )
in
  match x {
    x::xs => xs
    default => 0
  }

(2 3 4 5)
```

```
--> let
  x = 10
  in
    match x {
      numero ( x ) => x
      default => 0
    }

10
```

```
--> let
  x = "hola mundo"
  in
    match x {
      cadena ( x ) => x
      default => 0
    }

"hola mundo"
```

```
--> let
  x = true
  in
    match x {
      boolean ( x ) => x
      default => 0
    }

#t
```

```
--> let
  x = array(1,2,3,4,5)
  in
    match x {
      array(x,y,z) => list(x,y,z)
      default =>0
    }

(1 2 (3 4 5))
```

```
--> let
  f = func(x)
    match x {
      empty => 0
      x :: xs => (x + call f(xs))
    }
  in
    call f(list(1,2,3,4,5))

15
```