PROYECTO FINAL

PEDRO BERNAL LONDOÑO - 2259548-3743

JOTA EMILIO LÓPEZ - 2259394-3743

ESMERALDA RIVAS GUZMÁN - 2259580-3743

CARLOS STIVEN RUIZ - 2259629-3743

DIRECTOR

CARLOS ANDRÉS DELGADO

UNIVERSIDAD DEL VALLE SEDE TULUÁ

FACULTAD DE INGENIERÍA

PROGRAMA ACADÉMICO DE INGENIERÍA DE SISTEMAS

FUNDAMENTOS DE LENGUAJES DE PROGRAMACIÓN

TULUÁ – VALLE DEL CAUCA

2024

Este proyecto tiene como objetivo principal crear un intérprete para un lenguaje de programación específico utilizando las capacidades de Racket y la librería SLLGEN. Estas herramientas son ideales para la generación de analizadores léxicos (scanner) y sintácticos (parser), proporcionando un entorno robusto para el desarrollo del lenguaje. El lenguaje implementado cuenta con un conjunto diverso de características, incluyendo, tipos de datos variados (números en variables bases, booleanos, cadenas, arreglos y listas), estructuras de control (if-else, while, for, switch y match), funciones, estructuras de datos (structs) y una amplia gama de operadores (aritméticos, lógicos, de relación y de asignación).

Implementaciones

1. **Datos**

Especificación de tipos de valores numéricos: binarios, decimales, octales, hexadecimales y flotantes:

La base binaria se representa utilizando los dígitos 0 y 1. El sistema decimal utiliza dígitos del 0 al 9. La base octal, con dígitos del 0 al 7, y el sistema hexadecimal, con dígitos del 0 al 9 y letras de la A a la F, es eficiente en representar valores binarios grandes. Los números flotantes, como 8,1, son cruciales para cálculos precisos.
Teniendo esto en cuenta, para manejar y convertir números entre estas bases, se implementaron funciones de conversión que facilitan la compatibilidad y la manipulación de datos. Una función llamada (to-decimal) convierte números de cualquier base a decimal, mientras que otra (from-decimal), realiza la conversión inversa, de decimal a la base especificada. Esto permite una implementación generalizada de primitivas numéricas, asegurando que las operaciones matemáticas sean precisas independientemente de la base del número.

Especificación de tipos de valores

Implementación de cadenas:

cadena-exp: Toma dos parámetros y usa *letrec* para definir una función recursiva *crear_string*. Esta función convierte una lista de identificadores en una cadena de texto, si la lista está vacía, retorna una cadena vacía, de lo contrario, concatena el primer identificador convertido a string y el resultado de llamar recursivamente a *crear_string* con el resto de la lista. Por último, concatena el primer parámetro con la cadena creada por la función aplicada al segundo parámetro.

Implementación de listas:

lista-exp: Toma una lista de expresiones y utiliza *map* para evaluar cada una usando *eval-expression*. El resultado es una lista de los valores evaluados

Implementación de arrays:

array-exp: Toma una lista de expresiones y las evalúa utilizando *map*, aplicando *eval-expression* a cada una de ellas. Luego, convierte la lista de valores evaluados en un vector con el uso de *list→vector*

Implementación del void:

void-exp: Devuelve el símbolo *void*, indicando que la operación realizada no tiene un valor de retorno

2. **Entornos de ligaduras y variables**

Implementación de let:

let-exp: Toma identificadores (ids), expresiones (rands) y un cuerpo (body), Primero evalúa las expresiones en el entorno actual (env) con *eval-rands*, luego extiende el entorno con los nuevos identificadores y sus valores correspondientes usando *extend-env*, pero antes de evaluar el cuerpo, verifica si contiene una expresión set con contains-set?. Si es así, se genera un error utilizando eopl:error, ya que no se permite la mutación de variables en las asociaciones de let.

Implementación de var:

lvar-exp: Toma identificadores (ids), expresiones (rands) y un cuerpo (body). Primero evalúa las expresiones en el entorno actual (env) con *eval-rands*, luego extiende el entorno con los nuevos identificadores y sus valores correspondientes usando *extend-env*, y finalmente evalúa el cuerpo en este nuevo entorno extendido.

3. **Estructuras de control**

Implementación de while:

While-exp: Inicia un bucle recursivo (loop) donde evalúa si la condición (cond) es falsa (usando (not cond). si se cumple retorna el valor almacenado en (last). Por otro lado, si la condición es verdadera, evalúa la expresión del cuerpo (exp) y el resultado se almacena en la variable (value). Luego llama recursivamente a loop con la nueva condición evaluada (eval-expresión cond-exp env) y el valor calculado (value). El bucle continúa hasta que la condición sea falsa. En cada iteración, se evalúa la expresión del cuerpo y se actualiza el valor almacenado en last. Cuando la condición se vuelve falsa, se retorna el último valor almacenado en (last).

Implementación de for:

For-exp: Comienza extendiendo el entorno con el valor inicial de la variable de bucle (from-exp). Luego entra en un bucle recursivo (loop) donde verifica si el valor actual de la variable de bucle es menor que el valor hasta el cual debe ejecutarse (until-val). Si es así, evalúa la expresión del cuerpo (do-exp), actualiza el entorno con el nuevo valor de la variable de bucle incrementado por by-val, y continúa la recursión. Si no, retorna el resultado acumulado.

Implementación de switch:

Switch-exp: Evalúa la expresión de condición (cond-exp) y compara su resultado con una lista de casos (lexps). Utiliza un bucle recursivo para iterar a través de los casos, evaluando la expresión correspondiente al primer caso que coincida con el resultado de la condición. Si no se encuentra ninguna coincidencia, evalúa la expresión por defecto (default-exp). Este proceso se realiza dentro de un entorno (env) que puede contener variables y funciones definidas previamente.

4. **Reconocimiento de patrones:**

La implementación de la expresión *match-exp* se basa en la evaluación de una expresión *exp* junto con un conjunto de casos (*rexps* y *lexps*) que definen cómo manejar diferentes tipos de expresiones, se utiliza la función *apply-regular-exp* para mapear las expresiones regulares (*regular-exp*) a identificadores específicos de casos *(list-match, num-match, cad-match, etc.).* Estos identificadores fueron creados para facilitar la búsqueda de coincidencia, no pertenecen a la gramática del lenguaje. cLuego, se inicia un bucle recursivo (*loop*) que evalúa la *exp* y compara su tipo con los casos disponibles, cada caso verifica si el tipo de *exp* coincide con el tipo esperado y en caso verdadero, ejecuta la expresión correspondiente utilizando *eval-expression* en un entorno extendido (*extend-env)*. Si no se encuentra ninguna coincidencia y hay un caso por defecto definido en *default-match,* se ejecuta ese caso. En ausencia de una coincidencia y sin un caso por defecto, se genera un error para manejar la falta de coincidencia.

5. **Estructuras de datos**

La implementación maneja estructuras de datos, permitiendo su declaración, instanciación, acceso y modificación. Por esta razón, la función *eval-program*, evalúa si se tienen declaraciones de estructuras, de no existir se evalúa directamente el cuerpo con el ambiente inicial. Si existen estructuras, evalúa cada una de ellas, mediante *eval-struct*, que las convierte en una lista con sus identificadores y campos, extendiendo el ambiente inicial con estas estructuras. Dentro de *eval-expression* se manejan tres operaciones, *new-struct-exp*, que crea instancias verificando que el número de atributos es el correcto, *get-struct-exp*, que recupera valores de campos específicos de una estructura y *set-struct-exp*, que actualiza los valores de campos específicos.

Funcionalidad

1. **Test**

Nuestro proyecto incluye la creación de test para verificar su funcionalidad. Estas pruebas fueron diseñadas, estructuradas y ejecutadas con éxito.

1.1 **Ejecución de pruebas**

Para ejecutar todas las pruebas:

sh test.sh all

Para ejecutar pruebas de forma unitaria:

sh test.sh number

sh test.sh arrays

sh test.sh cadena

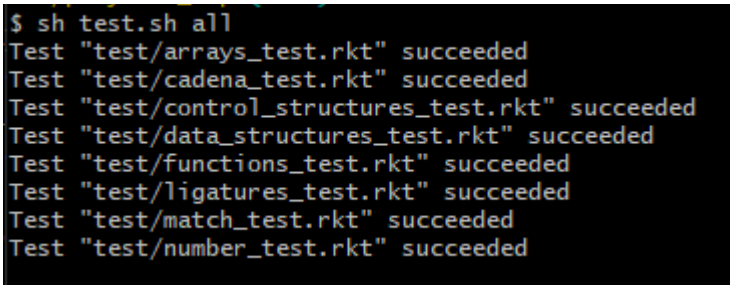
sh test.sh ligatures

sh test.sh control_structures

sh test.sh functions

sh test.sh data_structures

sh test.sh match



2. Pruebas en el interpretador

- Datos

```
--> b10101010
b10101010
--> -b01010101
-b01010101
--> 23213
23213
--> -12312
-12312
--> 0x213345
0x213345
--> -0x23123
-0x23123
--> hxFAB123
hxFAB123
--> -hx99EA
-hx99EA
--> 412312.2312
412312.2312
--> -23123.2312
-23123.2312
--> true
#t
--> false
#f
--> "hola mundo"
"hola mundo"
--> "hola que tal"
"hola que tal"
```

- **Listas**

```
--> list(1, 2, 3, 4)
(1 2 3 4)
--> cons(2 cons(2 empty))
(2 2)
--> empty
()
```

- **Arrays**

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

- **Primitivas numéricas**

```
--> (b1000 + b10)
b1010
--> (0x77 + 0x3)
0x102
--> (hx100 - hx2)
hxFE
--> (123.2 - 1.2)
122.0
```

- **Primitivas booleanas**

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

- **Primitivas de listas**

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

- **Primitivas de arrays**

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

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

- **Primitivas de cadenas**

```
--> let s = "hola mundo cruel" in string-length(s)
16
--> let s = "hola mundo cruel" in elementAt(s, 5)
"m"
--> let a = "hola" b = "mundo" c = "cruel" in concat(a,b,c)
"holamundocruel"
```

- **Entornos de ligaduras y variables**

```
--> var x = 10
                in begin set x = 20; x end
20
--> var f = func(x) if (x == 0) { 0 else (x + call f((x - 1))) }
                in call f(10)
55
```

```
--> let x = 10
 in begin set x = 20; x end
decl-exp: Cannot use 'set' in 'let' bindings
```

- **Condicionales**

```
--> let a = 5 in if (a > 3) {1 else 2}
1
--> let a = 2 in if (a > 3) {1 else 2}
2
```

- **Estructuras de control**

For

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

While

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

Switch

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

- **Begin y set**

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

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

- **Funciones**

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

```
--> var
x = 10 f = func(a) (a+x)
in
let x = 30 in call f(10)
40
```

- **Estructuras de datos**

```
--> 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 = "pepe"
void
```

- Reconocimiento de patrones (match)

```
--> 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)
```

```
--> let
f = func(x)
match x {
empty => 0
x::xs => (x + call f(xs))
}
in
call f(list(1,2,3,4,5))
15
```