PROYECTO FINAL - FUNDAMENTOS DE LENGUAJES DE PROGRAMACIÓN

EL TEMIBLE META-LENGUAJE

Jhon Alejandro Martinez - 2259565

Juan Miguel Posso - 2259610

Esteban Alexander Revelo - 2067507

Víctor Manuel Hernández - 2259520

I. ESPECIFICACIÓN DEL LENGUAJE

A. *Especificación de tipos de valores numéricos*

Todas las representaciones de los números fueron basadas en la gramática del proyecto. Más adelante explicaré cómo implementamos la manera en que se operan las diferentes representaciones entre ellos.

- **Números Binarios:** Inician con 'b' y están conformados por 0 y 1.
  Ejemplo: b1010 representa el número binario 1010.
- **Números Decimales:** Inician con cualquier dígito del 0 al 9 y pueden incluir dígitos decimales.
  Ejemplo: 123 representa el número decimal 123
- **Números Octales:** Inician con '0x' y están conformados por dígitos del 0 al 7.
  Ejemplo: ox755 representa el número octal 755.
- **Números Hexadecimales:** Inician con 'hx' y están conformados por dígitos del 0 al 9 y letras de la A a la F (en mayúscula o minúscula).
  Ejemplo: hx1A3F representa el número hexadecimal 1A3F.
- **Números Flotantes:** Los números decimales pueden tener una parte fraccionaria separada por un punto ('.').
  Ejemplo: 123.456 representa el número flotante 123.456.

B. *Especificación de otros tipos de valores*

- **Texto**
  Cadena de Texto: Representadas por caracteres entre comillas dobles ("..."). Ejemplo: "HelloWorld".
- **Listas**
  - **Listas:** Definidas con list y elementos separados por comas entre paréntesis.
    Ejemplo: list(1, 2, 3).
  - **Cons:** Agrega un elemento al frente de una lista.
    Ejemplo: cons(1, list(2, 3)).
  - **Lista Vacia:** Representada por empty.
    Ejemplo: empty
- **Arreglos**
  - **Arreglos:** Definidos con array y elementos separados por comas entre paréntesis.
    Ejemplo: array(1, 2, 3).
- **Void (Expresión Vacía)**
  - **Void:** Representa una expresión sin valor, usando void.
    Ejemplo: void

C. *Preparación de los Números*

Para ejecutar las diferentes operaciones numéricas, primero aseguramos la correcta creación de **eval-numero-exp**, que utiliza la función **convertir-string** para convertir representaciones numéricas especiales de cadenas a números o símbolos correspondientes. Además, usamos la función **operando-numeros**, que recibe una operación, dos expresiones y un dato booleano. Esta función define si la operación retorna una representación especial del número o si puede resolverse utilizando las primitivas previamente creadas.

D. *Operaciones entre Números*

Función que se explicara:

```
(define operando-numeros
  (lambda (op num1 num2 boolean?)
    (let ([base (extraer-base (to-string num1)
    )]
          [num1 (convertir-decimal num1 (
    extraer-base (to-string num1)))]
          [num2 (convertir-decimal num2 (
    extraer-base (to-string num2)))]
          )
      (if boolean?
          (op num1 num2)
          (crear-representacion (op num1 num2)
     base)
        )
    )
  )
)
```

Para gestionar las operaciones entre números, creamos un nuevo archivo utilizando ***#lang racket***. La función principal es **operando-numeros**, que recibe los parámetros mencionados y opera entre ellos. Primero, extrae la base del número con la función **extraer-base**, que identifica el prefijo en la representación del número y devuelve la letra correspondiente. Luego, **operando-numeros** usa **convertir-decimal**, que convierte el número a su forma decimal basada

en su base identificada (por ejemplo, 'b' para binario, que se convierte a base 2) utilizando la función **string-¿number**. Para devolver el resultado a su representación inicial, empleamos la función **crear-representacion**, que recibe el número y su base. Esta función agrega el prefijo apropiado al resultado utilizando la función **number-¿string** con la base correspondiente. En resumen, nuestro proceso convierte números de su representación especial a decimal, realiza las operaciones necesarias y luego los convierte de nuevo a su representación original.

Algunas pruebas de operaciones entre estos números, para ver su representación:

1)
```
my-language: --> (b101 + b111)
b1100
```

2)
```
my-language: --> (hxFA31 + hxB12D)
hx1AB5F
```

3)
```
my-language: --> (0x12312 - 0x12321)
-0x7
```

II. Primitivas del lenguaje

En la gramática, se han definido distintas de primitivas para los diferentes tipos de operaciones Tenemos los siguientes tipos

- Primitivas numéricas: Operaciones aritméticas y comparativas.
- Primitivas booleanas: Operaciones lógicas.
- Primitivas de listas: Operaciones sobre listas.
- Primitivas de arrays: Operaciones sobre arrays.
- Primitivas de cadenas: Operaciones sobre cadenas de texto.

Estas primitivas se utilizarán en el evaluador (**eval-expresion**) para realizar las operaciones correspondientes en las expresiones. Cuando se encuentra un **case** correspondiente a una de estas primitivas, se procede a evaluarla mediante una función concreta para cada tipo de primitiva.

A. *Funcionamiento de primitivas numéricas*

En la implementación del código para la evaluación de las primitivas numéricas se recibe el tipo de primitiva y dos expresiones que serán los operandos, posteriormente se evalúan los casos a los que corresponde esta primitiva y se aplica la operación, retornando el resultado.

6 ejemplos de implementación y funcionamiento correcto de primitivas numéricas:

1)
```
my-language: --> (1 + 2)
3
```

2)
```
my-language: --> (1.2 - 0.7)
0.5
```

3)
```
my-language: --> (0x123 * 0x123)
0x15351
```

4)
```
my-language: --> (hxAB3 / hx1)
hxAB3
```

5)
```
my-language: --> (2 pow 5)
32
```

6)
```
my-language: --> (13 mod 3)
1
```

B. *Funcionamiento de primitivas booleanas*

En los tipos de primitivas booleanas contamos con todas las operaciones lógicas, **and, or, xor y not**.La función de evaluación de estas primitivas representa lo que se hace en cada caso deprendiendo de el tipo de operación lógica. Cada una de estas hace uso de distintas funciones auxiliares que se enfocan en detectar casos específicos para determinar el resultado lógico de cada operación.

```
(define (trueExclusivo? lst acc)
  (cond
    [(null? lst) (if (= acc 1) #t #f)]
    [(> acc 1) #f]
    [(eqv? (car lst) #t) (trueExclusivo? (cdr
   lst) (+ acc 1))]
    [else (trueExclusivo? (cdr lst) acc)]
    )
  )
```

La anterior función verifica mediante un acumulador la cantidad de expresiones verdaderas, en caso de estas superar la cantidad de 1, retorna falso, de lo contrario quiere decir que solo existe un verdadero y retorna verdadero.

```
(define (unTrue? lst)
  (cond
    [(null? lst) #f]
    [(eqv? (car lst) #t) #t]
    [else (unTrue? (cdr lst))]))
```

La anterior función sirve para el operador **or**, ya que evalúa si en la expresión existe al menos un true, siendo esta la condición principal del operador **or** para devolver verdadero.

```
(define (todos-iguales? lst)
  (cond
    [(null? lst) #t]
    [(null? (cdr lst)) #t]
    [else
     (let loop ([primero (car lst)] [resto (
   cdr lst)])
       (if (null? resto)
           #t
           (if (eqv? primero (car resto)) (
   loop primero (cdr resto))
               #f))))]))
```

Con la anterior función auxiliar para el operador logico **and**, se evalúa si en la expresión existen elementos distintos, en

cuyo caso, la operación retorna falso, pero en caso contrario, siendo todos los elementos iguales, retorna verdadero.

4 ejemplos de implementación y funcionamiento correcto de las primitivas boolenas:

1)
```
my-language: --> and(true, true, false)
#f
```

2)
```
my-language: --> or(true, false)
#t
```

3)
```
my-language: --> xor(true, true)
#f
```

4)
```
my-language: --> not(false)
(#t)
```

C. *Funcionamiento de primitivas de listas*

Se cuenta con la función **aplicar-primitiva-listas** la cual se encarga de evaluar los **cases** para las operaciones primitivas dirigidas hacia listas. Estas operaciones en cuestión no requieren de auxiliares ya que interactuan de forma directa con la lista.

- **first-primList** saca el primer elemento de una lista, es decir, interpretado en racket como **car**
- **rest-primList** saca el resto de la lista, interpretado en racket como **cdr**
- **empty-primList** verifica si una lista se encuentra vacia haciendo uso de la función **null?**

3 ejemplos de implementación y funcionamiento correcto de las primitivas de listas:

1)
```
my-language: --> first(list(5,1,3,4))
5
```

2)
```
my-language: --> rest(list(5,1,4,1))
(1 4 1)
```

3)
```
my-language: --> empty?(list(1))
#f
```

D. *Funcionamiento de primitivas de arrays*

Internamente los arrays fueron representados con la estructura **vector** que posee racket, gracias a esto, las operaciones primitivas se limitan a trabajar con las funciones ya establecidas con las que cuenta la estructura **vector** Tenemos las siguientes primitivas:

- **length-primArr** extrae el tamaño de un array
- **index-primArr** se extrae el elemento que se indique en el indice, con indice desde 0
- **slice-primArr** porciona el array entre dos valores que se indiquen, incluyendo los elementos que se encuentran en estas posiciones
- **setlist-primArr** cambia el valor de un elemento del array

4 ejemplos de implementación y funcionamiento de las primitivas de los arrays:

1)
```
my-language: --> length(array(4,8))
2
```

2)
```
my-language: --> let a = array(1,2,3)
in index(a,1)
2
```

3)
```
my-language: --> let
a = array(4,1,8,4,9)
in slice(a, 1,3)
#(1 8 4)
```

4)
```
my-language: --> let
a = array(4,1,6,3)
in setlist(a,1,10)
#(4 10 6 3)
```

E. *Funcionamiento de primitivas de cadenas*

Las cadenas de texto cuentan con tres operaciones primitivas que trabajan sobre ellas.

- **concat-primCad** concatena varias cadenas de texto
- **length-primCad** halla la longitud de una cadena de texto
- **index-primCad** busca el elemento en un indice desde 0

3 ejemplos de implementación y funcionamiento de primitivas de cadenas:

1)
```
my-language: --> let
a = "hola que tal"
in string-length(a)
12
```

2)
```
my-language: --> let
a = "hola que tal"
in elementAt(a,5)
"q"
```

3)
```
my-language: --> let
a = "hello"
b = "how ya doin"
in concat(a,b)
"hellohow ya doin"
```

III. Estructuras de control

A. *Funcionamiento de la estructura if*

1)
```
my-language: --> var x=5
   in
    if (x <= 5) {((x*8)/4) else (x+1) }
10
```

El if se compone de tres argumentos claves, la condición, el then que es lo que pasaría si dicha condición es verdadera y los elses que son los que nos dicen que pasa si las condiciones son falsas, en el intérprete, el if se comprueba con eval-expresion dos cosas, la condición y el ambiente que nos dice cual es el valor actual de la variable en el contexto de ejecución y se retorna un valor booleano, Si la evaluación de condición resulta en **t** (verdadero), entonces se evalúa la expresión then en el mismo entorno env y si no entonces evalua la expresión else en el mismo entorno.

B. funcionamiento de la estructura switch

El switch funciona mediante cinco entradas las cuales son un item, unos cases, unos valores de cases, el valor por defecto y un entorno de evaluación, los cuales la función eval-switch-exp necesita y con ayuda de un cond se evalúa si un caso coincide con el item enviado y procede a retornar el valor que este en el case, en caso de que el item no coincida procede a recursivamente evaluarse el ítem con todos los cases y si con ninguno de los cases se cumple la igualdad este retornara el valor que se le dé por defecto.

1)
```
my-language: --> var x=2 in switch (x){
   case 1: ((x+9)*5)
   case 2: (5 +((x+6)/2))
   default: "no esta dentro de los cases"
}
9
```

C. funcionamiento de la estructura for

1)
```
my-language: --> var x = 0
in begin for i from 0 until 10 by 1
do set x = (x + i) ; x end
 55
```

En la implementacion del for se tuvo en cuenta sus entradas las cuales son, el iterador, un from que marca desde donde inicia, un until que marca un hasta cuando, un by que nos marcara el paso en el cual se va a avanzar, un body que es lo que se cumplira dentro de ese ciclo y el ambiente actual. En el con vamos a implementar si el iterador es menor o igual que el valor de until en ese ambiente, si esto es correcto entonces en el begin nos vamos al body donde se evaluara el body con el ambiente actual, después le aplicamos al iterador la suma de el iterador en ese ambiente mas lo que haya en el ambiente del by que marca el paso, después se llama la función recursivamente con los nuevos valores de ese ambiente, finalmente el ciclo se rompe cuando el iterador no sea menor que el until que marca hasta cuando se ejecuta y retorna los valores que tenga en el ambiente al final.

D. funcionamiento de la estructura while

Primero se implementa su especificación gramática, la cual nos mostrara como se debe de construir el while, luego en el interpretador se hace la evaluación del while, mandándole la función apply-while-exp que evalúa una condición en un entorno y mientras que esa condición se cumpla se evalúa el cuerpo en el mismo entorno, es una función recursiva que se llama así misma hasta que la condición sea falsa y retorna.

1)
```
my-language: --> var x=5 in begin
   while (x<15){set x= (x+3)};
   x
end
17
```

En este caso tenemos que x empieza en 5 y entra a evaluarse en el while, entonces mientras que 5 sea menor que 15 se le asignara a X el valor actual mas 3 y se llamara otra vez hasta que se rompa la condición. En este caso 5+3 =8, luego 8+3=11, 11+3=14 y finalmente 14+3 = 17, entonces 17¡15 = false por ende se rompe el while y devuelve x

IV. Funciones y estructuras de datos

Explicacion de representacipon de estrcuturas de datos

A. Creación de funciones

Para lograr este punto , manejamos las declaraciones de funciones como procedimientos por eso se definio este datatype

```
(define-datatype procval procval?
  (closure
   (ids (list-of symbol?))
   (body expresion?)
   (env environment?)))
```

Donde recibimos tres parametros para lograr crear la instancia de un closure para usarlo mas adelante en el llamado de funciones

B. Llamado de funciones

```
(define apply-procedure
  (lambda (proc args)
    (cases procval proc
      (closure (ids body env)
               (eval-expresion body (
   extend-env ids args env))))))
```

En el llamado de funciones se implementa esta funcion para poder extraer la informacion que contiene el procedure que nos envian de manera que se vean datos faciles de leer y no referencias de memoria

C. Funcionamiento correcto de todas las estructuras de datos y su manipulación

1)
```
my-language: --> var x = 10
in var f = func(a) (a + x)
in let x = 30 in call f(10)
20
```

D. Implementación de las declaraciones let y var

```
(define eval-decl-exp
  (lambda (decl env)
    (cases var-decl decl
      (let-exp (ids rands body)
        (if (contains-set-exp? body)
                    (eopl:error "No se puede
    modificar la ligadura de una variable en
    una declaracion let")
    (let ((args (eval-var-exp-rands rands env)
    ))
    (eval-expresion body (extend-env ids args
    env)))))

   (lvar-exp (ids rands body)
    (let ((args (eval-var-exp-rands rands env)
    ))
    (eval-expresion body (extend-env ids args
    env))))
      )))
```

Para implementar las declaraciones de variables **var** y **let** se definio esta funcion que recibe una var-decl-exp y dependiendo del caso, agrega la declaracion correspondiente al ambiente . Donde **ids** representa todos los identificadores que se hayan definido en la declaracion, **rands** o **args** representa los argumentos que tienen los identificadores declarados previamente y por ultimo el **body** representa lo que el programador quiere hacer con los identificadores que declaro antes

E. Diferencias entre var y let, ¿Por qué no permite la actualización de valores mientras que el otro si?

La diferencia esta en que la declaracion **var** es una declaracion dinamica de manera que se pueden modificar los argumentos de los identificadores creados en dicha declaracion, mientras que la declaracion **let** es una declaracion estatica por lo que no se pueden modificar los argumentos de sus identificadores declarados.

Debido a esto se definio la siguiente funcion:

```
(define (contains-set-exp? exp)
    (cases expresion exp
      (begin-exp (id rhs-exp) (
    contains-set-exp? id))

      (while-exp (condicion exp) (
    contains-set-exp? exp))
      (for-exp (itdor from until by body) (
    contains-set-exp? body))

    (if-exp (condicion then elses) (or (
    contains-set-exp? then) (contains-set-exp?
     elses)))

     (match-exp (item regular-exp casesValue)
    (car (map (lambda (x) (contains-set-exp? x
    )) casesValue)))

      (set-struct-exp (id exp1 exp2) #t)

      (set-exp (id exps) #t)

      (else #f)))
```

Que recibe una expresion y en caso de que esta expresion sea una **set-exp** o **set-struct-exp** nos retorna true indicandonos de esa manera que existe una expresion de ese tipo, luego se le manda un error al programador para indicarle que no puede hacer **"modificaciones de ligaduras en una declaracion let"**

F. Estrategia de separación de declaraciones de variables var y let

Como se indico en el punto anterior , con la funcion **contains-set-exp?** se asegura que en el momento que se haga una declaracion **let** no se este modificando variables, en el caso de que una declarcion **let** este declarada dentro de una declaracion **var** siempre y cuando en el cuerpo de la declaracion **let** no existan **set-exp** o **set-struct-exp** no se lanzara ningun error por intentar modificar variables y el codigo se ejecutara de manera normal

G. Implementacion y manejo de structs

Esta estructura de datos se manejo como una listas de listas donde el indice 0 de la lista siempre sera el nombre de la **struct** declarada. Aqui su definicion como listas de listas:

```
(define eval-struct-decl
  (lambda (structs)
    (cases struct-decl structs
      (struct-exp (id ids) (list id (map (
   lambda (x) x) ids))))) )
```

Ejemplo de funcionamiento

1)
```
my-language: --> struct perro { nombre edad color }
var x = new perro("lucas", 10,"verde")
in get x.nombre
"lucas"
```

2)
```
my-language: --> struct perro { nombre edad color }
var x = new perro("lucas", 10,"verde")
in begin set-struct x.nombre = "pedro" ; get x.nombre
 end
"pedro"
```

Aqui podemos ver que funciones correctamente sus funciones primitivas new , get y set-struct, que son para crear, traer datos y modificar datos respectivamente

V. Reconocimiento de patrones

Explicación de reconocmiento de patrones

Para lograr manejar esta estructura de datos, consideramos hace una funcion que evaluara cada **expresion regular** ya sea que fuera numero, cadena, boleano, lista, array o empty. En los ejemplos de mas abajo se quiere hacer match con la declaracion x, por lo que primero verificamos el tipo de dato con el que se quiere hacer match y de que comprueba que el dato coincida con alguno de los matches se ejecuta el cuerpo del match con el que coincidio como se puede observar

Funcionamiento correcto del reconocimiento de patrones con match

A. Ejemplos de reconocimiento

1) Listas con expresiones del tipo x::xs:

```
my-language: --> let x = list(1,2,3,4,5,6)
in match x {
numero(j) => j
boolean (s) => s
cadena(g) => g
x::xs => xs
array(x,y,z) => list(x,y,z)
default => 0
}
1) (2 3 4 5 6)
```

2) Números con expresiones del tipo numero(x):

```
my-language: --> let x = 10
in match x {
numero(j) => j
boolean (s) => s
cadena(g) => g
x::xs => xs
array(x,y,z) => list(x,y,z)
default => 0
}
1) 10
```

3) Cadenas con expresiones del tipo cadena(x):

```
my-language: --> let x = "prueba"
in match x {
numero(j) => j
boolean (s) => s
cadena(g) => g
x::xs => xs
array(x,y,z) => list(x,y,z)
default => 0
}
1) "prueba"
```

4) Booleanos con expresiones del tipo boolean(x):

```
my-language: --> let x = true
in match x {
numero(j) => j
boolean (s) => s
cadena(g) => g
x::xs => xs
array(x,y,z) => list(x,y,z)
default => 0

}
1) #t
```

5) Arrays con expresiones del tipo array(x,y,z):

```
my-language: -->  let x = array(1,2,3,4,5,6)
in match x {
numero(j) => j
boolean (s) => s
cadena(g) => g
x::xs => xs
array(x,y,z) => list(x,y,z)
default => 0
}
1) (1 2 #(3 4 5 6))
```

6) Listas vacías con empty:

```
my-language: --> let x = empty
in match x {
numero(j) => j
boolean (s) => s
cadena(g) => g
empty => empty
default => 0
}
1) ()
```