INFORME DEL PROYECTO FINAL DE FLP

ANDERSON GOMEZ GARCIA - 2266242
JUAN DAVID PÉREZ VALENCIA - 2266289
KEVIN ALEXIS LORZA RAMÍREZ - 2266098
JESUS EDIBER ARENAS -

FLP
UNIVERSIDAD DEL VALLE
19 DE DICIEMBRE DE 2024
TULUÁ (VALLE)

La implementación se lleva a cabo en Racket, un lenguaje ideal para la manipulación de lenguajes y construcción de interpretadores. Se desarrollan diferentes módulos que manejan aspectos específicos del lenguaje Obliq, tales como: Primitivas numéricas y de cadenas. Evaluación de expresiones booleanas y condicionales. Gestión de secuencias y procedimientos. Creación, manipulación y clonación de objetos. Manejo de variables y asignaciones, tanto locales como globales. Implementación de ciclos e iteraciones. Se presta especial atención al manejo de ambientes, asegurando que la extensibilidad de los mismos para soportar el alcance adecuado de variables y procedimientos, así como un correcto manejo de variables actualizables.

- Estructura y Organización: El código está bien estructurado y sigue una convención clara para la definición de primitivas, expresiones y operaciones booleanas. Extensibilidad: La generación automática de tipos de datos facilita la extensión y mantenimiento del código
- Funcionalidad: La función evaluar-programa sugiere que el código está diseñado para interpretar o evaluar programas escritos en un lenguaje específico
- Ambientes: La definición de ambientes sugiere que el código maneja contextos de ejecución, lo cual es crucial para la evaluación de programas

El primer paso en nuestro proyecto, es crear la especificación léxica, para poder detectar los identificadores, espacios en blanco, comentarios, etc

```
#lang eopl
(define especificacion-lexica
  '(
    (espacio-blanco (whitespace) skip)
    (comentario ("(*" (arbno (not #\newline))  "*)") skip)
    (identificador (letter (arbno (or letter digit "?" "$")))
symbol)
    (numero (digit (arbno digit)) number)
    (numero ("-" digit (arbno digit)) number)
    (numero (digit (arbno digit)"." digit (arbno digit))
number)
    (numero ("-" digit (arbno digit)"." digit (arbno digit))
number)
    ))
```

Después, procedemos con la construcción de la gramática, esto con el fin de crear un lenguaje a partir de la combinación de “tokens” de la especificación léxica

```
(define especificacion-gramatical
  '(
    (programa (expresion) a-program)
    (expresion (numero) lit-exp)
    (expresion (identificador) var-exp)
    (expresion ("var" (separated-list identificador "="
expresion ",") "in" expresion "end") variable-exp)
    (expresion ("let" (separated-list identificador "="
expresion "," ) "in" expresion "end") let-exp)
    (expresion ("proc" "(" (separated-list identificador ",")
")" expresion) proc-exp)
    (expresion ("begin" expresion (separated-list expresion
";") "end") begin-exp);; le puse ;
    (expresion ("set" identificador ":=" expresion)
set-exp);; le puse los 2 puntos
    ;;Primitivas
    (expresion (primitiva "(" (arbno expresion) ")")
prim-exp)
    (primitiva ("+") sum-prim)
    (primitiva ("-") minus-prim)
    (primitiva ("*") mult-prim)
    (primitiva ("/") div-prim)
    (primitiva ("add1") add-prim)
    (primitiva ("sub1") sub-prim)
    ;;primitivas booleanas
    (bool-prim (">") mayor-prim)
    (bool-prim (">=") mayorigual-prim)
    (bool-prim ("<") menor-prim)
    (bool-prim ("<=") menorigual-prim)
    (bool-prim ("is") igual-prim)
    ;primitiva clase
    (primitiva ("list") list-prim)
    ;;(expresion (arbno letter) cadena-exp)
    (expresion ("letrec" (arbno identificador "("
```

```
(separated-list identificador ",") ")" "=" expresion) "in"
expresion "end") letrec-exp);;tambien nuevo
    (expresion ("if" bool-expresion "then" expresion
                (arbno "elseif" bool-expresion "then"
expresion)
                    "else" expresion "end") if-exp)
    (expresion ("ok") ok-exp)
    (expresion ("apply" identificador "(" (separated-list
expresion ",") ")") apply-exp)
    (expresion ("meth" "(" identificador "," (separated-list
identificador ",") ")" expresion "end") meth-exp)
    (expresion ("for" identificador "=" expresion "to"
expresion "do" expresion "end\n") for-exp)
    (expresion ("object" "{" (arbno identificador "=>"
identificador) "}" ) object-exp)
    (expresion ("get" identificador "." identificador)
get-exp)
    (expresion ("send" identificador "." identificador
(separated-list identificador ",") ")") send-exp)
    (expresion ("update" identificador "." identificador ":="
expresion) update-exp)
    (expresion ("clone" "(" identificador (separated-list
identificador ",") ")") clone-exp)
    ;;operando booleanos
    (bool-oper ("not") not-prim)
    (bool-oper ("and") and-prim)
    (bool-oper ("or") or-prim)
    ;;bool-expresiones
    (bool-expresion ("true") true-exp)
    (bool-expresion ("false") false-exp)
    (bool-expresion (bool-prim "("  (separated-list expresion
",") ")") bool-exp-prim)
    (bool-expresion (bool-oper "(" (separated-list
bool-expresion "," ) ")") bool-exp-oper)
    (primitiva ("%") mod-prim)
    (primitiva ("&") concat-prim)
  ))
```

DATATYPES

sllgen:make-define-datatypes se utiliza para crear los tipos de datos automáticamente a partir de las especificaciones léxicas y gramaticales

```
;;Creamos los datatypes automaticamente
(sllgen:make-define-datatypes especificacion-lexica
especificacion-gramatical)
```

```
(define-datatype ambiente ambiente?
  (ambiente-vacio)
  (ambiente-extendido-ref
   (lids (list-of symbol?))
   (lvalue vector?)
   (old-env ambiente?)))
```

Representa el ambiente en el cual se evalúan las expresiones.

- Constructores:
  - ambiente-vacio:Un ambiente vacío.
  - ambiente-extendido: Un ambiente extendido con identificadores (ids), valores (vals) y un ambiente padre (parent).
  - ambiente-extendido-recursivo: Un ambiente extendido recursivamente con nombres de procedimientos (procnames), un vector de clausuras (vec-clausuras) y un ambiente padre (parent).

EVALUACION DE PROGRAMAS

evaluar-programa es una función que evalúa un programa utilizando la función evaluar-expresion y un ambiente inicial (ambiente-inicial)

```
;;Evaluar programa
(define evaluar-programa
  (lambda (pgm)
    (cases programa pgm
      (a-program (exp)
                 (evaluar-expresion exp ambiente-inicial)))))
```

VALORES POR DEFECTO DEL AMBIENTE

```
(define ambiente-inicial
  (ambiente-extendido '(x y z) '(4 2 5)
                      (ambiente-extendido '(a b c) '(4 5 6)
                                          (ambiente-vacio))))
```

```
  ;;ambientes
(define-datatype ambiente ambiente?
  (ambiente-vacio)
  (ambiente-extendido-ref
   (lids (list-of symbol?))
   (lvalue vector?)
   (old-env ambiente?)))

(define ambiente-extendido
  (lambda (lids lvalue old-env)
    (ambiente-extendido-ref lids (list->vector lvalue) old-env)))
```

El código define una función llamada ambiente-extendido-recursivo que extiende un ambiente con procedimientos recursivos.

- Letrec
  - nuevo-ambiente: Un nuevo ambiente extendido que referencia los nombres de los procedimientos (nombres-procedimientos) y las clausuras (vector-clausuras) en el ambiente anterior (ambiente-anterior).
  - obtener-clausuras: Una función recursiva que llena el vector vector-clausuras con las clausuras de los procedimientos.

```
;;Implementación ambiente extendido recursivo

(define ambiente-extendido-recursivo
  (lambda (procnames lidss cuerpos old-env)
    (let
        (
         (vec-clausuras (make-vector (length procnames)))
        )
      (letrec
          (
           (amb (ambiente-extendido-ref procnames vec-clausuras old-env))
           (obtener-clausuras
            (lambda (lidss cuerpos pos)
              (cond
                [(null? lidss) amb]
                [else
                 (begin
                   (vector-set! vec-clausuras pos
                                (closure (car lidss) (car cuerpos) amb))
                   (obtener-clausuras (cdr lidss) (cdr cuerpos) (+ pos 1)))]
                )
              )
            )
           )
        (obtener-clausuras lidss cuerpos 0)
        )
      )
    )
  )
```

EVALUADORES

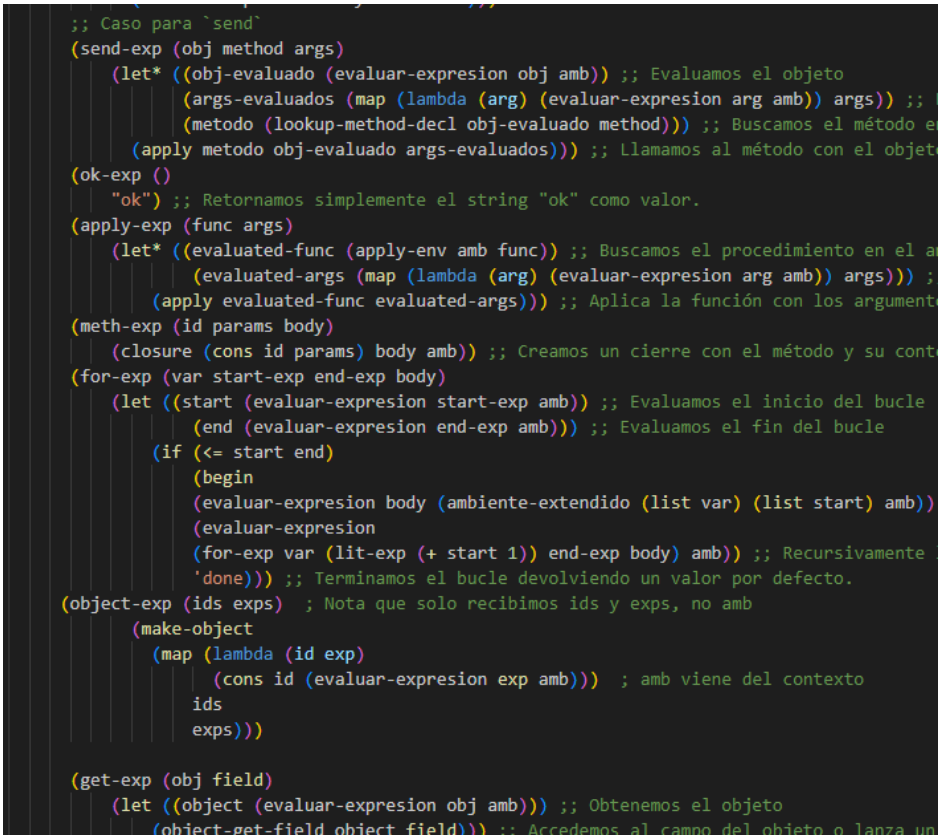
```
(define evaluar-expresion
  (lambda (exp amb)
    (cases expresion exp
        (variable-exp (ids vals body) ; Recibimos directamente los ids y las expresiones
            (let* ((valores-evaluados (map (lambda (exp) (evaluar-expresion exp amb)) vals))
                   (nuevo-amb (ambiente-extendido-ref ids
                                                      (list->vector valores-evaluados)
                                                      amb)))
              (evaluar-expresion body nuevo-amb)))
```

- Función: evaluar-programa esta función toma un programa (pgm) y lo evalúa utilizando la función evaluar-expresion en un ambiente inicial (ambiente-inicial).
- Parámetros: pgm el programa a evaluar.
- Retorno: El resultado de la evaluación del programa.

EVALUADOR DE EXPRESIONES

Aunque no se muestra explícitamente en el fragmento proporcionado, se asume que existe una función evaluar-expresion que evalúa expresiones individuales dentro del programa.

- Función: evaluar-expresion esta función evalúa una expresión en un ambiente dado.
- Parámetros:
    - exp: La expresión a evaluar.
    - ambiente: El ambiente en el cual se evalúa la expresión.
- Retorno: El resultado de la evaluación de la expresión



```
;; Caso para `send`
(send-exp (obj method args)
    (let* ((obj-evaluado (evaluar-expresion obj amb)) ;; Evaluamos el objeto
           (args-evaluados (map (lambda (arg) (evaluar-expresion arg amb)) args)) ;;
           (metodo (lookup-method-decl obj-evaluado method))) ;; Buscamos el método e
      (apply metodo obj-evaluado args-evaluados))) ;; Llamamos al método con el objet
(ok-exp ()
    "ok") ;; Retornamos simplemente el string "ok" como valor.
(apply-exp (func args)
    (let* ((evaluated-func (apply-env amb func)) ;; Buscamos el procedimiento en el a
           (evaluated-args (map (lambda (arg) (evaluar-expresion arg amb)) args))) ;
      (apply evaluated-func evaluated-args))) ;; Aplica la función con los argument
(meth-exp (id params body)
    (closure (cons id params) body amb)) ;; Creamos un cierre con el método y su cont
(for-exp (var start-exp end-exp body)
    (let ((start (evaluar-expresion start-exp amb)) ;; Evaluamos el inicio del bucle
          (end (evaluar-expresion end-exp amb))) ;; Evaluamos el fin del bucle
      (if (<= start end)
          (begin
          (evaluar-expresion body (ambiente-extendido (list var) (list start) amb))
          (evaluar-expresion
          (for-exp var (lit-exp (+ start 1)) end-exp body) amb)) ;; Recursivamente
          'done))) ;; Terminamos el bucle devolviendo un valor por defecto.
(object-exp (ids exps)  ; Nota que solo recibimos ids y exps, no amb
      (make-object
        (map (lambda (id exp)
               (cons id (evaluar-expresion exp amb)))  ; amb viene del contexto
             ids
             exps)))

(get-exp (obj field)
    (let ((object (evaluar-expresion obj amb))) ;; Obtenemos el objeto
      (object-get-field object field))) ;; Accedemos al campo del objeto o lanza un
```

EVALUADORES RECURSIVOS

```
(define ambiente-extendido-recursivo
  (lambda (procnames lidss cuerpos old-env)
    (let
        (
         (vec-clausuras (make-vector (length procnames)))
        )
      (letrec
          (
           (amb (ambiente-extendido-ref procnames vec-clausuras old-env))
           (obtener-clausuras
            (lambda (lidss cuerpos pos)
              (cond
                [(null? lidss) amb]
                [else
                 (begin
                   (vector-set! vec-clausuras pos
                                (closure (car lidss) (car cuerpos) amb))
                   (obtener-clausuras (cdr lidss) (cdr cuerpos) (+ pos 1)))]
              )
            )
           )
          )
        (obtener-clausuras lidss cuerpos 0)
        )
      )
    )
  )
```

- Función: ambiente-extendido-recursivo esta función extiende un ambiente con procedimientos recursivos.
- Parámetros:
    - procnames: Lista de nombres de los procedimientos.
    - lidss: Lista de listas de parámetros para cada procedimiento.
    - cuerpos: Lista de cuerpos de los procedimientos.
    - old-env: El ambiente anterior que se va a extender.
- Retorno: Un nuevo ambiente extendido con las clausuras de los procedimientos.

OBJETOS

Estas primitivas booleanas permiten realizar comparaciones básicas en el lenguaje definido.

```
(cases bool-prim prim
  (mayor-prim () (> (car lval) (cadr lval)))
  (mayorigual-prim () (>= (car lval) (cadr lval)))
  (menor-prim () (< (car lval) (cadr lval)))
  (menorigual-prim () (<= (car lval) (cadr lval)))
  (igual-prim () (= (car lval) (cadr lval)))
  (else (eopl:error "Primitiva booleana desconocida" prim)))
```

```
(define operacion-prim
  (lambda (lval op term)
    (cond
      [(null? lval) term]
      [else
       (op
        (car lval)
        (operacion-prim (cdr lval) op term))
       ]
      )
    )
  )
```

Función: operacion-prim realiza una operación primitiva recursiva sobre una lista de valores (lval), una operación (op) y un término (term).

```
(define-datatype referencia referencia?
  (a-ref (pos number?)
         (vec vector?)))
;;Extractor de referencias
(define deref
  (lambda (ref)
    (primitiva-deref ref)))

(define primitiva-deref
  (lambda (ref)
    (cases referencia ref
      (a-ref (pos vec)
             (vector-ref vec pos)))))
```

Función: deref extrae el valor de una referencia utilizando la función primitiva-deref.