INFORME DEL PROYECTO FINAL DE FLP

ANDERSON GOMEZ GARCIA - 2266242
JUAN DAVID PÉREZ VALENCIA - 2266289
KEVIN ALEXIS LORZA RAMÍREZ - 2266098
JESÚS EDIBER ARENAS - 2266066

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

ÍNDICE

Resumen del proyecto

El objetivo principal fue implementar un intérprete para un lenguaje inspirado en **Obliq**, utilizando Racket como lenguaje base. Este proyecto incluye la construcción de un sistema que maneja:

- Primitivas numéricas y booleanas.
- Evaluación de expresiones.
- Creación y manipulación de objetos.
- Definición y aplicación de ambientes jerárquicos.

Se prestó atención al manejo extensible de ambientes para permitir el alcance correcto de variables y procedimientos, con un enfoque especial en el soporte para variables actualizables.

Pasos principales

1. Especificación léxica

Se definieron reglas para reconocer elementos básicos del lenguaje:

- Identificadores.
- Espacios en blanco.
- Comentarios.

2. Construcción de la gramática

Se crearon reglas para:

- Estructurar programas.
- Definir expresiones numéricas, booleanas y condicionales.
- Modelar estructuras complejas como procedimientos y objetos.

3. Definición de datatypes

Se implementaron datatypes utilizando `sllgen` para:

- Representar programas.
- Modelar expresiones y primitivas.

4. Implementación de ambientes

Los ambientes se definen como estructuras jerárquicas:

- **Ambiente vacío:** Representa un entorno inicial sin variables.
- **Ambiente extendido:** Incluye identificadores, valores y un ambiente padre.
- **Ambiente recursivo:** Soporta procedimientos definidos recursivamente.

5. Evaluadores

Se desarrollaron evaluadores para interpretar expresiones y programas completos. Esto incluye:

- Evaluación de expresiones numéricas y booleanas.
- Evaluación de estructuras complejas como condicionales, `let`, `set`, y ciclos.

6. Manejo de objetos

Se implementaron funciones para:

- Crear objetos (`make-object`).
- Obtener y modificar campos de objetos (`get-field-object`, `set-field-object`).
- Clonar objetos (`clone-object`).

7. Interpretador

Se diseñó un interpretador que:

- Utiliza un bucle (`loop`) para evaluar programas.
- Realiza un escaneo y parseo de texto para ejecutar código en formato plano.

8. Pruebas y validación

Se usó **RacketUnit** para validar el intérprete, garantizando que las evaluaciones fueran correctas.

Definición léxica:

```
(define especificacion-lexica
  '(
    (espacio-blanco (whitespace) skip)
    (comentario ("(*" (arbno (not #\newline))  "*)") skip)
    (identificador (letter (arbno (or letter digit "?" "$"))) symbol)
    (cadena ("\"" (arbno (not #\")) "\"") string)
    (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)
    )
  )
```

Gramática:

Empezamos con un programa que contiene una expresion:

```
(define especificacion-gramatical
  '(
    (programa (expresion) a-program);; programa
```

Y definimos las expresiones:

```
;;expresiones
(expresion (numero) lit-exp)
(expresion (identificador) ide-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 "end") proc-exp)
(expresion ("begin" expresion (separated-list expresion ";") "end") begin-exp);; le puse ;
(expresion ("set" identificador ":=" expresion) set-exp);; le puse los 2 puntos
```

```
(expresion ("letrec" (arbno identificador "(" (separated-list identificador ",") ")" "=" expresion) "in" expresion "end") letrec-exp)
(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 ("for" identificador "=" expresion "to" expresion "do" expresion "end") for-exp)
;;expresiones para objetos
(expresion ("object" "{" (arbno identificador "=>" expresion) "}" ) 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)
(expresion ("meth" "(" identificador "," (separated-list identificador ",") ")" expresion "end") meth-exp)
```

Tambien definimos las primitivas (numericas) :

```
;;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)
(primitiva ("%") mod-prim)
(primitiva ("&") concat-prim)
(primitiva ("list") list-prim)
```

primitivas booleanas (relacionales) :

```
;;primitivas booleanas
(bool-prim (">") mayor-prim)
(bool-prim (">=") mayorigual-prim)
(bool-prim ("<") menor-prim)
(bool-prim ("<=") menorigual-prim)
(bool-prim ("is") igual-prim)
```

operandos booleanos (sobre cadenas):

```
;;operando booleanos
(bool-oper ("not") not-prim)
(bool-oper ("and") and-prim)
(bool-oper ("or") or-prim)
```

y finalmente bool expresions:

```
;;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)
```

Datatypes

Se generan los datatypes mediante sllgen:

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

Evaluadores:

Iniciamos evaluando el programa:

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

Ambientes:

Para asignar un ambiente primero debemos definir el datatype de un ambiente y su evaluador:

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

```
(define (ambiente-extendido ids vals amb)
  (if (or (null? ids) (null? vals))
      amb
      (ambiente-extendido-ref
       ids
       (list->vector vals)
       amb)))
```

Estando ya definiendo ambientes, definimos también el ambiente-extendido-recursivo:

```
;;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)
        ))))
```

Definimos nuestro ambiente inicial con los identificadores y sus valores con los que queremos empezar el programa:

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

Aplicar Ambientes:

```
;;Aplicar ambientes
(define apply-env
  (lambda (env var)
    (deref (apply-env-ref env var))))
```

```
(define apply-env-ref
  (lambda (env var)
    (cases ambiente env
      (ambiente-vacio () (eopl:error "No se encuentra la variable " var))
      (ambiente-extendido-ref (lid vec old-env)
                        (letrec
                            (
                             (buscar-variable (lambda (lid vec pos)
                                                (cond
                                                  [(null? lid) (apply-env-ref old-env var)]
                                                  [(equal? (car lid) var) (a-ref pos vec)]
                                                  [else
                                                   (buscar-variable (cdr lid) vec (+ pos 1)  )]
                                                 )
                                               )
                                              )
                             )
                          (buscar-variable lid vec 0)
                          )
                        )
      )
    )
  )
```

Evaluadores:

Teniendo ya la gramática y los datatypes de las expresiones definidas vamos a evaluarlos para agregarles funcionalidad:
var-expresion

```
;;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)))
```

Send-exp

```
;; 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)) ;; Evaluamos los argumentos
           (metodo (lookup-method-decl obj-evaluado method))) ;; Buscamos el método en el objeto
```

```
(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 ambiente
           (evaluated-args (map (lambda (arg) (evaluar-expresion arg amb)) args))) ;; Evaluamos los argumentos
      (apply evaluated-func evaluated-args))) ;; Aplica la función con los argumentos evaluados.
(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)) ;; Ejecutamos el cuerpo
            (evaluar-expresion
            (for-exp var (lit-exp (+ start 1)) end-exp body) amb)) ;; Recursivamente llamamos al siguiente ciclo
            'done))) ;; Terminamos el bucle devolviendo un valor por defecto.
```

Evaluadores para las expresiones relacionadas a los objetos:

```
(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)))
(meth-exp (id params body)
    (closure (cons id params) body amb)) ;; Creamos un cierre con el método y su contexto.
(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 error si no existe

(update-exp (obj field value)
    (let ((object (evaluar-expresion obj amb))
          (new-value (evaluar-expresion value amb))) ;; Evaluamos el nuevo valor
        (object-set-field! object field new-value))) ;; Actualizamos el campo del objeto o lanza un error si no existe

(clone-exp (obj new-fields)
    (let ((original (evaluar-expresion obj amb))
          (fields (map (lambda (field) (apply-env amb field)) new-fields))) ;; Evaluamos los nuevos campos
        (clone-object original fields))) ;; Clonamos el objeto con los nuevos campos.
```

Evaluar condicional:

```
;;Condicionales
(if-exp (condicion cuerpo-principal elseif-cuerpos cuerpo-else end)
  (let loop ((condiciones (cons (list condicion cuerpo-principal) elseif-cuerpos)))
     (if (null? condiciones)
         (evaluar-expresion cuerpo-else amb) ;; Si no hay condiciones, evaluar el "else"
         (let ((actual-condicion (car (car condiciones))) ;; Obtener la condición actual
             (actual-cuerpo (cadr (car condiciones)))) ;; Obtener el cuerpo asociado
         (if (evaluar-bool-expresion actual-condicion amb) ;; Cambia evaluar-expresion por evaluar-bool-expresion
             (evaluar-expresion actual-cuerpo amb) ;; Si es verdadera, evaluamos su cuerpo
             (loop (cdr condiciones)))))))
```

Evaluar let, procs, letrec, begin:

```
;;Ligaduras locales
(let-exp (ids rands body)
         (let
             (
              (lvalues (map (lambda (x) (evaluar-expresion x amb)) rands))
              )
           (evaluar-expresion body (ambiente-extendido ids lvalues amb))
           )
         )
;;procedimientos
(proc-exp (ids body)
          (closure ids body amb))
;;letrec
(letrec-exp (procnames idss cuerpos cuerpo-letrec)
            (evaluar-expresion cuerpo-letrec
                               (ambiente-extendido-recursivo procnames idss cuerpos amb)))
;;begin
(begin-exp (exp lexp)
           (if
            (null? lexp)
            (evaluar-expresion exp amb)
            (begin
              (evaluar-expresion exp amb)
              (letrec
                  (
                   (evaluar-begin (lambda (lexp)
                                    (cond
                                      [(null? (cdr lexp)) (evaluar-expresion (car lexp) amb)]
                                      [else
                                       (begin
                                         (evaluar-expresion (car lexp) amb)
                                         (evaluar-begin (cdr lexp))
                                         )])))
                   )
                (evaluar-begin lexp)
                ))))
```

Evaluar set:

```
;;set
(set-exp (id exp)
         (begin
           (setref!
            (apply-env-ref amb id)
            (evaluar-expresion exp amb))
           'ok))
           )))
```

Evaluadores auxiliares (para apoyar evaluadores de expresiones)

Son evaluadores de primitiva, bool-expresion, bool-expresion–primitiva

```
;;Evaluadores auxiliares
(define evaluar-primitiva
  (lambda (prim lval)
    (cases primitiva prim
      (sum-prim () (operacion-prim lval + 0))
      (minus-prim () (operacion-prim lval - 0))
      (mult-prim () (operacion-prim lval * 1))
      (div-prim () (operacion-prim lval / 1))
      (add-prim () (+ (car lval) 1))
      (sub-prim () (- (car lval) 1))
      (mod-prim () (remainder (car lval) (cadr lval))) ;; Añadimos el caso para mod-prim
      (concat-prim () (string-append (car lval) (cadr lval))) ;; Añadimos el caso para concat-prim
      (list-prim () lval)
      )))
```

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

operación-prim para las primitivas

```
(define evaluar-bool-expresion
    (lambda(exp env)
        (cases bool-expresion exp
            (true-exp () #true)
            (false-exp () #false)
            (bool-exp-prim (prim args)
                (let ((lista-valores (map (lambda (arg) (evaluar-expresion arg env)) args)))
                    (evaluar-primitiva-booleano prim lista-valores)))
            (bool-exp-oper (oper args)
                (let ((lista-bool (map (lambda (arg) (evaluar-bool-expresion arg env)) args)))
                    (evaluar-operador-booleano oper lista-bool))))))
```

```
(define evaluar-operador-booleano
    (lambda (operador lista-bool)
        (cases bool-oper operador
            (not-prim () (not (car lista-bool)))
            (and-prim () (and (car lista-bool) (cadr lista-bool)))
            (or-prim () (or (car lista-bool) (cadr lista-bool)))
            (else (eopl:error "Operador booleano desconocido" operador)))))
```

```
(define evaluar-primitiva-booleano
  (lambda (prim lval);;lval es una lista de valores
    (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)))))
```

Referencias para los valores (datatype, deref, setref)

```
;;Referencias
(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)))))
;;Asignación/cambio referencias
(define setref!
  (lambda (ref val)
    (primitiva-setref! ref val)))
(define primitiva-setref!
  (lambda (ref val)
    (cases referencia ref
      (a-ref (pos vec)
             (vector-set! vec pos val)))))
```

Funcionalidades para objetos:

Se define las acciones necesarias para los objetos (make-object, get-field-object, set-field-object, clone-object)

```
(define make-object
    (lambda (bindings)
        (let ((obj (make-vector (length bindings))))
        (letrec
            (
             (set-fields
                (lambda (bindings pos)
                (cond
                    [(null? bindings) obj]
                    [else
                     (begin
                     (vector-set! obj pos (car bindings))
                     (set-fields (cdr bindings) (+ pos 1))
                     )])))
            (set-fields bindings 0)))))
(define object-get-field
  (lambda (obj field-id)
    (let ((field-names (vector-ref obj 0))
          (field-values (vector-ref obj 1)))
      (let loop ((names field-names)
                 (values field-values))
        (if (null? names)
            (eopl:error "Campo no encontrado: " field-id)
            (if (equal? (car names) field-id)
                (car values)
                (loop (cdr names) (cdr values))))))))
```

```
(define object-set-field!
  (lambda (obj field-id new-val)
    (let ((field-names (vector-ref obj 0))
          (field-values (vector-ref obj 1)))
      (let loop ((names field-names)
                 (values field-values))
        (if (null? names)
            (eopl:error "Campo no encontrado: " field-id)
            (if (equal? (car names) field-id)
                (begin
                  (vector-set! field-values (vector-length field-names) new-val)
                  'ok)
                (loop (cdr names) (cdr values))))))))
```

```
(define (clone-object original new-fields)
  (let ((field-names (vector-ref original 0))
        (field-values (vector-ref original 1)))
    (let ((new-field-names (append field-names (map car new-fields)))
          (new-field-values (append field-values (map cdr new-fields))))
      (vector new-field-names new-field-values))))
```

Interpretador

Definimos el interpretador que es un loop de evaluar programa por medio de stream-parser también lo podemos definir mediante scan&parse para ejecutar texto plano:

```
(define interpretador
  (sllgen:make-rep-loop "-->" evaluar-programa
                        (sllgen:make-stream-parser
                         especificacion-lexica especificacion-gramatical)))
```

Ejecutamos el código….

Pruebas:

Para las pruebas usamos racketunit y redefinimos el interpretador con un scan&parse:
e instanciamos (provide (all-defined-out))
Se configura así:

```
(require rackunit "InterpretadorObliq.rkt")

(define exp1
  (scan&parse
    "var x = 10, y = 20, z = +(x y) in *(z 2) end"
  )
)

(define expected-exp1
  60
)

(define exp2
  (scan&parse
  "let x = 2 in *(x 4) end"
  )
)

(define expected-exp2
  8
)

(define exp3
  (scan&parse
    "if false then 2 elseif false then 3 else 4 end"
  )
)
```

```
(define test-list-functions
  (test-suite "lista de pruebas"
        (test-case "Test exp1"
    (check-equal? (evaluar-programa exp1) expected-exp1))
   (test-case "Test exp2"
    (check-equal? (evaluar-programa exp2) expected-exp2))
   (test-case "Test exp3"
    (check-equal? (evaluar-programa exp3) expected-exp3))
   (test-case "Test exp4"
    (check-equal? (evaluar-programa exp4) expected-exp4))
   (test-case "Test exp5"
    (check-equal? (evaluar-programa exp5) expected-exp5))
   (test-case "Test exp6"
    (check-equal? (evaluar-programa exp6) expected-exp6))
   (test-case "Test exp7"
    (check-equal? (evaluar-programa exp7) expected-exp7))
))
```

La terminal no tiene salidas resultando que no hubo errores:

```
PS C:\Users\Ander\Desktop\FLP_PROYECT> racket  c:/Users/Ander/Desktop/FLP_PROYECT/pruebas.rkt
```

Conclusiones

En este proyecto creamos un lenguaje Obliq y desarrollamos un intérprete funcional en Racket. Aprendimos a manejar ambientes, evaluar expresiones y construir objetos, todo diseñado por nosotros. Aunque hubo retos, como entender la recursividad y los ambientes extendidos, logramos que todo funcionara como esperábamos. Este trabajo nos dejó una base sólida para seguir explorando la creación de lenguajes y sistemas más complejos.