Proyecto Final

Construcción interpretador ObliQ

Integrantes:

Samuel Escobar Rivera - 2266363

César David Peñaranda Melo - 2266265

Joseph Herrera Libreros - 2266309

Juan David Cuellar Lopez - 2266087

Universidad del Valle - Seccional Tuluá

Facultad de Ingeniería

Fundamentos de Interpretación y Compilación de Lenguajes de Programación

Carlos Andres Delgado S, Msc

Diciembre de 2024

En este informe tenemos como objetivo número uno explorar y aplicar conceptos fundamentales en el diseño de lenguajes, análisis léxico, análisis sintáctico y evaluación de expresiones en un entorno controlado.

1. **Primitivas numéricas y cadenas**
   Las primitivas numéricas y cadenas son funciones básicas que manejan los tipos numéricos y las cadenas de textos.

   - **Primitivas numéricas**

```
;; Evaluador de primitivas
(define evaluar-primitiva
  (lambda (prim args)
    (cases primitiva prim
      (add-prim () (apply + args))
      (minus-prim () (apply - args))
      (mult-prim () (apply * args))
      (mod-prim () (apply modulo args))
      (concat-prim () (apply string-append args)))))
```

     Las operaciones aritméticas se implementan usando funciones primitivas como add-prim, minus-prim, mult-prim y mod-prim. Estas funciones toman una lista de argumentos, los evalúan y aplican la operación correspondiente.

   - **Primitiva de cadenas**

```
(define evaluar-primitiva
  (lambda (prim args)
    (cases primitiva prim
      (concat-prim () (apply string-append args)))))
```

     En el caso de las cadenas, la operación concat-prim se utiliza para concatenar cadenas. En este caso, se usa la función string-append de Scheme que une dos o más cadenas.

2. **Evaluación de expresiones booleanas y condicionales**
   Las expresiones booleanas permiten realizar operaciones lógicas, como comparar valores o verificar condiciones. Los operadores booleanos incluyen greater-prim, less-prim, is-prim, entre otros, estos se evalúan utilizando la función evaluar-bool-expression.

   - **Evaluación de expresiones booleanas**

```
(define evaluar-bool-expresion
  (lambda (exp env)
    (cases bool-expresion exp
      (true-exp () #true)
      (false-exp () #false)
      (bool-prim-exp (prim args)
                     (let ((lista-vals (map (lambda (x) (evaluar-expresion x env)) args)))
                       (evaluar-bool-primitiva prim lista-vals)))
      (bool-oper-exp (oper args)
                     (let ((lista-vals (map (lambda (x) (evaluar-bool-expresion x env)) args)))
                       (evaluar-bool-operacion oper lista-vals))))))
```

Estas se evalúan mediante técnicas primitivas como greater-prim o is-prim. Además, el operador lógico not, junto con and y or permite realizar evaluaciones complejas de condiciones.

- **Condicionales**

```
(define evaluar-elseifs
  (lambda (conditions expressions else-exp env)
    (cond
      [(null? conditions)
       (evaluar-expresion else-exp env)]
      [(evaluar-expresion (car conditions) env)
       (evaluar-expresion (car expressions) env)]
      [else
       (evaluar-elseifs (cdr conditions) (cdr expressions) else-exp env)])))
```

Se implementa un evaluador para las expresiones condicionales. Si la condición es verdadera, se ejecuta la expresión correspondiente, si no lo es se evalúan las expresiones de los elseif o else.

3. **Gestión de secuencias y procedimientos**
   En el caso de los procedimientos, se implementan expresiones proc-exp y apply-exp que permiten definir y aplicar funciones.

- **Procedimientos definidos por el usuario**

```
;; Procedimientos
(proc-exp (ids body)
          (closure ids body env))
```

Un procedimiento se define usando proc-exp, que toma una lista de parámetros y un cuerpo. Para aplicar el procedimiento se evalúan los argumentos en un nuevo entorno.

- **Aplicación de procedimientos**

```
(apply-exp (proc-id args)
        (let* ((proc (apply-env env proc-id))
               (args-vals (map (lambda (arg)
                                 (evaluar-expresion arg env))
                               args)))
          (if (procval? proc)
              (cases procval proc
                (closure (ids body saved-env)
                         (evaluar-expresion body
                                            (ambiente-extendido ids args-vals saved-env))))
              (eopl:error "No es un procedimiento: " proc))))
```

Los procedimientos se aplican usando apply-exp. Los argumentos se evalúan y se asignan a los parámetros del procedimiento, creando un entorno nuevo en el cual se evalúa el cuerpo del procedimiento.

4. **Creación, manipulación y clonación de objetos**
   Los objetos en este lenguaje son representados como estructuras con campos. Se implementan operaciones para crear objetos, acceder a sus campos y actualizar estos campos.

   - **Creación de objetos**

```
(object-exp (field-names field-exprs)
            (let ((field-values (map (lambda (expr)
                                       (evaluar-expresion expr env))
                                     field-exprs)))
              (make-object field-names field-values)))
```

Los objetos se crean utilizando la expresión object-exp. Esto genera una estructura que contiene campos cuyo valor es evaluado.

   - **Acceso a campos y clonación de objetos**

```
;; Definición de la función object-get-field
(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)))))))))
```

Para acceder a un campo de un objeto, se usa la operación object-get-field. Para modificar los valores de los campos de un objeto se usan funciones como object-set-field!.

5. **Manejo de variables y asignaciones, tanto locales como globales**
   Las variables se manejan mediante entornos que permiten la evaluación de expresiones y la asignación de valores.

   - **Entornos extendidos**

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

   Los entornos locales y globales se gestionan mediante ambiente-extendido que crea un nuevo entorno con las variables y sus valores.

   - **Asignaciones**

   ```
   (define apply-env
     (lambda (env var)
       (deref (apply-env-ref env var))))
   ```

   Las asignaciones de valores a variables se manejan usando la expresión set! que modifica una variable existente en el entorno.

6. **Implementación de ciclos e iteraciones**
   Los ciclos y las iteraciones se implementan utilizando recursión o utilizando expresiones como for-exp que repiten un bloque de código mientras se cumpla una condición.

   - **Ciclos**

```
;; Ciclo for
(for-exp (var init-exp final-exp body)
         (let ((init-val (evaluar-expresion init-exp env))
               (final-val (evaluar-expresion final-exp env)))
           (let loop ((i init-val))
             (if (> i final-val)
                 'ok
                 (begin
                   (evaluar-expresion body
                                      (ambiente-extendido (list var)
                                                          (list i)
                                                          env))
                   (loop (+ i 1)))))))
```

Un ciclo for se implementa recursivamente. Se evalúa la condición inicial y se realiza una iteración hasta que se alcanza el valor final. Durante cada iteración, se actualizan las variables y se ejecuta el cuerpo del ciclo.

PRUEBAS

Para validar la correcta funcionalidad del interpretador, se diseñaron y ejecutaron diversos casos de prueba que abarcan principales características del lenguaje. Los test evalúan expresiones aritméticas, booleanas, condicionales, bucles, manejo de variables y objetos.

1. **Pruebas de expresiones con let**

```
(define let-exp1
  (scan&parse "let x = 5 in x end"))

(define let-expect1
5)
```

Verifica que el intérprete pueda evaluar una expresión simple que asigna un valor a una variable y devuelve su valor.
Resultado esperado: 5

```
(define let-exp2
  (scan&parse "let x = 5 in let y = 3 in +(x, y) end end"))

(define let-expect2
8)
```

Comprueba que el intérprete soporte anidación de expresiones let y realice operaciones aritméticas correctamente.
Resultado esperado: 8

```
(define let-exp3
  (scan&parse "let x = 5 in let y = 3 in let z = 2 in +(+(x, y), z) end end end"))

(define let-expect3
10)
```

Valida que el intérprete maneje múltiples niveles de anidación en expresiones let y operaciones acumulativas.
Resultado esperado: 10

2. **Pruebas de expresiones secuenciales**

```
(define begin-exp1
  (scan&parse "begin let x = 5 in x end; let x = 3 in x end end"))

(define begin-expect1
3)
```

Prueba la ejecución secuencial de expresiones begin, asegurándose de que el valor de una variables puede redefinirse en un nuevo ámbito.
Resultado esperado: 3

```
(define begin-exp2
  (scan&parse "begin let x = 5 in let y = 3 in +(x, y) end end; let x = 3 in x end end"))

(define begin-expect2
3)
```

Valida que las operaciones dentro de un bloque secuencial se evalúan correctamente y las redefiniciones posteriores no afectan el bloque anterior.
Resultado esperado: 3

```
(define begin-exp3
  (scan&parse "begin let x = 5, y = 3 in if is(x, y) then 1 else 0 end end end"))

(define begin-expect3
0)
```

Comprueba que el intérprete evalúe condicionales con comparación de igualdad dentro de un bloque secuencial.
Resultado esperado: 0

3. **Pruebas de objetos**

```
(define exp1
  (scan&parse "object {x => 5}")) ;; esto

(define expect1
    (list (cons 'x 5))) ;; #(x 5)
```

Verifica la creación de un objeto simple con un campo y su representación interna.
Resultado esperado: (list(cons ‘x 5))

```
(define exp2
  (scan&parse "object {x => 5 y => 3}"))

(define expect2
    (list (cons 'x 5) (cons 'y 3)))
```

Valida que el intérprete pueda crear objetos con múltiples campos.
Resultado esperado: (list (cons ‘x 5) (cons ‘y 3))

4. **Pruebas de condicionales**

```
(define exp3
  (scan&parse "if >(5, 3) then 1 else 0 end"))

(define expect3
    1)
```

Comprueba la evaluación de un condicional con un operador booleano simple.
Resultado esperado: 1

```
(define exp4
  (scan&parse "let x = 2, y = 3 in if >(x, y) then 1 else 0 end end"))

(define expect4
    0)
```

Válida condicionales dentro de un bloque let con variables definidas en tiempo de ejecución.

Resultado esperado: 0

```
(define exp5
(scan&parse "if <(5, 3) then 1 else 0 end"))

(define expect5
    0)
```

Evalúa un condicional que devuelve falso.
Resultado esperado: 0

5. **Pruebas de bucles y variables**

```
(define var-exp1
  (scan&parse "var x = 5 in begin let y = +(x, 4) in y end end end"))

(define var-expect1
9)
```

Válida la definición de variables con var y su uso dentro de un bloque secuencial.
Resultado esperado: 9

```
(define for-exp4
  (scan&parse "for x = 1 to 5 do x end")) ;; esto retorna un ok

(define for-expect4
'ok)
```

Verifica que el bucle for recorre correctamente un rango y retorna ‘ok.
Resultado esperado: ‘ok

6. Lista de funciones

Para garantizar la correcta funcionalidad del interpretador, se implementó un conjunto de pruebas utilizando rackunit, este conjunto evalúa las características claves del lenguaje y compara los resultados obtenidos con los esperados.

```
(define test-list-functions
  (test-suite "Test de funciones"

   (check-equal? (evaluar-programa let-exp1) let-expect1)
   (check-equal? (evaluar-programa let-exp2) let-expect2)
   (check-equal? (evaluar-programa let-exp3) let-expect3)
   (check-equal? (evaluar-programa begin-exp1) begin-expect1)
   (check-equal? (evaluar-programa begin-exp2) begin-expect2)
   (check-equal? (evaluar-programa exp1) expect1)
   (check-equal? (evaluar-programa exp2) expect2)
   (check-equal? (evaluar-programa exp3) expect3)
   (check-equal? (evaluar-programa exp4) expect4)
   (check-equal? (evaluar-programa exp5) expect5)
   (check-equal? (evaluar-programa begin-exp3) begin-expect3)
   (check-equal? (evaluar-programa var-exp1) var-expect1)
   (check-equal? (evaluar-programa for-exp4) for-expect4)


   )
)


(run-test test-list-functions)
```