INFORME PROYECTO FINAL

AUTORES:

JUAN CAMILO GUTIERREZ VIVEROS - 2159874

CARLOS FERNANDO PADILLA MESA – 2059962

ANDRÉS FELIPE ROJAS – 2160328

DOCENTE:

CARLOS ANDRÉS DELGADO SAAVEDRA

UNIVERSIDAD DEL VALLE

INGENIERÍA DE SISTEMAS

TULUÁ, VALLE DEL CAUCA

19 DE DICIEMBRE DE 2024

Introducción

En el siguiente informe se encontrará cómo se desarrolló el interpretador en Racket de un lenguaje inspirado en Obliq, el cual facilita la programación orientada a objetos con estructuras de paso de parámetros por valor. El código implementado permite trabajar con ambientes, hacer evaluaciones de expresiones y también usar estructuras como ciclos, condicionales, procedimientos. Adicional a esto se creó un código adicional que nos permitirá probar que el interpretador de lenguaje Obliq funcionara correctamente. A continuación, se mostrará detalladamente cada parte del código y cómo fue implementado.

Realización del interpretador

El primer paso en la creación del interpretador del Lenguaje Obliq, fue hacer la representación de ambientes, recordemos que un ambiente es una estructura en donde se asocian las variables y sus valores y se usan para almacenar en ellos los estados de las variables en el transcurso de la ejecución de un programa.

1: interpretador.rkt | 2: pruebas.rkt

```
#lang racket
(require racket/list)

;; ============================
;; Obliq Interpreter - Starter
;; ============================

(provide empty-env lookup extend-env update-env eval-exp)

;; Representación de Ambientes
(define empty-env '())

;; Buscar una variable en el ambiente.
(define (lookup var env)
  (cond [(null? env) (error "Variable no encontrada" var)]
        [(eq? (car (car env)) var) (cdr (car env))]
        [else (lookup var (cdr env))]))

;; Extender el ambiente con una nueva asociación.
(define (extend-env var val env)
  (cons (cons var val) env))

;; Actualizar el valor de una variable en el ambiente.
(define (update-env var val env)
  (cond [(null? env) (error "Variable no encontrada para actualizar" var)]
        [(eq? (car (car env)) var) (cons (cons var val) (cdr env))]
        [else (cons (car env) (update-env var val (cdr env)))]))
```

Como se puede ver en el código, se hace la representación del ambiente vacío el cual se usa como punto de partida como extensión a más ambientes que se vayan creando. Después se define la función ‘lookup’ que nos permite buscar variables en los ambientes, teniendo como condición de parada que en caso de que el ambiente esté vacío no retornada nada y al caso contrario se hace una comparación recursiva hasta dar con la variable y valor deseada.

Se puede encontrar después la definición de ‘extend-env’ la cual nos permite extender o crear ambientes tomando como punto de partida el ambiente vacío. Y por ultimo la definición de ‘update-env’ que nos permite de manera recursiva recorrer los ambiente y actualizar los valores de las variables deseadas.

Representación de evaluación de expresiones

La segunda parte del interpretador de lenguaje Obliq consiste en la definición de las representaciones aritméticas y demás condicionales que pide el proyecto.

1: interpretador.rkt | 2: pruebas.rkt

```
1   #lang racket
33  ;; Función principal de evaluación
34  (define (eval-exp exp env)
35    (cond
36      ;; Constantes numéricas
37      [(number? exp) exp]
38
39      ;; Operaciones aritméticas
40      [(and (list? exp) (member (car exp) '(+ - * %)))
41       (let ([args (map (lambda (e) (eval-exp e env)) (cdr exp))])
42         (case (car exp)
43           [(+) (apply + args)]
44           [(-) (apply - args)]
45           [(*) (apply * args)]
46           [(%) (apply modulo args)]))]
47
48      ;; Operaciones relacionales
49      [(and (list? exp) (member (car exp) '(< <= > >= is)))
50       (let ([args (map (lambda (e) (eval-exp e env)) (cdr exp))])
51         (case (car exp)
52           [(<) (< (car args) (cadr args))]
53           [(<=) (<= (car args) (cadr args))]
54           [(>) (> (car args) (cadr args))]
55           [(>=) (>= (car args) (cadr args))]
56           [(is) (equal? (car args) (cadr args))]))]
57
```

Se ha llegado al núcleo del del interprete pues es en esta parte del código en la que se traduce expresiones en valores mediante reglas específicas.

Empezamos con ‘eval-exp’ que a través de un condicional permite decidir cómo evaluar cada expresión con las clausulas que se encuentran en ella.

Constantes numéricas: En esta cláusula se evalúa si la expresión indicada es un número.

Operaciones aritméticas: En esta parte del código se evalúan operaciones artimeticas representadas como listas. Primero verifica que la expresión evaluada sea una lista y que la operación esté en el conjunto, después evalúa recursivamente los operandos en ell ambiente y por ultimo por medio de los cases ejecuta las respectivas operaciones.

Operaciones relacionales: Esta parte del código presenta similitudes con la ya antes vista de operaciones aritméticas, con la diferencia que a las expresiones se les hace comparaciones de igualdad, mayor que, menor que, etc.

```
1  #lang racket
58     ;; Condicionales
59     [(and (list? exp) (equal? (car exp) 'if))
60      (let ([cond (eval-exp (cadr exp) env)])
61        (if cond
62            (eval-exp (caddr exp) env)
63            (eval-exp (cadddr exp) env)))]
64
65     ;; Ciclo for
66     [(and (list? exp) (equal? (car exp) 'for))
67      (let loop ([i (eval-exp (caddr exp) env)]
68                 [end (eval-exp (cadddr exp) env)]
69                 [env env])
70        (if (> i end)
71            'ok
72            (begin
73              (eval-exp (car (cddddr exp)) (extend-env (cadr exp) i env))
74              (loop (+ i 1) end env))))]
75
76     ;; Definición de procedimientos
77     [(and (list? exp) (equal? (car exp) 'proc))
78      (let ([params (cadr exp)]
79            [body (caddr exp)])
80        (lambda args
81          (if (not (= (length params) (length args)))
82              (error "Número incorrecto de argumentos" params args)
83              (let ([new-env (foldr (lambda (param-val env)
84                                      (extend-env (car param-val) (cdr param-val) env))
85                                    env
86                                    (map cons params args))])
87                (eval-exp body new-env)))))]
```

En esta parte del código podemos ver que se le hace ampliación al interpretador al agregarle cláusulas que permitan evaluar expresiones por medio de condicionales ‘if’ y ‘for’.

En la ultima parte de la imagen encontramos la definición de procedimientos, la cual este código nos permite definir un procedimiento o función con un cuerpo de instrucciones. Primero verifica que la expresión sea una lista y que el primer elemento de la expresión empiece con un ‘proc, seguido extrae los parámetros y el cuerpo y retorna una función anónima en donde se reciben estos parámetros y verifica que coincida con la cantidad de parámetros, y como ultimo acto extiende un nuevo ambiente y el cuerpo del procedimiento se evalúa en este mismo.

```
1  | #lang racket
89     ;; Aplicación de procedimientos
90     [(and (list? exp) (equal? (car exp) 'apply))
91      (let ([proc (eval-exp (cadr exp) env)]
92            [args (map (lambda (e) (eval-exp e env)) (caddr exp))])
93        (apply proc args))]
94
95     ;; Definición de una variable
96     [(and (list? exp) (equal? (car exp) 'define))
97      (let ([var (cadr exp)]
98            [val (eval-exp (caddr exp) env)])
99        (extend-env var val env))]
00
01     ;; Manejo de errores y excepciones
02     [(and (list? exp) (equal? (car exp) 'try))
03      (let ([try-block (cadr exp)]
04            [catch-block (caddr exp)])
05        (with-handlers ([exn:fail? (lambda (_) (eval-exp catch-block env))])
06          (eval-exp try-block env)))]
```

En esta sección del intérprete se añade soporte para invocar procedimientos, definir variables en el entorno, y manejar errores mediante bloques try-catch.

Aplicación de procedimientos: En esta parte del interpretador se permite hacer llamado o invocación a procedimientos ya previamente definidos con argumentos evaluados por medio del llamado ‘apply.

Definición de una variable: En esta parte del código se introduce una nueva variable en el entorno actual y se le asocia a un valor.

Manejo de errores y excepciones: Acá podemos por medio del uso de try catch darle excepciones a las evaluaciones de expresiones, cuando se ingresan mal las expresiones o cuando no queremos que se cumpla algo en las evaluaciones.

```
1  | #lang racket
108    ;; Manejo de objetos
109    [(and (list? exp) (equal? (car exp) 'object))
110     (let ([fields (map (lambda (pair)
111                          (let* ([key (car pair)]
112                                 [val (cdr pair)]
113                                 [evaluated-val (if (and (list? val) (equal? (car val) 'proc))
114                                                    (eval-exp val env)
115                                                    (eval-exp val env))]
116                                 [normalized-pair (if (equal? key '=>) (cons (cadr pair) (caddr pair)) pair)])
117                            (cons (car normalized-pair) evaluated-val)))
118                        (cdr exp))])
119       (lambda (msg . args)
120         (case msg
121           [(get) (let ([field (car args)])
122                    (if (assoc field fields)
123                        (cdr (assoc field fields))
124                        (error "Campo no encontrado" field)))]
125           [(update) (let ([field (car args)] [new-val (cadr args)])
126                       (if (assoc field fields)
127                           (set! fields (map (lambda (pair)
128                                               (if (eq? (car pair) field)
129                                                   (cons (car pair) new-val)
130                                                   pair))
131                                             fields))
132                           (error "Campo no encontrado para actualizar" field))
133                       new-val)]
134           [(send) (let ([method-name (car args)] [method-args (cdr args)])
135                     (if (assoc method-name fields)
136                         (apply (cdr (assoc method-name fields)) method-args)
137                         (error "Método no encontrado" method-name)))]
138           [else (error "Operación no válida en el objeto" msg)])))]
139
140    ;; Variable
141    [(symbol? exp) (lookup exp env)]
```

Manejo de objetos y variables

Por último se llega a la implementación más larga del interpretador pero no menos importante. En esta sección del código, se implementa el soporte para trabajar con objetos, lo que permite definir estructuras con campos y métodos, además de permitir la manipulación de variables en el entorno. Se define cómo crear objetos con campos y cómo interactuar con estos objetos a través de mensajes (get, update, send). También se incluye la evaluación de variables y el manejo de expresiones no reconocidas.

En esta parte del código se maneja la creación y manipulación de objetos, donde los objetos se representan como listas de pares clave-valor que pueden contener tanto campos como métodos. Al crear un objeto, los valores de sus campos se evalúan y se permite interactuar con ellos a través de mensajes como ‘get’ para obtener el valor de un campo, ‘update’ para modificarlo y ‘send’ para llamar a un método con los argumentos correspondientes. Si un campo o método no existe, se lanza un error. Además, si la expresión es una variable, se busca su valor en el entorno mediante la función lookup. Si la expresión no se ajusta a ninguna de las reglas definidas, se genera un error indicando que la expresión no es reconocida. Esto permite crear objetos con comportamiento, trabajar con variables en el entorno y manejar errores de manera adecuada cuando se encuentran expresiones no válidas.

```
140     ;; Variable
141     [(symbol? exp) (lookup exp env)]
142
143     ;; Error por expresión no reconocida
144     [else (error "Expresión no reconocida" exp)]))
```

Y por último para el manejo de errores en las expresiones, al interpretador se le agregan las siguientes clausular que comparar expresiones con símbolos incluidos en el entorno. Si este no se encuentra disponible se lanza un error, y para concluir, al escribir de forma una correcta una expresión lo que hace la última línea de código es lanzar el mensaje de error “Expresión no reconocida”.