Informe Final "Proyecto Lenguaje Obliq"

Jony Melgarejo, Sebastián Ríos

***Resumen*— Existe una amplia variedad de lenguajes de programación y a través del curso de Fundamentos de Lenguajes de Programación hemos podido identificar sus diferencias. El trabajo final del curso es implementar un interpretador para un lenguaje de programación llamado Obliq, al cual se le realiza un análisis de las estructuras sintácticas, de datos y de control para implementarlas en un interpretador que se desarrollará en DrRacket. En este trabajo se hace resumen de la manera en que se implementó el lenguaje de programación Obliq que soporta programación orientada a objetos y la semántica fundamental de los lenguajes de programación como son: las definiciones léxicas y gramaticales, la semántica de los condicionales, ligaduras, procedimientos, recursión, asignación, etc.**

***Palabras Clave* — Lenguajes de programación, inducción, abstracción de datos, semántica, programación orientada a objetos**

I. INTRODUCCIÓN

UN lenguaje en informática según la RAE es un conjunto de instrucciones codificadas que una computadora interpreta o también se puede entender como un lenguaje artificial diseñado para expresar una secuencia de instrucciones que pueden ser ejecutadas por máquinas como las computadoras.

En el documento se describirá el lenguaje Obliq, el cual es un lenguaje de programación que soporta programación orientada a objetos y tiene unas estructuras de paso de parámetros por valor.

La programación orientada a objetos es un paradigma de programación que usa objetos en sus interacciones, algunas de sus características son:

- Uso de clases y jerarquía.
- Polimorfismo.
- Encapsulamiento.
- Herencia.

Jony Melgarejo y Sebastián Ríos son estudiantes de Ingeniería de Sistemas, Universidad del Valle, Cali, Colombia. (e-mail: {jony.melgarejo, sebastian.rios}@ correounivalle.edu.co)

El paso de parámetros por valor consiste en copiar el contenido de la variable para luego ser pasados a las funciones, pero las variables reales no se pasan a la función.

En el lenguaje de programación Obliq también se contara con estructuras como: condicionales, iteradores, procedimientos, etc.

II. SINTAXIS DEL LENGUAJE

La sintaxis del lenguaje Obliq se caracteriza por seis tipos de dato, lo cuales son:

- **<programa>**: Es la parte más general de la sintaxis del lenguaje Obliq, es decir, es cualquier combinación de los otros tipos de dato.
- **<expresiones>**: Es el tipo de dato que combina los tipo de dato bool-expresion, primitiva, bool-primitiva y bool-oper para crear otras variantes.
- **<bool-expresion>**: Son el tipo de dato que su resultado es un booleano.
- **<primitiva>**: Son las operaciones básicas de enteros (+| - | * | / | %) y de textos (&).
- **<bool-primitiva>**: Son las operaciones elementales de comparación de números (< | > | <= | =>) y una operación general (is) para comparar si dos expresiones son iguales.
- **<bool-oper>**: Son los operadores lógicos (not, and y or).

En los tipos de dato <bool-oper>, <bool-primitiva> y <primitiva> no se describirán detalladamente porque son las unidades básicas del lenguaje Obliq, por lo tanto se procederá a explicar las variantes de los tipo de dato <bool-expresion> y <expresion>.

El tipo de dato <bool-expresion> tiene cuatro (4) variantes a saber:

```
::= true
    <true-exp>
::= false
    <false-exp>
::= <bool-primitiva> " ( " { <expresion> }*(,) " ) "
::= <aplicar-bool>
::= <bool-oper> " ( " { <bool-expresion> }*(,) " ) "
::= <bool-oper>
```

true: <true-exp> es un valor constante que retorna true.

false: <false-exp> es un valor constante que retorna false.

bool-primitiva…: <aplicar-bool> es una variante con la cual se realizan comparaciones entre expresiones.

bool-oper…: <bool-oper> es una variante con la cual se realizan operaciones lógicas (not, and y or) entre expresiones booleanas (bool-expresion).

El tipo de dato <expresion> tiene 22 variantes:

```
::= <bool-expresion>
    <bool-exp>
::= <identificador>
    <id-exp>
::= <numero>
    <num-exp>
::= <caracter>
    <caracter-exp>
::= <cadena>
    <cadena-exp>
::= ok
    <ok-exp>
::= "var" { <identificador> = <expresion> }*
    <var-exp>
::= "let" { <identificador> = <expresion> }*
    <let-exp>
::= "letrec" { <identificador> " ( " { <iden
    <letrec-exp>
::= "set" <identificador> " := " <expresion>
    <set-exp>
::= "begin" <expresion> { <expresion> }*(;)
    <begin-exp>
::= <primitiva> " ( " { <expresion> }* " ) "
    <primapp-exp>
::= "if" <bool-expresion> "then" <expresion>
    <cond-exp>
::= "proc" " ( " { <identificador> }*(,) " )
    <proc-exp>
::= "apply" <identificador> " ( " { <expresi
    <apply-exp>
::= "meth" " ( " <identificador> " , " { <id
    <meth-exp>
::= "for" <identificador> " = " <expresion>
    <for-exp>
::= "object" " { " { <identificador> " => "
    <obj-exp>
::= "get" <identificador> " . " <identificac
    <get-exp>
```

```
::= "send" <identificador> " . " <identificador> " (
    <send-exp>
::= "update" <identificador> " . " <identificador> "
    <update-exp>
::= "clone" " ( " <identificador> { <identificador> }
    <clone-exp>
```

bool-expresion: <bool-exp> se evalúa la bool-exp con la función **evaluar-bool-exp**, con el fin de identificar la variante a la que pertenece y retornar el valor correspondiente.

identificador: <id-exp> es una secuencia de caracteres alfanuméricos que comienzan con una letra, por lo cual se le aplica la función apply-env con el ambiente anterior, en el caso de que el ambiente anterior sea vacío o que el identificador no se encuentra en el ambiente muestra un error de lo contrario retorna el valor asociado.

numero: <num-exp> son valores enteros, por lo tanto recibe y retorna números enteros.

carácter: <carácter-exp> es una variante para recibir y retornar caracteres.

cadenas: <cadena-exp> es una variante para recibir y retornar cadenas (strings).

ok: <ok-exp> es el valor vacío, es decir, retorna un string "ok".

var: <var-exp> una definición **var** introduce una colección de identificadores actualizables y sus valores. Para realizar esto se importan el tipo de dato box y sus funciones unbox y set-box! de la siguiente manera:

```
(require (only-in racket/base box))
(require (only-in racket/base box?))
(require (only-in racket/base unbox))
(require (only-in racket/base set-box!))
```

Esta importación se hace con el fin de crear una lista de celdas, en el cual se asocia a cada variable con su valor, luego esta nueva lista se asigna a id-vals con la que se evalúa el cuerpo de la expresión en un ambiente extendido.

let: <let-exp> una definición **let** introduce una colección de identificadores no actualizables y sus valores iniciales, se usa la función evaluar-rands y asocia sus valores a id-vals con la que se evalúa el cuerpo de la expresión en un ambiente extendido.

letrec: <letrec-exp> una definición **letrec** incluye una colección de procedimientos (no actualizables) recursivos, en este caso se hace un llamado a la función evaluar-expresion con el cuerpo de la expresión y un ambiente extendido recursivo que recibe como parámetros los demás argumentos de la expresión letrec.

set: <set-exp> la variante set aplica un ambiente al id y la expresion ya evaluada.

begin: <begin-exp> la expresión begin se evalúa con la función evaluar-begin, la cual determina el valor de cada expresión y retorna el valor de la última expresión.

primitiva: <primapp-exp> en esta variante se usa la función

evaluar-rands para determinar el valor de los argumentos para luego ser usados en la función aplicar-primitiva que retorna su valor correspondiente después de identificar el tipo de primitiva (+ | - | * | / |% | &).

if: <cond-exp> el condicional **if**, está compuesto por los tipos de dato **bool-expresion** y **expresión**, por lo cual primero se determina el valor del **bool-expresion** como se indicó anteriormente, si el resultado de este es true se realiza la primera expresión de lo contrario si el **elseif** es vacío se retorna el valor del else.

proc: <proc-exp> en esta variante se usa el datatype para los procedimientos, es decir, crea la clausura con los ids, el body que es la expresión a evaluar y un ambiente.

apply: <apply-exp> la expresión apply es la manera de invocar un procedimiento con los parámetros que requiere para determinar su valor correspondiente.

meth: <meth-exp> en esta variante se crea un método para un objeto, por lo que recibe los parámetros y una expresión en la que se opera con los parámetros.

for: <for-exp> es un iterador que recibe un valor inicial (expresión) y un valor final (expresión) por los que se itera y se realiza una tercera expresión que retorna un valor.

object: <obj-exp> es la representación de objetos en el lenguaje Obliq, por lo tanto se usa el datatype para los objetos, es decir, crea una lista de sus campos y un vector con las expresiones que representan los valores asociados a los campos.

get: <get-exp> la expresión get es la forma en la que se accede o consulta el valor de un campo de un objeto, por lo cual recibe un objeto y un campo.

send: <send-exp> en esta variante se realiza la invocación de un método de un objeto y se le pasa los parámetros para que sean evaluados.

update: <update-exp> la expresión update nos permite actualizar los campos de un objeto con un nuevo valor, por lo cual recibe un objeto y un campo.

clone: <clone-exp> es la manera en la que se clona un objeto lo que implica herencia múltiple, ya que un clone retorna un objeto con los campos y métodos del objeto original.

III. FUNCIONES AUXILIARES

Las funciones auxiliares son aquellas que se crearon en el transcurso del desarrollo del interpretador para el lenguaje Obliq, en este caso nosotros no consideraremos las funciones que usamos de los interpretadores de clase, es decir, se usaron algunas funciones de los interpretadores simples,

Las funciones auxiliares construidas en este interpretador son:

La función **evaluar-bool-exp** es usada para determinar el valor de verdad de expresiones tipo **bool-expresion.**

```
;; Determinar el valor de verdad de las expresiones bool-expresion
(define evaluar-bool-exp
        (lambda (test-exp env)
                (cases bool-expresion test-exp
                        (true-exp() #t)
                        (false-exp() #f)
                        (apply-bool (oper args)
                                ;; se evaluan los argumentos
                                (let ((args (evaluar-rands args env)))
                                        ;; y se llama a la funcion auxili
                                        (aplicar-prim-bool oper args)))
                        (oper-bool (oper args)
                                ;; llamado a la funcion auxiliar con el o
                                (aplicar-bool-oper oper args env)))))
```

Al aplicar el cases de **bool-expresion** se tiene un comportamiento según el tipo de variante, cuando es una variante **true-exp** retorna un valor booleano #t, cuando es una variante **false-exp** retorna un valor booleano #f. En el caso de la variante **apply-bool** se evalúan los argumentos y retorna su valor de verdad, en el caso de ser una variante **oper-bool** se llama la función **aplicar-bool-oper** con el ambiente actual.

La función **aplicar-bool-oper** es usada para las expresiones que usan los operadores lógicos and, or y not.

```
;; Evalua las expresiones en las que se usa un operador logi
(define aplicar-bool-oper
        (lambda (oper args env)
                (cases bool-oper oper
                        ;; se aplica el operador not a el pr
                        (oper-not () (not (evaluar-bool-exp
                        ;; se aplica el operador and a el pr
                        (oper-and () (and (evaluar-bool-exp
                        ;; se aplica el operador or a el pri
                        (oper-or () (or (evaluar-bool-exp (c
```

La función **evaluar-for-exp** evalúa una expresión tipo for y el resultado de las iteraciones se pone en una lista.

```
;; Determina el resultado de las iteraciones de una expresion for y las p
(define evaluar-for-exp
        (lambda (id final-value exp env)
                (cond
                        ;; si el valor ligado a id es mayor que el valor
                        [(> (apply-env env id) final-value) '()]
                        [else
                                (begin
                                        ;; en caso contrario se increment
                                        (apply-env-for-set! env id (+ 1 (
                                        ;; y se va construyendo la lista
                                        (cons (evaluar-expresion exp env)
```

La función **crear-boxes** es usada para crear una lista de celdas, en el cual se asocia a cada variable con su valor.

```
;; Permite crear boxes para los valores de las expresiones
(define crear-boxes
        (lambda (exps env)
                ;;vals recibe las expresiones evaluadas
                (let ((vals (evaluar-rands exps env)))
                        (map
                                (lambda (val) (box val))
                                        vals))))
```

La función **evaluar-begin** determina el valor de cada expresión y retorna el valor de la última expresión.

```
;; Determina el resultado de una begin-exp
(define evaluar-begin
        (lambda (exp exps env)
                (cond
                        ;; si la lista de expresiones es vacia solo
                        [(null? exps) (evaluar-expresion exp env)]
                        [else
                                (begin
                                        ;; en caso contrario se eval
                                        (evaluar-expresion exp  env)
                                        ;; el resultado de la ultima
                                        (evaluar-begin (car exps) (c
```

IV. REPRESENTACIÓN DE AMBIENTES

En el interpretador realizado para el lenguaje Obliq los ambientes se implementaron de la siguiente forma:

Se crea un datatype para ambientes con tres variantes así: ambientes vacíos, extendidos y extendidos recursivos.

```
;*************************************************
;; Definicion de tipo de dato ambiente
(define-datatype environment environment?
        ; Ambiente vacio
        (empty-env-record)
        ; Ambiente extendido
        (extended-env-record
                (syms (list-of symbol?))
                (vals vector?)
                (env environment?))
        ; Ambiente recursivo extendido
        (recursively-ext-env-record
                (proc-names (list-of symbol?))
                (proc-ids (list-of (list-of symbol?)))
                (proc-bodies (list-of expresion?))
                (env environment?)))
```

Las funciones auxiliares asociadas a los ambientes son: **empty-env**, **extend-env**, **extend-env-recursively**, **apply-env** y **apply-env-for-set!** para la creación de cada una de las variantes de ambiente y aplicar un ambiente.

```
;; Funciones Auxiliares de Ambientes
;;
;; Crea un ambiente vacio
(define empty-env
  (lambda ()
    (empty-env-record)))
;; Crea un ambiente extendido
(define extend-env
  (lambda (syms vals env)
    (extended-env-record syms (list->vector vals)
;; Crea un ambiente recursivo extendido (letrec-e
(define extend-env-recursively
  (lambda (proc-names proc-ids proc-bodies env)
    (recursively-ext-env-record proc-names proc-i
;; Aplica un ambiente a un símbolo retorna su val
(define apply-env
        (lambda (env sym)
                (cases environment env
                        ; Caso ambiente vacio
                        (empty-env-record ()
                                ; Error cuando el
                                (eopl:error 'appl
                        ; Caso ambiente extendido
                        (extended-env-record (sym
```

```
(define apply-env-for-set!
        (lambda (env sym n-val)
                (cases environment env
                ; Caso ambiente vacio
                (empty-env-record ()
                        ; Error cuando el ambiente e
                        (eopl:error 'apply-env "No b
                ; Caso ambiente extendido
                (extended-env-record (syms vals env)
                        ; Busca la posicion del simb
                        (let ((pos (rib-find-positio
                                (if (number? pos)
                                        ; Busca el v
                                        (let ((val
                                                ;; s
                                                (if


                                        ;; llamado r
                                        (apply-env-f
                ; No aplica para el Caso ambiente re
                (else (eopl:error 'set "No es posibl
```

Las funciones **rib-find-position**, **list-find-position** y **list-index** se usan en las funciones de aplicar ambiente, con estas funciones es posible encontrar la posición de un símbolo en la lista de símbolos de un ambiente.

```
(define rib-find-position
        (lambda (sym los)
        (list-find-position sym los)))
;;
(define list-find-position
        (lambda (sym los)
        (list-index (lambda (sym1) (eqv?
;;
(define list-index
        (lambda (pred ls)
                (cond
                        ((null? ls) #f)
                        ((pred (car ls))
                        (else (let ((list
                                (if (numb
                                        (
                                        #
```

V. REPRESENTACIÓN DE OBJETOS

La representación de objetos (object-exp) fue realizada con el siguiente datatype:

```
;*************************************************
;;
;; Objetos
;;
;*************************************************
;;
;; Definicion del tipo de dato para la repre
(define-datatype objeto objeto?
        (un-objeto
                ;; una lista de simbolos con
                (fields (list-of symbol?))
                ;; un vector de expresiones
                (exps vector?)))
;;
```

Las funciones auxiliares al tipo de dato objeto son: el tipo de dato método (meth-exp) con su función aplicar-método, además de las funciones para consultar (get-exp) y modificar (update-exp) los campos del objeto.

```
;;
(define-datatype metodo metodo?
        (un-metodo
                ;; lista de simbolos que
                (ids (list-of symbol?))
                ;; exp que constituye el
                (body expresion?)))
;; Determinar el resultado de la aplicac
(define aplicar-metodo
        (lambda (obj meth args env)
                (cases objeto obj
                        (un-objeto (fiel
                                ;; busca
                                (let ((p
```

```
(define valor-atributo
        (lambda (sym obj env)
                (cases objeto obj
                        (un-objeto
                                ;;
                                (l
;; Actualiza el valor de un atribu
(define actualizar-atributo
        (lambda (obj id new-value)
                (cases objeto obj
                        (un-objeto
```

VI. PRUEBAS

Las pruebas realizadas en el interpretador fueron las siguientes:

Constantes

```
$command-> true
#t
$command-> false
#f
$command-> var x=1 in x end
1
$command-> 3
3
$command-> 'c'
|'c'|
$command-> "cadena"
"cadena"
$command-> ok
"ok"
```

Operadores de texto, booleanos y enteros

```
$command-> & ("Uno " "Dos " "Tres")
"Uno Dos Tres"
$command-> not (false)
#t
$command-> and (false,true)
#f
$command-> or (false,true)
#t
$command-> +(2 3 5)
10
$command-> -(2 3 5)
4
$command-> *(2 3 5)
30
$command-> /(2 3 5)
10/3
$command-> %(2 3)
2
$command-> <(3,5)
#t
$command-> >(10,5)
#t
$command-> <=(1,1)
#t
$command-> >=(4,4)
#t
$command-> is('a',"A")
#f
$command-> is(2,2)
#t
```

Estructuras de Control

```
$command-> var x=2 in +(x 3) end
5
$command-> let y=5 in *(y 2) end
10
$command-> letrec Fact(n) = if is(n,0) then 1 else *(n
apply Fact(-(n 1))) end in apply Fact(5) end
120
$command-> let funcion = proc (x,y) *(x y) end in apply
funcion(2,3) end
6
```

Creación de Objetos y Selección de Campos

```
$command-> let animal=object {patas=>4 edad=>2} in get
animal.patas end
4
```

Clonación

```
$command-> let animal=object {patas=>4 edad=>2} in var
perro= clone(animal) in get perro.patas end end
4
```

Secuenciación

```
$command-> begin var a=2 in a end *(5 2); ok end
"ok"
```

Condicional

```
$command-> if <(2,8) then /(10 3) else *(10 3) end
10/3
```

Iterador

```
$command-> let x = 1 in for i = 2 to 5 do +(x 1) end end
(2 2 2 2)
```

VII. CONCLUSIONES

Mediante la realización de este trabajo, se logró:

- Afianzar los conocimientos que se fueron adquiriendo en el desarrollo del curso como las definiciones léxicas y gramaticales, semántica fundamental de lenguajes de programación, etc.
- Comprender las diferencias entre los lenguajes de programación procedimentales, funcionales y orientados a objetos.
- Reconocer la importancia de cómo funcionan los lenguajes de programación internamente, porque usualmente somos un usuario que utiliza el lenguaje sin importar lo que existe detrás de él.
- Desarrollar habilidades de trabajo en equipo.

BIBLIOGRAFÍA

Real Academia Española [online]. España, 2017 Disponible en: http://dle.rae.es/?id=N7BnIFO

R.P.Jorge Iván, *"Programación Orientada a Objetos en Lenguajes no Orientados a Objetos: C una Experiencia"* [online]. Colombia: Universidad Tecnologica de Pereira, 2005 Disponible en: http://www.redalyc.org/pdf/849/84911948018.pdf

D.S Carlos Andrés, "*Notas de Clase*" [online]. Colombia: Universidad del Valle, 2017.