PROYECTO FINAL

FLP

Kevin Andres Bejarano - 2067678
Juan David Gutierrez Florez - 2060104
Johan Sebastián Laverde pineda - 2266278
Johan Sebastian Acosta Restrepo 2380393

2024 II
Universidad del valle
Sede Tuluá

```
;; Especificación léxica: Define las reglas para identificar y clasificar los tokens del programa
(define especificacion-lexica
  '((espacio-blanco (whitespace) skip)  ;; Saltar espacios en blanco
    (comentario ("%" (arbno (not #\newline))) skip)  ;; Saltar comentarios que comienzan con %
    (identificador (letter (arbno (or letter digit "_"))) symbol)  ;; Identificadores
    (numero (digit (arbno digit)) number)  ;; Números
    (numero ("-" digit (arbno digit)) number)))  ;; Números negativos
```

Esta función llamada especificacion-lexica define cómo se identifican y clasifican los tokens del lenguaje, como identificadores, números, espacios en blanco y comentarios.

```
;; Especificación gramatical: Define la estructura de las expresiones y otros elementos del lenguaje
(define especificacion-gramatical
  '((programa (expresion) a-program)  ;; Un programa es una expresión
    (expresion (numero) lit-exp)  ;; Una expresión puede ser un literal numérico
    (expresion (identificador) var-exp)  ;; O un identificador
    (expresion ("true") true-exp)  ;; O el valor booleano true
    (expresion ("false") false-exp)  ;; O el valor booleano false
    (expresion ("if" expresion "then" expresion "else" expresion) if-exp)  ;; Expresión if-then-else
    (expresion ("let" (arbno identificador "=" expresion) "in" expresion) let-exp)  ;; Expresión let
    (expresion ("proc" "(" (separated-list identificador ",") ")" expresion) proc-exp)  ;; Expresión proc
    (expresion ("(" expresion (arbno expresion) ")") app-exp)  ;; Aplicación de una expresión
    (expresion ("begin" expresion (arbno ";" expresion) "end") begin-exp)  ;; Expresión begin-end
    (expresion ("set" identificador "=" expresion) set-exp)  ;; Expresión set
    (expresion (primitiva "(" (separated-list expresion ",") ")") prim-exp)  ;; Primitivas
    (primitiva ("+") sum-prim)  ;; Suma
    (primitiva ("-") minus-prim)  ;; Resta
    (primitiva ("*") mult-prim)  ;; Multiplicación
    (primitiva ("/") div-prim)  ;; División
    (primitiva (">") mayor-prim)  ;; Mayor que
    (primitiva ("<") menor-prim)  ;; Menor que
    (primitiva ("==") igual-prim)  ;; Igualdad
    (campo (identificador "=>" expresion) campo-exp)  ;; Campo para objetos
    (expresion ("object" "{" (arbno campo) "}") object-exp)  ;; Expresión object
    (expresion ("send" identificador "." identificador "(" (separated-list expresion ",") ")") send-exp)  ;; Envío de mensaje
    (expresion ("clone" "(" (separated-list identificador ",") ")") clone-exp)))  ;; Clonación de objetos
```

Las especificaciones gramaticales describen las reglas para construir expresiones y estructuras del lenguaje, como literales, variables, expresiones condicionales, procedimientos y operaciones primitivas.

```
;; Ambientes
(define-datatype ambiente ambiente?
  (ambiente-vacio)  ;; Ambiente vacío
  (ambiente-extendido-ref  ;; Ambiente extendido
   (ids (list-of symbol?))  ;; Lista de identificadores
   (valores vector?)  ;; Vector de valores
   (old-env ambiente?)))  ;; Ambiente anterior

;; Crear un nuevo ambiente extendido
(define ambiente-extendido
  (lambda (ids valores old-env)
    (ambiente-extendido-ref ids (list->vector valores) old-env)))
```

Se crea un ambiente por referencia y se define, se crea un nuevo ambiente extendido que asocia identificadores con valores y enlaza al ambiente previo.

```
;; Aplicar el ambiente para obtener el valor de una variable
(define apply-env
  (lambda (env var)
    (cases ambiente env
      (ambiente-vacio () (eopl:error "Variable no encontrada: " var))  ;; Error si la variable no se encuentra
      (ambiente-extendido-ref (ids vals old-env)
        (let loop ((ids ids) (index 0))
          (cond
            [(null? ids) (apply-env old-env var)]  ;; Buscar en el ambiente anterior si no se encuentra en el actual
            [(equal? (car ids) var) (vector-ref vals index)]  ;; Retornar el valor si se encuentra la variable
            [else (loop (cdr ids) (+ index 1))]))))))  ;; Continuar buscando en la lista
```

Es el encargado de buscar el valor asociado a un identificador en un ambiente. Si no se encuentra en el ambiente actual, busca en ambientes anteriores.

```
;; Definición de Closure
(define-datatype procval procval?
  (closure (params (list-of symbol?))  ;; Lista de parámetros
           (cuerpo expresion?)  ;; Cuerpo de la función
           (ambiente ambiente?)))  ;; Ambiente donde se definió la función
```

Representa una función con sus parámetros, cuerpo y el ambiente donde fue definida, permitiendo ejecutar la función en contextos diferentes manteniendo acceso a las variables originales.

```
;; Funciones auxiliares
;; Crear un objeto a partir de una lista de campos
(define crear-objeto
  (lambda (campos)
    (foldl (lambda (campo obj)
             (cases campo
               (campo-exp (id val)  ;; Agregar cada campo al objeto
                          (cons (cons id val) obj))))
           '()  ;; Objeto inicial vacío
           campos)))
```

La función crear-objeto es una parte crucial del intérprete que se encarga de construir representaciones internas de objetos a partir de una lista de **campos,** asociando identificadores a los valores.

```
;; Invocar un método en un objeto
(define invocar-metodo
  (lambda (obj metodo args)
    (let ((metodo-funcion (assoc metodo obj)))  ;; Buscar el método en el objeto
      (if metodo-funcion
          (apply (cdr metodo-funcion) args)  ;; Invocar el método si se encuentra
          (eopl:error "Método no encontrado")))))  ;; Error si el método no se encuentra
```

Busca e invoca un método en un objeto, pasando los argumentos correspondientes. El objetivo es buscar un método en el objeto, ejecutarlo con los argumentos proporcionados y devolver el resultado.

```
;; Clonar objetos
(define clonar-objetos
  (lambda (objs)
    (map (lambda (obj)
           (foldl (lambda (campo copia)
                    (cons campo copia))  ;; Crear una copia de cada campo
                  '()
                  obj))
         objs)))
```

Crea una copia de objetos, clonando sus campos. crea clones de una lista de objetos, donde cada objeto es una lista de pares clave-valor (campos). Utiliza las funciones map y fold1 para construir una copia independiente de cada objeto.

```
;; Evaluar expresiones
(define evaluar-expresion
  (lambda (exp env)
    (cases expresion exp
      (lit-exp (dato) dato)  ;; Retornar literales tal como están
      (var-exp (id) (apply-env env id))  ;; Buscar el valor de la variable en el ambiente
      (true-exp () #t)  ;; Retornar true
      (false-exp () #f)  ;; Retornar false
      (if-exp (condicion exp-then exp-else)
              (if (evaluar-expresion condicion env)  ;; Evaluar la condición
                  (evaluar-expresion exp-then env)  ;; Evaluar el "then" si es verdadero
                  (evaluar-expresion exp-else env)))  ;; Evaluar el "else" si es falso
      (let-exp (ids valores cuerpo)
               (let ((evaluados (map (lambda (v) (evaluar-expresion v env)) valores)))  ;; Evaluar valores de let
                 (evaluar-expresion cuerpo (ambiente-extendido ids evaluados env))))  ;; Evaluar el cuerpo en el nuevo ambiente
      (proc-exp (params cuerpo)
                (closure params cuerpo env))  ;; Crear una clausura
      (app-exp (proc args)
               (let ((evaluados (map (lambda (arg) (evaluar-expresion arg env)) args))  ;; Evaluar argumentos
                     (procedimiento (evaluar-expresion proc env)))  ;; Evaluar el procedimiento
                 (cases procval procedimiento
                   (closure (params cuerpo old-env)
                            (if (= (length params) (length evaluados))  ;; Verificar cantidad de argumentos
                                (evaluar-expresion cuerpo (ambiente-extendido params evaluados old-env))  ;; Evaluar cuerpo de la clausura
                                (eopl:error "Número incorrecto de argumentos"))))))  ;; Error si los argumentos no coinciden
      (begin-exp (primera resto)
                 (let loop ((expresiones (cons primera resto)) (resultado #f))
                   (if (null? expresiones)
                       resultado  ;; Retornar el resultado de la última expresión
                       (loop (cdr expresiones) (evaluar-expresion (car expresiones) env)))))  ;; Evaluar todas las expresiones
      (set-exp (id val)
               (let ((evaluado (evaluar-expresion val env)))  ;; Evaluar el valor
                 (setref! (apply-env-ref env id) evaluado)  ;; Asignar el valor a la variable
                 1))
           (object-exp (campos)
                  (crear-objeto (map (lambda (campo)
                                       (cases campo
                                         (campo-exp (id val) (cons id (evaluar-expresion val env)))))
                                     campos)))  ;; Crear un objeto a partir de los campos evaluados
      (prim-exp (prim args)
                (evaluar-primitiva prim (map (lambda (arg) (evaluar-expresion arg env)) args)))  ;; Evaluar una expresión primitiva
      (send-exp (obj metodo args)
                (let ((objeto (evaluar-expresion obj env))  ;; Evaluar el objeto
                      (args-evaluados (map (lambda (arg) (evaluar-expresion arg env)) args)))  ;; Evaluar argumentos del método
                  (invocar-metodo objeto metodo args-evaluados)))  ;; Invocar el método en el objeto
      (clone-exp (objs)
                 (clonar-objetos (map (lambda (obj) (evaluar-expresion obj env)) objs)))  ;; Clonar los objetos
      )))
```

Esta función un poco más compacta se encarga de evaluar las diferentes expresiones del lenguaje:

- Literales, variables, y valores booleanos (`lit-exp`, `var-exp`, `true-exp`, `false-exp`).
- Condicionales (`if-exp`), procedimientos (`proc-exp`), y aplicaciones (`app-exp`).
- Bloques de instrucciones (`begin-exp`) y asignaciones (`set-exp`).
- Creación y manipulación de objetos (`object-exp`, `send-exp`, `clone-exp`).
- Operaciones primitivas como suma, resta, comparación, etc. (`prim-exp`).

```
;; Función auxiliar para evaluar primitivas
(define evaluar-primitiva
  (lambda (prim args)
    (cases primitiva prim
      (sum-prim () (apply + args))  ;; Suma de los argumentos
      (minus-prim () (apply - args))  ;; Resta de los argumentos
      (mult-prim () (apply * args))  ;; Multiplicación de los argumentos
      (div-prim () (apply / args))  ;; División de los argumentos
      (mayor-prim () (> (car args) (cadr args)))  ;; Comparación de mayor que
      (menor-prim () (< (car args) (cadr args)))  ;; Comparación de menor que
      (igual-prim () (= (car args) (cadr args)))  ;; Comparación de igualdad
      (else (eopl:error "Primitiva desconocida: " prim)))))  ;; Error para primitivas desconocidas
```

La evaluacion de las primitivas se encargan de realizar operaciones primitivas como lo dice la funcion como por ejemplo suma, resta, comparación, sobre argumentos evaluados.

```
;; Evaluar programa
(define evaluar-programa
  (lambda (pgm)
    (cases programa pgm
      (a-program (exp) (evaluar-expresion exp ambiente-vacio)))))  ;; Evaluar el programa en un ambiente vacío
```

Se encarga de evalúar el programa completo a partir de una expresión inicial, usando el ambiente vacío.

```
;; Crear analizador léxico y sintáctico
(define scan&parse
  (sllgen:make-string-parser especificacion-lexica especificacion-gramatical))  ;; Crear analizador léxico y sintáctico
```

Crea un analizador léxico y sintáctico que convierte cadenas de texto en estructuras del lenguaje para asi ser interpretado..

```
;; Interpretador interactivo
(define interpretador
  (sllgen:make-rep-loop "-->" evaluar-programa
                        (sllgen:make-stream-parser especificacion-lexica especificacion-gramatical)))  ;; Crear un bucle de lectura-evaluación-imprimir

(provide (all-defined-out))  ;; Proveer todas las definiciones
```

Implementa un bucle interactivo de lectura, evaluación e impresión, permitiendo interpretar código Obliq en tiempo real.

Tipos de datos definidos

`campo`: Representa un campo dentro de un objeto, con un identificador y un valor.

`ambiente`: Representa un ambiente, que puede ser vacío o extendido.

`procval`: Representa un procedimiento (función), almacenando sus parámetros, cuerpo, y ambiente.