

## TALLER 2



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

## Detallado del código en fragmentos

```
(define especificacion-lexica
  '(
    ;; Patrones básicos
    (espacio-blanco (whitespace) skip)
    (comentario ("% (arbno (not #\newline))) skip)
    (identificador (letter (arbno (or letter digit "?" "$"))) symbol)
    (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)
  ))
```

Este fragmento define la especificación léxica para el lenguaje, es decir, cómo identificar y clasificar las distintas cadenas de caracteres que conforman un programa válido

**comentario:** Ignora los comentarios que comienzan con %.

**identificador:** Define nombres de variables o símbolos, compuestos por letras, dígitos, ? o \$.

**número:** Define números enteros y decimales, positivos y negativos.

```
(define especificacion-gramatical
  '(
    (programa (expresion) a-program)
    (expresion (numero) lit-exp)
    (expresion (identificador) var-exp)
    ;; Agregamos la gramática de los condicionales y las ligaduras
    (expresion ("true") true-exp)
    (expresion ("false") false-exp)
    (expresion ("if" expresion "then" expresion "else" expresion) if-exp)
    ;; gramatica cond
    (expresion ("cond" (arbno expresion "==" expresion) "else" "==" expresion "end") cond-exp)
    ;; ligaduras locales
    (expresion ("let" (arbno identificador "=" expresion) "in" expresion) let-exp)
    ;; Fin de condicionales y ligaduras
  ))
```

Este fragmento define la especificación gramatical del lenguaje, que describe las reglas sintácticas para formar expresiones válidas.

**programa:** Es la estructura principal que contiene una expresión.

**expresion:** Puede ser un número, un identificador, o estructuras como condicionales (if, cond) o ligaduras locales (let).

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

Genera automáticamente los tipos de datos necesarios para representar los tokens y las estructuras gramaticales definidas.

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

Esta función evalúa un programa completo.

**evaluar-programa:** Evalúa un programa completo, que se supone es una expresión (de acuerdo con la gramática).

**evaluar-expresion:** Evalúa expresiones de acuerdo con su tipo:

**Literales:** Devuelven su valor directamente.

**Variables:** Se buscan en el ambiente actual.

**Condicionales:** Si la condición se evalúa a verdadero, se ejecuta la primera rama; si no, se evalúa la segunda.

**Operaciones primitivas:** Se evalúan las operaciones como suma, resta, etc.

**Listas:** Se maneja la creación y manipulación de listas con cons, first, rest, etc.

```

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

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

;;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)
    )
  )
)

```

Define el concepto de ambiente, que asocia identificadores (variables) con sus valores. mantienen las variables y sus valores. Son estructuras que asocian identificadores a valores y pueden ser extendidos con nuevas variables.

**ambiente-vacio:** No contiene asociaciones.

**ambiente-extendido-ref:** Asocia una lista de identificadores (lids) con valores (lvalue), extendiendo un ambiente previo (old-env).

**ambiente-extendido:** Crea un nuevo ambiente extendido con un conjunto de variables y sus valores.

**apply-env:** Se utiliza para buscar el valor de una variable en el ambiente.

```
;;Evaluar expresion
(define evaluar-expresion
  (lambda (exp amb)
    (cases expresion exp
      ;; Casos existentes
      (lit-exp (dato) dato) ; Caso para literales
      (var-exp (id) (apply-env amb id)) ; Caso para variables
      (true-exp () #true) ; Caso para booleanos
      (false-exp () #false)

      ;; Caso para lista vacía
      (list-empty-exp ()
        '()) ; Devuelve la lista vacía directamente

      ;; Caso para lista no vacía
      (cons-exp (exp1 exp2)
        (let ([first-val (evaluar-expresion exp1 amb)]
              [rest-val (evaluar-expresion exp2 amb)])
          (if (list? rest-val)
              (cons first-val rest-val)
              (eopl:error "El segundo argumento de cons no es una lista: " rest-val))))

      ;; Otros casos existentes...
      (prim-exp (prim args)
        (let ([lista-numeros (map (lambda (x) (evaluar-expresion x amb)) args)]
              (evaluar-primitiva prim lista-numeros)))

      (if-exp (condicion hace-verdadero hace-falso)
        (if (evaluar-expresion condicion amb)
            (evaluar-expresion hace-verdadero amb)
            (evaluar-expresion hace-falso amb))))))
```

Evalúa una expresión en el contexto de un ambiente.

**lit-exp:** Retorna directamente el valor de un literal (número).

**var-exp:** Busca el valor asociado a una variable en el ambiente.

**list-empty-exp y cons-exp:** Trabajan con listas.

**if-exp:** Evalúa condicionales if.

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

Define un ambiente inicial con las siguientes variables y valores:

x = 4, y = 2, z = 5

a = 4, b = 5, c = 6

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

**Referencias:** El código soporta la manipulación de referencias (en realidad, direcciones de memoria).

**Asignación:** Usa setref! para modificar el valor de una variable referenciada en el ambiente.

```
;; Crear el analizador léxico y sintáctico
(define scan&parse
  (sllgen:make-string-parser especificacion-lexica especificacion-gramatical))
;;Interpretador
(define interpretador
  (sllgen:make-rep-loop "-->" evaluar-programa
    (sllgen:make-stream-parser
      especificacion-lexica especificacion-gramatical)))
```

**scan&parse:** Usa la especificación léxica y gramatical para crear un analizador léxico y sintáctico, generando una representación interna del programa a partir del código fuente.

## Explicacion sobre el codigo modificado

### \*Especificacion gramatical:

De este punto se añadió las reglas para listas como “empty” o “cons” para resolver el punto 1 del taller con las siguientes definiciones “empty” para listas vacías (list-empty-exp), “cons” para listas no vacías (cons-exp) y las operaciones adicionales como: length, first, rest y nth.



### \*Evaluacion de expresiones:

En la función evaluar-expresión se añadieron casos para soportar list.empty-exp, cons-exp y otras operaciones de las listas.

**\*Silgen:** Se modificó la parte de make-stream-parser a make-string-parser con el fin de que reconociera palabras clave, porque anteriormente al usar la función scan&parse no reconocía a empty y otras palabras clave por lo que no podía construir las estructuras.

Cabe añadir que para el punto 2 también realicé el mismo procedimiento y se cambiaron los mismos fragmentos de código como en el punto 1.

## Explicacion para los tests:

 Pruebas_Taller2_punto1	30/11/2024 2:28 p. m.	Racket Document	2 KB
 Pruebas_Taller2_punto2	30/11/2024 2:53 p. m.	Racket Document	3 KB

Para el proceso de pruebas se generaron dos archivos para el punto 1 y el punto dos con el fin de tener mayor orden en el código del proyecto.

```
(require rackunit "5.InterpretadorAsignacion.rkt") ; Cambia el nombre si tu archivo tiene otro
```

Para ambos archivos se utilizó rackunit para el proceso de importación del intérprete.

```

; Test 1: Lista vacía
(define exp1
  (scan&parse
    "empty"
  ))
(define expected-exp1
  '())
(equal? (evaluar-programa exp1) expected-exp1)

; Test 2: Construcción de listas
(define exp2
  (scan&parse
    "cons (1 cons (2 cons (3 empty)))"
  ))
(define expected-exp2
  '(1 2 3)
)
(check-equal? (evaluar-programa exp2) expected-exp2)

; Test 3: Longitud de listas
(define exp3
  (scan&parse
    "length (cons(1 cons (2 cons (3 empty))))"
  ))
(define expected-exp3
  3
)
(check-equal? (evaluar-programa exp3) expected-exp3)

; Test 4: Primer elemento
(define exp4
  (scan&parse
    "first (cons (1 cons (2 cons (3 empty))))"
  ))

```

Para el test1 se generaron 8 pruebas en las que se puso en uso las listas vacías, construcción de listas, longitud, extracción del primer elemento, extracción del resto del elemento y elemento en posición n.

Además, de los 8 test se modificaron 3 para que retornaran error al momento de acceder a una lista vacía y errores al acceder a un índice fuera del elemento.

```

; Test 7: Error al acceder a una lista vacía
(define exp7
  (scan&parse
    "first (empty)"
  ))
(check-exn exn:fail?
  (lambda () (evaluar-programa exp7)))

; Test 8: Error al acceder a índice fuera de rango
(define exp8
  (scan&parse
    "nth (cons (1 cons (2 cons (3 empty))) 5)"
  ))
(check-exn exn:fail?
  (lambda () (evaluar-programa exp8)))

```



Para el test 2 se crearon 7 pruebas de las cuales se tomaron en cuenta casos como el cond con multiples condiciones, el caso de cond con solo else, caso donde cond evalúa el primer valor verdadero y casos donde ninguna opcion es verdadera.

```
; Test 1: Caso basico de cond con multiples condiciones
(define expl
  (scan&parse
    "let x = 2 in
      cond
        ==(x,1) ==> 1
        ==(x,2) ==> 2
        ==(x,3) ==> 4
      else ==> 9
    end"
  ))
(define expected-expl
  2 ; Se evalúa `(x 2)` porque x es igual a 2
)
(check-equal? (evaluar-programa expl) expected-expl)

; Test 2: Caso de cond con solo else
(define exp2
  (scan&parse
    "let x = 2 in
      cond
        else ==> 9
      end"
  ))
(define expected-exp2
  9 ; Solo está else, así que retorna 9
)
(check-equal? (evaluar-programa exp2) expected-exp2)

; Test 3: Caso donde cond evalúa el primer valor verdadero
(define exp3
  (scan&parse
    "let x = 3 in
      cond
        ==(x,1) ==> 1
      end"
  ))
```

Ademas como en el punto 1 tambien se creo un test con error para el caso de no finalizar bien la gramática.

```
; ; Test 7: Error al no finalizar bien la gramatica
; (define exp7
;   (scan&parse
;     "let x = 10 in
;       cond
;         ==(x,10) ==> 5
;         ==(x,20) ==> 15
;       end"
;   ))
; (check-exn exn:fail?
;   (lambda () (evaluar-programa exp7)))
```