# Informe de Proyecto Final FLP.


Integrantes:

Juan David Camacho 2266292

Wilson Andres Martinez 2266319

Juan Gabriel Paredes 2266183


Materia:

Fundamentos de Interpretación y Compilación de Lenguajes.

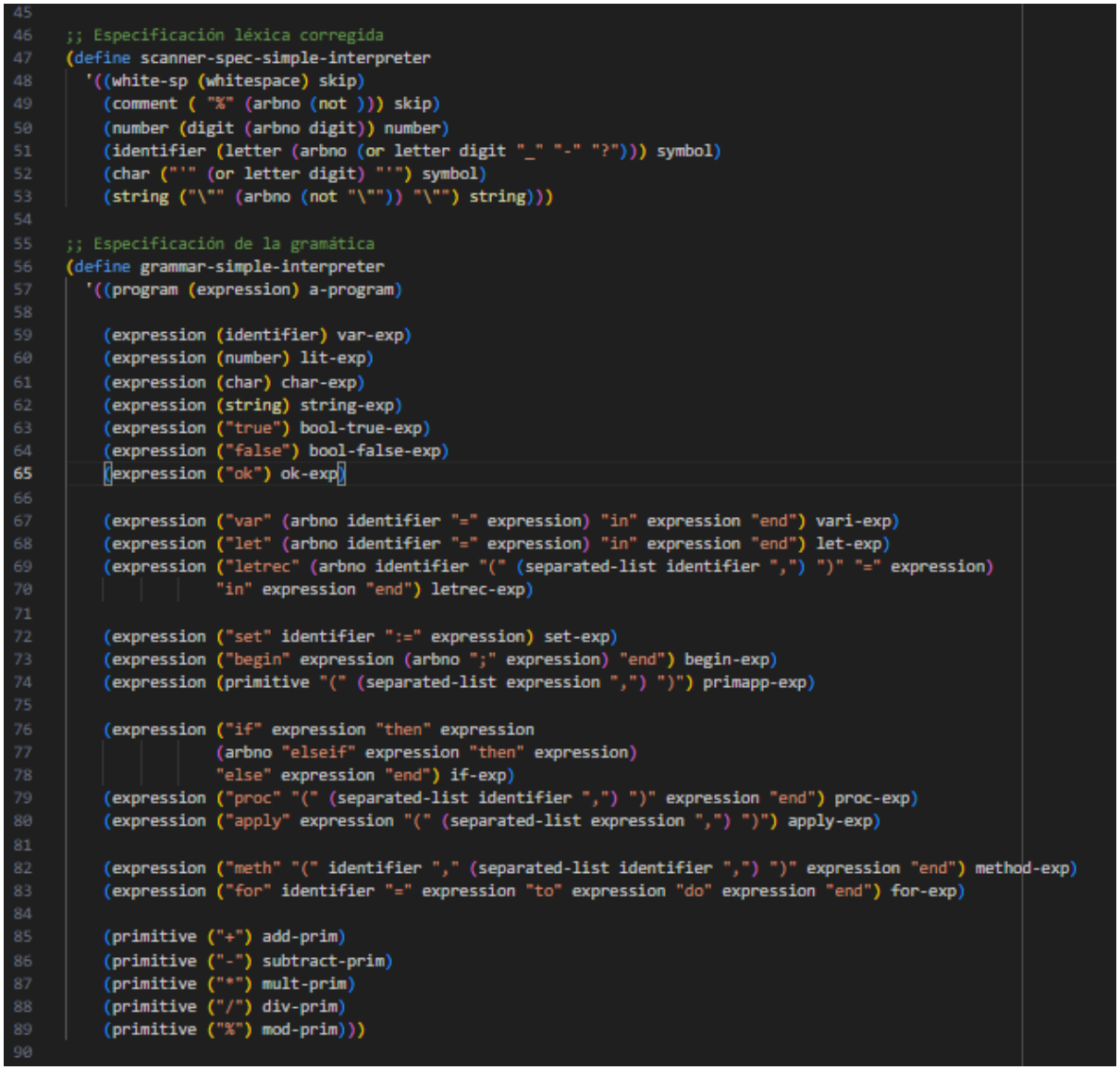
Docente:

Carlos Andres Delgado.


Universidad del Valle

Valle del Cauca – Tuluá.

Presentación de todo el programa.



```
;; Especificación léxica corregida
(define scanner-spec-simple-interpreter
  '((white-sp (whitespace) skip)
    (comment ( "%" (arbno (not )) skip)
    (number (digit (arbno digit)) number)
    (identifier (letter (arbno (or letter digit "_" "-" "?"))) symbol)
    (char ("'" (or letter digit) "'") symbol)
    (string ("\"" (arbno (not "\"")) "\"") string)))

;; Especificación de la gramática
(define grammar-simple-interpreter
  '((program (expression) a-program)

    (expression (identifier) var-exp)
    (expression (number) lit-exp)
    (expression (char) char-exp)
    (expression (string) string-exp)
    (expression ("true") bool-true-exp)
    (expression ("false") bool-false-exp)
    (expression ("ok") ok-exp)

    (expression ("var" (arbno identifier "=" expression) "in" expression "end") vari-exp)
    (expression ("let" (arbno identifier "=" expression) "in" expression "end") let-exp)
    (expression ("letrec" (arbno identifier "(" (separated-list identifier ",") ")" "=" expression)
                "in" expression "end") letrec-exp)

    (expression ("set" identifier ":=" expression) set-exp)
    (expression ("begin" expression (arbno ";" expression) "end") begin-exp)
    (expression (primitive "(" (separated-list expression ",") ")") primapp-exp)

    (expression ("if" expression "then" expression
                (arbno "elseif" expression "then" expression)
                "else" expression "end") if-exp)
    (expression ("proc" "(" (separated-list identifier ",") ")" expression "end") proc-exp)
    (expression ("apply" expression "(" (separated-list expression ",") ")") apply-exp)

    (expression ("meth" "(" identifier "," (separated-list identifier ",") ")" expression "end") method-exp)
    (expression ("for" identifier "=" expression "to" expression "do" expression "end") for-exp)

    (primitive ("+") add-prim)
    (primitive ("-") subtract-prim)
    (primitive ("*") mult-prim)
    (primitive ("/") div-prim)
    (primitive ("%") mod-prim)))
```

Aquí implementamos la especificación léxica y tambien la especificación de la gramática, de la cual cada parte tiene su implementación propuesta en el proyecto para la realizacion del proyecto de todad esta gramática se puede llegar a observar que no se implementó todos los casos hasta llegar a los metodos. Cada expresion esta hecha a ejemplos de implementación de interpretadores que fueron expuestos en clase.

```
;; Funciones nuevas añadidas
(define show-the-datatypes
  (lambda () (sllgen:list-define-datatypes scanner-spec-simple-interpreter grammar-simple-interpreter)))

(define just-scan (sllgen:make-string-scanner scanner-spec-simple-interpreter grammar-simple-interpreter))

;; Crea tipo de datos
(sllgen:make-define-datatypes scanner-spec-simple-interpreter grammar-simple-interpreter)

;; Análisis léxico y sintáctico integrados
(define scan&parse
  (sllgen:make-string-parser scanner-spec-simple-interpreter grammar-simple-interpreter))
```

El uso de los sllgen están propuestos por su uso en cada parte del proyecto.

-El make-define-datatype nos ayuda a crear nuestros propios datatype.

-El make-string-parser nos ayuda en leer nuestras cadenas de codigo.

-Los otros dos sllgen nos pueden ayudar a entender la función de front-end con la ayuda de leer cadenas y nos da la muestra de datatype.

```
;; Definicion del entorno
(define empty-env '())

(define (extend-env ids vals env)
  (if (null? ids)
      env
      (cons (cons (car ids) (car vals))
            (extend-env (cdr ids) (cdr vals) env))))

(define (apply-env env id)
  (cond
    ((null? env) (my-error (string-append "Unbound identifier: " (symbol->string id))))
    ((eq? id (car (car env))) (cdr (car env)))
    (else (apply-env (cdr env) id))))
```

Estas tres definiciones nos ayudan a emplear un entorno o (environments) la primera crea un entorno vacío, es decir no contiene más que una lista vacía, la definición de extend-env nos a ver cómo se puede extender un entorno para un vacío de los mismos entornos de entrada con los ids de identificadores y los vals de los valores correspondientes.

La definición de apply-env nos ayuda a buscar un identificador en el entorno env devolviendo un valor. Con estas funciones se puede tener claro sus entornos.

```
;; Función auxiliar para evaluar expresiones
(define eval-rands
  (lambda (rands env)
    (map (lambda (x) (eval-expression x env)) rands)))

;; Intérprete
(define eval-program
  (lambda (pgm)
    (cases program pgm
      (a-program (body)
        (eval-expression body (init-env))))))
```

Estas funciones nos ayudan a crear por inicio es evaluar los operadores de una expresión y si nos vamos a eval-program nos da inicialización a nuestra propia gramática.

```
;; Ambiente Inicial
(define init-env
  (lambda ()
    empty-env))

;;  aplicar primitivas
(define apply-primitive
  (lambda (prim args)
    (cases primitive prim
      (add-prim  () (+ (car args)  (cadr args)))
      (subtract-prim () (- (car args) (cadr args)))
      (mult-prim  () (* (car args) (cadr args)))
      (div-prim  () (/ (car args) (cadr args)))
      (mod-prim  () (modulo (car args) (cadr args))))))
```

Se crea un principal ambiente de todos los ingresos de los interpretadores, lo cual el ambiente de caso base es vacío.

Al aplicar las primitivas nos quiere dar la funcionalidad de que las operaciones son las más básicas y nos puede dar el resultado de las primitivas.

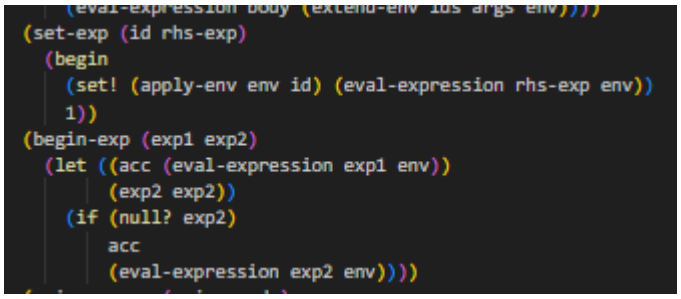
```
;; Evaluador de expresiones
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (var-exp (id)
        (apply-env env id))
      (lit-exp (datum)
        datum)
      (char-exp (char)
        char)
      (string-exp (string)
        string)
      (bool-true-exp ()
        'true)
      (bool-false-exp ()
        'false)
      (ok-exp ()
        'ok)
```

Esta función de evaluar expresión nos trae todos los casos de la gramática con los propios argumentos específicos de cada uno, estos son unos de los principales antes de entrar a la evolución de variables mutables e inmutables.

```
(vari-exp (ids rands body)
  (let ((args (eval-rands rands env)))
    (eval-expression body (extend-env ids args env))))
(let-exp (ids rands body)
  (let ((args (eval-rands rands env)))
    (eval-expression body (extend-env ids args env))))
```

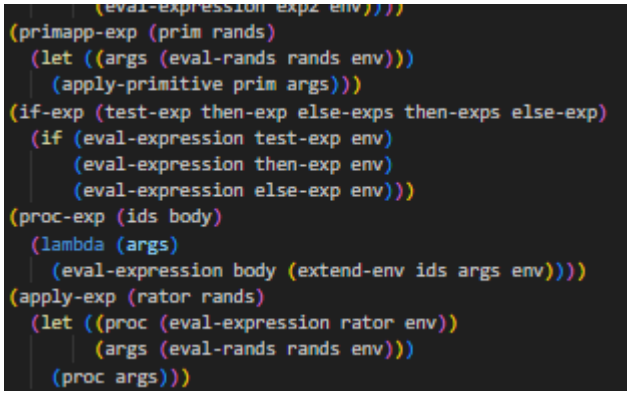
En esta parte continuamos con las funcionalidades (var) y (let) de la cual la función mutable var nos da un incremento de la expansión de nuestro interpretador igual que el let que nos ayuda con las variables inmutables.

Tambien se puede tener la función letrec que usamos como recursividad, pero en este caso no la implementamos de una manera específica.



```
(set-exp (id rhs-exp)
  (begin
    (set! (apply-env env id) (eval-expression rhs-exp env))
    1))
(begin-exp (exp1 exp2)
  (let ((acc (eval-expression exp1 env))
        (exp2 exp2))
    (if (null? exp2)
        acc
        (eval-expression exp2 env))))
```

`set-exp` es una función que asigna un valor a una variable en un entorno. Aquí, se está actualizando el valor asociado con el identificador `id` en el entorno `env, rhs-exp` es la expresión cuyo valor se evalúa y se asigna a `id` en el entorno, `eval-expression rhs-exp env` evalúa la expresión `rhs-exp` dentro del entorno `env` y asigna el resultado al identificador `id,set!` se utiliza para actualizar la asignación de la variable en el entorno, La función termina con `1`, esto da el valor de retorno del bloque `begin`, aunque esto no afecta directamente al entorno. Tambien el begin-exp nos ayuda a evaluar la secuencia de las siguientes expresiones.



```
(primapp-exp (prim rands)
  (let ((args (eval-rands rands env)))
    (apply-primitive prim args)))
(if-exp (test-exp then-exp else-exps then-exps else-exp)
  (if (eval-expression test-exp env)
      (eval-expression then-exp env)
      (eval-expression else-exp env)))
(proc-exp (ids body)
  (lambda (args)
    (eval-expression body (extend-env ids args env))))
(apply-exp (rator rands)
  (let ((proc (eval-expression rator env))
        (args (eval-rands rands env)))
    (proc args)))
```

Aquí continúa evaluando las funciones donde empezamos con la primapp-exp en la cual nos ayuda a evaluar la primitivas básicas, después gracias a los if nos da la ayuda de interpretar las condiciones que puede tener una validación a la hora de evaluar una función, con la ayuda de los procedimientos creamos clausulas como en los ambientes expuestos a papel.

Terminamos con los apply-exp que nos da la implementación de aplicaciones de funciones y procedimientos.

```
(for-exp (id start-exp end-exp body)
  (let loop ((i (eval-expression start-exp env))
             (end (eval-expression end-exp env)))
    (if (> i end)
        'ok
        (begin
          (eval-expression body (extend-env (list id) (list i) env))
          (loop (+ i 1) end)))))
(else (my-error (string-append "Unknown expression type: " (format "~a" exp)))))))
```

Esta implementación del for, nos quiere llevar a entender un método de ciclos que nos puede ayudar a entender el interpretador con funciones de ciclos entendidos por la maquina.

```
;; Función para iniciar el intérprete
(define interpretador
  (sllgen:make-rep-loop "-->"
    (lambda (pgm) (eval-program pgm))
    (sllgen:make-stream-parser
      scanner-spec-simple-interpreter
      grammar-simple-interpreter)))

;; Iniciar el intérprete
(interpretador)
```

Terminamos con la funcionalidad del expresar el interpretador.

Concluimos con este proyecto que nos falta mucha implementación a la hora de interpretar un programa, o mejor dicho nos quedamos corto a la hora de entender de mejor manera el proyecto. Entendimos la problemática, pero nos faltó el conocimiento específico para implementarlo bien.

(PROFESOR COMO ULTIMO COMIT SE HIZO A LAS 8 DE LA NOCHE QUE FUE HECHO A ESTA HORA POR EL INCONVENIENTE DE QUE SE CREÍA HABER SUBIDO ANTES (DESPUÉS DE LA SUSTENTACIÓN) Y ESTE COMIT TIENE EL PROGRAMA EXPUESTO EN LA SUSTENTACIÓN)

(Si hay algún inconveniente por la hora de subir este informe es que no se subio a otro git pero no a este que tenemos ahora mismo como ultimo.)

Pruebas

```
C:\Program Files\Racket\collects\racket\enter.rkt:54:
> -> a
  apply-env-let: No binding for a
  -> 3
 string:1:0: ->: bad syntax
  in: ->
  [,bt for context]
 a: undefined;
  cannot reference an identifier before its definition
   in module: top-level
  [,bt for context]
 apply-env-let:: undefined;
  cannot reference an identifier before its definition
   in module: top-level
  [,bt for context]
 No: undefined;
  cannot reference an identifier before its definition
   in module: top-level
  [,bt for context]
 binding: undefined;
  cannot reference an identifier before its definition
   in module: top-level
  [,bt for context]
 string:2:26: for: bad syntax
  in: for
  [,bt for context]
 a: undefined;
  cannot reference an identifier before its definition
   in module: top-level
  [,bt for context]
 string:3:0: ->: bad syntax
  in: ->
  [,bt for context]
 3
```

```
> -> 'a'
  |'a'|
  -> "acascas2123ascsa acsc"
string:1:0: ->: bad syntax
  in: ->
 [,bt for context]
'a
'|'a'|
string:3:0: ->: bad syntax
  in: ->
 [,bt for context]
"acascas2123ascsa acsc"
```

```
> -> var x=10 y=20 in +(x,y) end
string:1:0: ->: bad syntax
  in: ->
 [,bt for context]
var: undefined;
 cannot reference an identifier before its definition
  in module: top-level
 [,bt for context]
x=10: undefined;
 cannot reference an identifier before its definition
  in module: top-level
 [,bt for context]
y=20: undefined;
 cannot reference an identifier before its definition
  in module: top-level
 [,bt for context]
in: undefined;
 cannot reference an identifier before its definition
  in module: top-level
 [,bt for context]
#<procedure:+>
string:1:23: unquote: not in quasiquote
  in: (unquote y)
 [,bt for context]
end: undefined;
 cannot reference an identifier before its definition
  in module: top-level
 [,bt for context]
```