Informe proyecto final

Jhojan Stiven Castaño Jejen

William Alexander Franco Otero

Cristian David Rivera Torres

David Santiago Velasco Triana

Fundamentos de Interpretación y Compilación de LP

Ingeniería Sistemas

Universidad Del Valle

Quinto Semestre

Tuluá, Valle Del Cauca

2024

Explicación del Código

1. Función determinar-expresión

En esta parte del código tiene como propósito principal tomar una expresion exp y un entorno env y evalúa exp basándose su tipo. Se declara un cases que evalua exp en función de su tipo de expresión. Donde los cases son los siguientes:

Num-exp: Este es el caso para las expresiones numéricas. Se encarga de evaluar el numero en función de su tipo (decimal, binario, octal, hexadecimal, flotante)

Var-exp: en este caso para la variable se encarga de buscar el valor de la variable en el entorno.

True-exp y false-exp: En el caso de que el valor sea verdadero o falso retornará #T or #F

Func-exp: en este caso para las funciones, se crea un cierre con los identificadores, el cuerpo de la función y el entorno.

App-exp: En este caso se encarga de evaluar el operador y los operandos, y luego aplicar el operador a los operandos. En resumen, el operador es la función que se va a aplicar, y los operandos son los argumentos a los que se va a aplicar la función.

Set-exp: Es para las asignaciones de variables donde se evalúa el nuevo valor y luego modifica la variable en el entorno con el nuevo valor

Prim-num-exp: Se encarga para las operaciones primitivas con números, donde evalúa las dos expresiones y luego aplica la función operación primitiva.

Prim-cad-exp: es para las oprecions primitivas con cadenas, donde prim es la operación primitiva que se va a aplicar y operandos son las expresiones que se van a evaluar para lograr obtener los argumentos de la operación

Begin-exp: la expresión Begin se encarga de evaluar cada expresión en la lista en orden, donde exp es la primera expresión que se va a evaluar y lista_exps es una lista de las expresiones restantes que se van a evaluar.

Let-exp: Los identificadores son las variables de que se van a vincular, operandos son las expresiones que se van a evaluar para así obtener los valores a los que se van a vincular las variables, y cuerpo vienes siendo la expresión que se va evaluar en el nuevo ámbito

Var-exp: En este caso el var expresión viene siendo similar al let expresión solo con la diferencia de que este permite la modificación de las variables

Cadena-exp: Se encarga de construir una cadena a partir de los identificadores.

Lista-exp: Se evalúa cada expresión en la lista, lista_exp es una lista de expresiones donde se van a evaluar en el entorno env utilizando la función evaluar-operandos

Cons-exp: para la expresion cons, exp1 y exp2 son dos expresiones donde se van a evaluar en el entorno env utilizando la función evaluar-operando. Luego, se encarga de crear un par con los resultados de estas evaluaciones utilizando la función cons.

Prim-list-exp: Este es para las opresiones primitivas con listas. Donde se evalúa la expresioni y luego se aplica la operación primitiva.

For-exp: Este es el caso para la expresión for, donde var es la variable de control del bucle, la expresion-inicio es la expresión que determina el valor inicial del var, expresión-final es la expresión que determina el valor final de var, el expresion-suma es la expresión que determina cuanto se incrementa var en cada iteración del bucle, y cuerpo_exp es la expresión donde se ejecuta en cada iteración del bucle.

Array-exp: Evalua cada espresion en la lista y luego las convierte en un vector.

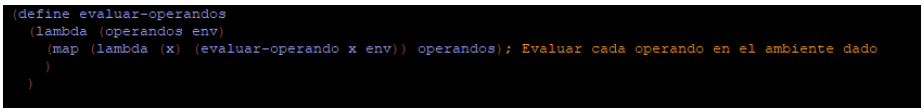
Prim-array-exp: En este caso es para las operaciones primitivas con arrays. Se evalúa los argumentos y luego plica la operación primitiva.

Empty-list: En este caso es para retornar una lista vacia

Match-exp: Se evalua la expresion variable y luego busca una coincidencia dentro de los casos.

2. **Funciones listas de operandos**
3.

Evaluar-operandos: En esta función se encarga de tomar dos argumentos, operandos y env.operandos es una lista de operandos. Donde la función map es utilizado para aplica una función a cada elemento de una lista. Para este caso, la función que se aplica es la función evaluar operando, donde toma un operando X y lo evalua en el ambiente env utilizando la función evaluar-operando, donde resulta una nueva lista que contiene el resultado de evaluar cada operando en el ambiente dado.



Evaluar-operando: En esta fucion toma dos argumentos, operando y env. Donde esta función pasa los dos argumentos a la función determinar-expresion donde evalua el operanod en el ambiente dado.

```
(define evaluar-operando
  (lambda (operando env)
    (determinar-expresion operando env); Evaluar el operando en el ambiente dado
    )
  )
```

4. Modificación de Strings

Eliminar-carácter: Esta función se toma dos argumentos, cadena y carácter. Cadena que es una cadena de caracteres y carácter es el carácter del cual se desea eliminar de la cadena. La función convierte la cadena en una lista de caracteres, para luego llamar la función remover-carácter para eliminar el carácter especificado, y finalmente convierte la lista resultante en una nueva cadena. La función remover carácter es una función recursiva que se encarga de recorrer la lista de caracteres y construye una nueva lista omitiendo el carácter especificado.

```
(define eliminar-caracter 4 bound occurrences
  (lambda (cadena caracter)
    (letrec
        [
         (remover-caracter
          (lambda (cadena caracter)
            (cond
              [(null? cadena) '()] ; Si la cadena está vacía, retornar una lista vacía
              [else
               (if (char=? caracter (car cadena)); Si el caracter actual es igual al caracter a eliminar
                           )
                   (remover-caracter (cdr cadena) caracter) ; Continuar con el resto de la cadena sin añadir el caracter actual
                   (cons (car cadena)
                         (remover-caracter (cdr cadena) caracter); Añadir el caracter actual y continuar con el resto de la cadena
                         )
                   )
               ]
              )
            )
          )
         ]
      (list->string (remover-caracter (string->list cadena) caracter)) ; Convertir la cadena a una lista, eliminar el caracter y convertir de nuevo a cadena
      )
    )
  )
```

Reemplazar-carácter: Aquí se toma 3 argumentos, cadena que es una cadena de caracteres, carácter que es el carácter que se quiere remplazar y nuevo_caracter es el carácter que se va a usar para remplazar. La función convierte la cadena en una lista de caracteres, para luego llamar la función reemplazar-car para reemplazar todas las ocurrencias del carácter especificado por el nuevo carácter, y luego convierte la lista resultante en una cadena. La función reemplazar-car ya es una función recursiva que se encarga de recorrer la lista y construye una nueva lista reemplazando el carácter por un nuevo carácter.

```
(define reemplazar-caracter
  (lambda (cadena caracter nuevo_caracter)
    (letrec
        [
         (reemplazar-car
          (lambda (cadena caracter nuevo_caracter)
            (cond
              [(null? cadena) '()] ; Si la cadena está vacia, retornar una lista vacia
              [else
               (if (char=? caracter (car cadena)) ; Si el caracter actual es igual al caracter a reemplazar
                   (cons nuevo_caracter
                         (reemplazar-car (cdr cadena) caracter nuevo_caracter); Añade el nuevo caracter y continuar con el resto de la cadena
                         )
                   (cons (car cadena) (
                                       reemplazar-car (cdr cadena) caracter nuevo_caracter); Añade el caracter actual y continuar con el resto de la cadena
                         )
                   )
               ]
              )
            )
          )
         ]
      (list->string (reemplazar-car (string->list cadena) caracter nuevo_caracter)) ; Convertir la cadena a una lista, reemplazar el caracter y convertir de nuev
      )
    )
  )
```

5. Aplicación de operación de acuerdo con la base del numero

Condicionales-bases: Se toma cuatro argumentos que son operación, arg1, arg2 y convertir. La función verifica si arg1 es una cadena de que representa un numero en una base especifica (binario, octal, hexadecimal) que realiza una operación en los números convertiros a la base 10, y luego se convierte de nuevo a la base original. En caso de que arg1 no sea una cadena, la operación se realiza directamente en arg1 y arg2.

Retornar-booleano: Se encarga de retornar su argumento salida sin cambio, donde se utiliza en las operaciones de comparación, donde le resultado es un booleano y no necesita ser convertido en cadena.

```
(define retornar-booleano
  (lambda (salida base) salida) ; Retorna el valor de salida sin cambios
  )
```

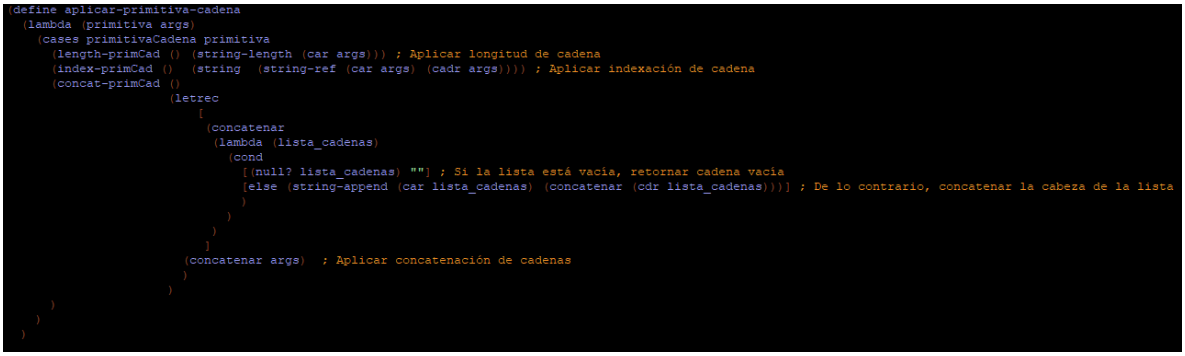
Convertir-número-a-base: En esta función se toma dos argumentos que son numero y base, donde transforma numero a una cadena de acuerdo con la base especificada. En caso de que el numero sea negativo, se añade un ‘- ‘al inicio de la cadena. Si la cadena es para base 2, se añade la b al inicio de la cadena, si es base 8, se añade el “0x” y para base 16, se añade “hx”.

```
(define convertir-numero-a-base
  (lambda (numero base)
    (let* ((es-negativo? (< numero 0)) ; Verificar si el número es negativo
           (valor-absoluto-numero (abs numero))) ; Obtener el valor absoluto del número
      (case base
        [(2) (if es-negativo?
                 (string-append "-" "b" (number->string valor-absoluto-numero 2)) ; Si es negativo, añadir '-' y 'b' al inicio
                 (string-append "b" (number->string valor-absoluto-numero 2))) ; Si no es negativo, añadir 'b' al inicio
        ]
        [(8) (if es-negativo?
                 (string-append "-" "0x" (number->string valor-absoluto-numero 8)) ; Si es negativo, añadir '-' y '0x' al inicio
                 (string-append "0x" (number->string valor-absoluto-numero 8))) ; Si no es negativo, añadir '0x' al inicio
        ]
        [(16) (if es-negativo?
                  (string-append "-" "hx" (number->string valor-absoluto-numero 16)) ; Si es negativo, añadir '-' y 'hx' al inicio
                  (string-append "hx" (number->string valor-absoluto-numero 16))) ; Si no es negativo, añadir 'hx' al inicio
        ]
        [else (eopl:error "Base no soportada")] ; Si la base no es 2, 8 o 16, mostrar un error
        )
      )
    )
  )
```

Aplicar-primitiva: Hay 3 argumentos que son primitiva, arg1 y arg2 y se aplica la operación especificada por primitiva a los arg1 y arg2 utilizando la función condicionales-bases. Se utilizan las operaciones disponibles que son suma, resta, multiplicación, mayor que, menor que, menor o igual, mayor o igual, diferente, igual, modulo y potencia.

6. Tomar primitiva y una lisita de argumentos y aplicar su operación

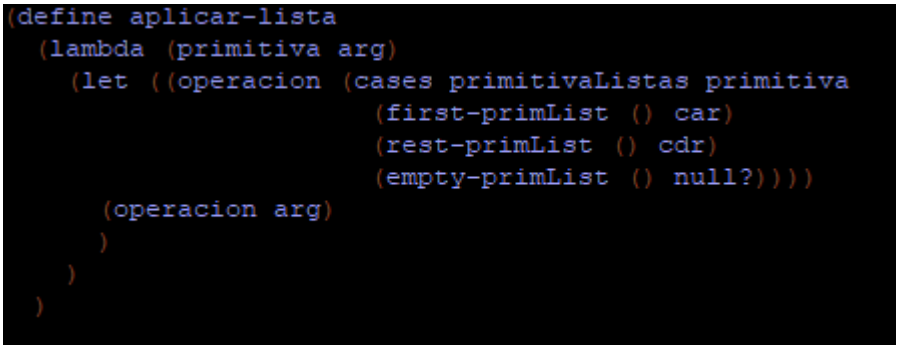
Aplicar-primitiva-cadena: Se toma una operación y una lista de argumentos, se realiza la operación especificada en los argumentos, Las operaciones son obtener la longitud de una cadena, obtener un carácter en un índice especifico de una cadena, o concatenar una lista de cadenas.



```
(define aplicar-primitiva-cadena
  (lambda (primitiva args)
    (cases primitivaCadena primitiva
      (length-primCad () (string-length (car args))) ; Aplicar longitud de cadena
      (index-primCad ()  (string  (string-ref (car args) (cadr args)))) ; Aplicar indexación de cadena
      (concat-primCad ()
                      (letrec
                          [
                           (concatenar
                            (lambda (lista_cadenas)
                              (cond
                                [(null? lista_cadenas) ""] ; Si la lista está vacia, retornar cadena vacia
                                [else (string-append (car lista_cadenas) (concatenar (cdr lista_cadenas)))] ; De lo contrario, concatenar la cabeza de la lista
                                )
                              )
                            )
                           ]
                        (concatenar args)  ; Aplicar concatenación de cadenas
                        )
                      )
      )
    )
  )
```

7. Función aplicar lista

Aplicar-lista: Se toman dos argumentos, primitiva y arg.primitiva que representa una operación para realizar en una lista y arg es la lista en la que se realizará la operación. Luego se usa un case que selecciona la operación para realizar en base la primitiva proporcionada, si la primitiva es first se aplica la función car a arg, que devuelve el primer elemento de la lista; en caso de que la primitiva sea el rest aplica la función cdr a arg, donde devuelve la lista sin el primer elemento, en caso de que sea vacio aplica la función null a arg, que devuelve true si la lista es vacia o false en caso contrario. Para finalizar devuelve su resultado.



```
(define aplicar-lista
  (lambda (primitiva arg)
    (let ((operacion (cases primitivaListas primitiva
                       (first-primList () car)
                       (rest-primList () cdr)
                       (empty-primList () null?))))
      (operacion arg)
      )
    )
  )
```

8. Obtener subvector y primitiva del vector

Obtener-subvector: La función se encarga de devolver una lista que contiene los elementos del vector desde el índice de inicio hasta el índice del final. En caso de que el inicio y el final sea iguales, devuelve la lista con un solo elemento. En caso contrario, se utiliza la recursión para obtener los elementos del vector desde el índice del inicio hasta el final.

```
(define obtener-subvector
  (lambda (vector inicio fin)
    (if (= inicio fin)
        (list (vector-ref vector inicio)
              )
        (cons (vector-ref vector inicio) (obtener-subvector vector (+ inicio 1) fin)
              )
        )
    )
  )
```

Primitiva-vector: Se realiza de acuerdo con la operación especificada en los argumentos. Las operaciones puedes ser obtener la longitud de un vector, obtener un subvector, obtener el elemento en un índice específico del vector, o establecer el valor de un elemento en un índice especifico de un vector.

```
(define primitiva-vector
  (lambda (primitiva arg)
    (let ((primer-arg (car arg))
          (segundo-arg (cadr arg))
          (tercer-arg (caddr arg)))
      (cases primitivaArray primitiva
        (length-primArr () (vector-length primer-arg))
        (slice-primArr () (list->vector (obtener-subvector primer-arg segundo-arg tercer-arg)))
        (index-primArr () (vector-ref primer-arg segundo-arg))
        (setlist-primArr ()
                         (vector-set! primer-arg segundo-arg tercer-arg)
                        primer-arg
                         )
        )
      )
    )
  )
```

9. Definición del bucle for

Bucle-for: Esta función es una simulación del bucle for, donde se evalúa en cada iteración del bucle, con una variable de control que se incrementa en cada iteración hasta que alcanza el valor final.

```
(define bucle-for imported from eopl
  (lambda (expresion-cuerpo variable indice incremento fin entorno)
    (if (< indice fin) ; Comprueba si el indice es menor que el valor final
        (begin
          (determinar-expresion expresion-cuerpo (extend-env (list variable) (list indice) entorno)) ; Evalúa la expresión del cuerpo con la variable establecida
          (bucle-for expresion-cuerpo variable (+ indice incremento) incremento fin entorno)) ; Llama recursivamente a bucle-for con el indice incrementado
        'void-exp ; Si el indice no es menor que el valor final, retorna 'void-exp'
        )
    )
```

10. Función coincidir valor (match)

Coincidir valor: Se encarga de tomar cuatro argumentos: valor que es el valor que se va a comparar, primit-coincidencia que es una lista de primitivas donde representan

patrones de coincidencia, expresion_coincidencia que es una lista de expresiones que se evaluarán si se cumple la coincidencia y por ultimo el env que es el entorno en el que se evaluarán las expresiones. Se hace una expresión cases: donde primero donde se comprueba si el valor coincide con el patrón de la primitiva de coincidencia. Si es así, se evaluará la primera expresión de coincidencia donde se extiende con el valor. En caso contrario, se llama recursivamente a coincidir-valor con el resto de las primitivas y expresiones de coincidencia. Si es por defecto, ninguna de las otras ramas se selecciona, se evalúa la primera expresión de coincidencia en el entorno actual.

11. Closure

Procval: Esta función tiene tres argumentos que es identificadores, que es una lista de símbolos donde representan los parámetros de la función, los cuerpos, que representa el código que se ejecutará en el momento que se llame a la función, y por último el env que es el entorno donde se va a evaluar el cuerpo de la función, donde contiene las asociaciones entre variables y sus valores.

```
(define-datatype procval procval?
  (closure
   (identificadores (list-of symbol?); Lista de identificadores
                    )
   (cuerpos expresion?) ; Cuerpo del procedimiento
   (env environment?); Entorno de evaluación
   )
  )
```

12. Ambientes

Se proporcionan tres constructores que es el empty para el entorno vacío, el extend-env para un entorno extendido y extend-modificable-env para un entorno extendido con un vector de valores, permitiendo modificaciones posteriores

13. Buscar variable especifica en un ambiente dado.

Search-no-modificable-env: se encarga de buscar el valor buscado en una lista de identificadores, en caso de que se encuentre el identificador, devuelve el valor correspondiente de la lista de valores, en caso contrario y si la lista está vacia, busca el siguiente entorno. Si el identificador no está en la lista y la lista no está vacia entonces se llama la función a si misma recursivamente para buscar en el resto de la lista.

14. Buscar el valor en un entorno modificable

Es una función que se encarga de buscar un identificador en una lista de identificadores y devuelve el valor correspondiente de un vector de valores. Si el valor es una expresión, se evalúa el entorno dado, en caso de que no se encuentre en la lista el identificador, se busca en el siguiente entorno.

15. Buscar símbolo en el ambiente

Se encarga de buscar un símbolo en un entorno, si el entorno es vació se devuelve un error, en caso de que sea entorno extendido, busca el símbolo usando search no modificable, en caso de que el entorno sea modificable, se busca el símbolo usando el search modificable.

16. Modificar una ligadura en un ambiente

La función aplicar ambiente se encarga de buscar un símbolo en un entorno, depende del tipo del entorno, utiliza diferentes métodos de búsqueda; en caso de que sea vacio se genera un error, si es extendido se utiliza la función searh-no-modificable para buscar el símbolo y si es modificable el entorno se utilizará la función search-modificable para buscar el símbolo.