UNIVERSIDAD DEL VALLE - SEDE REGIONAL TULUÁ
PROGRAMA DE INGENIERÍA DE SISTEMAS E INFORMÁTICA
FUNDAMENTOS DE INTERPRETACIÓN Y COMPILACIÓN LP

Informe Proyecto FLP

Asignatura:

Fundamentos de interpretación y compilación lp

Programa:

Ingeniería de Sistemas

Estudiantes:

Victor Manuel Álzate Morales - 202313022
Juan Esteban Lopez Aránzazu - 202313026
Joan Sebastián Saavedra Perafán - 202313025
Diego Alejandro Tolosa Sanchez - 202313023

Semestre:

Segundo/Sexto semestre

Docente:

Carlos Delgado

Universidad del Valle – Sede Tuluá

UNIVERSIDAD DEL VALLE - SEDE REGIONAL TULUÁ
PROGRAMA DE INGENIERÍA DE SISTEMAS E INFORMÁTICA
FUNDAMENTOS DE INTERPRETACIÓN Y COMPILACIÓN LP

Universidad del Valle

INFORME

Para cada expresión se debe agregar en la función eval-expression donde se manejan todas las expresiones del programa, luego para cada variante de expresión en caso de tener más variantes se evalúa en otra función auxiliar para poder evaluar cada caso, por ejemplo para los números, que tiene las variantes de entero, flotante, hexadecimal, octal y binario. Por otro lado se manejan cases porque las expresiones son datatype y en los casos donde la variante tiene variantes también son de tipo datatype.

Funciones principales

1. La especificación léxica define las reglas para poder analizar los tokens que se manejan en el lenguaje. Es decir para reconocer los identificadores, números, comentarios, ignorar espacios entre otros.
2. La gramática define la estructura del programa en la cual se definen unas reglas para cada expresión que se usa en el lenguaje.
3. La función eval-program es una función que evalúa un programa, el cual contiene una variante a-programa que recibe dos argumentos, una lista de estructuras y una expresión, la función evalúa la expresión usando eval-expression, para esto se debe tener en cuenta el ambiente, se puede pasar un ambiente inicial con variables inicializadas o un ambiente vacío.
    a. Si pasamos un ambiente inicial se crea un ambiente que extiende del ambiente vacío.
    b. Si la expresión contiene una ligadura se crea un ambiente extendido que contendrá las variables denotadas.
4. La función interpretador se usa para evaluar el programa dado una especificación léxica y una gramática.

Primitivas numéricas

1. Se agrega la variante num-exp en la función eval-expression, para manejar los números dentro de esta variante se invoca la función auxiliar eval-num-expresion que tiene los casos de entero, flotante, hexadecimal, octal y binario.
2. Se agrega la variante prim-num-exp a la función eval-expression para manejar las primitivas numéricas donde se invoca la función apply-primitive que hace la conversión entre bases y luego hace la operación. La función apply-primitive evalúa cada variante de las operaciones como +, -, * entre otros.
    a. Para las primitivas numéricas se implementaron dos funciones para convertir entre entero y una base b, 0x o hx

b. Se implementó la función decimal-to-base-x es una función auxiliar para convertir un numero decimal a base x
c. Se implementó la función base-x-to-decimal es una función auxiliar para convertir un número en base x a decimal
d. Se implementó la función check-base-apply-operation es una función auxiliar el cual le llega dos números y una función de operación, los números los convierte a decimal, luego los opera y por último los convierte en la base x que inicialmente tenían los números.
    i. La función valida que los números sean de bases iguales, en caso que si cumpla hace la operación y en caso contrario lanza un error indicando que no se puede hacer la operación

Primitivas booleanas

1. Se agrega la variante bool-exp en la función eval-expression, para manejar los tipos de booleanos true y false se usa una función auxiliar eval-bool-expresion que contiene las variantes para true y false
2. Se agrega la variante prim-bool-exp para manejar las primitivas booleanas que contiene los casos de and, or, xor y not, desde esta variante se invoca la función apply-primitive-bool
    a. La función apply-primitive-bool evalúa los casos de las primitivas booleanas, el cual tiene como argumentos una primitiva y lista de expresiones booleanas.
    b. Para el caso de la primitiva and, or y xor, se usa la función operation que recorre una lista de forma recursiva operando la función que recibe como parámetro.
        i. and((5 > 3), true, (3 == 3)), en este caso la primitiva es and y la lista de expresiones booleanas seria (#t, #t, #t), la función operation inicialmente tiene como acomulador #t, entonces rompería cuando una expresión booleana sea #f
        ii. or(((8 mod 2) == 0), false, true), en este caso la primitiva es or y la lista de expresiones booleanas seria (#t, #f, #t), la funcion operation inicialmente tiene como acomulador #f, entonces rompería cuando una expresión booleana sea #t

Primitivas cadenas

1. Se agrega la variante cadena-exp en la función eval-expression el cual tiene como argumentos un identificador y una lista de identificadores, en esta variante se invoca una función auxiliar create-cadena-exp el cual crea un string dado el identificador y la lista de identificadores

   a. Para la función create-cadena-exp este usa una función auxiliar string-join el cual concatena una lista de identificadores con un delimitador en este caso “ “
      i. (hola, mundo, cruel) en este caso este seria una lista de identificadores que se debe convertir a string, al final haciendo la concatenación “hola mundo cruel”
      ii. “ ”.join([“hola”,”mundo”,”cruel”]) este sería un ejemplo usando python para hacer la concatenación entre string dado un array
2. Se agrega la variante prim-cad-exp en la función eval-expression para manejar las primitivas de cadenas, en este caso las primitivas de concat, string-length y elementAt, para esto invoca la función auxiliar apply-primitive-cad que evalúa cada variante de primitivas
   a. Para la función apply-primitive-cad recibe dos argumentos la primitiva y la lista de expresiones de cadena
   b. Para la primitiva concat, se usa la función string-join para concatenar la lista de expresiones de cadena usando como delimitador “”, esta función fue implementada desde cero usando recursividad a una lista
   c. Para la primitiva length, se usa la función por defecto de racket string-length y se calcula el tamaño del string del primero de la lista (car lst)
   d. Para la primitiva index, se usa la función por defecto de racket string-ref el cual recibe como parámetros un string y un número que representa el índice
      i. elementAt(“hola mundo cruel”,5) en este caso la primitiva sería elementAt, el string seria “hola mundo cruel” y el índice 5 y daría como resultado “m”

Primitivas listas

1. Se agrega la variante lista-exp en la función eval-expression, para manejar las expresiones devolviendo una lista con los resultados de las evaluaciones.
2. Las funciones también se puede expresar de diferentes maneras, se puede usar el cons y empty, entonces se debe agregar dos variantes en la función eval-expression, una para poder usar cons y otra para poder usar empty
   a. En el caso de cons éste recibe dos argumentos que son dos expresiones y crea una lista usando cons para crear una lista
      i. cons (1 cons (2 empty))
   b. En el caso de empty este no recibe ningún argumento y crea una lista vacía ‘()
   c. Para este caso se usa el map para evaluar cada argumento
      i. list(1,2,3,true,(5 > 2)) en este caso crea una lista con los siguientes elementos ‘(1 2 3 #t #t)
3. Se agrega la variante prim-list-exp para manejar las operaciones de primitivas de listas y La función apply-primitive-list evalúa los casos de las primitivas de listas en

donde first-primList devuelve el primer elemento de la lista utilizando la función car, rest-primList devuelve el resto de la lista con la función cdr y empty-primList verifica si la lista está vacía usando la función null? por defecto de racket, en caso que la lista esté vacía retorna #t si es verdadero y en caso contrario #f.

Primitivas array

1. Se agrega la variante array-exp en la función eval-expression donde toma cada elemento de args, y evalúa la expresión en el entorno utilizando eval-expression,la lista se convierte en un vector utilizando la función list-> vector, por lo tanto devuelve un vector como resultado. La diferencia entre array y listas es que los arrays se pueden modificar.
    a. Al igual que la expresión para listas también usa map para evaluar cada expresión de lista de argumentos que recibe.
2. Se agrega la variante prim-array-exp para manejar las primitivas de arrays, donde se invoca la función auxiliar apply-primitive-array el cual evalúa cada caso de las primitivas para array, las primitivas son las siguientes length-primArr el cual devuelve el tamaño de un vector, index-primArr el cual obtiene un elemento a partir de un índice del vector, slice-primArr el cual genera un nuevo vector dado un número de inicio y un número final y setlist-primArr el cual modifica un elemento del vector dado un índice.
    a. Para el caso de slice, la primitiva espera tres argumentos, el vector, el número de inicio y el número final
        i. slice(array(1,2,3,4),1,3) el cual retorna array(2,3), en racket los vectores tendrían esta forma #(2 3)
        ii. [1,2,3,4][1:3] este sería un ejemplo en python el cual retornaría [2,3]

Estructuras de control (for y while)

Iterador for

1. Se agrega la variante for-exp en la función eval-expression para manejar el bucle for en el programa, esta variante recibe cuatro argumentos un identificador, from que es una expresión, until que es una expresión, by que es una expresión y por ultimo do otra expresion que se estara haciendo hasta que el bucle llega a una condición de parada o condición terminal, esta variante invoca una función auxiliar eval-for-exp que itera de manera recursiva hasta llegar a la condición de parada.
2. Para el caso de los argumentos se espera que la variable from sea un numero que seria el valor inicial, para la variable until se espera que sea un número que tomaría el valor

final y la variante by también se espera sea un numero, que seria el incremento del identificador hasta que cumpla la condición de parada.

a. Este sería un ejemplo usando el iterador for i from 0 until 10 by 1 do set x = (x + i), en este ejemplo el iterador inicia desde 0 hasta 10 incrementando su valor en 1 en cada iteración, la expresión set x = (x + i); se iria ejecutando por cada iteración modificando el valor de x.
b. for i in range(0,10,1) este sería un ejemplo usando python

Iterador while

1. Se agrega la variante while-exp en la función eval-expression para manejar iterador while en el programa, esta variante recibe dos argumentos, una variable exp expresión y una variable body que es otra expresión, el while funciona de forma similar que el for la diferencia es la expresión que define la condición de parada o condición terminal, para que el bucle rompa y no siga de manera infinita, el cuerpo es una expresión que se iria ejecutando por cada iteración del bucle.
2. Para el caso de los argumentos se espera que la expresión sea un tipo test-exp como el caso de los condicionales una expresión booleana, para poder validar si sale de la función recursiva o continua iterando, y el cuerpo sería una expresión que se ejecuta por cada iteración
   a. Este sería un ejemplo usando el iterador while (x < 10) { set x = (x + 1); }, en este ejemplo, la exp seria (x < 10) que es una expresión tipo test-exp para validar una expresión booleana y el cuerpo sería set x = (x + 1); que iría modificando el valor de x por cada iteración
   b. while(x<10): x+=1 este sería un ejemplo usando python

Switch

1. Se agrega la variante switch-exp en la función eval-expression, cuando se encuentra una expresión switch, se llama la función eval-switch-exp con la expresión de control, las expresiones de casos, las expresiones de los cuerpos de los casos, la expresión por defecto y el entorno actual evaluando la expresión de control y luego ejecutando el cuerpo del caso correspondiente.
2. eval-switch-exp es una función para evaluar un switch. El cual recibe cinco argumentos los cuales son:

   i. exp que determina que caso ejecutará

   ii. cases una lista de expresiones con los casos posibles

   iii. casesExp lista de expresiones que se evalúan si su caso es seleccionado

iv. default evalúa si ninguno de los casos coincide

v. env entorno donde se evalúan las expresiones

En esta función evaluamos la expresiones de los casos, expresiones de los cuerpos de los casos y evaluar la expresión por defecto, además evaluamos si la lista de casos está vacía, si el valor es igual al caso y si no cumple se itera.

Condicionales

1. Se agrega la variante if-exp en la función eval-expression donde se manejara las condicionales en el programa, esta variante recibe tres argumentos, el test-exp, true-exp y false-exp, el cual son expresiones, donde el test-exp se espera sea un booleano para validar si retorna el caso true-exp o el caso false-exp
    a. Este sería un ejemplo if (3 > 1) { “hola” else “mundo” }, la estructura de la gramática, entonces se evalúa el test-exp y debería dar un booleano #t o #f en caso que sea #t evalua la expresion “hola” de lo contrario evalúa la expresión “mundo”
    b. “hola” if(3>1) else “mundo” este sería un ejemplo usando python

Funciones (func y call)

1. Se agrega la variante func-exp en la función eval-expression, la estructura de la función func contiene una lista de identificadores que serán los parámetros de la función y una expresión que es el cuerpo a evaluar de la función. Cuando se encuentra esta función se ejecutará la función func-exp el cual crea una clausura con los identificadores, el cuerpo y el ambiente actual.
    a. La clausura es un datatype que contiene las variables de ids, body y env. Donde ids es una lista de identificadores de tipo symbol, el body es una expresión para evaluar y env un ambiente.
2. Se agrega la variante call-exp en la función eval-expression, la estructura de la función call contiene una expresión donde llamará a la función y una lista de expresiones donde serán los argumentos de la función a invocar. En esta variante se debe evaluar la exp y los rands, luego se debe validar que la exp sea un procedimiento, en caso que el resultado sea verdadero se invoca la función apply-procedure el cual ejecutará el cuerpo del procedimiento en el ambiente que se guardó en la clausura, en otro caso de lanza un error indicando que la exp no es un procedimiento y por lo tanto no se puede aplicar la función en el programa.

Ligadura modificable y no modificable (var y let)

1. Se agrega la variante decl-exp en la función eval-expression para manejar las ligaduras modificables y no modificables var y let en el programa, en esta variante se invoca una función auxiliar eval-var-decl para evaluar los casos de lvar-exp y let-exp.
2. En la función eval-var-decl, cuando se encuentra una expresión let se evalúan los argumentos rands en el entorno para obtener los valores de las variables. eval-rands toma una lista de expresiones y un entorno devolviendo la lista de los resultados de evaluar las expresiones en ese entorno. Se evalúa el cuerpo body de la expresión let en un nuevo entorno extendido del actual. Por cada let se debe crear un ambiente extendido del actual, diferente es para el caso del var en este se crea un ambiente extendido recursivo del actual, porque se manejan funciones recursivas, es decir tiene un comportamiento similar al letrec.

Reconocimiento de patrones (regular-exp)

1. Se agrega la variante match-exp en la función eval-expression, con la variante match-exp implementa el reconocimiento de patrones tomando tres argumentos, exp dónde cuyo valor se comparará con lo patrones, regularExp que es la lista de expresiones de los patrones y cases que es la lista de expresiones que se evaluaran si su patrón coincide con el valor de exp.
2. Dentro de esta función utilizamos letrec con las variables value e iterate, value es el resultado de evaluar exp en el entorno env e iterate toma dos listas e itera de manera recursiva hasta encontrar la expresión regular que hace match con la expresión a evaluar en el match-exp.
    a. En la función iterate cuando la lista de expresiones regulares llega a una lista vacía null? entonces lanza un error indicando que no se encontró ninguna expresión regular que hiciera match con la expresión regular
    b. También se hace uso de una función auxiliar eval-regular-exp, esta función evalúa los casos de las expresiones regulares, para listas, array, cadenas, booleanos y números
        i. La técnica que utiliza es devolver una lista con la funcion de validacion y un nuevo ambiente que puede ser el ambiente actual o un ambiente extendido del actual
        ii. La función iterate tiene que descomponer la lista para obtener la funcion de validacion y el nuevo ambiente, en caso que la validación sea verdadera se evalúa la expresión con el nuevo ambiente en otro caso continuar la iteración de la siguiente expresión regular

iii. El nuevo ambiente es necesario para crear nuevas variables que no estaban en el ambiente, por ejemplo para el caso de las listas la expresión regular x::xs, donde x es el primer elemento de la lista y xs es el resto de la lista, entonces se crea un ambiente extendido del actual donde contiene una lista con dos identificadores y una lista de valores con el primer elemento y el resto de elementos de la lista

iv. Al evaluar la expresión con el nuevo ambiente, buscará los identificadores en los ambientes hasta encontrar los nuevos identificadores y ejecutará la expresión que tienen

Estructuras

1. Para las estructuras se usó una lista de listas que contienen las estructuras, con su identificador y su lista de campos, al ejecutar la función eval-program, que recibe una lista de estructuras y una expresión, se debe modificar la lista por la lista de estructuras que recibe
2. Se debe agregar tres variantes en eval-expression, new-struct-exp, get-struct-exp y set-struct-exp para poder crear una estructura, obtener el valor de los campos de la estructura y modificar el valor de los campos de una estructura
    a. Para el caso de new-struct-exp, que recibe un identificador y una lista de expresiones que se deben evaluar, se usa una función auxiliar apply-env-struct que busca el identificador de la estructura en la lista de listas, retornando los campos en caso que la estructura se encuentre en otro caso lanza un error indicando que la estructura no se encuentra, al final se maneja un vector para almacenar los campos y el valor de los campos para posteriormente modificar el vector
    b. Para el caso de get-struct-exp, que recibe una expresión que se debe evaluar y un identificador para encontrar el campo de la estructura, en este caso se usa una función auxiliar search-fields para buscar de manera recursiva el identificador dado una lista de campos y de esta forma obtener el valor del campo
    c. Para el caso de set-struct-exp, que recibe dos expresiones que se deben evaluar y un identificador, la primera expresión es para obtener el vector que almacena los campos y los valores de los campos del vector y la otra expresión es el valor para modificar, en este caso se usa la función auxiliar set-field que se encarga de buscar el identificador de campo en la lista de campos de la estructura retornando un índice, este índice nos permite modificar el vector usando una función propia de racket vector-set! para así modificar la lista de listas de la estructura

Pruebas

Para las pruebas se usó el script para correr todas las pruebas de la carpeta tests donde contiene los archivos para validar primitivas y expresiones. El script tiene especificado la carpeta "./tests" y ejecuta cada archivo "test{numero}_{nombre}", en caso de éxito muestra el mensaje "Test succeeded" en otro caso muestra el error de racket del interpretador o lanza un error en la validación de lo esperado y lo retornado por la función eval-program. Para poder usar la función eval-program es necesario usar la línea de código (provide (all-defined-out)) para exportar todas las definiciones del archivo y poder ser usados en otros archivos racket.

Para ejecutar las pruebas de forma unitaria

sh test.sh 1 iteradores

sh test.sh 2 condicionales

sh test.sh 3 switch

sh test.sh 4 primitivaBooleana

sh test.sh 5 primitivaNumerica

sh test.sh 6 primitivaCadena

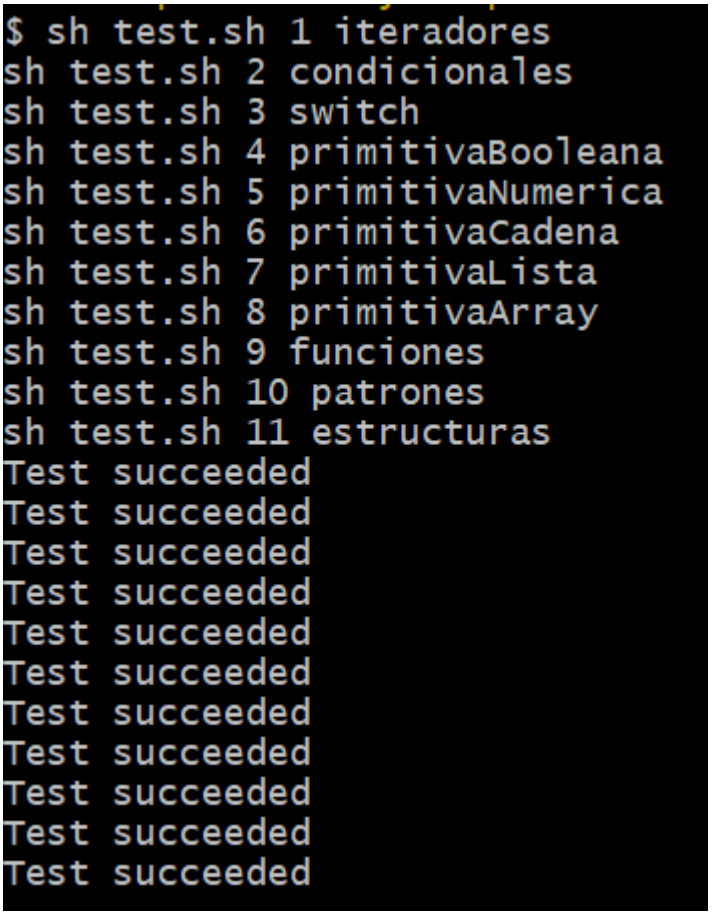
sh test.sh 7 primitivaLista

sh test.sh 8 primitivaArray

sh test.sh 9 funciones

sh test.sh 10 patrones

sh test.sh 11 estructuras



Se ejecutaron los test, en este caso fueron 11 archivos de racket, donde se evalúan las expresiones y primitivas.