INFORME DEL PROYECTO FINAL – FUNDAMENTOS DE LENGUAJES DE PROGRAMACIÓN

Integrantes:
Christian Camilo Taborda Campiño - Código 1632081
Cristian Camilo Vallecilla Cuellar - Código 1628790
Esneider Arbey Manzano Arango - Código 1628373
Santiago Hernández Arias - Código 1631281
Oliver Henao Cárdenas - Código 1631951

I. Descripción general del interpretador

Para la implementación del interpretador se utilizaron los lineamientos de objetos simples y paso de variables por referencia vistos en clase. En términos generales se intentó agrupar parámetros en el interior de paréntesis y cuerpos entre llaves.

II. Definiciones BNF de los elementos del lenguaje

- Definición para un programa:

```
<program>     ::== {<dec-table>}* {<dec-class>}* main { <expression> }
                   <p (dec-tablas dec-clases codigo)>
```

- Definición para las tablas:

```
<dec-table>   ::== table <identifier> { {field <identifier>}* {<dec-method>}* }
                   <dt (nombre-tabla ids metodos)>
```

- Definición para las clases:

```
<dec-class>   ::= class <identifier> { {field <identifier>}* {<dec-method>}* }
                   <dc (nombre-clase ids metodos)>
```

- Definición para los métodos:

```
<dec-method>  ::= method <identifier> ({<identifier>}*(,)) {<expression>}
                   <mc (nombre-metodo ids cuerpo)>
```

- Definición para las expresiones:

<expression> ::= <number>

<int-exp (dato)>

::= <string>

<string-exp (dato)>

::= <boolean>

<bool-exp (dato)>

::= <identifier>

<var-exp (id)>

::= <primitive> ({<expression>}*(,))

<prim-exp (operador operandos)>

::= if (<expression>) {<expression>} {elseif (<expression>) {<expression>}}* (else {<expression>}

<if-exp (pregunta1 positiva1 preguntas positivas negativa)>

::= let ({<identifier> = <expression>}*) in {<expression>}

<let-exp (ids valores cuerpo)>

::= proc ({<identifier>}*(,)) <expression>

<fun-exp (ids cuerpo)>

::= apply <expression> ({<expression>}*(,))

<app-exp (nombre entradas)>

::= letrec ({identifier ({identifier}*) = <expression>}*(,)) in {<expression>}

<letrec-exp(nombres idss cuerpos cuerpo)>

::= begin {<expression> {; <expression>}* ;} end

<beg-exp (principal otras)>

::= set <identifier> := <expression>

<set-exp (id valor)>

::= list ({expression}*(,))

<lis-exp (elementos)>

::= for (<identifier> = <expression>) to (<expression>) each (<expression>) do {<expression>}

<for-exp (id valor parada incremento cuerpo)>

::= select (<identifier>) from (<expression>) where (<expression>)

<sel-exp (actual elementos condicion)>

::= new <identifier> ({<expression}*(,))

<new-exp (clase entradas)>

::= send <expression> <identifier> ({<expression}*(,))

<met-exp (objeto metodo entradas)>

::= insert ({expression}*(,)) in identifier

<ins-exp (entradas tabla)>

::= access identifier (expression)

<acc-exp (tabla posicion)>

::= table. (expression) . identifier ({<expression>}*(,))

<tab-exp (registro metodo entradas)>

::= selectTable (expression) from identifier where (expression)

<selT-exp (selector tabla condicion)>

- Definición para las primitivas:

<primitive> ::= + || - || * || / || ++ || -- || == || max || min || < || > || <= || >= || or || ¬

::= append || length || cons || car || cdr || null?

III. Datatypes utilizados en el lenguaje

- **blanco** – Para el manejo de blancos directos e indirectos.
- **referencia** – Para el manejo de las referencias en los ambientes.
- **funcion** – Para el manejo de los procedimientos.
- **ambiente** – Para el manejo de los ambientes.
- **objeto** – Para el manejo de las clases y los objetos.
- **registro** – Para el manejo de las entradas en las tablas.

IV. Instancias de programas – ejemplos del enunciado del proyecto

- **Ejemplo de la figura 1 (Primitivas)**

```
-->main{ +(4, 5, 6, 7)}
22
```

Figura 1

El cuerpo del main se encierra entre llaves y los parámetros de las primitivas entre paréntesis, además de estar separados por comas.

En términos generales, para este caso lo primero es evaluar cada expresión que pueda entrar en la aplicación de una operación primitiva, llegando en todo caso hasta el valor expresado, luego se procede a validar el tipo de los datos de entrada de acuerdo a la operación a aplicar para captar posibles errores, en caso de darse las condiciones sólo se aplica la operación a todos, a los dos primeros o al primer parámetro según el contexto de la expresión.

- **Ejemplo de la figura 2 (Let, If elseif else, begin)**

```
-->main{
     let (x = 5 y = 10)
     in{
        if (&&(<=(x, 6), ==(y, 5))){
          begin{
                set x := +(x, 1);
                x
          }
          end
        } elseif (&&(>(x, 6), <=(y, 4))){
          -(x, y)
        } else{
          begin{
                set x := -(x, 1);
                x
          }
          end
        }
     }
}
4
```

Figura 2

En este caso vemos el uso de tres nuevas expresiones:

- **Let:** Para este caso evaluamos cada expresión de la primera parte del let y los convertimos en blancos directos, luego evaluamos el cuerpo del let en el ambiente extendido con los símbolos de entrada y las expresiones evaluadas anteriormente.
- **If elseif:** Para este caso lo primero es evaluar la expresión del if, se valida que sea un booleano para captar posibles errores, si es un true evaluamos la expresión del if, en otro caso procedemos a evaluar del mismo modo las preguntas de los elseif recursivamente con sus expresiones, al final, si ninguna expresión fue encontrada como true se evalúa el cuerpo del else.
- **Begin:** En este caso lo que se hace es evaluar la primera expresión entrante, si es la única se retorna su resultado, si no se procede por medio de un loop a evaluar las demás expresiones y a retornar la última respuesta al llegar al empty.

La declaración de variables en la expresión **let** se hace entre paréntesis y el cuerpo del **in** correspondiente entre llaves. La expresión de chequeo de la expresión **if** y sus **elseifs** correspondientes se realizan entre paréntesis, las consecuencias positivas y el cuerpo del **else** entre llaves. Además, la expresión **if** no se culmina con **end.** El cuerpo de la expresión **begin** se escribe entre llaves y las operaciones antes del valor de retorno se separan con **;**

- **Ejemplo de la figura 3 (Proc, apply)**

```
-->main{
     let (
          f = proc(x, y, z) if (==(x, #T)){
                              max(y, z)
                            } else{
                              min(y, z)
                            }
           m = 0
     )
     in{
        apply f(#T, m, -4)
     }
}
0
```

Figura 3

En este caso encontramos dos nuevas expresiones:

- **Proc:** Con esta expresión se procede a crear una clausura usando los símbolos, la expresión del cuerpo y el ambiente actual.

- **Apply:** Con esta expresión evaluamos el identificador de la función para obtener su clausura y evaluamos cada argumento entrante. Luego se evalúa el cuerpo de la clausura en el ambiente extendido con los símbolos de la clausura y los argumentos evaluados anteriormente.

Los argumentos de los procedimientos/funciones se escriben entre paréntesis y se separan con comas. Los booleans se expresan como **#T** o **#F**. La expresión de aplicación **apply** viene acompañada del procedimiento y de sus parámetros, entre paréntesis y se separados con comas. Además, ésta no se culmina con **end**.

- **Ejemplo de la figura 4 (Set)**

```
-->main{
     let (x=0)
     in{
        begin{
              set x := +(x, 10);
              x
        }
        end
     }
}
10
```

Figura 4

Para esta nueva expresión primero se obtiene la referencia correspondiente al símbolo de entrada, luego se busca la referencia más interna, es decir, la que apunta al valor expresado, y, con la expresión de entrada evaluada, se procede a actualizar la entrada del vector de blancos de acuerdo a la posición contenida en la referencia.

- **Ejemplo de la figura 5 (List)**

```
-->main{
     let(
          x = list(5, 6)
          y = 0
     )
     in{
          begin{
                set y := cdr(x);
                y
          }
          end
     }
}
(6)
```

Figura 5

```
-->main{
     let(
          x = list(5, 6)
          y = 0
     )
     in{
          begin{
                x
          }
          end
     }
}
(5 6)
```

Esta nueva expresión sólo crea una lista con los valores expresados resultantes de evaluar cada expresión entrante. Al declarar una lista, los elementos se separan por comas y van entre paréntesis. Sin embargo, al mostrarlas se separan solo con espacios. Las funciones de listas han sido implementadas como primitivas.

- **Ejemplo de la figura 6 (For)**

```
-->main{
     let (
          sum = proc(x)
                let (sum = 0)
                in{
                   begin{
                         for (y = 1)
                         to (x)
                         each(1)
                         do{
                            set sum := +(sum, y)
                         };
                         sum
                   }
                   end
                }
     )
     in{
        apply sum(10)
     }
}
55
```

Figura 6

Para esta expresión lo primero es evaluar las expresiones correspondientes al valor inicial de iteración, el límite de iteraciones y el incremento que va a tener cada iteración, luego se

extiende el ambiente con la variable y su valor inicial, si este aún no llega a su límite, se llama recursivamente la función actualizando el ambiente, el valor inicial y la salida a retornar, cuando la variable llegue a su límite se retorna el último valor almacenado en la salida.

Pare la iteración **for** se utilizan paréntesis para encerrar los parámetros y el valor de incremento del acumulador y llaves para el cuerpo.

- **Ejemplo de la figura 7 (Class, method, new, send)**

```
-->class cl{
          field i
          field j

          method cl(x){
                       begin{
                             set i := x;
                             set j := -(0, x)
                       }
                       end
          }
          method countup(d){
                            begin{
                                  set i := +(i, d);
                                  set j := +(j, d)
                            }
                            end
          }
          method getstate(){
                             list(i, j)
          }
}
main{
     let (
          t1 = 0
          t2 = 0
          o1 = new cl(3)
     )
     in{
        begin{
              set t1 := send o1 getstate();
              send o1 countup(2);
              set t2 := send o1 getstate();
              list(t1, t2)
        }
        end
     }
}
((3 -3) (5 -1))
```

Figura 7

- **Ejemplo de la figura 8 (Class, method, new, send)**

```
-->class interior_node{
                    field left
                    field right

                    method interior_node(l, r){
                                        begin{
                                              set left := l;
                                              set right := r
                                        }
                                        end
                    }
                    method sum(){
                                begin{
                                      +(send left sum(), send right sum())
                                }
                                end
                    }
}
class leaf_node{
                field value

                method leaf_node(v){
                                    set value := v
                }
                method sum(){
                            value
                }
}

main{
     let (
          ol = new interior_node(
                                new interior_node(new leaf_node(3), new leaf_node(4)),
                                new leaf_node(5)
               )
     )
     in{
        send ol sum()
     }
}
12
```

Figura 8

Para este caso tenemos cuatro nuevas expresiones:

- **Class:** Con esta expresión se procede a crear una declaración de clase, la cual contiene los símbolos de sus atributos, su nombre y sus declaraciones de métodos. Las declaraciones de clases se almacenan en una lista a la cual se accede cada vez que se necesite.
- **Method:** Con esta expresión se procede a crear una declaración de método, la cual contiene los símbolos de sus parámetros de entrada, su nombre y la expresión de su cuerpo. Las declaraciones de método hacen parte de las declaraciones de tablas y clases, por lo cual permanecen almacenadas en los respectivos ambientes de clases y tablas.

- **New:** Para esta expresión primero se evalúan las expresiones de entrada y luego se procede a crear un objeto con el nombre de la clase invocada y un vector vacío del tamaño de los símbolos contenidos en la declaración de clase correspondiente, luego se busca la declaración de método correspondiente al constructor de la clase y se evalúa su cuerpo en el ambiente extendido con las expresiones evaluadas al principio, tal y como se haría con la aplicación normal de un procedimiento.

Send: Para esta expresión se evalúa cada entrada y la expresión correspondiente al objeto con el que se invoca el método, luego se busca la declaración correspondiente al método y se repite el mismo procedimiento que con el método constructor del punto anterior.

- **Ejemplo de la figura 9 (Select)**

Figura 9

```
-->main{
     let (
          x = list(5, 6, 7, 8)
          y = 0
     )
     in{
        select (current)
        from (x)
        where (>(current, 6))
     }
}
(7 8)
```

- **Ejemplo de la figura 10 (Select)**

Figura 10

```
-->main{
     let (
          x = list(5, 6, 7, 8)
          y = 0
     )
     in{
        select (current)
        from (x)
        where (>(7, y))
     }
}
(5 6 7 8)
```

Para esta expresión primero se evalúa la parte correspondiente a la lista de entrada, luego se extiende el ambiente con el símbolo referente al elemento actual y el primer elemento de la lista y se procede a evaluar la expresión correspondiente a la condición de inclusión en la nueva lista, en el caso positivo se hace uso de cons con este elemento y se hace un llamado recursivo a la función con el resto de la lista, en caso negativo se ignora el elemento y sólo se hace el llamado recursivo, al final de todos los llamados se concatena con una lista vacía y se retorna la nueva lista.

- **Ejemplo de la figura 11 (Table, insert, table.(_____).(_____))**

```
-->table student{
              field name
              field grade

              method accessName(){
                                  name
              }
              method passed(){
                              if (>(grade, 3)){
                                                  |passed|
                              } else{
                                      |failed|
                              }
              }
}
main{
     let(t1 = |paola| t2 = |alexander|)
     in{
        begin{
              insert(t1, 4) in student;
              insert(t2, 2) in student;

              list(
                   table.(access student(0)).passed(),
                   table.(access student(1)).passed()
              )
        }
        end
     }
}
("|passed|" "|failed|")
```

Figura 11

Para este caso tenemos tres nuevas expresiones:

- **Table:** Con esta expresión se procede a crear una declaración de tabla, la cual contiene los símbolos de sus atributos, su nombre y sus declaraciones de métodos. Las declaraciones de tablas se almacenan en una lista a la cual se accede cada vez que se necesite, todo de manera análoga a las declaraciones de clase.
- **Insert:** Con esta expresión primero se procede a evaluar cada expresión de entrada y almacenarlas en un vector, luego se genera una posición contando el número de registros correspondientes a una tabla según el nombre de la tabla suministrada, luego se actualiza el ambiente de registros, el cual es una lista, con un nuevo registro que contiene el nombre de su tabla, su posición en ella y un vector con los campos respectivos.
- **Table.(**____).(____): Para esta expresión se evalúa cada entrada y la expresión correspondiente al registro con el que se invoca el método, luego se busca la declaración

correspondiente al método y se aplica de la misma manera que hemos venido haciendo en la parte orientada a objetos.

- **Ejemplo de la figura 12 (selectTable)**

```
-->table student{
              field name
              field grade

              method accessName(){
                                  name
              }
              method getgrade(){
                                grade
              }
              method passed(){
                              if (>(grade, 3)){
                                                  |passed|
                              } else{
                                    |failed|
                              }
              }
}
main{
     let(t1 = |andrea| t2 = |alexander| t3 = |paola| t4 = |jairo| t5 = |guillermo|)
     in{
        begin{
              insert(t1, 4) in student;
              insert(t2, 2) in student;
              insert(t3, 4) in student;
              insert(t4, 2) in student;
              insert(t5, 1) in student;

              selectTable(table.(current).accessName())
              from student
              where(>=(table.(current).getgrade(), 3))
        }
        end
     }
}
("|andrea|" "|paola|")
```

Figura 12

Para esta expresión primero se extraen los registros correspondientes a la tabla de entrada, luego se extiende el ambiente con el símbolo obtenido de la expresión de acceso a la tabla y el primer elemento de la lista de registros extraídos, luego se procede a evaluar la expresión correspondiente a la condición de inclusión en la nueva lista, en el caso positivo se hace uso de cons con la evaluación de la expresión de acceso del registro actual y se hace un llamado recursivo a la función con el resto de registros, en caso negativo se ignora la evaluación y sólo se hace el llamado recursivo, al final de todos los llamados se concatena con una lista vacía y se retorna la nueva lista.