Proyecto Final
Fundamentos de Lenguajes Programación

Carlos Andres Delgado S, Msc
carlos.andres.delgado@correounivalle.edu.co
Mayo de 2024

1. Reglas del Proyecto

1. No se permite copiar código de Internet ni de sus compañeros. Si se encuentra código copiado el proyecto será anulado por completo.

2. Puede trabajar en grupos de hasta 4 personas

3. La entrega del proyecto se realizará con un enlace a un repositorio Github o similar en el campus virtual, este debe contener el código fuente y el informe en formato PDF

4. Este proyecto debe ser sustentado, la nota de sustentación es individual y va entre 0 y 1. Esta nota es multiplicada por la nota obtenida en el proyecto. Por ejemplo si su grupo obtuvo 5.0 en nota y usted 0.5 en la sustentación, su nota será 2.5. La sustentacion será el Jueves 20 de Junio de 7am a 1pm en el salón de clase.

2. El temible Meta-lenguaje: !El concilio de los paradigmas de programación!

La Universidad del Valle, seccional Tulua, ha priorizado los programas STEM (Ciencia, tecnología, ingeniería y matemáticas), por ello ha decidido que los programas en Tecnología en Desarrollo de Software e Ingeniería de Sistemas incrementen su cupo de admisión a ∞, pero tenemos el problema de la disponibilidad de las salas, por ello ha decidido fusionar los cursos de fundamentos de programación imperativa, fundamentos de programación O.O, programación orientada e eventos, programación funcional, fundamentos de lenguajes de programación e infraestructuras paralelas en un super curso llamado `Meta programación modalidad bootcamp`, esto posibilita que los estudiantes adquieran más de 20 resultados de aprendizaje en un semestre. Para implementar esto ningún lenguaje de programación existente puede cumplir esta tarea, por ello debe desarrollarse uno totalmente nuevo. Usted y su grupo de trabajo ha encontrado al director de la sede muy triste sentado en una banca del parque chilocote, este le cuenta este problema, por lo que ustedes deciden ayudarlo desarrollando este lenguaje de programación.

Figura 1: Recreación del profesor en el lago Chilocote

2.1. Especificacion del lenguaje

Los valores expresados son:

1. Números (tenemos varias representaciones)
2. Texto
3. Listas
4. Void (expresión vacía)

Los valores denotados son arboles de sintaxis abstracta, es decir **en el ambiente almacenamos de esta forma los valores**

La sintaxis y semántica adaptada a la librería SLLGEN.

2.2. Gramática del proyecto

Al final de cada producción aparece el nombre de la variante entre paréntesis.

```
;;**************Especificaciones léxicas*****************
<digitoBinario> ::= "b" (0|1)(0|1)*
<digitoDecimal> ::= <digito><digito>*
<digitoDecimal> ::= "-" <digito><digito>*
<digitoOctal> ::= "0x" (0|1|2|3|4|5|6|7) (0|1|2|3|4|5|6|7)*
<digitoOctal> ::= "-" "0x" (0|1|2|3|4|5|6|7) (0|1|2|3|4|5|6|7)*
<hexadecimal> ::= "hx" (0|1|2|3|4|5|6|7|"A"|"B"|"C"|"D"|"E"|"F")
                  (0|1|2|3|4|5|6|7|"A"|"B"|"C"|"D"|"E"|"F")*
<hexadecimal> ::= "-" "hx" (0|1|2|3|4|5|6|7|"A"|"B"|"C"|"D"|"E"|"F")
                  (0|1|2|3|4|5|6|7|"A"|"B"|"C"|"D"|"E"|"F")*
<flotante> ::= <digito><digito>*"."<digito><digito>*
<flotante> ::= "-" <digito><digito>*"."<digito><digito>*
<identificador> ::=  <simbolo><simbolo>*
<comentario> ::=  "\\" (not #\newline)
<espacio>  ::= <whitespace>


;;**************Especificaciones gramaticales*****************

<programa>  ::= <struct-decl>* <expresion> (a-programa)
```

```
<expresion> ::= <bool-expresion>  (bool-exp)
            ::= <identificador>   (var-exp)
            ::= <numero>      (num-exp)
      ::=   '"' <identificador> <identificador>* '"' (cadena-exp)
        ::= "list" "(" (<expresion>)*(,) ")" (lista-exp)
      ::= "cons" "(" <expresion>  <expresion> ")" (cons-exp)
      ::= "empty" (empty-list-exp)
            ::= "array" "(" (<expresion>)*(,) ")" (array-exp)
      ::= <var-decl> (decl-exp)
      ::= "void" (void-exp)
            ::= "set" <identificador> "=" <expresion> (set-exp)

      ;;Primitiva numérica
            ::= "(" <expresion> <primitiva> <expresion> ")" (prim-num-exp)
      ;;Primitiva listas
      ::= primitivalListas "(" <expresion> ")" (prim-list-exp)
        ;;Primitiva booleanas
      ::= primitivaBooleana "(" <expresion>*(,) ")" (prim-bool-exp)
      ;;Primitiva binaria
      ::= primitivaBinaria "(" <expresion>*(,) ")" (prim-bin-exp)
      ;;Primitiva de arrays
      ::= primitivaArray "(" <expresion>*(,) ")" (prim-array-exp)
      ;;Primitiva de cadenas
      ::= primitivaCadena "(" <expresion>*(,) ")" (prim-cad-exp)

      ;;Condicionales y estructuras de control
        ::= "if" <expresion> "{" <expresion> "else" <expresion> "}" (if-exp)
        ::= "switch" "(" expresion ")" "{" ( "case" expresion ":" expresion)* "default" ":"
            ↪ expresion"}" (switch-exp)
        ::= "for" <identificador> "from" <expresion> "until" expresion "by" expresion "do"
            ↪ expresion (for-exp)
        ::= "while" <expresion> "{"  <expresion> "}" (while-exp)
        ;;Secuenciación y asignación
        ::= "begin" expresion (";" <expresion>)* "end" (begin-exp)
      ::= "set" <identificador> "=" <expresion> (set-exp)
        ;;Funciones
            ::= "func" "(" (<identificador>*(,) ")" <expresion>  (func-exp)
        ::= "call" <expresion> "(" <expresion>*(,) ")" (call-exp)
        ;;Invocación y llamado de estructuras
        ::= "new"  <identificador> "(" <expresion>*(,) ")" (new-struct-exp)
      ::= "get" <expresion> "." <identificador> (get-struct-exp)
      ::= "set-struct" <expresion> "." <identificador> "=" <expresion> (set-struct-exp)
      ;;Reconocimiento de patrones
      ::= "match" <expresion> "{" (<regular-exp> "=>" <expresion>)* "}" (match-exp)

<numero-exp>::=  <digitoDecimal> (decimal-num)
        ::=  <digitoBinario> (bin-num)
            ::=  <digitoOctal> (octal-num)
            ::=  <digitoHexadecimal> (hex-num)
            ::=  <flotante> (float-num)

<bool-expresion>
          ::= "true"   (true-exp)
              ::= "false"   (false-exp)

        ;;Primitivas
<primitiva> ::= + (sum-prim) ;;Suma
            ::= - (minus-prim) ;;Resta
            ::= * (mult-prim) ;;Multiplicación
            ::= mod (modulo-prim) ;;Modulo
            ::= pow (elevar-prim) ;;Potencia
            ::= < (menor-prim) ;;Menor que
            ::= > (mayor-prim) ;;Mayor que
            ::= <=(menorigual-prim) ;;Menor igual que
            ::= >=(mayorigual-prim) ;;Mayor igual que
            ::= !=(diferent-prim) ;;Diferente
            := ==(igual-prim) ;;Igual
<primitivaBooleana>
            ::= and (and-prim) ;;Conjunción
            ::= or (or-prim)   ;;Disyunctión
            ::= xor (xor-prim)  ;;Or exclusivo
```

```
            ::= not (not-prim) ;;negación
<primitivaListas>
            ::= first (first-primList) ;;Primero
            ::= rest (rest-primList) ;;Resto
            ::= empty? (empty-primList) ;;Vacio
<primitivaBinaria>
          ::= && (and-primBin) ;; and binario
          ::= || (or-primBin)  ;; or binario
          ::= ^^ (xor-primBin)  ;; xor binario
          ;;Declaración variables
<primitivaArray>
          ::= length (length-primArr) ;;Tamaño
          ::= index (index-primArr) ;;Acceso a elemento
          ::= slice (slice-primArr) ;;Acceso a subarray
          ::= setlist (setlist-primArr) ;;Cambiar elemento
<primitivaCadenas>
          ::= concat (concat-primCad) ;;concatenar
          ::= string-length (length-primCad) ;;Longitud
          ::= elementAt (index-primCad) ;;Acceso a elemento

<var-decl> ::= "var" (<identificador> "=" <expresion>)* "in" expresion (lvar-exp)
           ::= "let" <identificador> "=" <expresion>)* "in" exprsion let-exp)

        ;;Estructuras de datos

<struct-decl> ::=  "struct" <identificador> "{" <identificador>* "}"  (struct-exp)

         ;;match
<regular-exp> ::= <identificador> "::" <identificador> (list-match-exp) ;;listas
          ::= "numero" "(" <identificador> ")" (num-match-exp)  ;;numeros
          ::= "cadena" "(" <identificador> ")" (cad-match-exp)  ;;cadenas
          ::= "boolean" "(" <identificador> ")" (bool-match-exp) ;;booleanos
          ::= "array" "(" (<identificador>",")* ")" (array-match-exp) ;;arreglos
          ::= "empty" (empty-match-list)  ;;lista vacia
          ::= "default" (default-match)  ;;default
```

2.3. Valores denotados y expresados

Los valores denotados son referencias, dado que manejamos asignaciòn

Los valores expresados son números (en todas sus formas), booleanos, cadenas, void y valores procedimiento.

2.4. Estructuras léxicas

- **Identificadores:** Son secuencias de caracteres alfanuméricos que comienzan con una letra
- **Comentarios:** Inician con // y terminan con fin de linea.
- **Números enteros:** Los números son un conjunto de dígitos. Se dividen de la siguiente forma:

  1. Dígitos binarios del 0 y 1
  2. Dígitos decimales del 0 al 9
  3. Dígitos octales del 0 al 7
  4. Dígitos hexadecimales del 0 al F.

  Estos números pueden ser antecedidos por ”para indicar que son negativos.

- **Números flotantes:** Los números son un conjunto de dígitos seguidos por un punto y seguidos por un conjunto de dígitos. Estos números pueden ser antecedidos por "para indicar que son negativos. **únicamente tenemos números decimales flotantes**
- **Cadenas:** Inician con comillas seguidas por un conjunto de dígitos o letras y terminadas en comillas, estas admiten espacios :).

En el archivo `proyecto.rkt` se especifica la gramática completa

2.5. Datos

Los binarios se pueden expresar así:

```
b10101010
-b01010101
```

Los decimales

```
23213
-12312
```

Los octales

```
0x213345
-0x23123
```

Los hexadecimales

```
hxFAB123
-hx99EA
```

Los flotantes

```
412312.2312
-23123.2312
```

Los booleanos

```
true
false
```

Las cadenas de texto:

```
"hola mundo"
"hola que tal"
```

2.6. Primitivas numéricas

Para este proyecto se tienen las primitivas numéricas, estas deben retornar el mismo tipo numérico que se envio, ejemplo

```
(b1000 + b10)
```

Debe retornar b1010

```
(0x77 + 0x3)
```

Debe retornar 0x102

```
(hx100 - hx2)
```

Debe retornar FE

```
(123.2 - 1.2)
```

Debe retornar 122.0

2.7. Primitivas booleanas

Las primitivas booleanas esperan expresiones de valor expresado booleano, ejemplo:

```
and((1 > 2), false)
```

Observese que la primitiva numérica > retorna un booleano

2.8. Listas

Las listas son agrupaciones de elementos, los cuales son inmutables (no pueden cambiar), se declaran de la siguiente forma:

Ejemplo:

```
list(1,2,3,4)
cons(1 cons(2 empty))
empty
```

Para la listas vamos a usar la representación de cabeza que es cualquier elemento y cola que es una lista.

2.9. Primitivas de listas

Para las listas tenemos las siguiente primitivas

```
let
   l = cons(1 cons(2 cons(3 empty)))
   in
     list(first(l), rest(l), empty?(rest(l)))
```

Va a retornar (1,list(2,3), false)

2.10. Arrays

Las arrays o arreglos son colecciones la listas ya que pueden cambiar.

Ejemplo:

```
array(1,2,3,4,5)
```

2.11. Primitivas de arrays

Las primitivas que tenemos son de acceso, modificación, longitud y slice.

Para conocer el tamaño tenemos lenght:

```
let
  t = array(1,2,3,4,5)
  in
    length(t)
```

Debe retornar 5

Para acceder a un elemento, el cual indexamos desde 0, tenemos index:

```
let
  t = array(1,2,3,4,5)
  in
    index(t, 2)
```

Debe retornar 3

Para acceder a sub-arreglo tenemos slice, el cual tendrá inicio y fin (ambos incluidos), ejemplo:

```
let
  k = list(1,2,3,4,5,6,7,8,9,10)
  in
    slice(k, 2, 5)
```

Debe retornar list(3,4,5,6)

Para cambiar un elemento tenemos setlist, el cual recibe el arreglo, la posición y el valor:

```
let
  t = array(1,2,3,4,5)
  in
    setlist(t, 2, 10)
```

Debe retornar array(1,2,10,4,5)

2.12. Primtivas de cadenas

Para las primitivas de candenas tenemos las siguientes:

2.12.1. length

```
let
  s = "hola mundo cruel"
  in
    string-length(s)
```

Debe retornar 16

2.12.2. elementAt

Este recibe una cadena y una posición numérica que inicia en 0.

```
let
  s = "hola mundo cruel"
  in
    elementAt(s, 5)
```

Debe retornar “m”

2.12.3. concat

Permite pegar dos o más cadenas

```
let
  a = "hola"
  b = "mundo"
  c = "cruel"
  in
    concat(a,b,c)
```

Retornará “holamundocruel”

2.13. Entornos de ligaduras y variables

Se tienen los entornos `let` el cual crea ligadura **no modificable** y `var` el cual crea ligadura **modificable**, ambos trabajan similar, sólo que el var permite el uso de set y el let no.

```
var x = 10
  in begin set x = 20; x end
```

Retornara 20.

```
let x = 10
  in begin set x = 20; x end
```

Fallará porque no se puede modificar la ligadura.
Importante: Estos entornos son naturalmente recursivos:

```
var x = x in x
```

Se quedará en un ciclo infinito.
Aqui también podemos aplicar funciones recursivas:

```
var f = func(x) if (x == 0) { 0 else (x + call f((x - 1))) }
  in call f(10)
```

Retornará 55

2.14. Condicionales

Retornará 40, dado que el x cambió a 30.

```
yif (a > 3) { 1 else 2 }
```

2.15. Estructuras de control

2.15.1. For

La estructura for es de la siguiente forma:

```
let
  x = 0
  in
    begin
      for i from 0 until 10 by 1 do set x = (x + i);
      x
    end
```

Retornará 45

2.15.2. While

El while es de la siguiente forma:

```
let
  x = 0
  in
    begin
      while (x < 10) { set x = (x + 1) };
      x
    end
```

Retornará 10

2.15.3. Switch

El switch es de la siguiente forma:

```
let
  x = 0
  in
    begin
      switch (x) {
        case 1: 1
        case 2: 2
        default: 3
      }
    end
```

Retornará 3

2.16. Begin y set

El begin permite ejecutar una serie de instrucciones, siempre retornará la última:

```
begin 1; 2; 3 end
```

Retornará 3

El set permite cambiar el valor de una variable, su valor de retorno es void-exp.

```
let
  x = 0
  in
    set x = 10
```

Retornará void-exp

2.17. Funciones

Para las funciones tenemos func para su creación y call para su aplicación:

```
var
  f = func(x) x
  in
    call f(10)
```

Retornará 10.

Importante: Como el paradigma es imperativo las funciones si se ven afectadas por el valor de una variable externa, es decir trabajarán con el que vean. Ejemplo:

```
  var x = 10 f = func(a) (a + x)
    in
    let  x = 30 in call f(10)
```

2.18. Estructuras de datos

Las estructuras de datos son similares a los objetos, solamente que no poseen métodos, estas se deben declarar y posteriormente se pueden extraer sus datos.

```
 struct perro {nombre edad color}
let
  t = new perro("lucas",10,"verde")
  in get t.nombre
```

Retonará “lucas”.

También podemos cambiar datos:

```
struct perro {nombre edad color}
let
  t = new perro("lucas",10,"verde")
  in
    begin
      set-struct t.nombre = "pepe";
      get t.nombre
    end
```

Retornará “pepe”.
Debe tomar en cuenta que set-struct retorna void como toda operación de efecto y no valor.

```
struct perro {nombre edad color}
let
  t = new perro("lucas",10,"verde")
  in
    set-struct t.nombre = "pepe"
```

Retornará void-exp.

3. Reconocimiento de patrones

Para el reconocimiento de patrones usaremos match, podremos reconocer

1. Listas con expresiones del tipo x::xs
2. Números con expresiones del tipo numero(x)
3. Cadenas con expresiones del tipo cadena(x)
4. Booleanos con expresiones del tipo boolean(x)
5. Arrays con expresiones del tipo array(x,y,z), el último capturará el resto de elementos, si se pasa del tamaño fallará.
6. Listas vacías con empty

Ejemplos:

```
let
  x = list(1,2,3,4,5)
  in
    match x {
      x::xs => xs
      default => 0
    }
```

Retornará list(2,3,4,5)

```
let
  x = 10
  in
    match x {
      numero(x) => x
      default => 0
    }
```

Retornará 10

```
let
  x = "hola mundo"
  in
    match x {
      cadena(x) => x
      default => 0
    }
```

Retornará “hola mundo”

```
let
  x = true
  in
    match x {
      boolean(x) => x
      default => 0
    }
```

Retonará true

```
let
  x = array(1,2,3,4,5)
  in
    match x {
      array(x,y,z) => list(x,y,z)
      default => 0
    }
```

Retornará list(1,2,array(3,4,5))

Por ejemplo una función recursiva sobre listas usando empty.

```
  let
    f = func(x)
      match x {
        empty => 0
        x::xs => (x + call f(xs))
      }
    in
    call f(list(1,2,3,4,5))
```

Retornará 15

4. Entrega

Se debe entregar un enlace de github con el informe en formato PDF y el código del proyecto (archivo rkt), recordar dejar el enlace público o agregar como colaborador al profesor (@cardel). El informe debe contener capturas de pantalla con las pruebas realizadas, en las rùbricas de evaluación se indicarán cómo deben ir.

5. Evaluación

El proyecto debe ser entregado a más tardar el día 20 de Junio de 2024 a las 23:59:59 hora de Colombia, a partir de allí aplicará 0.3 por hora o fracción de retraso, por ejemplo, si usted entrega el 20 de Junio a las 00:01:00, se le descontará 0.3 de la nota obtenida, si entrega a las 01:00:01 se le descontará 0.6 y así sucesivamente.

Así mismo se aplicará las reglas expresadas al inicio del enunciado con respecto a la sustentación como nota individual. La evaluación del proyecto se aplicará vía rubríca, la cual será socializada a los estudiantes en el transcurso de los próximos días.