Proyecto de Curso
Lenguaje SubOz

Fundamentos de Lenguajes Programación
Escuela de Ingeniería de Sistemas y Computación
Universidad del Valle

Profesor Carlos Alberto Ramírez Restrepo
carlos.a.ramirez@correounivalle.edu.co

1. Introducción

El presente proyecto tiene por objeto enfrentar a los estudiantes del curso:

- A la implementación de un lenguaje de programación.
- Al uso de ayudas para la creación de interpretadores y compiladores.
- Al análisis de estructuras sintácticas, de datos y de control de un lenguaje de programación para la optimización de un interpretador o compilador asociado a él.

2. SubOz

Oz es un lenguaje de programación multiparadigma que integra muy bien y de forma muy coherente los diferentes modelos computacionales. En este proyecto se requiere que usted implemente un subconjunto de Oz llamado *SubOz*.

En las siguientes secciones de explica y describe la gramática y semántica de SubOz y se dan algunos ejemplos de utilización.

2.1. Semántica operativa y declarativa

En esta sección se describe la semántica declarativa (gramática) y operativa (semántica) del lenguaje

SubOz.

2.1.1. Programa

Un programa está compuesto por una sola expresión. El resultado de evaluar un programa consiste de evaluar la expresión por la que está compuesto, en un ambiente inicial que puede contener funciones predeterminadas del sistema. SubOz realiza un chequeo dinámico de tipos. En SubOz no todas las expresiones retornan algo, asi que debe tener eso en cuenta para cuando evalue el programa.

⟨programa⟩ ::= ⟨expresión⟩
`a-program (exp)`

2.1.2. Cuerpo de uso general

Existe una variante gramatical que se usará mucho y es el *cuerpo*. Esta variante se utiliza cuando se necesita definir la ejecución secuencial de una o más expresiones en orden. El resultado de evaluar un cuerpo es el resultado de la evaluación de la última expresión.

⟨cuerpo⟩ ::= **{**⟨expresión⟩**}**$^+$
`cuerpoc(exps)`

2.1.3. Variables

Las variables en Oz son de una sola asignación, esto es, una vez se asigne el valor de una variable, no es posible cambiar su valor. Suboz también posee esta característica. La asignación se hace utilizando la expresión *set* o a través de la primitiva de unificación. Una variable es cualquier identificador que comienze con una letra mayuscula. Ejemplos de variables son: `X`, `Var`, `Uno`, entre otros. De igual forma, también es posible crear variables anónimas. Las variables anónimas son aquellas que se comportan como variables pero que no tienen nombre. Son muy útiles para realizar la unificación. La creación de variables anónimas se realiza utilizando el símbolo _ (underscore, en inglés).

Existe una diferencia en el trato de variables entre Oz y SubOz debido a que Oz soporta hilos (modelo computacional *concurrente*) y SubOz no. Cuando se tiene un sistema concurrente y es necesario conocer el valor de una variable, el sistema puede simplemente esperar a que la variable se determine, pues un hilo podría hacerlo. Mientras que en SubOz dado que no exiten hilos, si un proceso necesita conocer el valor de una variable que no está determinada, no tiene caso esperar pues nadie la va a determinar. Por esta razón si en algún momento se necesita conocer el valor de una variable para realizar una operación y la variable no está determinada entonces se debe mostrar un mensaje de error. Note que no todas las operaciones necesitan conocer el valor de una variable (ej. la unificación o la primitiva `isdet?`).

Es importante resaltar que aunque las variables sean de una sola asignación, es posible realizar programación imperativa utilizando mecanismos como *celdas* o *puertos*.

⟨expresion⟩ ::= ⟨variable⟩
`var-exp(id)`

2.1.4. Creación de variables locales

La creación de variables locales se hace utilizando la expresión `local`, que crea las variables localmente e inicialmente sin determinar (sin ningún valor asociado).

⟨expresion⟩ ::= **local {**⟨variable⟩**}**$^+$ **in** ⟨cuerpo⟩ **end**
`local-exp(vars cuerpo)`

2.1.5. Asignación

La asignación de variables es llevada a cabo mediante las expresiones `set`. La expresión `set` consta de dos subexpresiones que son evaluadas y sus variables son unificadas.

⟨expresion⟩ ::= **set** ⟨expresion⟩ **=** ⟨expresion⟩
`set-exp(exp1 exp2)`

2.1.6. Operaciones sobre variables

Existen dos primitivas básicas que permiten obtener información de las variables.

- `isdet?(t)` retorna `true` si la variable `t` está asignada a algún valor y `false` si no.

⟨primitiva⟩ ::= "**isdet?**"
`isdet-prim`

- `isfree?(t)` retorna `true` si la variable `t` no está asignada a algún valor y `false` si no.

⟨primitiva⟩ ::= "**isfree?**"
`isfree-prim`

2.1.7. Números

Existen dos clases de números, las operaciones entre dos tipos diferentes de números no están permitidas. Por ejemplo $+(2,5{,}4)$ debería mostrar un error de tipos. Los números negativos inician con el símbolo virgulilla " ".

⟨expresion⟩ ::= ⟨entero⟩
`entero-exp(ent)`
⟨expresion⟩ ::= ⟨flotante⟩
`flotante-exp(flt)`

2.1.8. Registros

Los registros juegan un papel muy importante pues con estos se puede modelar muchas cosas como listas, vectores y símbolos. En SubOz los registros están compuestos de una *etiqueta* y a veces contienen más información en *campos*, cada uno con un *valor*. Los campos de un registros pueden ser números o identificadores. Las etiquetas y campos son cualquier identificador que comience con una letra minúscula o este encerrado entre apóstrofes. Note que $'a'$ es lo mismo que `a`.

A los registros que están compuestos solo por su etiqueta se les llama *átomos*. Existen átomos especiales como `nil`, `true`, `false`; que denotan respectivamente una lista, la expresión verdadera y la expresión falsa. Ejemplos de átomos son: `nil`, `hola`, `carro`, `'Atomo'`, '|', '234', entre otros.

Un caso especial de registros se utiliza para la construcción de parejas. Este registro utiliza como etiqueta el identificador '|' y tiene dos campos 1 y 2. Las listas en SubOz están construidas utilizando este registro. La construcción de listas se puede hacer de dos formas. La primera es utilizando los registros '|' y la otra forma es utilizar el constructor `[e1 e2 ... en]` que construye una lista (utilizando el registro '|') de n elementos. Por ejemplo las expresiones `[a b 1 2]` y .'|'(h:a t:'|'(h:b t:'|'(h:1 t:'|'(h:2 t:nil)))) son equivalentes.

En SubOz para acceder a un campo de un registro se utiliza la expresión: . < *expresionderegistro* > . < *campo* >.

⟨expresion⟩ ::= ⟨identificador⟩
{"("**{**⟨identificador⟩" : "⟨expresion⟩**}**")"**}**$^{?}$
`record-exp(label features exps)`

⟨expresion⟩ ::= "["**{**⟨expresion⟩**}**$^{+}$"]"
`list-exp(exps)`

2.1.9. Funciones y procedimientos

En SubOz los procedimientos y funciones se diferencian en que estás últimas retornan el valor de lo que retorne la última instrucción del programa, mientras que los procedimientos no retornan valores.

El llamado a una función o procedimiento se realiza mediante la construcción `app-exp`. Se debe crear variables frescas para los parámetros formales, y después cada argumento se unifica con cada variable creada por cada parámetro formal. Esto implica que todas las variables son pasadas por referencia.

Una característica importante que se encuentra en las funciones y procedimientos es el nombre del procedimiento. Cuando se crea una función o procedimiento, se liga la variable del nombre al procedimiento. Si se desea que el procedimiento sea anónimo entonces se puede utilizar como nombre el símbolo $ que crea y devuelve un procedimiento anónimo (así como la función `lambda` en Scheme o `proc` en el lenguaje del curso).

⟨expresion⟩ ::= **proc** "{" ⟨nombre-proc⟩ **{**⟨variable⟩**}*** "}"
⟨cuerpo⟩
end
`proc-exp(nombre vars cuerpo)`

⟨expresion⟩ ::= **fun** "{" ⟨nombre-proc⟩ **{**⟨variable⟩**}*** "}"
⟨cuerpo⟩
end
`fun-exp(nombre vars cuerpo)`

⟨expresion⟩ ::= "{" ⟨expresion⟩ **{**⟨expresion⟩**}*** "}"
`app-exp(rator rands)`

Nombre para procedimientos y funciones

⟨nombre-proc⟩ ::= "$"
`nombre-proc-anon`

⟨nombre-proc⟩ ::= ⟨variable⟩
`nombre-proc-var(nombre-var)`

2.1.10. Ejecución condicional

La expresión condicional `if` compara la evaluación de la expresión condicional con el átomo `true`, y si son iguales entonces ejecuta el código asociado al `then`. Si la expresión condicional es `false` entonces se ejecuta el código asociado al `else` si existe, sino entonces no se ejecuta código alguno. La expresión condicional tiene que ser `true` o `false`.

Para explicar la expresión `case` utilizaremos el siguiente ejemplo:

```
case E
of P1 then C1
[] P2 then C2
.
.
.
[] PN then CN
else
  CE
end
```

Aquí se evalua la expresión `E` y se verifica si es posible unificar el término `E` con `P1` (las variables en `P1` son variables frescas). Si es posible entonces se crea un ambiente K y se retorna la evaluación de $C1$ en ese ambiente. El ambiente K se crea asociando cada variable en `P1` al valor que le corresponda después de la unificación de la evaluación de la expresión `E` con `P1`.

⟨expresion⟩ ::= **if** ⟨expresion⟩ **then** ⟨cuerpo⟩
{else⟨cuerpo⟩**}**$^{?}$ **end**

`if-exp(cond-exp true-exp false-exp)`

⟨expresion⟩ ::= **case** ⟨expresion⟩
of ⟨patron⟩ **then** ⟨cuerpo⟩
{“[] ”⟨patron⟩ **then** ⟨cuerpo⟩ **}***
{else ⟨cuerpo⟩**}***

```
case-exp(eval-exp pat conseq pats conseqs else)
```

Sintaxis patrón

⟨patron⟩ ::= ⟨identificador⟩
{“(”**{**⟨identificador⟩“ : ”**}***“)”**}**$^?$

```
record-pat(label idents patrs)
```

⟨patron⟩ ::= ⟨variable⟩

```
var-pat(var-name)
```

⟨patron⟩ ::= ⟨entero⟩

```
int-pat(int)
```

⟨patron⟩ ::= ⟨flotante⟩

```
float-pat(flt)
```

2.1.11. Especiales

La expresión `skip` no realiza acción alguna. Tampoco retorna valor alguno.

⟨expresion⟩ ::= **skip**

```
skip-exp
```

2.1.12. Primitivas

⟨expresion⟩ ::= ⟨primitiva⟩“{”**{**⟨expresion⟩**}***“}”

```
prim-exp(prim rands)
```

Operaciones aritméticas

- $+(v_1, ..., v_n)$: Suma los número v_1 hasta v_n. Si no son del mismo tipo entonces ocurre un error.
  ⟨primitiva⟩ ::= “ + ”
  `sum-prim`

- $*(v_1, ..., v_n)$: Multiplica los número v_1 hasta v_n. Si no son del mismo tipo entonces ocurre un error.
  ⟨primitiva⟩ ::= “ * ”
  `mult-prim`

- $-(v_1 v_2)$: Retorna $v_1 - v_2$. Si no son del mismo tipo entonces ocurre un error.
  ⟨primitiva⟩ ::= “ − ”
  `sub-prim`

- $/(v_1 v_2)$: Retorna v_1/v_2. Si no son del mismo tipo entonces ocurre un error. El valor retornado es del mismo tipo de v_1 y v_2.
  ⟨primitiva⟩ ::= “/”
  `div-prim`

Comparación

Cuando las comparaciones se realizan, siempre retornan el átomo `true` o el átomo `false`. Cuando se comparan átomos, se hace una comparación lexicográfica. Por ejemplo las siguientes comparaciones son `true <(a,b) <(aa, ab)`.

- $< (t_1, t_2)$: Retorna `true` si t_1 es menor que t_2.
  ⟨primitiva⟩ ::= “ < ”
  `menor-prim`

- $<= (t_1, t_2)$: Retorna `true` si t_1 es menor o igual que t_2.

⟨primitiva⟩ ::= " =< "
`meneq-prim`

- $>(t_1, t_2)$: Retorna `true` si t_1 es mayor que t_2.

  ⟨primitiva⟩ ::= " > "
  `mayor-prim`

- $>=(t_1, t_2)$: Retorna `true` si t_1 es mayor o igual que t_2.

  ⟨primitiva⟩ ::= " >= "
  `mayig-prim`

- $==(t_1, t_2)$: Retorna `true` si t_1 es el mismo termino que t_2. No pueden existir variables libres ni en t_1 ni en t_2.

  ⟨primitiva⟩ ::= " == "
  `igual-prim`

Lógicas

La primitiva `orelse` realiza un OR lógico y retorna la comparación. La primitiva `andthen` realiza un AND lógico y retorna la comparación. Solo se aceptan tipos booleanos.

- $orelse(t_1, t_2)$:

  ⟨primitiva⟩ ::= "*orelse*"
  `orelse-prim`

- $andthen(t_1, t_2)$:

  ⟨primitiva⟩ ::= "*andthen*"
  `andthen-prim`

Unificación

- $=(t_1, t_2)$: Realiza la unificación de dos términos t_1 y t_2. Ver algoritmo de unificación en la sección 2.1.13.

  ⟨primitiva⟩ ::= " = "
  `unif-prim`

Operaciones sobre puertos

- $newcell(v)$: Crea un nuevo puerto asociado al flujo `s`.

  ⟨primitiva⟩ ::= "*newport*"
  `newport-prim`

- $isport?(p)$: Retorna `true` si `p` es un puerto. Si no entonces `false`.

  ⟨primitiva⟩ ::= "*isport?*"
  `isport-prim`

- $send(p, v)$: Envia el dato `v` usando el puerto `p`.

  ⟨primitiva⟩ ::= "*send*"
  `send-prim`

Operaciones de celdas

- $newcell(v)$: Crea una nueva celda usando `v` como valor inicial.

  ⟨primitiva⟩ ::= "*newcell*"
  `newcell-prim`

- $iscell?(c)$: Retorna `true` si `c` es una celda. Si no entonces `false`.

  ⟨primitiva⟩ ::= "*iscell?*"
  `iscell-prim`

- @(c): Retorna el valor que almacena `c`.

  ⟨primitiva⟩ ::= "@"
  `acces-prim`

- *setcell*(c, v): Asigna a la celda `c` el valor `v`.

  ⟨primitiva⟩ ::= "*setcell*"
  `setcell-prim`

2.1.13. Unificación

El algoritmo de unificación tiene como objetivo unificar los valores y las variables en dos términos t_1 y t_2. Las consecuencias de la unificación de dos términos son descritas a continuación (en la siguiente descripción no importa el orden de t_1 y t_2):

- Si t_1 es un registro entonces tiene que ocurrir todo lo siguiente (si alguno falla, la unificación falla):
  - t_2 tiene que ser un registro.
  - t_1 y t_2 tienen la misma etiqueta.
  - t_1 y t_2 tienen los mismos campos (número y nombres).
  - La unificación de cada campo de t_1 con su respectivo campo en t_2 debe ser exitosa.
- Si t_1 y t_2 son el mismo término, la unificación no tiene problema.
- Si t_1 es una variable libre entonces t_1 hará referencia al mismo valor que haga referencia t_2. Esto quiere decir que si t_2 es un término o variable no libre, t_1 hará referencia al mismo valor; pero si t_2 es una variable libre entonces t_1 y t_2 se convertiran en la misma variable.
- Si t_1 y t_2 son terminos diferentes entonces la unificación falla.
- Si t_1 es una celda, procedimiento o puerto y t_2 no es una variable vacia entonces la unificación falla.

2.1.14. Ciclos

SubOz cuenta con una abstracción de ciclos mediante la expresión `for`.

⟨expresion⟩ ::= **for** ⟨variable⟩ **in** ⟨expresion⟩**..** ⟨expresion⟩
do ⟨cuerpo⟩ **end**

`for-exp(var exp-ini exp-fin cuerpo)`

3. Ejemplos

En esta sección se muestran algunos ejemplos de programas escritos en *SubOz*. Los ejemplos son mostrados en las figuras 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 y 11.

En la figura 1, se muestra un programa sencillo donde se suman varios valores enteros. Este programa tiene como valor 12.

En la figura 2, se muestra un ejemplo de creación de variables locales y de asignación. La variable `X` es unificada al valor 5 y posteriormente su valor es retornado. El valor de todo el programa es 5.

En la figura 3, se muestra otro ejemplo de creación de variables locales y de asignación. Sin embargo, en este caso las variables son unificadas con valores flotantes, además se muestra que no importa el orden de los elementos en una expresión `set`. Este programa tiene como valor 16.0.

En la figura 4, se muestra un programa donde se crea una variable local `X` y luego una variable local `Y` en un contexto interno. Luego la variable `Y` es unificada con el valor 12 y posteriormente con la vari-

```
+{3 4 5}
```

Figura 1: Un programa sencillo.

```
local
  X
in
  set X = 5
  X
end
```

Figura 2: Un programa con creación de variables locales y asignación.

```
local
  X Y
in
  set X = ~5.0
  set 3.2 = Y
  *{X Y}
end
```

Figura 3: Un programa con creación de variables locales y asignación.

```
local
  X
in
  local
    Y
  in
    set Y=12
    set Y=X
  end

  X
end
```

Figura 4: Un programa con unificación de variables.

```
local
  Rec X
in
  set Rec = miregistro(campo1:X campo2:4)
  set X = 10
  *{.Rec.campo1 .Rec.campo2}
end
```

Figura 5: Un programa con registros.

able `X` (recuerde, que esto es posible solo porque `X` es una variable no ligada, en caso contraria mostraría error). Finalmente, el valor de `X` es retornado. El programa retorna 12.

En la figura 5, se muestra un programa donde se manipulan registros. Inicialmente, la variable `Rec` es creada y unificada con el registro `miregistro(campo1:X campo2:4)` donde el campo `campo1` es unificado con la variable `X`. Luego, la variable `X` es unificada con el valor 10 y posteriormente se retorna el producto del valor de los campos del registro `Rec`. El valor del programa es 40 puesto que en virtud de la unificación el `campo1` del registro tiene como valor 10.

En la figura 6 se muestra un programa donde se unifican registros. Los registros se pueden unificar

```
local
  Rec1 Rec2 X Y
in
  set Rec1 = miregistro(campo1:X campo2:4)
  set Rec2 = miregistro(campo1:5 campo2:Y)
  ={Rec1 Rec2}
  .Rec1.campo1
end
```

Figura 6: Un programa con unificación de registros.

```
local
  Proc1 X Y Z
in
  proc{Proc1 X Y Z}
     set Z = +{X Y}
  end

  set X = 5
  set Y = 10
  {Proc1 X Y Z}
  Z
end
```

Figura 7: Un programa con procedimientos.

```
local
  Lista X Y
in
  set Lista = [1 X Y 4]
  set X = 2
  set Y = X
  -{Y .Lista.1}
end
```

Figura 8: Un programa con listas.

```
local
  R X Y
in
  set R = rec(a:12 b:X)
  set X = 10

  if =={.R.a 12}
  then set Y = *{X 2}
  else set Y = /{X 2}
  end

  Y
end
```

Figura 9: Un programa con condicionales.

dado que tienen la misma etiqueta (`miregistro`) y tienen los mismos campos (`campo1` y `campo2`). Además los valores en los campos se pueden unificar (`X` con el valor 5 y `Y` con el valor 4). El programa retorna 5.

En la figura 7 se muestra un programa donde se crea un procedimiento. Dicho procedimiento recibe 3 parámetros `X`. `Y` y `Z` y unifica `Z` con el valor asociado a la suma de las variables `X` y `Y`. El programa retorna 15.

En la figura 8 se muestra un programa donde se crea una lista con el valor 1, las variables `X` y `Y` y el valor 4. Finalmente, el programa retorna el valor de la resta entre `Y` y el primer elemento de la lista. El valor es 1.

En la figura 9 se muestra un programa donde se muestra el uso de condicionales en *SubOz*. Inicialmente se crea un registro con campos `a` y `b`. Luego se verifica si el valor del registro en el campo `a` es 12. Si esto es cierto, entonces se unifica `Y` con la multiplicación de `X` por 2 y si no, se unifica `Y` con la división de `X` por 2. El programa retorna 20.

En la figura 10 se muestra un programa donde se define la función factorial. El programa retorna 120.

El programa en la figura 11 retorna 3, pues el lenguaje verifica si se puede unificar `r(a:X)` con `r(a:M)` y como es así, unifica `M` con 3 en un ambiente donde `M` es `X` pues esa es na de las consecuencias de unificar `r(a:X)` con `r(a:M)`.

El programa mostrado en la figura 12 retorna 27

```
local
  Fact
in
  set Fact =  fun{$ X}
                if =={X 1} then 1
                else *{X {Fact -{X 1}}}
                end
              end

  {Fact 5}
end
```

Figura 10: Un programa con funciones.

```
local
  X
in
  case r(a:X)
  of r(a:M) then
    set 3 = M
  end

  X
end
```

Figura 11: Un programa con case.

```
local
  C X Y
in
  set C = newcell{12}
  set X = @{C}
  setcell{C 15}
  set Y = @{C}

  +{X Y}
end
```

Figura 12: Un programa con celdas.

```
local
  P X Y Fl
in
  set P = newport{Fl}
  send{P 12}
  send{P 3}

  set X = .Fl.1
  set Y = .(.Fl.2).1

  *{X Y}
end
```

Figura 13: Un programa con puertos.

puesto que inicialmente se crea una celda con valor 12, este valor es extraido y unificado con `X`. Luego el valor de la celda es cambiado a 15 y ese valor es extraido y unificado con `Y`. Finalmente, se retorna la suma de `X` y `Y` que da 27.

En la figura 14 se muestra un programa donde se crea un puerto `P` al cual se le envia los valores 12 y 3. Luego dado que estos valores son puestos en el flujo (variable) `Fl` son extraidos y multiplicados. El valor del programa es 36.

En la figura **??** se muestra un programa donde se utiliza una celda y un ciclo `for` para calcular la sumatoria entre 1 y 10. El programa retorna 55.

```
local
  Ini Fin Cell
in
  set Ini = 1
  set Fin = 10
  set Cell = newcell{0}

  for I in Ini .. Fin do
    setcell{Cell +{@{Cell} I}}
  end

  @{Cell}
end
```

Figura 14: Un programa con puertos.

4. Trabajo a realizar

Cada grupo debe entregar lo siguiente:

a) La definición de la gramática y la especificación léxica, sintáctica y semántica del lenguaje *SubOz*.

b) La implementación del lenguaje *SubOz*.

c) Opcionalmente, cada grupo puede implementar características adicionales útiles.

Aclaraciones

- ☞ El proyecto se debe desarrollar en grupos de máximo tres (3) alumnos.
- ☞ En todo momento se debe tener en cuenta las restricciones y condiciones impuestas.
- ☞ Existe libertad para hacer cambios en la gramática pero dichos cambios deben ser claramente sustentados.
- ☞ Recuerdo que su equipo de trabajo es una unidad, lo cual significa que debe haber comunicación entre los miembros, donde cada cual puede trabajar en aspectos diferentes del proyecto, pero debe conocer (al menos someramente) el trabajo de los demás.
- ☞ No se debe compartir información específica de las implementaciones, ni tampoco debe haber reuniones entre varios grupos.
- ☞ Cualquier indicio de fraude o copia, asignará la nota de CERO (0.0) a todos los miembros del equipo. Tener en cuenta el punto anterior, es decir, si varios equipos hacen reuniones, puede dar un indicio de copia.
- ☞ El equipo de trabajo debe realizar el trabajo por si mismo. No se recomienda DE NINGUNA FORMA que se busque ayuda externa para la realización de la implementación del proyecto. Solamente busque ayuda del profesor y del monitor.
- ☞ Utilice únicamente las herramientas que usted maneje y entienda completamente, recuerde que se debe sustentar el trabajo.
- ☞ Los criterios de calificación de la parte práctica del proyecto serán:
  - **[30 %]** Algoritmo de unificación.
  - **[15 %]** Celdas y puertos.
  - **[15 %]** Expresión case.
  - **[30 %]** Otras primitivas, aplicación de funciones y procedimientos, aplicación de primitivas, variables locales, ejecución condicional, ciclos, chequeo dinámico de tipos.
  - **[10 %]** Registros (y listas).
- ☞ Para la calificación se tendrá en cuenta dos aspectos, la especificación e implementación del

lenguaje (40 %) y la sustentación (60 %)·. Para la sustentación cada estudiante debe tener la capacidad de explicar, entender, modificar y añadir pequeñas características al lenguaje.

- ☞ Se debe entregar el código fuente de la implementación del lenguaje. Así mismo se debe presentar un informe (pdf) donde se especifique las decisiones tomadas para definir el lenguaje. Se debe presentar la descripción de la gramática, la definición de las estructuras y tipos de datos definidos y utilizados y las técnicas empleadas.

- ☞ Se debe comentar en el código fuente cada función definida especificando el tipo de dato de entrada y salida, así como una breve descripción de la utilidad de dicha función.

- ☞ El proyecto (implementación e informe) se debe enviar al correo `carlos.a.ramirez@correounivalle.edu.co` a más tardar el día **11 de Diciembre a las 8:00 am**. Se debe enviar un mail con asunto **Proyecto FLP** y con un archivo adjunto que siga la convención *Apellido1Apellido2Apellido3 ProyectoFLP10.zip*.

- ☞ La sustentación del proyecto se llevara a cabo el día **11 de Diciembre** entre las 9am y 12am y entre las 2pm y 4pm. Durante esa semana se colocará un papel en la puerta del laboratorio Avispa para que cada grupo indique el horario en el que desea sustentar.

- ☞ Las fechas de entrega y sustentación del proyecto solo se modificarán teniendo en cuenta la disponibilidad de todos los estudiantes y de los espacios en la universidad.

- ☞ Recuerde, en caso de dudas y aclaraciones puede preguntar al inicio de las clases o enviar un correo con su consulta al profesor.