University of Puerto Rico at Mayaguez
Computer and Electrical Engineering Department

Phase 3: Final Report and Demo

Yosiris Andujar
Jesmarie Hernandez
Alcibiades Bustillo

Introduction:

PLA, A Programming Language for Linear Algebra, es un lenguaje de programación que se centra en resolver los problemas primordiales o básicos del tema de álgebra lineal. Esta herramienta desarrollada durante el semestre actúa primordialmente como una calculadora que soluciona diferentes operaciones relacionadas a vectores. Es importante mencionar que PLA resuelve operaciones para vectores de tres dimensiones y dos dimensiones. Algunas de las funciones implementadas (que se verán más a fondo según continue el documento) son suma y resta de vectores, producto cruz, norma, angulo entre dos vectores, etc.

El propósito principal de este lenguaje es que sirva de guía para aquellas personas que estén comenzando a aprender algebra lineal. Que este lenguaje sea como una calculadora donde puedan rápidamente verificar alguna respuesta o calcular un dato sencillo. Por ende, en este documento se presenta un tutorial del lenguaje donde el usuario puede observar cómo puede comenzar a utilizarlo. Adicionalmente, se encuentra un manual de referencia, y otra parte donde se explica cómo se desarrolló el lenguaje. Finalmente, en la conclusión recapitulamos sobre la propuesta sometida en la primera fase. Se discutirá que se implementó y que no, si se tuvieron que hacer algunos cambios a lo largo del semestre, entre otros.

Language tutorial:

Modo interactivo

Se dice que estamos usando el intérprete en modo interactivo, cuando los comandos son leídos desde una terminal. Se espera el siguiente comando después de los signos: *paréntesis que abre-adición-guión bajo-adición-paréntesis que cierra- mayor que* **(+_+) >**

Números

El intérprete actúa como una simple calculadora; podés ingresar una expresión y éste escribirá los valores. La sintaxis es sencilla: existen los operadores **+, -, * ,/** .
Además, los paréntesis pueden ser usados para agrupar. Por ejemplo:

(+_+) > 1+1
2.0
(+_+) > 32-11
31.0
(+_+) > (1-2)-4
-5.0

El signo igual (**=**) es usado para asignar un valor a una variable. Luego, ningún resultado es mostrado antes del próximo prompt:

```
(+_+) > x=2
(+_+) >x
2.0
```

El programa soporta completamente los números de punto flotante; las operaciones con mezclas en los tipos de los operandos convierten los enteros a punto flotante:

```
(+_+) > 3.0 -4
-1.0
```

Vectores 2d y 3d

Al igual que con los números, el intérprete también actúa como calculadora para vectores de 2 y 3 dimensiones. podés ingresar una expresión y éste escribirá los valores. La sintaxis es sencilla: existen los operadores **+, -, * ,/** . Además, los paréntesis pueden ser usados para agrupar. Por ejemplo:

```
(+_+) > (1,1)
(1.0,1.0)
(+_+) > (-1,-2.7,2.3)
(-1.0,-2.7,-2.3)
(+_+) > (1,1,1)+(1,3,4)
(2.0,4.0,5.0)
(+_+) > ((1,1,1)+(1,3,4))-(1,1,1)
(1.0,3.0,4.0)
```

El signo igual (**=**) también es usado para asignar un valor a una variable. Luego, ningún resultado es mostrado antes del próximo prompt:

```
(+_+) > x=(1,4,5)
(+_+) > x
(1.0,4.0,5.0)
```

Norma de un vector

Para hallar la norma de un vector se usa la palabra reservada **norma.**

```
(+_+) > norma((1,1))
1.4142
(+_+) > norma((1,2,3))
3.7416
```

Perpendicularidad de un vector

Para saber si dos vectores son vectores usamos el comando **isper.** Si son los dos vectores perpendiculares retorna True y si no False

(+_+) > isper((1,0),(0,1))
True
(+_+) > isper((1,0,9),(0,1,7))
False

Angulo entre vectores

Para hallar el ángulo entre dos vectores se usa el comando **angulo.** Devuelve el resultado en grados.

(+_+) > angulo((1,3),(2,5))
3.36
(+_+) > angulo((-1,3,4),(2,5,7))
24.43

Determinante de vectores 2d

Para hallar el determinante de dos vectores se usa el comando **det**.

(+_+) > det((1,4),(-4,6))
22.0

Para tres dimensiones no se ha implementado.

Producto Cruz

Para hallar el producto cruz de dos vectores de tres dimensiones se utiliza en el comando **cruz.**

(+_+) > cruz((2,5,7),(-9,4,9))
(17.0,-81.0,53.0)

Coordenada de un vector

Se utiliza la siguiente sintaxis

(+_+) > x=(1,23)
(+_+) > (1,23)[1]
1.0
(+_+) > (1,23)[2]
23.0
(+_+) > x[1]

1.0
(+_+) > x[2]
23.0

Language reference manual:

Operadores: (donde a y b son números)

`a + b`	`a plus b`
`a - b`	`a minus b`
`a * b`	`a multiplied by b`
`a / b`	`a divided by b`
`(a-b)-c`	`a minus b, minus c`

Los operadores + - * / se utilizan para sumar, restar multiplicar y dividir, respectivamente. Los paréntesis ( ) se utilizan para asociar operaciones.

Vectores: (2 y 3 dimensiones, donde las coordenadas deben ser valores numéricos)

`(x,y)`	`2 dim vector of x,y coordinates`
`(x,y,z)`	`3 dim vector of x,y,z coordinates`

Este es el formato de los vectores y también se utilizará de esta manera para operaciones vectoriales.

Asignación de valores: (donde var es el nombre de una variable y a, b c deben ser valores numéricos, en el caso de vectores aplica para 2 y 3 dimensiones)

`x = a`	`assign value a to x`
`x = (a,b,c)`	`assign vector with coordinates a,b,c to x`

El signo de igualdad se utiliza para asignarle un valor a x.

Operaciones vectoriales: (donde (x,y) o (x,y,z) son vectores)

norma((x,y))	**magnitude of vector with x,y coordinates**
norma((x,y,z))	**magnitude of vector with x,y,z coordinates**
isper((x$_1$,y$_1$),(x$_2$,y$_2$))	**vector1 perpendicular to vector2 (dim 2)**
isper((x$_1$,y$_1$,z$_1$),(x$_2$,y$_2$,z$_2$))	**vector1 perpendicular to vector2 (dim 3)**
angulo((x$_1$,y$_1$),(x$_2$,y$_2$))	**angle between vector1 and vector2 (dim 2)**
angulo((x$_1$,y$_1$,z$_1$),(x$_2$,y$_2$,z$_2$))	**angle between vector1 and vector2 (dim 3)**
det((x$_1$,y$_1$),(x$_2$,y$_2$))	**determinant for vector1 and vector2 (dim 2)**
cruz((x$_1$,y$_1$,z$_1$),(x$_2$,y$_2$,z$_2$))	**cross product for vector1 and vector2 (dim 3)**

Coordenadas: (donde v es un vector y a es un valor numérico)

v[a]	**get coordinate a of v**

Para conseguir las coordenadas x,y,z de un vector v, se utilizará v[1],v[2],v[3], respectivamente.

Language development:

- *Translator architecture*
  La arquitectura utilizada, primeramente se establecieron los tokens que se estarían utilizando con su definición de sintaxis. Tales como el vector a, vector b, signo de suma, signo de resta, signo de igualdad, etc. Luego se definieron las funciones que se podrían hacer en el lenguaje. Una vez definidos los tokens, se continúa a construir el lexer. Como habíamos mencionado en nuestra propuesta, utilizamos PLY para el lexer y parser. Al importar ply.lex,

ya el lexer es construido. Luego se prosigue con la creación de la clase auxiliar Vector. Aquí se realizan las operaciones para cada vector, dependiendo si es de dos dimensiones o de tres dimensiones. Para diferenciar si los vectores son de tres dimensiones o dos dimensiones se utiliza self.dim. Esta si es igual a 7, los vectores son de tres dimensiones, y si es 3 los vectores son de dos dimensiones. Finalmente, se crea el parser donde se establece como los “statements” y expresiones serán identificados. Adicionalmente, se importa el ply.yacc para que el parser funcione.

- *Describe the interfaces between the modules.*

  Module: lex.py

  Este módulo construye escáneres basados en reglas de expresiones regulares. Para utilizar el módulo, sólo tiene que escribir una colección de reglas de expresiones regulares y acciones como esta:

```
# lexer.py
import lex

# Define a list of valid tokens
tokens = (
   'NUMBER', 'PLUS', 'MINUS'
   )

# Define tokens as functions
def t_NUMBER(t):
   r' \d+ '
   return t

# Some simple tokens with no actions
t_PLUS = r'\+'
```

```
t_MINUS = r'-'

# Initialize the lexer
lex.lex()
```

Module: Yacc

Este módulo nos permitirá crear un programa que tome un flujo de tokens como entrada y reconozca un lenguaje a partir de ellos. Utiliza una gramática BNF que describe cómo se ensamblan estos tokens.

Se debe definir una función por cada una de las reglas. Cada función recibe como parámetro un objeto iterable (muy parecido a una lista, pero que no se comporta totalmente como tal) que contiene los valores de cada símbolo de la regla.

```
def p_statement_assign(p):
    'statement : NAME EQUALS expression'
    names[p[1]] = p[3]

def p_statement_expr(p):
    'statement : expression'
    print(p[1])

import ply.yacc as yacc

yacc.yacc()
```

- *Describe the software development environment used to create the Translator*

El ambiente que se utilizó para crear el traductor fue PyCharm. Este es simple de utilizar, user-friendly y provee todas las herramientas necesarias para desarrollar programas en Python.

- *Describe the test methodology used during development.*
  Show programs used to test your translator.

  La metodología usada para hacer pruebas durante el desarrollo de **Vector** fueron utilizando inicialmente el modo interactivo, es decir, desde línea de comando. A medida que se fueron implementando las diferentes funciones se fueron probando en la línea de comando. Por ejemplo cuando queríamos probar la función norma entre dos vectores colocabamos lo siguiente:

```
(+_+) > norma((1,1))
1.4142
(+_+) > norma((1,2,3))
3.7416
```

  Cuando el número de funciones soportadas por **vector** fue aumentando se hizo tedioso ir probando una a una cada función, por lo tanto se utilizó un archivo con todas las funciones y los resultados eran mostrados por consola.
  Ejemplo del fichero (prueba.txt) :

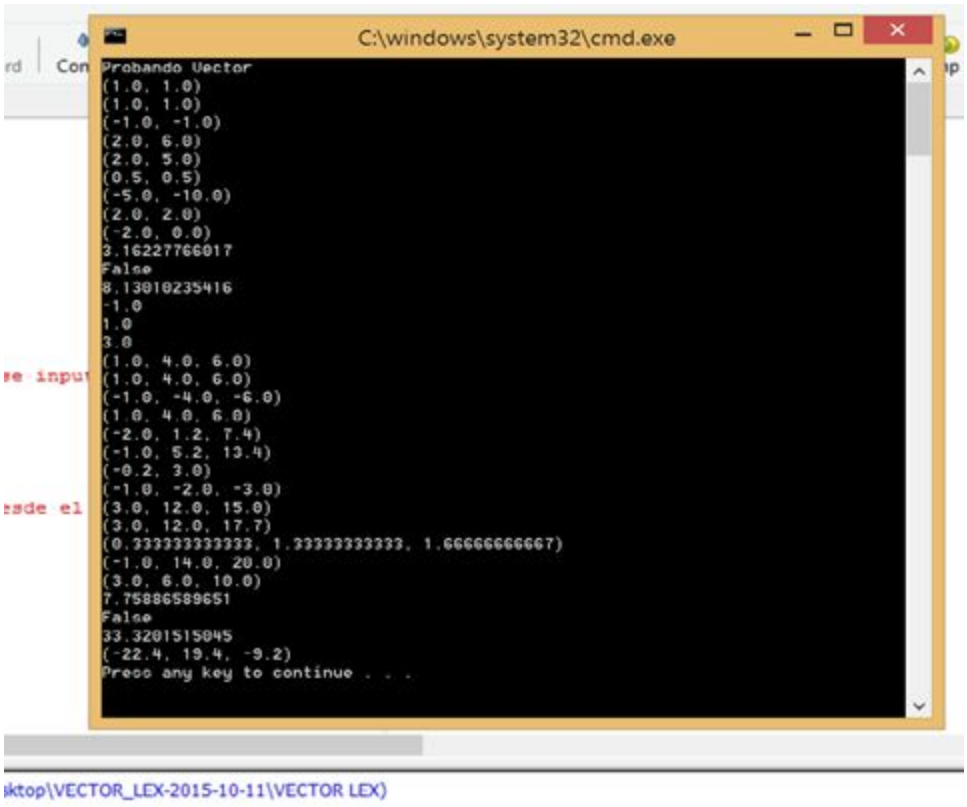
```
########################################################################
(1,1)
+(1,1)
-(1,1)
x=(1,3)
x*2
y=(1,2)
x+y
(1,1)/2
(1,2)*(-5)
(1,1)+(1,1)
(-1,1)-(1,1)
norma(x)
isper(x,y)
```

```
angulo(x,y)
det(x,y)
x[1]
x[2]
(1,4,6)
+(1,4,6)
-(1,4,6)
w=(1,4,6)
w
z=(-2,1.2,7.4)
z
w+z
(-1.2,2)+(1,1)
(-1,-2,-3)
(1,4,5)*3
(1,4,5.9)*3
(1,4,5)/3
(1,10,15)+(-2,4,5)
(1,10,15)-(-2,4,5)
norma(z)
isper(w,z)
angulo(z,w)
cruz(z,w)
#########################################################################

#########################################################################
Ejemplo del script que lee el fichero:
infile = open('prueba.txt', 'r')
# Mostramos por pantalla lo que leemos desde el fichero
print 'Probando Vector'
for line in infile:
        yacc.parse(line)

# Cerramos el fichero.
infile.close()
#########################################################################
```

Salida de consola



Conclusions:

Durante el transcurso de este proyecto, pudimos adquirir varios conocimientos sobre como es el proceso de crear un lenguaje de programación. Al poder ir desde “scratch”, se visualiza mejor y por ende se entiende con mayor claridad los diferentes pasos a seguir para poder llegar al resultado final el cual es el funcionamiento de un lenguaje de programación.

En nuestra propuesta establecimos que el lenguaje de programación sería centrado al álgebra lineal. Propusimos varios “features” que el lenguaje podría realizar. Muchos de estos pudieron ser implementados. Tales como, operaciones de suma y resta para vectores de dos y tres dimensiones, asignaciones de variables, producto cruz, etc. Estas pudieron ser realizadas y funcionan en nuestro lenguaje. No obstante, ciertas características que inicialmente queríamos fueran parte del lenguaje no se pudieron completar por diferentes razones. Tales como matrices y las diversas funciones que se pueden realizar con ellas, estructuras de control y operadores lógicos. Para matrices, resultó que la sintaxis para estas es un poco complicado al definir, por ende no se pudo incluir. Sin embargo, consideramos que para que PLA sea un lenguaje más completo debe incluir en un futuro el tema de matrices y sus operaciones,

que son esenciales en el álgebra lineal. Para las estructuras de control y los operadores lógicos decidimos eliminarlos ya que PLA actúa más como una calculadora y no encontrábamos el uso que tendrían estos. También fue mencionado en la propuesta la integración de nuestro lenguaje con otro ambiente el cual fue Python. Adicionalmente, fuimos consistentes desde el comienzo con el uso de la herramienta de lexer y parsing, PLY la cual fue de mucha ayuda y simple de usar.

En conclusión, este proyecto fue muy innovador ya que permite al programador ir más a fondo a lo que de verdad conlleva un lenguaje. Estos conocimientos adquiridos abren el camino a entender la diversidad de lenguajes que existen. Además, si en un momento en el futuro algún integrante del grupo tenga la necesidad de crear un lenguaje de programación, este ya tendrá una base sobre cómo desarrollarlo.