INTEGRANTES:

- Ariel Humberto Valle Escoto
- Eduardo Quetzal Delgado Pimentel
- Ricardo David López Arellano

Materia: Traductores de lenguajes II

2023b

INTRODUCCIÓN:

Un analizador sintáctico descendente verifica si una cadena de tokens pertenece al lenguaje definido por una gramática, siguiendo una derivación por la izquierda del símbolo inicial.

Hay diferentes tipos de analizadores sintácticos descendentes, como el por descenso recursivo, el predictivo y el con retroceso, que usan distintos métodos para elegir la regla de producción adecuada en cada paso.

El código de tres direcciones es un tipo de código intermedio que usan los compiladores para optimizar el rendimiento y el uso de recursos de los programas. Cada instrucción de este código tiene como máximo tres operandos: un destino y dos fuentes. Por ejemplo, t1 := t2 + t3. El código de tres direcciones facilita la evaluación de expresiones aritméticas, el control del flujo, la asignación de registros y la detección de subexpresiones comunes. Hay distintas formas de representar el código de tres direcciones, como los cuartetos, los tercetos y los tercetos indirectos, que tienen sus ventajas e inconvenientes.

DESARROLLO:

Para comenzar, la gramática mostrada a la derecha se implementó en el código fuente por medio del analizador léxico. Se modificó la práctica 5 usando esta nueva gramática previamente corregida en la práctica 3.

El objetivo de esta primera parte del proyecto es verificar que la gramática introducida sea válida, y, de no serlo, muestre el error.

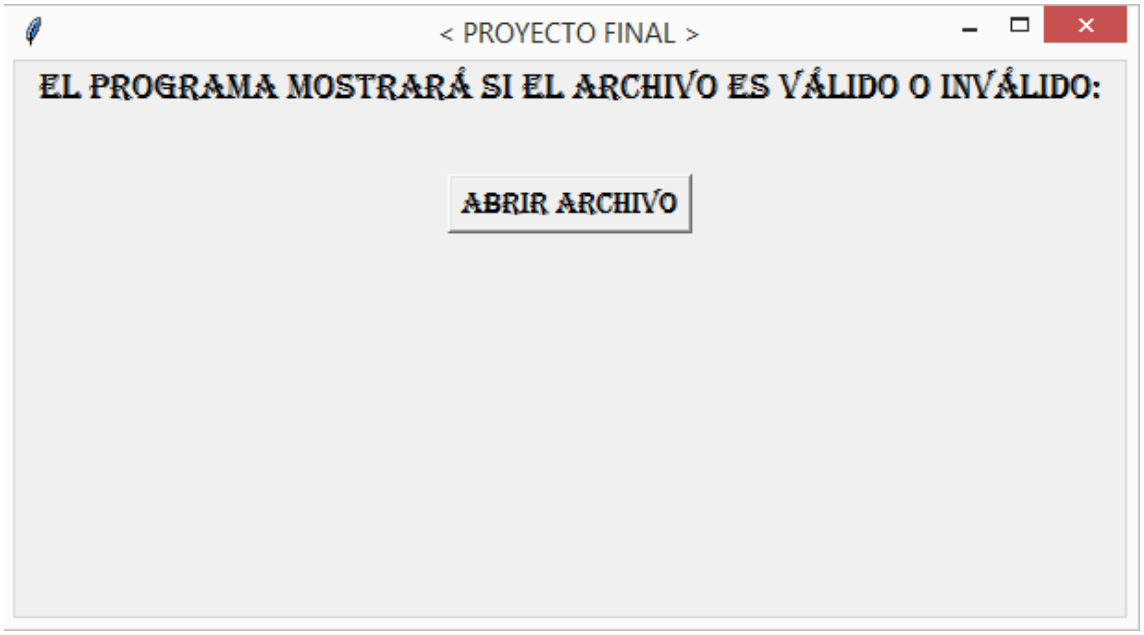
La segunda parte del proyecto requiere que se imprima el código de 3 direcciones generado por el analizador sintáctico descendente predictivo, además de la función de la primera parte.

Terminales	Símbolo
begin	a
end	b
;	c
entero	d
real	e
identificador	f
,	g
if	h
(	i
)	j
else	k
=	l
<=	m
>=	n
<>	o
<	p
>	q
num_entero	r
num_real	s
while	t
endwhile	u
:=	v
+	w
*	x
-	y
/	z
ε	ε

1. Primero, para probar la primera parte del programa, se ejecutará el siguiente archivo de entrada con un error en la 4ta línea (falta un punto y coma al final):

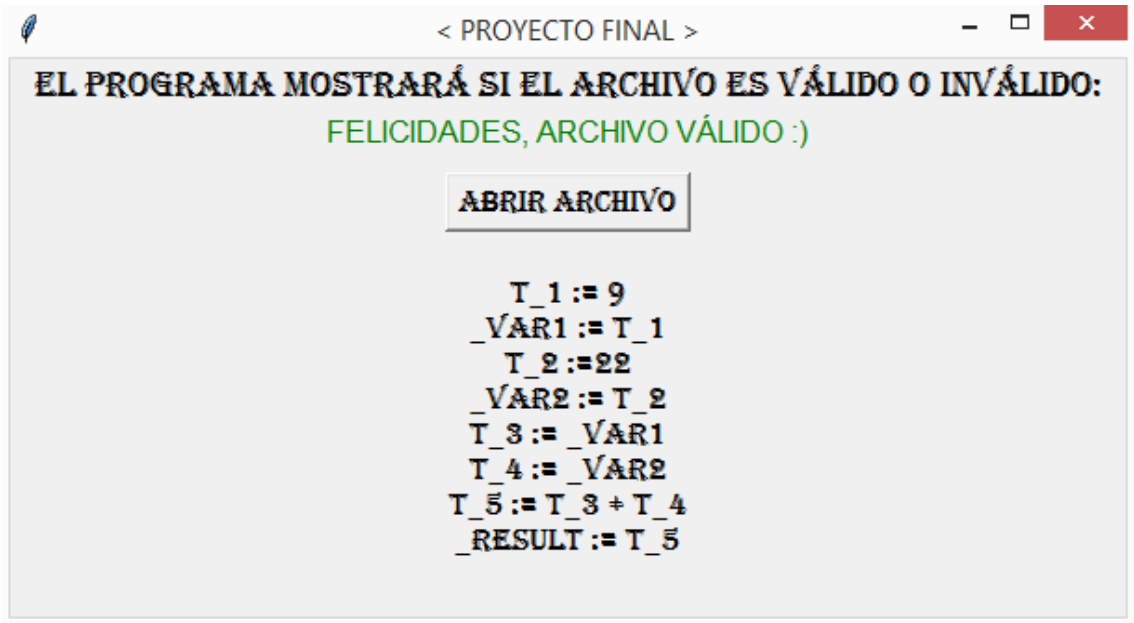
```
begin
        real var1, var2;
        real result;
        var1:= 9
        var2 := 22;
        result:= var1 + var2;
end
```

En el programa se mostrará así:



Y como el archivo le falta el “;” mostrará error:

Ya modificando el “;” en el .txt mostrará lo siguiente el programa:



2. Ahora este es el siguiente ejemplo:

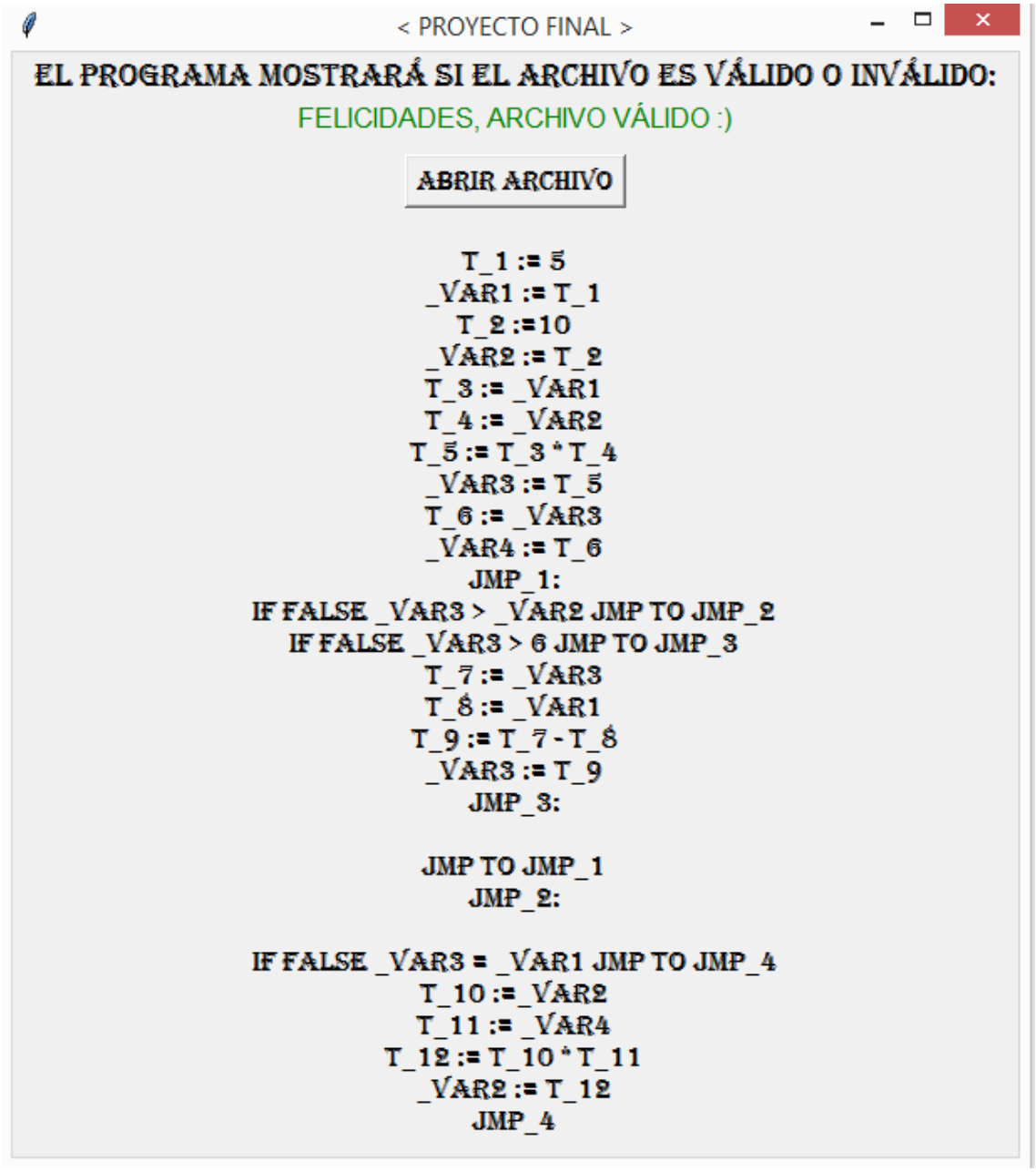
```
begin
        entero var1, var2;
        real result;
        var1:= 9;
        var2 := 22;
        result := var1 / var2;
        if (result > 25)
                var2 := 4;
        else
                var2 := 10;
        end;
        while(var1 < 15)
                var1 := var1 + 1;
        endwhile;
end
```

El programa mostrará así el resultado:

3. Ahora será con el siguiente algoritmo que es más complejo:

```
begin
        entero var1, var2;
        real var3;
        real var4;
        var1 := 5;
        var2 := 10;
        var3 := (var1 * var2);
        var4 := var3;
        while(var3 > var2)
                if(var3 > 6)
                        var3 := var3 - var1;
                end;
        endwhile;
        if(var3 = var1)
                var2 := var2 * var4;
        end;
end
```

El programa mostrará así el resultado:



CÓDIGO:

Main.py

```
import ply.yacc as yacc
from lexer import tokens
import tkinter as tk
from tkinter import filedialog, font
import re
import tkinter as tk
from tkinter import font
from PIL import Image, ImageTk
import networkx as nx
```

```
import matplotlib.pyplot as plt

# Reglas de la gramática
def p_expression(p):
    '''expression : expression PLUS term
                  | expression MINUS term
                  | term'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        if p[2] == '+':
            p[0] = p[1] + p[3]
        elif p[2] == '-':
            p[0] = p[1] - p[3]

def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        if p[2] == '*':
            p[0] = p[1] * p[3]
        elif p[2] == '/':
            p[0] = p[1] / p[3]

def p_factor(p):
    '''factor : NUMBER
              | LPAREN expression RPAREN'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = p[2]

# Regla para manejar errores de sintaxis
def p_error(p):
    print("ARCHIVO INVÁLIDO :(")
    raise SyntaxError("Error de sintaxis en la entrada")

# Construir el analizador
parser = yacc.yacc()
# Crear la interfaz gráfica
ventana = tk.Tk()
ventana.title("< PROYECTO FINAL >")

# Agregar un subtítulo
subtitulo_label = tk.Label(ventana, text="El programa mostrará si el
archivo es válido o inválido:", font=("Algerian", 14))
```

```
subtitulo_label.pack()

# Variable para almacenar el resultado y enlazarla con la etiqueta
result_var = tk.StringVar()

# Cambia el tamaño de la ventana
ventana.geometry("600x300")  # Cambia el ancho y alto

# Crear un Label para mostrar el resultado
resultado_label = tk.Label(ventana, textvariable=result_var,
font=("Helvetica", 12))
resultado_label.pack()

# Función para analizar el archivo fuente
def parse_file(file_path):
    try:
        with open(file_path, 'r') as file:
            source_code = file.read()
            parser.parse(source_code)
            result_var.set("FELICIDADES, ARCHIVO VÁLIDO :)")
            resultado_label.config(fg="green")  # Cambia el color del
texto a verde para indicar un resultado válido
    except SyntaxError as e:
        result_var.set("LO SENTIMOS, ARCHIVO INVÁLIDO :(")
        resultado_label.config(fg="red")  # Cambia el color del texto a
rojo para indicar un resultado inválido

# Función para abrir un cuadro de diálogo y obtener la ruta del archivo
def open_file_dialog():
    file_path = filedialog.askopenfilename()
    if file_path:
        parse_file(file_path)

# Botón para abrir el cuadro de diálogo
open_button = tk.Button(ventana, text="Abrir Archivo",
command=open_file_dialog, font=("Algerian", 12))
open_button.pack(pady=10)

# Iniciar el bucle de eventos de la interfaz gráfica
ventana.mainloop()
```

lexer.py

```
import ply.lex as lex

# Lista de tokens
tokens = (
    'NUMBER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
)

# Expresiones regulares para tokens simples
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'

# Token para números
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Ignorar caracteres espacio en blanco
t_ignore = ' \t'

# Manejo de errores
def t_error(t):
    #print("Carácter ilegal:", t.value[0])
    t.lexer.skip(1)

# Construir el analizador léxico
lexer = lex.lex()

# Ejemplo de uso
if __name__ == '__main__':
    lexer.input("3 + 4 * 5")
    while True:
        tok = lexer.token()
        if not tok:
            break
        print(tok)
```

parsetab.py

```
# parsetab.py
# This file is automatically generated. Do not edit.
# pylint: disable=W,C,R
_tabversion = '3.10'

_lr_method = 'LALR'

_lr_signature = 'DIVIDE LPAREN MINUS NUMBER PLUS RPAREN TIMESexpression :
expression PLUS term\n                  | expression MINUS
term\n                  | termterm : term TIMES factor\n            |
term DIVIDE factor\n            | factorfactor : NUMBER\n              |
LPAREN expression RPAREN'

_lr_action_items =
{'NUMBER':([0,5,6,7,8,9,],[4,4,4,4,4,4,]),'LPAREN':([0,5,6,7,8,9,],[5,5,5
,5,5,5,]),'$end':([1,2,3,4,11,12,13,14,15,],[0,-3,-6,-7,-1,-2,-4,-5,-
8,]),'PLUS':([1,2,3,4,10,11,12,13,14,15,],[6,-3,-6,-7,6,-1,-2,-4,-5,-
8,]),'MINUS':([1,2,3,4,10,11,12,13,14,15,],[7,-3,-6,-7,7,-1,-2,-4,-5,-
8,]),'RPAREN':([2,3,4,10,11,12,13,14,15,],[-3,-6,-7,15,-1,-2,-4,-5,-
8,]),'TIMES':([2,3,4,11,12,13,14,15,],[8,-6,-7,8,8,-4,-5,-
8,]),'DIVIDE':([2,3,4,11,12,13,14,15,],[9,-6,-7,9,9,-4,-5,-8,]),}

_lr_action = {}
for _k, _v in _lr_action_items.items():
   for _x,_y in zip(_v[0],_v[1]):
      if not _x in _lr_action:  _lr_action[_x] = {}
      _lr_action[_x][_k] = _y
del _lr_action_items

_lr_goto_items =
{'expression':([0,5,],[1,10,]),'term':([0,5,6,7,],[2,2,11,12,]),'factor':
([0,5,6,7,8,9,],[3,3,3,3,13,14,]),}

_lr_goto = {}
for _k, _v in _lr_goto_items.items():
   for _x, _y in zip(_v[0], _v[1]):
       if not _x in _lr_goto: _lr_goto[_x] = {}
       _lr_goto[_x][_k] = _y
del _lr_goto_items
_lr_productions = [
  ("S' -> expression","S'",1,None,None,None),
  ('expression -> expression PLUS
term','expression',3,'p_expression','proyecto.py',6),
  ('expression -> expression MINUS
term','expression',3,'p_expression','proyecto.py',7),
  ('expression -> term','expression',1,'p_expression','proyecto.py',8),
```

```
  ('term -> term TIMES factor','term',3,'p_term','proyecto.py',18),
  ('term -> term DIVIDE factor','term',3,'p_term','proyecto.py',19),
  ('term -> factor','term',1,'p_term','proyecto.py',20),
  ('factor -> NUMBER','factor',1,'p_factor','proyecto.py',30),
  ('factor -> LPAREN expression
RPAREN','factor',3,'p_factor','proyecto.py',31),
]
```

CONCLUSIONES:

En conclusión, lo que aprendimos al terminar de realizar el proyecto de traductores II, es la importancia de todo lo que venimos viendo en el semestre, ya que desde el código de tres direcciones, hasta los analizadores sintácticos, es importante aprender de ellos, ya que estos nos han mejorado para ser mejores programadores y a su vez también nos ayudó a mejorar el cómo trabajamos en equipo y lo bueno de este proyecto fue que integramos mucho de lo visto durante el semestre que se siente como armar un rompecabezas, que acabamos de construir ya que por fin vemos todas las piezas juntas y en funcionamiento.

Para terminar esperemos que como equipo este proyecto nos haya enseñado las bases de los compiladores para así luego si se necesitara pues aprender más de ellos, pero con unas buenas bases para hacer de ese aprendizaje algo más rápido.