Logotipo

Descripción generada automáticamente

Integrantes:

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

**Materia**: Traductores de lenguajes II

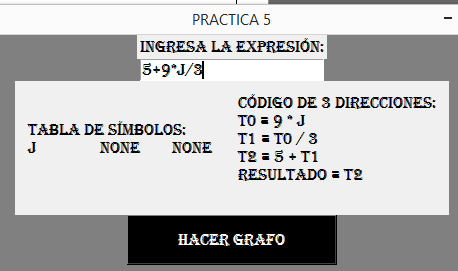
2023b

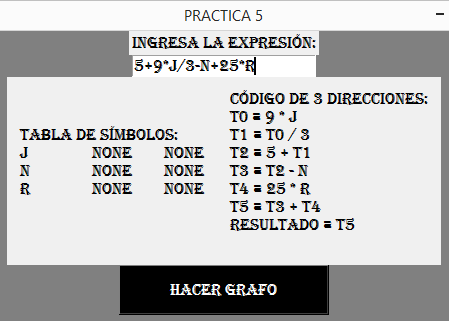
Introducción

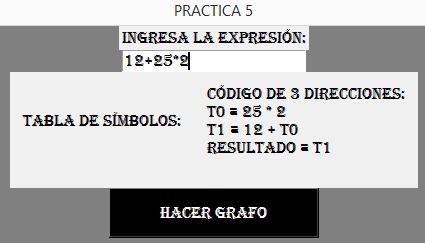
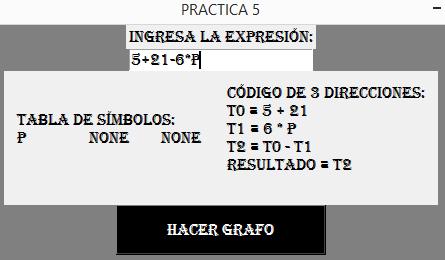
Un analizador sintáctico descendente predictivo es un tipo de analizador sintáctico utilizado en el campo de la informática y la teoría de la compilación. Su objetivo principal es analizar la estructura gramatical de un programa fuente para determinar si cumple con la gramática del lenguaje de programación en cuestión. En otras palabras, verifica si el código fuente está escrito correctamente en términos de la sintaxis del lenguaje de programación.

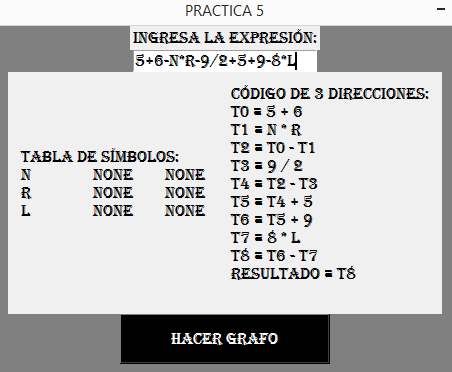
En la programación, el "código de tres direcciones" (también conocido como "código de tres direcciones intermedio") es una representación intermedia del código fuente de un programa. Esta representación se caracteriza por tener instrucciones con hasta tres operandos y un operador. Cada instrucción de tres direcciones realiza una operación simple con los operandos y almacena el resultado en una variable temporal.

Capturas









Código

import re

import tkinter as tk

from tkinter import font

from PIL import Image, ImageTk

import networkx as nx

import matplotlib.pyplot as plt

class Nodo:

    def \_\_init\_\_(self, operador, izquierda, derecha):

        self.operador = operador

        self.izquierda = izquierda

        self.derecha = derecha

class Hoja:

    def \_\_init\_\_(self, tipo, valor):

        self.tipo = tipo

        self.valor = valor

class TablaSimbolos:

    def \_\_init\_\_(self):

        self.tabla = {}

    def agregar(self, identificador, valor):

        self.tabla[identificador] = valor

    def obtener(self, identificador):

        return self.tabla.get(identificador, None)

def analizador\_sintactico(expresion):

    tokens = re.findall(r'\d+|[-+\*/()]|\w+', expresion)

    idx = 0

    def match(expected\_token):

        nonlocal idx

        if idx < len(tokens) and tokens[idx] == expected\_token:

            idx += 1

        else:

            raise Exception(f"Error de sintaxis: Se esperaba '{expected\_token}'")

    def factor():

        if tokens[idx] == '(':

            match('(')

            sub\_arbol = expresion()

            match(')')

            return sub\_arbol

        elif tokens[idx].isdigit():

            num = Hoja('num', tokens[idx])

            match(tokens[idx])

            return num

        elif tokens[idx].isalpha():

            id = Hoja('id', tokens[idx])

            match(tokens[idx])

            return id

        else:

            raise Exception("Error de sintaxis: Factor no válido")

    def termino():

        left = factor()

        while idx < len(tokens) and tokens[idx] in ['\*', '/']:

            operador = tokens[idx]

            match(operador)

            right = factor()

            left = Nodo(operador, left, right)

        return left

    def expresion():

        left = termino()

        while idx < len(tokens) and tokens[idx] in ['+', '-']:

            operador = tokens[idx]

            match(operador)

            right = termino()

            left = Nodo(operador, left, right)

        return left

    tabla\_simbolos = TablaSimbolos()

    raiz = expresion()

    def construir\_tabla\_simbolos(nodo):

        if isinstance(nodo, Nodo):

            construir\_tabla\_simbolos(nodo.izquierda)

            construir\_tabla\_simbolos(nodo.derecha)

        elif isinstance(nodo, Hoja) and nodo.tipo == 'id':

            tabla\_simbolos.agregar(nodo.valor, None)

    construir\_tabla\_simbolos(raiz)

    return raiz, tabla\_simbolos

def mostrar\_tabla\_simbolos(tabla\_simbolos):

    global tSimbolos

    tSimbolos = "Tabla de símbolos:\n"

    i = 1

    for identificador in tabla\_simbolos.tabla:

        valor = tabla\_simbolos.obtener(identificador)

        if valor is None:

            valor = "None"

        tSimbolos += f"{identificador}\t{valor}\t{valor}\n"

        i += 1

def mostrar\_codigo\_tres\_direcciones(raiz):

    global tGrafo

    tGrafo = "Código de 3 direcciones:\n"

    codigo\_3d = []

    def dfs(nodo):

        global tGrafo

        if isinstance(nodo, Nodo):

            left\_id = dfs(nodo.izquierda)

            right\_id = dfs(nodo.derecha)

            temp\_var = f"t{len(codigo\_3d)}"

            tGrafo += f"{temp\_var} = {left\_id} {nodo.operador} {right\_id}\n"

            codigo\_3d.append((temp\_var, left\_id, right\_id))

            return temp\_var

        elif isinstance(nodo, Hoja):

            return nodo.valor

    result = dfs(raiz)

    tGrafo += f"Resultado = {result}\n"

def main():

    global tGrafo, tSimbolos

    expresion = entrada.get()

    raiz, tabla\_simbolos = analizador\_sintactico(expresion)

    mostrar\_codigo\_tres\_direcciones(raiz)

    mostrar\_tabla\_simbolos(tabla\_simbolos)

    tablaGrafo.config(text=tGrafo, anchor='w', justify='left')

    #tablaSimbolos.config(text=tSimbolos, anchor='w', justify='left')

ventana = tk.Tk()

ventana.title("PRACTICA 5")

# Cambia el tamaño de la ventana

ventana.geometry("600x500")  # Cambia el ancho y alto

# Cambia el color de fondo de la ventana

ventana.configure(bg="gray")  # Cambia el color de fondo

# Crear una fuente personalizada después de crear la ventana

fuente\_personalizada = font.Font(family="Algerian", size=12)  # Cambia "Arial" al tipo de fuente que desees y el tamaño

# Aplica la fuente personalizada a la etiqueta

etiqueta = tk.Label(ventana, text="Ingresa la expresión:", font=fuente\_personalizada)

etiqueta.pack()

entrada = tk.Entry(ventana, font=fuente\_personalizada)

entrada.pack()

# Crear un marco para contener las tablas

marco\_tablas = tk.Frame(ventana)

marco\_tablas.pack()

# Tabla de símbolos

tablaSimbolos = tk.Label(marco\_tablas, text="", anchor='w', justify='left', font=fuente\_personalizada)

tablaSimbolos.pack(side="left", padx=10, pady=10)

# Tabla de grafo

tablaGrafo = tk.Label(marco\_tablas, text="", anchor="w", justify='left', font=fuente\_personalizada)

tablaGrafo.pack(side="right", padx=10, pady=10)

# Cambia el tamaño y la forma de los botones a "ridge"

calcular\_button = tk.Button(ventana, text="Hacer Grafo", command=main, font=fuente\_personalizada, width=20, height=2, bg="black", fg="white")

calcular\_button.pack()

ventana.mainloop()

Conclusiones

Lo que podemos concluir como equipo al terminar la practica 4 es que comprendimos de mejor manera el tema de los analizadores sintácticos descendentes ya que con este conocimiento podremos realizar nuestro proyecto final, ya después de varias practicas podemos ver como poco a poco logramos entender o juntar las piezas de lo que realizaremos en un futuro ya que es fundamental entender bien desde el principio para así poder seguir desarrollando las piezas de este compilador ya que si lo hiciéramos sin ninguna estructura, estamos seguros que no funcionaria, pero con estos avances logramos observar una mejoría en las habilidades que tenemos para poder desarrollar estos programas.

En conclusión, un analizador sintáctico descendente predictivo es una herramienta esencial en la teoría de compiladores y la informática. Su función es verificar si un programa fuente sigue la sintaxis del lenguaje de programación, permitiendo detectar errores gramaticales en el código. Estos analizadores utilizan funciones de predicción basadas en la gramática del lenguaje para determinar las reglas de producción, lo que los hace más fáciles de entender e implementar en comparación con otros tipos de analizadores. A pesar de su simplicidad, los analizadores sintácticos descendentes predictivos tienen limitaciones y no pueden manejar todas las gramáticas, especialmente las ambiguas o recursivas izquierdas, lo que requiere técnicas más avanzadas en esos casos.