Compilador para lenguaje BPD

INFO 165- Compiladores

Instituto de Informática, Universidad Austral de Chile.

Integrantes: Sebastián Luarte
Matias Martinez
Victor Moya

Académico: Maria Eliana De la Maza

Fecha: 12 de Diciembre de 2022

Introducción

Un compilador es un programa que traduce código escrito en un lenguaje de programación de alto nivel (cercano al humano) a un lenguaje de máquina (entendido por la computadora), para realizar la compilación se utilizan 2 fases, la de análisis y la de síntesis. En este trabajo nos enfocaremos en la fase de análisis, principalmente en el analizador léxico y el analizador sintáctico.

El analizador léxico es la primera etapa de la fase de análisis, se encarga de dividir el programa en tokens según la tabla de símbolos definida por el mismo lenguaje y clasificarlos según su significado para ser procesados posteriormente. En concreto, vamos a estudiar Lex con el objetivo de especificar los analizadores léxicos. En este lenguaje, los patrones se especifican por medio de expresiones regulares, y el metacompilador de Lex genera un reconocedor de las expresiones regulares mediante un autómata finito.

Terminando el proceso anterior continúa el analizador sintáctico, en esta fase el analizador se encarga de chequear la secuencia de tokens que representa al texto de entrada, en base a una gramática dada. En caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce en base a una representación computacional. Este árbol es el punto de partida de la fase posterior de la etapa de análisis: el analizador semántico.

El principal objetivo de este trabajo radica en la implementación de un traductor dirigido por sintaxis para el lenguaje BDP (Base de datos de personas). El lenguaje manejador de base de datos debe poseer las siguientes instrucciones:

- **inicia**: Primera instrucción de cualquier programa.
- **crea(arch)**: Crear y abre un archivo llamado ‘arch’, que contendrá la base de datos.
- **abre(arch)**: Abre un archivo, ya existente, que contiene una base de datos.
- **ingresa(código, nombre, edad, ocupación, dirección)**: Agrega a continuación del último registro del archivo de la base de datos en uso, un nuevo registro con el código, nombre, edad, ocupación, y dirección de alguna persona.
- **lista**: Despliega en pantalla el contenido del archivo en uso.
- **muestra(código)**: Despliega en pantalla, el registro del archivo en uso, cuyo código se ingresa.
- **cierra**: Cierra el archivo en uso.
- **termina**: Última instrucción de cualquier programa.

Desarrollo

Para el desarrollo de este trabajo se utilizó la librería ply, la cual es una implementación de las herramientas de análisis de LEX y YACC para Python, proporcionando las funcionalidades básicas de éstas.

Analizador léxico

El primer paso fue definir los tokens a implementar en el analizador léxico, estos fueron:

- **INICIA**: para la palabra reservada “inicia”.
- **CREA**: para la palabra reservada “crea”.
- **ABRE**: para la palabra reservada “abre”.
- **INGRESA**: para la palabra reservada “ingresa”.
- **LISTA**: para la palabra reservada “lista”.
- **MUESTRA**: para la palabra reservada “muestra”.
- **CIERRA**: para la palabra reservada “cierra”.
- **TERMINA**: para la palabra reservada “termina”.
- **NOMBRE**: para los nombres que se ingresarán en el archivo.
- **DIRECCION**: para las direcciones que se ingresarán en el archivo.
- **NUMERO**: para los códigos y edades que se ingresarán en el archivo.
- **ARCH**: para el nombre del archivo que se desea crear o abrir.
- **OCUPACION**: para la ocupación que se ingresará en el archivo.
- **APARENT**: para el símbolo “(“.
- **CPARENT**: para el símbolo “)”.
- **COMA**: para el símbolo “,”.

Para cada uno de los tokens anteriormente mencionados se definió una expresión regular en Python, donde “r” indica que se trata de una expresión regular, lo encerrado en *[]* corresponde a un conjunto de caracteres, *[A-ZÑ]* corresponde a un conjunto de letras mayúsculas (incluyendo la Ñ), *[a-zñ]* corresponde a un conjunto de letras minúsculas (incluyendo la ñ) y *[0-9]* corresponde a un conjunto de números entre el 0 y 9, el símbolo * significa “0 o más veces” y el símbolo + significa “1 o más veces”. Finalmente, para poner uno de los metacaracteres de las expresiones regulares como parte de una expresión regular (como *,+,*[* o *]*), debe ser precedido por el símbolo \, por ejemplo para el símbolo * se debe colocar *.

Nuestras expresiones regulares para cada token fueron las siguientes:

- INICIA: **r’inicia’**, que corresponde a la palabra “inicia”.
- CREA: **r’crea’**, que corresponde a la palabra “crea”.
- ABRE: **r’abre’**, que corresponde a la palabra “abre”.
- INGRESA: **r’ingresa’**, que corresponde a la palabra “ingresa”.
- LISTA: **r‘lista’**, corresponde a la palabra “lista”.

- MUESTRA: **r'muestra'**, corresponde a la palabra "muestra".
- CIERRA: **r'cierra'**, corresponde a la palabra "cierra".
- TERMINA: **r'termina'**, corresponde a la palabra "termina".
- NOMBRE: **r'[A-ZÑ][a-zñ]+[ ][A-ZÑ][a-zñ]+'**, corresponde a una palabra que comienza con una letra mayúscula seguida de 1 o más letras minúscula, luego un espacio en blanco y nuevamente una letra mayúscula seguida de 1 o más letras minúscula.
- DIRECCION: **r'[A-ZÑ][a-zñ]+[ ][0-9]+'**, corresponde a una palabra que comienza con una letra mayúscula seguida de 1 o más letras minúscula, luego un espacio en blanco y a continuación 1 o más números entre 0 y 9.
- NUMERO: **r'[1-9]+'**, corresponde a una palabra con 1 o más números entre 0 y 9.
- ARCH: **r'[a-zñ][a-zñ]* \. txt'**, corresponde a una palabra que comienza con una letra minúscula seguida de 1 o más letras minúsculas, luego un "." y a continuación la secuencia "txt".
- OCUPACION: **r'([a-zñ]+[ ]*)+'**, corresponde a una palabra que comienza con 1 o más letras minúsculas, seguida de 0 o más espacios en blanco, todo esto repetido 1 o más veces.
- APARENT: **r'\('**, corresponde a la palabra "(".
- CPARENT: **r'\)'**, corresponde a la palabra ")".
- COMA: **r'\,'**, corresponde a la palabra ",".

```
reservadas = {
    'inicia':   'INICIA',
    'crea':     'CREA',
    'abre':     'ABRE',
    'ingresa':  'INGRESA',
    'lista':    'LISTA',
    'muestra': 'MUESTRA',
    'cierra': 'CIERRA',
    'termina': 'TERMINA'}
# Expresión Regular para INICIA
def t_INICIA(t):
    r'inicia'
    t.type = reservadas.get(t.value, 'INICIA')
    return t
```

Fig. 1:*Diccionario con las palabras reservadas y función t_INICIA.*

```
# Expresión Regular para los Nombre
def t_NOMBRE(t):
    r'[A-ZÑ][a-zñ]+[ ][A-ZÑ][a-zñ]+'
    t.value = t.value
    return t
```

Fig. 2: *función t_NOMBRE.*

Para asignar estas expresiones a los tokens se crearon funciones nombradas como *t_NombreDelToken(t)* dentro del archivo *AnalizadorLexico.py*, donde *t* corresponde al string que se analizará. Dentro de la función, se define la expresión regular que corresponde, luego realiza la asignación *t.type = reservadas.get(t.value, NombreDelToken)* para las palabras reservadas (**Fig. 1**) (donde reservadas corresponde a un diccionario con las palabras reservadas) y la asignación *t.type = t.type* para el resto de tokens (**Fig. 2**). Finalmente se retorna t.

```
t_APARENT  = r'\('
t_CPARENT  = r'\)'
t_COMA     = r'\,'
```

Fig. 3: *expresiones regulares para los tokens APARENT, CPARENT y COMA*.

Para los tokens correspondientes a símbolos, sólo se realizó la asignación *t_ NombreDelToken = ExpresiónRegular,* como se muestra en **Fig.3**.

Para que las palabras reservadas no sean reconocidas como un token OCUPACION ni que el número de DIRECCION sea reconocido como NUMERO, las funciones se definieron en el siguiente orden: Primero todas las funciones correspondientes a las palabras reservadas, luego t_NOMBRE, t_DIRECCION, t_NUMERO, t_ARCH y t_OCUPACION (en ese orden). Finalmente, se definió la función t_newline, que registra los saltos de línea, se asignó t_ignore = ' \t' que corresponden a los caracteres que serán ignorados (espacio en blanco y tabulaciones) y por último la función t_error, que imprime en consola cuando exista un carácter que no se identificara como uno de los tokens anteriormente nombrados.
Para instanciar el analizador léxico se utiliza la función lex.lex() que será almacenado en la variable **lexer** para su posterior utilización.

Analizador sintáctico

Para crear el analizador sintáctico, primero se identificó la gramática que genera el lenguaje:

- S → inicia A termina
- A → A A
- A → crea(arch)
- A → abre(arch)
- A → ingresa(numero, nombre, numero, ocupacion, direccion)
- A → lista
- A → muestra(numero)
- A → cierra

El problema se presentó al tener como requisito del programa que éste ejecute las instrucciones a medidas que se ingresan, ya que esta gramática sólo funciona para entradas que estén entre un “inicia” y un “termina”. Por lo tanto, la gramática utilizada para este programa es la siguiente:

- A → inicia
- A → termina
- A → crea(arch)
- A → abre(arch)
- A → ingresa(numero, nombre, numero, ocupacion, direccion)
- A → lista
- A → muestra(numero)
- A → cierra

Para implementar esta gramática dentro de Python se creó el archivo *AnalizadorSintactico.py* y por cada producción se definió una función y se ingresó la producción utilizando los tokens definidos en AnalizadorLexico.py. Las funciones y sus producciones son las siguientes:

Función	**Producción**
p_inicia(p)	‘A:INICIA’
p_termina(p)	‘A:TERMINA’
p_crea(p)	‘A:CREA APARENT ARCH CPARENT’
p_abre(p)	‘A:ABRE APARENT ARCH CPARENT’
p_ingresa(p)	‘A:INGRESA APARENT NUMERO COMA NOMBRE COMA OCUPACION COMA DIRECCION CPARENT’
p_lista(p)	‘A:LISTA’
p_muestra(p)	‘A:MUESTRA APARENT NUMERO CPARENT’
p_cierra(p)	‘A:CIERRA’

Donde *p* (parámetro de la función) es una lista con los valores de los tokens.

Luego de ingresar la gramática, se ingresa el código en Python que realizará esa producción, según lo indicado en la introducción de este informe. Como se utilizó una interfaz donde se mostrarán las salidas de las funciones, se creó la variable global *result*, donde se añadirán los strings que se mostrarán en pantalla.
Las funciones realizan las siguientes acciones escritas en Python:

- **p_inicia**: agrega a *result* el string "Bienvenido! \n"
- **p_termina**: agreda a *result* el string "Adios \n"
- **p_crea**: abre un archivo, con el nombre correspondiente al valor del token ARCH (almacenado en *p[3]*), en la carpeta "Datos" con permiso "w+", el cual indica que se debe crear un nuevo archivo y reemplazar, en caso de que ya exista uno con el mismo nombre, y luego se debe abrir con permisos de lectura y escritura. Además, este archivo es almacenado en la variable global *f*, para que se pueda leer y cerrar posteriormente.

```
def p_abre(p):
    'A : ABRE APARENT ARCH CPARENT'
    try:
        global f
        f = open('./Datos/' + p[3], "r+")
    except:
        result.append("El archivo no existe!\n")
```

Fig. 4: *función p_abre(p).*

- **p_abre**: mediante *try*, intenta abrir un archivo en la carpeta "Datos" con permiso "r+", con el nombre correspondiente al valor del token ARCH (almacenado en *p[3]*), y almacenarlo en *f*. Este permiso indica que se debe abrir un archivo ya existente con permisos de lectura y escritura. En el caso de que el archivo no exista, se añadirá a la variable *result* el string "El archivo no existe! \n" (**Fig. 4**).

```
def p_ingresa(p):
    'A : INGRESA APARENT NUMERO COMA NOMBRE COMA NUMERO COMA OCUPACION COMA DIRECCION CPARENT'
    global f
    try:
        f.write("{:<3} {:<18} {:<4} {:<18} {:<18}\n".format(p[3], p[5], p[7], p[9], p[11]))
    except:
        result.append("No se ha abierto ningún archivo!\n")
```

Fig. 5: *función p_ingresa(p).*

- **p_ingresa**: mediante *try*, intenta escribir en el archivo almacenado en *f* los valores de los tokens NUMERO, NOMBRE, NUMERO, OCUPACION y DIRECCION, almacenado en *p[3]*, *p[5]*, *p[7]*, *p[9]* y *p[11]* respectivamente. Para que los campos estén a la misma distancia sin importar la cantidad de caracteres, se realizó un *format* al string, como se muestra en **Fig. 5**. En caso de que *f* no tenga almacenado ningún archivo, se añadirá a *result* el string "No se ha abierto ningún archivo!\n".

```
def p_lista(p):
    'A : LISTA'
    try:
        global f
        f.seek(0)
        result.append("{:<3} {:<18} {:<4} {:<18} {:<18}\n".format("ID", "Nombre", "Edad", "Ocupación", "Dirección"))
        result.append("{:<3} {:<18} {:<4} {:<18} {:<18}\n".format("--", "------", "----", "---------", "---------"))
        result.append(f.read()[0:-1] + "\n")
    except:
        result.append("No se ha abierto ningún archivo!\n")
```

Fig. 6: *función p_lista(p).*

- **p_lista**: mediante *try*, intenta leer un archivo almacenado en *f*. Primero se agrega a *result* los strings "ID", "Nombre", "Edad", "Ocupación" y "Dirección" con el mismo formato que la función *p_ingresa*, junto con los strings "--", "------", "----", "--------" y "---------" con el mismo formato anterior, con el objetivo de generar una tabla y que se pueda apreciar de mejor manera qué significa cada campo. Finalmente, se añade el contenido del archivo concatenado con el string "\n" (**Fig. 6**). En caso de que *f* no tenga almacenado ningún archivo, se añadirá a *result* el string "No se ha abierto ningún archivo!\n".

```
def p_muestra(p):
    'A : MUESTRA APARENT NUMERO CPARENT'
    try:
        f.seek(0)
        result.append("{:<3} {:<18} {:<4} {:<18} {:<18}\n".format("ID", "Nombre", "Edad", "Ocupación", "Dirección"))
        result.append("{:<3} {:<18} {:<4} {:<18} {:<18}\n".format("--", "------", "----", "---------", "---------"))
        for i in f.read()[0:-1].split("\n"):
            if(p[3] == i.split(" ")[0]):
                result.append(i + "\n")
    except:
        result.append("No se ha abierto ningún archivo!\n")
```

Fig. 7: *función p_muestra(p).*

- **p_muestra**: mediante *try*, intenta leer un archivo almacenado en *f*. Primero, se añaden los strings con formato descritos en la función *p_lista*, con el mismo objetivo indicado en esa función, luego se separa el texto que se leyó del archivo por cada salto de línea mediante la función *split("\n")*, con el objetivo de tener cada registro por separado y se recorre cada uno de estos. Por cada registro, se compara el primer valor de éste (Código), obtenido mediante *split(" ")[0]*, con el valor del token NUMERO, almacenado en *p[3]*, si es verdadero, se agrega a *result* (**Fig. 7**). En caso de que *f* no tenga almacenado ningún archivo, se añadirá a *result* el string "No se ha abierto ningún archivo!\n".
- **p_cierra**: cierra el archivo almacenado en la variable global *f*.

Finalmente, se añadió la función *p_error(p)* que no tiene gramática e indica cuando hay un error de sintaxis, agregando a *result* el string "Error de sintaxis!\n"

```
def Analizar(palabra):
    global result
    result = []
    parser.parse(palabra)
    return result
```

Fig. 8: *función Analizar(palabra)*.

Para instanciar el analizador léxico se utiliza la función *yacc.yacc()* que será almacenado en la variable *parser*. Para realizar la etapa de análisis sintáctico de una entrada, se definió la función *Analizar(palabra)* (**Fig.8**), la cual asigna a la variable *result* (utilizada por las funciones anteriormente mencionadas) una lista vacía y realiza el análisis mediante el método *parser.parse(palabra)*. Finalmente retorna la lista *result* con los strings que se mostrarán en pantalla

Interfaz

Para la interfaz se utilizó la librería PyQt5 de Python, donde la clase *Ui_MainWindow* corresponde a la ventana que se desplegará y los objetos de esta ventana se añadirán como atributos de esta clase.

```
def pulsa(tecla):
    if(str(tecla) == "Key.enter"):
        analizar_presionado()

escuchador = kb.Listener(pulsa)
escuchador.start()
```

Fig. 9: *función pulsa(tecla)*.

Además, se utilizó keyboard de la librería pynput, que mediante la función *pulsa(tecla)* y los métodos mostrados en **Fig. 9**, "escucha" las teclas pulsadas por el usuario. Al apretar la tecla "Enter", se llama a la función *analizar_presionado()* (**Fig. 10**).

```
def analizar_presionado():
    lista = self.text_Entrada.toPlainText().split("\n")
    analizar = ''
    for i in range(len(lista) -1, -1, -1):
        if(lista[i] != ''):
            analizar = lista[i]
            pos = i
            break
    global prepos
    global preanalizar
    if(analizar != '' and (pos != prepos or analizar != preanalizar)):
        resultado = Analizar(analizar)
        prepos = pos
        preanalizar = analizar

        if(len(resultado) > 0):
            terminar = 0
            if(self.text_salida.toPlainText() == ""):
                salida=""
            else:
                salida = "\n"
            for i in resultado:
                if(i == "Adios\n"):
                    terminar = 1
                salida = salida + i

            font = QtGui.QFont("Consolas")
            font.setPointSize(12)
            self.text_salida.setFont(font)
            self.text_salida.insertPlainText(salida)
            if(terminar == 1):
                time.sleep(1.5)
                os._exit(1)
```

Fig. 10: *función analizar_presionado()*.

En primer lugar, esta función separa el texto ingresado por el usuario en la interfaz (almacenado en el atributo *text_entrada*), utilizando *split("\n")*, luego recorre el arreglo mediante un ciclo *for*, partiendo desde el último elemento y selecciona el primer campo distinto de "", guardando el string en la variable *analizar* y su posición en el arreglo en la variable *pos*. Esto tiene como objetivo seleccionar la última línea escrita por el usuario, sin contar los "Enters" que pueden encontrarse accidentalmente por delante del texto escrito.

Posteriormente, se comprueba que *analizar* sea distinto de "" (por sí solo se ingresaron "Enters") y se revisa que *pos* no sea igual a la variable *prepos*, que almacena la posición de la última línea ejecutada, o que *analizar* no sea igual a la variable *preanalizar*, que almacena el string de la última línea analizada. Esto tiene como objetivo que la instrucción no se siga ejecutando si el usuario solo ingresa "Enters", pero que sí se ejecute en caso de que se realice algún cambio a esa línea. Si la condición se cumple, entonces se le asigna a *preanalizar* el

valor de *analizar* y se le asigna a *prepos* el valor de *pos* (ya que es la última instrucción que se ejecutará) y se realiza el análisis sintáctico mediante la función *Analizar(analizar)* descrita en la sección anterior y se almacena en la variable *resultado*.

Se comprueba que el largo de *resultado* del análisis sea mayor a 0, ya que hay funciones que no retornan texto (como *p_cierra* o *p_abre*). Si es mayor, se imprime un salto de línea para distinguir el resultado de distintas entrada, luego se recorre *resultado* y cada línea se añade a la variable *salida*, que será la que se mostrará en la interfaz mediante el comando *self.text_salida.insertPlainText(salida)*.

Para terminar el programa con la instrucción “termina”, surgió un problema, ya que, al usar una interfaz, el programa se debe cerrar desde el archivo que despliega la interfaz. Además, cuando se ingresa esta instrucción, se debe mostrar en pantalla el texto “Adios” antes de cerrar el programa. La solución fue crear una condición al recorrer la variable *resultado*, cuando la salida sea igual a “Adios\n” (retornado por la función *p_termina*), se cambiará el valor de la variable *termina* a 1, lo que hará que se cierre el programa luego de un tiempo de espera de 1,5 segundos, realizado mediante el comando *time.sleep(1.5)*.

Especificación del programa

Estructura

Nuestro programa presenta las siguientes carpetas:

- **Código**: Esta carpeta contiene los archivos AnalizadorLexico.py, AnalizadorSintactico.py, interfaz.py, parser.out y parsetab.py, además de la carpeta Datos.
- **Ejecutable**: Esta carpeta contiene los archivos mencionados anteriormente, CompiladorBDP.exe y otros archivos necesarios para su correcto funcionamiento.

Como se mencionó en el desarrollo, AnalizadorLexico.py contiene el código que utiliza la librería ply.lex, AnalizadorSintactico.py contiene las funciones escritas usando la librería ply.yacc, estos corresponden al analizador léxico y sintáctico del programa respectivamente. Al ejecutar el archivo AnalizadorSintactico.py, se generan automáticamente parser.out y parsetab.py, utilizados para realizar el análisis correspondiente. Finalmente, interfaz.py contiene el código de la interfaz del programa, donde se leerán las instrucciones y se mostrará su resultado.

Para una mejor vista al código, en caso de un posterior análisis, se creó la carpeta "Ejecutable", que contiene todos los archivos que genera la librería auto-py-to-exe al crear el ejecutable llamado *CompiladorBDP.exe,* lo que dificulta la detección de los archivos importantes.

Requisitos

El programa fue construido a través de python, en su versión 3.11.1, con las librerías ply, pynput y PyQt5 en sus versiones 3.11, 1.7.6 y 5.15.7 respectivamente.
Este programa no posee un requerimiento de hardware específico ya que su uso no supera los 23 MB de memoria RAM en el sistema. Para ejecutar este programa, bastará con ingresar a la carpeta ejecutable y hacer doble clic en *CompiladorBDP.exe*. Si existe algún problema de permisos o similar, puede desactivar su antivirus, si no lo desea, será necesario instalar Python 3, luego instalar pip con el comando *python get-pip.py*, además tendrá que instalar la librerías "ply" con el comando *pip install ply* y "pynput" con *pip install pynput*, finalmente para la interfaz gráfica sera necesario instalar la librería PyQt5 con el comando *pip install pyqt5*, luego abrir una terminal en el directorio "Código" y escribir el comando *python3 interfaz.py*.

Funcionamiento

Al ejecutar el programa con alguno de los métodos descritos anteriormente, se puede visualizar lo mostrado en **Fig. 11**. En la parte superior se muestra el nombre del programa, junto con las instrucciones que se pueden ingresar. En el recuadro izquierdo, el usuario podrá ingresar las instrucciones y en el recuadro derecho se mostrarán los respectivos resultados.

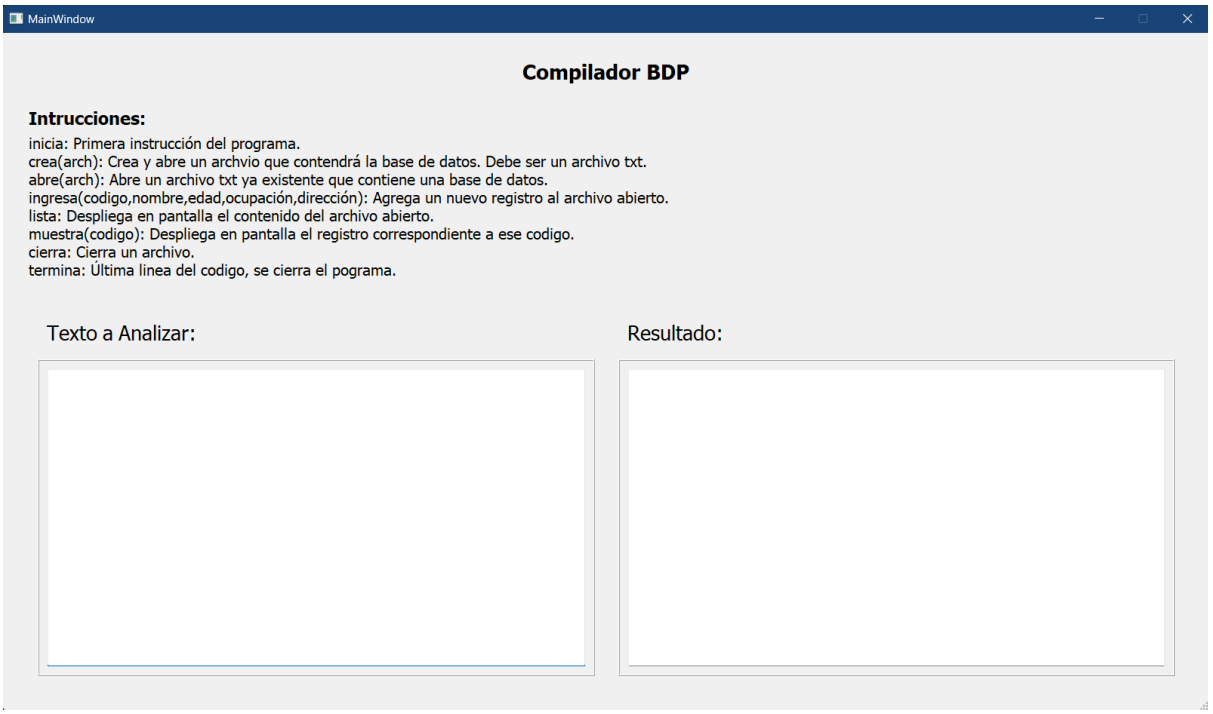

Fig. 11: *interfaz del programa*.

En **Fig. 12** se muestra un ejemplo del funcionamiento, donde la primera instrucción es "inicia" y cada vez que se presione la tecla "Enter", se realizará el análisis y se mostrará el resultado en pantalla.

En **Fig. 13** se muestra el resultado de distintas instrucciones, junto con un error de sintaxis, que ocurre cuando una instrucción se ingresó de forma incorrecta. Si los resultados exceden el recuadro, se creará una barra lateral, donde se podrán apreciar todas las salidas del programa durante la ejecución.

Para finalizar el programa, se puede presionar el símbolo "X" que se encuentra en la esquina superior derecha o escribir la instrucción "terminar", que muestra un mensaje de despedida y cierra el programa luego de 1,5 segundos.

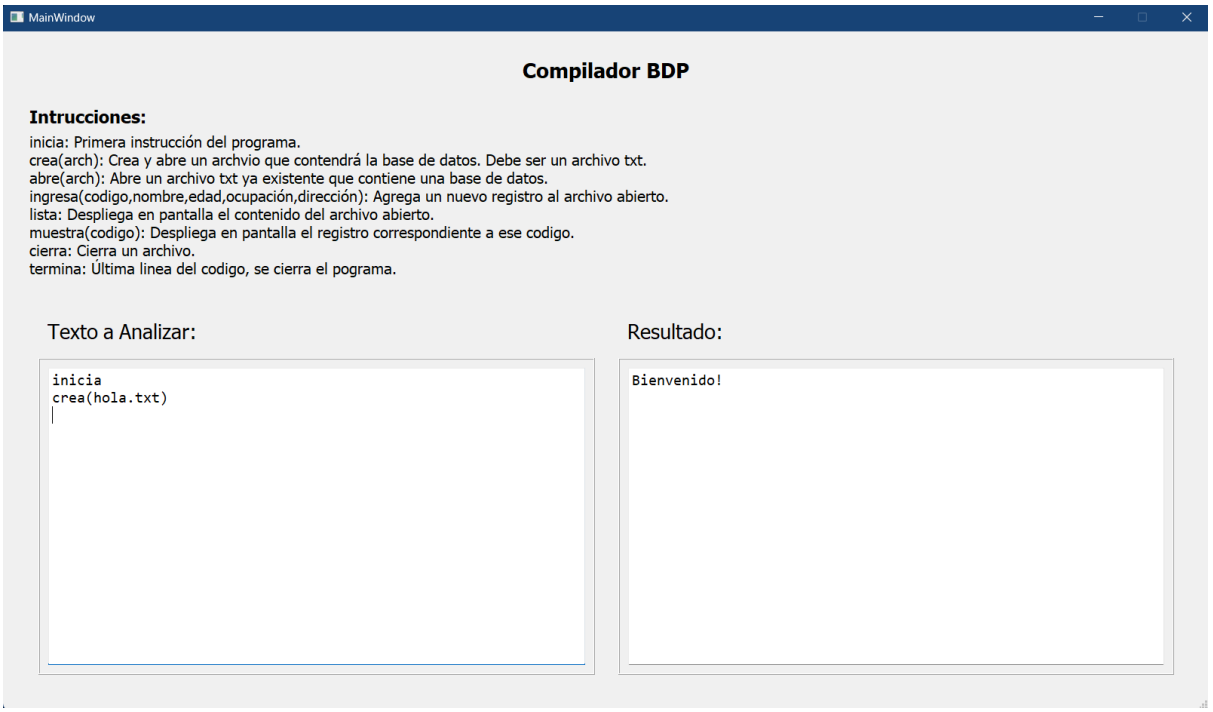

Fig. 12: *interfaz del programa con primeras instrucciones.*

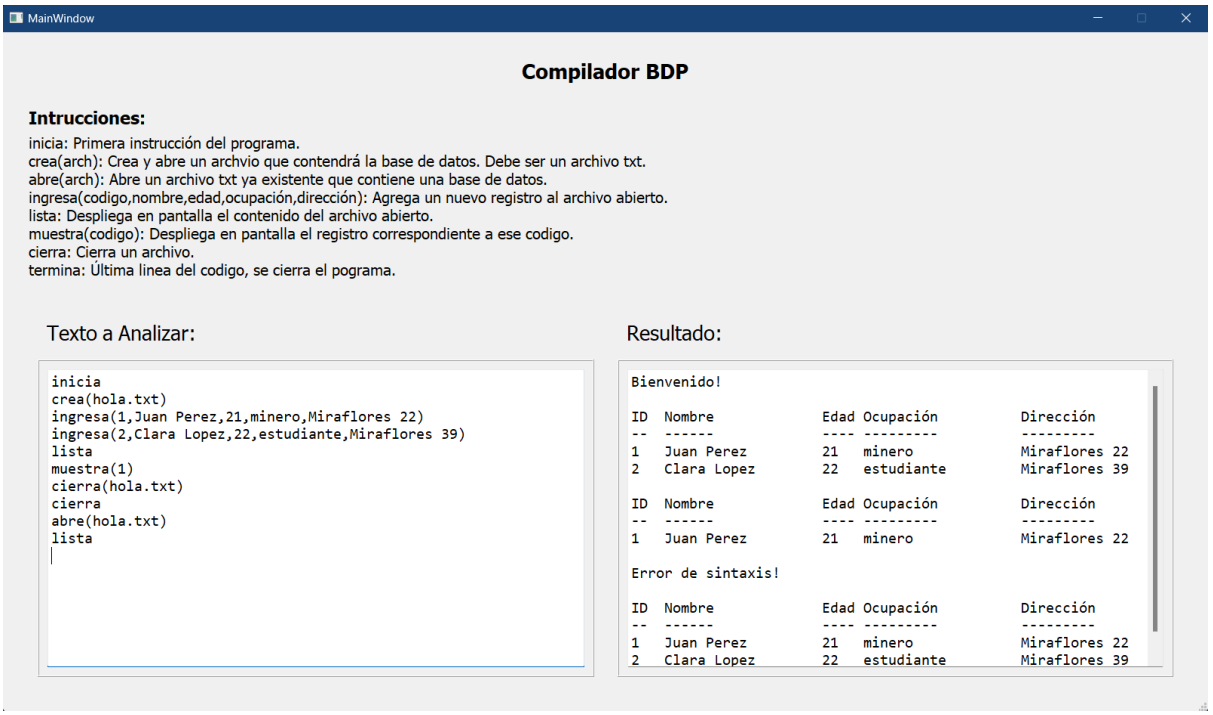

Fig. 13: *interfaz del programa tras varias instrucciones.*

Conclusión

La importancia de los compiladores radica en la traducción de un lenguaje de alto nivel a un lenguaje legible por la máquina llamado código objeto, esto nos permite escribir las instrucciones en un lenguaje muy parecido al de los humanos. Además, de examinar las instrucciones en busca de errores, informando al usuario.

LEX y YACC suelen ser utilizados juntos, generando analizadores eficientes, incluso más que los que pudiéramos hacer de manera manual. LEX nos permitió reconocer expresiones regulares en una cadena de inputs en cadena de caracteres que coinciden con las expresiones regulares, ejecutando las instrucciones de los programas proporcionadas por los usuarios y YACC nos facilitó la aceptación de una entrada como gramática de libre contexto, produciendo como salida el autómata finito que reconoce el lenguaje generado por dicha gramática. Aunque YACC no acepta todas las gramáticas, sino, sólo las LALR. Además, nos permitió asociar acciones a cada regla de producción y asignar un atributo a cada símbolo de la gramática (terminal o no terminal). Esto facilitó al desarrollador la creación de esquemas de traducción en un reconocimiento dirigido por sintaxis.

A lo anterior, también podemos destacar lo importante que es comprender estos procesos (análisis léxico y análisis sintáctico) pues estos son una parte fundamental de la compilación. Es fundamental entender el proceso de compilación de un programa, es algo que como programadores usamos todos los días, y no está de más comprender cómo es que nuestros códigos se llegan a ejecutar.

Como limitaciones y mejoras de nuestro programa, destacamos los siguientes puntos:

- Detección y corrección de errores: Aunque en el proyecto no se requiere tener una corrección de errores nos limita a que todas las ejecuciones tienen que ser correctas, de lo contrario nuestro programa mostrará un mensaje de error de sintaxis sin especificar donde ocurrió tal error. Por ende, sería una muy buena mejora añadir un buen manejo de errores y que se especifique claramente de qué tipo son y donde ocurren (ejemplo: Error: ingresó un carácter no admitido en ingresa(1, Juan Perez5, 23, estudiante24, Miraflores 123)).
- Solo acepta archivos .txt: Al momento de crear un archivo con *crea()*, se tiene que especificar que el archivo es txt, pues nuestro programa no está capacitado para recibir otro tipo de formatos para almacenar los datos, debido a la diferencias de sintaxis al momento de crear y almacenar información en cada uno de estos (por ejemplo en archivos .csv los campos se separan por *","* y para archivos *".xlsx"* habría que implementar más librerías). Una posible mejora sería implementar al menos 3 tipos de formatos para guardar nuestros datos, por ejemplo archivos ".csv", ".xlsx" etc.

- Sin orden: Al cambiar la gramática no tenemos un orden específico en el que se ingresan las instrucciones, esto debido a que se adaptó la gramática para que las instrucciones se ejecuten línea por línea (por ejemplo la primera instrucción del programa puede ser *crea(archivo.txt)*). Una mejora podría ser verificar el orden en el que se ingresan las instrucciones, de esta manera verificaremos que la primera instrucción siempre tiene que ser *inicia*.

Bibliografía

- Documentación de PLY.
- Documentación de PyInput.
- Documentación de PyQt5.
- Documentación de Python 3.11.
- Documentación de Pip.