INFO165 - Compiladores

Tarea

Profesora: Maria Eliana de la Maza W.

Integrantes

- Jorge González
- Nicol Huaiquil
- Gustavo Reyes

12 de diciembre del 2022

Introducción

A lo largo de la historia de informática han surgido diferentes programas de distinta generación, primera generación que son los programas que utilizan lenguaje de máquina en código binario, segunda generación es un lenguaje de símbolos y tercera generación es el lenguaje de alto nivel, el más entendido por las personas.

Un compilador es un traductor específicamente para un lenguaje de alto nivel a lenguaje de máquina. El compilador cuenta con diferentes fases necesarias para poder realizar su función, las cuales son: analizador léxico, analizador sintáctico, generador de código intermedio, optimizador de código, generador de código, manejo de errores en cada una de estas fases y el administrador de la tabla de símbolos.

El analizador léxico entrega tokens en base a las instrucciones que se entrega al programa, estas se van leyendo por el analizador sintáctico, y este revisa si estas instrucciones pertenecen a la gramática del lenguaje. En la fase de análisis sintáctico, se utiliza un traductor dirigido por sintaxis en conjunto con la tabla de símbolos y un manejador de errores.

En este informe se detalla el trabajo realizado de un traductor dirigido por sintaxis para el lenguaje BDP (base de datos de personas), el cual deberá aceptar una serie de instrucciones para el manejo de una base de datos y efectuar las acciones a medida que se realiza el análisis sintáctico. Para esto utilizamos un analizador léxico y un analizador sintáctico, ambos producidos por los programas generadores JFLEX y JCUP respectivamente.

Desarrollo

De la tarea anterior, el archivo “lexer.flex”, el cual ya contiene una parte de los tokens necesarios para el trabajo, fue renombrado a “**LexerC.flex**” y se le aplicaron una serie de modificaciones. Estas modificaciones fueron hechas para integrar el analizador léxico generado con JFLEX y con el analizador sintáctico producido por JCUP. En el archivo modificado, todas las palabras que antes eran conocidas con el token “PALABRA_RESERVADA”, ahora cuentan con su token único (la palabra inicia ahora es reconocida con un token “INICIA”, por ejemplo). En adición a los tokens ya existentes, se añadieron:

- **“ARCHIVO”** es un token para reconocer archivos con un formato cualquiera. Un archivo puede ser una combinación de dígitos, letras mayúsculas o minúsculas, seguido de un punto y seguido de otra combinación de dígitos y letras mayúsculas o minúsculas.
- **“OCUPACION”** es un token para reconocer la ocupación de una persona. Una ocupación puede ser una combinación de letras minúsculas, seguida de una o más letras minúsculas o espacios.
- **“ENTERO”** es un token para reconocer números enteros. Un número entero es uno o más dígitos desde el 0 al 9.
- **“MUESTRA”, “CIERRA”** y **“TERMINA”** son tokens para reconocer las palabras reservadas muestra, cierra y termina, respectivamente.

Se creó el archivo “**a_sintaxis.cup**”, utilizado para generar el analizador sintáctico con JCUP. En este archivo se introdujo la gramática, la cual se utilizará para generar el analizador sintáctico, respetando el formato descrito por el manual de JCUP. Este archivo contiene una sección “**action code**”, en el cual puede ir código adicional que puede ser llamado durante las acciones semánticas. En dicha sección, se creó una clase pública con el nombre “**Archivo**”, la cual contiene la información de un archivo que se haya creado o abierto, también contiene funciones para lectura, escritura y creación de un archivo.

La gramática del lenguaje introducida en el archivo es la siguiente:

1. **comienza → INICIA**
2. **comienza → TERMINA**
3. **comienza → CREA SIMBOLO ARCHIVO SIMBOLO**
4. **comienza → ABRE SIMBOLO ARCHIVO SIMBOLO**
5. **comienza → CIERRA**
6. **comienza → LISTA**
7. **comienza → INGRESA SIMBOLO ENTERO^{1} SIMBOLO NOMBRE SIMBOLO ENTERO^{2} SIMBOLO OCUPACION SIMBOLO DIRECCION SIMBOLO**
8. **comienza → MUESTRA SIMBOLO ENTERO SIMBOLO**

Donde las terminales son:
INICIA, TERMINA, CREA, SIMBOLO, ARCHIVO, ABRE, CIERRA, LISTA, INGRESA, MUESTRA, ENTERO, OCUPACION, NOMBRE y **DIRECCION**

Las producciones de la gramática tienen las siguientes acciones semánticas adjuntas:

- **Producción 2**: Imprime un mensaje, notificando la terminación del programa.
- **Producción 3:** Llama al método crear().
- **Producción 4:** Llama al método abrir().
- **Producción 5:** Llama al método cerrar().
- **Producción 6:** Llama al método getData().
- **Producción 7:** Crea un valor de nombre data, de tipo string con la información contenida en las terminales **ENTERO^{1}.id, NOMBRE.name, ENTERO^{2}.age, OCUPACION.ocup, DIRECCION.dir**; luego, este string es utilizado al llamar al método escribir(data).
- **Producción 8:** Llama al método getData(id), donde id es el valor guardado en la terminal **ENTERO.id**.

La clase "**Archivo**" es una clase pública única, cuyos atributos y métodos son compartidos de manera global. Los métodos son descritos a continuación:

- **crear(nombre)** : Crea y abre un archivo nuevo con el nombre ingresado. Los atributos "name" y "file" de la clase son iniciados con el nombre ingresado.
- **abrir(nombre)** : Abre un archivo existente con el nombre ingresado. Los atributos "name" y "file" de la clase son iniciados con el nombre ingresado.

- **cerrar()**: Cierra el archivo. Estableciendo los valores de los atributos de la clase a “null”.
- **escribe(data)** : Ingresa información “data” en un archivo ya abierto.
- **getData()** : Lee información del archivo ya abierto e imprime todo su contenido. Para ello, recorre línea por línea desde el principio hasta el final, imprimiendo cada una individualmente.
- **getData(id)** : Lee información del archivo ya abierto e imprime la línea cuyo código sea igual a “id”. Para ello, se recorre el archivo desde el principio hasta el final o hasta encontrar la línea cuyo código sea igual a “id”.

Todos los métodos son llamados en las acciones semánticas de las producciones de la gramática.

Finalmente, se escribió un archivo llamado “**PorConsola.java**” que contiene el código principal. Este código se encarga de recibir entradas escritas por pantalla y entregarlas al analizador léxico, el cual le entregará al analizador sintáctico un conjunto de tokens en base a la entrada que el usuario va a ingresar.

Problemas encontrados y sus soluciones

- Al comienzo del desarrollo del trabajo, se iba a hacer uso del generador BISON, el cual es compatible con Java. Debido a la falta de documentación relacionada con la integración de JFLEX y su correcta implementación con el lenguaje que elegimos, optamos por cambiar de generador a JCUP.
- Anteriormente, se presentó una gramática diferente a la que está en este documento. Dicha gramática, es capaz de reconocer las instrucciones pedidas en la tarea, pero solo como un conjunto y no individuales. Tomando en consideración que las instrucciones van a ser ingresadas una a una durante la ejecución en conjunto con lo anterior, se decidió optar por una gramática más simple, que sea capaz de reconocer sólo instrucciones individuales.
- Durante la primera ejecución del programa, hubo dificultades centradas en el archivo “**a_sintaxis.cup**” y el programa generador JCUP. El programa no tiene instalador propio, por lo cual uno debe hacer las configuraciones de instalación manualmente. A pesar de ello, no se logró hacer funcionar el programa de manera apropiada, por lo cual se optó por agregarlo como dependencias del código y adicionalmente, se utilizó un código “**EjecutarJavaCup.java**” para ejecutar el programa.

Especificación del programa

El programa fue escrito en Java, con el soporte del programa de entorno de desarrollo integrado Eclipse. Para la generación de los analizadores léxico y sintáctico se utilizaron los generadores JFLEX y JCUP, respectivamente.

El programa ejecutable "**EjecutableTarea.jar**" contiene las dependencias necesarias y el código escrito. Para ejecutarlo, debe acceder a la línea de comandos de su sistema operativo(CMD, Windows PowerShell o una terminal de linux) en el directorio que contenga al programa ejecutable, y utilizar el comando:

java -jar EjecutableTarea.jar

.
Los requerimientos básicos de hardware y software que se necesitan son:

Hardware:

- PC.
- Teclado y mouse.
- Intel Core i5-4690
- 2 GB de ram mínimo.
- 10 MB de almacenamiento.
- Pantalla.

Software:

- Sistema operativo (Windows o Linux).
- Java 8.
- Java Development Kit 19.0.1 https://www.oracle.com/java/technologies/downloads/#jdk19-windows

Al descomprimir "**Tarea.rar**" encontrará el archivo "**EjecutableTarea.jar"**, "**a_sintaxis.cup"**, "**LexerC.flex"** y un archivo llamado "**Trabajo_Codigo.zip"** el cual contiene el código fuente.

Observación*:* Para ver el código fuente del programa entregado, se recomienda instalar Eclipse, en el cual se debe importar el archivo "**Trabajo_Codigo.zip"** (https://www.eclipse.org/downloads/download.php?file=/oomph/epp/2022-09/R/eclipse-inst-jre-win64.exe)

Ejemplos de funcionamiento

En la Imagen 1 se puede apreciar un conjunto de instrucciones ya ingresadas: la creación del archivo, el ingreso de información al archivo, listar la información del archivo, mostrar un registro individual, el cierre del archivo y la terminación del programa:

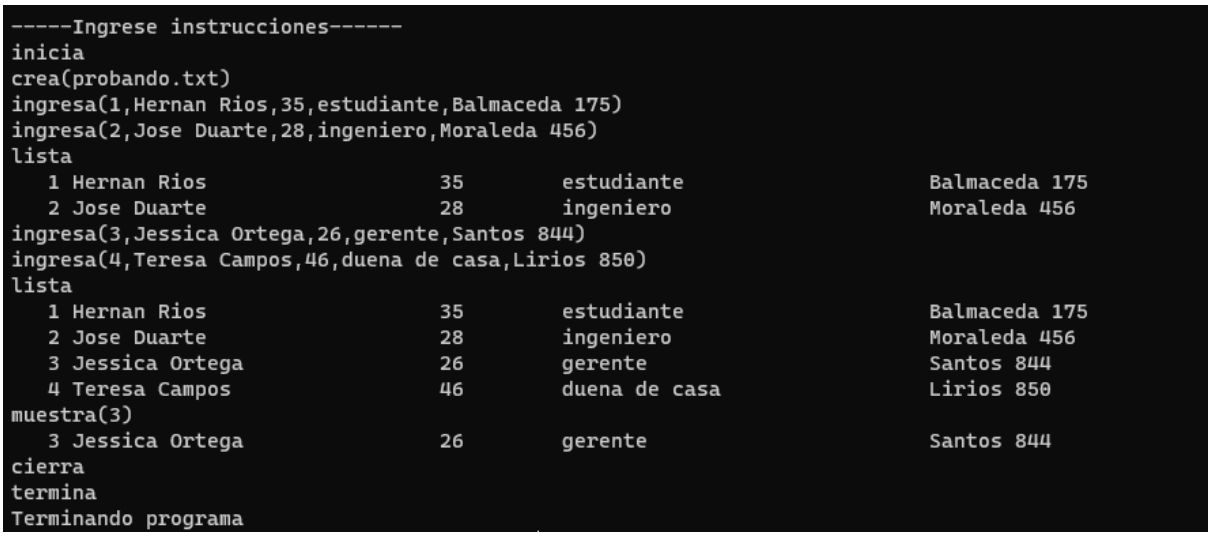

Imagen 1: *Conjunto de instrucciones ya ingresadas.*

La Imagen 2 presenta otro conjunto de instrucciones ya ingresadas, las cuales describen el procedimiento de abrir el archivo que se creó en la imagen anterior, listar la información de dicho archivo, mostrar un registro particular, el cierre del archivo. También es posible crear otro archivo durante la ejecución y aplicar las mismas instrucciones en el archivo nuevo. El programa finaliza al introducir la instrucción termina:

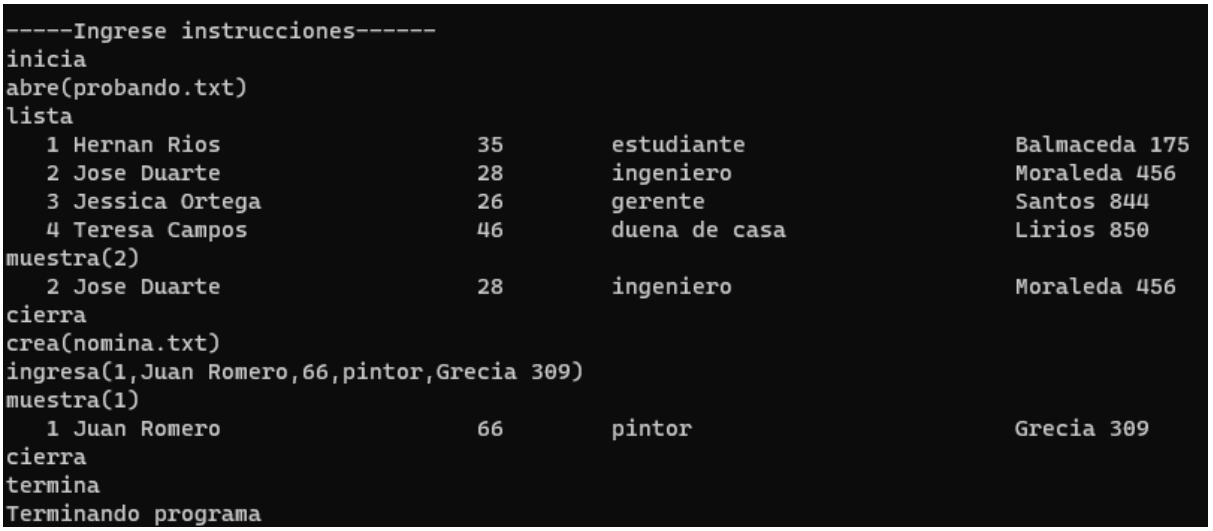

Imagen 2: *Otro conjunto de instrucciones ya ingresadas. Se muestra que no es necesario terminar el programa para abrir o crear otro archivo.*

Las instrucciones deben ser introducidas una a una. A continuación, en la Imagen 3 y 4 se describe la ejecución de una instrucción individual, en este caso, la instrucción muestra(2) es introducida por el usuario:

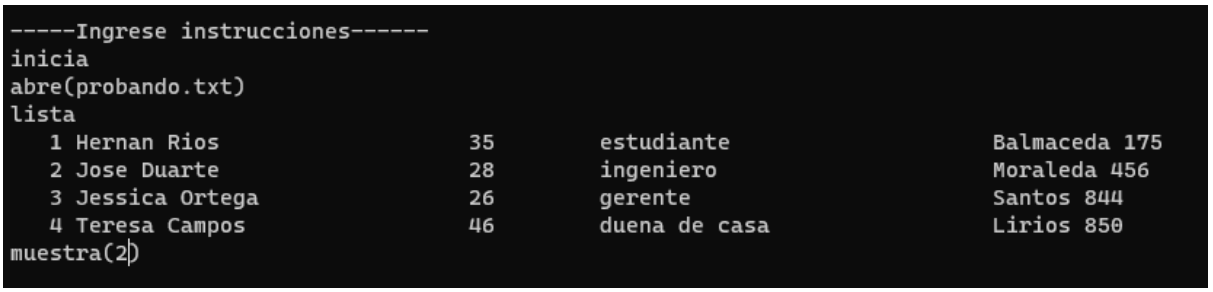

***Imagen 3**: Ejecución del programa. El usuario introduce la instrucción muestra(2)*

Para ejecutar la instrucción, el usuario debe presionar Enter y el programa responderá con la siguiente salida.

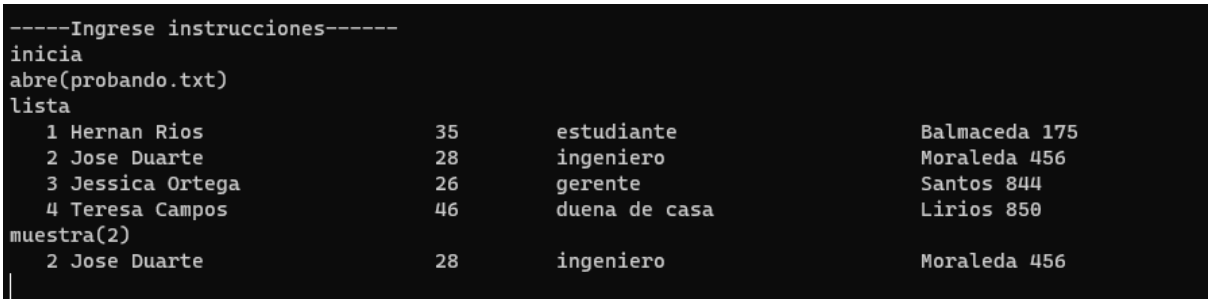

***Imagen 4**: Ejecución del programa. El programa responde con la salida correspondiente.*

Conclusiones

Se ha podido implementar un traductor dirigido por sintaxis para el lenguaje BDP, con el cual se puede abrir y crear archivos, listar el contenido de un archivo completo, mostrar un registro específico indicando su código, ingresar información de nuevas personas, cerrar el archivo abierto y finalmente terminar la ejecución del programa.

Durante el desarrollo, surgieron problemas, los cuales fueron mencionados anteriormente en conjunto con sus soluciones. También se consideró la adición de nuevas características y mejoras que no fueron implementadas ya que no son parte del programa base, pero se podrían agregar en el futuro. Las mejoras consideradas son las siguientes:

- Hacer el programa más responsivo, entregando salidas apropiadas por cada instrucción ingresada.
- Introducir validaciones, para mantener un flujo de instrucciones bien definido.
- Agregar nuevas instrucciones de eliminación de datos en un archivo y también la eliminación de los archivos que son creados por el programa.

Finalmente, durante la implementación del traductor, se hallaron limitantes que no tuvieron solución posible:

- La imposibilidad de ingresar y leer archivos con caracteres que tienen tilde.
- La imposibilidad de ingresar y leer caracteres “ñ”, “ç”, y muchos otros más que son importantes para cualquier otro idioma que no sea inglés.

Bibliografía

CUP User's Manual, Scott E, Hudson, Frank Flannery, C. Scott Ananian, Dan Wang
https://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html

JFlex User's Manual, Gerwin Klein
https://inst.eecs.berkeley.edu/~cs164/sp05/docs/jflex/manual.pdf

Computer Science 330 Computer Language Implementation 2006 Lecture Notes, Capítulo 4. Bruce Hutton.
https://www.cs.auckland.ac.nz/courses/compsci330s1c/lectures/330ChaptersPDF/Chapt4.pdf