Diseño de Compiladores

Entrega Nro 1°

(Trabajo nro 1 y nro 2)

UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS

Grupo 17:

- Faiella, Santiago. (sfaiella@alumnos.exa.unicen.edu.ar) LU: 249929
- Schuenemann, Tomas Ezequiel. (tschuenemann@alumnos.exa.unicen.edu.ar) LU: 249222
- Viduzzi, Franco. (fviduzzi@alumnos.exa.unicen.edu.ar) LU: 249942

FACULTAD DE CIENCIAS EXACTAS
UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES

Temas incluidos para trabajo: **2 8 9 (10) (12) 15* 17 19 (22) 23* 25 26**

2. Enteros cortos sin signo

8. Dobles

9. Discard

15.For y **Continue**

17. Break

19. Diferimiento

23. Sin conversiones

25. Comentarios multilínea (<<comentario>>)

26. Cadenas de 1 línea ('cadena')

- Introducción.

Para este trabajo tomamos la decisión de utilizar Java, dejando el código fuente en un repositorio de Github, y también todos los diagramas correspondientes.

Uso del programa:

Como aclaración para que el .jar ejecute y cargue correctamente se debe de colocar los siguientes archivos en una carpeta testFiles donde el ejecutable .jar es una desventaja de la decisión de utilizar archivos pero consideramos que la versatilidad de poder cambiar el comportamiento del mismo sin tener que compilar de vuelta el ejecutable lo compensa:

- palabras_reservadas.txt
- TablaDistintosSimbolos.txt
- TablaSemantica.txt
- TablaTransicion.txt

-Código fuente:
https://github.com/Beltonidas/CompiladoresTrabajo

- Parte del Analizador Léxico

- **GestorArchivo**: Es el encargado de levantar el archivo donde se encuentra el código fuente del programa a compilar, donde se optó por usar listas de listas de caracteres con la finalidad de facilitar el saber en qué línea y caracter el analizador léxico se encuentra. Esto es de mucha utilidad ya que a la hora de tener errores en los códigos podemos determinar la línea o el carácter con el llamado de un método.

- **MatrizTransicion:** El objeto representa la tabla de transición de estados, donde con una matriz de dos dimensiones "matrizEstado" que se carga del archivo que se encuentra en "testFiles/tablaTransicion.txt", en el que se encuentran los valores enteros correspondientes al autómata finito. También se tiene otra estructura llamada "tablaSimbolos" que es la encargada de detectar los distintos caracteres y retorna su respectiva columna que lo representa, por ejemplo, si se encuentra el caracter "!" el método "identificarSimbolo(caracter)" nos retorna el valor 10 representando la columna en la tabla de transición.
  Como funcionalidad principal tenemos a "transicionCaracter(caracterArchivo)", metodo que cambia el estado interno y llama al método "dispararAccionSemantica" con la accion

correspondiente al estado de la matriz.

- **MatrizAccionSemantica:** En esta clase su principal funcionalidad es contener las Acciones Semanticas y a medida que se van leyendo los distintos caracteres se van ejecutando para corroborar que no existen errores o se van añadiendo caracteres a los lexemas de los Tokens.
  Como principal funcionalidad es quien ejecuta "dispararAccionSemantica(indexFila, indexCol, caracterArchivo)". Dentro de este método el valor de accion es la que determina qué acción semántica se va a disparar.
  Entonces:
  - accion >= 0 → se llama una acción semántica y se ejecuta
  - accion == -1 → es un error detectado en la tabla de transición de estado
  - accion == -2 → no se hace nada por lo que solo se avanza con la lectura.

  Por otro lado la variable respuesta, depende de la accionSemantica que se ejecutó, por lo que si:

  - respuesta < 0 → se trata de un error, se notifica como tal y ademas se descarta el token incompleto como este.
  - respuesta = 0 → se trata de una accion que verifica el lexema del token actual y por tanto no se avanza con la lectura dado que el caracter que acaba de venir no es parte del mismo.
  - respuesta > 0 → se trata de una accion que consume el caracter que vino, sea que lo agrega al lexema o simplemente avanza el contador de caracter.

  Dependiendo de la variable accion es si se entrega el token al analizador léxico y dependiendo de respuesta y accion es que se avanza con la lectura del siguiente caracter.

- **TablaSimbolos:** En esta clase por un lado se contienen los Tokens y por otro lado se contienen las palabras reservadas, estas mismas se cargan a través del método "inic()" que básicamente lee un archivo e inicializa los hashMap de las palabras reservadas y los tokens.

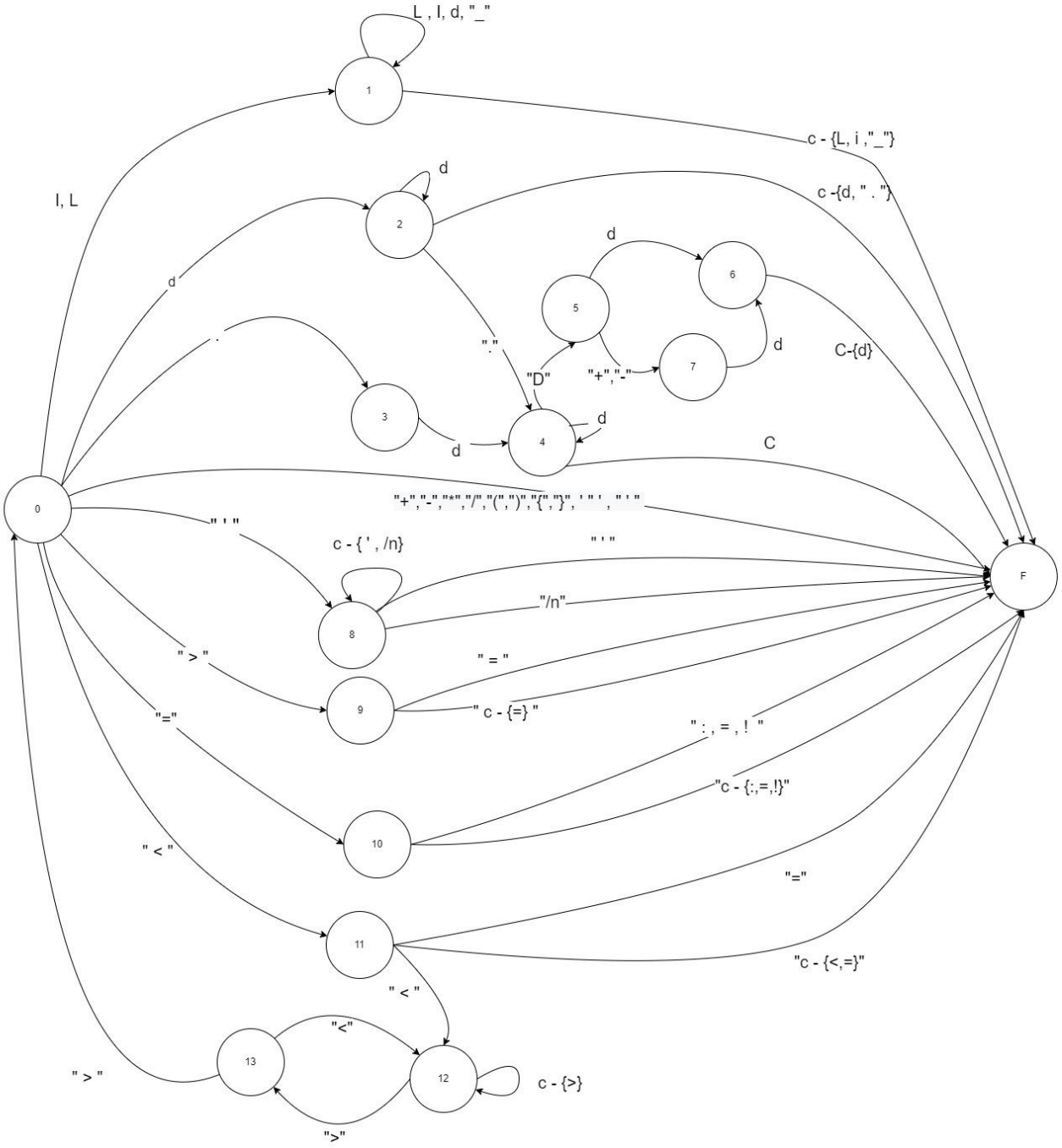
- **TokenLexema:** Es simplemente las variables id y lexema, dos tipos de constructores y sus respectivos métodos de get, set e equals. La funcion que tiene es por si mas adelante en el desarrollo del compilador se requiere agregar informacion que no se tuvo en cuenta.

- **AnalizadorLexico:** La función del método "siguienteToken()" es retornar un valor entero correspondiente al siguiente token encontrado en el código fuente, entonces principalmente se itera por la cantidad de líneas del código fuente y a su vez por la cantidad de caracteres que corresponde a la línea que se está procesando. Se obtiene el nuevo carácter y se llama al método "transicionCaracter(carácter)" de la clase MatrizTransicion hasta que la variable de AnalizadorLexico tokenEntregar sea diferente a -1, en cuyo caso retorna el valor de tokenEntregar, o hasta que no hay más caracteres por procesar en el archivo de código fuente, en cuyo caso entrega -1. En el caso que el parser requiera del lexema correspondiente al token recién entregado entonces se accede al mismo por el atributo anteriorToken y se puede consultar la información que se desee del mismo.

- **AccionSemantica:** esta clase contiene como atributo estatico un "TokenLexema" con el nombre de "tokenActual" y también consta de 4 métodos con el nombre "ejecutar(carácter)", "getToken()" "getNewToken()" y "getNewToken(string)".
  para esta clase se generaron otras clases que extienden de AccionSemantica para poder implementar el método "ejecutar(carácter)".

- **Diagrama de transición de estados.**

Automata completo



Referencia al autómata:
https://drive.google.com/file/d/1ms0BUiSBMHlZTSoT9HsqXrVsEqMrpguh/view?usp=sharing

- Matriz de transición de estados.

Referencia a la matriz de estado:
https://docs.google.com/spreadsheets/d/1avik6z0EhykK16zyVaa4ku2DkqfJWdhhXy-l7uFsJkM/edit#gid=2014562682

- Lista de acciones semánticas asociadas a las transiciones del autómata del Analizador Léxico, con una breve descripción de cada una:

1. **AddCaracterToken**: Agrega un caracter al lexema del token actual
2. **VerificarIdentificador**: Corrobora que el lexema del token no contenga más de 25 caracteres, en el caso de que se tenga más de 25 caracteres el token se trunca a 25 y se elimina el sobrante.
3. VerificarRangoEntero: Verifica que el valor del lexema parseado a entero no supere 255, caso contrario se setea a 255.
4. **VerificarRangoDouble:** Verifica el rango del lexema parseado a double y caso contrario lo setea a 0.0. Para parsear el lexema a double hubo que cambiar la "D" por la "E" que utiliza Java para los exponentes y poder verificar el rango, por lo que en la tabla de símbolos las constantes se verán con la "E" debido a que no queremos cambiar la D del lexema por la E cada vez que queramos parsear el lexema a double.
5. **VerificarComparador:** Identifica el carácter y dependiendo de este asigna el id de token que corresponde.
    a. En caso que se tenga un '=' en el lexema y venga otro '=' se entrega un token de '=' y no se consume el carácter de igual que hay sino que en el siguiente pedido a token se entrega otro token de igual y si viene uno de mayor o menor se entrega el de mayor/menor o igual según corresponda.
    b. En el caso que venga un ':' si ya se tiene otro carácter en el lexema entonces es porque ese carácter era un = y se entrega el token de '=:' sino solo se entrega ':'.
    c. En caso venga un carácter de '!' si se tiene otro carácter en el lexema es debido a que ese carácter era un '=' y se devuelve el token de '=!' sino se devuelve el token '!'.
    d. Si no se da ningún caso del primer switch se cae en la accion por defecto que consta de fijarse si lo que había era un mayor, menor o igual y setea los id según corresponda así como también no consume el carácter (esto es debido a que si viene un =, >, < no se puede saber si es todo el token o falta todavía que venga un !, :, = o <)
6. **ComentarioConsumirCaracter**: Setea el lexema del token en nulo dado que como estamos en un comentario no hay que retornar nada.
7. **VerificarCadenaCaracteres:** Verifica que la cadena no tenga un salto de línea y si lo tiene informa del error.

- Errores Léxicos considerados

- Constantes fuera de rango, ya sean dobles o enteros sin signo. Para el caso de los enteros estos van desde 0 a 255, por lo que si se pasan de dicho rango se pondrá dicho valor a 255 avisando con un warning. En el caso de los dobles estos consideran el rango 2.2250738585072014D-308 < x < 1.7976931348623157D+308 ∪ -1.7976931348623157D+308 < x < -2.2250738585072014D-308 ∪ 0.0, por lo que si el valor ingresado no cumple esta condición se pondrá en 0.0.
- Identificadores que superan la longitud máxima de 25 caracteres serán truncados a dicha longitud, avisando al programador con un warning.
- Cadenas con salto de línea, se avisará del error al programador y para que se siga la compilación se entregará un token de identificador pero sin lexema, se tomó esta decisión para que el programa pueda seguir compilando a pesar del error.

- Caracteres inválidos, por ejemplo “!” (como identificador y no siendo parte del operador de comparación “!=”), se informará del error y no se entrega ningún tipo de token.
- Detección de error cuando se declara un double, en la parte del exponente cuando se coloca una “D” no puede venir otra cosa que no sea un dígito o “+” o “-”, en estos casos se entrega el token de cte pero no su lexema.
- Similar al anterior caso, cuando se están definiendo identificadores (estado 1 en el autómata) si viene un carácter que no sea letra, número o “_” entonces se entrega el token de id pero sin lexema, para que se siga compilando.

- Analizador Sintáctico

En esta parte del trabajo se pidió adaptar el analizador léxico del que hablamos anteriormente para convertirlo en el método yylex(), el nombre de dicha función podrá variar de acuerdo a la herramienta que se utilice para generar el parser, en nuestro caso utilizamos Byacc para Java. Dicho método devolverá al Parser un token cada vez que este sea invocado o 0 si ya se procesó todo el archivo. Además aquellos identificadores, cadenas y constantes deberán entregar, además de su token, un lexema el cual será una referencia a la tabla de símbolos. Para dicha referencia se utiliza la variable yylval.

Una peculiaridad que tuvimos en el desarrollo (Bastante obvia una vez detectada) fue que al declarar algunas de las palabras reservadas en la gramática como lo son el return, for, if, continue tuvimos que declararlas con mayúsculas dado que java nos tiraba error que esas palabras no podían ser usadas como nombre de variable.

Luego se tuvo que tener en cuenta que aquellos tipos de datos que permitan valores negativos, en nuestro caso doubles (figura como “f64” en la tabla de palabras reservadas), deberá controlarse el valor ya que un valor aceptado para una constante por el analizador léxico, el cual desconoce su signo, podría estar fuera de rango si la constante llega a ser negativa. Por lo que la función verificarRangoDoubleNegativo() se encarga de dicha consigna. Cabe aclarar que la gramática ante errores encontrados debe poder continuar, para esto último a veces se utilizaron reglas de error con ejemplos erróneos y a veces usando el token “error” que nos brinda la herramienta. Ante un error encontrado, mediante acciones y métodos definidos, invocamos a la errorEnXY() e imprimimos dichos errores por pantalla, además llevamos un contador de errores mediante la variable yynerrs y otro de línea (desde el analizador léxico), para informar la cantidad de errores y en donde se encuentran estos y además al implementar el método yyerror() también llamamos al método errorEnXY() dado que su funcionalidad ya estaba implementada y nos dimos cuenta después que había que implementar el método sí o sí.

En las consultas con el profesor designado llegamos a la decisión de que la sentencia break y continue se pueden encontrar en una sentencia de selección (if) sin la necesidad de que este se encuentre en un for, el comportamiento de dichas instrucciones se definirá más adelante.

En el repositorio compartido se encuentran, además del código fuente, dos archivos de ejemplos para probar el compilador, uno sin errores y otro repleto de estos básicamente para que se vea que en cualquier caso el compilador llega al final del programa.

Por último, al momento de ejecutar el programa, al ver los mensajes de informacion, errores y warnings detectados se utilizaron los códigos de escape ANSI para dar color a la salida de la terminal de texto, diferenciando entre cada impresión e imitar a los tipicos compiladores que resaltan los errores en rojo y los warnings en amarillo, por ejemplo. Realizamos esta aclaración ya que si se ejecuta el programa en ciertas terminales que no soporten ANSI como lo es el powershell o cmd de windows, entonces a la hora de mostrar los mensajes se verán ciertos mensajes con las codificaciones de los colores al principio y final de cada mensaje, en lugar de cambiar el color de la salida, testeado está en terminales de visual studio code y en la de eclipse a partir de una versión de 2022 ya viene por defecto soportado y sino queda instalar un plugin que lo permita.

-Errores Sintácticos Considerados:

1. Que el nombre del programa sea una constante o una cadena de caracteres empezando con comillas simples.
2. Que a los bloques de sentencias les falte o bien la llave de comienzo o la de final.
3. Que en la declaración de variables falte el tipo de la variable.
4. Que el tipo de las variables declaradas sea un id y no una palabra reservada
5. Que en la declaración de variables no se respete el formato o bien de “tipo” “id” “;” o “tipo” “id” “,” “id” “;” y así para las n repeticiones dada su definición recursiva.
6. Que en la declaración de la función falte el nombre de la misma o que falten las llaves, paréntesis o dos puntos.
7. Que en un parámetro falte el identificador del mismo.
8. Que en el return de una función falten los paréntesis, o los paréntesis de inicio/final o punto y coma.
9. Que en la sentencia de asignación no haya una expresión a la que asignar, o que el operador ”=:” haya sido escrito al revés “:=”.
10. Que en la sentencia de impresión “out” falte algún paréntesis de inicio/final.
11. Que al invocar funciones sin utilizar el valor de retorno o sea como si fuera una sentencia ejecutable pero sin el discard.
12. Que en la declaración de una instrucción For los id que se usan sean diferentes para la condición.
13. Que el valor de una constante al tener un “-” adelante se corresponda con los limites validos para los dobles si es que es un doble.

-Conclusión:

A la hora de desarrollar estos trabajos prácticos pudimos comprender mejor el funcionamiento de un compilador, principalmente en sus primeras etapas: analizador léxico y analizador sintáctico. Lo que en

cierta forma nos ayuda a entender el porqué cuando un programa tira un error o no compila lo complicado que puede ser en ciertos casos ser específico en la causa o en la posición del mismo.

La decisión de utilizar archivos para cargar las matrices es debido a que sopesamos la posibilidad de que nuestro compilador pueda ser cambiado sin necesidad de tener el código fuente VS la posibilidad de permitir que por estar hardcodeado (cargar las tablas a una variable en el código directamente en vez de cargarlas de un archivo) el compilador sea más fácil de usar sin necesidad de pegar tb la carpeta con las tablas requeridas,nos decantamos sobre la versatilidad de modificarlo sin compilarlo, pese a sus desventajas en la facilidad de uso del mismo.

Para las pruebas tenemos 2 archivos prueba.txt y test.txt en la carpeta testFiles donde prueba .txt tiene todos los errores que pudimos poner y que permite que siga compilando el programa y el test.txt tiene un programa correcto donde se muestra el mensaje de compilación exitoso.