TRABAJO PRÁCTICO COMPILADOR

CONSIDERACIONES GENERALES

Es necesario cumplir con las siguientes consideraciones para evaluar el TP.

1. Cada grupo deberá desarrollar el compilador teniendo en cuenta:
   - Todos los temas comunes (Ver ANEXO TEMAS)
   - El tema especial asignado al grupo.
2. Se fijarán puntos de control con fechas y consignas determinadas

PRIMERA ENTREGA

OBJETIVO: Realizar un **analizador lexicográfico** utilizando la herramienta JFLEX. La aplicación realizada debe mostrar una interfaz gráfica que pueda utilizarse como IDE del compilador, en la cual se debe poder:

1. Ingresar código de nuestro programa manualmente en un cuadro de texto adecuado a tal propósito, por ejemplo dentro del lenguaje Java JTextArea.
2. Cargar un archivo con código y poder editarlo dentro del cuadro de texto.
3. Compilar el programa ingresado (análisis léxico) y mostrar un texto aclaratorio, identificando los tokens reconocidos por el parser u errores encontrados en el análisis. Las impresiones deben ser claras. Los elementos que no generan tokens no deben generar salida.

El material a entregar será:

- El archivo jflex que se llamará **Lexico.flex**
- Un archivo de pruebas generales que se llamará **prueba.txt** incluyendo la prueba del tema asignado al grupo
- Un archivo con la tabla de símbolos **ts.txt**
- Código fuente del proyecto
- Archivo JAR ejecutable

Todo el material deberá ser subido a algún repositorio GIT (Github, Gitlab, etc.) y su enlace enviado a teoria1.unlu@gmail.com
Asunto: GrupoXX
Fecha de entrega: 26/10/21

SEGUNDA ENTREGA

OBJETIVO: Realizar un **analizador sintáctico** utilizando la herramienta JAVA CUP. La aplicación realizada debe mostrar una interfaz gráfica que pueda utilizarse como IDE del compilador, en este caso deberá permitir:

1. Cumplir mismos puntos que en la primer entrega
2. Compilar el programa ingresado (análisis sintáctico) y mostrar por pantalla un texto aclaratorio identificando las reglas sintácticas que va analizando el parser. Las impresiones deben ser claras. Las reglas que no realizan ninguna acción no deben generar salida.

El material a entregar será:

- El archivo jflex que se llamará **Lexico.flex**
- El archivo jcup que se llamará **Sintactico.cup**
- Un archivo de pruebas generales que se llamará **prueba.txt** incluyendo la prueba del tema asignado al grupo
- Un archivo con la tabla de símbolos **ts.txt**
- Código fuente del proyecto
- Archivo JAR ejecutable

Todo el material deberá ser subido a algún repositorio GIT (Github, Gitlab, etc.) y su enlace enviado a teoria1.unlu@gmail.com
Asunto: GrupoXX
Fecha de entrega: 09/11/21

ANEXO TEMAS

TEMAS COMUNES

WHILE
Implementación de *While* utilizando el formato que el grupo desee

DECISIONES
Implementación de *IF* utilizando el formato que el grupo desee

ASIGNACIONES
Asignaciones simples *a ::= b*

TIPO DE DATOS
Constantes numéricas

- enteras (16 bits)
- reales (32 bits)

El separador decimal será el punto “.”

Ejemplo:

```
a ::= 99999.99
a ::= 99.
a ::= .9999
```

Constantes string
Constantes de 30 caracteres alfanuméricos como máximo, limitada por comillas (" ") ,de la forma *"XXXX"*

Ejemplo:

```
WRITE "@sdADaSjfla%dfg"
var ::= "HOLA MUNDO"
```

Las constantes deben ser reconocidas y validadas en el *analizador léxico*, de acuerdo a su tipo.
Las constantes guardan su valor en tabla de símbolos.

VARIABLES

Variables numéricas
Estas variables reciben valores numéricos tales como constantes numéricas, variables numéricas u operaciones que arrojen un valor numérico, del lado derecho de una asignación.

Variables string
Estas variables pueden recibir una constante string

Las variables no guardan su valor en tabla de símbolos.
Las asignaciones deben ser permitidas, solo en los casos en los que los tipos son compatibles, caso contrario deberá desplegarse un error.

COMENTARIOS

Deberán estar delimitados por " //* " y " *// " y podrán estar anidados en un solo nivel.

Ejemplo1:

```
//*
IF (a <= 30)
      b = ”correcto” //* asignación string *//
ENDIF
*//
```

Ejemplo2:

```
//* Así son los comentarios *//
```

Los comentarios se ignoran de manera que no generan un componente léxico o token.

SALIDA

Las salidas se implementarán como se muestra en el siguiente ejemplo:

Ejemplo:

```
write “ewr”     //* donde “ewr” debe ser una cte string *//
write 99.999 //* donde 99.999 debe ser cualquier cte numérica *//
write var    //* donde var debe ser cualquier variable numérica *//
```

CONDICIONES

Las condiciones para un constructor de ciclos o de selección, deberán ser comparaciones binarias que pueden estar ligadas por un único conector lógico.

```
(expresión) < (expresión)
(expresión >= expresión) && (expresión < expresión)
```

DECLARACIONES

Todas las variables deberán ser declaradas dentro de un bloque especial para ese fin, delimitado por las palabras reservadas DECLARE.SECTION y ENDDECLARE.SECTION, siguiendo el siguiente formato:

```
DECLARE.SECTION
            Línea_de_Declaración_de_Tipos
ENDDECLARE.SECTION
```

Cada *Línea_de_Declaración_de_Tipos* tendrá la forma: *[Lista de Variables] := [Lista de Tipos]*

La *Lista de Variables* debe ser una lista de variables entre corchetes separadas por comas al igual que la Lista de Tipos. Cada variable se deberá corresponder con cada tipo en la lista de tipos según su posición. No deberán existir más variables que tipos, ni más tipos que variables.
Pueden existir varias líneas de declaración de tipos.

Ejemplos de formato:

```
DECLARE.SECTION
      [a1, b1] := [FLOAT, INT]
      [p1, p2, p3] := [FLOAT,FLOAT,INT]
ENDDECLARE.SECTION
```

PROGRAMA

Todas las sentencias del programa deberán ser declaradas dentro de un bloque especial para ese fin, delimitado por las palabras reservadas PROGRAM.SECTION y ENDPROGRAM.SECTION, siguiendo el siguiente formato:

```
PROGRAM.SECTION
      Lista_de_Sentencias
ENDPROGRAM.SECTION
```

La zona de declaración de variables deberá ser previa a la sección del programa.

TABLA DE SIMBOLOS

La tabla de símbolos tiene la capacidad de guardar las variables y constantes con sus atributos.
Los atributos aportan información necesaria para operar con constantes y variables.

Ejemplo 1er ENTREGA (sin agregar tipos de datos)

NOMBRE	TOKEN	TIPO	VALOR	LONG
a1	ID		—	—
b1	ID		—	—
_hola	CTE_STR	—	hola	4
_mundo	CTE_STR	—	mundo	5
_30.5	CTE_F	—	30.5	—
_55	CTE_E	—	55	—

Tabla de símbolos

Ejemplo 2da ENTREGA (agregando tipos de datos)

NOMBRE	TOKEN	TIPO	VALOR	LONG
a1	ID	Float	—	—
b1	ID	Integer	—	—
_hola	CTE_STR	—	hola	4
_mundo	CTE_STR	—	mundo	5
_30.5	CTE_F	—	30.5	—
_55	CTE_E	—	55	—

Tabla de símbolos

TEMAS ESPECIALES

- **Grupo 1 ASIGNACIÓN TAKE**
  (Lopez de Calle, Normand, Sala, Scarnatto, Soto)

Calcula el valor de la función TAKE y se lo asigna a un identificador.
Esta función toma como entrada un operador de suma, una variable entera positiva y una lista de constantes (todos los números del 1 al 20). Esta función devuelve el valor que resulta de aplicar el operador suma a los primeros "n" elementos de la lista. El valor de n quedará establecido en la componente id. Los elementos de la lista están separados por comas (,).
La lista de constantes podría ser vacía en cuyo caso se emitirá un mensaje "La lista está vacía"
El resultado será asignado al ID del lado izquierdo de la asignación

Formato:
```
id ::= TAKE(+; id; [lista de constantes])
```

Ejemplos:
```
resul::= TAKE(+; id; [3]) //* si id tomase el valor 1 devuelve 3 y se asigna al id resul, si id resulta
                            mayor a 1 devuelve 3 y se asigna al id resul *//
resul::= TAKE(+; id; [1,2,3,4,5]) //* si id tomase el valor 3 devuelve 6, resultado de sumar las 3
                                    primeras constantes de la lista y se asigna al id resul *//
resul::= TAKE (+; id; []) //* mensaje la lista esta vacía y se asigna 0 al id resul*//
```

- **Grupo 2 COLA**
  (Cisneros, Claros Garcia, Obregon, Scafati, Torres)

La sentencia permite sumar las últimas posiciones de una lista de constantes determinadas por un elemento pivot y se debe asignar el resultado a un identificador.
El elemento pivot deberá ser mayor o igual a uno. Se deberá verificar que el pivot no sea mayor a la longitud de la lista. Si este fuera el caso se deberá emitir un mensaje "La lista no tiene suficientes elementos"
La lista de constantes podría ser vacía en cuyo caso se emitirá un mensaje "La lista está vacía"

Formato:
```
id ::= COLA(constante_entera; [lista de constantes])
```

Ejemplos:
```
resul::= COLA(4;[10,20,30,40,5,4])  //* La suma es: 79 que será asignado al id resul *//
resul::= COLA(5;[2,2,2,4])   //* La lista tiene menos elementos que el indicado *//
resul::= COLA(1;[2,1,1,4]) //* La suma es: 4 que será asignado al id resul *//
resul::= COLA(2;[10,20,30,40,50,40])  //* La suma es: 90 que será asignado al id resul*//
resul::= COLA(1;[])  //* mensaje la lista esta vacía y se asigna 0 al id resul*//
```

- **Grupo 3 POSITION**
  (Iarza, Laffont, Nocella, Schuckmann, Vazquez)

La sentencia permite encontrar en que posición de una lista de constantes enteras positivas se encuentra un elemento pivot. El elemento pivot deberá ser mayor o igual a uno. Si el elemento pivot estuviera varias veces en la lista, el resultado de la sentencia es la primera posición en la que aparece.
En caso de que no sea encontrado se emitirá el mensaje "Elemento no encontrado"
La lista de constantes podría ser vacía en cuyo caso se emitirá un mensaje "La lista está vacía"

Formato:
```
id ::= POSITION(constante_entera; [lista de constantes])
```

Ejemplos:

```
resul::= POSITION(4;[10,20,30,40,5,4]) //* Elemento encontrado en posición 6, valor que será asignado al id resul *//
resul::= POSITION(5;[2,2,2,4]) //* mensaje elemento no encontrado y se asigna 0 la id resul *//
resul::= POSITION(1;[2,1,1,4])  //* Elemento encontrado en posición 2, valor que será asignado al id resul *//
resul::= POSITION(1;[]) //* mensaje la lista esta vacía y se asigna 0 al id resul*//
```

- **Grupo 4 PROMR**
  (Benitez, Cantou, Medina, Viani)

La sentencia permite calcular el promedio de los valores de una lista de constantes que cumplen con una condición específica. La condición se especificará con un elemento pivot y se verificará por menor o igual.
El elemento pivot deberá ser mayor o igual a uno.
La lista de constantes podría ser vacía en cuyo caso se emitirá un mensaje “La lista está vacía”
Si todos los elementos de la lista resultasen mayores al elemento pivot, se emitirá un mensaje “No existen elementos para el cálculo”

Formato:

```
id ::= PROMR(id; <=; [lista de constantes])
```

Ejemplo:

```
resul:: PROMR(id; <=; [10,35,4,5]) //* Si id toma valor 5 el promedio es: 4.5 que será asignado al id resul *//
resul::= PROMR(id; <=; [2,2,2,4,10])  //* Si id toma valor 12 el promedio es: 4 que será asignado al id resul *//
resul::= PROMR(id; <=; []) //* mensaje la lista esta vacía y se asigna 0 al id resul*//
resul::= PROMR(id; <=; [20,20,7,40])  //* Si id toma valor 3 mensaje no existen elementos para el cálculo y se asigna 0 al id resul*//
```

- **Grupo 5 SUMAIMPAR**
  (Barragan, Bert, Lopez, Sciarrotta)

La sentencia permite sumar los primeros elementos impares dentro de una lista de constantes enteras positivas. La cantidad de elementos impares a sumar estará determinada por un elemento pivot.
El elemento pivot deberá ser mayor o igual a uno.
En caso de que no se encuentre un elemento impar en la lista, o de que haya menos elementos impares en la lista que los indicados por el pivot, se emitirá el mensaje “No puede realizarse la operación”
La lista de constantes podría ser vacía en cuyo caso se emitirá un mensaje “La lista está vacía”

Formato:

```
id ::= SUMAIMPAR(constante_entera; [lista de constantes])
```

Ejemplo:

```
resul::= SUMAIMPAR(4;[1,2,3,4,5]) //* mensaje no puede realizarse la operación y se asigna 0 la id resul *//
resul::=  SUMAIMPAR(5;[2,2,2,4])  //* mensaje no puede realizarse la operación y se asigna 0 la id resul *//
resul::= SUMAIMPAR(1;[])  //* mensaje la lista esta vacía y se asigna 0 al id resul*//
resul::= SUMAIMPAR(3;[2,21,7,44,40,33,5]) //* La suma de los elementos impares es: 61 que sera asignado al id resul *//
```