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 las herramientas de análisis léxico propuestas por la cátedra. La aplicación realizada debe mostrar una interfaz gráfica que pueda utilizarse como interfaz del compilador, en la cual se debe poder ingresar código escrito en el lenguaje a compilar, cargar un archivo de código y editarlo, compilar el contenido del programa ingresado y mostrar por pantalla los errores y mensajes

El material a entregar será:

- El proyecto utilizando la herramienta seleccionada para la implementación del analizador léxico.
- Un archivo de pruebas generales que se llamará **prueba.txt** que incluya pruebas exhaustivas de los temas comunes y los temas especiales asignados a cada grupo.
- Un archivo ejecutable que se deberá llamar **Compilador** (por ejemplo Compilador.jar que permita ejecutar la aplicación mediante java runtime enviroment, o el archivo .py de punto de entrada que permita ejecutarlo mediante un intérprete de Python).

La entrega deberá subirse a las tareas que los docentes habiliten para tal fin en la plataforma ED.

Fecha de entrega: 29/03/2022

SEGUNDA ENTREGA

OBJETIVO: Realizar un analizador lexicográfico y un analizador sintáctico utilizando las herramientas propuestas por la cátedra las cuales deberán operar en conjunto. La aplicación realizada debe mostrar una interfaz gráfica que pueda utilizarse como interfaz del compilador, en la cual se debe poder ingresar código escrito en el lenguaje a compilar, cargar un archivo de código y editarlo, compilar el contenido del programa ingresado y mostrar por pantalla un texto aclaratorio identificando las reglas sintácticas que va analizando el parser. Las impresiones deben ser claras.

El material a entregar será:

- El proyecto utilizando las herramientas seleccionadas para la implementación del analizador léxico y para el analizador sintáctico.
- Un archivo de pruebas generales que se llamará **prueba.txt** que incluya pruebas exhaustivas de los temas comunes y los temas especiales asignados a cada grupo.
- El proyecto deberá generar un archivo con la tabla de símbolos **ts.txt** donde figuren los nombres de todos los identificadores detectados junto con los tipos correspondientes.

- Un archivo ejecutable que se deberá llamar **Compilador** (por ejemplo Compilador.jar que permita ejecutar la aplicación mediante java runtime enviroment, o el archivo .py de punto de entrada que permita ejecutarlo mediante un intérprete de Python).

La entrega deberá subirse a las tareas que los docentes habiliten para tal fin en la plataforma ED.

Fecha de entrega: 19/04/2022

TERCERA ENTREGA

OBJETIVO: Realizar un generador de código intermedio utilizando el archivo obtenido en la segunda entrega. El programa ejecutable deberá procesar el archivo de entrada (prueba.txt) y devolver el árbol sintáctico del mismo, junto con la tabla de símbolos. La interfaz gráfica realizada debe contener un menú con pestañas que permitan acceder a la tabla de símbolos y árbol sintáctico.

El material a entregar será:

- El proyecto utilizando las herramientas seleccionadas para la implementación del analizador léxico y para el analizador sintáctico.
- Un archivo de pruebas generales que se llamará **prueba.txt** que incluya pruebas exhaustivas de los temas comunes y los temas especiales asignados a cada grupo.
- El proyecto deberá generar un archivo con la tabla de símbolos **ts.txt** donde figuren los nombres de todos los identificadores detectados junto con los tipos correspondientes.
- El compilador deberá generar un archivo PNG con la estructura del árbol sintáctico y un archivo .DOT con la definición en lenguaje dot de graphviz del árbol generado, ambos generados a partir del código en el archivo **prueba.txt.**
- El compilador deberá verificar que los nombres de variables utilizados hayan sido declarados previamente.
- Un archivo ejecutable que se deberá llamar **Compilador** (por ejemplo Compilador.jar que permita ejecutar la aplicación mediante java runtime enviroment, o el archivo .py de punto de entrada que permita ejecutarlo mediante un intérprete de Python).

La entrega deberá subirse a las tareas que los docentes habiliten para tal fin en la plataforma ED.

Fecha de entrega: 10/05/2022

CUARTA ENTREGA

OBJETIVO: Realizar un compilador generando código IR de LLVM a partir del árbol sintáctico obtenido en la tercera entrega. Mediante la interfaz se deberá poder cargar el archivo de entrada (prueba.txt), modificarlo si resultara necesario, compilarlo y ejecutarlo mostrando por pantalla el resultado de la ejecución. Se debe incorporar un último ítem al menú que muestre el código IR generado por LLVM. Se deberá además realizar los chequeos de tipos y conversiones implícitas correspondientes.

El material a entregar será:

- El proyecto utilizando las herramientas seleccionadas para la implementación del analizador léxico y para el analizador sintáctico.
- Un archivo de pruebas generales que se llamará **prueba.txt** que incluya pruebas exhaustivas de los temas comunes y los temas especiales asignados a cada grupo.

- El proyecto deberá generar un archivo con la tabla de símbolos **ts.txt** donde figuren los nombres de todos los identificadores detectados junto con los tipos correspondientes.
- El compilador deberá realizar el chequeo de tipos y conversiones implícitas correspondientes.
- El compilador deberá generar un archivo PNG con la estructura del árbol sintáctico y un archivo .DOT con la definición en lenguaje dot de graphviz del árbol generado, ambos generados generado a partir del código en el archivo **prueba.txt**
- El compilador deberá generar un archivo LL con el código fuente legible en el lenguaje IR de LLVM y un archivo ejecutable correspondiente al código fuente anterior.
- Un archivo ejecutable que se deberá llamar **Compilador** (por ejemplo Compilador.jar que permita ejecutar la aplicación mediante java runtime enviroment, o el archivo .py de punto de entrada que permita ejecutarlo mediante un intérprete de Python).

La entrega deberá subirse a las tareas que los docentes habiliten para tal fin en la plataforma ED.

Fecha de entrega: 07/06/2022

ANEXO TEMAS

TEMAS COMUNES

OPERADORES ARITMÉTICOS

Se deberán admitir las operaciones de suma (a + b), resta (a - b), multiplicación (a * b) y división (a / b) junto con la de valor opuesto (-a). Se podrán agregar otras operaciones adicionales que se consideren apropiadas. Se cumplirá con la precedencia y asociatividad usual de las operaciones matemáticas, pudiendo igualmente usarse los paréntesis para modificarla.

OPERADORES DE COMPARACIÓN

Se podrán comparar dos expresiones por igualdad (a == b), desigualdad (a != b), mayor (a > b), mayor o igual (a >= b), menor (a < b) o menor o igual (a <= b). Todos los operadores de comparación poseen la misma precedencia y no son asociativos.

OPERADORES LÓGICOS

Se deberán admitir las operaciones de conjunción (a and b), disyunción (a or b) y negación (not a). Se cumplirá con la precedencia y asociatividad usual de dichos operadores (negación, conjunción, disyunción), pudiendo usarse los paréntesis para modificarla.

EXPRESIONES

Una expresión puede combinar operaciones aritméticas entre constante y variables numéricas o uso a funciones que retornan un valor numérico compatible.

Una expresión también puede incluir operadores lógicos y relacionales aplicados a constantes, variables o usos de funciones que retornan un valor apropiado para ese contexto.

La precedencia y asociatividad de los operadores se indica en la siguiente tabla:

Operador	Asociatividad
or	Izquierda
and	Izquierda
not	Izquierda
== != > >= < <=	No es posible asociar
+ -	Izquierda
*** /**	Izquierda
- (unario)	Izquierda

SENTENCIAS DE ITERACIÓN

Los ciclos WHILE estarán compuestos por la palabra reservada WHILE seguidos de una expresión lógica, luego de la palabra reservada DO y finalmente un conjunto de sentencias y la palabra reservada END.

SENTENCIAS DE SELECCIÓN

Las instrucciones de selección estarán compuestas por la palabra reservada IF seguidas de una expresión lógica, luego de la palabra reservada THEN a continuación un conjunto de sentencias y finalmente la palabra reservada END. También podrán estar compuestas por la palabra reservada IF seguidas de una expresión lógica, luego de la palabra reservada THEN a continuación un conjunto de sentencias, la palabra reservada ELSE seguida de otro conjunto de sentencias y finalmente la palabra reservada END

TIPO DE DATOS

Se soportan los tipos booleano (boolean), número entero (integer) y número representado en punto flotante (float).

Para las constantes booleanas literales se utilizan los valores **true** y **false** para representar verdadero y falso respectivamente.

Las constantes numéricas literales pueden definirse con o sin parte decimal. Si tienen parte decimal, se asumen de punto flotante. En caso contrario, se asumen enteras. Para las constantes de valores reales deberá utilizarse obligatoriamente el punto (.) como separador decimal. Algunos ejemplos de constantes de valores en punto flotante:

99999.99 99. .9999

Las constantes de cadenas de caracteres se representan como un texto delimitado por comillas dobles (“”), de la forma "xxx xxx xxxx". Se acepta que dentro de la cadena puedan escribirse comillas dobles siempre que se las anteponga con una barra invertida (“\””). De forma similar, la combinación “\n” se interpretará como un carácter de nueva línea y la combinación “\t”, como carácter de tabulación. La barra invertida, por usarse como carácter de escape dentro de la cadena, deberá siempre ser escrita ante poniéndole otra barra invertida (quedando finalmente como “\\”) para que sea interpretada como un único carácter “\”. Algunos ejemplos de cadenas literales:

"Esta es una cadena de caracteres."

"Primera línea\nSegunda \"línea\"\nLa \\última/ línea"

Las constantes deben ser reconocidas y validadas en el analizador léxico, de acuerdo a su tipo.

VARIABLES

Los nombres de las variables definidas en este lenguaje deberán cumplir la restricción de comenzar con una letra, siguiendo con cero o más letras, dígitos o guiones bajos (‘_’). Las letras soportadas serán aquellas que el estándar Unicode considera como tal, pudiendo ser tanto mayúsculas como minúsculas. Se distinguen las diferencias entre mayúsculas y minúsculas. Ejemplos:

i monto_total aux_1 año función

Las variables no guardan su valor en la 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. Las variables y constantes integer y float deben ser interoperables entre sí y el resultado de la operación siempre deberá ser un float, mientras que las variables bool serán incompatibles con el resto.

ASIGNACIONES

Las asignaciones en este lenguaje serán simples, de la forma A:=B, siendo A una variable previamente declarada y B una expresión aritmético-lógica del mismo tipo o de un tipo compatible. Para determinar qué

asignaciones deberán permitirse en el lenguaje, según los tipos de A y B, deberá tenerse en cuenta la siguiente tabla:

A ↓ B →	integer	float	bool
integer	permitido	no permitido	no permitido
float	permitido	permitido	no permitido
bool	no permitido	no permitido	permitido

COMENTARIOS

Los comentarios multilínea deberán estar delimitados tanto por los caracteres “(*” y “*)”, permitiendo y controlando posibles niveles de anidamiento, asegurando que cada apertura se corresponda con su cierre correspondiente.

También es posible definir un comentario con “#”, indicando que el comentario termina al final de la línea. Ejemplos:

..... # .....#.....#..... (válido)

..... (* .....*) ..... (válido)

..... (* .....(*.....*)..... *) ..... (válido)

..... (* .....*).....*) ..... (inválido, no están balanceados)

ENTRADA Y SALIDA

Las salidas se implementarán mediante la instrucción DISPLAY. Esta deberá permitir imprimir expresiones (enteras, booleanas o float) y constantes string. Como se muestra en los siguientes ejemplos:

DISPLAY(“ewr”) (* donde “ewr” debe ser una cte literal string *)

DISPLAY(5.5 + aux)

DISPLAY(7 <= mi_var)

Por otro lado, la lectura de valores por consola se realizará mediante las instrucciones **INPUT_INT()** **INPUT_FLOAT()** e **INPUT_BOOL()**.

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: Tipo : Lista de Variables ;

La Lista de Variables debe ser una lista de variables separadas por comas. Pueden existir varias líneas de declaración de tipos.

Ejemplos de formato:

```
DECLARE.SECTION
        INT : a, b , c;
```

FLOAT : d, e;
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

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

TABLA DE SÍMBOLOS

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

Ejemplo 2da ENTREGA (agregando tipos de datos)

NOMBRE	TOKEN	TIPO	VALOR	LONG
a1	ID	Float	_	_
b1	ID	Integer	_	_
	CTE_STR	_	hola	4
	CTE_STR	_	mundo	5

TEMAS ESPECIALES

COLA

La sentencia permite sumar las últimas posiciones de una lista de expresiones enteras determinadas por un elemento pivot.

La lista de expresiones puede ser cambiada alterando los valores y su cantidad en el programa test.txt.

El elemento pivot deberá ser mayor o igual a uno y será un valor constante o una variable siempre entera, si no cumpliese con la validación deberá mostrar un mensaje “El valor debe ser >=1”.

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 tiene menos elementos que el indicado”.

La lista de expresiones podría ser vacía en cuyo caso se emitirá un mensaje “La lista está vacía.”,

En los casos en los que no pueda efectuarse la suma el valor retornado por la función será cero.

Por ejemplo:

resul=cola (4;[10,20,30,40,5,4]) La suma es: 79

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

resul=cola (2;[10,20,30,40,50,40]) La suma es: 90

resul =cola (1;[]) La lista está vacía.

POSICION

La sentencia permite encontrar en qué posición de una lista de expresiones enteras se encuentra un elemento pivot.

La lista de expresiones puede ser cambiada alterando los valores y su cantidad en el programa *test.txt*.

El elemento pivot deberá ser mayor o igual a uno y será un valor constante o una variable siempre entera, si no cumpliese con la validación deberá mostrar un mensaje “El valor debe ser >=1”

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 expresiones podría ser vacía en cuyo caso se emitirá un mensaje “La lista está vacía”.

En los casos en los que no pueda efectuarse la operación el valor retornado por la función será cero.

Por ejemplo:

resul =posicion (4;[10,20,30,40,5,4]) Elemento Encontrado en posición 6

resul =posicion (5;[2,2,2,4]) Elemento no encontrado

resul =posicion (51;[2,2,2,4]) Elemento no encontrado

resul =posicion (1;[2,1,1,4]) Elemento Encontrado en posición 2

resul =posicion (1;[]) La lista está vacía

SUMAIMPAR

La sentencia permite sumar los primeros elementos impares dentro de una lista de expresiones.

La cantidad de elementos impares a sumar estará determinada por un elemento pivot.

La lista de expresiones puede ser cambiada alterando los valores y su cantidad en el programa *test.txt*.

El elemento pivot deberá ser mayor o igual a uno y será un valor constante o una variable siempre entera, si no cumpliese con la validación deberá mostrar un mensaje “El valor debe ser >=1”

En caso de que no se encuentre un elemento impar en la lista, se emitirá el mensaje “Elementos impares no encontrados”

En caso de que haya menos elementos impares en la lista que los indicados por el pivot, se emitirá el mensaje “No existen suficientes elementos impares para el cálculo”.

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 tiene menos elementos que el indicado”

La lista de expresiones podría ser vacía en cuyo caso se emitirá un mensaje “La lista está vacía”.

En los casos en los que no pueda efectuarse la suma el valor retornado por la función será cero.

Por ejemplo:

resul =sumaimpar (4;[1,2,3,4,5]) No existen suficientes elementos impares para el cálculo

resul =sumaimpar (5;[2,2,2,4]) La lista tiene menos elementos que el indicado.

resul =sumaimpar (1;[]) La lista está vacía

resul =sumaimpar (3;[2,21,7,44,40,33,5]) La suma de los elementos impares es: 61

BETWEEN

Tomará como entrada una variable numérica y dos expresiones numéricas encerradas entre corchetes y separadas por punto y coma. Devolverá verdadero o falso según la variable enunciada se encuentre dentro del rango definido por ambas constantes.

Esta función será utilizada en las condiciones, presentes en ciclos y selecciones. Ejemplo:

BETWEEN(variable, [constante1; constante2])

BETWEEN (a, [2 ; 12]) /* Verdadero si 2<=a<=12 Falso en caso contrario */

BETWEEN (z, [2.3 ; 11.22]) /* Verdadero si 2.3<=z<=11.22 Falso en caso contrario */

```
DECLARE.SECTION
        a1 : INTEGER
        d1 : FLOAT
ENDDECLARE.SECTION

PROGRAM.SECTION
        a1:=12
        d1:=3,14
        IF BETWEEN(a1,[10;20]) THEN
        d1 := 3,14
        END IF
ENDPROGRAM.SECTION
```