Ejercicio 2 optimización:

1) En este compilador se puede aplicar la optimización por redundancia, como se puede ver en la imagen, la función “maxima” se utiliza 2 veces con los mismos datos, esto se lo puede optimizar haciendo/ejecutando la función “maxima” una sola vez, esto se logra reemplazando “maxima (3, 9)” por la variable “max” dado que en el medio la variable no cambia de valor o se puede aplicar mismo pero en vez de usar la variable “max” usaremos una variable interna “@max” que tendrá el número máximo, esto es más conveniente debido a que nos desvinculamos de la variable “max” si es que en el medio cambia de valor o no.

SIN OPTIMIZACIÓN

```
max = maximo (3, 9)
resul = maximo (3, 9) / X
```

CON OPTIMIZACIÓN

```
max = maximo (3, 9)
resul = max / X
```

En cuanto a los tercetos se eliminaría todos los tercetos relacionados con la segunda llamada de la función maxima que son del 6 hasta 10 para este caso en particular, como se puede ver en la imagen del terceto optimizado, se usó una variable interna “@max” y la optimización se hizo una vez generado el terceto sin optimización, en la representación intermedia.

SIN OPTIMIZACIÓN

```
[0]-->(=, _3, @max)
[1]-->(=, _9, @aux)
[2]-->(CMP, @aux, @max)
[3]-->(JNB, [5],  )
[4]-->(=, @aux, @max)
[5]-->(=, @max, max)
[6]-->(=, _3, @max)
[7]-->(=, _9, @aux)
[8]-->(CMP, @aux, @max)
[9]-->(JNB, [11],  )
[10]-->(=, @aux, @max)
[11]-->(/, @max, X)
[12]-->(=, [11], resul)
```

CON OPTIMIZACIÓN

```
[0]-->(=, _3, @max)
[1]-->(=, _9, @aux)
[2]-->(CMP, @aux, @max)
[3]-->(JNB, [5],  )
[4]-->(=, @aux, @max)
[5]-->(=, @max, max)
[11]-->(/, @max, X)
[12]-->(=, [11], resul)
```

2) En este compilador también se puede aplicar otra optimización por redundancia, aplicado a la lista se le pasa a la función “maxima”, esta optimización es aplicable cuando hay números repetidos en la lista, no es necesario que haga la comparación de un número que se ya se hizo previamente. Lo que se hará es, de alguna manera, eliminar los numero repetidos para poder obtener la optimización

SIN OPTIMIZACIÓN

```
max = maximo (3, 3, 2, 2, 1, 1)
```

CON OPTIMIZACIÓN

```
max = maximo (3, 2, 1)
```

En cuanto a los tercetos se eliminará todos los tercetos relacionados con los números repetidos y sus comparaciones, un ejemplo de esto es el terceto “[1]”, que esta repetido, se elimina los tercetos del 1 hasta 4. Además, hay que tener en cuenta si algún terceto del conjunto optimizado hace referencia a un terceto que se eliminó, un ejemplo de esto es el terceto “[7]” que hacía referencia el terceto “[9]”, debido a que el terceto “[9]” se eliminó, su referencia pase a hacer el terceto “[13]”. La optimización se hizo una vez generado el terceto sin optimización, en la representación intermedia.

SIN OPTIMIZACIÓN

```
[0]-->(=, _3, @max)
[1]-->(=, _3, @aux)
[2]-->(CMP, @aux, @max)
[3]-->(JNB, [5],  )
[4]-->(=, @aux, @max)
[5]-->(=, _2, @aux)
[6]-->(CMP, @aux, @max)
[7]-->(JNB, [9],  )
[8]-->(=, @aux, @max)
[9]-->(=, _2, @aux)
[10]-->(CMP, @aux, @max)
[11]-->(JNB, [13],  )
[12]-->(=, @aux, @max)
[13]-->(=, _1, @aux)
[14]-->(CMP, @aux, @max)
[15]-->(JNB, [17],  )
[16]-->(=, @aux, @max)
[17]-->(=, _1, @aux)
[18]-->(CMP, @aux, @max)
[19]-->(JNB, [21],  )
[20]-->(=, @aux, @max)
[21]-->(=, @max, max)
```

CON OPTIMIZACIÓN

```
[0]-->(=, _3, @max)
[5]-->(=, _2, @aux)
[6]-->(CMP, @aux, @max)
[7]-->(JNB, [13],  )
[8]-->(=, @aux, @max)
[13]-->(=, _1, @aux)
[14]-->(CMP, @aux, @max)
[15]-->(JNB, [21],  )
[16]-->(=, @aux, @max)
[21]-->(=, @max, max)
```

3) En este compilador se puede aplicar la optimización por código muerto, en el cual la primera sentencia “max = maximo (3, 1)” se la puede inferir o descartar debido a que en la siguiente sentencia hay una asignación de la misma variable.

SIN OPTIMIZACIÓN

```
max = maximo (3, 1)
max = maximo (2, 6)
```

CON OPTIMIZACIÓN

```
max = maximo (2, 6)
```

En cuanto a los tercetos lo que se hará es eliminar todos los tercetos que estén implicados con la primera sentencia “max = maximo (3, 1)” para este caso los tercetos que se eliminaran son los tercetos del 0 hasta 5.

SIN OPTIMIZACIÓN

```
[0]-->(=, _3, @max)
[1]-->(=, _1, @aux)
[2]-->(CMP, @aux, @max)
[3]-->(JNB, [5],  )
[4]-->(=, @aux, @max)
[5]-->(=, @max, max)
[6]-->(=, _2, @max)
[7]-->(=, _6, @aux)
[8]-->(CMP, @aux, @max)
[9]-->(JNB, [11],  )
[10]-->(=, @aux, @max)
[11]-->(=, @max, max)
```

CON OPTIMIZACIÓN

```
[6]-->(=, _2, @max)
[7]-->(=, _6, @aux)
[8]-->(CMP, @aux, @max)
[9]-->(JNB, [11],  )
[10]-->(=, @aux, @max)
[11]-->(=, @max, max)
```

Ejercicio 3 tipo de dato:

3.1)

Todas las etapas se verían afectadas, tanto la parte del analizador léxico, analizador semántica y generación de código assembler. Tengo que aclarar que el manejo de fecha es algo complejo de analizar, sobre todo si nos metemos con un lenguaje de muy bajo nivel como assembler como por ejemplo hacer el calculo del año bisiesto para saber la cantidad de días del año y también sobre el acumulamiento de días para pasar al siguiente mes, lo mismo para los meses. Por este motivo se considera a modo de ejemplo que todos los meses tendrán 31 días.

En la etapa del analizador léxico: se deberá tener un regex que detecte el formato de la fecha “10/07/2020”, con el objetivo de no causar conflicto con la operación división “2/2/9” que tiene el compilador, esto se debe tener en cuenta porque no se podrá hacer divisiones con el siguiente formato “10/07/2020” dado que sería una fecha, si el formato fuera “10-07-2020” no habrá conflicto con la operación división del compilador. También hay que agregar el tipo de dato al momento de declarar la/s variables en el bloque inicial.

Regex a utilizar: “[0-9][0-9][/][0-9][0-9][/][0-9][0-9][0-9][0-9]”

```
"FECHA"                 { return TIPO_FECHA;}
```

Tabla de símbolos: Al momento de insertar en la tabla de símbolos las fechas, se deberá tener 2 tipos de datos más, lo cuales son CTE_FECHA que representa a las constantes “10/07/2020” y FECHA que representa a las variables. Al insertar variables de tipo fecha en la tabla de símbolos se podrán guardar con el siguiente formato nombreDeVariable@dia, nombreDeVariable@mes y nombreDeVariable@año, estas variables se las manejara internamente y nos facilitara a la hora de realizar operaciones con las fechas y generar el código assembler.

Ejemplo:
d := 02/06/2020;

tabla de símbolo

Nombre	Tipo	Valor	Longitud
d@dia	FECHA		
d@mes	FECHA		
d@año	FECHA		
_02/06/2020	CTE_FECHA		

En la etapa del analizador semántico: En esta etapa se deberá agregar unas reglas y sus correspondientes acciones para que el compilador tenga en cuenta la asignación y operaciones de variable de tipo fecha, esto se logra agregando una regla en "factor" dado que en factor están las variables de tipo FLOAT, ENTERO, ID que ya tienen en cuanta la asignación y operaciones (+, -, *, /) dentro del compilador. Hay que aclarar que la verificación de tipos en las expresiones/operaciones se lo explicara en el punto **3.2.**

d= 02/06/2020
factor -----> ENTERO
factor -----> FLOAT
factor -----> ID
factor -----> FECHA

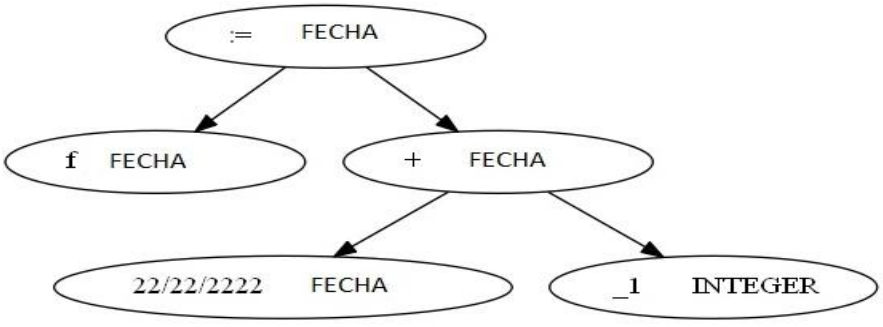
Además, también hay que agregar otra regla para las declaraciones de las variables de tipo fecha, esta regla se la agrega en la parte de "declaración", donde se declara la lista de id de un tipo de dato.

Ejemplo:
DEFVAR
FECHA : f, f2
ENDVAR

declaración -----> TIPO_INT DECS_2PTOS lista_variables
declaración -----> TIPO_FLOAT DECS_2PTOS lista_variables
declaración -----> TIPO_STRING DECS_2PTOS lista_variables
declaración -----> TIPO_FECHA DECS_2PTOS lista_variables

Árbol sintáctico: En esta etapa según como se ataque el problema puede tener cambios significativos como, por ejemplo.

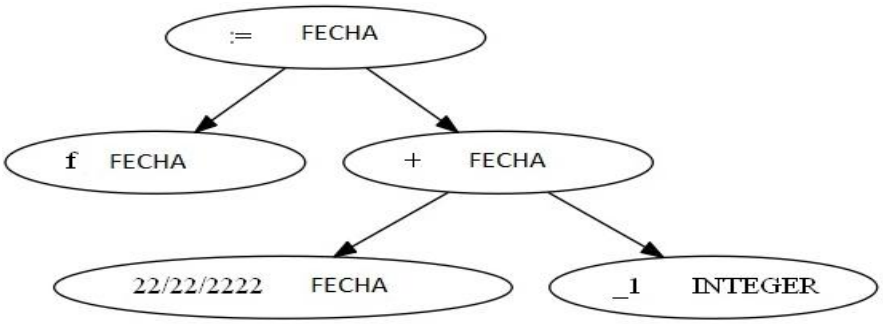
d := 02/06/2020 + 1;
Para este caso el árbol sintáctico no tendría grandes cambios, solamente que cada nodo tendría su tipo de dato, en la parte de generación de código assembler tendría los cambios importantes.



d := 02/06/2020 + 1;
Para este caso el árbol sintético debería tener todos los nodos correspondientes al siguiente "CODIGO" esto se lo puede hacer en las acciones. En este caso el árbol sintáctico tendría algunos cambios muy importantes, pero en la generación de código assembler no se vería tan afectada.

"CODIGO"
```
d@dia := 02 + 1;
IF (d@dia>31) {
        d@mes := 06+ d@dia / 31;
        d@dia := d@dia % 31;
} ELSE {
        d@mes := 06;
}
IF (d@mes>31) {
        d@año := 2020 + d@mes / 12;
        d@mes := d@mes  % 12;
} ELSE {
        d@año := 2020;
}
```

Generación de código assembler: Como se dijo previamente dependiendo de como se ataque el problema la generación de código assembler tendré cambios significativos.
Si partimos del siguiente árbol sintáctico en la generación de código assembler tendremos los cambios significativos. Básicamente tendríamos que generar el código assembler del "CODIGO" escrito anteriormente, esto se lo puede hacer simplemente sabiendo el tipo de dato de las variables que están involucradas en la operación (+, - , / , *), si alguna tiene el tipo de FECHA realizo el código assembler que representa lo escrito en el "CODIGO"



Para este caso que tengamos el árbol sintético con todos los nodos correspondientes al "CODIGO", la generación de código assembler no tendré cambios significativos solamente la validación de tipos de datos.

3.2)

Aclaraciones: Por lo que entiendo solamente se permite las expresiones que tengan un tipo de dato fecha y un tipo de dato entero, solamente para las operaciones + y -, la operación * y / tendrían un error. También no se permite la operación + y - entre variables o constantes de tipo fecha. Bajo estas condiciones se analizo el punto **3.2**, que es lo que entendí del enunciado y respeta los ejemplos que nos pasaron en la EA3

EJEMPLOS:

d= 02/06/2020 ***/ d es de tipo fecha, ok /***
a=31/05/2020 + 10 ***/ a tipo fecha es la suma de fecha + 10 dias /***
a= d - 1 ***/ a tipo fecha es la resta de d - 1 dia /***
w=31/05/2020 + 30/06/2020 ***/ error /***
w=31/05/2020 *10 ***/ error /***

La forma en que se resolvería la verificación de tipos en las expresiones seria mediante la matriz de tipos de datos y validaciones que se realizaría en las acciones de las reglas correspondiente.

validaciones en las acciones de las reglas: Para que el compilador de error al momento de hacer la operación de * y / que involucre alguna variable de tipo de dato FECHA, lo que se puede hacer es validar los tipos de datos que se involucran en la operación, si alguna variable tiene el tipo de dato FECHA entonces haremos que salte un error de compilación. Esto se puede realizar en las acciones de las reglas de "termino", tendremos una función que validará el tipo de dato. Esta función "validarTipodeDatoOperacion" se le pasara los nodos F y T que tendrán los datos y sus TIPOS DE DATOS, los cuales estos nodos son los que están implicados en la operación / y *.

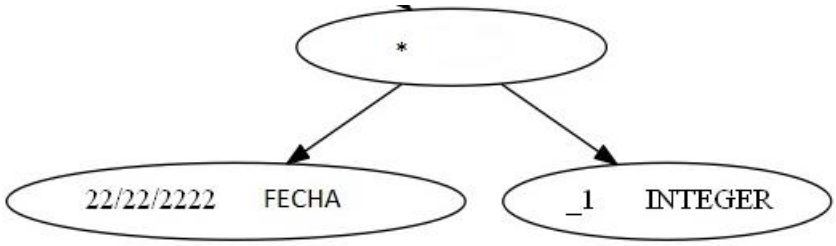
termino ----> termino OP_MUL factor {validarTipodeDatoOperacion(T, F)}

termino ----> termino OP_DIV factor {validarTipodeDatoOperacion(T, F)}

termino ----> termino

Ejemplo:

w=22/22/2222 *1 ERROR



matriz de tipos de datos: Una vez resuelto las validaciones de las operaciones * y / tendremos que validar los tipos de datos para la suma y resta, esto lo podemos hacer con la matriz de tipos de datos, lo que se hará es agregar una nueva columna y fila que tenga el tipo de dato FECHA, con esta matriz podemos validar la suma y resta de variables que tengan tipo de dato FECHA.

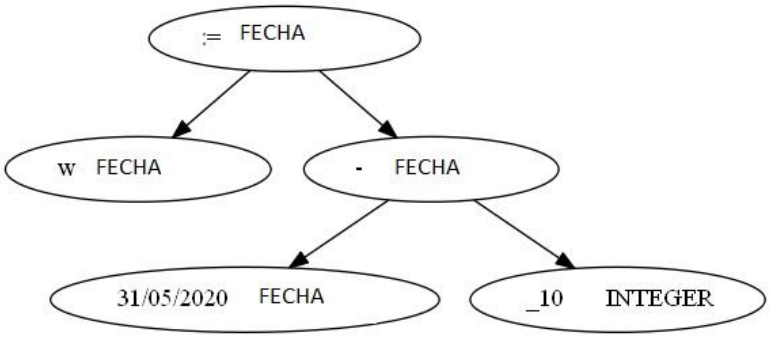
Ejemplo:

w=31/05/2020 + 30/06/2020 ***/ error /***
w=31/05/2020 - 30/06/2020 ***/ error /***

Como "31/05/2020" y "30/06/2020" son de tipo de dato fecha nos fijaremos en la matriz de tipo de datos, fila: T_FECHA, columna: T_FECHA esto nos dice que esta operación no es validad T_ERROR, esto va a hacer igual con la operación menos "-".

Ejemplo:

w=31/05/2020 + 1 ***/ ok /***
w=31/05/2020 - 10 ***/ ok /***



Como “31/05/2020” es de tipo de dato fecha y “1” es de tipo de dato integer nos fijaremos en la matriz de tipo de datos, fila: T_FECHA, columna: T_INTEGER esto nos dice que esta operación es validad y que el resultado de realizar esta operación devuelve un tipo de dato T_FECHA, esto va a hacer igual con la operación menos “-”.

MATRIZ DE TIPOS DE DATOS:

	T_NULL	T_INTEGER	T_FLOAT	T_STRING	T_FECHA
T_NULL	T_ERROR	T_ERROR	T_ERROR	T_ERROR	T_ERROR
T_INTEGER	T_ERROR	T_INTEGER	T_FLOAT	T_ERROR	T_FECHA
T_FLOAT	T_ERROR	T_FLOAT	T_FLOAT	T_ERROR	T_ERROR
T_STRING	T_ERROR	T_ERROR	T_ERROR	T_STRING	T_ERROR
T_FECHA	T_ERROR	T_FECHA	T_ERROR	T_ERROR	T_ERROR