Extensión de la Práctica Principal de Procesadores de Lenguajes (Examen febrero 2019)

Esta práctica consiste en la implementación mediante JFlex y Cup de un compilador de un pequeño lenguaje de programación, similar a C, denominado lenguaje PLX. El lenguaje PLX es una extensión del lenguaje PL que se describe como práctica de la asignatura, pero en esta versión extendida se requieren algunas funciones adicionales. Se presupone que todos los elementos del lenguaje PL están presentes en el lenguaje PLX y que no se modifica su funcionamiento al incluir los nuevos elementos de esta extensión.

EL CÓDIGO FUENTE (Lenguaje PLX):

El lenguaje PLX incluye todas las sentencias definidas en el lenguaje PL y algunas más. Asimismo, se modifica ligeramente el lenguaje intermedio CTD, de manera que soporte algunas instrucciones adicionales.

ASPECTOS LEXICOS

Todos los aspectos léxicos relacionados con este lenguaje se resuelven, a menos que se especifique lo contrario, de la misma manera que en el lenguaje JAVA. Así, por ejemplo, los identificadores validos son secuencias de caracteres que comienzan por una letra, seguidos de ninguno, uno o mas caracteres alfanuméricos. Las constantes enteras se escriben mediante dígitos, si comienzan por “0” se interpreta que están escritos en octal y si comienzan por “0x”, se interpreta que están escritos en hexadecimal.

Cualquier duda que surja al implementar, y que no estuviera suficientemente clara en este enunciado debe resolverse de acuerdo a las especificaciones del lenguaje JAVA.

COMPILADOR DE PRUEBA

Se proporciona una versión compilada del compilador, que puede usarse como referencia para la generación de código. No es necesario que el código generado por el compilador del alumno, sea exactamente igual al generado por el compilador de prueba, basta con que produzca los mismos resultados al ejecutarse para todas las entradas.

El compilador de prueba se entrega solamente a titulo orientativo. Si hubiese errores en el compilador de prueba, prevalecen las especificaciones escritas en este enunciado. Estos posibles errores en ningún caso eximen al alumno de realizar una implementación correcta.

Matrices unidimensionales de **char**. (Implementado ya en el ejercicio de PLX-2018)
Al igual que con los enteros, se permite la declaración de matrices unidimensionales de caracteres, y su inicialización mediante la notación de llaves, tanto en la declaración como en las instrucciones de asignación.
Para obtener la máxima calificación debe comprobarse que el rango del ***array*** coincide con el asignado. Se acepta que el rango asignado sea menor o igual, pero no que sea mayor.
Se puede acceder a la longitud de la matriz mediante el operador **.length**
Deben poder combinarse las asignaciones entre variables de tipo ***array*** y ***arrays constantes*** definidos mediante llaves, tal y como puede hacerse en la declaración e inicialización de variables tipo ***array*** en el lenguaje Java. En PLX se acepta este tipo de expresiones también en las sentencias de asignación.

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
`char st[3];` `st[0] = 'A';` `st[1] = 'B';` `st[2] = 'C';` `print(st[2]);` `print(st[1]);` `print(st[0]);`	…	**C** **B** **A**
`char st[3];` `st = { 'A', 'B', 'C' };` `print(st[2]);` `print(st[1]);` `print(st[0]);`	…	**C** **B** **A**
`char st[3] = {'X','/','Z'};` `print(st[2]);` `print(st[1]);` `print(st[0]);`	…	**Z** **/** **X**
`char st[2];` `st = { 'A', 'B', 'C' };` `print(st[0]);`	… `error;` …	**--**
`char a[3];` `a[0] = 'A';` `a[1] = 'B';` `a[2] = 'C';` `int i;` `for(i=0; i<a.length; i=i+1) {` `  print (a[i]);` `}` `print (a.length);`	…	**A** **B** **C** **3**
`char a[3];` `char b[5];` `a = {'A','B','C'};` `b = a;` `print(b[1]);`	…	**B**
`char a[3];` `char b[2];` `a = {'A','B','C'};` `b = a;` `print(b[1]);`	… `# las matrices no son compatibles` `error;` …	**--**
`int i;` `char a[3];` `for(i=0; i<5; i=i+1) {` `  a[(i+i)/2] = 'X';` `  print (a[i]);` `}`	… `if (t2 < 0) goto L4;` `  if (3 < t2) goto L4;` `  if (3 == t2) goto L4;` `  goto L5;` `L4:` `  error;` `  halt;` `L5:` `….`	**X** **X** **X** **runtime error**
`char a[5];` `a[0] = 'A'; a[4] = 'E';` `a = {'X','B','Z'};` `a[2] = 'C'; a[3]='D';` `print(a[0]); print(a[2]);` `print(a[3]); print(a[4]);`	…	**X** **C** **D** **E**

Sentencia **print** para matrices.
Implementar la sentencia **print** cuando su argumento es de tipo matriz unidimensional, ya sea de enteros o de caracteres.

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
`int n[3];` `n[0] = 65;` `n[1] = 66;` `n[2] = 67;` `print(n);`	`...`	`65` `66` `67`
`int n[3];` `n = { 65, 66, 67};` `print(n);`	`$n_length = 3;` `n[0] = 65;` `n[1] = 66;` `n[2] = 67;` `$t1 = n[0];` `print $t1;` `$t1 = n[1];` `print $t1;` `$t1 = n[2];` `print $t1;`	`65` `66` `67`
`print({ 65, 66, 67});`	`print 63;` `print 64;` `print 65;`	`65` `66` `67`
`char st[3];` `st[0] = 'A';` `st[1] = 'B';` `st[2] = 'C';` `print(st);`	…	`A` `B` `C`
`char st[3];` `st = { 'A', 'B', 'C' };` `print(st);`	`$st_length = 3;` `st[0] = 65;` `st[1] = 66;` `st[2] = 67;` `$t1 = st[0];` `printc $t1;` `$t1 = st[1];` `printc $t1;` `$t1 = st[2];` `printc $t1;`	`A` `B` `C`
`print( { 'A', 'B', 'C' } );`	`print 65;` `print 66;` `print 67;`	`A` `B` `C`

Introducción del tipo **string**.

Se permite la definición de constantes y variables de tipo **string**. Para las constantes se emplea la misma sintaxis que en Java, usando comillas dobles, (por ejemplo **"abc"**) y pudiendo usar las secuencias de escape al igual que en Java para los caracteres especiales dentro de las comillas **(\b,\n,\f,\r,\t,\",\\,\')**, así como secuencias en Unicode (por ejemplo \u1234). Las variables de tipo **string** pueden inicializarse en la definición. La sentencia **print** (del lenguaje fuente) debe admitir argumentos de tipo **string**, en cuyo caso generara la instrucción de código de tres direcciones **writec** por cada uno de los caracteres de entrada y un carácter de fin de línea al final. (La diferencia entre las instrucciones **printc** y **writec** del código de tres direcciones es que la primera imprime un salto de línea tras el carácter, y la segunda no).

AYUDA: Existen diversas formas de implementar cadenas de longitud variable, una de ellas consiste en definir una matriz de caracteres y una variable que contenga la longitud de la cadena. No obstante hay diferencias entre las matrices de caracteres y el tipo **string**, Una diferencia es que el tipo **string** puede tener una longitud variable, mientras que las matrices son siempre de tamaño constante. Por otra parte, la sentencia **print** se comporta de forma diferente con matrices de caracteres y con **string** (Ver ejemplos)

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
`print("abc");`	`writec 97;` `writec 98;` `writec 99;` `writec 10;`	`abc`
`print("a\\b");`	…	`a\b`
`print("a\"b\"c\"d\"");`		`a"b"c"d"`
`print("Espa\u00f1a");`		`España`
`string a;` `a = "abc";` `print(a);`		`abc`
`string a = "abc";` `print(a);`		`abc`
`string a = "abc", b="xyz";` `print(a);` `print(b);`		`abc` `xyz`
`string a = "abc", b, c="xyz";` `b=c;` `print(a);` `print(b);` `print(c);`		`abc` `xyz` `xyz`

Operaciones básicas con el tipo **string** e interoperabilidad con el tipo **char**.
Se admite la conversión implícita de **char** a **string** en caso necesario, de manera que a una cadena de caracteres se le puede asignar un solo carácter. También puede hacerse la conversión de tipos explicita usando el operador unario **(string)**, aplicado a caracteres. Para acceder al carácter i-ésimo de una variable de tipo **string**, se utilizará la notación matricial, por ejemplo **st[0]** denota el primer carácter de la cadena **st,** etc. Al igual que en el caso de las matrices, para obtener la máxima puntuación en este apartado deberá comprobarse el rango en tiempo de ejecución.
El operador **.length** aplicado a una variable de tipo **string** devuelve su longitud. Nótese que no puede modificarse directamente el valor de esta variable.

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
`string st;` `st='a';` `print(st);`		`a`
`string st;` `char ch;` `ch='z';` `st = ch;` `print(st);`		`z`
`string st;` `int n;` `n=65;` `st = (string) (char) n;` `print(st);`		`A`
`string  st;` `st="PLXC";` `char ch;` `ch = st[1];` `print(ch);` `print(st[3]);`		`L` `C`
`string  st;` `st="PLXC";` `print(st[4]);`		`....` `runtine error`
`string  st;` `st="PLXC";` `int l;` `l = st.length;` `print(l);`		`4`
`string  st;` `st="PLXC";` `int tam;` `tam=4;` `st.length=tam;` `print(tam);`	`..` `error;`	

Interoperabilidad del tipo **string** con las matrices de caracteres de tipo **char**.
Se admite la conversión explicita de **arrays** de **char** a **string** en caso necesario, de manera que a una variable de tipo string se le puede asignar un **array** de **char** y viceversa, con el casting **`(string)`** y **`(char[])`** respectivamente.
Se admite la conversión explicita del tipo **string** con el tipo **array** de enteros, (mediante i**`nt()`**) y de éste con el de **array** de **char**.
Si el tamaño del **array** es menor que el número de caracteres del **string**, se producirá un error en tiempo de ejecución. Este tipo de error no se puede detectarse en tiempo de compilación, porque la longitud de las variables de tipo **string** puede variar, pero si en tiempo de ejecución al comprobar los rangos.
Asignar un **string** a un **array** no modificará la longitud del **array**.

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
`string st;` `char mat[5];` `mat[0]='A';` `mat[1]='B';` `mat[2]='C';` `st = (string) mat;` `print(st);`		`ABC`
`int mat[5];` `mat[0]=65;` `mat[1]=66;` `mat[2]=67;` `print((char[]) mat);`		`A` `B` `C`
`int mat[5];` `mat[0]=65;` `mat[1]=66;` `mat[2]=67;` `print((string) mat);`		`ABC`
`string st;` `st="PLXC";` `int aint[5];` `aint = (int[]) st;` `print(aint[1]);` `print(aint[3]);`		`76` `67`
`string st;` `st="PLXC";` `char achar[5];` `achar = (char[]) st;` `print(achar[1]);` `print(achar[3]);`		`L` `C`
`string st;` `st="PLXC";` `char achar[2];` `achar = (char[]) st;` `print(achar[1]);` `print(achar[3]);`		`....` `runtine error`

Operación de concatenación del tipo **string**.

Al igual que en Java, el operador + aplicado a dos variables o constantes de tipo **string** cadenas de representa la concatenación. Se admite la concatenación mixta entre caracteres y cadenas de caracteres, siendo siempre el resultado una cadena de caracteres.

Tenga en cuenta que dado que el operador + es asociativo por la izquierda, al concatenar dos caracteres y una cadena de caracteres, primero se operan los dos caracteres, dando como resultado una cadena de caracteres que luego se concatena con la otra cadena de caracteres. (Ver ejemplos)

Implementar el operador de asignación += aplicado a **string**, con conversión implícita en el caso en el que el operador derecho sea de tipo **char**.

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
`print(”abc”+”-xyz”);`		`abc-xyz`
`print(”abc”+’z’);`		`abcz`
`print(’a’+”xyz”);`		`axyz`
`string st;` `st=”AEIOU”;` `string vocales;` `vocales= st + ‘+’ + ’a’+’e’+’i’+’o’+’u’;` `print(st);` `print(vocales);`		`AEIOU` `AEIOU+aeiou`
`string st;` `st=”aeiou”;` `st=st+st+st;` `print(st.length);` `print(st);`		`15` `aeiouaeiouaeiou`
`print(’a’+’z’+ “==az”);`		`az==az`
`string ab=”ab”;` `string xyz=”xyz”;` `print(ab+”-pqrst-”+xyz);`		`ab-pqrst-xyz`
`string min;` `min="aeiou";` `string may;` `int i;` `for(i=0;i<min.length;i=i+1) {` `  may=may+(char)(((int)min[i])-32);` `}` `print(may);`		`AEIOU`
`string min;` `min="aeiou";` `string may;` `int i;` `for(i=0;i<min.length;i=i+1) {` `  may+=(char)(((int)min[i])-32);` `}` `min += "-" + may ;` `print(may);` `print(min);`		`AEIOU` `aeiou-AEIOU`
`string min;` `min="aei" + (min+="ou");` `string may;` `int i;` `for(i=0;i<min.length;i=i+1) {` `  may+=((char)(((int)min[i])-32))+’-‘;` `  print(may);` `}`		`A-` `A-E-` `A-E-I-` `A-E-I-O-` `A-E-I-O-U-`

EL CÓDIGO OBJETO (Código de tres direcciones):

El código objeto es a su vez una extensión del código intermedio utilizado en la practica principal de la asignatura. Se añaden algunas instrucciones necesarias para generar el código requerido por el lenguaje PLX. Todas las variables del código intermedio se considera que están previamente definidas y que su valor inicial es 0.

El conjunto de instrucciones del código ensamblador, y su semántica son las siguientes:

Instrucción	Acción
`x = a ;`	Asigna el valor de a en la variable x
`x = a + b ;`	Suma los valores de a y b, y el resultado lo asigna a la variable x
`x = a - b ;`	Resta los valores de a y b, y el resultado lo asigna a la variable x
`x = a * b ;`	Multiplica los valores de a y b, y el resultado lo asigna a la variable x
`x = a / b ;`	Divide (div. entera) los valores de a y b , y el resultado lo asigna a la variable x
`x = (int) a ;`	Convierte un valor real a, en un valor entero, asignándoselo a la variable x
`x = (float) a ;`	Convierte un valor entero a, en un valor real, asignándoselo a la variable x
`x = y[a] ;`	Obtiene el a-ésimo valor del array y, asignando el contenido en x
`x[a] = b ;`	Coloca el valor b en la a-ésima posición del array x
`goto l ;`	Salto incondicional a la posición marcada con la sentencia "`label l`"
`if (a == b) goto l ;`	Salta a la posición marcada con la sentencia "`label l`", si y solo si el valor de a es igual que el valor de b
`if (a < b) goto l ;`	Salta a la posición marcada con la sentencia "`label l`", si y solo si el valor de a es estrictamente menor que el valor de b.
`l:`	Indica una posición de salto.
`label l ;`	Indica una posición de salto. Es otra forma sintáctica equivalente a la anterior.
`write a ;`	Imprime el valor de a
`writec a ;`	Imprime el carácter Unicode correspondiente al número a
`print a ;`	Imprime el valor de a, y un salto de línea
`printc a ;`	Imprime el carácter Unicode correspondiente al número a, y un salto de línea
`error ;`	Indica una situación de error, pero no detiene la ejecución.
`halt ;`	Detiene la ejecución. Si no aparece esta instrucción la ejecución se detiene cuando se alcanza la última instrucción de la lista.
`# ….`	Cualquier línea que comience con un # se considera un comentario.

En donde `a,b` representan tanto variables como constantes enteras o reales, `x,y` representan siempre una variable y `l` representa una etiqueta de salto.

IMPLEMENTACIÓN DE LA PRÁCTICA:

Se proporciona una solución compilada del ejercicio (versiones para Linux, Windows y Mac). Esto puede servir de ayuda para comprobar los casos de prueba y las instrucciones intermedio,. No es necesario que el código generado sea idéntico al que se propone como ejemplo (que de hecho no es óptimo), basta con que sea equivalente, es decir que dé los mismos resultados al ejecutar los casos de prueba. Para compilar y ejecutar un programa en lenguaje PLX, pueden utilizarse las instrucciones

	Linux
Compilación	`java PLXC prog.plx prog.ctd`
Ejecución	`./ctd prog.ctd`

En donde `prog.plx` contiene el código fuente en PLX, `prog.ctd` es un fichero de texto que contiene el código intermedio válido según las reglas gramaticales de este lenguaje. El programa `plxc` es un *script* del *shell* del sistema operativo que llama a (`java PLXC`), que es el programa que se pide construir en este ejercicio. El programa `ctd` es un interprete del código intermedio. Asimismo, para mayor comodidad se proporciona otro *script del shell* denominado `plx` que compila y ejecuta en un solo paso, y al que se pasa el nombre del fichero sin extensión.

	Linux
Compilación + Ejecución	`./plx prog`

NOTAS IMPORTANTES:

1. Toda práctica debe contener al menos tres ficheros denominados "`PLXC.java`", "`PLXC.flex`" y "`PLXC.cup`", correspondientes respectivamente al programa principal y a las especificaciones en JFlex y Cup. Para realizar la compilación se utilizarán las siguientes instrucciones:

```
cup PLXC.cup
jflex PLXC.flex
javac *.java
```

   y para compilar y ejecutar el programa en PLX

```
java PLXC prog.plx prog.ctd
./ctd prog.ctd
```

2. Puede ocurrir que al descargar los ficheros y descomprimirlos en Linux se haya perdido el carácter de fichero ejecutable. Para poder ejecutarlos debe modificar los permisos:

```
chmod +x plx plxc ctd
chmod +x plx-linux plxc-linux ctd-linux
```

3. El programa `ctd`, interprete del código intermedio, tiene una opción –v para generar trazas que pueden ayudar en la depuración de errores:

```
./ctd –v prog.ctd
```

4. Los ejemplos que se proponen como casos de prueba no definen exhaustivamente el lenguaje. Para implementar esta práctica es necesario generar otros casos de prueba de manera que se garantice un funcionamiento en todos los casos posibles, y no solo en este limitado banco de pruebas.

5. En todas las pruebas en donde el código **plx** produce un "*error*", para comprobar que el compilador realmente detecta el error, se probará también que el código corregido compila adecuadamente, y si no es así la prueba no se considerará correcta.