Extensión de la Práctica Principal de Procesadores de Lenguajes (Examen febrero 2018)

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 o 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.

Introducción del tipo **char**.

Se permite la definición de constantes y variables de tipo **char**. Para las constantes se emplea la misma sintaxis que en Java, usando comillas simples, (por ejemplo 'a') y pudiendo usar las secuencias de escape al igual que en Java para los caracteres especiales ('\b','\n','\f','\r','\t','\"','\\','\''), así como secuencias en Unicode (por ejemplo '\u1234').

La sentencia **print** debe admitir argumentos de tipo **char**, en cuyo caso generara la instrucción de código de tres direcciones **printc**, en vez de la sentencia de código de tres direcciones **print**, lo que provocará que se escriba un carácter en vez del número.

Al igual que en los lenguajes C y Java, el tipo **char** interopera con el tipo **int**, de manera que se puede realizar la conversión explicita de tipos (pero no la implícita) entre ambos, mediante los operadores **(char)** y **(int)**. Nótese que en el código de tres direcciones no es necesario realizar la conversión de tipos porque en el código objeto es exactamente lo mismo un número que un carácter.

Las variables de tipo **char** pueden inicializarse en la definición, y declararse en una sola instrucción. Las variables no inicializadas contienen el carácter'\u0000'.

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
`print(65);` `print('A');`	`print 65;` `printc 65;`	**65** **A**
`char a;` `a='&';` `print(a);`	`a = 38;` `printc a`	**&**
`char a;` `int x;` `x = 67;` `a = (char) x;` `print(x);` `print(a);`	`x = 67;` `a = x;` `print x;` `printc a;`	**67** **C**
`char a;` `int x;` `x = 67;` `a = x;` `print(x);` `print(a);`	`# Error de tipos` `error;`	
`print('\\');` `print(1);`	`printc 92;` `print 1;`	\ **1**
`print('\u0043');` `print('\u00f1');` `print('\u25b2');`	`printc 67;` `printc 241;` `printc 9650;`	**C** **ñ** **▲**
`print((int)'\u0043');` `print((int)'\u00f1');` `print((int)'\u25b2');`	`print 67;` `print 241;` `print 9650;`	**67** **241** **9650**
`char a = '0';` `char b = '1';` `print(a);` `print(b);`	`a = 48;` `b = 49;` `printc a;` `printc b;`	**0** **1**
`char a='X', b='Y', c='Z';` `print(a);` `print(b);` `print(c);`	`...`	**X** **Y** **Z**
`char a1='1',a1, a3;` `char a4,a5='5';` `print((int)a1+(int)a2+(int)a5);`	`...`	**102**
`int a;` `int b;` `int c;` `a = (int) 'A';` `c = (int) 'C';` `b = ((int) a + (int) c) / 2;` `print(a);` `print((char)b);` `print(c);`	`a = 65;` `b = 241;` `$t0 = 65 * 66;` `$t1 = $t0 * 67;` `c = $t1;` `print a;` `print b;` `print c;`	**65** **241** **287430**
`int x=(int)'a'*(int)(char)(65+1);` `print(x);`	`...`	**6402**

Operaciones básicas del tipo **char**.

En el lenguaje PLX los caracteres se pueden sumar, y restar (pero no multiplicar o dividir), usando los operadores + y – respectivamente. El resultado de la operación es un número entero correspondiente a la diferencia de sus valores. Para no confundir a suma con la concatenación, se requiere que al menos uno de los operandos sea un número entero, es decir se puede suma **'A'+1** , o bien **1+'A'**, pero no se puede sumar **'A'+'B'**. Si se quiere realizar esta operación, dado que el operador + es asociativo por la izquierda bastaría usar la expresión **0+'A'+'B'**. Estos operadores se pueden aplicar tanto a variablescomo a constantes de tipo **char.**

El operador unario **!** cambia el carácter por el mismo en mayúsculas, si se trata de una letra. Si se aplica a algún caracteres fuera del rango de las letras no tiene ningún efecto. El operador unario ~ cambia mayúsculas por minúsculas y viceversa, es decir, una letra mayúscula la cambia por minúscula, y una minúscula por mayúscula. La combinación de ambos, es decir **~!** hace la conversión en minúsculas, sea cual sea la entrada. Estos operadores tiene mayor prioridad que cualquier otro.

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
`print('z'-'a');`	`$t0 = 122 - 97;` `print $t0;`	**25**
`int x;` `x = 0 + 'a' + 'b' + 'c';` `print(x);`	`$t0 = 0 + 97;` `$t1 = $t0 + 98;` `$t2 = $t1 + 99;` `x = $t2;` `print x;`	**294**
`int num;` `num = 'A' + ('Z'-'A') / 2;` `print(num);`	`$t0 = 90 - 65;` `$t1 = $t0 / 2;` `$t2 = 65 + $t1;` `num = $t2;` `print num;`	**77**
`char ch1, ch2;` `ch1 = '+';` `ch2 = '*';` `print( 10 + ch1 – ch2 );`	`ch1 = 43;` `ch2 = 42;` `$t0 = 10 + ch1;` `$t1 = $t0 - ch2;` `print $t1;`	**11**
`char a;` `char B;` `char x;` `a='a';` `B='B';` `x='7';` `print(!a);` `print(!B);` `print(!x);`	…	**A** **B** **7**
`char a;` `char B;` `char x;` `a='a';` `B='B';` `x='7';` `print(~a);` `print(~B);` `print(~x);`	…	**A** **b** **7**
`char a, sub, b;` `a = 'A';` `sub = '_';` `b = 'B';` `print(~a);` `print(~sub);` `print(~!b);`	…	**a** **_** **b**
`char b='z';` `print(~(char)(~!'a'-~~b+90));`	…	**a**

Matrices unidimensionales de **char**.

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 = { 'P', 'Q', 'R' };` `print(st[2]);` `print(st[1]);` `print(st[0]);`	…	**R** **P** **Q**
`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**

Bucle for para matrices unidimensionales de tipo **int** y **char**.

Para recorrer un array unidimensional, ya sea de enteros o de caracteres puede utilizarse una forma abreviada de la sentencia **for**, de forma similar a como se hace en Java. Para ello, debe utilizarse una variable auxiliar que va tomando valores desde el primer hasta el ultimo valor del array. (Veanse ejemplos). Los bucles **for** se pueden anidar sin limite.

Es importante que el tipo de la variable que recorre el bucle y el tipo del *array* coincidan, es decir para recorrer un *array* de enteros hay que usar una variable entera, y para un array de caracteres, una variable de tipo **char**.

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
`char st[2];` `st[0] = 'A';` `st[1] = 'B';` `char ch;` `for(ch : st)` `  print(ch);`	…	**A** **B**
`char st[3];` `int m[3];` `m[0] = 0;` `m[1] = 2;` `m[2] = 4;` `char ch;` `int i,j;` `for(i : m) {` `  st[j] = (char) ('K' + i);` `  j=j+1;` `}` `for(ch : st)` `  print(ch);`		**K** **M** **O**
`char st1[3];` `char st2[2];` `st1[0] = 'A'; st2[0] = 'X';` `st1[1] = 'B'; st2[1] = 'Y';` `st1[2] = 'C';` `char ch1,ch2,ch3;` `for(ch1 : st1) {` `  print(ch1);` `  for( ch2 : st2)` `  print(ch2);` `}`	…	**A** **X** **Y** **B** **X** **Y** **C** **X** **Y**
`int m[3];` `char st[3];` `int i,x,y;` `for(i=0; i<m.length;i++) {` `  m[i] = i+65;` `  st[i] = (char) m[i];` `}` `for(i=m.length-1;i>0;i--) {` `  for(x : m) {` `  m[i] = x + m[i-1];` `  print(x);` `  }` `}`	…	**65** **66** **132** **65** **130** **198**
`int i1,i2,i3;` `int m[3]={0,1,2};` `for(i1:m) {` `  for(i2:m)` `  for(i3:m)` `  m[(i2+i3)%3] = i2+i3;` `  print(i1);` `}`		**0** **7** **62**
`int m[5];` `char ch;` `for(ch : m)` `  print(ch);`	… `# Error de tipos en for` `error;`	--

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 = 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.
`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.