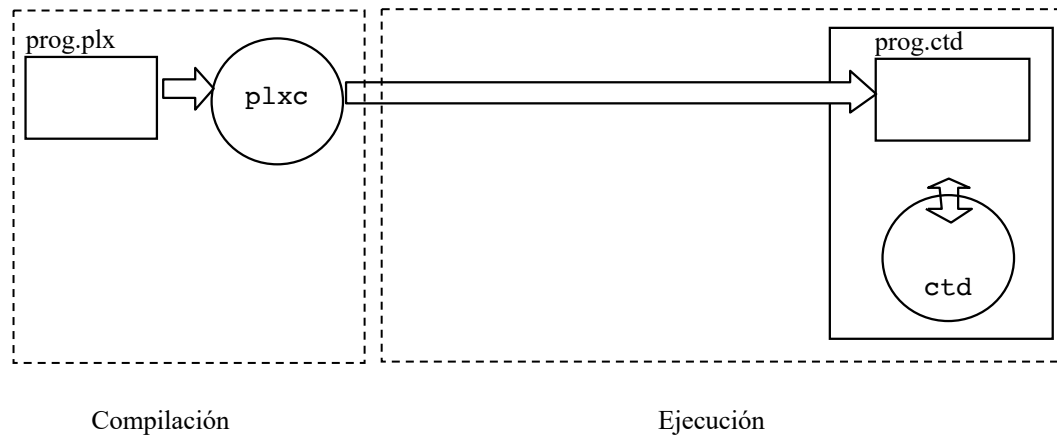


Práctica Principal de Procesadores de Lenguajes

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 practica 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.

La mayor diferencia entre PL y PLX es que en PLX se exige la declaración de variables previa. El código intermedio generado por ambos lenguajes es el mismo y el esquema de compilación también.



Es decir, el código fuente de `prog.plx` que se traduce a código intermedio generando el fichero `prog.ctd`. Este código de tres direcciones es la entrada a otro programa, denominado `ctd`, que ejecuta una a una las instrucciones.

SE PIDE: implementar con Java, JFlex y Cup el compilador del lenguaje fuente PLX al código intermedio. Para ello, será necesario (al menos) implementar los ficheros **PLXC.java**, **PLXC.flex** y **PLXC.cup**, que una vez compilados darán lugar a varias clases Java, entre ellas **PLXC.class**. Es decir, construir el compilador que antes hemos denominado **plx**, (equivalente a “**java PLXC**”), según las instrucciones indicadas en los siguientes enunciados.

COMPILADOR DE PRUEBA

Se proporcionan versiones compiladas de “plx” (el compilador) y de “ctd” (el intérprete de código intermedio) para distintos sistemas operativos.

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 título 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.

DETECCION DE ERRORES

El compilador no incorpora la recuperación de fallos, de manera que detiene su ejecución al encontrar la primera instrucción incorrecta, ya sea en el análisis léxico, sintáctico o semántico. En estos casos basta con que la salida contenga la instrucción “**error**”, no siendo necesario indicar la causa.

EL LENGUAJE PLX

Aspectos léxicos

Desde el punto de vista léxico el lenguaje PLX es igual que los lenguajes C, C++ y JAVA, y rigen las mismas reglas de ámbito de declaración de variables. PLX admite el mismo tipo de comentarios que C++. Cualquier duda que surja al implementar, y que no estuviera suficientemente clara en este enunciado debe resolverse de acuerdo con las especificaciones de estos lenguajes.

Tipos

El lenguaje PLX tiene tres tipos básicos correspondientes a los tipos en Java **int**, **float** y **char**. Cada uno de estos tipos tiene asociadas constantes, en el caso de los enteros se usará la notación decimal, en los reales la notación con punto decimal y exponente, y en los caracteres, las constantes aparecen entre comillas simples.

Se utiliza también el tipo compuesto **String**, cuyas constantes son secuencias de caracteres entre comillas dobles, al igual que en Java.

Se pueden definir asimismo matrices unidimensionales de cualquiera de los tipos simples anteriores, siendo las matrices de caracteres equivalentes al tipo **String**

Declaración de variables

Para poder usar variables es necesario declararlas, asignándoles un tipo. Los identificadores de las variables se componen (al igual que en Java) de caracteres alfanuméricos, comenzando por un carácter no numérico.

En una misma línea se pueden declarar varias variables, separándolas por comas, y opcionalmente se les puede asignar un valor inicial. Si una variable no se inicializa se considera que su valor es cero (o el equivalente según el tipo).

Expresiones aritméticas

Se incluyen en el lenguaje expresiones aritméticas con los cuatro operadores de suma, resta, multiplicación y división. La operación que se traduzca a nivel de código intermedio dependerá del tipo de la expresión, pudiendo interoperar tanto números enteros como números reales, realizando conversiones implícitas o explícitas entre enteros y reales en caso necesario.

La operación de suma aplicada a dos caracteres implica la concatenación de ambos (como en Java) dando lugar a un objeto de tipo **String**.

Condiciones

Las condiciones resultan de la aplicación de operadores relacionales (igual, distinto, menor, mayor, menor que, etc.) entre dos expresiones. Las condiciones se pueden componer mediante los operadores lógicos de conjunción, disyunción y negación. (**&&**, **||** y **!**) realizando evaluaciones en cortocircuito.

Sentencia de salida

La única sentencia de salida es la instrucción **print**, que puede aplicarse a cualquiera de los tipos del lenguaje PLX. El código generado será diferente según el tipo. (ver ejemplos)

Sentencias de control

El núcleo del lenguaje dispondrá de sentencias de asignación, y de las sentencias de control **if-else**, **while**, **do-while** y **for**, con la misma semántica que en el lenguaje Java. Las sentencias de control pueden anidarse indistintamente unas sobre otras. (ver ejemplos)

* Declaración obligatoria de variables. Todas las variables que aparezcan deben haber sido previamente declaradas. Para este apartado se considera que las variables son todas de tipo entero, y que se inicializan a cero de forma automática. La inicialización de variables puede ocurrir en cualquier parte del código, no solo al principio, y siempre antes de que se haga uso de la variable. En caso de que se produzca un error, el programa deberá detectarlo e informar mediante un mensaje. Para estos primeros apartados, puede considerarse que las variables son todas globales, aunque se declaren dentro de un bloque.

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
<code>int x; print (x); print (x);</code>	<code>print x;</code>	0
	<code>... error; # variable no declarada ...</code>	--

* Declaración múltiple de variables. En una misma línea se pueden declarar más de una variable, separándolas por comas. Debe controlarse que no se declare una misma variable dos veces.

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
<code>int x,y; x=y=1; print (x+y);</code>	<code>y=1; x=y; t0=x+y; print t0;</code>	2
<code>int x,y,x; x=y=1; print (x+y);</code>	<code>... error; # variable ya declarada ...</code>	--

* Inicialización de variables. Las variables se podrán inicializar mediante una expresión al ser declaradas. En el caso de declaración múltiple, puede haber también múltiples inicializaciones.

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
<code>int x=1; print (x);</code>	<code>x=1; print x;</code>	1
<code>int x=1, y=1+2*3, z=0; print (x+y);</code>	<code>x=1; t0=2*3; t1=1+t0; y=t1; z=0; t2=x+y; print t2;</code>	8

* Expresiones aritméticas y operadores. Se incluyen en el lenguaje expresiones aritméticas con los cuatro operadores de suma, resta, multiplicación y división. La operación que se traduzca a nivel de código intermedio dependerá del tipo de la expresión, pudiendo interoperar tanto números enteros como números reales. La prioridad y asociatividad de los operadores es la habitual en Java, es decir, todos los operadores son asociativos por la izquierda salvo el operador de asignación es asociativo por la derecha y tiene la menor prioridad de operación.

Código fuente (PL)	Código intermedio (CTD)	Resultado de la ejecución
<code>print (1+2*3);</code>	<code>\$t0 = 2 * 3;</code> <code>\$t1 = 1 + \$t0;</code> <code>print \$t1;</code>	7
<code>print(1 - -201*3 + 40*50 - -6);</code>	<code>\$t0 = -201;</code> <code>\$t1 = \$t0 * 3;</code> <code>\$t2 = 1 - \$t1;</code> <code>\$t3 = 40 * 50;</code> <code>\$t4 = \$t2 + \$t3;</code> <code>\$t5 = -6;</code> <code>\$t6 = \$t4 - \$t5;</code> <code>print \$t6;</code>	2610
<code>print(1*2+((3+2)*4)/2);</code>	<code>\$t0 = 1 * 2;</code> <code>\$t1 = 3 + 2;</code> <code>\$t2 = \$t1 * 4;</code> <code>\$t3 = \$t2 / 2;</code> <code>\$t4 = \$t0 + \$t3;</code> <code>print \$t4;</code>	12
<code>print(-64 / -16 / -2 * -4 / -2);</code>	<code>\$t0 = -64;</code> <code>\$t1 = -16;</code> <code>\$t2 = \$t0 / \$t1;</code> <code>\$t3 = -2;</code> <code>\$t4 = \$t2 / \$t3;</code> <code>\$t5 = -4;</code> <code>\$t6 = \$t4 * \$t5;</code> <code>\$t7 = -2;</code> <code>\$t8 = \$t6 / \$t7;</code> <code>print \$t8;</code>	-4
<code>print(3/-2 * ((4+5) - -9/4));</code>	<code>\$t0 = -2;</code> <code>\$t1 = 3 / \$t0;</code> <code>\$t2 = 4 + 5;</code> <code>\$t3 = -9;</code> <code>\$t4 = \$t3 / 4;</code> <code>\$t5 = \$t2 - \$t4;</code> <code>\$t6 = \$t1 * \$t5;</code> <code>print \$t6;</code>	-11

* Asignaciones. El operador de asignación es asociativo por la derecha y tiene la menor prioridad de operación. El resultado de una asignación es una expresión que toma el valor de la variable que se está asignando.

Código fuente (PL)	Código intermedio (CTD)	Resultado de la ejecución
<pre>int x; int y; int zeta; int p; int q; x=6; y=2; { p = q = 2; zeta = p+q; } print (zeta+x/y);</pre>	<pre>x = 6; y = 2; q = 2; p = q; \$t0 = p + q; zeta = \$t0; \$t1 = x / y; \$t2 = zeta + \$t1; print \$t2;</pre>	7
<pre>int x; int y; int z; { x=1; { y = 2; { x = 4; } z = 3; } } print (x); print (y); print (z);</pre>	<pre>x = 1; y = 2; x = 4; z = 3; print x; print y; print z;</pre>	4 2 3
<pre>int x,y,z; x = 1+2*3+4*5; y = x+5*8; z= x+3+4+y*x; print(x+y*z);</pre>	<pre>\$t0 = 2 * 3; \$t1 = 1 + \$t0; \$t2 = 4 * 5; \$t3 = \$t1 + \$t2; x = \$t3; ...</pre>	123508
<pre>int alfa,beta; alfa = beta = 3+(alfa+1)*(beta+2); int delta; delta = 2*alfa - beta; print(delta);</pre>	<pre>\$t0 = alfa + 1; \$t1 = beta + 2; \$t2 = \$t0 * \$t1; \$t3 = 3 + \$t2; beta = \$t3; alfa = beta; \$t4 = 2 * alfa; \$t5 = \$t4 - beta; delta = \$t5; print delta;</pre>	5
<pre>int x,y,z,t; t = x = (y=2*4+5) + (z=y=3+1*4); print(t+x+y+z);</pre>	<pre>\$t0 = 2 * 4; \$t1 = \$t0 + 5; y = \$t1; \$t2 = 1 * 4; \$t3 = 3 + \$t2; y = \$t3; z = \$t3; \$t4 = \$t1 + \$t3; x = \$t4; t = \$t4; \$t5 = t + x; \$t6 = \$t5 + y; \$t7 = \$t6 + z; print \$t7;</pre>	54

* Sentencias if

Código fuente (PL)	Resultado de la ejecución
<code>if (1 == 1) print(2);</code>	2
<code>int a; int b; int c; a=1; b=2; c=3; if (a == 0) { a = -1; if (a <= b) { if (c >= a*b) print (c); b = -1; } c = -1; } print(a*b*c);</code>	6
<code>int a; int b; int c; a = 1; b = 2; c = 2; if (a <= b) { if (b < c) print (a*b); print (b*c); } print (a+b+c);</code>	4 5
<code>int a; int b; int c; a = 1; b = 2; c = 3; if (a*b <= b*c) { if ((a=b=c) == (c=b)) { print (a*b*c); } }</code>	27
<code>int a,b,c; a = 1; b = 2; c = 3; if (a < b) print(1); if (a <= b) print(2); if (b > a) print(3); if (b >= b) print(4); if (b == b) print(5); if (a != b) print(6); if (a <= c) { if (a == b) print (a*b*c); if (b == c) { print(b*c); if (b != c) a=2; c=7; if (a<=b) if (b<c) if(c != b) b=5; } a=6; } print(a+b+c+a*b*c);</code>	1 2 3 4 5 6 47

* Sentencias **if-else**

Código fuente (PL)	Resultado de la ejecución
<pre> if (1 < 2) print(1); else print(2); if (3 < 2) print(3); else print(4); </pre>	<p>1 4</p>
<pre> int a,b,c,x; x=1; if (a != b) if (b == c) x = x+2; else x = x+3; print (x); </pre>	1
<pre> int a,b,c,x; x=1; if (a != b) if (b == c) x = x+2; else if (a == c) x = x+3; else x = x+4; print (x); </pre>	1
<pre> int a,b,c,x; x=1; if (a != b) { x = x+2; if (b == c) { x = x+3; } } else if (a == c) { if (c != d) x = x+4; else { x = x+4; if (a == c) x = x+5; } } print (x); </pre>	10
<pre> int a,b,c,d,x; x=1; if (a != b) { x = x+2; if (b == c) x = x+3; if (a != b) x = x-3; else x = x+8; } else if (a == c) { if (c != d) x = x+4; else { x = x+4; if (a == c) x = x+5; else if (b == c) x = x+6; else x = x+7; } } </pre>	10

* Expresiones lógicas. Se consideran aquí los operadores relacionales relacionales: igual, distinto, menor, mayor, mayor o igual, menor o igual. Y los operadores lógicos de negación conjunción y disyunción con evaluación en cortocircuito.

Código fuente (PL)	Resultado de la ejecución
<pre>if (2 == 2 && 3 == 3 && 2!=3) print (2*3); print(9*8-7);</pre>	6 65
<pre>int a; int b; a = 3; b = 12; if (1<=a && a<=5 (b=a) > 10 ! b == a) { a=b; }</pre>	144
<pre>int a,b,c; a = 6; b = 12; c = 18; if (1<=a && a<=5 (b=a) > 10 ! b == a) { a=b; } if (a==b b==c) print(1); if (!a==b b==c) print(2); if (a==b !b==c) print(3); if (!a==b !b==c) print(4); if (a==b && b==c) print(5); if (!a==b && b==c) print(6); if (a==b && !b==c) print(7); if (!a==b && !b==c) print(8); print(a*b);</pre>	1 3 4 7 36
<pre>int a = 2; int b = 3; int c = 4; if (a<(b=a) b<(c=b=a)) { a = 5; if ((a<b b<a) && (b<c c<b)) b = 6; if (!a==5 ! a==b && !b==c) c = 7; }</pre>	8
<pre>int a = 2; int b = 3; int c = 4; if (a<(b=a) !b<(c=b=a)) { a = 3; print(1); } if (a<b !b<c && c<a a>c && c<b && a>b) { a = 4; print(2); } print(a*b*c);</pre>	1 2 16

* Sentencias **while**

Código fuente (PL)	Resultado de la ejecución
<pre>int i; int suma; i=1; while (i<10) { suma = suma + i; i = i+1; } print (suma);</pre>	45
<pre>int a,b,c,suma; while (a=a+1 < 10) while (b=b+1 < 10) { while (c=c+1 < 10) suma = suma+a+b+c; } print(a); print(b); print(c); print(suma);</pre>	10 18 18 63
<pre>int a,b,c,suma; while (a < 10) { a = a+1; while (b < 10) { b = b+2; suma = suma+a+b+c; } c = 15; while (c<10) c=c+1; } print(suma);</pre>	35
<pre>int a,b,c,suma; while (b < 10) { while (a < 10) while (a < 10) { suma = suma+a; a = a+1; print(a+b); } b = b+2; suma = suma+b; c = a; } print(suma);</pre>	1 2 3 4 5 6 7 8 9 10 75
<pre>int a,b,c,suma; while (a < 10) a = a+1; b = b+2; while (b < 10) { b = b+1; while (c < 10) { a = a+1; c = c+3; suma = suma+a+b+c; } } print(suma);</pre>	92

* Sentencias **do-while**

Código fuente (PL)	Resultado de la ejecución
<pre>int i; int suma; i=1; do { suma = suma + i; i = i+1; } while (i<10); print (suma);</pre>	45
<pre>int a,b,c,suma; do do { do suma = suma+a+b+c; while (c=c+1 < 10); } while (b=b+1 < 10); while (a=a+1 < 10); print(a); print(b); print(c); print(suma);</pre>	10 19 28 594
<pre>int a,b,c,suma; do { a = a+1; do { b = b+2; suma = suma+a+b+c; } while (b < 10); c = 15; do c=c+1; while (c<10); } while (a < 10); print(suma);</pre>	413
<pre>int a,b,c,suma; do { do do { suma = suma+a; a = a+1; print(a+b); } while (a < 10); while (a < 10) ; b = b+2; suma = suma+b; c = a; } while (b < 10); print(suma);</pre>	1 2 3 4 5 6 7 8 9 10 13 16 19 22 121
<pre>int a,b,c,suma; do a = a+1; while (a < 10); b = b+2; do { b = b+1; do { a = a+1; c = c+3; suma = suma+a+b+c; } while (c < 10); } while (b < 10); print(suma);</pre>	435

* Sentencias for

Código fuente (PL)	Resultado de la ejecución
<pre>int i; for(i=0; i<5; i=i+1) { print(i); }</pre>	0 1 2 3 4
<pre>int i,j,k,x; for (i=1; i<10; i=i+1) { k=2; for(j=1; j<k; j=j+i) { k = k*i; x = x+i; } } print(x);</pre>	580
<pre>int i,j,k,x; for (i=0; k<=i; i=i/2) for(j=0; k<=j; j=j/2) for(k=1; x<=k; k=k*k+1) x = x+i+j+k; print(x);</pre>	459041
<pre>int i,j,k,x; for (; i<10; i=i+1) { k=2; for(; j<k; j=j+i) { k = k*i; x = x+i; print(x); } } print(x);</pre>	0 1 2 2
<pre>int i,j,k,x; for (i=1; i<10;) { k=2; i=i+1; for(;j<=k;) { k = k*i; x = x+i; j = j+1; } } print(x);</pre>	60

Introducción del tipo real. El uso del tipo real implica reconocer constantes con punto flotante, poder declarar variables de tipo real, comprobar la adecuación de los tipos en las expresiones y asignaciones, generar instrucciones **ctd** diferentes para las operaciones entre enteros y entre reales; y conversión explícita o implícita de tipos en caso necesario. (5 puntos)

NOTA: Desde el punto de vista léxico las constantes reales válidas son iguales a las constantes reales del lenguaje Java, salvo que en PLX no se distingue entre `float` y `double`.

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
<code>print (3.1416);</code> <code>print (3.1416E-2);</code>	<code>print 3.1416;</code> <code>print 3.1416E-2;</code>	3.141600 0.031416
<code>float e;</code> <code>e=2.7172;</code> <code>float pi;</code> <code>pi=3.1416;</code> <code>print (e*pi);</code>	<code>e = 2.7172;</code> <code>pi = 3.1416;</code> <code>t0 = e *r pi;</code> <code>print t0;</code>	8.536355
<code>int ix;</code> <code>ix = 2;</code> <code>int iy;</code> <code>iy = 3;</code> <code>float fx;</code> <code>fx = ix;</code> <code>float fy;</code> <code>fy = iy;</code> <code>print (iy/ix + fx/fy);</code>	<code>ix = 2;</code> <code>iy = 3;</code> <code>fx = (float) ix;</code> <code>fy = (float) iy;</code> <code>t0 = iy / ix;</code> <code>t1 = fx /r fy;</code> <code>t2 = (float) t0;</code> <code>t3 = t2 +r t1;</code> <code>print t3;</code>	1.666667
<code>print((int) (2.0/3.0</code> <code>+(float)3/2));</code>	<code>t0 = 2.0 /r 3.0;</code> <code>t1 = (float) 3;</code> <code>t2 = (float) 2;</code> <code>t3 = t1 /r t2;</code> <code>t4 = t0 +r t3;</code> <code>t5 = (int) t4;</code> <code>print t5;</code>	2
<code>int x;</code> <code>x = 3.1416;</code> <code>print(x);</code>	<code>...</code> <code>error;</code> <code>...</code>	--

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', '\\'', '\\\\', '\\\"').

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 explícita 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 correspondiente al ASCII número 0.

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

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 sumar '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 variables como a constantes de tipo **char**. También se puede convertir explícitamente un carácter en un número entero o viceversa usando los operadores de "casting" (int) y (char) respectivamente.

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
<code>print('z'-'a');</code>	<code>\$t0 = 122 - 97; print \$t0;</code>	25
<code>int x; x = 0 + 'a' + 'b' + 'c'; print(x);</code>	<code>\$t0 = 0 + 97; \$t1 = \$t0 + 98; \$t2 = \$t1 + 99; x = \$t2; print x;</code>	294
<code>int num; num = 'A' + ('Z'-'A') / 2; print(num);</code>	<code>\$t0 = 90 - 65; \$t1 = \$t0 / 2; \$t2 = 65 + \$t1; num = \$t2; print num;</code>	77
<code>char ch1, ch2; ch1 = '+'; ch2 = '*'; print(10 + ch1 - ch2);</code>	<code>ch1 = 43; ch2 = 42; \$t0 = 10 + ch1; \$t1 = \$t0 - ch2; print \$t1;</code>	11
<code>print((int)'a'*(int)'b'); print((char)('a'+1));</code>	<code>\$t0 = 97 * 98; print \$t0; \$t1 = 97 + 1; printc \$t1;</code>	9506 b

* Introducción del tipo **array** de una sola dimensión constante. La dimensión se especifica en la declaración y debe ser constante. En principio no es necesario realizar comprobación de rangos, pero para obtener la máxima calificación en este apartado deberá incluirse en el código generado una serie de instrucciones para detectar este error en tiempo de ejecución (ver ejemplo 5).

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
<pre>int x[10] ; x[1]=3; print (x[1]);</pre>	<pre>x[1] = 3; t0 = x[1]; print t0;</pre>	3
<pre>int x[10] ; x[1] = 1; x[2] = 2; x[3] = 3; int z; z = x[1]+x[2]*x[3]; print(z);</pre>	<pre>x[1] = 1; x[2] = 2; x[3] = 3; t0 = x[1]; t1 = x[2]; t2 = x[3]; t3 = t1 * t2; t4 = t0 + t3; z = t4; print z;</pre>	7
<pre>int i; int a[10]; int suma; suma=0; for(i=0; i<10; i=i+1) { a[i] = i; suma = suma + a[i]*a[i]; } print(suma);</pre>	<pre>suma = 0; i = 0; L0: if (i < 10) goto L1; goto L2; L3: t0 = i + 1; i = t0; goto L0; L1: a[i] = i; t1 = a[i]; t2 = a[i]; t3 = t1 * t2; t4 = suma + t3; suma = t4; goto L3; L2: print suma;</pre>	285
<pre>int p; p=10; int x[p]; x[1]=3; print (x[1]);</pre>	<pre>... error; ...</pre>	--
<pre>int i; int a[3]; for(i=0; i<5; i=i+1) { a[(i+i)/2] = i*i; print (a[i]); }</pre>	<pre>... # Comprobacion de rango if (t2 < 0) goto L4; if (3 < t2) goto L4; if (3 == t2) goto L4; goto L5; L4: error; halt; L5: ...</pre>	<pre>0 1 4 runtime error</pre>

* Introducción de *arrays* de números enteros o reales con comprobaciones de tipos. En este caso se pueden definir arrays tanto de tipo `int` y como de tipo `float`, comprobando que todos los valores del *array* son asignados de forma homogénea.

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución
<code>float x[10] ; x[1]=3.1416; print (x[1])</code>	...	3.141600
<code>float x[10] ; x[1] = 1.1; x[2] = 2.2; x[3] = 3.3; float z; z = x[1]+x[2]+x[3]; print(z);</code>	...	6.600000
<code>float x[10] ; x[1] = 1.1; x[2] = 1+1; x[3] = 3.3; float z; z = x[1]+x[2]+x[3]; print(z);</code>	... t0 = 1+1 t1 = (float) t0; x[2] = t0; ...	6.400000
<code>float x[3] ; x = {1.1, 2.2, 3.3}; print(x[0]+x[1]*x[2]);</code>	...	8.360001
<code>int x[3] ; x = {1.1, 2.2, 3.3}; print(x[0]+x[1]*x[2]);</code>	... # error de tipos error; ...	--
<code>float x[3] ; x = {1.1, 2, 3.3}; print(x[0]+x[1]*x[2]);</code>	... # error de tipos error; ...	--
<code>float x[3]; float pi; pi = 3.1416; x = {1.0, 1.1+2.2, 2.0*pi}; print(x[0]+x[1]+x[2]);</code>	...	10.583200
<code>float x[3] ; float y[3] ; x[0] = 1.1; x[1] = 2.2; x[2] = 3.3; y = x; print(y[0]+y[1]+y[2]);</code>	...	6.600000
<code>float x[3] ; int y[3] ; x[0] = 1.1; x[1] = 2.2; x[2] = 3.3; y = x; print(y[0]+y[1]+y[2]);</code>	... # error de tipos error; ...	--
<code>int x[3] ; float y[3] ; x[0] = 1; x[1] = 2; x[2] = 3; y = x; print(y[0]+y[1]+y[2]);</code>	... # error de tipos error ...	--

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
<pre>char st[3]; st[0]='A';st[1]='B';st[2]='C'; print(st[2]); print(st[1]); print(st[0]);</pre>	...	C B A
<pre>char st[3]; st = { 'P', 'Q', 'R' }; print(st[2]); print(st[1]); print(st[0]);</pre>	...	R P Q
<pre>char st[3] = { 'X', '/', 'Z' }; print(st[2]); print(st[1]); print(st[0]);</pre>	...	Z / X
<pre>char st[2]; st = { 'A', 'B', 'C' }; print(st[0]);</pre>	... error; ...	--
<pre>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);</pre>	...	A B C 3
<pre>char a[3]; char b[5]; a = { 'A', 'B', 'C' }; b = a; print(b[1]);</pre>	...	B
<pre>char a[3]; char b[2]; a = { 'A', 'B', 'C' }; b = a; print(b[1]);</pre>	... # las matrices no son compatibles error; ...	--
<pre>int i; char a[3]; for(i=0; i<5; i=i+1) { a[(i+i)/2] = 'X'; print (a[i]); }</pre>	... 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
<pre>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]);</pre>	...	X C D E
<pre>char st[6]; float mat[6], prod = 1.0; int i; for(i=0; i<st.length; i=i+1) { st[i] = (char)(i+63); mat[i] = (int)st[i]; prod = prod * mat[i]; } print(prod);</pre>		78806409216

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
<pre>int n[3]; n[0] = 65; n[1] = 66; n[2] = 67; print(n);</pre>	...	65 66 67
<pre>int n[3]; n = { 65, 66, 67}; print(n);</pre>	<pre>\$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;</pre>	65 66 67
<pre>print({ 65, 66, 67});</pre>	<pre>print 63; print 64; print 65;</pre>	65 66 67
<pre>char st[3]; st[0] = 'A'; st[1] = 'B'; st[2] = 'C'; print(st);</pre>	...	A B C
<pre>char st[3]; st = { 'A', 'B', 'C' }; print(st);</pre>	<pre>\$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;</pre>	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
<code>print("abc");</code>	<code>writec 97; writec 98; writec 99; writec 10;</code>	<code>abc</code>
<code>print("a\\b");</code>	...	<code>a\b</code>
<code>print("a\"b\"c\"d\");</code>		<code>a"b"c"d"</code>
<code>string a = "abc", b="xyz"; print(a); print(b);</code>		<code>abc xyz</code>
<code>string a = "abc", b, c="xyz"; b=c; print(a); print(b); print(c);</code>		<code>abc xyz xyz</code>

EL CÓDIGO OBJETO (Código de tres direcciones):

El código objeto CTD implementa una maquina abstracta con infinitos registros a los que se accede mediante variables. Todas las variables se considera que están previamente definidas y que su valor inicial es 0. No se diferencia entre números enteros y reales, salvo para realizar operaciones.

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

Instrucción	Acción
<code>x = a ;</code>	Asigna el valor de a en la variable x
<code>x = a + b ;</code>	Suma los valores de a y b, y el resultado lo asigna a la variable x
<code>x = a - b ;</code>	Resta los valores de a y b, y el resultado lo asigna a la variable x
<code>x = a * b ;</code>	Multiplica los valores de a y b, y el resultado lo asigna a la variable x
<code>x = a / b ;</code>	Divide (div. entera) los valores de a y b, y el resultado lo asigna a la variable x
<code>x = a +r b ;</code>	Suma de dos valores reales
<code>x = a -r b ;</code>	Resta de dos valores reales
<code>x = a *r b ;</code>	Multiplicación de dos valores reales
<code>x = a /r b ;</code>	Division de dos valores reales
<code>x = (int) a ;</code>	Convierte un valor real a, en un valor entero, asignándose a la variable x
<code>x = (float) a ;</code>	Convierte un valor entero a, en un valor real, asignándose a la variable x
<code>x = y[a] ;</code>	Obtiene el a-esimo valor del array y, asignando el contenido en x
<code>x[a] = b ;</code>	Coloca el valor b en la a-esima posición del array x
<code>x = *y ;</code>	Asigna a x el valor contenido en la memoria referenciada por y.
<code>*x = y ;</code>	Asigna el valor y en la posición de memoria referenciada por x.
<code>x = &y ;</code>	Asigna a x la dirección de memoria en donde esta situado el obteto y.
<code>goto l ;</code>	Salto incondicional a la posición marcada con la sentencia "label l"
<code>if (a == b) goto l ;</code>	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
<code>if (a != b) goto l ;</code>	Salta a la posición marcada con la sentencia "label l", si y solo si el valor de a es distinto que el valor de b
<code>if (a < b) goto l ;</code>	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.
<code>l:</code>	Indica una posición de salto.
<code>label l ;</code>	Indica una posición de salto. Es otra forma sintáctica equivalente a la anterior.
<code>function f :</code>	Indica una posición de salto para comienzo de una función.
<code>end f ;</code>	Indica el final del código de una función.
<code>param n = x;</code>	Indica que x debe usarse como parámetro n-simo en la llamada a la próxima función.
<code>x = param n ;</code>	Asigna a la variable x el valor del parámetro n-simo definido antes de la llamada a la función.
<code>call f ;</code>	Salto incondicional al comienzo de la función f. Al alcanzar la sentencia return el control vuelve a la instrucción inmediatamente siguiente a esta
<code>gosub l ;</code>	Salto incondicional a la etiqueta f. Al alcanzar la sentencia return el control vuelve a la instrucción inmediatamente siguiente a esta
<code>return ;</code>	Salta a la posición inmediatamente siguiente a la de la instrucción que hizo la llamada (call f) o (gosub l)
<code>write a ;</code>	Imprime el valor de a (ya sea entero o real)
<code>writec a ;</code>	Imprime el carácter Unicode correspondiente al número a
<code>print a ;</code>	Imprime el valor de a, y un salto de línea
<code>printc a ;</code>	Imprime el carácter Unicode correspondiente al número a, y un salto de línea
<code>error ;</code>	Indica una situación de error, pero no detiene la ejecución.
<code>halt ;</code>	Detiene la ejecución. Si no aparece esta instrucción la ejecución se detiene cuando se alcanza la última instrucción de la lista.
<code># ...</code>	Cualquier línea que comience con # se considera un comentario.
<code>. <nombre fichero></code>	Incluye el contenido del fichero indicado, buscándolo en el directorio actual.

En donde **a**, **b** representan tanto variables como constantes enteras, **x**, **y** representan siempre una variable, **n** representa un numero entero, **l** representa una etiqueta de salto y **f** un nombre de función.

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,. Para compilar y ejecutar un programa en lenguaje PLX, pueden utilizarse las instrucciones

	Linux
Compilación	<code>./plx prog.plx prog.ctd</code>
Ejecución	<code>./ctd prog.ctd</code>

El programa a enviar para su corrección automática debe probarse previamente y comparar los resultados de la ejecución anterior. 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.

	Linux
Compilación	<code>java PLXC prog.plx prog.ctd</code>
Ejecución	<code>./ctd prog.ctd</code>

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 `plx` 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 intérprete 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	<code>./plx prog</code>

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 `plx`, que implementa el compilador de `plx` y que sirve para comparar los resultados obtenidos, incluye una opción `-c` para generar comentarios que pueden ayudar a identificar el código generado para cada línea del programa fuente:

```
./plx -c prog.ctd
```

4. El programa `ctd`, interprete del código intermedio, tiene una opción `-v` que sirve para mostrar la ejecución del código de tres direcciones paso a paso y que pueden ayudar en la depuración de errores:

```
./ctd -v prog.ctd
```

5. 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.
6. 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.