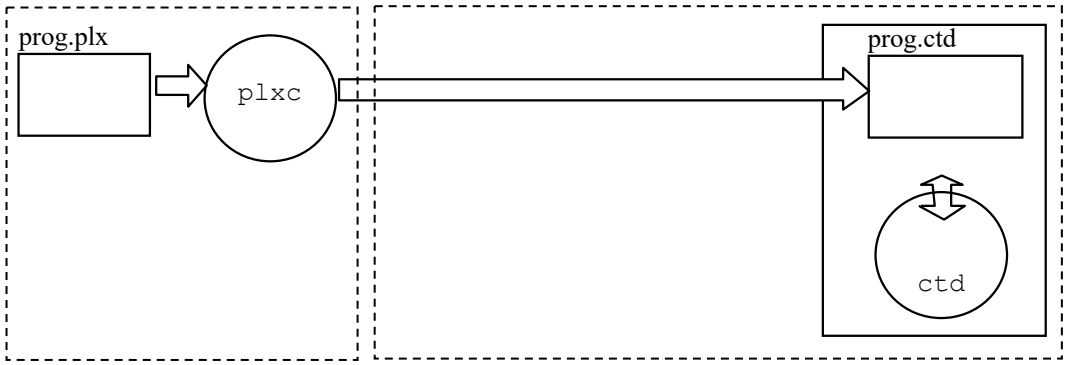
Práctica Principal de Procesadores de Lenguajes

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 `plxc`, (equivalente a "`java PLXC`"), según las instrucciones indicadas en los siguientes enunciados.

COMPILADOR DE PRUEBA

Se proporcionan versiones compiladas de "plxc" (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
`int x;` `print (x);`	`print x;`	**0**
`print (x);`	… `error;` `# variable no declarada` …	--

* 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
`int x,y;` `x=y=1;` `print (x+y);`	`y=1;` `x=y;` `t0=x+y;` `print t0;`	**2**
`int x,y,x;` `x=y=1;` `print (x+y);`	… `error;` `# variable ya declarada` …	--

* 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
`int x=1;` `print (x);`	`x=1;` `print x;`	**1**
`int x=1, y=1+2*3, z=0;` `print (x+y);`	`x=1;` `t0=2*3;` `t1=1+t0;` `y=t1;` `z=0;` `t2=x+y;` `print t2;`	**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
print (1+2*3);	$t0 = 2 * 3; $t1 = 1 + $t0; print $t1;	**7**
print(1 - -201*3 + 40*50 - -6);	$t0 = -201; $t1 = $t0 * 3; $t2 = 1 - $t1; $t3 = 40 * 50; $t4 = $t2 + $t3; $t5 = -6; $t6 = $t4 - $t5; print $t6;	**2610**
print(1*2+((3+2)*4)/2);	$t0 = 1 * 2; $t1 = 3 + 2; $t2 = $t1 * 4; $t3 = $t2 / 2; $t4 = $t0 + $t3; print $t4;	**12**
print(-64 / -16 / -2 * -4 / -2);	$t0 = -64; $t1 = -16; $t2 = $t0 / $t1; $t3 = -2; $t4 = $t2 / $t3; $t5 = -4; $t6 = $t4 * $t5; $t7 = -2; $t8 = $t6 / $t7; print $t8;	**-4**
print(3/-2 * ((4+5) - -9/4));	$t0 = -2; $t1 = 3 / $t0; $t2 = 4 + 5; $t3 = -9; $t4 = $t3 / 4; $t5 = $t2 - $t4; $t6 = $t1 * $t5; print $t6;	**-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
int x; **int y;** **int zeta;** **int p;** **int q;** **x=6;** **y=2;** **{** **p = q = 2;** **zeta = p+q;** **}** **print (zeta+x/y);**	x = 6; y = 2; q = 2; p = q; $t0 = p + q; zeta = $t0; $t1 = x / y; $t2 = zeta + $t1; print $t2;	**7**
int x; **int y;** **int z;** **{** **x=1;** **{** **y = 2;** **{** **x = 4;** **}** **z = 3;** **}** **print (x);** **print (y);** **print (z);**	x = 1; y = 2; x = 4; z = 3; print x; print y; print z;	**4** **2** **3**
int x,y,z; **x = 1+2*3+4*5;** **y = x+5*8;** **z= x+3+4+y*x;** **print(x+y*z);**	$t0 = 2 * 3; $t1 = 1 + $t0; $t2 = 4 * 5; $t3 = $t1 + $t2; x = $t3; …	**123508**
int alfa,beta; **alfa = beta =** **3+(alfa+1)*(beta+2);** **int delta;** **delta = 2*alfa - beta;** **print(delta);**	$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;	**5**
int x,y,z,t; **t = x = (y=2*4+5) + (z=y=3+1*4);** **print(t+x+y+z);** **// Ver NOTA**	$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;	**54**

NOTA: La expresión: **`t = x = (y=2*4+5) + (z=y=3+1*4);`** se evalúa como **`t = x = 13 + 7;`**
Por lo que las variables al final toman estos valores: **`z=20; y=7; z=7; t= 20;`**

* Sentencias **if**

Código fuente (PL)	Resultado de la ejecución
`if ( 1 == 1 ) print(2);`	**2**
`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);`	**6**
`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);`	**4** **5**
`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);` `  }` `}`	**27**
`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);`	**1** **2** **3** **4** **5** **6** **47**

* Sentencias **if-else**

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

* Sentencias **while**

Código fuente (PL)	Resultado de la ejecución
`int i;` `int suma;` `i=1;` `while (i<10) {` `  suma = suma + i;` `  i = i+1;` `}` `print (suma);`	**45**
`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);`	**10** **18** **18** **63**
`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);`	**35**
`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);`	**1** **2** **3** **4** **5** **6** **7** **8** **9** **10** **75**
`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);`	**92**

* Sentencias **do-while**

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

* Sentencias **for**

Código fuente (PL)	Resultado de la ejecución
`int i;` `for(i=0; i<5; i=i+1) {` `  print(i);` `}`	**0** **1** **2** **3** **4**
`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);`	**580**
`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);`	**459041**
`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);`	**0** **1** **2** **2**
`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);`	**60**

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
`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 = a +r b ;`	Suma de dos valores reales
`x = a -r b ;`	Resta de dos valores reales
`x = a *r b ;`	Multiplicación de dos valores reales
`x = a /r b ;`	Division de dos valores reales
`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-esimo valor del array y, asignando el contenido en x
`x[a] = b ;`	Coloca el valor b en la a-esima posición del array x
`x = *y ;`	Asigna a x el valor contenido en la memoria referenciada por y.
`*x = y ;`	Asigna el valor y en la posición de memoria referenciada por x.
`x = &y ;`	Asigna a x la dirección de memoria en donde esta situado el obteto y.
`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 distinto 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.
`function f :`	Indica una posición de salto para comienzo de una función.
`end f ;`	Indica el final del código de una función.
`param n = x;`	Indica que x debe usarse como parámetro n-simo en la llamada a la próxima función.
`x = param n ;`	Asigna a la variable x el valor del parámetro n-simo definido antes de la llamada a la función.
`call f ;`	Salto incondicional al comienzo de la función f. Al alcanzar la sentencia return el control vuelve a la instruccion inmediatamente siguiente a esta
`gosub l ;`	Salto incondicional a la etiqueta f. Al alcanzar la sentencia return el control vuelve a la instruccion inmediatamente siguiente a esta
`return ;`	Salta a la posición inmediatamente siguiente a la de la instruccón que hizo la llamada (call f) o (gosub l)
`write a ;`	Imprime el valor de a (ya sea entero o real)
`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 # se considera un comentario.
`.` *<nombre fichero>*	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	`./plxc prog.plx prog.ctd`
Ejecución	`./ctd prog.ctd`

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	`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 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	`./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 `plxc`, 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:

```
./plxc –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.