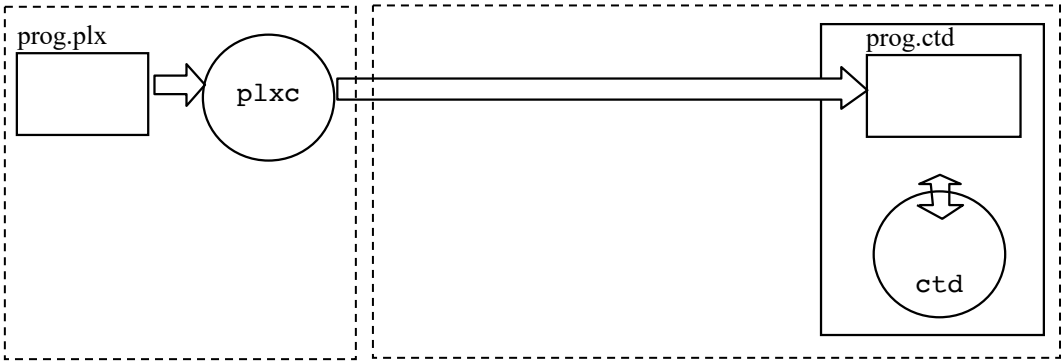
Práctica Principal de Procesadores de Lenguajes

Esta práctica consiste en la implementación mediante JFlex y Cup de una extensión del compilador del lenguaje PLX. La aportación de esta extensión abunda en el uso de variables de tipo boolean, la introducción de nuevos operadores condicionales y algunas nuevas sentencias de control. Se suponen previamente implementadas la funcionalidad básica del lenguaje PLX y sus sentencias de control. 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.

Inicialización de variables booleanas. (`boolinit*.plx`)

Las variables booleanas se pueden inicializar en la declaración mediante expresiones booleanas al igual que en la asignación. Si una variable booleana no se inicializa su valor por defecto es **`false`**.
También se pueden declarar múltiples variables booleanas e inicializarlas en una sola línea de declaración.

Código fuente (PLX)	Código intermedio (CTD)	Resultado de la ejecución

`boolean b = true;`
`if (b) {`
`  print(1);`
`} else {`
`  print(2);`
`}`
`print(3);`
		`1`
		`3`

`boolean p = true;`
`boolean q = false;`
`if (p) {`
`  print(1);`
`  if (q) {`
`  print(2);`
`  } else {`
`  print(3);`
`  }`
`} else {`
`  print(4);`
`}`
`print(5);`
		`1`
		`3`
		`5`

`int x=12;`
`int y=15;`
`int z=20;`
`boolean p = x<15 && y<20 || z<20;`
`boolean q = x<15 && y<15 || z<25;`
`if (p && q)`
`  print(1);`
`if (p || q)`
`  print(2);`
`if (p && !q)`
`  print(3);`
`if (!p || q)`
`  print(4);`
`print(5);`
		`1`
		`2`
		`4`
		`5`

`boolean p=true,q,r=false;`
`int i=0;`
`for(p=true; i<3; i=i+1) {`
`  for(q=true; q; q=false) {`
`  r = !r;`
`  print(i);`
`  }`
`}`
`if(r) print(4);`
`else print(5);`
		`0`
		`1`
		`2`
		`4`

`int x=1;`
`boolean bx = x = 0;`
`if(!bx) {`
`  print(1);`
`}`
`print(x);`
	`..`
	`error;`
	`# boolean bx = x == 0`
		…

NOTA: Ver más ejemplos en los ficheros auxiliares

Conversión explicita de variables booleanas enteros y viceversa. (`boolcast*.plx`)

Las variables booleanas se pueden convertir en variables enteras o mediante el correspondiente "casting". Una variable booleana que tome el valor verdadero se convierte en el número 1; y las variables que toman el valor false, se transforman en el valor 0.

Por el contrario, cualquier variable entera puede convertirse en una variable booleana, aplicando la operación de "casting" correspondiente. Si la variable entera tiene un valor 0, la variable booleana tomara el valor false, y si es distinto de cero (sea cual sea su valor), tomara el valor true.

Código fuente (PLX)	Resultado de la ejecución

`boolean p;`
`int x;`
`x = (int) (p || !p);`
`print(x);`
`x = (int) (p && !p);`
`print(x);`
	`1`
	`0`

`boolean p,q;`
`p = false;`
`int x;`
`x = 2;`
`p = (boolean) (x*x-x+x);`
`q = (boolean) (x+x-x*x);`
`if (p) print(1);`
`if (q) print(2);`
`if (p || q) print(3);`
`print(0);`
	`1`
	`3`
	`0`

`boolean p,q;`
`int x,y;`
`x=1; y=2;`
`p = x<=y;`
`q = (int)p <= (int)q && x<y ;`
`if(p) print(1); else print(2);`
`if(q) print(3); else print(4);`
	`1`
	`4`

`boolean p,q;`
`int s1,s2;`
`int i;`
`i=0;`
`p=true;`
`for(i=0;i<10;i=i+1){`
`  s1 = s1 + (int) p;`
`  s2 = s2 + (int) q;`
`}`
`print(s1);`
`print(s2);`
	`10`
	`0`

`boolean alfa,beta;`
`int x,y;`
`int i;`
`i=0;`
`x=12;`
`y=15;`
`for(i=0; i<x; i=i+1){`
`  alfa = (boolean) (i*i);`
`  beta = (boolean) (i-1);`
`  y = y + (int) (boolean) i * i - (int) (boolean) i - 1;`
`}`
`print(y);`
`if (alfa) print(x); else print(y-x);`
	`58`
	`12`

Operador booleano de doble implicación. (`dimplica*.plx`)

Implementar el operador booleano correspondiente a la doble implicación. El resultado de la doble implicación es verdadero si ambas expresiones se evalúan al mismo valor lógico, es decir, el resultado es verdadero si y solo si, ambas expresiones son verdaderas o ambas son falsas.
Este operador es asociativo por la derecha, y tiene menor prioridad que cualquier otro operador booleano.

Código fuente (PLX)	Resultado de la ejecución

`if (false <--> true)`
`  print(1);`
`else`
`  print(2);`
`if (false <--> false)`
`  print(3);`
`else`
`  print(4);`
`print(0);`
	2
	3
	0

`if ( 1<2 <--> 2<3 )`
`  print(1);`
`if ( 1<2 <--> 2<1 )`
`  print(2);`
`if ( 2<1 <--> 2<3 )`
`  print(3);`
`if ( 2<1 <--> 4<3 )`
`  print(4);`
`print(0);`
	1
	4
	0

`int a;`
`a=1;`
`int  b;`
`b=2;`
`int c;`
`c=3;`
`if ((a<b && b<c) <--> a<c) {`
`  print(a+b*c);`
`}`
`print(0);`
	7
	0

`oolean p,q;`
`if (p <--> q <--> !q) {`
`  print(1);`
`  p = true;`
`  if (p <--> q <--> q) {`
`  print(2);`
`  }`
`}`
`print(0);`
	1
	2
	0

`if (1<2)`
`  print(1);`
`if (1<2 <--> 2<3)`
`  print(2);`
`if (1<2 <--> 2<3 <--> 3<4)`
`  print(3);`
`if (1<2 <--> 2<3 <--> 3<4 || 4<5)`
`  print(4);`
`if (2<1 <--> 1<2 || 3<4)`
`  print(5);`
`if (2<1 <--> 3<2 && 1<2 <--> 2<3)`
`  print(6);`
`if (2<1 <--> 3<2 <--> 4<3)`
`  print(7);`
`if (2<1 <--> 3<2 <--> 4<3 && 5<4)`
`  print(8);`
`print(0);`
	1
	2
	3
	4
	6
	0

Condición *exists* con variable booleanas. (`existsbool*.plx`)

Al igual que la condición *forall*, la condición *exists*, comprueba si la expresión base es verdadera para todos los posibles valores de la variable booleana, pero para que la la expresión *exists* sea verdadera solo es necesario que lo sea para un caso. Por ejemplo, la expresión **`exists p,!p`** es una expresión verdadera, ya que si p toma el valor **`false`**, la condición *exists* se cumple. La expresión **`exists p, p||!p`** también es una expresión verdadera, y la expresión **`exists p, p&&!p`** es falsa, ya que para todo valor de **`p`**, no se cumple la condición.
Las expresiones *exists* puede anidarse recursivamente por la derecha, con lo que se pueden probar condiciones que combinan todos los posibles valores de dos o más variables booleanas

Código fuente (PLX)	Resultado de la ejecución

`boolean p;`
`if (exists p , p)`
`  print(1);`
`if (exists p , !p)`
`  print(2);`
`if (exists p , p && !p) {`
`  print(3);`
`}`
`print(0);`
	1
	2
	0

`boolean p,q;`
`if (exists p, p || (!q && !p) ) {`
`  print(1);`
`}`
`if (exists p, p && (!q && p) ) {`
`  print(2);`
`}`
`print(0);`
	1
	2
	0

`boolean p,q;`
`if (exists p , exists q, p <--> q <--> p)`
`  print(1);`
`if (exists p , exists q, p <--> p <--> q)`
`  print(2);`
`if (exists q , exists p, !(p <--> p) <--> q)`
`  print(3);`
`print(4);`
	1
	2
	3
	4

`boolean p,q;`
`int i;`
`while (i<5  && exists p, (p || i/5 < 1) ) {`
`  print(i);`
`  i = i+1;`
`}`
`while (exists p, exists q, (!p && q) && i>0) {`
`  p = !p;`
`  print(i);`
`  i = i-1;`
`}`
`print(0);`
	0
	1
	2
	3
	4
	5
	4
	3
	2
	1
	0

`boolean p,q;`
`if (exists p, exists q, p && q <--> p||q)`
`  print(1);`
`if ((exists p, exists q, p&&q <--> p||q) && (exists p, p&&!p) )`
`  print(2);`
`if (exists p, exists q, p || q <--> p && q )`
`  print(3);`
`print(0);`
	1
	3
	0

Condición *exists* para variables numéricas. . (`existsint*.plx`)

Implementar una nueva condición lógica que exprese que una variable entera en un determinado rango cumple en algún caso una determinada condición. Es decir, a partir de una condición base, la condición *exists* se satisface si para alguno los posibles valores de una variable entera se cumple la condición base.
Esta condición es similar a la condición *exists* con variables lógicas, salvo que usa variables enteras.
Para definir el rango de elementos para los que se va a probar la condición, se usa la sintaxis *from.. to step*, similar a la de la sentencia *for* del lenguaje PASCAL. Por defecto si no se especifica el valor de salto (*step*) se toma el valor 1. Ver ejemplos
Al termino de la ejecución de la condición el valor de la variable entera se habrá modificado respecto al que tenia anteriormente. Si la condición se cumple, el valor de la variable será precisamente el primer valor que hace que se cumpla la condición base, y si no se cumple tomará el valor final del rango más el salto.
Al igual que en el caso de variables booleanas, la condición *exists* se puede anidar recursivamente por la derecha, y/o combinar con otras expresiones lógicas

Código fuente (PLX)	Resultado de la ejecución

`int x;`
`if (exists x from 0 to 10 step 2 , x-2*(x/2) == 0) {`
`  print(1);`
`}`
`if (exists x from 0 to 10 step 3 , x-2*(x/2) == 0) {`
`  print(2);`
`}`
`print(0);`
	1
	2
	0

`int x;`
`if (exists x from 1 to 10, x*x < 0) {`
`  print(1);`
`}`
`if (exists x from 1 to 5, x*x > 10) {`
`  print(2);`
`}`
`print(0);`
	2
	0

`int x;`
`if (exists x from 1 to 10, x*x > 10 && x*x < 25) {`
`  print(2);`
`}`
`print(x);`
`print(0);`
	2
	4
	0

`int x,y;`
`if (exists x from 1 to 10, exists y from 1 to 10, x*y > 50) {`
`  print(x); print(y);`
`}`
`if (exists x from 2 to 7, exists y from 1 to 5, x-y >= 0) {`
`  print(x); print(y);`
`}`
`print(0);`
	6
	9
	2
	1
	0

`int x,y;`
`x=0; y=10;`
`if (x<y && exists x from 1 to 10, x <= y) {`
`  print(1);`
`}`
`print(x); print(y);`
`if (x<y && exists x from 1 to 9, x <= y) {`
`  print(2);`
`}`
`print(0);`
	1
	1
	10
	2
	0

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.

IMPLEMENTACIÓN DE LAS VARIABLES BOOLEANAS:

Al implementar las variables booleanas sobre la implementación anterior de la práctica PLX-CORE se produce un conflicto **Reducción-Reducción** entre las reglas de *Expresión* → IDENT y *Condición* → IDENT en las siguientes circunstancias:

Expresion → *Expresion* + *Expresion*
| …
| *IDENT*
| …
Condicion → *Expresion* **==** *Expresion*
| …
| *IDENT*
| *Condicion* **&&** *Condicion*
| …

Para resolver este conflicto hay al menos dos estrategias alternativas distintas:

Solución 1: Unificar las reglas de Expresión y Condición, y generando distinto código según un atributo que contenga el tipo de cada expresión, con lo que a nivel sintáctico la gramática sería:

Expresion → *Expresion* + *Expresion*
| …
| *IDENT*
| …
| *Expresion* **==** *Expresion*
| …
| *Expresion* **&&** *Expresion*
| …

Solución 2: Devolver desde el analizador léxico tokens diferentes en el caso de identificadores booleanos y el resto de los tipos, de manera que no se produzca el conflicto: (Aunque desde el punto de vista lexicográfico los lexemas son iguales, se pueden diferenciar los tokens devueltos según el tipo declarado previamente)

Expresion → *Expresion* + *Expresion*
| …
| *IDENT*
| …
Condicion → *Expresion* **==** *Expresion*
| …
| *IDENT_BOOL*
| *Condicion* **&&** *Condicion*
| …