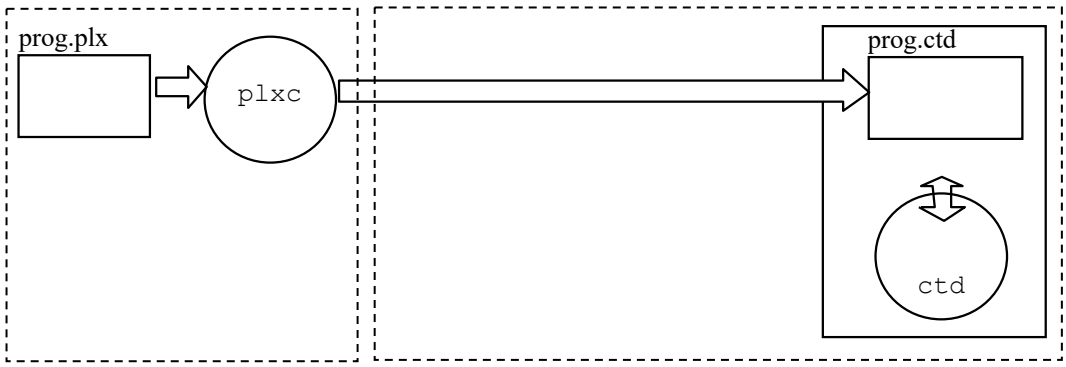
Práctica Principal de Procesadores de Lenguajes

Esta práctica consiste en la implementación mediante JFlex y Cup de una extensión del compilador del lenguaje PLX. 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.

Operadores de bits de desplazamiento. (`bitlr*.plx`)

Se introducen los operadores binarios de desplazamiento de bits a derecha >> y a izquierda <<, que se aplican solamente a expresiones enteras, y cuyo segundo operando debe ser siempre un número entero mayor o igual que 0 constante. Este operador equivale a dividir o multiplicar por 2 tantas veces como diga el argumento. Por ejemplo 16 << 2, equivale a multiplicar 16 por 4, es decir, tiene el valor 64.

Estos operadores son asociativos por la izquierda, tienen menor prioridad de operación que los operadores aritméticos binarios ordinarios. (Al igual que en los lenguajes C y JAVA).

NOTA: Si el desplazamiento viene dado por una constante, puede implementarse de forma más simple, aunque para superar todas las pruebas en las que el desplazamiento viene dado por una variable o una expresión es necesaria una implementación algo más compleja. (ver últimos ejemplos)

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

`print(16>>2);`	`$t0 = 16 / 2;`
	`$t0 = $t0 / 2;`
	`print $t0;`
		4

`print(16<<2);`	`$t0 = 16 * 2;`
	`$t0 = $t0 * 2;`
	`print $t0;`
		64

`int x;`
`x = 16;`
`x = x << 2;`
`print(x);`
	`x = 16;`
	`$t0 = x * 2;`
	`$t0 = $t0 * 2;`
	`x = $t0;`
	`print x;`
		64

`int x;`
`x = 16;`
`x = x << 3 << 2;`
`print(x);`
	`x = 16;`
	`$t0 = x * 2;`
	`$t0 = $t0 * 2;`
	`$t0 = $t0 * 2;`
	`$t1 = $t0 * 2;`
	`$t1 = $t1 * 2;`
	`x = $t1;`
	`print x;`
		512

`print( 25 >> 2 >> 2 << 3 );`	`$t0 = 25 / 2;`
	`$t0 = $t0 / 2;`
	`$t1 = $t0 / 2;`
	`$t1 = $t1 / 2;`
	`$t2 = $t1 * 2;`
	`$t2 = $t2 * 2;`
	`$t2 = $t2 * 2;`
	`print $t2`
		8

`int x;`
`x=1+1;`
`print (16 >> x);`
	`$t0 = 1 + 1;`
	`x = $t0;`
	`$t1 = 16;`
	`$t2 = 1;`
	`L0:`
	`if(x<$t2)goto L1;`
	`$t1 = $t1 / 2;`
	`$t2 = $t2 + 1;`
	`goto L0;`
	`L1:`
	`print $t1;`
		4

`print(8+8 << 1+1);`	`$t0 = 8 + 8;`
	`$t1 = 1 + 1;`
	`$t2 = $t0;`
	`$t3 = 1;`
	`L0:`
	`if($t1<$t3)goto L1;`
	`$t2 = $t2 + $t2;`
	`$t3 = $t3 + 1;`
	`goto L0;`
	`L1:`
	`print $t2;`
		64

Sentencia REPEAT-TIMES. (`repeattimes*.plx`)

Añadir al lenguaje una sentencia `repeat` en donde se especifique exactamente el número de repeticiones que se van a realizar. Este número puede ser constante o variable, y en el caso de que sea variable, debe comprobarse su valor cada vez que se ejecuta la instrucción, deteniendo su ejecución si el contador del número de ejecuciones supera este valor. Si el número de repeticiones es igual o menor que 1, la sentencia solo se ejecutará una vez

La sentencia repeat-times, puede anidarse, e incluir en el cuerpo una o más sentencias de cualquier tipo: `if-then-else, for, do-while, while`, etc.

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

`repeat print(1); 3 times;`	`$t0 = 1;`
	`L0:`
	`print 1;`
	`$t0 = $t0 + 1;`
	`if (3 < $t0) goto L1;`
	`goto L0;`
	`L1:`
		1
		1
		1

`int x;`
`x=2;`
`repeat print(x); x+x times;`
	`x = 2;`
	`$t0 = 1;`
	`L0:`
	`print x;`
	`$t1 = x + x;`
	`$t0 = $t0 + 1;`
	`if ($t1 < $t0) goto L1;`
	`goto L0;`
	`L1:`
		2
		2
		2
		2

`repeat`
`repeat`
`repeat print(7); 2 times;`
`3 times;`
`-1 times;`
	…	**7**
		7
		7
		7
		7
		7

`int x;`
`x=6;`
`repeat {`
`x = x-1;`
`print(x);`
`} x times;`
	…	**5**
		4
		3

`int x;`
`int y;`
`repeat {`
`x = 2;`
`repeat {`
`print(x+y);`
`y=x+1;`
`} x times;`
`if (x<y) y=x;`
`} x times;`
	…	**2**
		5
		4
		5

Sentencia SELECT. (`select*.plx`)

La sentencia *select* permite elegir el valor de una variable que hace que se cumpla una determinada condición. (Ver ejemplos).
En caso de que múltiples valores de la variable hagan que se verifique la condición se tomará por defecto el primero de ellos, (opción **first** por defecto) salvo que se indique lo contrario con la opción **last**.
Por defecto se considera que la variable se incrementa en intervalos de 1, aunque puede modificarse con la opción **step**.
Si ninguna de los posibles valores de la variable consigue satisfacer la condición, por defecto se devuelve el valor 0, aunque este valor puede modificarse con a opción **default.**
Los valores que se introducen tras las palabras reservadas **from**, **to**, **step** y **default**, pueden ser expresiones de tipo entero.

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

`int x;`
`select x from 1 to 10`
`  where ( x+x == x*x );`
`print(x);`
		2

`int x;`
`select first x from 1 to 10`
`  where ( x >= 5 );`
`print(x);`
		5

`int x;`
`select last x from 1 to 10`
`  where ( x >= 5 );`
`print(x);`
		10

`int x;`
`select x from 1 to 10 step 3`
`  where ( x >= 5 );`
`print(x);`
		7

`int x;`
`select x from 1 to 10 default 11`
`  where(x*x==-1);`
`print(x);`
		11

`int x;`
`select x from 1 to 10 step 3 default 11`
`  where(x*x==-1);`
`print(x);`
		11

`int x,y;`
`select first x from 1 to 10`
`  select first y from 1 to 10`
`  where (x+2*y+1 == x*y);`
`print(x);`
`print(y);`
		3
		4

`int x,y;`
`select last x from 1 to 10`
`  select first y from 1 to 10`
`  where (x+2*y+1 == x*y );`
`print(x);`
`print(y);`
		5
		2

`int x,y,z;`
`select last x from 1 to 10`
`  select y from 1 to 10`
`  select z from 1 to 10`
`  where ( x+y+z == x*y*z);`
`print(x);`
`print(y);`
`print(z);`
		3
		1
		2

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 ;`	División 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 está 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 instrucción 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 instrucción 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.