Extensión de la Práctica Principal de Procesadores de Lenguajes (Examen febrero 2020)

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-2020 es una extensión del lenguaje PLX que se describe como prácticas de la asignatura, pero en esta versión extendida se requieren algunas funciones adicionales. Se presupone que todos los elementos del lenguaje PLX anteriores están presentes en el lenguaje PLX-2020 y que no se modifica su funcionamiento al incluir los nuevos elementos de esta extensión, aunque obviamente en este examen solo se evaluaran los casos que se describen en este enunciado, minimizando en la medida de lo posible el uso de funcionalidades anteriores.

EL CÓDIGO FUENTE (Lenguaje PLX):

El lenguaje PLX-2020 incluye todas las sentencias definidas en el lenguaje PLX anterior y algunas más. Asimismo, se modifica ligeramente el lenguaje intermedio CTD, de manera que soporte algunas instrucciones adicionales.

ASPECTOS LEXICOS

Todos los aspectos léxicos relacionados con este lenguaje se resuelven, a menos que se especifique lo contrario, de la misma manera que en el lenguaje JAVA.

Cualquier duda que surja al implementar, y que no estuviera suficientemente clara en este enunciado debe resolverse de acuerdo a las especificaciones del lenguaje JAVA.

COMPILADOR DE PRUEBA

Se proporciona una versión compilada del compilador, que puede usarse como referencia para la generación de código. No es necesario que el código generado por el compilador del alumno, sea exactamente igual al generado por el compilador de prueba, basta con que produzca los mismos resultados al ejecutarse para todas las entradas.

El compilador de prueba se entrega solamente a titulo orientativo. Si hubiese errores en el compilador de prueba, prevalecen las especificaciones escritas en este enunciado. Estos posibles errores en ningún caso eximen al alumno de realizar una implementación correcta.

Funciones.

Se añade la posibilidad de declarar funciones y realizar la llamada desde el código principal. Las funciones se declaran en la cabecera del código. Puede haber cero, una o mas funciones. Sintácticamente las funciones se declaran de forma similar a las funciones en Java o en C.

Las funciones pueden declararse en cualquier punto del código, pero la ejecución del código de PLX no incluirá la ejecución del código de la función hasta que se realice la llamada.

La forma mas sencilla de funciones son las que no tienen ningún argumento y no devuelven ningún valor, (en este caso hay que declararlas de tipo **void**). Puede haber funciones que tengan varios argumentos, pero solo pueden devolver un único valor. Para las funciones que devuelven **void** puede usarse la sentencia **return** sin argumentos.

Una función puede tener mas de una sentencia **return**. Una vez que se alcanza esta sentencia, la ejecución de la función termina, y no se ejecuta ninguna sentencia situada detrás. (No es necesario comprobar si hay o no código inaccesible).

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

`void f() {`
`  print(2);`
`}`
`print(1);`
`f();`
`print(3);`
	`function f:`
	`  print 2;`
	`end f;`
	`  print 1;`
	`  call f;`
	`  print 3;`
		1
		2
		3

`print(1);`
`void dos() {`
`  print(2);`
`}`
`dos();`
`print(3);`
	`  print 1;`
	`function dos:`
	`  print 2;`
	`end dos;`
	`  call dos;`
	`  print 3;`
		1
		2
		3

`void tres(int x) {`
`  print(x);`
`  return;`
`}`
`print(1);`
`tres(2);`
`print(3);`
	`function tres:`
	`  x_2 = param 1;`
	`  print x_2;`
	`end tres;`
	`  print 1;`
	`  param 1 = 2;`
	`  call tres;`
	`  print 3;`
		1
		2
		3

`void f(int x) {`
`  print(x);`
`}`
`void g(int y) {`
`  print(y);`
`}`
`print(1);`
`f(2);`
`g(3);`
	`function f:`
	`  x_2 = param 1;`
	`  print x_2;`
	`end f;`
	`function g:`
	`  y_4 = param 1;`
	`  print y_4;`
	`end g;`
	`...`
		1
		2
		3

`int f(int x) {`
`  return 2*x-8;`
`}`
`print(1);`
`int a;`
`a=f(5);`
`print(a);`
`print(3);`
		1
		2
		3

`int f(int x, int y) {`
`  return 2*x-y;`
`}`
`print(1);`
`int a;`
`a=f(5,8);`
`print(a);`
`print(3);`
		1
		2
		3

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

`int f(int x,int y,int z,int t) {`
`  return x-y+z-t;`
`}`
`print(1);`
`int a;`
`a=f(9,8,7,6);`
`print(a);`
`print(3);`
		1
		2
		3

`void f(int x) {`
`  print(x);`
`  return;`
`}`
`void g(int y) {`
`  return;`
`  print(y);`
`}`
`print(1);`
`f(2);`
`g(3);`
		1
		2

`int abs(int a) {`
`  if (a<0)`
`  return -a;`
`  return a;`
`}`
`print(4);`
`int x;`
`x = abs(-5);`
`print(x);`
`x = abs(6);`
`print(x);`
		4
		5
		6

`int f(int x) {`
`  print(x);`
`  return -x;`
`}`
`print(1);`
`int a;`
`a = f(3.7);`
`print(a);`
	…
	`error;`
	`# error de tipos`
	…	

`int x;`
`x = 2;`
`int f() {`
`  print(x);`
`  return -x;`
`}`
`print(1);`
`f(2);`
	…
	`error;`
	`# error de tipos`
	…	

`int f() {`
`  print(5);`
`  return 5;`
`}`
`print(1);`
`int a;`
`a = f(2);`
`print(a);`
	…
	`error;`
	`# error de tipos`
	…	

Funciones con argumentos y variables y reglas de ámbito.

Una función puede tener cualquier numero de argumentos, y también puede definir variables locales. Deben respetarse la reglas de ámbito, de manera que no se confundan las variables declaradas en la función con las variables declaradas en el programa principal.

Las variables globales son visibles en la función, siempre y cuando se declaren antes, pero no al revés.

A su vez las funciones pueden declarar variables dentro de bloques, y se deben respetar las reglas de ámbito al igual que en el código principal. (ya implementado en el ejercicio de PLX-2014).

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

`int f(int x,int y,int z,int t) {`
`  return x-y+z-t;`
`}`
`print(1);`
`int x=7;`
`x=f(9,8,x,6);`
`print(x);`
`print(3);`
		1
		2
		3

`int f(int x,int y,int z,int t) {`
`  int a = x-y;`
`  int b = z-t;`
`  return a+b;`
`}`
`print(1);`
`int x=7;`
`int a=9;`
`x=f(a,8,x,6);`
`print(x);`
`print(a);`
		1
		2
		9

`int z=7;`
`int t=6;`
`int f(int x,int y) {`
`  int a = x-y;`
`  int b = z-t;`
`  return a+b;`
`}`
`print(1);`
`z=f(9,8);`
`print(z);`
`print(3);`
		1
		2
		3

`void f(int x, int y) {`
`  if (x==y) {`
`  int a = 2;`
`  x = 3;`
`  print(a+x);`
`  }`
`}`
`print(1);`
`f(1,1);`
		1
		5

`void f(int x, int y) {`
`  if (x==y) {`
`  int a = 2;`
`  x = 3;`
`  }`
`  print(a+x);`
`}`
`print(1);`
`f(1,1);`
	…
	`error;`
	`# variable 'a' no declarada`
	…
		--

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

`int x=1;`
`int y=3;`
`void f(int x, int y) {`
`  if (x==y) {`
`  int x=3;`
`  print(x);`
`  }`
`  print(x);`
`}`
`print(x);`
`f(x+y,y+x);`
`print(x);`
		1
		3
		4
		1

`int x=7;`
`int y=7;`
`void f(int x, int y) {`
`  if (x==y) {`
`  y = y+2;`
`  if (x<y) {`
`  int y;`
`  x = y = x+y+3;`
`  print(x+y);`
`  }`
`  x = x*y+4;`
`  print(x);`
`  }`
`}`
`print(x+y);`
`f(x,y);`
`y = x*y+7;`
`print(x+y);`
		14
		20
		94
		63

`int x=2;`
`int y=7;`
`void f(int x, int y) {`
`  if (x==y) {`
`  y = y+2;`
`  if (x<y) {`
`  int y;`
`  x = y = x+y+3;`
`  print(x+y);`
`  }`
`  x = x*y+4;`
`  print(x);`
`  }`
`}`
`void g(int x) {`
`  int y=4;`
`  print(x+y);`
`}`
`print(y-x);`
`f(x+y,x+y);`
`print(y-x);`
`g(x);`
`print(y-x);`
`y = x*y+7;`
`print(y-x);`
		5
		24
		136
		5
		6
		5
		19

Funciones con distintos tipos de argumentos.

Los argumentos de una función pueden ser enteros, reales o caracteres. Cuando se realiza la llamada a la función es necesario comprobar que los tipos de los argumentos son los adecuados.

Recuerde que se puede cambiar el tipo de cualquier expresión explícitamente mediante "casting". (Esta funcionalidad ya se implemento en las practicas de PLX-2015 y PLX-2018).

Debe comprobarse que el tipo devuelto por la función corresponde al tipo declarado. Igualmente debe comprobarse que las funciones que devuelven **void** usan la sentencia **return** sin argumentos.

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

`int f(int x,float y,char z,char t) {`
`  print(t);`
`  print(z);`
`  return x-(int)y;`
`}`
`print(1);`
`int x;`
`x=f(9,7.1,'B','A');`
`print(x);`
`print(3);`
		1
		A
		B
		2
		3

`void f(int x) {`
`  print(x);`
`}`
`void g(char ch) {`
`  print(ch);`
`}`
`void h(int y,char z) {`
`  f(y);`
`  g(z);`
`}`
`g('A');`
`f(7);`
`h(9,'B');`
		A
		7
		9
		B

`int f(int x,int y,char z,char t) {`
`  print(t);`
`  print(z);`
`  return x-(int)y;`
`}`
`char g(char a, char b) {`
`  int x = (int) a;`
`  int y = (int) b;`
`  return (char) ((x+y)/2);`
`}`
`print(1);`
`char x;`
`x=g('A','C');`
`f(1, (int)x, x, 'Z');`
`print(x);`
`print(3);`
		1
		Z
		B
		B
		3

`char f(int x, int y) {`
`  return (x+y)/2;`
`}`
`print(1);`
`char ch;`
`ch = f(66,68);`
`print(ch);`
	…
	`error;`
	`# error de tipos`
	…
		--

`char f(int x, int y) {`
`  return (char) (x+y);`
`}`
`print(1);`
`int s;`
`s = f(11,66);`
`print(s);`
	…
	`error;`
	`# error de tipos`
	…
		--

Uso de funciones.

Las funciones pueden formar parte de cualquier expresión, pueden llamarse mutuamente e incluir dentro de su propio código la llamada a otras funciones. (al igual que en C y Java).

Puede haber llamadas a funciones anidadas.

Recuerde que el operador ‘,’ que separa los parámetros en la llamada a funciones tiene menos prioridad que los operadores aritméticos y asociatividad derecha, por lo que la ejecución de las funciones comienza de derecha a izquierda, lo que tiene influencia en el caso de que se modifiquen los valores de las variables globales o haya cualquier otro tipo de efectos colaterales.

NOTA: En el lenguaje PLX, puesto que la gestión de memoria es completamente estática, no está permitido el uso de llamadas recursivas, ya sea de forma directa o indirecta, pero no es necesario comprobar que esta condición se cumple. En caso de llamadas recursivas el código generado simplemente no funcionaria correctamente.

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

`int f(int x,int y) {`
`  return x+y;`
`}`
`int g(int z) {`
`  return z*z;`
`}`
`print(f(1,2)+g(3)*4);`
		39

`int f(int x,int y) {`
`  return x+y;`
`}`
`int g(int z) {`
`  return z*z;`
`}`
`print(f(1,g(3)));`
		10

`int g(int z) {`
`  return z*z;`
`}`
`int f(int x,int y) {`
`  return g(x+y);`
`}`
`print(f(1,g(3)));`
		100

`int h(int t) {`
`  return t/2;`
`}`
`int g(int z) {`
`  return h(z)*h(z);`
`}`
`int f(int x,int y) {`
`  return g(x+y);`
`}`
`print(f(6,g(10)));`
		225

`int h(int t) {`
`  return t/2;`
`}`
`int g(int z) {`
`  return h(z)*h(z);`
`}`
`int f(int x,int y) {`
`  return g(x+y);`
`}`
`print(f(6,g(h(8))));`
		25

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

`int a = 2;`
`int f(int x,int y) {`
`  a = a+x+y;`
`  print(a);`
`  return x+y;`
`}`
`int g(int z) {`
`  print(z);`
`  a = a+z;`
`  return z*z;`
`}`
`print(f(g(a),g(a)));`
`print(a);`
		2
		4
		40
		32
		40

`int a = 2;`
`int b = 3;`
`int f(int x,int y,int z) {`
`  return z-x+y;`
`}`
`int g(int z) {`
`  print(z);`
`  a = a+z;`
`  return z*z;`
`}`
`a = f(a, b, g(3)) + g(b);`
`print(a);`
`print(b);`
		3
		3
		16
		3

`int doble(int x) {`
`  return 2*x;`
`}`
`int triple(int x) {`
`  return doble(x)*x;`
`}`
`print(triple(3));`
`print(doble(triple(2)));`
		18
		16

Declaración implícita de funciones y prototipos.

Al igual que en C, las funciones en PLX deben declararse antes de poder usarlas. No obstante, si una función no se declara previamente, el compilador puede asumir una declaración implícita, según el tipo de parámetros con el que se haga la llamada. Por ejemplo si una función se llama como **f('A',65),** se puede asumir que es una función que tiene dos argumentos, el primero de tipo **char** y el segundo de tipo **int.** El tipo devuelto no puede inferirse y por tanto en las declaraciones implícitas se asume siempre que es **int**.

Cuando una función se define implícitamente es necesario que luego aparezca su definición, que debe coincidir con la declaración implícita.

Para evitar las declaraciones implícitas, en PLX al igual que en C, se usan prototipos de funciones, que son cabeceras de funciones sin cuerpo, terminadas en punto y coma, que lo único que hacen es declarar previamente la función para evitar la declaración implícita. La definición de la función aparecerá después como en el caso de las declaraciones implícitas, y deberá coincidir con los tipos del prototipo.

Los nombres de los atributos de la función no tienen porque coincidir, siendo opcionales en el prototipo.

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

`print(suma('A',60));`
`int suma(char a,int y) {`
`  return (int)a + y;`
`}`
		125

`print(resta(60,'A'));`
`int resta(int y, char a) {`
`  return (int)a - y;`
`}`
		5

`void f(char a, int y);`
`print('B');`
`f('A',60);`
`print('C');`
`void f(char a, int y) {`
`  print(a);`
`  print(y);`
`}`
		B
		A
		60
		C

`void f(char, int);`
`print('B');`
`f('A',60);`
`print('C');`
 	
`void f(char a, int y) {`
`  print(a);`
`  print(y);`
`}`
		B
		A
		60
		C

`void f(char b, int z);`
`print('B');`
`void f(char a, int y) {`
`  print(a);`
`  print(y);`
`}`
`f('A',60);`
`print('C');`
		B
		A
		60
		C

`int f(int x);`
`int f(int x) {`
`  int y = g(g(x));`
`  return y;`
`}`
`int g(int z) {`
`  print(2*z);`
`  return 2*z;`
`}`
`f(f(7));`
		14
		28
		56
		112

Paso de parámetros por referencia y por valor.
Al igual que en C, las funciones pueden recibir parámetros por valor, mediante el uso de punteros
.

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

`int refer(int* x) {`
`  *x = *x * *x;`
`  return *x;`
`}`
`int a = 7;`
`int b;`
`b = refer(&a);`
`print(a);`
`print(b);`
		49
		49

`int a =7;`
`int b =5;`
`int refer(int* x, int b) {`
`  *x = *x * b;`
`  return *x;`
`}`
`b = refer(&a, b);`
`print(a);`
`print(b);`
		35
		35

`int a =7;`
`int b =5;`
`int* refer(int* x, int b) {`
`  *x = *x * b;`
`  return x;`
`}`
`int *c;`
`c = refer(&a, b);`
`print(a);`
`print(*c);`
		35
		35

`int a =7;`
`int b =5;`
`int* refer(int* x, int b) {`
`  *x = *x * b;`
`  return x;`
`}`
`b = *refer(&a, b);`
`print(a);`
`print(b);`
		35
		35

`int refer(int** x) {`
`  int *p;`
`  *p = 12;`
`  **x = *p + **x;`
`  return *p;`
`}`
`int* a;`
`int b;`
`*a = 4;`
`b = refer(&a);`
`print(*a);`
`print(b);`
		16
		12

`int* refer(int **x) {`
`  int y = 4 * **x;`
`  **x = y + 4;`
`  return &y;`
`}`
`int *a;`
`int *b;`
`*a = 4;`
`b = refer(&a);`
`print(*a);`
`print(*b);`
		20
		16

EL CÓDIGO OBJETO (Código de tres direcciones):

El código objeto es a su vez una extensión del código intermedio utilizado en la practica principal de la asignatura. Se añaden algunas instrucciones necesarias para generar el código requerido por el lenguaje PLX. Todas las variables del código intermedio se considera que están previamente definidas y que su valor inicial es 0.

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

Instrucción	Acción
`x = a ;`	Asigna el valor de a en la variable x
`x = a + b ;`	Suma los valores de a y b, y el resultado lo asigna a la variable x
`x = a - b ;`	Resta los valores de a y b, y el resultado lo asigna a la variable x
`x = a * b ;`	Multiplica los valores de a y b, y el resultado lo asigna a la variable x
`x = a / b ;`	Divide (div. entera) los valores de a y b , y el resultado lo asigna a la variable x
`x = 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 un # se considera un comentario.

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.
NOTA: Se han resaltado en negrita las nuevas instrucciones para el manejo de funciones.

IMPLEMENTACIÓN DE LA PRÁCTICA:

Se proporciona una solución compilada del ejercicio (versiones para Linux, Windows y Mac). Esto puede servir de ayuda para comprobar los casos de prueba y las instrucciones intermedio,. No es necesario que el código generado sea idéntico al que se propone como ejemplo (que de hecho no es óptimo), basta con que sea equivalente, es decir que dé los mismos resultados al ejecutar los casos de prueba. Para compilar y ejecutar un programa en lenguaje PLX, pueden utilizarse las instrucciones

	Linux
Compilación	`java PLXC prog.plx prog.ctd`
Ejecución	`./ctd prog.ctd`

En donde `prog.plx` contiene el código fuente en PLX, `prog.ctd` es un fichero de texto que contiene el código intermedio válido según las reglas gramaticales de este lenguaje. El programa `plxc` es un *script* del *shell* del sistema operativo que llama a (`java PLXC`), que es el programa que se pide construir en este ejercicio. El programa `ctd` es un interprete del código intermedio. Asimismo, para mayor comodidad se proporciona otro *script del shell* denominado `plx` que compila y ejecuta en un solo paso, y al que se pasa el nombre del fichero sin extensión.

	Linux
Compilación + Ejecución	`./plx prog`

NOTAS IMPORTANTES:

1. Toda práctica debe contener al menos tres ficheros denominados "`PLXC.java`", "`PLXC.flex`" y "`PLXC.cup`", correspondientes respectivamente al programa principal y a las especificaciones en JFlex y Cup. Para realizar la compilación se utilizarán las siguientes instrucciones:

```
cup PLXC.cup
jflex PLXC.flex
javac *.java
```

   y para compilar y ejecutar el programa en PLX

```
java PLXC prog.plx prog.ctd
./ctd prog.ctd
```

2. Puede ocurrir que al descargar los ficheros y descomprimirlos en Linux se haya perdido el carácter de fichero ejecutable. Para poder ejecutarlos debe modificar los permisos:

```
chmod +x plx plxc ctd
chmod +x plx-linux plxc-linux ctd-linux
```

3. El programa `ctd`, interprete del código intermedio, tiene una opción –v para generar trazas que pueden ayudar en la depuración de errores:

```
./ctd –v prog.ctd
```

4. Los ejemplos que se proponen como casos de prueba no definen exhaustivamente el lenguaje. Para implementar esta práctica es necesario generar otros casos de prueba de manera que se garantice un funcionamiento en todos los casos posibles, y no solo en este limitado banco de pruebas.

5. En todas las pruebas en donde el código **plx** produce un "*error*", para comprobar que el compilador realmente detecta el error, se probará también que el código corregido compila adecuadamente, y si no es así la prueba no se considerará correcta.