Extensión de la Práctica Principal de Procesadores de Lenguajes (Examen febrero 2015)

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 práctica 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.

EL CÓDIGO FUENTE (Lenguaje PLX):

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

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 explicita 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
`print (3.1416);` `print (3.1416E-2);`	`print 3.1416;` `print 3.1416E-2;`	**3.141600** **0.031416**

`float e;`
`e=2.7172;`
`float pi;`
`pi=3.1416;`
`print (e*pi);`
	`e = 2.7172;`
	`pi = 3.1416;`
	`t0 = e *r pi;`
	`print t0;`
		8.536355

`int ix;`
`ix = 2;`
`int iy;`
`iy = 3;`
`float fx;`
`fx = ix;`
`float fy;`
`fy = iy;`
`print (iy/ix + fx/fy);`
	`ix = 2;`
	`iy = 3;`
	`fx = (float) ix;`
	`fy = (float) iy;`
	`t0 = iy / ix;`
	`t1 = fx /r fy;`
	`t2 = (float) t0;`
	`t3 = t2 +r t1;`
	`print t3;`
		1.666667

`print((int)(2.0/3.0 +(float)3/2));`	`t0 = 2.0 /r 3.0;`
	`t1 = (float) 3;`
	`t2 = (float) 2;`
	`t3 = t1 /r t2;`
	`t4 = t0 +r t3;`
	`t5 = (int) t4;`
	`print t5;`
		2

`int x;`
`x = 3.1416;`
`print(x);`
	…
	`error;`
	…
		--

* 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). (5 puntos)

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

`int x[10] ;`
`x[1]=3;`
`print (x[1]);`
	`x[1] = 3;`
	`t0 = x[1];`
	`print t0;`
		3

`int x[10] ;`
`x[1] = 1;`
`x[2] = 2;`
`x[3] = 3;`
`int z;`
`z = x[1]+x[2]*x[3];`
`print(z);`
	`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;`
		7

`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);`
	`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;`
		285

`int p;`
`p=10;`
`int x[p];`
`x[1]=3;`
`print (x[1]);`
	…
	`error;`
	….
		--

`int i;`
`int a[3];`
`for(i=0; i<5; i=i+1) {`
`  a[(i+i)/2] = i*i;`
`  print (a[i]);`
`}`
	…
	`# Comprobacion de rango`
	`if (t2 < 0) goto L4;`
	`if (3 < t2) goto L4;`
	`if (3 == t2) goto L4;`
	`goto L5;`
	`L4:`
	`error;`
	`halt;`
	`L5:`
	….
		0
		1
		4
		runtime error

* Constantes de tipo ***array*** *unidimensional* e inicialización. Para ello se usarán expresiones separadas por comas y contenidas entre llaves. Las expresiones pueden ser calculadas en tiempo de ejecución. 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 (5 puntos)

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

`int a[3];`
`a = {1,2,3};`
`print(a[0]+a[1]*a[2]);`
	`t0[0] = 1;`
	`t0[1] = 2;`
	`t0[2] = 3;`
	`t1 = t0[0];`
	`a[0] = t1;`
	`t1 = t0[1];`
	`a[1] = t1;`
	`t1 = t0[2];`
	`a[2] = t1;`
	`a = t0;`
	`t2 = a[0];`
	`t3 = a[1];`
	`t4 = a[2];`
	`t5 = t3 * t4;`
	`t6 = t2 + t5;`
	`print t6;`
		7

`int a[3];`
`int x;`
`x = 2;`
`a = {1,1+1,1+x*1};`
`print(a[0]+a[1]*a[2]);`
	`x = 2;`
	`t0[0] = 1;`
	`t1 = 1 + 1;`
	`t0[1] = t1;`
	`t2 = x * 1;`
	`t3 = 1 + t2;`
	`t0[2] = t3;`
	`t4 = t0[0];`
	`a[0] = t4;`
	`t4 = t0[1];`
	`a[1] = t4;`
	`t4 = t0[2];`
	`a[2] = t4;`
	`a = t0;`
	`t5 = a[0];`
	`t6 = a[1];`
	`t7 = a[2];`
	`t8 = t6 * t7;`
	`t9 = t5 + t8;`
	`print t9;`
		7

`int a[3] = {1,2,3};`
`print(a[0]+a[1]*a[2]);`
	`t0[0] = 1;`
	`t0[1] = 2;`
	`t0[2] = 3;`
	`t1 = t0[0];`
	`a[0] = t1;`
	`t1 = t0[1];`
	`a[1] = t1;`
	`t1 = t0[2];`
	`a[2] = t1;`
	`t2 = a[0];`
	`t3 = a[1];`
	`t4 = a[2];`
	`t5 = t3 * t4;`
	`t6 = t2 + t5;`
	`print t6;`
		7

`int a[5];`
`a[3] = 4;`
`a = {1,2,3};`
`a[4] = 5;`
`print(a[0]+a[1]*a[2]+a[3]*a[4]);`
		27

`int a[2];`
`a = {1,2,3};`
`print(a[0]+a[1]*a[2]);`
	…
	`error;`
	…
		--

* Asignación de *arrays unidimensionales* con variables de tipo *array unidimensionales*. En este caso deben 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. (5 puntos)

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

`int a[3];`
`int b[3];`
`int suma;`
`a = {1,2,3};`
`b = a;`
`int i;`
`for(i=0; i<3; i=i+1) {`
`  suma = suma + b[i];`
`}`
`print(suma);`
		6

`int a[3];`
`int b[5];`
`int suma;`
`a = {1,2,3};`
`b = a;`
`b[3] = 4;`
`b[4] = 5;`
`int i;`
`for(i=0; i<5; i=i+1) {`
`  suma = suma + b[i];`
`}`
`print(suma);`
		15

`int a[3];`
`int b[2];`
`int suma;`
`a = {1,2,3};`
`b = a;`
`int i;`
`for(i=0; i<3; i=i+1) {`
`  suma = suma + b[i];`
`}`
`print(suma);`
	…
	`# las matrices no son compatibles`
	`error;`
	…
		--

`int a[3];`
`int b[3];`
`int c[3];`
`int p;`
`a = {1,2,3};`
`c = b = a;`
`int i;`
`for(i=0; i<3; i=i+1) {`
`  p = p+ a[i]*b[i]+c[i];`
`}`
`print(p);`
		20

`int a[3];`
`int b[3];`
`int c[3];`
`int p;`
`c = b = a = {1,2,3};`
`int i;`
`for(i=0; i<3; i=i+1) {`
`  p = p+ a[i]*b[i]+c[i];`
`}`
`print(p);`
		20

* 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. (10 puntos)

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

`float x[10] ;`
`x[1]=3.1416;`
`print (x[1])`
	…	**3.141600**

`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);`
	…	**6.600000**

`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);`
	….
	`t0 = 1+1`
	`t1 = (float) t0;`
	`x[2] = t0;`
	…
		6.400000

`float x[3] ;`
`x = {1.1, 2.2, 3.3};`
`print(x[0]+x[1]*x[2]);`
	…	**8.360001**

`int x[3] ;`
`x = {1.1, 2.2, 3.3};`
`print(x[0]+x[1]*x[2]);`
	…
	`# error de tipos`
	`error;`
	…
		--

`float x[3] ;`
`x = {1.1, 2, 3.3};`
`print(x[0]+x[1]*x[2]);`
	…
	`# error de tipos`
	`error;`
	…
		--

`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]);`
	…	**10.583200**

`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]);`
	…	**6.600000**

`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]);`
	…
	`# error de tipos`
	`error;`
	…
		--

`int x[3] ;`
`float y[3] ;`
`x[0] = 1;`
`x[1] = 2;`
`x[2] = 3;`
`y = x;`
`print(y[0]+y[1]+y[2]);`
	…
	`# error de tipos`
	`error`
	…
		--

* Sentencia **for….in…** . Añadir una nueva sentencia *for* al lenguaje de manera que recorra todos los valores del *array*. El *array* del bucle puede ser constante, variable o incluso una expresión. Se valorará que se realicen las comprobaciones de tipos necesarias en tiempo de compilación. Para obtener la máxima calificación el compilador debe aceptar que la variable de control del bucle también puede ser el elemento de un *array*. (9 puntos)

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

`int a[3];`
`a[0] = 1;`
`a[1] = 3;`
`a[2] = 5;`
`int x;`
`for x in a do {`
`  print(x);`
`}`
		1
		3
		5

`int x;`
`for x in {7,4,9} do {`
`  print(x);`
`}`
		7
		4
		9

`int x;`
`for x in {7+3,4+6,5+5} do {`
`  print(x);`
`}`
		10
		10
		10

`int x;`
`for x in a do {`
`  print(x);`
`}`
	…
	`# variable no declarada`
	`error;`
	…
		--

`int a;`
`int x;`
`for x in a do {`
`  print(x);`
`}`
	…
	`# tipo incorrecto`
	`error;`
	…
		--

`int a[3];`
`a[0] = 1;`
`a[1] = 3;`
`a[2] = 5;`
`int b[3];`
`int i;`
`int x;`
`for(i=0; i<3; i++) {`
`  for b[i] in a do {`
`  x = x + b[i];`
`  }`
`}`
`print(x);`
`print(b[0]+b[1]+b[2]);`
		27
		15

`int a[3];`
`int b[3];`
`a[0] = 4;`
`a[1] = 5;`
`a[2] = 6;`
`b[0] = 11;`
`b[1] = 20;`
`b[2] = 21;`
`int x;`
`int y;`
`int z;`
`for x in a do {`
`  for y in b do {`
`  z = z + x*y;`
`  }`
`}`
`print(z);`
		780

float a[3]; **a[0] = 1.3;** **a[1] = 3.5;** **a[2] = 5.8;** **float x;** **float z;** **for x in a do {** ** z = z + x;** **}** **print(z);**		**10.600000**
float a[3]; **a[0] = 1.3;** **a[1] = 3.5;** **a[2] = 5.8;** **int x;** **float z;** **for x in a do {** ** z = z + x;** **}** **print(z);**	… **# tipos incompatibles** **error;** …	**--**

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-ésimo valor del array y, asignando el contenido en x
`x[a] = b ;`	Coloca el valor b en la a-ésima posición del array x
`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 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.
`print a ;`	Imprime el valor de a
`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 o reales, `x,y` representan siempre una variable y `l` representa una etiqueta de salto.

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.