Extensión de la Practica Principal de Procesadores de Lenguajes (Examen febrero 2014)

Esta practica 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.

EL CÓDIGO FUENTE (Lenguaje PLX):

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

* 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 debe 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. (2 puntos)

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 mas de una variable, separándolas por comas. Debe controlarse que no se declare una misma variable dos veces. (2 punto)

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. (2 puntos)

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

* Declaración de variables con ámbito mediante estructura de bloques. La declaración de variables puede ocurrir en cualquier punto del programa. Una variable estará activa desde que se define hasta que finaliza el bloque en el que se declaró. Una variable declarada en un bloque, estará visible en los bloques contenidos en él, pero las variables definidas dentro de un bloque no podrán usarse fuera de él. Una variable declarada en un bloque podrá ser redefinida en los bloques incluidos en él y la nueva variable ocultará la definida anteriormente con el mismo nombre. (Las mismas reglas de ámbito que usa el lenguaje C, no las de Java). (4 puntos)

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

`int x;`
`int y;`
`x=y=1;`
`if (x==y) {`
`  int a;`
`  a = 2;`
`  x = 3;`
`  print(a+x);`
`}`
`print(x);`
	`  y = 1;`
	`  x = y;`
	`  if (x == y) goto L0;`
	`  goto L1;`
	`L0:`
	`  a = 2;`
	`  x = 3;`
	`  t0 = a + x;`
	`  print t0;`
	`  goto L2;`
	`L1:`
	`L2:`
	`  print x;`
		5
		3

`int x;`
`int y;`
`x=1;`
`if (x==y) {`
`  int a;`
`  a = 2;`
`}`
`print(a);`
	…
	`error;`
	`# variable no declarada`
	…
		--

`int x;`
`int y;`
`x=y=1;`
`if (x==y) {`
`  int x;`
`  x = 3;`
`  print(x+y);`
`}`
`print(x+y);`
	`  y = 1;`
	`  x = y;`
	`  if (x == y) goto L0;`
	`  goto L1;`
	`L0:`
	`  x_0 = 3;`
	`  t0 = x_0 + y;`
	`  print t0;`
	`  goto L2;`
	`L1:`
	`L2:`
	`  t1 = x + y;`
	`  print t1;`
		4
		2

`int x;`
`int y;`
`x=y=1;`
`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);`
`}`
`y = x*y+7;`
`print(y);`
	`  y = 1;`
	`  x = y;`
	`  if (x == y) goto L0;`
	`  goto L1;`
	`L0:`
	`  t0 = y + 2;`
	`  y = t0;`
	`  if (x < y) goto L3;`
	`  goto L4;`
	`L3:`
	`  t1 = x + y_1;`
	`  t2 = t1 + 3;`
	`  y_1 = t2;`
	`  x = y_1;`
	`  t3 = x + y_1;`
	`  print t3;`
	`  goto L5;`
	`L4:`
	`L5:`
	`  t4 = x * y;`
	`  t5 = t4 + 4;`
	`  x = t5;`
	`  print x;`
	`  goto L2;`
	`L1:`
	`L2:`
	`  t6 = x * y;`
	`  t7 = t6 + 7;`
	`  y = t7;`
	`  print y;`
		8
		16
		55

* Expresiones de pre-incremento y post-incremento. (y también de pre-decremento y post-decremento). Estos operadores solo pueden aplicarse a variables, no a numero o expresiones. (3 puntos)

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

`int x;`
`int y;`
`int z;`
`x=3;`
`y=x++;`
`z=++x;`
`print (x*y*z);`
	`x = 3;`
	`t0 = x;`
	`x = x + 1;`
	`y = t0;`
	`x = x + 1;`
	`z = x;`
	`t1 = x * y;`
	`t2 = t1 * z;`
	`print t2;`
		75

`int x;`
`int y;`
`x=2;`
`y=++x +x+ x++;`
`print (y*x++*++x-x--*--x);`
	`x = 2;`
	`x = x + 1;`
	`t0 = x + x;`
	`t1 = x;`
	`x = x + 1;`
	`t2 = t0 + t1;`
	`y = t2;`
	`t3 = x;`
	`x = x + 1;`
	`t4 = y * t3;`
	`x = x + 1;`
	`t5 = t4 * x;`
	`t6 = x;`
	`x = x - 1;`
	`x = x - 1;`
	`t7 = t6 * x;`
	`t8 = t5 - t7;`
	`print t8;`
		192

`int x;`
`x= 1;`
`x= 1 + (x+x)++ ;`
`print (x);`
	…
	`error;`
	…
		--

* Implementación del operador modulo %. Este operador halla el resto de la división del primer operando entre el segundo (ejemplo 1). Este operador tiene la misma asociatividad y prioridad de operación que en el lenguaje Java. (Cada alumno deberá realizar pruebas en Java para determinar cuál es exactamente. (ejemplo 2)). (2 puntos)

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

`print ( 18 % 7 );`	`t0 = 18 / 7;`
	`t1 = t0 * 7;`
	`t2 = 18 - t1;`
	`print t2;`
		4

`print(17%13%6/3+19%7*20%17)`	`t0 = 17 / 13;`
	`t0 =  t0 * 13;`
	`t0 = 17 -  t0;`
	`t1 =  t0 / 6;`
	`t1 =  t1 * 6;`
	`t1 =  t0 -  t1;`
	`t2 =  t1 / 3;`
	`t3 = 19 / 7;`
	`t3 =  t3 * 7;`
	`t3 = 19 -  t3;`
	`t4 =  t3 * 20;`
	`t5 =  t4 / 17;`
	`t5 =  t5 * 17;`
	`t5 =  t4 -  t5;`
	`t6 =  t2 +  t5;`
	`print  t6;`
		16

* Sentencia ***for-to*** y ***for-downto*** al estilo PASCAL. En este tipo de sentencias la variable de control del bucle es siempre entera, y se incrementa (o decrementa) en una cantidad fija (por defecto se incrementa o decrementa en 1, ejemplos 1 y 2). El valor final se incluye dentro de la iteración. Al igual que en Pascal, el valor final de comparación es constante en todas las iteraciones. Puede usarse una expresión, pero en ese caso, su valor solo se calcula al principio, y no se recalcula tras cada iteración. (ejemplo 3). Implementar asimismo una modificación de esta instrucción de manera que pueda especificarse el valor del incremento o decremento en cada iteración. El valor del incremento se calculará dinámicamente en cada iteración. (ejemplos 4 y 5). (5 puntos)

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

`int i;`
`int suma;`
`for i=1 to 10 do {`
`  suma = suma +i;`
`}`
`print (suma);`
	`  i = 1;`
	`L0:`
	`  if (10 < i) goto L3;`
	`  goto L2;`
	`L1:`
	`  i = i + 1;`
	`  goto L0;`
	`L2:`
	`  t0 = suma + i;`
	`  suma = t0;`
	`  goto L1;`
	`L3:`
	`  print suma;`
		`55`

`int i;`
`int suma;`
`for i=2*5 downto 10/10 do {`
`  suma = suma +i;`
`}`
`print (suma);`
	`  t0 = 2 * 5;`
	`  i = t0;`
	`  t1 = 10 / 10;`
	`L0:`
	`  if (i < t1) goto L3;`
	`  goto L2;`
	`L1:`
	`  i = i - 1;`
	`  goto L0;`
	`L2:`
	`  t2 = suma + i;`
	`  suma = t2;`
	`  goto L1;`
	`L3:`
	`  print suma;`
		`55`

`int i;`
`int suma;`
`for i=1 to 10*i do {`
`  suma = suma +i;`
`}`
`print (suma);`
	`  i = 1;`
	`  t0 = 10 * i;`
	`L0:`
	`  if (t0 < i) goto L3;`
	`  goto L2;`
	`L1:`
	`  i = i + 1;`
	`  goto L0;`
	`L2:`
	`  t1 = suma + i;`
	`  suma = t1;`
	`  goto L1;`
	`L3:`
	`  print suma;`
		`55`

`int i;`
`int suma;`
`for i=1 to 10*i step 2 do {`
`  suma = suma +i;`
`}`
`print (suma);`
	`  i = 1;`
	`  t0 = 10 * i;`
	`L0:`
	`  if (t0 < i) goto L3;`
	`  goto L2;`
	`L1:`
	`  i = i + 2;`
	`  goto L0;`
	`L2:`
	`  t1 = suma + i;`
	`  suma = t1;`
	`  goto L1;`
	`L3:`
	`  print suma;`
		`25`

`int i;`
`int suma;`
`for i=100 downto suma/10`
`  step suma/10 do {`
`  suma = suma +i;`
`}`
`print (suma);`
	`  i = 100;`
	`  t0 = suma / 10;`
	`L0:`
	`  if (i < t0) goto L3;`
	`  goto L2;`
	`L1:`
	`  t1 = suma / 10;`
	`  i = i - t1;`
	`  goto L0;`
	`L2:`
	`  t2 = suma + i;`
	`  suma = t2;`
	`  goto L1;`
	`L3:`
	`  print suma;`
		`321`

* Pruebas de integración de diferentes componentes del lenguaje. Se muestran solo algunos ejemplos de este tipo de casos de prueba. (10 puntos).

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

`int i=1,j,n=i+9;`
`int suma = 0;`
`for( i=1; i<n; i++) {`
`  for j=1 to i do {`
`  suma = suma+n*i+j;`
`  }`
`}`
`print(suma);`
	`--`	`3015`

`int x,y=10;`
`for (x=1; x<y; x=y=x++) {`
`  print(x);`
`}`
	`--`	`1`

`int alfa;`
`if (alfa == 0) {`
`  int alfa = 12;`
`  if (alfa % 2 == 0) {`
`  print(++alfa);`
`  int alfa = 7;`
`  print(alfa);`
`  } else {`
`  print(alfa);`
`  }`
`  print(alfa++);`
`}`
`print(alfa++);`
	`--`	`13`
		`7`
		`13`
		`0`

`int alfa;`
`for alfa=8 downto 2 step alfa/2 do`
`{`
`  int x;`
`  if (alfa % alfa/2 == 0 ) x=alfa`
`;`
`  else x=0;`
`  print(x);`
`}`
`print(alfa);`
	`--`	`8`
		`4`
		`2`
		`1`

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
`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, `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	`./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 JPLC 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. Los ejemplos que se proponen como casos de prueba no definen exhaustivamente el lenguaje. Para implementar esta practica 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.

4. 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
```