Extensión de la Práctica Principal de Procesadores de Lenguajes

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-2022-C 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-2022-C 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. En concreto, para este ejercicio se utilizarán explícitamente solo los tipos “int”, “float” y “char” y el constructor de tipos “set”.

EL CÓDIGO FUENTE (Lenguaje PLX):

El lenguaje PLX-2022-C incluye todas las sentencias de control definidas en el lenguaje PLX anterior y alguna más. El lenguaje intermedio CTD, no es necesario modificarlo.

ASPECTOS LEXICOS

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

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 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.

NOTAS:

La versión compilada de “plxc” se ha actualizado en el repositorio para cubrir los cambios requeridos en este ejercicio, por lo que, aunque ya se hubiera descargado anteriormente debería actualizarse.

IMPORTANTE: Al imprimir los elementos del conjunto <u>da igual el orden</u> en el que se muestren. Al imprimirlos hay un espacio (carácter 32) entre cada dos elementos y un salto de línea (carácter 10) al final.

El conjunto será compatible con el tipo array hacia atrás, es decir, un array unidimensional se puede convertir implícitamente en un conjunto, pero no al revés. PERO en este examen no será necesario comprobar la compatibilidad entre conjunto y array.

La implementación interna de los conjuntos coincide con la del array unidimensional.

Operaciones con conjuntos. Union

La operación básica en un conjunto es añadir elementos mediante el operador +, Estos operadores funcionan entre conjuntos y conjuntos, entre conjuntos y arrays unidimensionales o bien entre conjuntos y elementos (ver ejemplos).

Se implementa asimismo el operador += para añadir elementos, o conjuntos al propio conjunto (ver ejemplos.

Debe tenerse en cuenta que la unión de conjuntos no añade elementos repetidos

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

`set int c;`
`set int d;`
`c = {1,2,3};`
`d = {2,4,5,6};`
`d = c + d;`
`print(c);`
`print(d);`
		1 2 3
		1 2 3 4 5 6
			setunion1.plx

`set int c;`
`set int d;`
`c = {1,2,3};`
`d = c + {2,3,4,5};`
`print(c);`
`print(d);`
		1 2 3
		1 2 3 4 5
			setunion2.plx

`set int c;`
`set int d;`
`c = {1,2,3};`
`d = c + 4 + {5};`
`print(c);`
`print(d);`
		1 2 3
		1 2 3 4 5
			setunion3.plx

`set int c;`
`c = {1,2,3};`
`c += 4;`
`print(c);`
		1 2 3 4	setunion4.plx

`set int c;`
`c = {1,2,3};`
`c += {3,4};`
`print(c);`
		1 2 3 4	setunion5.plx

`set int c;`
`set int d;`
`c = {1,2,3};`
`d = {3,4,5};`
`c += d + 6 + {7};`
`print(c);`
`print(d);`
		1 2 3 4 5 6 7
		3 4 5
			setunion6.plx

`set char c;`
`c = {'A','B','C'};`
`set char d;`
`d = {'B','A','D'};`
`print(c+d);`
		A B C D	setunion7.plx

`set char c;`
`c = {'A','B','C'};`
`set char d;`
`d = {'B','A','D'};`
`set char e;`
`e = {'X','A','Z','B'};`
`print(c+d+e);`
		A B C D X Z	setunion8.plx

Operaciones con conjuntos. Resta

La resta de conjunto se implementa mediante el operador -, y consiste en quitar del conjunto original uno o varios elementos. Estos operadores funcionan entre conjuntos y conjuntos, entre conjuntos y arrays unidimensionales, y entre conjuntos y elementos. (ver ejemplos).

El resultado de la resta es un nuevo conjunto que a su vez puede volver a operarse con otro conjunto, array o elemento.

Se implementa asimismo el operador -= para realizar la intersección de un conjunto con otro manteniendo en el primero solo sólo los elementos que no se hayan eliminado.

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

`set int c;`
`set int d;`
`set int e;`
`c = {1,2,3};`
`d = {2,3,4,5,6};`
`e = d - c;`
`print(c);`
`print(d);`
`print(e);`
		1 2 3
		2 3 4 5 6
		4 5 6
			`setsub1.plx`

`set int c;`
`set int d;`
`c = {1,2,3};`
`d = c - {1,4} - 3;`
`print(c);`
`print(d);`
		1 2 3
		2
			`setsub2.plx`

`set int c;`
`c = {1,2,3};`
`c -= {1,3,5};`
`print(c);`
		1 3	`setsub3.plx`

Operaciones con conjuntos. Intersección

La intersección de conjunto se implementa mediante el operador *, Estos operadores funcionan entre conjuntos y conjuntos, entre conjuntos y arrays unidimensionales (ver ejemplos).

Se implementa asimismo el operador *= para realizar la intersección de un conjunto con otro manteniendo en el primero sólo los elementos comunes.

Debe tenerse en cuenta que la intersección de dos conjuntos nunca se añaden elementos repetidos

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

`set int c;`
`set int d;`
`c = {1,2,3};`
`d = {2,3,4,5,6};`
`d = c * d;`
`print(c);`
`print(d);`
		1 2 3
		2 3
			setinter1.plx

`set int c;`
`c = {1,2,3};`
`d = c * {2,3,4,5};`
`print(c);`
`print(d);`
		1 2 3
		2 3
			setinter2.plx

`set int c;`
`c = {1,2,3};`
`c *= {1,3,5};`
`print(c);`
		1 3	setinter3.plx

`set int c;`
`set int d;`
`c = {1,2,3};`
`d = {1,3,5};`
`c = d * {3,5,7} +`
`{2,4,6};`
`print(c);`
`print(d);`
		3 5 2 4 6
		1 3 5
			setinter4.plx

`set int a;`
`set int b;`
`set int c;`
`a = {1,2,3,4,5,6,7};`
`b = {1,3,5};`
`c = {2,4,6,8};`
`c = a*b + b*c;`
`print(c);`
		1 3 5	setinter5.plx

`set int a;`
`set int b;`
`set int c;`
`a = {1,2,3,4,5,6,7};`
`b = {1,3,5};`
`c = {2,4,5,6,7,8};`
`c *=  a*b + b*c + 7;`
`print(c);`
		5 7	setinter6.plx

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. Para realizar la corrección automática es necesario que la salida sea la misma que la esperada, independientemente del orden de los elementos del conjunto. Para esto se usa un pequeño script `lineSort.py` en Python

```
./ctd prog.ctd | python3 lineSort.py
```

6. 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.

7. 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.