Extensión de la Práctica Principal de Procesadores de Lenguajes (Examen febrero 2017)

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.

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.

* Operador ternario. Este operador tiene menor precedencia que cualquiera de los otros incluyendo los operadores de asignación y los operadores, de relación y operadores booleanos.

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

`print ( 1 < 2 ? 3 : 4 );`	`if (1 < 2) goto L0;`
	`goto L1;`
	`L0:`
	`t0 = 3;`
	`goto L2;`
	`L1:`
	`t0 = 4;`
	`L2:`
	`print t0;`
		3

`int x;`
`x=1;`
`int y;`
`y=2;`
`int z;`
`z = (x*y<x+y) ? x>y?x=2:y=3 : 4;`
`print (x*y*z);`
	`x = 1;`
	`y = 2;`
	`t0 = x * y;`
	`t1 = x + y;`
	`if (t0 < t1) goto L0;`
	`goto L1;`
	`L0:`
	`if (y < x) goto L3;`
	`goto L4;`
	`L3:`
	`x = 2;`
	`t3 = x;`
	`goto L5;`
	`L4:`
	`y = 3;`
	`t3 = y;`
	`L5:`
	`t2 = t3;`
	`goto L2;`
	`L1:`
	`t2 = 4;`
	`L2:`
	`z = t2;`
	`t4 = x * y;`
	`t5 = t4 * z;`
	`print t5;`
		9

`print ( 1+2+3 < 2*6+1 ?`
`3+(1<2?5:6)+7 :`
`4<5?8:9 );`
	`...`	**15**

* Operador Elvis. **`?:`** Este operador es un operador binario, pero muy similar al operador ternario clásico. Una expresión que use este operador es de la forma: *exp* **`?:`** *exp*. Si el resultado de la primera expresión es distinto de cero, el resultado es el mismo de la primera expresión, pero si el resultado de la primera expresión es 0, se toma el resultado de la segunda. Este operador es asociativo por la derecha, y tiene menor prioridad de operación que los demás operadores aritméticos binarios.

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

`print ( 1+2*3-7 ?: 25 );`	`t0 = 2 * 3;`
	`t1 = 1 + t0;`
	`t2 = t1 - 7;`
	`t3 = t2;`
	`if (t2 != 0) goto L0;`
	`t3 = 25;`
	`L0:`
	`print t3;`
		25

`print ( 1+2*3-7 ?: 1+2*3 ?: 2*5 );`	`...`	**7**

`int x;`
`int y;`
`int z;`
`x = 1+2*3;`
`y = 2*2+3;`
`z = x-y ?: (y-x ?: x*y) ?: x+y ;`
`print(z);`
	`...`	**49**

* Sentencia ***switch-case*** al estilo Java. Inicialmente, es más fácil asumir que en todos los casos debe haber una sentencia ***break***, que detiene la ejecución, es decir, inicialmente (ejemplos 1 y 2), puede asumirse que la palabra reservada ***break*** forma parte de la sentencia ***switch***. Sin embargo, para obtener la máxima calificación en este apartado, se debe intentar programarla de manera que la sentencia ***break*** sea opcional. Si en un determinado caso no aparece la sentencia ***break***, se continúa ejecutando el resto de sentencias, es decir, la primera vez que se cumple una condición se ejecutan todas las sentencias restantes hasta el final del ***switch*** (ejemplos 3, 4, 5 y 6). En todos los casos, al final del ***switch*** puede aparecer opcionalmente una cláusula ***default.*** El valor de comparación puede ser una constante (ejemplos 1, 2, 3, 5 y 6), o una expresión tanto en el valor de comparación como en cada uno de los casos (ejemplo 4). En todos los ***case*** puede haber ninguna, una o más sentencias, incluyendo otras sentencias ***switch-case*** (ejemplo 5 y 6).

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

`int x;`
`x=2;`
`int y;`
`y=x;`
`switch (x) {`
`  case 1:`
`  y=y+1;`
`  break;`
`  case 2:`
`  y=y+2;`
`  break;`
`  case 3:`
`  y=y+3;`
`  break;`
`}`
`print(x*y);`
	`  x = 2;`
	`  y = x;`
	`  if (x != 1) goto L1;`
	`  t0 = y + 1;`
	`  y = t0;`
	`  goto L0;`
	`L1:`
	`  if (x != 2) goto L3;`
	`L2:`
	`  t1 = y + 2;`
	`  y = t1;`
	`  goto L0;`
	`L3:`
	`  if (x != 3) goto L5;`
	`L4:`
	`  t2 = y + 3;`
	`  y = t2;`
	`  goto L0;`
	`L5:`
	`L6:`
	`L0:`
	`  t3 = x * y;`
	`  print t3;`
		`8`

`int x;`
`x=10;`
`int y;`
`y=x;`
`switch (x) {`
`  case 1:`
`  y=y+1;`
`  break;`
`  case 2:`
`  y=y+2;`
`  break;`
`  case 3:`
`  y=y+3;`
`  break;`
`  default:`
`  y=y+10;`
`}`
`print(x*y);`
	`  x = 10;`
	`  y = x;`
	`  if (x != 1) goto L1;`
	`  t0 = y + 1;`
	`  y = t0;`
	`  goto L0;`
	`L1:`
	`  if (x != 2) goto L3;`
	`L2:`
	`  t1 = y + 2;`
	`  y = t1;`
	`  goto L0;`
	`L3:`
	`  if (x != 3) goto L5;`
	`L4:`
	`  t2 = y + 3;`
	`  y = t2;`
	`  goto L0;`
	`L5:`
	`  t3 = y + 10;`
	`  y = t3;`
	`L6:`
	`L0:`
	`  t4 = x * y;`
	`  print t4;`
		`200`

`int x;`
`x=2;`
`int y;`
`y=x;`
`switch (x) {`
`  case 1:`
`  y=y+1;`
`  case 2:`
`  y=y+2;`
`  case 3:`
`  y=y+3;`
`  default:`
`  y=y+10;`
`}`
`print(x*y);`
	`...`	`34`

int x;
x=2;
int y;
y=x;
switch ((x+x)/2) {
  case 1:
  y=y+1;
  case 1+1:
  y=y+2;
  case 1+2+3:
  y=y+3;
  break;
  default:
  y=y+10;
}
print(x*y);
	...	14

int x;
x=2;
int y;
y=1;
switch (x) {
  case 1:
  x = x*2;
  case 2:
  switch (y) {
  case 1:
  x = x+1;
  case 2:
  x = x+1;
  break;
  default:
  y = 5;
  }
  y=x+2;
  case 3:
  y=y+3;
  break;
}
print(x);
print(y);
	...	4
		9

int x;
x=3;
int y;
y=x;
switch (x) {
  case 1:
  break;
  case 2:
  case 3:
  case 4:
  x=x+2;
  y=y+3;
  y=y+4;
  break;
  case 5:
  case 6:
  y=y+7;
  default:
  y=y+10;
}
print(x);
print(y);
	...	5
		10

* Introducción del tipo ***puntero (o referencia)***. Los ***punteros*** en PLX funcionan de forma parecida a como funcionan en C, salvo que se supone que el alojamiento y desalojo de objetos se hace de forma implícita, es decir, que no hay que reservar memoria previamente para usar un puntero (no hay que usar ***malloc*** o ***new***), ni liberarla una vez terminado su uso (no hay que usar ***free***). Las variables de tipo puntero necesitan ser declaradas como tales. (ejemplo 1). Puede usarse el operador `&` para obtener la dirección de memoria correspondiente a una variable (ejemplo 2), y se aceptan punteros a punteros en cualquier nivel de anidación (ejemplo 3). Debe realizarse comprobación de tipos en las asignaciones y expresiones, (ejemplo 4) pero no se acepta aritmética de punteros (ejemplo 5), ni la aplicación del operador `&` para obtener direcciones de otra cosa que no sean variables (ejemplo 6).

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

`int *p;`
`int *q;`
`*p = 7;`
`q = p;`
`print (*p * *q);`
	`* p = 7;`
	`q = p;`
	`t0 = * p;`
	`t1 = * q;`
	`t2 = t0 * t1;`
	`print t2;`
		49

`int x;`
`x = 7;`
`int *p;`
`int *q;`
`p = &x;`
`q = p;`
`print (*p * *q);`
	`x = 7;`
	`t0 = & x;`
	`p = t0;`
	`q = p;`
	`t1 = * p;`
	`t2 = * q;`
	`t3 = t1 * t2;`
	`print t3;`
		49

`int x;`
`x = 7;`
`int *p;`
`int **q;`
`int ***r;`
`p = &x;`
`q = &p;`
`r = &q;`
`print (*p * ***r);`
	`x = 7;`
	`t0 = & x;`
	`p = t0;`
	`t1 = & p;`
	`q = t1;`
	`t2 = & q;`
	`r = t2;`
	`t3 = * p;`
	`t4 = * r;`
	`t5 = * t4;`
	`t6 = * t5;`
	`t7 = t3 * t6;`
	`print t7;`
		49

`int x;`
`x = 7;`
`int y;`
`y = 1;`
`int *p;`
`int **q;`
`int ***r;`
`int ****s;`
`p = &x;`
`q = &p;`
`r = &q;`
`s = &r;`
`y=2;`
`**q = y;`
`y=3;`
`**r = &y;`
`y=4;`
`***s = &y;`
`y=5;`
`****s = 6;`
`print (x);`
`print (y);`
`print (*p);`
`print (**q);`
`print (***r);`
`print (****s);`
	`x = 7;`
	`y = 1;`
	`$t0 = & x;`
	`p = $t0;`
	`$t1 = & p;`
	`q = $t1;`
	`$t2 = & q;`
	`r = $t2;`
	`$t3 = & r;`
	`s = $t3;`
	`y = 2;`
	`$t4 = * q;`
	`* $t4 = y;`
	`y = 3;`
	`$t5 = * r;`
	`$t6 = & y;`
	`* $t5 = $t6;`
	`y = 4;`
	`$t7 = * s;`
	`$t8 = * $t7;`
	`$t9 = & y;`
	`* $t8 = $t9;`
	`y = 5;`
	`$t10 = * s;`
	`$t11 = * $t10;`
	`$t12 = * $t11;`
	`* $t12 = 6;`
	`print x;`
	`print y;`
	`....`
		2
		6
		6
		6
		6
		6

int x;
x = 7;
int *p;
int *q;
int *r;
p = &x;
q = &x;
r = &x;
r = p+q; // Error
print (*p * *q * *r);
	…
	error;
	…
		--

int x;
x = 3;
int *px;
int **ppx;
px = &x;
ppx = &px;
ppx = & & x; // Error
print(ppx);**
	…
	error;
	…
		--- 	

EL CÓDIGO OBJETO (Código de tres direcciones):

El código objeto es a su vez una extensión del código intermedio utilizado en la práctica 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 = *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 objeto `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.
`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.