Práctica. Cuarta fase

Finalización del procesador para Tiny

En esta última fase debe completarse el desarrollo de un procesador para **Tiny**. Para ello deberá realizarse el siguiente trabajo:

1) Una memoria con los siguientes aspectos:
   - Especificación del procesamiento de vinculación.
   - Especificación del procesamiento de pre-tipado.
   - Especificación del procesamiento de comprobación de tipos.
   - Especificación del procesamiento de asignación de espacio.
   - Descripción del repertorio de instrucciones de la máquina-p necesario para soportar la traducción de **Tiny** a código-p.
   - Especificación del procesamiento de etiquetado.
   - Especificación del procesamiento de generación de código.
2) Una implementación del procesador para **Tiny**. Dicho procesador integrará los constructores de ASTs desarrollados en la fase 3. El programa dirigido al usuario recibirá como argumento (i) el archivo a analizar; (ii) una opción *op* que indique el constructor de árboles a aplicar (si *op* es `desc` el constructor a aplicar será el descendente; si es `asc` será el ascendente). Entonces:
   - El procesador aplicará el constructor de árboles elegido para construir el AST. En caso de detectar errores durante esta fase mostrará un mensaje legible del primer error (léxico o sintáctico) detectado, y terminará la ejecución.
   - Si el procesador ha construido satisfactoriamente el AST, aplicará sobre el mismo el procesamiento de vinculación. En caso de descubrir errores de vinculación, mostrará mensajes legibles de **todos** los errores encontrados. Dichos mensajes deberán ir acompañados de la fila y la columna en el texto fuente donde se ha producido el error. Una vez finalizado el procesamiento, si ha detectado algún error, deberá terminar la ejecución del procesador.
   - Si la vinculación se ha realizado correctamente, el procesador llevará a cabo el procesamiento de pretipado, que comprobará el cumplimiento de las reglas de *pretipado* del lenguaje. Lo mismo que en el caso anterior, si descubre errores deberá mostrar mensajes legibles de **todos** los errores descubiertos (junto con su localización en el texto fuente). En caso de detectarse errores, al finalizar el procesamiento el procesador detendrá su ejecución.
   - Si el pretipado se ha realizado correctamente, el procesador llevará a cabo el procesamiento de comprobación de tipos. Lo mismo que en el caso anterior, si descubre errores deberá mostrar mensajes legibles de **todos** los errores descubiertos (junto con su localización en el texto fuente). En caso de detectarse errores, al finalizar el procesamiento el procesador detendrá su ejecución.
   - Una vez superadas con éxito las fases de vinculación y de comprobación de tipos, el procesador realizará el procesamiento de asignación de espacio, seguido del procesamiento de etiquetado, seguido del procesamiento de generación de código. Durante este último procesamiento, poblará la memoria de datos de la máquina-p.
   - Por último, el procesador invocará la ejecución de la máquina-p, con el fin de proceder a ejecutar el programa procesado.

   El procesador se adaptará, además, para su ejecución en DomJudge (ver **apéndice A**).

   Para realizar los procesamientos, pueden elegirse los estilos de procesamiento que se estimen más oportunos, e, incluso, combinar estilos de procesamiento (no obstante, se recomienda que el estilo de procesamiento principal sea el visitante, utilizándose programación recursiva para procesamientos minoritarios; el estilo basado en el patrón "intérprete" se desaconseja).
3) Una implementación de un emulador para la máquina-p apropiado para ejecutar programas **Tiny**. Para ello, podrá completarse el emulador básico que se proporciona como código de apoyo (dicho emulador proporciona ya soporte para todas las características de la máquina-p, incluida un mecanismo básico de gestión de memoria dinámica, e instrucciones para el manejo de registros de activación).

Fecha límite de entrega: **Viernes 9 de mayo de 2024, a las 11:59 pm**.

Modo de entrega: A través del campus virtual, en un único .zip. Dicho archivo debe contener: (i) un documento PDF `memoria.pdf` con la memoria requerida en el punto 1) del trabajo a realizar; (ii) una carpeta `implementacion`, en el interior de la cuál debe incluirse toda la implementación requerida por el procesador (incluyendo el emulador de la máquina-p); (iii) una carpeta `pruebas` con distintos programas de prueba que permitan probar el procesador. La entrega debe ser realizada solamente por un miembro del grupo.
Las implementaciones deberán, además, entregarse a través del juez DomJudge de la asignatura, siguiendo las indicaciones que se proporcionan en el **apéndice B**.

Apéndice A. **Preparación del procesado para su ejecución en DomJudge**

Sección A.1: Requisitos para procesar programas correctos en el juez

- La entrada esperada por el juez:
  1) Comienza por una línea de selección del método de construcción de ASTs (a para el constructor ascendente implementado con cup+jflex, o d para el constructor descendente implementado con javacc, como en la fase 3).
  2) A continuación, se incluye el programa a procesar
  3) Por último, **en caso de que el programa realice lecturas, se incluye $$ y una línea con cada dato que el programa debe leer** (si el programa no lee datos, esta última sección puede omitirse).

Ejemplo 1 (programa sin datos)	Ejemplo 2 (programa con datos)

a
{
int x;
bool y;
string z;
real k
&&
@ x = 25;
@ y = true;
@ z = "Esto es un mensaje";
@ k = 25.6;
write x; nl;
write y; nl;
write z; nl;
write k; nl
}
	a
	{
	int x;
	int y;
	int z
	&&
	read x;
	write "LEIDO: "; write x; nl;
	read y;
	write "LEIDO: "; write y; nl;
	read z;
	write "LEIDO: "; write z
	}
	$$
	56
	-45
	789

En el ejemplo del programa con datos, el primer `read` leerá el primer valor (56), el segundo el segundo valor (-45), y el tercero el tercer valor (789).

- La salida será la producida por la ejecución del programa, o bien información sobre posibles errores, en caso de que el programa no haya podido compilar (los convenios seguidos en el caso de detectar errores se describirán en la sección A.2 de este apéndice).

Salida para el ejemplo 1	Salida para el ejemplo 2
25 true Esto es un mensaje 25.6	LEIDO: 56 LEIDO: -45 LEIDO: 789

- Para conseguir este comportamiento, deben realizarse las siguientes extensiones en las implementaciones:
  1) `$$` debe reconocerse como EOF. Para ello:

- En la especificación para `jflex` debe definirse una nueva clase léxica (eof), como $, y devolver una "unidadEof()" cuando se reconozca un $. Se trata un único $ como EOF, y no los dos, porque el analizador generado por CUP invoca al analizador léxico dos veces cuando recibe EOF (por tanto, esto permitirá que reciba los dos *eofs* que necesita para terminar el análisis).

Extensiones a realizar en la entrada para **jflex**

```
…
… // se añade una nueva definición regular, eof, definida como $
eof = \$
…
… // se añade un nuevo patrón, para devolver EOF cuando se reconoce $
{eof} {return ops.unidadEof();}
…
```

- En la especificación javacc se añade una alternativa para reconocer el programa: que el programa vaya seguido por $$.

Ejemplo de regla inicial en la especificación javacc original	Extensión a realizar para que funcione con programas que tienen datos a continuación

`Prog inicio() :`
`  {Prog p;}`
`  {p=programa() <EOF> {return p;}}`
	`Prog inicio() :`
	`  {Prog p;}`
	`  {p=programa() **fin()** {return p;}}`
	`void fin()  :`
	`  {}`
	`  {**<EOF>** | **"$$"**}`

2) El analizador léxico debe utilizar un *reader* que consuma únicamente un carácter cada vez. Por motivos de eficiencia, el analizador generado por jflex lee, de ser posible, varios caracteres en cada operación de lectura. Esto, sin embargo, puede originar que, en la última lectura, se lean parte de los datos. Para evitarlo hay que utilizar un *reader* como el indicado. Esto puede conseguirse especializando `InputStreamReader` como sigue:

```
class BISReader extends InputStreamReader {
    public BISReader(InputStream is) {
        super(is);
    }
    @Override
    public int read(char[] cbuf,
                    int offset,
                    int length) throws IOException {
        int c = read();
        if (c == -1) return -1;
        else {
          cbuf[offset] = (char) c;
          return 1;
        }
    }
}
```

En esta especialización siempre se lee un único carácter, independientemente de que se soliciten la lectura de varios. Con ello, aunque se realizarán más operaciones de lectura, podemos garantizar que el analizador léxico nunca consumirá trozos de los datos (`BISReader` viene de `BlockingInputStreamReader`, en el sentido de que se comporta como si se estuviera leyendo por consola).

3) La máquina-P debe utilizar, para implementar las instrucciones de lectura, el mismo *stream* que el que utiliza el analizador léxico. Esto es debido a que el *stream* también puede mantener un *buffer* interno que, una vez finalizado el

análisis léxico, podría contener parte de los datos. Esto no es problemático, ya que dicha información se irá consumiendo en las sucesivas operaciones de lectura, pero sí obliga a utiliza un único *reader*, común al analizador y a la máquina-P. En la **sección A.3** se muestra un ejemplo de clase DomJudge que tiene en cuenta todas estas consideraciones:

4) La implementación de la entrada/salida en la máquina-p debe ser consistente con los convenios esperados por el juez. En concreto, cada dato leído estará en una línea. Se recomienda utilizar, para realizar las lecturas, la clase `java.util.Scanner` como envoltorio al *reader* pasado a la máquina-p en el constructor. De esta forma, en las respectivas instrucciones de la máquina-p:
   - La lectura de un entero puede realizarse con `nextInt()`, seguido de `nextLine()` para descartar el fin de línea.
   - La lectura de un real puede realizarse con `nextFloat()`, seguido de `nextLine()` para descartar el fin de línea.
   - La lectura de un *string* puede realizarse directamente con `nextLine()`.

5) Debe modificarse la máquina-P para que acepte, en su constructor, el *reader* del que debe leer.

```
…
  private Scanner input; // flujo que representa la entrada de la máquina-P
…
   public MaquinaP(Reader input, int tamdatos, int tampila, int tamheap, int ndisplays) {
…
      this.input = new Scanner(input);
…
   }
…
```

Sección A.2: Informando al juez sobre los errores detectados

- El juez presupone el siguiente modelo de procesamiento para comprobar las restricciones de la semántica estática:
  - Se comprueban las restricciones de vinculación (que todo uso de identificador esté declarado). Si se descubren errores de vinculación, se informa de los mismos, y se termina la ejecución.
  - Si no se descubren errores de vinculación, se comprueban las restricciones de pretipado (definiciones de tipos *struct* sin campos duplicados, dimensiones de tipos array no negativas, definiciones de tipos mediante identificadores vinculadas a declaraciones de tipos). Si se descubren errores de pretipado, se informa de los mismos, y se termina la ejecución.
  - Si no se descubren errores de vinculación ni de pretipado, se comprueban el resto de restricciones (restricciones de tipo). Si se descubren errores de tipado, se informa de los mismos y se termina la ejecución.
- Si durante la comprobación de las restricciones de vinculación se descubren errores, se producen mensajes `Errores_vinculacion fila: X col: Y`, uno por cada nodo del AST al que se han referido errores descubiertos, ordenados primero por fila, y, para filas iguales, por columna.
- En el caso de que, durante la comprobación de las restricciones de pretipado, se descubran errores, los mensajes emitidos son del tipo `Errores_pretipado fila: X col: Y` (se ordenan como los de vinculación).

- En el caso de que, durante la comprobación de las restricciones de tipado, se descubran errores, los mensajes emitidos son del tipo `Errores_tipado fila: X col: Y` (se ordenan como los de vinculación y los de pretipado).
- Indicaciones:
    - Para adaptar la implementación a su prueba en el juez, se aconseja centralizar la emisión de mensajes de error en una única clase, y, en caso de funcionamiento adaptado al juez, utilizar una especialización que marque que en el nodo del AST al que se refiere el error hay un error.
    - Una vez finalizado el procesamiento, en caso de error el AST puede procesarse para generar los mensajes esperados en el juez, en el orden esperado.
- Casos de prueba. Para cada caso, se proporciona una versión que funciona con el constructor de ASTs ascendente, y otra que funciona con el constructor de ASTs descendentes. La salida para cada caso es la misma, independientemente de la modalidad de construcción.
    - Caso 1: `01_errores_vinculacion_`*x*. Este programa tiene una primera parte que no presenta errores de vinculación, y una segunda parte que sí presenta este tipo de errores. Al ser procesado, el procesamiento de vinculación descubre dichos errores, por lo que no se continua con el procesamiento. Se imprime la secuencia de avisos de error esperados por el juez.

      Para facilitar la comprobación, a continuación se incluye una explicación de cada error de vinculación reportado (esta explicación es el mensaje de error emitido por el procesador desarrollado por el profesor funcionando en modo *para humanos*).

Mensaje emitido para juez	Mensaje emitido *para humanos*
`Errores_vinculacion fila:48 col:3`	`48,3:identificador no declarado:l`
`Errores_vinculacion fila:48 col:7`	`48,7:identificador no declarado:l3`
`Errores_vinculacion fila:49 col:4`	`49,4:identificador no declarado:nd1`
`Errores_vinculacion fila:49 col:11`	`49,11:identificador no declarado:nd2`
`Errores_vinculacion fila:49 col:18`	`49,18:identificador no declarado:nd3`
`Errores_vinculacion fila:49 col:27`	`49,27:identificador no declarado:nd4`
`Errores_vinculacion fila:49 col:35`	`49,35:identificador no declarado:nd5`
`Errores_vinculacion fila:49 col:41`	`49,41:identificador no declarado:nd6`
`Errores_vinculacion fila:49 col:47`	`49,47:identificador no declarado:nd7`
`Errores_vinculacion fila:49 col:54`	`49,54:identificador no declarado:nd8`
`Errores_vinculacion fila:49 col:61`	`49,61:identificador no declarado:nd9`
`Errores_vinculacion fila:49 col:68`	`49,68:identificador no declarado:nd10`
`Errores_vinculacion fila:50 col:12`	`50,12:identificador no declarado:nd11`
`Errores_vinculacion fila:50 col:17`	`50,17:identificador no declarado:nd12`
`Errores_vinculacion fila:50 col:33`	`50,33:identificador no declarado:nd13`
`Errores_vinculacion fila:51 col:6`	`51,6:identificador no declarado:nd14`
`Errores_vinculacion fila:51 col:13`	`51,13:identificador no declarado:nd15`
`Errores_vinculacion fila:51 col:20`	`51,20:identificador no declarado:nd16`
`Errores_vinculacion fila:51 col:31`	`51,31:identificador no declarado:nd17`
`Errores_vinculacion fila:51 col:38`	`51,38:identificador no declarado:nd18`
`Errores_vinculacion fila:51 col:44`	`51,44:identificador no declarado:nd19`
`Errores_vinculacion fila:51 col:49`	`51,49:identificador no declarado:nd20`
`Errores_vinculacion fila:52 col:4`	`52,4:identificador no declarado:x`
`Errores_vinculacion fila:52 col:16`	`52,16:identificador no declarado:y`
`Errores_vinculacion fila:53 col:4`	`53,4:identificador no declarado:x`
`Errores_vinculacion fila:53 col:12`	`53,12:identificador no declarado:y`
`Errores_vinculacion fila:53 col:32`	`53,32:identificador no declarado:z`
`Errores_vinculacion fila:54 col:7`	`54,7:identificador no declarado:x`
`Errores_vinculacion fila:54 col:22`	`54,22:identificador no declarado:y`

Errores_vinculacion fila:55 col:6	55,6:identificador no declarado:noesta
Errores_vinculacion fila:57 col:4	57,4:identificador no declarado:entero
Errores_vinculacion fila:60 col:7	60,7:declaracion duplicada:una
Errores_vinculacion fila:61 col:12	61,12:declaracion duplicada:dos
Errores_vinculacion fila:62 col:8	62,8:declaracion duplicada:una
Errores_vinculacion fila:62 col:39	62,39:declaracion duplicada:tres

- Caso 2: `02_errores_pretipado_x`. Este programa no tiene errores de vinculación, pero sí tiene errores de pretipado. Se imprime la secuencia de avisos de error esperados por el juez, y se interrumpe el procesamiento.
  A continuación se muestra la lista de mensajes emitida para el juez, junto con los correspondientes mensajes emitidos por la versión para *humanos*.

Mensaje emitido para juez	Mensaje emitido *para humanos*
Errores_pretipado fila:5 col:7	5,7:la dimension no puede ser negativa
Errores_pretipado fila:9 col:6	9,6:campo duplicado:ia
Errores_pretipado fila:10 col:6	10,6:campo duplicado:ib
Errores_pretipado fila:13 col:7	13,7:la dimension no puede ser negativa
Errores_pretipado fila:14 col:7	14,7:campo duplicado:a
Errores_pretipado fila:15 col:7	15,7:campo duplicado:b
Errores_pretipado fila:20 col:7	20,7:campo duplicado:a
Errores_pretipado fila:22 col:1	22,1:v1 no esta declarado como un tipo
Errores_pretipado fila:23 col:33	23,33:campo duplicado:a
Errores_pretipado fila:23 col:48	23,48:p no esta declarado como un tipo
Errores_pretipado fila:23 col:53	23,53:p1 no esta declarado como un tipo
Errores_pretipado fila:26 col:5	26,5:p2 no esta declarado como un tipo
Errores_pretipado fila:26 col:8	26,8:campo duplicado:a
Errores_pretipado fila:27 col:9	27,9:campo duplicado:a
Errores_pretipado fila:34 col:5	34,5:v2 no esta declarado como un tipo
Errores_pretipado fila:35 col:5	35,5:p no esta declarado como un tipo
Errores_pretipado fila:36 col:5	36,5:p no esta declarado como un tipo
Errores_pretipado fila:36 col:7	36,7:campo duplicado:a

- Caso 3: `03_errores_tipado1_x`. Programa sin errores de vinculación ni pretipado, pero con errores de tipos.
  Se muestra la lista de mensajes para el juez, con sus correspondencias para *humanos*.

Mensaje emitido para juez	Mensaje emitido *para humanos*
Errores_tipado fila:57 col:5	57,5:tipos incompatibles en asignacion
Errores_tipado fila:58 col:5	58,5:tipos incompatibles en operacion
Errores_tipado fila:59 col:7	59,7:tipos incompatibles en operacion
Errores_tipado fila:60 col:20	60,20:tipos incompatibles en operacion
Errores_tipado fila:61 col:5	61,5:tipos incompatibles en operacion
Errores_tipado fila:62 col:5	62,5:tipos incompatibles en operacion
Errores_tipado fila:63 col:6	63,6:tipos incompatibles en operacion
Errores_tipado fila:64 col:5	64,5:tipo incompatible en operacion
Errores_tipado fila:65 col:3	65,3:tipo incompatible en operacion
Errores_tipado fila:66 col:5	66,5:campo inexistente:x
Errores_tipado fila:67 col:4	67,4:tipos incompatibles en indexacion
Errores_tipado fila:68 col:4	68,4:tipos incompatibles en indexacion
Errores_tipado fila:69 col:5	69,5:se trata de acceder a un campo de un objeto que no es un registro
Errores_tipado fila:70 col:11	70,11:tipo incompatible con tipo de parametro formal
Errores_tipado fila:71 col:16	71,16:tipo incompatible con tipo de parametro formal
Errores_tipado fila:72 col:12	72,12:el tipo debe ser real
Errores_tipado fila:73 col:11	73,11:se esperaba un designador
Errores_tipado fila:74 col:7	74,7:p no es variable ni parametro
Errores_tipado fila:75 col:7	75,7:tr2 no es variable ni parametro
Errores_tipado fila:76 col:6	76,6:tipos incompatibles en asignacion
Errores_tipado fila:77 col:13	77,13:esperado tipo puntero
Errores_tipado fila:78 col:9	78,9:esperado tipo puntero
Errores_tipado fila:79 col:16	79,16:designador esperado
Errores_tipado fila:80 col:8	80,8:la parte izquierda debe ser un designador
Errores_tipado fila:81 col:6	81,6:valor no legible

Errores_tipado fila:82 col:7	82,7:valor no imprimible
Errores_tipado fila:83 col:4	83,4: esperada expresion booleana
Errores_tipado fila:84 col:7	84,7: esperada expresion booleana
Errores_tipado fila:87 col:10	87,10: esperada expresion booleana

- Caso 4: `04_errores_tipado2_x`. Otro programa sin errores de vinculación ni pretipado, pero con errores de tipos, en el que, además, se chequea el correcto comportamiento de la comprobación de compatibilidad estructural entre tipos.
  Se muestra la lista de mensajes para el juez, con sus correspondencias para *humanos*.

Mensaje emitido para juez	Mensaje emitido *para humanos*
Errores_tipado fila:23 col:7	23,7:el numero de parametros reales no coincide con el numero de parametros formales
Errores_tipado fila:24 col:7	24,7:l1 no es un subprograma
Errores_tipado fila:24 col:15	24,15:tipos incompatibles en operacion
Errores_tipado fila:24 col:27	24,27:tipos incompatibles en operacion

Sección A.3: Ejemplo de clase DomJudge

A continuación se muestra un ejemplo de clase DomJudge que tiene en cuenta las distintas consideraciones realizadas:

```
public class DomJudge {

       // Construccion del AST
   private static Prog construye_ast(Reader input, char constructor) throws Exception {
       if(constructor == 'a') {
         try {
           AnalizadorLexicoTiny alex = new AnalizadorLexicoTiny(input);
                // En esta fase no necesitamos volcar los distintos tokens leídos: utilizamos
               // directamente la clase ConstructorASTTiny, en lugar de su especialización
               // ConstructorASTTinyDJ, e invocamos a parse, en lugar de debug_parse.
           ConstructorASTTiny asint = new ConstructorASTTiny(alex);
           Prog p = (Prog)asint.parse().value;
           return p;
       }
        catch(ErrorLexico e) {
           System.out.println("ERROR_LEXICO");
        }
        catch(ErrorSintactico e) {
           System.out.println("ERROR_SINTACTICO");
           System.exit(0);
        }
       }
       else if(constructor == 'd') {
         try {
                // no necesitamos volcar los tokens: usamos directamente la clase generada
               // por javacc, y deshabilitamos la traza (por si estuviera activo DEBUG_PARSER.
           c_ast_descendente.ConstructorASTTiny asint =
                     new c_ast_descendente.ConstructorASTTiny(input);
           asint.disable_tracing();
           return asint.inicio();
       }
        catch(TokenMgrError e) {
           System.out.println("ERROR_LEXICO");
        }
        catch(ParseException  e) {
           System.out.println("ERROR_SINTACTICO");
           System.exit(0);
        }
       }
       else {
         System.err.println("Metodo de construccion no soportado:"+constructor);
       }
       return null;
   }


     // Procesamiento del AST
```

```
   public static void procesa(Prog p, Reader datos) throws Exception {
            // Utilizamos el módulo de gestión de errores adaptado a DomJudge
        Errores errores = new ErroresDJ();
        new Vinculacion(errores).procesa(p);
        if(! errores.hayError()) {
            new Pretipado(errores).procesa(p);
            if(! errores.hayError()) {
              new Tipado(errores).procesa(p);
              if(! errores.hayError()) {
                new AsignacionEspacio().procesa(p);
                new Etiquetado().procesa(p);
                MaquinaP m = new MaquinaP(datos,500,5000,5000,10);
                new GeneracionCodigo(m).procesa(p);
                m.ejecuta();
              }
              else {
                   // ImpresionErrores es un procesamiento que recorre
                   // el AST para descubrir nodos con errores, e imprimir
                   // los mensajes esperados por el juez
                new ImpresionErrores("Errores_tipado").procesa(p);
              }
            }
            else {
               new ImpresionErrores("Errores_pretipado").procesa(p);
            }
        }
        else {
            new ImpresionErrores("Errores_vinculacion").procesa(p);
        }
   }

   public static void main(String[] args) throws Exception {
        char constructor = (char)System.in.read();
        Reader r = new BISReader(System.in);
        Prog prog = construye_ast(r,constructor);
        if(prog != null) {
             procesa(prog, r);
        }
    }
}
```

En cuanto al tratamiento de errores, puede tenerse un módulo de gestión de errores para la implementación orientada al *humano*, que muestre los mensajes notificados por los diferentes procesamientos:

```
public class Errores {
    protected boolean hay_error;
    public Errores() {
        hay_error = false;
    }
    public boolean hayError() {
        return hay_error;
    }
          // Método para notificar errores
    public void error(Nodo n, String msg) {
      hay_error = true;
      System.err.println(n.leeFila()+","+n.leeCol()+":"+msg);
    }
}
```

y especializarla para que, en lugar de emitir un mensaje de error, fije en el modo del AST un *flag* que indica que, asociado con dicho nodo, se ha descubierto un error. Con ello, tras procesar el AST, puede recorrerse en el orden adecuado, para generar los mensajes que espera DomJudge:

```
public class ErroresDJ extends Errores {
        public void error(Nodo n, String msg) {
            hay_error = true;
            n.ponError();
        }

}
```

Apéndice B. Entrega a través de DomJudge

Para validar la implementación, se proporcionan distintos problemas en el juez. Debe enviarse la implementación, empaquetada tal y como se describe en las fases previas (fase 2, fase 3), a cada uno de los siguientes problemas:

- `07 FASE 4 - CASOS BASICOS`. Contienen distintos casos que prueban distintas características del lenguaje. Todos ellos involucran programas correctos.
- `08 FASE 4 - CASOS ELABORADOS`. Contiene un par de programas más elaborados (uno de ellos es el programa de ejemplo que aparece en la descripción del lenguaje). Ambos son programas correctos.
- `09 FASE 4 - ERRORES VINCULACION`. Contiene casos en los que aparecen errores de vinculación.
- `10 FASE 4 - ERRORES PRETIPADO`. Contiene casos libres de errores de vinculación, pero en los que aparecen errores de pretipado.
- `11 FASE 4 - ERRORES TIPADO`. Casos libres de errores de vinculación y pretipado, pero con errores de tipado.