- 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 otro documento).

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. A continuación se muestra un ejemplo de clase DomJudge que tiene en cuenta todas estas consideraciones:

```
public class DomJudge {
   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;
   }

   public static void procesa(Prog p, Reader datos) throws Exception {
       Errores errores = new Errores();
       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();
       }
   }
   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);
       }
    }
}
```

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);
…
   }
…
```