PROCESADORES DE LENGUAJES Segunda Convocatoria

Definición del Lenguaje

FF es un lenguaje de programación secuencial basado en el uso de funciones. El programa FF consta de una primera sección para declarar funciones y una segunda dedicada a la declaración de las variables e instrucciones del programa principal. Por ejemplo:

```
FUNCIONES   //declaración de funciones

   func insertar(entero e, lista(entero) l) dev lista(entero) :
        VARIABLES lista(entero) s;
        si (longitud(l)==0) s=[e];
        sino
           si (e>=primero(l)) s=[e|l];
          sino s=[primero(l)|insertar(e,resto(l))];
          fsi
       fsi
       dev s;
   ffunc

   ...
VARIABLES    //declaración de variables
   lista(entero) l, s;
   lista(lista(entero)) r;   ...

   l=[-2,3,1,2,8];    //instrucciones
   s=insertar(3,l);   ...
```

La declaración de función en FF tiene dos partes: cabecera y cuerpo. En la cabecera se define el nombre de la función, los parámetros y el tipo devuelto. En el cuerpo se declaran las variables (opcionales) y luego las instrucciones. Cada parámetro en la declaración de una función está precedido de su tipo. Ejemplo:

```
. func longitud(lista(entero) l) dev entero :  //cabecera
      si (l==[]) dev 0;                        //cuerpo
      sino dev longitud(resto(l))+1;
      fsi
   ffunc
```

FF dispone de tres tipos de datos: (a) *entero*, (b) *lógico* y (c) *lista*. Este último es un tipo paramétrico siendo su parámetro el tipo de los elementos de la lista (ej. `lista(entero)`, `lista(lista(entero))`).

Cada tipo dispone de un conjunto de operadores predefinidos.
El tipo entero dispone de las operaciones suma, resta, producto y división (entera).
El tipo lógico dispone de la conjunción, disyunción y negación.
El tipo lista dispone de dos constructores: uno que permite construir la lista por enumeración (`[e1,e2, ...]`) y otro para añadir elementos por el extremo izquierdo de la lista (`[elemento|lista]`). También se dispone de dos funciones predefinidas para deconstruir la lista: `primero(lista)` para obtener el primer elemento de la lista y `resto(lista)` para obtener la lista sin su primer elemento.

Para programar el cuerpo de una función o del programa principal, FF dispone de cuatro tipos de instrucciones: (a) asignación, (b) bifurcación, (c) devolución del valor de una expresión y (d) muestra por pantalla de valores de expresiones.

La instrucción *asignación* asocia el valor de una expresión a una variable. Por ejemplo, `r=[primero(l)|concatenar(resto(l),s)]` asocia a `r` el valor de la expresión `[primero(l)|concatenar(resto(l),s)]`.

La instrucción *bifurcación* permite seleccionar uno de los dos bloques de instrucciones vinculados al valor de verdad de una expresión lógica. Sintácticamente, la bifurcación tiene la forma:

```
si (expresión_logica) A sino B fsi
```

siendo `expresión_logica` la condición que permite bifurcar, `A` el bloque de instrucciones seleccionado si la condición es cierta y `B` el bloque de instrucciones seleccionado si la condición es falsa. Para expresar condiciones, FF incluye operadores relacionales para construir relaciones y operadores lógicos para conectar relaciones. Los operadores relacionales en FF son : mayor (`>`), menor (`<`), mayor o igual (`>=`), menor o igual (`<=`), igual (`==`) y distinto (`!=`). Los operadores lógicos en FF son : conjunción (`&&`), disyunción (`||`) y negación (`!`).

La instrucción *devolución* permite devolver el valor de una expresión en una función. Sintácticamente, la devolución tiene la forma: `dev exp` donde el valor de la expresión `exp` es devuelto al exterior de la función en la que ocurre.

La instrucción *muestra* permite mostrar por pantalla el valor de una secuencia de expresiones. Sintácticamente, la muestra tiene la forma: `mostrar(exp,...)` donde el valor de la expresión `exp` es mostrado por pantalla.
A continuación, se muestra un programa de ejemplo completo:

```
FUNCIONES
   func longitud(lista(entero) l) dev entero :
      si (l==[]) dev 0;
      sino dev longitud(resto(l))+1;
      fsi
   ffunc

   func esPositivo(entero e) dev logico :  dev (e>0); ffunc


   func ordenar(lista(entero) l) dev lista(entero) :
      VARIABLES lista(entero) s;
      si (contar(l)<=1)  s=l;
      sino  s=insertar(primero(l),ordenar(resto(l)));
      fsi
      dev s;
   ffunc

   func insertar(entero e, lista(entero) l) dev lista(entero) :
      VARIABLES lista(entero) s;
      si (longitud(l)==0) s=[e];
      sino
         si (e>=primero(l)) s=[e|l];
         sino s=[primero(l)|insertar(e,resto(l))];
         fsi
      fsi
      dev s;
   ffunc

   func cero() dev entero : dev 0; ffunc

   func suma(entero i, entero j) dev entero : dev i+j; ffunc


VARIABLES
   lista(entero) l,s;
   lista(lista(entero)) r;
   entero suma;

   l=[-2,3,1,2,8];
   mostrar(l);
   s=ordenar(l);
   mostrar(s);
   r=[s|[l]];
```

1) Análisis Léxico-Sintáctico (2.5 puntos).

SE PIDE:

1. Definición de los lexemas de FF **(1 punto)**

```
FUNCIONES: 'FUNCIONES';
FUNC: 'func';
FFUNC: 'ffunc';
VARIABLES: 'VARIABLES';

ENTERO: 'entero';
LOGICO: 'logico';
LISTA: 'lista';
CIERTO: 'cierto';
FALSO: 'falso';
SI: 'si';
SINO: 'sino';
FSI: 'fsi';

RESTO: 'resto';
PRIMERO: 'primero';

DEV: 'dev';
MOSTRAR: 'mostrar';

fragment NUEVALINEA: '\r'?'\n';
fragment DIGITO: [0-9];
fragment LETRA: [a-zA-Z];

SEPARADORES: (' '|'\t'|NUEVALINEA) -> skip ;
COM_LINEA: '//'(.)*? NUEVALINEA -> skip;
COM_BLOQUE: '/*'(.)*? '*/' -> skip;

IDENT: LETRA(LETRA|DIGITO)*;
NUMERO:('-')?(DIGITO)+;

Y:'&&';
O:'||';
NO:'!';

MAYOR:'>';
MENOR:'<';
MAYORIGUAL:'>=';
MENORIGUAL:'<=';
IGUAL:'==';
DISTINTO:'!=';

MAS:'+';
MENOS:'-';
POR:'*';
DIV:'/';

ASIG:'=';

PARENTESISABIERTO: '(';
PARENTESISCERRADO: ')';
CORCHETEABIERTO: '[';
CORCHETECERRADO: ']';
PUNTOYCOMA: ';';
```

```
COMA: ',';
PUNTO: '.';
DOSPUNTOS: ':';
BARRA: '|';
```

2. Gramática del lenguaje FF. **(1,5 puntos)**

```
programa: (decl_funciones)? (variables)? instrucciones

decl_funciones: FUNCIONES funciones

funciones: (funcion)+

funcion: FUNC cabecera DOSPUNTOS cuerpo FFUNC

cabecera: IDENT PARENTESISABIERTO (parametros)? PARENTESISCERRADO DEV tipo

parametros: parametro COMA parametros | parametro

parametro: tipo IDENT

cuerpo: (variables)? instrucciones

variables: VARIABLES (vars)+

idents: IDENT COMA idents | IDENT

vars: tipo idents PUNTOYCOMA

tipo: tipo_basico | tipo_lista

tipo_basico: ENTERO | LOGICO

tipo_lista: LISTA PARENTESISABIERTO tipo PARENTESISCERRADO

instrucciones: (instruccion)+

instruccion: asignacion | condicional | retorno | mostrar

asignacion: IDENT ASIG expresion PUNTOYCOMA

condicional: SI PARENTESISABIERTO expresion_logica PARENTESISCERRADO
             instrucciones
             (SINO instrucciones)?
             FSI

retorno: DEV expresion PUNTOYCOMA

mostrar: MOSTRAR PARENTESISABIERTO expresiones PARENTESISCERRADO PUNTOYCOMA

expresion :  expresion_entera | expresion_logica | expresion_lista

expresiones: expresion COMA expresiones | expresion

expresion_lista:
      RESTO PARENTESISABIERTO expresion_lista PARENTESISCERRADO
    | PRIMERO PARENTESISABIERTO expresion_lista PARENTESISCERRADO
    | llamada
    | CORCHETEABIERTO expresion BARRA expresion_lista CORCHETECERRADO
#CabeceraRestoLista
    | CORCHETEABIERTO (expresiones)? CORCHETECERRADO
    | IDENT
```

```
llamada: IDENT PARENTESISABIERTO (expresiones)? PARENTESISCERRADO

expresion_logica: expresion_logica Y expresion_logica
         | expresion_logica O expresion_logica
         | NO expresion_logica
         | PARENTESISABIERTO expresion_logica PARENTESISCERRADO
         | expresion_lista IGUAL expresion_lista
         | expresion_entera IGUAL expresion_entera
         | expresion_logica IGUAL expresion_logica
         | expresion_lista DISTINTO expresion_lista
         | expresion_entera DISTINTO expresion_entera
         | expresion_logica DISTINTO expresion_logica
         | PRIMERO PARENTESISABIERTO expresion_lista PARENTESISCERRADO
         | llamada
         | orden
         | IDENT
         | CIERTO
         | FALSO

orden:  expresion_entera MAYOR expresion_entera
     | expresion_entera MENOR expresion_entera
     | expresion_entera MAYORIGUAL expresion_entera
     | expresion_entera MENORIGUAL expresion_entera


expresion_entera: expresion_entera MAS expresion_entera
   | expresion_entera MENOS expresion_entera
   | expresion_entera POR expresion_entera
   | expresion_entera DIV expresion_entera
   | PARENTESISABIERTO expresion_entera PARENTESISCERRADO
   | PRIMERO PARENTESISABIERTO expresion_lista PARENTESISCERRADO
   | llamada
   | IDENT
   | NUMERO
```

2) Análisis Semántico (2.5 puntos).

Con el objetivo de calcular el tipo de una expresión en FF, se propone la siguiente gramática:

```
expresion: expresion op_aritm expresion                          #OpArit
    | expresion op_log expresion                                 #OpLog
    | PARENTESISABIERTO expresion PARENTESISCERRADO               #Par
    | PRIMERO PARENTESISABIERTO expresion PARENTESISCERRADO       #Primero
    | RESTO PARENTESISABIERTO expresion PARENTESISCERRADO         #Resto
    | CORCHETEABIERTO expresion BARRA expresion CORCHETECERRADO   #CabezaResto
    | CORCHETEABIERTO (expresiones)? CORCHETECERRADO              #Enumeracion
    | NO expresion                                               #Negacion
    | expresion op_ig expresion                                  #Igualdad
    | expresion op_ord expresion                                 #Orden
    | llamada                                                    #LlamadaFuncion
    | cte_log                                                    #CteLog
    | IDENT                                                      #Var
    | NUMERO                                                     #Num
    ;

llamada: IDENT PARENTESISABIERTO (expresiones)? PARENTESISCERRADO ;
expresiones: expresion COMA expresiones | expresion ;
op_aritm: MAS | MENOS | POR | DIV ;
op_log: Y | O ;
op_ig: IGUAL | DISTINTO ;
op_ord: MAYOR | MENOR | MAYORIGUAL | MENORIGUAL ;
cte_log: CIERTO | FALSO ;
```

y las siguientes decisiones:

> Se define la notación `tipo1, … tipok -> tipo` para representar el tipo de una función con k parámetros de entrada cuyos tipos respectivos son `tipo1, … tipok` y `tipo` como el tipo devuelto por dicha función. Por ejemplo, el tipo de la función:
>
> `func concatenar(lista(entero) l, lista(entero) s) dev lista(entero)...`
>
> se representa como
>
> `lista(entero), lista(entero)->lista(entero)`

(1) Se supone que los tipos de las variables están almacenados en una memoria. El tipo de la variable se obtiene mediante consulta a dicha memoria. Si la variable no está en la memoria, su tipo es indefinido.

(2) Se supone que los tipos de las funciones están almacenados en una memoria. Si una llamada usa una función que no está en la memoria, su tipo es indefinido. En caso contrario, toda llamada a una función de tipo `tipo1, ..., tipok -> tipo` es una expresión de tipo `tipo` si las expresiones usadas como parámetros de entrada son de tipo `tipo1,..., tipok` respectivamente. En otro caso, la llamada tiene tipo indefinido.

(3) Toda lista construida por extensión (ej `[-2,3,1,2,x]`) es una expresión de tipo `lista(T)` si todos los elementos de la lista son de tipo `T` . Por ejemplo, `[-2,3,1,2,2]` es una expresión de tipo `lista(entero)` y `[[-2,3],[1],[],[2,8]]` es una expresión de tipo `lista(lista(entero))`). La lista vacía es un símbolo que puede actuar como lista de enteros, o de lógicos o de listas.

(4) Toda lista construida mediante cabecera y resto (ej. `[e|l]`) es una expresión de tipo `lista(T)` si la cabecera es de tipo `T` y el resto de tipo `lista(T)`. Por ejemplo, `[1|[2,3,4,5]]` es una expresión de tipo `lista(entero)` y `[e|l]` es una expresión de tipo `lista(lista(entero))` suponiendo `e` de tipo `lista(entero)` y `l` de tipo `lista(lista(entero))`.

(5) El primero de una lista es una expresión de tipo `T` si los elementos de la lista son de tipo `T`.

(6) El resto de una lista de tipo `lista(T)` es una lista de tipo `lista(T)`

(7) Toda expresión con operador aritmético binario (mas, menos, por, división) es una expresión tipo `entero` sólo si sus subexpresiones son de tipo `entero`. Los números son expresiones de tipo `entero`.

(8) Toda expresión con operador lógico binario (conjunción, disyunción o negación) es una expresión tipo `logico` sólo si sus subexpresiones son de tipo `logico`. Las constantes lógicas son expresiones de tipo `lógico`.
(9) Toda expresión con operador igual o distinto es una expresión tipo `logico` sólo si sus subexpresiones son del mismo tipo definido (no pueden ser tipo indefinido).
(10) Toda expresión con operador de orden (mayor, menor, mayor o igual y menor o igual) es una expresión tipo `logico` sólo si sus subexpresiones son de tipo entero.
(11) Toda expresión sin tipo definido por las reglas anteriores tendrá tipo indefinido. Igualmente, una expresión es de tipo indefinido si alguna subexpresión es de tipo indefinido.

SE PIDE:

1) Gramática atribuida para un analizador semántico que calcule el tipo de `expresion` según las decisiones dadas. **(1,5 puntos**).

```
OBJETIVO
--------
   Analizador semántico de lenguaje FF que calcule el tipo de una expresión
   según las decisiones dadas.

DECISIONES Y GRAMATICA
----------------------
(1)    Se supone que los tipos de las variables están almacenados en una
    memoria.

                | variable   |       tipo         |
                 ------------------------------------
                |   l        |   lista(entero)    |
                |   b        |   logico           |

    El tipo de una variable se obtiene mediante consulta a dicha memoria.
    Si la variable no está en la memoria, su tipo es indefinido.

    expresion dev tipo:
              ...
            | IDENT {si (IDENT no está en memoria_variables) entonces
                       tipo = indefinido
                   sino
                       tipo = consulta tipo en memoria_variables para IDENT
                   fsi }


(2)    Se supone que los tipos de las funciones están almacenados en una memoria.

              | funcion   |           tipo            |
              -----------------------------------------
              |  sumatorio |   lista(entero) -> entero |
              |  cero      |                 -> entero |
              |  suma      |  entero, entero -> entero |

   Si una llamada usa una función que no está en la memoria, su tipo es indefinido.
   En caso contrario,  toda llamada a una función de tipo tipo1, ..., tipok  -> tipo
   es una expresión de tipo tipo sólo si las expresiones usadas como parámetros de
   entrada son de tipo tipo1,..., tipok respectivamente.

    expresion dev tipo:
              ...
            | tipo=llamada

    llamada dev tipo: { sec_tipos = secuencia vacia }
       IDENT PARENTESISABIERTO (sec_tipos=expresiones)? PARENTESISCERRADO
       {si (IDENT no está en memoria_funciones) entonces
          tipo = indefinido
```

```
        sino
           si (sec_tipos coincide con los tipos de los
                   parametros de IDENT en memoria_funciones) entonces
               tipo = consultar en memoria_funciones el tipo devuelto por IDENT
           sino
               tipo = indefinido
           fsi
        fsi }

    expresiones dev sec_tipos:
        tipo=expresion COMA sec_tipos=expresiones {añadir tipo al principio de
sec_tipos}
      | tipo=expresion {añadir tipo al principio de sec_tipos}
```

(3) Toda lista construida por extensión (ej [-2,3,1,2,x]) es una expresión de tipo lista(T) si todos los elementos de la lista tienen un tipo definido de tipo T . Por ejemplo, [-2,3,1,2,2] es una expresión de tipo lista(entero) y [[-2,3],[1],[],[2,8]] es una expresión de tipo lista(lista(entero))). La lista vacía es un símbolo polisémico que puede actuar como lista de enteros, de lógicos o de listas.

```
   expresion dev tipo: { sec_tipos = secuencia vacia }
             ...
           | CORCHETEABIERTO (sec_tipos=expresiones)? CORCHETECERRADO
             { si (sec_tipos es igual a secuencia vacia) entonces
                  tipo = lista  //cualquier tipo lista
               sino
                  si (algun tipo en sec_tipos es igual a indefinido) entonces
                     tipo = indefinido
                  sino
                     si (todos los tipos en sec_tipos coinciden en un tipo t) entonces
                        tipo = lista(t)
                     sino
                        tipo = indefinido
                     fsi
                  fsi
               fsi }
```

(4) Toda lista construida mediante cabecera y resto (ej. [e|l]) es una expresión de tipo lista(T) si la cabecera es de tipo definido T y el resto de tipo lista(T). Por ejemplo, [1|[2,3,4,5]] es una expresión de tipo lista(entero) y [e|l] es una expresión de tipo lista(lista(entero)) suponiendo e de tipo lista(entero)y l de tipo lista(lista(entero)).

```
    expresion dev tipo:
              ...
            | CORCHETEABIERTO tipo1=expresion BARRA tipo2=expresion CORCHETECERRADO
            { si (tipo1 y tipo2 están definidos)
                 si (tipo2 es lista(t) y tipo1 es t) entonces
                    tipo = lista(t)
                 sino
                    tipo = indefinido
                 fsi
              sino
                 tipo = indefinido
              fsi }
```

(6) El primero de una lista es una expresión de tipo definido T si los elementos de la lista son de tipo T.

```
    expresion dev tipo:
             ...
            | PRIMERO PARENTESISABIERTO tipo1 = expresion PARENTESISCERRADO
              { si (tipo1 es igual lista(t)) entonces
```

```
                tipo = t
              sino
                tipo = indefinido
              fsi }
```

(7) El resto de una lista de tipo lista(T) con T definido es una lista de tipo lista(T)

```
expresion dev tipo:
          ...
        | RESTO PARENTESISABIERTO tipo1=expresion PARENTESISCERRADO
          { si (tipo1 es igual lista(t)) entonces
              tipo = tipo1
            sino
               tipo = indefinido
            fsi }
```

(7) Toda expresión con operador aritmético binario (mas, menos, por, división) es una expresión tipo entero sólo si sus subexpresiones son de tipo entero. Los números son expresiones de tipo entero.

```
expresion dev tipo: tipo1=expresion op_aritm tipo2=expresion
   { si (tipo1 y tipo2 son enteros)
        tipo = entero
     sino
        tipo = indefinido
     fsi }
          ...

expresion dev tipo: ...
   | NUMERO { tipo = entero }
```

(8) Toda expresión con operador lógico binario (conjunción, disyunción o negación) es una expresión tipo logico sólo si sus subexpresiones son de tipo logico. Las constantes lógicas son expresiones de tipo lógico.

```
expresion dev tipo: ...
   | tipo1=expresion op_log tipo2=expresion
   { si (tipo1 y tipo2 son logicos)
       tipo = logico
    sino
       tipo = indefinido
    fsi }

expresion dev tipo: ...
   | NO tipo1 = expresion
   { si (tipo1 es igual a logico)
      tipo = logico
   sino
      tipo = indefinido
   fsi }

   expresion dev tipo: ...
          | cte_log { tipo = logico }
```

(9) Toda expresión con operador igual o distinto es una expresión tipo logico sólo si sus subexpresiones son del mismo tipo definido (no pueden ser tipo indefinido).

```
expresion dev tipo:
    | tipo1=expresion op_ig tipo2=expresion
    { si (tipo1 y tipo2 son iguales y definidos)
          tipo = logico
      sino
          si (tipo1 es lista(t) y tipo2 es lista o al revés)
             tipo = logico
          sino
             tipo = indefinido
          fsi
```

```
        fsi }
```

(10) Toda expresión con operador de orden (mayor, menor, mayor o igual y menor o igual) es una expresión tipo logico sólo si sus subexpresiones son de tipo entero.

```
expresion dev tipo:
        | tipo1=expresion op_ord tipo2=expresion
        { si (tipo1 y tipo2 son enteros)
              tipo = logico
          sino
              tipo = indefinido
          fsi }
```

(11) Toda expresión sin tipo definido por las reglas anteriores tendrá tipo indefinido. Igualmente, uan expresión es de tipo indefinido si alguna subexpresión es de tipo indefinido.
(Ya tenido en cuenta en las gramáticas correspondientes a las reglas anteriores)

2) Implementación Antlr4/Java de las decisiones (3) y (4) para la regla `expresion` de la gramática atribuida propuesta en `1)` **(1 punto)**

```
    //expresion dev tipo: { sec_tipos = secuencia vacia }
    //              ...
    //            | CORCHETEABIERTO (sec_tipos=expresiones)? CORCHETECERRADO
    //              { si (sec_tipos es igual a secuancia vacia) entonces
    //                   tipo = lista
    //                sino
    //                   si (algun tipo en sec_tipos es igual a indefinido)
entonces
    //                      tipo = indefinido
    //                   sino
    //                      si (todos los tipos en sec_tipos coinciden en un tipo
t) entonces
    //                         tipo = lista(t)
    //                      sino
    //                         tipo = indefinido
    //                      fsi
    //                   fsi
    //                fsi }
    @Override
    public Object visitEnumeracion(Anasint2.EnumeracionContext ctx) {
        List<String> tipo = new LinkedList<>();
        List<List<String>>sec_tipos = new LinkedList<>();
        if (ctx.expresiones()!=null)
            sec_tipos = (List<List<String>>)visitExpresiones(ctx.expresiones());
        if (sec_tipos.size()==0){
            tipo.add("lista");
        }
        else {
            if (sec_tipos.stream().anyMatch(e -> e.contains("indefinido")))
                tipo.add("indefinido");
            else {
                boolean coinciden=true;
                List<String>t=sec_tipos.get(0);
                int i=1;
                while(coinciden && i<sec_tipos.size()){
                    List<String>s=sec_tipos.get(i);

                    coinciden=coinciden && (t.equals(s) ||
                            (t.size()==1 && t.get(0).equals("lista") &&
s.get(0).equals("lista")) ||
                            (s.size()==1 && s.get(0).equals("lista") &&
```

```
t.get(0).equals("lista")));
                    if (s.size()>=t.size())
                        t=s;
                    i++;
                }
                if (coinciden) {
                    tipo.add("lista");
                    tipo.addAll(t);
                }
                else
                   tipo.add("indefinido");
            }
        }

        return tipo;
    }

    //expresion dev tipo:
    //              ...
    //            | CORCHETEABIERTO tipo1=expresion BARRA tipo2=expresion
CORCHETECERRADO
    //            { si (tipo1 y tipo2 están definidos)
    //                 si (tipo2 es lista(t) y tipo1 es t) entonces
    //                    tipo = lista(t)
    //                 sino
    //                    tipo = indefinido
    //                 fsi
    //              sino
    //                 tipo = indefinido
    //              fsi }
    @Override
    public Object visitCabezaResto(Anasint2.CabezaRestoContext ctx) {
        List<String> tipo = new LinkedList<>();
        List<String>tipo1=(List<String>)visit(ctx.expresion(0));
        List<String>tipo2=(List<String>)visit(ctx.expresion(1));
        if (!tipo1.contains("indefinido") && !tipo2.contains("indefinido")){
            List<String>aux=new LinkedList<>();
            aux.addAll(tipo2); aux.remove(0);
            if (aux.equals(tipo1) ||
                    (aux.size()==1 && aux.get(0).equals("lista") &&
tipo1.get(0).equals("lista")))
                tipo = tipo2;
            else
                tipo.add("indefinido");
        }
        else
            tipo.add("indefinido");
        return tipo;
    }
```

3) Intérprete (2.5 puntos).

Con el objetivo de interpretar una expresión simplificada (sin llamadas a funciones, ni relaciones de igualdad y orden) en el lenguaje FF, se propone la siguiente gramática:

```
expresion: expresion op_aritm expresion                          #OpArit
    | expresion op_log expresion                                 #OpLog
    | PARENTESISABIERTO expresion PARENTESISCERRADO              #Par
    | PRIMERO PARENTESISABIERTO expresion PARENTESISCERRADO       #Primero
    | RESTO PARENTESISABIERTO expresion PARENTESISCERRADO         #Resto
    | CORCHETEABIERTO expresion BARRA expresion CORCHETECERRADO  #CabezaResto
    | CORCHETEABIERTO (expresiones)? CORCHETECERRADO              #Enumeracion
    | NO expresion                                               #Negacion
    | cte_log                                                    #CteLog
    | IDENT                                                      #Var
    | NUMERO                                                     #Num
    ;

expresiones: expresion COMA expresiones | expresion ;
op_aritm: MAS | MENOS | POR | DIV ;
op_log: Y | O ;
cte_log: CIERTO | FALSO ;
```

y las siguientes decisiones:

(1) Se supone que los valores y tipos de las variables están almacenados en una memoria. La variable se interpreta como el valor almacenado en dicha memoria.
(2) Los números se interpretan como valores enteros y las constantes lógicas como valores booleanos.
(3) La lista dada por extensión (ej `[-2,3,1,2,x]`) se interpreta como lista de valores enteros/booleanos/lista si son listas de expresiones enteras/lógicas/listas. La lista vacía (ej `[]`) se interpreta como lista sin elementos.
(4) La lista construida mediante cabecera y resto (ej. `[e|l]`) se interpreta como lista cuyo primer elemento es el valor correspondiente a la cabecera y los restantes elementos corresponden a la interpretación del resto.
(5) El primero de una lista se interpreta como el valor correspondiente al primer elemento de la lista.
(6) El resto de la lista se interpreta como el valor de la lista sin su cabecera.
(7) La expresión con operador aritmético binario (suma, resta, producto o división) se interpreta como el valor de la operación correspondiente sobre los valores respectivos de las dos expresiones conectadas por el operador.
(8) La expresión con operador lógico binario (conjunción, disyunción o negación) se interpreta como el valor de la operación correspondiente sobre los valores respectivos de las dos expresiones conectadas por el operador.

SE PIDE:

1) Gramática atribuida para intérprete que calcule el valor de `expresion` según decisiones dadas. **(1,5 puntos**).

```
OBJETIVO
--------
   Intérprete de lenguaje FF que calcule el valor de una expresión
   según las decisiones dadas.

DECISIONES Y GRAMATICA
----------------------
(1)    Se supone que los valores y tipos de las variables están almacenados en una
memoria.
    La variable se interpreta como el valor almacenado en dicha memoria.

                | variable   |       tipo          |  valor
                 ----------------------------------------------
                |   l        |   lista(entero)     | [-1,2,4]
                |   b        |   logico            |  cierto
```

(2) Los números se interpretan como valores enteros y las constantes lógicas como valores booleanos.

```
expresion dev valor:
          ...
        | NUMERO { valor = interpretar NUMERO como valor entero }

expresion dev valor:
             ...
           | valor = cte_log

cte_log dev valor: CIERTO  { valor = cierto }
    | FALSO  { valor = falso }
```

(3) La lista dada por extensión (ej [-2,3,1,2,x]) se interpreta como lista de valores enteros/booleanos/lista si son listas de expresiones enteras/lógicas/listas.

```
expresion dev valor: { sec_valores = secuencia vacia }
          ...
        | CORCHETEABIERTO (sec_valores=expresiones)? CORCHETECERRADO
          { si (sec_valores es igual a secuencia vacia) entonces
               valor = lista vacia
            sino
               valor = lista con la secuencia de valores sec_valores
            fsi }

expresiones dev sec_valores:
       valor=expresion COMA sec_valores=expresiones {añadir valor al principio de
sec_valores}
     | valor=expresion {añadir valor al principio de sec_valores}
```

(4) La lista construida mediante cabecera y resto (ej. [e|l]) se interpreta como lista cuyo primer elemento es el valor correspondiente a la cabecera y los restantes elementos es valor de la lista correspondientes al resto.

```
expresion dev valor:
          ...
        | CORCHETEABIERTO valor1=expresion BARRA valor2=expresion CORCHETECERRADO
        { valor = añadir valor1 a la lista valor2 }
```

(5) El primero de una lista se interpreta como el valor correspondiente al primer elemento de la lista.

```
expresion dev valor:
          ...
        | PRIMERO PARENTESISABIERTO valor1 = expresion PARENTESISCERRADO
          { si valor1 no es lista vacia entonces  valor = primer elemento de valor1
sino valor = no hay elemento fsi }
```

(6) El resto de la lista se interpreta como el valor de la lista sin su cabecera.

```
expresion dev valor:
          ...
        | RESTO PARENTESISABIERTO valor1=expresion PARENTESISCERRADO
          { valor = valor1 sin su primer elemento }
```

(7) La expresión con operador aritmético binario (suma, resta, producto o división) se interpreta como el valor de la operación correspondiente sobre los valores respectivos de las dos expresiones conectadas por el operador.

```
expresion dev valor: valor1=expresion op_aritm valor2=expresion
   { valor = operacionAritmetica(valor1,valor2,op_aritm) }
          ...
```

(8) La expresión con operador lógico binario (conjunción, disyunción o negación) se interpreta como el valor de la operación correspondiente sobre los valores respectivos de las dos expresiones conectadas por el operador.

```
expresion dev valor: ...
   | valor1=expresion op_log valor2=expresion
   { valor = operacionLogica(valor1,valor2,op_log) }

expresion dev valor: ...
   | NO valor1 = expresion
   { valor = operacionLogicaNeg(valor1) }
```

2) Implementación Antlr4/Java de las decisiones (3) y (4) para la regla `expresion` de la gramática atribuida propuesta en `1)` **(1 punto)**

```
    //expresion dev valor: { sec_valores = secuencia vacia }
    //              ...
    //            | CORCHETEABIERTO (sec_valores=expresiones)? CORCHETECERRADO
    //              { si (sec_valores es igual a secuencia vacia) entonces
    //                   valor = lista vacia
    //                sino
    //                   valor = lista con la secuencia de valores sec_valores
    //                fsi }
    @Override
    public Object visitEnumeracion(Anasint2.EnumeracionContext ctx) {
        List<Object> valor = new LinkedList<>();
        List<Object>sec_valores = new LinkedList<>();
        if (ctx.expresiones()!=null)
            sec_valores = (List<Object>)visitExpresiones(ctx.expresiones());
        valor.addAll(sec_valores);
        return valor;
    }

    //expresion dev valor:
    //              ...
    //            | CORCHETEABIERTO valor1=expresion BARRA valor2=expresion
CORCHETECERRADO
    //            { valor = añadir valor1 a la lista valor2 }
    @Override
    public Object visitCabezaResto(Anasint2.CabezaRestoContext ctx) {
        List<Object> valor = new LinkedList<>();
        Object valor1=visit(ctx.expresion(0));
        List<Object> valor2=(List<Object>)visit(ctx.expresion(1));
        valor.add(valor1);
        valor.addAll(valor2);
        return valor;
    }
```

4) Compilador (2.5 puntos).

Con el objetivo de compilar en Java el código que calcule el valor de una expresión simplificada (sin contener llamadas a funciones, ni relaciones de igualdad y orden), se propone la siguiente gramática:

```
expresion: expresion op_aritm expresion                          #OpArit
   | expresion op_log expresion                                  #OpLog
   | PARENTESISABIERTO expresion PARENTESISCERRADO               #Par
   | PRIMERO PARENTESISABIERTO expresion PARENTESISCERRADO       #Primero
   | RESTO PARENTESISABIERTO expresion PARENTESISCERRADO         #Resto
   | CORCHETEABIERTO expresion BARRA expresion CORCHETECERRADO   #CabezaResto
   | CORCHETEABIERTO (expresiones)? CORCHETECERRADO              #Enumeracion
   | NO expresion                                                #Negacion
   | cte_log                                                     #CteLog
   | IDENT                                                       #Var
   | NUMERO                                                      #Num
   ;

expresiones: expresion COMA expresiones | expresion ;
op_aritm: MAS | MENOS | POR | DIV ;
op_log: Y | O ;
cte_log: CIERTO | FALSO ;
```

SE PIDE:

1. Comprensión del problema. Escriba el código Java generado por su compilador correspondiente a las expresiones:

```
(a)      [primero(l)|resto(l)]
(b)      [-1,[2],e]
(c)      (a+1)
(d)      (b && !c)
```

(1 punto)

```
OBJETIVO
--------
   compilador de lenguaje FF que calcule la traducción Java de una expresión.

COMPRENSIÓN DEL PROBLEMA
------------------------
1. Comprensión del problema. Escriba el código Java que el compilador genera para
siguientes las expresiones:

(a)    [primero(l)|resto(l)]     se corresponde con la variable aux2

    //código correspondiente a primero(l)
    Object aux0;
    aux0=l.get(0);
    //código correspondiente a resto(l)
    List<Object>aux1 = new LinkedList<>();
    aux1.addAll(l);
    aux1.remove(0);
    //código correspondiente a [primero(l)|resto(l)]
    List<Object>aux2 = new LinkedList<>();
    aux2.add(aux0);
    aux2.addAll(aux1);

(b)    [-1,[2],e]  se corresponde con la variable aux1

    //código correspondiente a [2]
```

```
    List<Object>aux0 = new LinkedList<>();
    aux0.add(2);
    //código correspondiente a [-1,[2],e]
    List<Object>aux1 = new LinkedList<>();
    aux1.addA(-1);
    aux1.add(aux0);
    aux1.add(e);

(c)    (a+1)  no se corresponde con ninguna variable

    //código correspondiente a (a+1)
    (a+1)

(d)    (b && !c)

    //código correspondiente a (b && !c)
    (b && !c)
```

2. Decisiones y Gramática atribuida para el compilador propuesto. **(1.5 puntos)**

```
DECISIONES
----------
De la compresión del problema se extraen las siguientes decisiones.

(1)    A las listas por enumeración (ej. [-1,2,4]) y a las listas construidas con
    cabecera y resto se les asocia una variable nueva auxi de tipo List<Object>.
    Esto se hace para facilitar la composición de código.
    En los ejemplos (a) y (b) de comprensión del problema puede comprobarse este punto.

(2)    Las subexpresiones de un orden también se le asocian variables nuevas auxi pero
    de tipo Integer ya que sólo se pueden comparar enteros en el lenguaje FF.
    En el ejemplo (c) de compresión del problema puede comprobarse este punto.

(3) Las subexpresiones en una igualdad/desigualdad también se le asocian variables
    nuevas auxi pero de tipo Object ya que se pueden comparar elementos de distinto
tipo
    en el lenguaje FF.
    En el ejemplo (d) de compresión del problema puede comprobarse este punto.

(4) A las funciones predefinidas primero y resto también se se asocian variables nuevas
    auxi detipo Object para primero y de tipo List<Object> para el resto.
    En el ejemplo (a) de compresión del problema puede comprobarse este punto.

(5) Al resto de expresiones en FF no se les asocian variables nuevas.

(6) La regla expresión supone un índice (global) por el que se indexarán
    las potenciales variables nuevas generadas en el código de la expresión y producirá
    dos informaciones de salida: (a) el código Java generado para la expresión y (b)
    la variable nueva asociada si procede (_ significa no asociada).

GRAMÁTICA ATRIBUIDA
-------------------
(1) código de las expresiones con operador binario aritmético:

     expresion dev cod, var:
          (cod1,var1)=expresion (cod_op,_)=op_aritm (cod2,var2)=expresion
          { si (var1 y var2 son distintas a _) entonces
               cod = cod1+"\n"+
                     cod2+"\n"+
                     var1+" "+cod_op+" "+var2
            sino
               si (var1 es distinta a _) entonces
                  cod = cod1+"\n"+
                        var1+" "+cod_op+" "+cod2
               sino
                  si (var2 es distinta a _) entonces
```

```
                    cod = cod2+"\n"+
                          cod1+" "cod_op+" "var2
                sino
                   cod = cod1+" "+cod_op+" "+cod2
                fsi
             fsi
          fsi
          var = _  }
```

(2) código de las expresiones con operador binario lógico:

```
     expresion dev cod, var:
              (cod1,var1)=expresion (cod_op,_)=op_log (cod2,var2)=expresion
         { si (var1 y var2 son distintas a _) entonces
            cod = cod1+"\n"+
                  cod2+"\n"+
                  var1+" "+cod_op+" "+var2
           sino
              si (var1 es distinta a _) entonces
                 cod = cod1+"\n"+
                    var1+" "+cod_op+" "+cod2
              sino
                  si (var2 es distinta a _) entonces
                     cod = cod2+"\n"+
                           cod1+" "+cod_op+" "+var2
                  sino
                      cod = cod1+" "+cod_op+" "+cod2
                  fsi
               fsi
           fsi
           var = _
       }
```

(3) código de las expresiones entre paréntesis:

```
     expresion dev cod, var:
           PARENTESISABIERTO (cod1,var1)=expresion PARENTESISCERRADO
           { si (var1 es distinto a _) entonces
                cod = cod1+"\n"+
                      "("+var1+")"
             sino
                cod = "("+cod1+")"
             fsi
             var = _  }
```

(4) código de las expresiones PRIMERO:

```
        expresion dev cod, var:
            PRIMERO PARENTESISABIERTO (cod1,var1)=expresion PARENTESISCERRADO
            { si (var1 es igual a _) entonces
                cod = "Object aux"+indice+";\n"+
                      "aux"+indice+"="+cod1"+".get(0);\n"

              sino
                 cod = cod1+"\n"+
                       "Object aux"+indice+";\n"+
                       "aux"+indice+"="+var1"+".get(0);\n"
              fsi
              var = "aux"+indice
              indice = indice + 1 }
```

(5) código de las expresiones RESTO:

```
        expresion dev cod, var:
           RESTO PARENTESISABIERTO (cod1,var1)=expresion PARENTESISCERRADO
         { si (var1 es igual a _) entonces
              cod = "List<Object> aux"+indice+"= new LinkedList<>();\n"+
                    "aux"+indice+".addAll("+cod1+");\n"+
```

```
                    "aux"+indice+".remove(0);\n"
          sino
             cod = cod1+"\n"+
                   "List<Object> aux"+indice+"= new LinkedList<>();\n"+
                   "aux"+indice+".addAll("+var1+");\n"+
                   "aux"+indice+".remove(0);\n"
          fsi
          var = "aux"+indice
          indice = indice + 1  }
```

(6) código de las expresiones lista construidas con cabeza y resto:

```
     expresion dev cod, var:
         CORCHETEABIERTO (cod1,var1)=expresion BARRA (cod2,var2)=expresion
CORCHETECERRADO
         { si (var1 y var2 son distintas a _) entonces
              cod = cod1+"\n"
                    cod2+"\n"
                    "List<Object> aux"+indice+"= new LinkedList<>();\n"+
                     aux+indice+".add("+var1+");\n"
                     aux+indice+".addAll("+var2+");\n"
           sino
              si (var1 es distinta a _) entonces
                  cod = cod1+"\n"
                  "List<Object> aux"+indice+"= new LinkedList<>();\n"+
                   aux+indice+".add("+var1+");\n"
                   aux+indice+".addAll("+cod2+");\n"
              sino
                  si (var2 es distinta a _) entonces
                       cod = cod2+"\n"
                       "List<Object> aux"+indice+"= new LinkedList<>();\n"+
                        aux+indice+".add("+cod1+");\n"
                        aux+indice+".addAll("+var2+");\n"
                  sino
                       cod = "List<Object> aux"+indice+"= new LinkedList<>();\n"+
                             aux+indice+".add("+cod1+");\n"
                             aux+indice+".addAll("+cod2+");\n"
                  fsi
              fsi
           fsi
           var = aux+indice
           indice = indice + 1  }
```

(7) código de las expresiones lista construidas por enumeración:

```
    expresion dev cod, var:
         CORCHETEABIERTO ((sec_cod,sec_var)=expresiones)? CORCHETECERRADO
        { para cada indice i en sec_var hacer
            si (elemento i de sec_var es distinto a _) entonces
               cod = cod + elemento i de sec_cod+"\n"
            fsi
          fpara
          cod = cod + "List<Object>aux"+indice+" = new LinkedList<>();\n"
          para cada indice i en sec_var hacer
             si (elemento i de sec_var es distinto a _) entonces
                cod = cod + "aux"+indice+".add("+elemento i de sec_var+");\n"
             sino
                cod = cod + "aux"+indice+".add("+elemento i de sec_cod+");\n"
             fsi
          fpara
          var = "aux"+indice
          indice = indice + 1
        }
```

(8) código de las expresiones negadas:

```
    expresion dev cod, var:
         NO (cod1,var1)=expresion
         { si (var1 es igual a _) entonces
              cod = "!"+cod1
           sino
              cod = cod1+"\n"+"!"+var1
           fsi
           var = _ }
```

(9) código de las expresiones constantes lógicas:

```
      expresion dev cod, var:  cte_log  { si (cte_log es cierto) cod = true
                                          sino cod = false
                                          fsi
                                          var = _ }
```

(10) código de las expresiones variables:

```
        expresion dev cod, var: IDENT   { cod = IDENT  var = _ }
```

(11) código de las expresiones variables:

```
            expresion dev cod, var:  NUMERO  { cod = NUMERO  var = _ }
```

(12) código de la secuencia de expresiones

```
    expresiones dev sec_cod,sec_var:
            (cod1,var1)=expresion COMA sec_cod,sec_var=expresiones
                {añadir cod1 al principio de sec_cod y var1 al principio de sec_var}
          | (cod1,var1)=expresion
               {añadir cod1 a sec_cod y var1 a sec_var}
```