

# MANUAL DE USO DEL PROCESADOR DE BOREAL CON ANÁLISIS SINTÁCTICO DESCENDENTE RECURSIVO

## Introducción

El presente manual pretende aportar los fundamentos para ayudar en la comprensión, modificación y personalización del Analizador del Procesador de lenguaje Boreal de cara a desarrollar la práctica de Traductores de Lenguajes.

En el Procesador de Boreal se ha utilizado la librería del gestor de Tablas de Símbolos: TS\_Gestor. Por tanto, deberá consultarse su correspondiente manual en la web para saber cómo utilizar las Tablas de Símbolos.

Tras una introducción general del Procesador, se comentan algunos detalles sobre las acciones semánticas del Analizador Semántico que pueden ser de utilidad para incorporar el Generador de Código Intermedio.

## Procesador de Boreal

El Procesador de Boreal incluye un **Analizador Léxico** completo para todo el lenguaje. Cada grupo de prácticas tendrá solamente algunas de las opciones del lenguaje, pero se recomienda no modificar el Analizador Léxico (aunque reconozca más elementos de los necesarios para el grupo de prácticas). Este Analizador Léxico funciona como un servicio que proporciona al Analizador Sintáctico el siguiente *token* disponible en el fichero de entrada, además de introducir en la Tabla de Símbolos correspondiente todos los identificadores que aparezcan en el programa fuente.

El Procesador también incluye un **Analizador Sintáctico Descendente Recursivo LL(1)** (si se desea utilizar otro tipo de analizador Sintáctico, en la web de Traductores de Lenguajes hay más disponibles). El Analizador Sintáctico es para el lenguaje Boreal completo. De nuevo, se recomienda no modificar el Analizador Sintáctico (aunque reconozca más declaraciones, sentencias y expresiones de las necesarias para el grupo de prácticas). Se ha realizado una función para cada símbolo no terminal de la gramática. El anexo 1 contiene la gramática utilizada para construir el Analizador Sintáctico.

Las acciones correspondientes al **Analizador Semántico** se encuentran intercaladas en el Analizador Sintáctico. Igual que en los casos anteriores, se recomienda no eliminar nada, aunque no se corresponda con las opciones de la práctica asignada al grupo. La gramática del Analizador Sintáctico permite el uso de acciones semánticas intercaladas dentro de la Traducción Dirigida por la Sintaxis, tanto para el Analizador Semántico como para el Generador de Código Intermedio. El manejo de los atributos del Analizador Semántico será principalmente mediante atributos de tipo sintetizado. El Anexo 2 muestra el Esquema de Traducción utilizado para desarrollar el Analizador Semántico.

Juntamente con los tres módulos de Análisis, se dispone de una **Tabla de Símbolos** completa, que permite almacenar toda la información necesaria sobre todos los identificadores de Boreal (además de las palabras reservadas). Los analizadores completarán estas Tablas de Símbolos (Global y Locales) con la información obtenida por el Analizador Léxico y por el Analizador Semántico. Por tanto, aunque la información es completa hasta ese punto, podría ser necesario incluir nueva información necesaria para la fase de síntesis del Compilador. Los detalles sobre este módulo se muestran en la página web de Traductores de Lenguajes.

Finalmente, se dispone de un sencillo **Gestor de Errores** que, en función del error enviado por el resto de los módulos, informarán al usuario de los posibles errores detectados durante el análisis del programa fuente.

Para la construcción del Compilador, deberá añadirse el **Generador de Código Intermedio** y el **Generador de Código Objeto**. El primero deberá integrarse junto al Analizador Sintáctico/Semántico existente, para que

funcione con el Analizador Léxico. Por ello, en los siguientes apartados de este manual, se darán algunas indicaciones sobre cómo incorporar el Esquema de Traducción del Generador de Código Intermedio. El Generador de Código se añadirá como un nuevo módulo que reciba los cuartetos del Generador de Código Intermedio para traducirlos directamente a ensamblador.

## Analizador Sintáctico y Semántico

### Estructura de la Clase ASin

#### Variables de la Clase

En la clase ASin hay variables que se usan en el Analizador Sintáctico, el Analizador Semántico y en la gestión de errores

```
public class ASin {
    // Declaraciones del analizador sintáctico
    private static Token<> tokenActual;           // Representa el último token leído
    ...
    // Declaraciones del analizador semántico
    public static boolean tsGlobal;               // Indica si existe TS global
    private static boolean zonaDeclaracion;      // Indica si se está en zona de declaraciones
    private static Integer despGlobal, despLocal; // Desplazamientos de Las variables
    private static Integer numEtiquetas;         // Número de etiquetas generadas
    ...
}
```

- **tokenActual**: Representa el último *token* devuelto por el analizador léxico.
- **tsGlobal**: Define el estado de la tabla de símbolos, almacenando *true* si la tabla global es la activa, o *false* en caso contrario.
- **zonaDeclaracion**: Define el estado de la zona de declaración, almacenando *true* si se está en una declaración, o *false* en caso contrario.
- **despGlobal**, **despLocal**: Acumula los desplazamientos en la tabla de símbolos global y local, respectivamente.
- **numEtiquetas**: Contador de las etiquetas asignadas.

#### Inicialización

El análisis comienza con la llamada a la función `analizar()` en el Analizador Sintáctico. Esta función se encarga de la inicialización de las variables globales y la declaración de los atributos de la Tabla de Símbolos mediante una llamada a `iniciarTS()`, además de comenzar el análisis solicitando el primer *token* y llamando a la función correspondiente al axioma de la gramática (P).

```
public static void analizar () {
    // Se inicializan las variables de la clase:
    hayError= false;
    numEtiquetas= 0;
    // Se configura la Tabla de Símbolos:
    iniciarTS ();
    // Se inicia el análisis:
    ...
    tokenActual= Alex.generarToken ();
    P();
}

private static void iniciarTS () {
    if (debug) Procesador.gestorTS.activarDebug ();
    Procesador.gestorTS.createAtributo("desplazamiento",
        DescripcionAtributo.DIR, TipoDatoAtributo.ENTERO);
    Procesador.gestorTS.createAtributo("etiqueta",
        DescripcionAtributo.ETIQUETA, TipoDatoAtributo.CADENA);
}
```

```

Procesador.gestorTS.createAtributo("pasoParametros",
    DescripcionAtributo.MODO_PARAM, TipoDatoAtributo.LISTA);
Procesador.gestorTS.createAtributo("tipoParametros",
    DescripcionAtributo.TIPO_PARAM, TipoDatoAtributo.LISTA);
Procesador.gestorTS.createAtributo("tipoRetorno",
    DescripcionAtributo.TIPO_RET, TipoDatoAtributo.CADENA);
Procesador.gestorTS.createAtributo("numParametro",
    DescripcionAtributo.NUM_PARAM, TipoDatoAtributo.ENTERO);
Procesador.gestorTS.createAtributo("modoParametro",
    DescripcionAtributo.PARAM, TipoDatoAtributo.ENTERO);
}

```

## Diseño de una Producción

### Función por Símbolo No Terminal

Para cada símbolo no terminal hay implementada una función que realiza de manera conjunta las acciones sintácticas y semánticas asociadas. Dentro de estas funciones se determina qué producción aplicar y se devuelve una instancia de Atributos con la información semántica sintetizada durante la producción. Con la llamada recursiva a estas funciones se realiza el análisis completo del programa.

Al comienzo de cada función se declaran las variables de tipo Atributo necesarias. Se define una variable para almacenar los atributos del propio símbolo no terminal y tantas variables como sean necesarias para recoger los atributos devueltos por las llamadas a las funciones de otros símbolos no terminales que puedan aparecer en las producciones desde el símbolo actual.

Dentro de cada función se compara el `tokenActual` con los conjuntos del FIRST de las producciones, o del FOLLOW en las producciones que derivan en lambda, para determinar qué producción debe aplicarse. Esto se implementa con sentencias condicionales (*if*, *else if*) y la función `tokenActualCoincide`. Esta parte del código se recomienda no modificarla, ya que corresponde al análisis sintáctico.

Cada bloque condicional corresponde a una producción y contiene los siguientes elementos:

- Una sentencia para imprimir el número de la regla que se está aplicando en el fichero *parse.txt*.
- Llamadas recursivas a las funciones correspondientes a los símbolos no terminales que aparecen en el consecuente de la producción. Cada llamada devuelve una instancia de Atributos, que se almacena en una variable local para su uso en las acciones semánticas.
- Consumo de los símbolos terminales que aparecen en el consecuente de la producción. Se hace mediante la función `match`, que verifica que `tokenActual` coincida con el símbolo esperado y solicita el siguiente *token* al Analizador Léxico.
- Sentencias correspondientes al análisis semántico. Estas sentencias hacen uso de los Atributos devueltos por los símbolos no terminales del consecuente.

Todo este código está comentado para facilitar su comprensión. Se muestra un ejemplo a continuación:

```

// Función Correspondiente al símbolo no terminal: D
// Reglas disponibles:
// D → var id : T ;
// D → λ
private static Atributos D() {
    Atributos atrD = new Atributos();
    Atributos atrID;
    Atributos atrT;

    if (tokenActualCoincideCualquiera("VAR")) {
        printParse("9"); // 9. D → var id : T ; DD
        //ASin: var
        match("VAR");
        //ASin: id
        atrID = match("ID");
        int idPos = atrID.getPos();
    }
}

```

```

//ASin: :
match("DOSPUNTOS");
//ASin: T
atrT = T();
//ASin: ;
match("PYC");
//ASem: InsertarTipoTS (id.pos, T.tipo)
Procesador.gestorTS.setTipo(idPos, atrT.getTipo());
//ASem:
if (tsGlobal) {
    Procesador.gestorTS.setValorAtributoEnt(idPos, "desplazamiento", despGlobal);
    despGlobal += atrT.getAncho();
} else {
    Procesador.gestorTS.setValorAtributoEnt(idPos, "desplazamiento", despLocal);
    despLocal += atrT.getAncho();
}
//ASin: DD
DD();
} else if (tokenActualCoincideCualquiera("begin", "function", "procedure", "program",
    "eof")) {
    printParse("10"); //10. D → λ
    //Sin acciones semánticas
}
debug(atrD);
return atrD;
}

```

Finalmente, la función devuelve la instancia de Atributos correspondiente al símbolo no terminal actual. Esta instancia contiene los atributos sintetizados durante la producción y serán utilizados por otras funciones no terminales.

## Atributos

Cada función retorna una clase contenedora de atributos, denominada Atributos. Estos atributos contienen la información semántica obtenida durante las acciones semánticas correspondientes al símbolo No Terminal.

La clase Atributos contiene las variables y métodos para manejar los atributos de cada símbolo gramatical, como posición (pos), tipo (tipo), cantidad de sentencias exit (exit), tamaño del tipo (ancho), lexema de una constante cadena (lex), etc.

El acceso y modificación de cada atributo se realiza mediante las correspondientes funciones get y set de la clase Atributos.

```

public class Atributos {
    private Integer pos; // Atributo posición en la Tabla de Símbolos
    private String tipo; // Atributo tipo del símbolo
    private Integer exit; // Atributo para contar el número de instrucciones exit
    private String ret; // Atributo para el tipo de retorno
    private Integer ancho; // Atributo para el tamaño de un tipo de datos
    private Integer longs; // Atributo para calcular la longitud de una lista
    private String referencia; // Atributo que indica si un parámetro es por referencia
    private String etiqueta; // Atributo para guardar una etiqueta
    private Integer val; // Atributo para guardar el valor numérico del símbolo
    private String lex; // Atributo para guardar el valor cadena del símbolo
    private Integer program_count; // Atributo para contar cuántos programas principales hay
    private Boolean asig; // Atributo para determinar si es asignación o llamada a función

    // Constructor que inicializa todas las variables de la clase
    public Atributos() {
        this.pos = null;
        this.tipo = null;
        this.exit = null;
        this.ret = null;
        this.ancho = null;
        this.longs = null;
        this.referencia = null;
    }
}

```

```

        this.etiqueta = null;
        this.val = null;
        this.lex = null;
        this.program_count = null;
        this.asig = null;
    }
    // Funciones get y set para las variables de la clase:
    public void setPos(Integer pos) { this.pos = pos; }
    public Integer getPos() {
        if (pos == null) return 0;
        return pos;
    }

    public void setTipo(String tipo) { this.tipo = tipo; }
    public String getTipo() {
        if (tipo == null) return "";
        return tipo;
    }

    public void setExit(Integer exit) { this.exit = exit; }
    public Integer getExit() {
        if (exit == null) return -1;
        return exit;
    }

    public void setRet(String ret) { this.ret = ret; }
    public String getRet() {
        if (ret == null) return "";
        return ret;
    }

    public void setAncho(Integer ancho) { this.ancho = ancho; }
    public Integer getAncho() {
        if (ancho == null) return -1;
        return ancho;
    }

    public void setLong(Integer longs) { this.longs = longs; }
    public Integer getLong() {
        if (longs == null) return -1;
        return longs;
    }

    public void setReferencia(String ref) { this.referencia = ref; }
    public String getReferencia() {
        if (referencia == null) return "";
        return referencia;
    }

    public void setEtiqueta(String et) { this.etiqueta = et; }
    public String getEtiqueta() {
        if (etiqueta == null) return "";
        return etiqueta;
    }

    public void setVal(Integer val) { this.val = val; }
    public Integer getVal() {
        if (val == null) return 32768; //Boreal solo admite valores hasta 32767
        return val;
    }

    public void setLex(String lex) { this.lex = lex; }
    public String getLex() {
        if (lex == null) return "";
        return lex;
    }

    public void setProgramCount(Integer i) { this.program_count = i; }
    public int getProgramCount() {
        if (program_count == null) return 0;
    }

```

```

        return this.program_count;
    }

    public void    setAsig(Boolean asig) { this.asig = asig; }
    public Boolean getAsig() {
        if (asig == null) return false;
        return asig;
    }
}

```

Si se desea añadir un nuevo atributo, se debe crear una nueva variable de clase del tipo deseado, inicializarla en el constructor y definir las respectivas funciones set y get de dicho nuevo atributo.

Por ejemplo, si fuera necesario crear un nuevo atributo (ficticio) para contabilizar la cantidad de bloques *begin-end*, se podría hacer de la siguiente manera:

```

class Atributos {
    ...
    private Integer begincont;                                // Nuevo atributo

    public Atributos() {
        ...
        this.begincont = null;    // Inicialización del nuevo atributo en el constructor
    }
    ...
    // get y set para el nuevo atributo
    public void    setBeginCont(Integer begincont) { this.begincont = begincont; }
    public Integer getBeginCont() { return this.begincont; }
}

```

## Acción Semántica

Las acciones semánticas se ejecutan intercaladas entre el análisis sintáctico. Al usarse atributos de tipo sintetizado, solo se puede realizar acciones semánticas con los datos de los símbolos no terminales ya ejecutados.

## Modificación de una Acción Semántica

Modificar una acción semántica de una de las reglas para añadir las acciones semánticas del Generador de Código Intermedio implica cambiar directamente la función del símbolo no terminal correspondiente al antecedente de la acción semántica que se quiere ampliar.

Por ejemplo, si se quisiera realizar una modificación (ficticia) sobre la regla 24 (Bloque → begin C end), se parte de la acción semántica actual:

```

Bloque → begin C end {    Bloque.tipo:= C.tipo
                        Bloque.exit:= C.exit
                        Bloque.tipoRet:= C.tipoRet }

```

El código correspondiente al símbolo no terminal de Bloque sería:

```

private static Atributos Bloque() {                                // Símbolo no terminal Bloque
    //Se crean los atributos correspondientes al símbolo no terminal
    Atributos atrBloque = new Atributos();
    // Se crean los atributos de los símbolos no terminales del consecuente
    Atributos atrC;
    // Se determina qué producción aplicar
    if (tokenActualCoincideCualquiera("BEGIN")) {
        // ASin: begin
        match("BEGIN");
        // ASin: C
        // Se obtienen los atributos de C
        atrC = C();
        // ASin: end
        match("END");
        // ASem: Bloque.tipo:= C.tipo
    }
}

```

```

    atrBloque.setTipo(atrC.getTipo());
    // ASem: Bloque.exit:= C.exit
    atrBloque.setExit(atrC.getExit());
    // ASem: Bloque.tipoRet:= C.tipoRet
    atrBloque.setRet(atrC.getRet());
}
debug(atrBloque);
// Se devuelven los atributos del antecedente
return atrBloque;
}

```

Ahora, si se supone que la nueva acción semántica que se quiere realizar para dicha regla es:

```

Bloque → begin C end {   Bloque.tipo:= C.tipo
                        Bloque.exit:= C.exit
                        Bloque.tipoRet:= C.tipoRet
                        Bloque.BeginCont:= C.BeginCont + 1 }

```

El nuevo código de la acción semántica de la regla 26 sería:

```

private static Atributos Bloque() { // Símbolo no terminal Bloque
    // Se crean los atributos correspondientes al símbolo no terminal
    Atributos atrBloque = new Atributos();
    // Se crean los atributos de los símbolos no terminales del consecuente
    Atributos atrC;
    // Se determina qué producción aplicar
    if (tokenActualCoincideCualquiera("BEGIN")) {
        // ASin: begin
        match("BEGIN");
        // ASin: C
        // Se obtienen los atributos de C
        atrC = C();
        // ASin: end
        match("END");
        // ASem: Bloque.tipo:= C.tipo
        atrBloque.setTipo(atrC.getTipo());
        // ASem: Bloque.exit:= C.exit
        atrBloque.setExit(atrC.getExit());
        // ASem: Bloque.tipoRet:= C.tipoRet
        atrBloque.setRet(atrC.getRet());
        // ASem: Bloque.BeginCont:= C.BeginCont + 1
        atrBloque.setBeginCont(atrC.getBeginCont() + 1);
    }
    debug(atrBloque);
    // Se devuelven los atributos del antecedente
    return atrBloque;
}

```

### Creación de una Nueva Acción Semántica

Una de las características de cómo se ha implementado el Analizador Descendente Recursivo, es que se puede introducir acciones semánticas en cualquier punto del consecuente de una regla. No obstante, algo para tener en cuenta es que estas acciones semánticas solo pueden acceder a la información de los Símbolos No Terminales que están antes de la acción.

Por lo tanto, para añadir una acción semántica nueva, solo debe escribirse el código correspondiente dentro de la función asociada al símbolo no terminal, dentro de la producción que se desee.

Por ejemplo, si en la regla 10 (que inicialmente no tiene ninguna acción semántica) se quiere incluir una acción semántica para determinar que el atributo val de D debe ser vacío, el diseño de la regla quedaría:

```

D → lambda { D.val:= "vacío" }

```

El nuevo código de la función D sería:

```
private static Atributos D() {
    Atributos atrD = new Atributos();
    Atributos atrID;
    Atributos atrT;
    if (tokenActualCoincideCualquiera("VAR")) {
        ... // 9. D → var id : T ; DD
        // No se modifica
    }
    else if (tokenActualCoincideCualquiera("begin", "function", "procedure", "program",
        "eof")){
        printParse("10"); // 10. D → λ
        atrD.setValor("vacío");
    }
    return atrD;
}
```

## Tipos de Datos

El Analizador Semántico maneja una serie de tipos asociados a los distintos símbolos gramaticales. Para ello, se utilizan cadenas con los nombres de los tipos escritos en minúsculas de la siguiente manera: “cadena”, “entero”, “lógico”, “función”, “procedimiento”, “tipo\_ok”, “tipo\_error” y “vacío”.

## Gestor de Errores

El Gestor de Errores permite informar al usuario los errores detectados durante la compilación. Para el análisis sintáctico y semántico se han usado los errores flexibles. Se emplea la función `writeError` de la clase `GestorError` para imprimir un mensaje de error puntual. Tienen dos argumentos, el primero recoge el tipo de error y el segundo una pequeña descripción.

```
GestorError.writeError("Sintáctico", "Exit fuera de bucle detectado en Procedure");
```

Este código imprimirá:

Error Sintáctico en la línea [1]: Exit fuera de bucle detectado en Procedure

También se pueden usar los errores predefinidos. Se crea una instancia en la clase `Accion`, se añade un valor en el enum `Acciones` y una entrada en un nuevo case con el mensaje de error en la función `printError()` de la clase `GestorError`. La llamada se realiza usando la función `setError`, a la que se le puede enviar información adicional como segundo parámetro. Esta forma es la recomendada si el mensaje de error se puede producir en distintas situaciones.

```
GestorError.setError(Acciones.eSem5_case_otherwise_error, "");
```

## Anexo 1: Gramática del Analizador Sintáctico

```
//// Conjunto de símbolos terminales
Terminales = { program ; id procedure function var boolean integer string ( : ) if then begin end else
while do repeat until loop for := to case of entero otherwise write writeln read return
exit when , or xor and = <> > >= < <= + - * / mod ** not cadena true false in max min }
```

```
//// Conjunto de símbolos no terminales
NoTerminales = { P R PP PR PF D DD T A AA C B Bloque N O S LL L Q V W Y E F G H I J K Z X
Eprima Fprima Gprima Hprima Iprima Jprima Zprima Bcola Scola BlqElse }
```

```
//// Axioma
Axioma = P
```

```
//// Lista de producciones
Producciones = {
P -> D R
R -> PP R
R -> PR R
```



```

R -> PF R
R -> lambda
PP -> program id ; D Bloque ;
PR -> procedure id A ; D Bloque ;
PF -> function id A : T ; D Bloque ;
D -> var id : T ; DD
D -> lambda
DD -> id : T ; DD
DD -> lambda
T -> boolean
T -> integer
T -> string
A -> ( X id : T AA )
A -> lambda
AA -> ; X id : T AA
AA -> lambda
X -> var
X -> lambda
C -> B C
C -> lambda
Bloque -> begin C end
B -> S
B -> if E then Bcola
Bcola -> S
Bcola -> Bloque ; BlqElse
BlqElse -> else Bloque ;
BlqElse -> lambda
B -> while E do Bloque ;
B -> repeat C until E ;
B -> loop C end ;
B -> for id := E to E do Bloque ;
B -> case E of N O end ;
N -> entero : Bloque ; N
N -> lambda
O -> otherwise : Bloque ;
O -> lambda
S -> write LL ;
S -> writeln LL ;
S -> read ( V ) ;
S -> id Scola
Scola -> := E ;
Scola -> LL ;
S -> return Y ;
S -> exit when E ;
LL -> ( L )
LL -> lambda
L -> E Q
Q -> , E Q
Q -> lambda
V -> id W
W -> , id W
W -> lambda
Y -> E
Y -> lambda
E -> F Eprima
Eprima -> or F Eprima
Eprima -> xor F Eprima
Eprima -> lambda
F -> G Fprima
Fprima -> and G Fprima
Fprima -> lambda
G -> H Gprima
Gprima -> = H Gprima
Gprima -> <> H Gprima
Gprima -> > H Gprima
Gprima -> >= H Gprima
Gprima -> < H Gprima
Gprima -> <= H Gprima
Gprima -> lambda
H -> I Hprima

```

```

Hprima -> + I Hprima
Hprima -> - I Hprima
Hprima -> lambda
I -> J Iprima
Iprima -> * J Iprima
Iprima -> / J Iprima
Iprima -> mod J Iprima
Iprima -> lambda
J -> K Jprima
Jprima -> ** K Jprima
Jprima -> lambda
K -> not K
K -> + K
K -> - K
K -> Z
Z -> entero Zprima
Z -> cadena
Z -> true
Z -> false
Z -> id LL Zprima
Z -> ( E ) Zprima
Z -> max ( L )
Z -> min ( L )
Zprima -> in ( L )
Zprima -> lambda
}

```

## Anexo 2: Esquema de Traducción del Analizador Semántico

```

1.  P →      {   TSG:= CreadTS ()
                TSActual:= TSG
                desp_global:=0
                zona_decl:= true
            }

    DR
    {   If (R.program ≠ 1) Then Error ("Debe haber 1 y solo 1 Program")
        DestruirTS (TSG)
    }

2.  R → PP R1 {R.program:= 1 + R1.program}
3.  R → PR R1 {R.program:= R1.program}
4.  R → PF R1 {R.program:= R1.program}
5.  R → λ      {R.program:= 0}

6.  PP → program id ; {   InsetarTipoTS (id.pos, vacío→vacío)
                        InsertarEtiquTS (id.pos, "main")
                        TSL:= CreadTS ()
                        TSActual:= TSL
                        despl_local:= 0
                    }

    D      {   zona_decl:= false }
    Bloque ; {   if (Bloque.tipo = tipo_error)
                  then Error ("Error detectado en el cuerpo del Programa principal")
                  if(Bloque.tipoRet ≠ tipo_ok AND Bloque.tipoRet ≠ vacío)
                  then Error ("Programa Principal con instrucción de retorno no vacío")
                  if(Bloque.exit > 0)
                  then Error ("Exit fuera de bucle detectado en Programa Principal")
                  destruirTS (TSL)
                  TSActual:= TSG
                  zona_decl:= true
                }

```

```

7.  PR → procedure id {   TSL:= CrearTS ()
                          TSActual:= TSL
                          despl_local:= 0
                          }
    A;                    {   InsertarTipoTS (id.pos, A.tipo→vacío)
                          InsertarModoTS (id.pos, A.referencia)
                          // A.referencia es un producto cartesiano de Lógicos
                          InsertarEtiquetaTS (id.pos, nuevaEt ())
                          }
    D                      {   zona_decl:= false }
    Bloque;               {   if (Bloque.tipo = tipo_error)
                          then Error ("Error detectado en el cuerpo del Procedure")
                          if (Bloque.tipoRet ≠ tipo_ok AND Bloque.tipoRet ≠ vacío)
                          then Error ("Procedure con instrucción de retorno no vacío")
                          if (Bloque.exit > 0)
                          then Error ("Exit no puede situarse fuera del bucle en el Procedure")
                          DestruirTS (TSL)
                          TSActual:= TSG
                          zona_decl:= true
                          }

8.  PF → function id    {   TSL:= CrearTS ()
                          TSActual:= TSL
                          Despl_local:= 0
                          }
    A:T;                 {   InsertarTipoTS (id.pos, A.tipo→T.tipo)
                          InsertarModoTS (id.pos, A.referencia)
                          // A.referencia es un producto cartesiano de Lógicos
                          InsertarEtiquetaTS (id.pos, nuevaEt ())
                          }
    D                      {   zona_decl:= false }
    Bloque;               {   if (Bloque.tipo = tipo_error)
                          then Error ("Error detectado en el cuerpo de la función")
                          if (Bloque.tipoRet ≠ tipo_ok AND Bloque.tipoRet ≠ T.tipo)
                          then Error ("Función con retorno inválido")
                          if (Bloque.exit > 0)
                          then Error ("Exit no puede situarse fuera del bucle en la función")
                          DestruirTS (TSL)
                          TSActual:= TSG
                          zona_decl:= true
                          }
    }

9.  D → var id:T; {   InsertarTipoTS (id.pos, T.tipo)
                      if (TSActual = TSG) then
                      {
                          InsertarDespTS (id.pos, despl_global)
                          despl_global:= despl_global + T.ancho
                      }
                      else
                      {
                          InsertarDespTS (id.pos, despl_local)
                          despl_local:= despl_local + T.ancho
                      }
                      }

    DD
10. D → λ              { }

```

```

11. DD  $\rightarrow$  id : T; {  InsertarTipoTS (id.pos, T.tipo)
                        if (TSActual = TSG) then
                        {
                            InsertarDespTS (id.pos, despl_global)
                            despl_global:= despl_global + T.ancho
                        }
                        else
                        {
                            InsertarDespTS (id.pos, despl_local)
                            despl_local:= despl_local + T.ancho
                        }
                        }

                        DD
12. DD  $\rightarrow$   $\lambda$       { }

13. T  $\rightarrow$  boolean {T.tipo:= lógico; T.ancho:= 1}
14. T  $\rightarrow$  integer {T.tipo:= entero; T.ancho:= 1}
15. T  $\rightarrow$  string  {T.tipo:= cadena; T.ancho:= 64}

16. A  $\rightarrow$  (X id : T {  InsertarTipoTS (id.pos, T.tipo)
                        InsertarDespTS (id.pos, despl_local)
                        if (X.referencia) then
                        {
                            InsertaAtributoPasoParametrosTS (id.pos, "referencia")
                            despl_local:= despl_local + 1
                        }
                        else
                        {
                            InsertaAtributoPasoParametrosTS (id.pos, "valor")
                            despl_local:= despl_local + T.ancho
                        }
                        }
                        AA ) {
                        if (AA.tipo  $\neq$  vacío) then
                        {
                            A.tipo:= T.tipo x AA.tipo
                            A.referencia:= X.referencia x AA.referencia // producto cartesiano de lógicos
                        }
                        else
                        {
                            A.tipo:= T.tipo
                            A.referencia:= X.referencia
                        }
                        A.long:= 1 + AA.long
                        }

17. A  $\rightarrow$   $\lambda$  {  A.tipo:= vacío
                  A.long:= 0
                  A.referencia:=  $\emptyset$ 
                  }

```

```

18. AA → ; X id : T {      InsertarTipoTS (id.pos, T.tipo)
                           InsertarDespTS (id.pos, despl_local)
                           if (X.referencia) then
                           {
                               InsertaAtributoPasoParametrosTS (id.pos, "referencia")
                               despl_local:= despl_local + 1
                           }
                           else
                           {
                               InsertaAtributoPasoParametrosTS (id.pos, "valor")
                               despl_local:= despl_local + T.ancho
                           }
                           }

    AA1      {
                if (AA1.tipo ≠ vacío) then
                {
                    AA.tipo:= T.tipo x AA1.tipo
                    AA.referencia := X.referencia x AA1.referencia
                                                    // producto cartesiano de lógicos
                }
                else
                {
                    AA.tipo:= T.tipo
                    AA.referencia:= X.referencia
                }
                AA.long:= 1 + AA1.long
            }

19. AA → λ      {      AA.tipo:= vacío
                        AA.long:= 0
                        AA.referencia:= ∅
                    }

20. X → var      {X.referencia = true}
21. X → λ        {X.referencia = false}

22. C → B C1    {      C.tipo:= if (B.tipo = tipo_ok)
                        then C1.tipo
                        else tipo_error
                        C.exit:= B.exit + C1.exit
                        C.tipoRet:= if (B.tipoRet = C1.tipoRet)
                                    then B.tipoRet
                                    else if (B.tipoRet = tipo_ok)
                                        then C1.tipoRet
                                    else if (C1.tipoRet = tipo_ok)
                                        then B.tipoRet
                                    else tipo_error
                    }

23. C → λ        {      C.tipo:= tipo_ok
                        C.exit:= 0
                        C.tipoRet:= tipo_ok
                    }

24. Bloque → begin C end {      Bloque.tipo:= C.tipo
                                   Bloque.exit:= C.exit
                                   Bloque.tipoRet:= C.tipoRet
                               }

25. B → S        {      B.tipo:= S.tipo
                        B.exit:= S.exit
                        B.tipoRet:= S.tipoRet
                    }

26. B → if E then Bcola {      B.tipo:= if (E.tipo = lógico)
                                   then Bcola.tipo
                                   else tipo_error
                                   B.exit:= Bcola.exit
                                   B.tipoRet:= Bcola.tipoRet
                               }

```

```

27. Bcola → S      {   Bcola.tipo:= S.tipo
                        Bcola.exit:= S.exit
                        Bcola.tipoRet:= S.tipoRet
                      }
28. Bcola → Bloque ; BlqElse
    {   Bcola.tipo:= if(Bloque.tipo = tipo_ok)
                        then BlqElse.tipo
                        else tipo_error
        Bcola.exit:= Bloque.exit + BlqElse.exit
        Bcola.tipoRet:= if (Bloque.tipoRet = BlqElse.tipoRet)
                        then Bloque.tipoRet
                        else if (Bloque.tipoRet = tipo_ok)
                            then BlqElse.tipoRet
                        else if (BlqElse.tipoRet = tipo_ok)
                            then Bloque.tipoRet
                        else tipo_error
    }
29. BlqElse → else Bloque ; {   BlqElse.tipo:= Bloque.tipo
                                BlqElse.exit:= Bloque.exit
                                BlqElse.tipoRet:= Bloque.tipoRet  }
30. BlqElse → λ      {   BlqElse.tipo:= tipo_ok
                        BlqElse.exit:= 0
                        BlqElse.tipoRet:= tipo_ok }
31. B → while E do Bloque ;
    {   B.tipo:= if (E.tipo = lógico)
                        then Bloque.tipo
                        else tipo_error
        B.exit:= Bloque.exit
        B.tipoRet:= Bloque.tipoRet
    }
32. B → repeat C until E ;
    {   B.tipo:= if (E.tipo = lógico)
                        then C.tipo
                        else tipo_error
        B.exit:= C.exit
        B.tipoRet:= C.tipoRet
    }
33. B → loop C end ; {   if (C.exit ≠ 1)
                        then Error ("dentro de loop debe haber 1 y solo 1 exit")
        B.tipo:= C.tipo
        B.exit:= 0
        B.tipoRet:= C.tipoRet
    }
34. B → for id := E1 to E2 do Bloque ;
    {   B.tipo:= if (E1.tipo=entero AND E2.tipo=entero AND BuscarTipoTS(id.pos) = entero)
                        then Bloque.tipo
                        else tipo_error
        B.exit:= Bloque.exit
        B.tipoRet:= Bloque.tipoRet
    }
35. B → case E of N O end;
    {   B.tipo:= if (E.tipo = entero AND N.tipo = O.tipo = tipo_ok)
                        then tipo_ok
                        else tipo_error
        B.exit:= N.exit + O.exit
        B.tipoRet:= if (N.tipoRet = O.tipoRet)
                        then N.tipoRet
                        else if (N.tipoRet = tipo_ok)
                            then O.tipoRet
                        else if (O.tipoRet = tipo_ok)
                            then N.tipoRet
                        else tipo_error
    }

```

```

36. N → entero : Bloque ; N1
    {   N.tipo:= if (N1.tipo = tipo_ok)
        then Bloque.tipo
        else tipo_error
      N.exit:= Bloque.exit + N1.exit
      N.tipoRet:= if (Bloque.tipoRet = N1.tipoRet)
                  then Bloque.tipoRet
                  else if (Bloque.tipoRet = tipo_ok)
                      then N1.tipoRet
                  else if (N1.tipoRet = tipo_ok)
                      then Bloque.tipoRet
                  else tipo_error
    }

37. N → λ
    {   N.tipo:= tipo_ok
      N.exit:= 0
      N.tipoRet:= tipo_ok   }

38. O → otherwise : Bloque ; {   O.tipo:= Bloque.tipo
                                O.exit:= Bloque.exit
                                O.tipoRet:= Bloque.tipoRet   }

39. O → λ    {   O.tipo:= tipo_ok
                O.exit:= 0
                O.tipoRet:= tipo_ok}

40. S → write LL ; {   S.tipo:= tipo_ok
                      if (LL.tipo ≠ vacío) then
                      {
                        for i:= 1 to LL.long
                          if (LL.tipo[i] ≠ entero AND LL.tipo[i] ≠ cadena)
                            then S.tipo:= tipo_error
                      }
                      S.exit:= 0
                      S.tipoRet:= tipo_ok
                    }

41. S → writeln LL ; {   S.tipo:= tipo_ok
                      if (LL.tipo ≠ vacío) then
                      {
                        for i:= 1 to LL.long
                          if (LL.tipo[i] ≠ entero AND LL.tipo[i] ≠ cadena)
                            then S.tipo:= tipo_error
                      }
                      S.exit:= 0
                      S.tipoRet:= tipo_ok
                    }

42. S → read (V) ; {   S.tipo:= tipo_ok
                      for i:= 1 to V.long
                        if (V.tipo[i] ≠ entero AND V.tipo[i] ≠ cadena)
                          then S.tipo:= tipo_error
                      S.exit:= 0
                      S.tipoRet:= tipo_ok
                    }

43. S → id Scola    {   id.tipo:= BuscaTipoTS (id.pos)
                      id.et:= BuscaEtiqTS (id.pos)
                      if (id.tipo = Scola.tipo AND Scola.asig) then
                      {
                        S.tipo:= tipo_ok
                      }
                      else if (id.tipo = Scola.tipo→vacío AND NOT Scola.asig AND id.et ≠ "main") then
                      {
                        S.tipo:= tipo_ok
                      }
                      else S.tipo:= tipo_error // Asignación con tipos distintos, llamada a identificador que
                                                // no es procedimiento o identificador es program
                      S.exit:= 0
                      S.tipoRet:= tipo_ok
                    }

```

```

44. Scola → E; {   Scola.tipo := E.tipo
                   Scola.long := 0
                   Scola.asig := true   }
45. Scola → LL; {   Scola.tipo := LL.tipo
                   Scola.long := LL.long
                   Scola.asig := false  }

46. S → return Y; {   S.tipo:= if (Y.tipo ≠ tipo_error)
                       then tipo_ok
                       else tipo_error
                       S.exit:= 0
                       S.tipoRet:= Y.tipo
                       // tipoRet puede ser vacío (para procedimientos), un tipo (para funciones) o tipo_ok (cuando no hay return)
                       }
47. S → exit when E; {   S.tipo:= if (E.tipo = lógico)
                           then tipo_ok
                           else tipo_error
                           S.exit:= 1
                           S.tipoRet:= tipo_ok
                           }

48. LL → (L) {   LL.tipo:= L.tipo
                 LL.long:= L.long }
    }
49. LL → λ {   LL.tipo:= vacío
                 LL.long:= 0 }

50. L → E Q {   L.tipo:= if (Q.tipo = vacío)
                 {
                     L.tipo:= E.tipo
                 }
                 else
                 {
                     L.tipo:= E.tipo x Q.tipo
                 }
                 L.long:= 1 + Q.long
                 }
51. Q → ,E Q1 {   if (Q1.tipo = vacío) then
                 {
                     Q.tipo:= E.tipo
                 }
                 else
                 {
                     Q.tipo:= E.tipo x Q1.tipo
                 }
                 Q.long:= 1 + Q1.long
                 }
52. Q → λ {   Q.tipo:= vacío
                 Q.long:= 0 }

53. V → id W {   if (W.tipo = vacío)
                 {
                     V.tipo:= buscaTipoTS (id.pos)
                 }
                 else
                 {
                     V.tipo:= buscaTipoTS (id.pos) x W.tipo
                 }
                 V.long:= 1 + W.long
                 }

```



```

54. W → id W1 { if (W1.tipo = vacío) then
                    {
                        W.tipo:= buscaTipoTS (id.pos)
                    }
                    else
                    {
                        W.tipo:= buscaTipoTS (id.pos) x W1.tipo
                    }
                    W.long:= 1 + W1.long
                }

55. W → λ { W.tipo:= vacío
              W.long:= 0 }

56. Y → E { Y.tipo:= E.tipo }
57. Y → λ { Y.tipo:= vacío }

58. E → F Eprima { if (Eprima.tipo = vacío)
                    then
                    {
                        E.tipo:= F.tipo
                    }
                    else if (F.tipo = Eprima.tipo = lógico)
                    then
                    {
                        E.tipo:= lógico
                    }
                    else F.tipo:= tipo_error
                }

59. Eprima → or F Eprima1 { if (Eprima1.tipo = vacío)
                              then if(F.tipo = lógico)
                              then
                              {
                                  Eprima.tipo:= lógico
                              }
                              else Eprima.tipo:= tipo_error
                              else if (F.tipo = Eprima1.tipo = lógico)
                              then
                              {
                                  E.prima.tipo:= lógico
                              }
                              else Eprima.tipo:= tipo_error

60. Eprima → xor F Eprima1 { if (Eprima1.tipo = vacío)
                              then if(F.tipo = lógico)
                              then
                              {
                                  Eprima.tipo:= lógico
                              }
                              else Eprima.tipo:= tipo_error
                              else if (F.tipo = Eprima1.tipo = lógico)
                              then
                              {
                                  E.prima.tipo:= lógico
                              }
                              else Eprima.tipo:= tipo_error

61. Eprima → λ { Eprima.tipo:= vacío }

```

```

62. F → G Fprima {   if (Fprima.tipo = vacío)
                        then
                        {
                            F.tipo:= G.tipo
                        }
                        else if (G.tipo = Fprima.tipo = lógico)
                        then
                        {
                            F.tipo:= lógico
                        }
                        else F.tipo:= tipo_error
                    }

63. Fprima → and G Fprima1 {   if (Fprima1.tipo = vacío)
                                then if (G.tipo = lógico)
                                    then
                                    {
                                        Fprima.tipo:= lógico
                                    }
                                    else Fprima.tipo:= tipo_error
                                else if (G.tipo = Fprima1.tipo = lógico)
                                then
                                {
                                    Fprima.tipo:= lógico
                                }
                                else Fprima.tipo:= tipo_error
                            }

64. Fprima → λ    {   Fprima.tipo:= vacío    }

65. G → H Gprima {   if (Gprima.tipo = vacío)
                        then
                        {
                            G.tipo:= H.tipo
                        }
                        else if (H.tipo = entero AND Gprima.tipo = lógico)
                        then G.tipo:= lógico
                        else G.tipo:= tipo_error
                    }

66. Gprima → = H Gprima1 {   if (Gprima1.tipo = vacío)
                                then if (H.tipo = entero)
                                    then
                                    {
                                        Gprima.tipo:= lógico
                                    }
                                    else Gprima.tipo:= tipo_error
                                else Gprima.tipo:= tipo_error
                            }

67. Gprima → <> H Gprima1 {   if (Gprima1.tipo = vacío)
                                then if (H.tipo = entero)
                                    then
                                    {
                                        Gprima.tipo:= lógico
                                    }
                                    else Gprima.tipo:= tipo_error
                                else Gprima.tipo:= tipo_error
                            }

68. Gprima → > H Gprima1 {   if (Gprima1.tipo = vacío)
                                then if (H.tipo = entero)
                                    then
                                    {
                                        Gprima.tipo:= lógico
                                    }
                                    else Gprima.tipo:= tipo_error
                                else Gprima.tipo:= tipo_error
                            }

```

```

69. Gprima  $\rightarrow \geq$  H Gprima1 {   if (Gprima1.tipo = vacío)
                                   then if (H.tipo = entero)
                                       then
                                           {
                                               Gprima.tipo:= lógico
                                           }
                                       else Gprima.tipo:= tipo_error
                                   else Gprima.tipo:= tipo_error
                                   }

70. Gprima  $\rightarrow <$  H Gprima1 {   if (Gprima1.tipo = vacío)
                                   then if (H.tipo = entero)
                                       then
                                           {
                                               Gprima.tipo:= lógico
                                           }
                                       else Gprima.tipo:= tipo_error
                                   else Gprima.tipo:= tipo_error
                                   }

71. Gprima  $\rightarrow \leq$  H Gprima1 {   if (Gprima1.tipo = vacío)
                                   then if (H.tipo = entero)
                                       then
                                           {
                                               Gprima.tipo:= lógico
                                           }
                                       else Gprima.tipo:= tipo_error
                                   else Gprima.tipo:= tipo_error
                                   }

72. Gprima  $\rightarrow \lambda$  {Gprima.tipo:= vacío}

73. H  $\rightarrow$  I Hprima {   if (Hprima.tipo = vacío)
                        then
                        {
                            H.tipo:= I.tipo
                        }
                        else if (I.tipo = Hprima.tipo = cadena)
                        then
                        {
                            H.tipo:= cadena
                        }
                        else if (I.tipo = entero AND Hprima.tipo = entero)
                        then
                        {
                            H.tipo:= entero
                        }
                        else H.tipo:= tipo_error
                    }

74. Hprima  $\rightarrow +$  I Hprima1 {   if (Hprima1.tipo = vacío)
                                   then if (I.tipo  $\in$  {entero, cadena})
                                       then
                                           {
                                               Hprima.tipo:= I.tipo
                                           }
                                       else Hprima.tipo:= tipo_error
                                   else if (I.tipo = Hprima1.tipo  $\in$  {entero, cadena})
                                       then
                                           {
                                               Hprima.tipo:= I.tipo
                                           }
                                       else Hprima.tipo:= tipo_error
                                   }

```

```

75. Hprima → - I Hprima1    {   if (Hprima1.tipo = vacío)
                                then if (I.tipo = entero)
                                    then
                                        {
                                            Hprima.tipo:= entero
                                        }
                                    else Hprima.tipo:= tipo_error
                                else if (I.tipo = Hprima1.tipo = entero)
                                    then
                                        {
                                            Hprima.tipo:= entero
                                        }
                                    else Hprima.tipo:= tipo_error
                                }

76. Hprima → λ              {Hprima.tipo:= vacío}

77. I → J Iprima            {   if (Iprima.tipo = vacío)
                                {
                                    I.tipo:= J.tipo
                                }
                                else if (J.tipo = Iprima.tipo = entero)
                                    then
                                        {
                                            I.tipo:= entero
                                        }
                                else J.tipo:= tipo_error
                                }

78. Iprima → * J Iprima1    {   if (Iprima1.tipo = vacío)
                                then if (J.tipo = entero)
                                    then
                                        {
                                            Iprima.tipo:= entero
                                        }
                                    else Iprima.tipo:= tipo_error
                                else if (J.tipo = Iprima1.tipo = entero)
                                    then
                                        {
                                            Iprima.tipo:= entero
                                        }
                                    else Iprima.tipo:= tipo_error
                                }

79. Iprima → / J Iprima1    {   if (Iprima1.tipo = vacío)
                                then if (J.tipo = entero)
                                    then
                                        {
                                            Iprima.tipo:= entero
                                        }
                                    else Iprima.tipo:= tipo_error
                                else if (J.tipo = Iprima1.tipo = entero)
                                    then
                                        {
                                            Iprima.tipo:= entero
                                        }
                                    else Iprima.tipo:= tipo_error
                                }

80. Iprima → mod J Iprima1 {   if (Iprima1.tipo = vacío)
                                then if (J.tipo = entero)
                                    then
                                        {
                                            Iprima.tipo:= entero
                                        }
                                    else Iprima.tipo:= tipo_error
                                else if (J.tipo = Iprima1.tipo = entero)
                                    then
                                        {
                                            Iprima.tipo:= entero
                                        }
                                    else Iprima.tipo:= tipo_error
                                }

```

```

81. Iprima  $\rightarrow \lambda\{\text{Iprima.tipo} := \text{vacío}\}$ 

82. J  $\rightarrow$  K Jprima {   if (Jprima.tipo = vacío)
                        then
                        {
                            J.tipo := K.tipo
                        }
                        else if (K.tipo = Jprima.tipo = entero)
                        then
                        {
                            J.tipo := entero
                        }
                        else J.tipo := tipo_error
                    }

83. Jprima  $\rightarrow$  ** K Jprima1 {   if (Jprima1.tipo = vacío)
                                then if (K.tipo = entero)
                                    then
                                    {
                                        Jprima.tipo := entero
                                    }
                                    else Jprima.tipo := tipo_error
                                else if (K.tipo = Jprima1.tipo = entero)
                                then
                                {
                                    Jprima.tipo := entero
                                }
                                else Jprima.tipo := tipo_error
                            }

84. Jprima  $\rightarrow \lambda\{\text{Jprima.tipo} := \text{vacío}\}$ 

85. K  $\rightarrow$  not K1 {   if (K1.tipo = lógico)
                        then
                        {
                            K.tipo := lógico
                        }
                        else K.tipo := tipo_error
                    }

86. K  $\rightarrow$  + K1 {   if (K1.tipo = entero)
                        then
                        {
                            K.tipo := entero
                        }
                        else K.tipo := tipo_error
                    }

87. K  $\rightarrow$  - K1 {   if (K1.tipo = entero)
                        then
                        {
                            K.tipo := entero
                        }
                        else K.tipo := tipo_error
                    }

88. K  $\rightarrow$  Z {K.tipo := Z.tipo}

89. Z  $\rightarrow$  entero Zprima {   if (Zprima.tipo = vacío)
                            then
                            {
                                Z.tipo := entero
                            }
                            else if (Zprima.tipo = entero)
                            then
                            {
                                Z.tipo := lógico
                            }
                            else Z.tipo := tipo_error
                        }

90. Z  $\rightarrow$  cadena {Z.tipo := cadena}
91. Z  $\rightarrow$  true {Z.tipo := lógico}

```

```

92. Z → false {Z.tipo:= lógico}
93. Z → id LL Zprima { id.tipo:= BuscaTipoTS (id.pos)
                        if (id.tipo = LL.tipo→t)
                        then if (t ≠ vacío) // id es función
                            then if (Zprima.tipo = vacío)
                                then
                                    {
                                        Z.tipo:= t
                                    }
                                else if (Zprima.tipo = t = entero)
                                    then
                                        {
                                            Z.tipo:= lógico
                                        }
                                else Z.tipo:= tipo_error
                            else Z.tipo:= tipo_error // la función tendría que devolver un valor
                        else if (LL.tipo = vacío) // id es variable
                        then if (Zprima.tipo = vacío)
                            then
                                {
                                    Z.tipo:= id.tipo
                                }
                            else if (Zprima.tipo = id.tipo = entero)
                                then
                                    {
                                        Z.tipo:= lógico
                                    }
                            else Z.tipo:= tipo_error
                        else Z.tipo:= tipo_error // variable con parámetros
                    }
94. Z → ( E ) Zprima { if (Zprima.tipo = vacío)
                        then
                            {
                                Z.tipo:= E.tipo
                            }
                        else if (Zprima.tipo = E.tipo = entero)
                            then
                                {
                                    Z.tipo:= lógico
                                }
                        else Z.tipo:= tipo_error
                    }
95. Z → max ( L ) { Z.tipo:= entero
                    for i:= 1 to L.long
                        if (L.tipo[i] ≠ entero)
                            then Z.tipo:= tipo_error
                    }
96. Z → min ( L ) { Z.tipo:= entero
                    for i:= 1 to L.long
                        if (L.tipo[i] ≠ entero)
                            then Z.tipo:= tipo_error
                    }
97. Zprima → in (L) { Zprima.tipo:= entero
                        for i:= 1 to L.long
                            if (L.tipo[i] ≠ entero)
                                then Zprima.tipo:= tipo_error
                        }
98. Zprima → λ {Zprima.tipo:= vacío}

```