Práctica de Compilación

Curso 2020-2021

Grupo: Triple Ese

Autor: Unai Sainz de la Maza Gamboa

Email: usainzdelamaza001@ikasle.ehu.eus

Introducción

Partimos de implementar en la parte grupal el analizador léxico y sintáctico, desarrollado el etds y se ha implementado el traductor siguiendo dicho ETDS. Una vez conseguidos estos objetivos obligatorios, para la realización de la parte individual, se han elegido varios objetivos opcionales.

Se ha realizado la especificación e implementado una nueva estructura de control, esta estructura es For siguiendo el estilo del lenguaje C. Por otro parte se han añadido expresiones booleanas (and, or, not), se han añadido las llamadas a procedimientos y las comprobaciones semánticas. Gracias a estas últimas comprobaciones, conseguimos evitar situaciones anómalas en la traducción de código intermedio.

Autoevaluación

De acuerdo a las puntuaciones de cada objetivo opcional:

5 puntos (Implementación del traductor)
- \+ 1 punto (Estructura de control nueva: For).
- \+ 1 punto (Expresiones booleanas)
- \+ 1 punto (Llamada a procedimientos)
- \+ 2 puntos (Restricciones semánticas, uso correcto de identificadores, etc.)

Análisis léxico

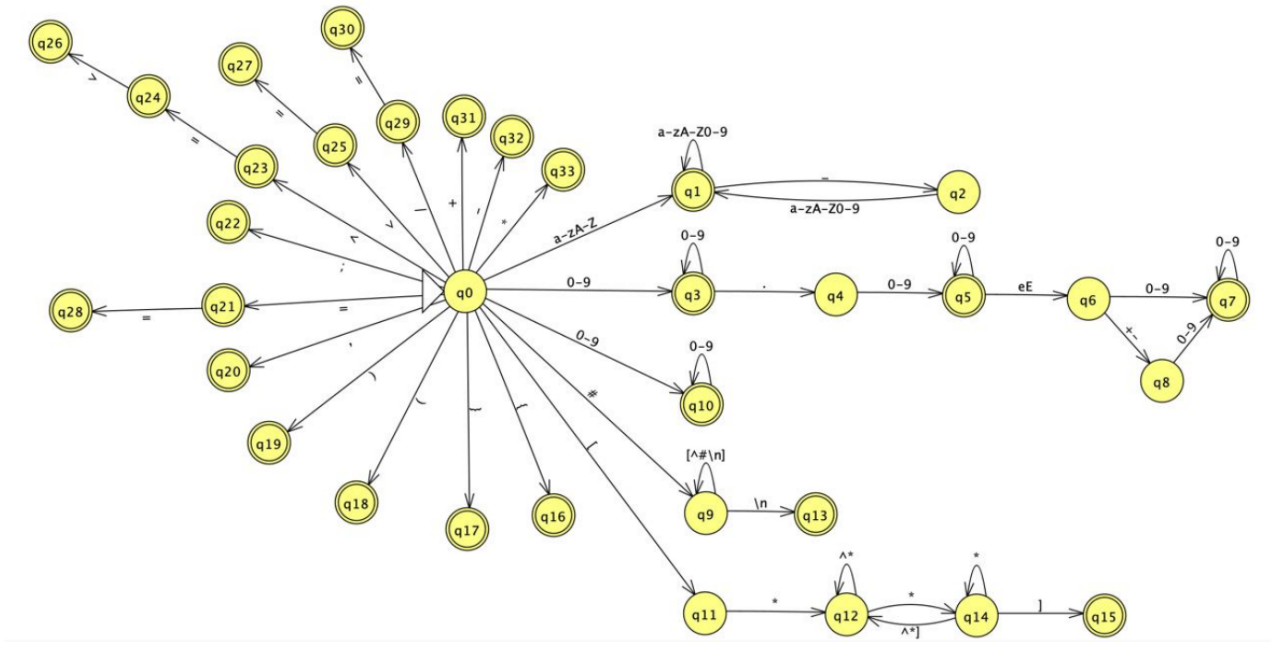
Especificación de los tokens

Nombre	Descripción	Expresión regular	Lexemas
TIDENTIFIER	Identificadores	[a-zA-Z]+(_?[a-zA-Z] | [0-9])*	- variable1 - var_as1 - var1
TINTEGER	Tipo numérico entero	[0-9]+	- 9 - 1000 - 12
TDOUBLE	Tipo numérico real	[0-9]+\.[0-9]+([eE][\+\-]?[0-9]+)?	- 0.99 - 1.11e+10 - 0.50e10
com	Comentario de línea	#[^#\n]*	- # comment - # com (a) - C[a, b]

com_multi	Comentario multilínea	(\[*)([^*]|*+[^*\]])*(*+\)	- [* c(a,b) *] - [* ()(-) *] - [****a*]
RPROGRAM	Palabra reservada program	program	program
RINTEGER	Palabra reservada int	int	int
RFLOAT	Palabra reservada float	float	float
RWHILE	Palabra reservada while	while	while
RUNTIL	Palabra reservada until	until	until
REXIT	Palabra reservada exit	exit	exit
RPROC	Palabra reservada proc	proc	proc
RIF	Palabra reservada if	if	if
RELSE	Palabra reservada else	else	else
RFOREVER	Palabra reservada forever	forever	forever
RDO	Palabra reservada do	do	do
RSKIP	Palabra reservada skip	skip	skip
RREAD	Palabra reservada read	read	read
RPRINTLN	Palabra reservada println	println	println
RAND	Palabra reservada and	and	and
ROR	Palabra reservada or	or	or
RNOT	Palabra reservada not	not	not
RFOR	Palabra reservada for	for	for
TLBRACE	Token llave izquierda	"{"	{
TRBRACE	Token llave derecha	"}"	}
TLPAREN	Token paréntesis izquierda	"("	(
TRPAREN	Token paréntesis derecha	")"	)
TCOMMA	Token coma	","	,
TASSIG	Token asignación	"="	=
TSEMIC	Token punto y coma	";"	;
TCGLE	Token para in-out	"<=>"	<=>
TCLT	Token menor	"<"	<
TCLE	Token menor igual	"<="	<=
TCGT	Token mayor	">"	>
TCGE	Token mayor igual	">="	>=
TEQUAL	Token igual	"=="	==
TNEQUAL	Token no igual	"/="	/=
TDIV	Token división	"/"	/

TPLUS	Token suma	"+"	+
TMINUS	Token resta	"-"	-
TMUL	Token multiplicar	"*"	*

Autómata



Esquema de Traducción Dirigido por la Sintaxis

Abstracciones Funcionales

añadir_inst: código x inst → código
- **Descripción**: Dada una estructura de código numerada y una inst (String), escribe inst en la siguiente línea de la estructura de código.

unir: lista x lista → lista
- **Descripción**: Dadas dos listas, devuelve la unión de las mismas.

nuevo_id: → string
- **Descripción**: no recibe parámetros, genera el siguiente id disponible y lo devuelve.

iniLista: entero→ lista
- **Descripción**: recibe un cero como parámetro, devuelve una lista vacía de enteros.

iniLista: string→ lista

- **Descripción**: recibe una cadena vacía como parámetro, devuelve una lista vacía de strings.

añadir_declaraciones: lista x tipo→ código

- **Descripción**: dada una lista de identificadores (strings) y un tipo de una clase (ent o real), escribe por cada identificador de la lista una línea con la clase, el identificador y un ";".
- **Ejemplo:** añadir_declaraciones([id1, id2], ent)
    - ent id1;
    - ent id2;

añadir_params: lista x clase_par→ código

- **Descripción:** dada una lista de identificadores (strings), un tipo y una clase_par, escribe por cada identificador de la lista una línea con la clase_par seguido de una "_", el tipo del identificador y el identificador, terminado por una ";".
- **Ejemplo:** añadir_params([id1, id2], ref, real):
    - ref_real id1;
    - ref_real id2;

añadir: lista x entero→ lista

- **Descripción:** dada una lista de enteros, añade un valor de tipo entero al final de la lista.

obtenref: código → ref_codigo

- **Descripción:** no recibe parámetros y devuelve la dirección (ref) actual.

completar: código X lista_ref_código X ref_código → código

- **Descripción**: introduce, al final, en la posición indicada por el primer entero de lista_ref_codigo, la dirección ref_codigo.
- Ejemplo:
    - Si tenemos…
        - 1. goto
        - 2. a+b
        - 3. a-b
    - Y llamamos a completar([1], 3), obtendremos el siguiente código:
        - 1. goto 3
        - 2. a+b
        - 3. a-b

error: string → string

- **Descripción:** dado un mensaje de error, imprime por pantalla otro mensaje de error con una breve descripción del problema.

esTipo: string x string → bool

- **Descripción:** dados dos tipos (strings), comprueba si el primero es el mismo tipo que el segundo.
- **Ejemplo:** es_tipo(“ent”, “ent”), devuelve true.

verificarNumArgs: string x int → error
- **Descripción:** comprueba en la pila en la que se han ido almacenando los datos, el número de argumentos que tiene el procedimiento con el nombre indicado (string). En caso de no existir el procedimiento o de no coincidir el número de elementos indicados con los que aparezcan en la tabla, dará un mensaje de error indicando y se detendrá la ejecución.

esVacia: lista → bool
- **Descripción:** dada una lista, devuelve si es vacía o no.

error: string → string
- **Descripción:** dado un mensaje de error, imprime por pantalla otro mensaje de error con una breve descripción del problema.

obtenerTiposParametro: string x int → clase_par x tipo
- **Descripción:** dado el nombre de un procedimiento y un número i, devuelve la clase_par y el tipo que se encuentra en la pila asociado al argumento i-ésimo del procedimiento con el nombre indicado.

obtenTipo: string → tipo
- **Descripción:** dado el nombre de una variable, busca y devuelve el tipo de la variable almacenada en la tabla que se encuentre en la cima de la pila. En caso de que no exista una variable con dicho nombre, dará un mensaje de error indicándolo y se detendrá la ejecución.

existeId: string → bool
- **Descripción:** dado el nombre de una variable, busca en la tabla almacenada en la pila la existencia de dicha variable. Si la encuentra, true, si no, false.

añadir_params_pila: tipo x lista x clase_par
- **Descripción:** introduce en la tabla que se encuentre en la cima de la pila, asociándolo con el procedimiento actual, todos los parámetros de la lista con el tipo y clase_par indicado.

añadir_procedimiento: string
- **Descripción:** almacena en la tabla que se encuentre en la cima de la pila un nuevo procedimiento con el nombre proporcionado.

empilar:
- **Descripción:** introduce una nueva tabla vacía en la cima de la pila.

desempilar:
- **Descripción:** elimina la tabla que se encuentre en la cima de la pila.

añadir_declaraciones_pila: tipo x lista

- **Descripción:** introduce en la tabla que se encuentre en la cima de la pila todas las variables que se encuentren en la lista con el tipo proporcionado.

Atributos

(L: léxico, S: sintetizado)

Símbolo	Nombre	Tipo	L/S	Descripción
id	nom	string	L	Contiene la cadena de caracteres del id.
num_entero	nom	string	L	Contiene la cadena de caracteres que representan el valor de un número entero.
num_real	nom	string	L	Contiene la cadena de caracteres que representan el valor de un número real.
tipo	clase	string	S	Contiene el tipo de variable, puede ser ent o real.
clase_par	tipo	string	S	Contiene el tipo, puede ser val o ref.
lista_de_ident	lnom	lista de strings	S	Lista de strings con los ids.
resto_lista_id	lnom	lista de strings	S	Lista de strings con los ids.
lista_de_sentencias	exits	lista de int	S	Lista numérica con referencias.
lista_de_sentencias	skips	lista de int	S	Lista numérica con referencias.
sentencia	exits	lista de int	S	Lista numérica con referencias.
sentencia	skips	lista de int	S	Lista numérica con referencias.
variable	nom	string	S	Contiene la cadena de caracteres del nombre de la variable.
expresion	TRUE	lista de int	S	Lista con referencia del goto que saltará en la primera parte del if.
expresion	FALSE	lista de int	S	Lista con referencia del goto que saltará a la siguiente instrucción del if.
expresion	nom	string	S	Contiene la cadena de caracteres de la expresión, o de la variable temporal donde se guardará la expresión.
expresion	tipo	string	S	Guarda la cadena de caracteres del tipo de elemento que es la

				expresión.
M	ref	int	S	Contiene la referencia numérica.
proc_call_param	lparam	lista de strings	S	Guarda una lista de las expresiones que se van a pasar como parámetros para realizar la llamada a procedimiento, solo almacena nombre de las expresiones.
resto_proc_call_param	lparam	lista de strings	S	Guarda una lista de las expresiones que se van a pasar como parámetros para realizar la llamada a procedimiento, solo almacena nombre de las expresiones.
claseTipo	clase	string	S	Guarda la clase_par de una variable.
claseTipo	tipo	string	S	Guarda el tipo de una variable.

ETDS

- Guia de los colores de los objetivos:
  - Objetivos básicos: negro y azul.
  - Comprobaciones semánticas: naranja.
  - Expresiones booleanas: verde.
  - Llamada a procedimientos: morado.
  - Nueva estructura de control: magenta.

programa → **program id** { añadir_inst(prog || || id.nom); }
declaraciones
decl_de_subprogs
{ lista_de_sentencias **}**
{
añadir_inst(halt);
desempilar();
}

declaraciones → tipo lista_de_ident
{
añadir_declaraciones(lista_de_ident.lnom, tipo.clase);
añadir_declaraciones_pila(tipo.clase, lista_de_ident.lnom);
}
; declaraciones
| ξ

lista_de_ident → **id** resto_lista_id

{

lista_de_ident.lnom = inilista(id.nom);

unir(lista_de_ident.lnom, resto_lista_id.lnom);

}

resto_lista_id → **,** **id** resto_lista_id

{

resto_lista_id.lnom = inilista(id.nombre);

resto_lista_id.lnom = unir(resto_lista_id.lnom, resto_lista_id1.lnom);

}

| ξ { resto_lista_id.lnom = inilista(“ “); }

tipo → **int** { tipo.clase = “ent”; }

| **float** { tipo.clase = “real”; }

decl_de_subprogs → decl_de_subprograma decl_de_subprogs

| ξ

decl_de_subprograma →**proc id**

{

añadir_inst(proc.nom || || id.nom);

añadirProcedimiento(id.nombre);

empilar();

}

argumentos

declaraciones

decl_de_subprogs

{ lista_de_sentencias **}**

{

añadir_inst(endproc || ;);

desempilar();

}

argumentos → **(** lista_de_param **)**

| ξ

lista_de_param → tipo clase_par lista_de_ident

{

añadir_params(lista_de_ident.lnom, clase_par.tipo, tipo.clase);

añadir_params_pila(tipo.clase, lista_de_ident.lnom, clase_par.clase);

```
        }
        resto_lis_de_param

clase_par → => { clase_par.tipo = ref; }
          | <= { clase_par.tipo = val; }
          | <=> { clase_par.tipo = ref; }

resto_lis_de_param → ; tipo clase_par lista_de_ident
        {
            añadir_params(lista_de_ident.lnom, clase_par.tipo, tipo.clase);
            añadir_params_pila(tipo.clase, lista_de_ident.lnom, clase_par.clase);
        }
        resto_lis_de_param
        | ξ

lista_de_sentencias → sentencia  lista_de_sentencias
    {
        lista_de_sentencias.exits = unir(sentencia.exits, lista_de_sentencias1.exits);
        lista_de_sentencias.skips = unir(sentencia.skips, lista_de_sentencias1.skips);
    }
    | ξ
    {
        lista_de_sentencias.exits = inilista(0);
        lista_de_sentencias.skips = inilista(0);
    }

sentencia → variable = expresion ;
          {
            if (obtenTipo(variable.nom) != expresion.tipo) {
                Error(TipoIncosistente);
            }
            añadir_inst(variable.nom || := || expresion.nom);
            sentencia.exit = inilista(0);
            sentencia.skip = inilista(0);
          }

        | if expresion {M lista_sentencias } M;
            {
                if (expresion.tipo != “bool”) {
                    Error(CondicionNoEvaluable);
                }
                completar(expresion.true, M1.ref);
                completar (expresion.false, M2.ref);
                sentencia.exit = lista_sentencia.exit;
```

```
            sentencia.skip = lista_sentencia.skip;
    }

| while forever {M lista_sentencias } M;
  {añadir_inst(goto || M1.ref || ;);
  completar (lista_sentencias.exit, M2.ref+1);
  sentencia.exit = inilista(0);
  sentencia.skip = inilista(0);}

| do { M lista_sentencias } until M expresion
      { if (expresion1.tipo != "bool")
            Error(CondicionNoEvaluable);
      }
else {M lista_sentencias } M ;
  {completar(expresion.true, M3.ref);
  completar(expresion.false, M1.ref);
  completar(lista_sentencias1.skip, M2.ref);
  completar(lista_sentencias1.exit, M4.ref);
  completar(lista_sentencias2.exit, M4.ref);
  sentencia.exit = inilista(0);
  sentencia.skip = inilista(0);}

| skip if expresion M ;
{
      if (expresion1.tipo != "bool") {
            Error(CondicionNoEvaluable);
      }
      completar(expresion.false, M1.ref );
      sentencia.skip := expresion.true;
      sentencia.exit = inilista(0);
}

| exit M ;
  {sentencia.skips = inilista(0);
  sentencia.exits = inilista(M1.ref);
  añadir_inst(goto);}

| read ( variable ) ;
  {añadir_inst(read || || variable.nom || ;);
  sentencia.exits = inilista(0);
  sentencia.skips = inilista(0);}

| println ( expresion ) ;
  {añadir_inst(write || || expresion.nom || ;);
  añadir_inst(writeln || ;);
  sentencia.exits = inilista(0);
  sentencia.skips = inilista(0);}
```

```
| for ( tipo id = expresion
{
        if (expresion1.tipo != tipo1.clase) {
                Error(TipoInconsistente);
        }
        añadir_inst(id.nom || ":=" || expresion1.nom || ";");
}
; M expresion
{
        if (expresion.tipo != "bool") {
                Error(Error_Tipo);
        }
}
M ; variable = expresion ) { lista_de_sentencias M } ;
{
        if (id1.nom != variable1.nom) {
                Error(VariableInadecuada);
        }
        if (expresion3.tipo != variable1.tipo) {
                Error(TIpoInconsistente);
        }
        añadir_inst(variable1.nom || " := " || expresion3.nom || ";");
        añadir_inst("goto " || M1.ref || ";");
        completar(expresion2.true, M2.ref);
        completar(expresion2.false, M3.ref+2);
        completar(lista_de_sentencias.exits, M3.ref+2);
        completar(lista_de_sentencias.skips, M1.ref);
        sentencia.exits = inilista(0);
        sentencia.skips = inilista(0);
}

| id ( proc_call_param) ;
{
        verificarNumArgs(id.nom, size(proc_call_param1));
        for (int i = 0; i < size(proc_call_param1); i++) {
                // para cada expresion de la lista…
                claseTipo = obtenerTiposParametro(id.nombre, i);
                if (existeId(proc_call_param1[size(proc_call_param1)-i]) {
                        if (obtenTipo(proc_call_param1[size(proc_call_param1)-i] !=
                        claseTipo.tipo) {
                                Error(Error_TIpo);
                        }
                }
                añadir_inst("param_" || claseTipo.first || " " ||
                        proc_call_param[size(proc_call_param)-i]);
        }
```

```
            añadir_inst("call " || id.nom);
            sentencia.exits = inilista(0);
            sentencia.skips = inilista(0);
        }


proc_call_param → expresion resto_proc_call_param
        {
            proc_param_param.lparam = añadir(resto_proc_call_param1.lparam,
                expresion1.nom);
        }
        | ξ { proc_call_param.lparam = inilista(""); }

resto_proc_call_param → , expresion resto_proc_call_param
        {
            resto_proc_call_param.lparam = añadir(resto_proc_call_param1.lparam,
                expresion.nom);
        }
        | ξ { resto_proc_call_param1.lparam = inilista(""); }

variable → id { variable.nom = id.nom; variable.tipo = obtenTipo(variable.nom); }

expresion →  expresion = expresion
         {
            if (expresion1.tipo != expresion2.tipo) {
                Error(TipoInconsistente);
            }
            expresion.nom = iniNom();
            expresion.tipo = "bool";
            expresion.true = inilista(obtenref());
            expresion.false = inilista(obtenref()+1);
            añadir_inst(if || expresion1.nom || = || expresion2.nom || goto);
            añadir_inst(goto);
        }
        | expresion > expresion
        {
            if (! esTipoNumerico(expresion1.tipo, expresion2.tipo) {
                Error(Error_Tipo);
            }
            expresion.nom = iniNom();
            expresion.tipo = "bool";
            expresion.true = inilista(obtenref());
            expresion.false = inilista(obtenref()+1);
            añadir_inst(if || expresion1.nom || > || expresion2.nom || goto);
```

```
        añadir_inst(goto);
}
| expresion < expresion
{
        if (! esTipoNumerico(expresion1.tipo, expresion2.tipo) {
                Error(Error_Tipo);
        }
        expresion.nom = iniNom();
        expresion.tipo = “bool”;
        expresion.true = inilista(obtenref());
        expresion.false = inilista(obtenref()+1);
        añadir_inst(if || expresion1.nom || < || expresion2.nom || goto);
        añadir_inst(goto);
}
| expresion >= expresion
{
        if (! esTipoNumerico(expresion1.tipo, expresion2.tipo) {
                Error(Error_Tipo);
        }
        expresion.nom = iniNom();
        expresion.tipo = “bool”;
        expresion.true = inilista(obtenref());
        expresion.false = inilista(obtenref()+1);
        añadir_inst(if || expresion1.nom || >= || expresion2.nom || goto);
        añadir_inst(goto);
}
| expresion <= expresion
{
        if (! esTipoNumerico(expresion1.tipo, expresion2.tipo) {
                Error(Error_Tipo);
        }
        expresion.nom = iniNom();
        expresion.tipo = “bool”;
        expresion.true = inilista(obtenref());
        expresion.false = inilista(obtenref()+1);
        añadir_inst(if || expresion1.nom || <= || expresion2.nom || goto);
        añadir_inst(goto);
}
| expresion /= expresion
{
        if (expresion1.tipo != expresion2.tipo) {
                Error(Error_Tipo);
        }
        expresion.nom = iniNom();
        expresion.tipo = “bool”;
        expresion.true = inilista(obtenref());
        expresion.false = inilista(obtenref()+1);
```

```
        añadir_inst(if || expresion1.nom || /= || expresion2.nom || goto);
        añadir_inst(goto);
}
| expresion and M expresion
{
        if (expresion1.tipo != “bool” || expresion2.tipo != “bool”) {
                Error(Error_Tipo);
        }
        expresion.nom = iniNom();
        expresion.tipo = “bool”;
        completar(expresion1.true, M1.ref);
        expresion.true = expresion2.true;
        expresion.false = unir(expresion1.false, expresion2.false);
}
| expresion or M expresion
{
        if (expresion1.tipo != “bool” || expresion2.tipo != “bool”) {
                Error(Error_Tipo);
        }
        expresion.nom = iniNom();
        expresion.tipo = “bool”;
        completar(expresion1.false, M1.ref);
        expresion.true = unir(expresion1.true, expresion2.true);
        expresion.false = expresion2.false;
}
| not expresion
{
        if (expresion1.tipo != “bool”) {
                Error(Error_Tipo);
        }
        expresion.nom = iniNom();
        expresion.tipo = “bool”;
        expresion.true = expresion1.false;
        expresion.false = expresion1.true;
}
| expresion + expresion
{
        if (! esTipoNumerico(expresion1.tipo, expresion2.tipo) {
                Error(Error_Tipo);
        }
        expresion.nom = nuevo_id();
        expresion.tipo = expresion1.tipo;
        añadir_inst(expresion.nom || = || expresion1.nom || + || expresion2.nom);
        expresion.true = inilista(0);
        expresion.false = inilista(0);
}
| expresion - expresion
```

```
{
        if (! esTipoNumerico(expresion1.tipo, expresion2.tipo) {
                Error(Error_Tipo);
        }
        expresion.nom = nuevo_id();
        expresion.tipo = expresion1.tipo;
        añadir_inst(expresion.nom || = || expresion1.nom || - || expresion2.nom);
        expresion.true = inilista(0);
        expresion.false = inilista(0);
}
| expresion * expresion
{
        if (! esTipoNumerico(expresion1.tipo, expresion2.tipo) {
                Error(Error_Tipo);
        }
        expresion.nom = nuevo_id();
        expresion.tipo = expresion1.tipo;
        añadir_inst(expresion.nom || = || expresion1.nom || * || expresion2.nom);
        expresion.true = inilista(0);
        expresion.false = inilista(0);}
  | expresion / expresion
  {
        if (! esTipoNumerico(expresion1.tipo, expresion2.tipo) {
                Error(Error_Tipo);
        }
        añadir_inst("if" || expresion2.nom || "= 0" || ErrorDiv0 || ";");
        expresion.nom = nuevo_id();
        expresion.tipo = expresion1.tipo;
        añadir_inst(expresion.nom || = || expresion1.nom || / || expresion2.nom);
        expresion.true = inilista(0);
        expresion.false = inilista(0);
}
| variable
{
        expresion.nom = variable.nom;
        expresion.tipo = variable.tipo;
        expresion.true = inilista(0);
        expresion.false = inilista(0);
}
| num_entero
{
        expresion.nom = num_entero.nom;
        expresion.tipo = "ent";
        expresion.true = inilista(0);
        expresion.false = inilista(0);
}
| num_real
```

```
{
        expresion.nom = num_real.nom;
        expresion.tipo = “real”;
        expresion.true = inilista(0);
        expresion.false = inilista(0);
}
| ( expresion )
{
        expresion.nom = expresion1.nom;
        expresion.tipo = expresion1.tipo;
        expresion.true = expresion1.true;
        expresion.false = expresion1.false;
}
```

M →ξ { M.ref = obtenRef();

Aclaraciones

- En el código, en lo que se refiere a las operaciones aritméticas se ha añadido una especie de casteo entre los tipos, no se ha añadido al ETDS por no complicar demasiado las comprobaciones semánticas de todas las operaciones, pero sería el siguiente esquema:
    - Donde tenemos declarada una función llamada operaciónAritmética, que recibe como parámetros: la variable $$, las estructuras de los dos operadores y la operación a realizar.
    - Esta función comprueba los tipos de la siguiente forma:
        - Si el primer operador es de tipo entero, y el segundo es de tipo real, el tipo final después de realizar la operación será de tipo real. Pero el casteo será del primer operador.
        - Si el primer operador es de tipo real, y el segundo operador es de tipo entero, el tipo final será de tipo real. Pero el casteo será del segundo operador.
        - Si los dos operadores tienen el mismo tipo:
            - Si la operación a realizar es una división, el tipo final será de tipo real.
            - Si no, se obtiene el tipo del segundo operador.
    - Una vez tengamos que tipo de casteo tenemos, añadimos las instrucciones pertinentes para el casteo.
- Este mismo proceso de casteo se ha añadido a la hora de asignar una expresión a una variable, de esta forma introducimos este paso intermedio para evitar sacar el error cuando por ejemplo queremos asignar un real a una variable entero.
    - Todo lo anterior se hace en la función, pero El ETDS quedaría (para la suma) así:

      | expresion + expresion
      {

```
                if (esTipo(“ent”, exp1.tipo) && esTipo(“real”, exp2.tipo){
                        tipoFinal = “real”;
                        opCast = 1;
                        tmp = nuevoId();
                } else if (esTipo(“ent”, exp1.tipo) && esTipo(“real”, exp2.tipo){
                        tipoFinal = “real”;
                        opCast = 2;
                        tmp = nuevoId();
                } else if (esTipo(exp1.tipo, exp2.tipo)) {
                        // en el caso de la division, el tipo final aquí es “real”.
                        tipoFinal = exp2.tipo; // sin variable para tmp casteo.
                } else {
                        Error(BooleanosEnAritmeticas);
                }
                $$->nom = nuevoId();
                $$->trues = iniLista(0);
                $$->falses = iniLista(0);
                $$->tipo = tipoFinal;

                // casteo
                if (opCast == 0) {
                        añadir_inst($$->nom || “ := “ || exp1.nom || “+“ || exp2.nom || “;”);
                } else if (opCast == 1) {
                        añadir_inst(tmp || “ := ent2real ” || exp1.nom || “;”);
                        añadir_inst($$->nom || “ := “ || tmp || “+“ || exp2.nom || “;”);
                } else if (opCast == 2) {
                        añadir_inst(tmp || “ := ent2real ” || exp2.nom || “;”);
                        añadir_inst($$->nom || “ := “ || exp1.nom || “+“ || tmp || “;”);
                }
        }
```

- En el código también hemos añadido la opción de declarar la variable con la que vamos a iterar dentro de la propia estructura de control.
- Los **skip if** dentro del **for** saltan a evaluar la expresión booleana, y los **exit** salen fuera del mismo.

Pruebas

PruebaBuena1

Entrada:

```
program ejemplo_con_nombre_muy_largo

 int a, b, c ;
 float d, e ;
 [* esto es un **
```

```
 comentario multilinea *]

   proc suma(int <= x; int <= y; int <=> resul) int aux, iteraciones;
   {
              aux = x;
              resul = y;
              if resul < 1000  {
                iteraciones = 0;
                do {
                      resul = resul + 1;
                      skip if resul > 1000000;
                      aux = aux - 1;
                      iteraciones = iteraciones + 1;
                     } until aux == 0
                else {
                      if resul < 0 { exit; };
                      println(iteraciones);
                };    #fin del do
             };  #fin del if
   } # fin de proc

  {
  read(a); read(b);
  d = 1/b;
  e = 0.1e-1/a;
  suma(a,b,c); [* Llamada a PROCEDIMIENTO *]
  c = c*(c*d)+e;
  println(c*c);
}
```

Salida:

```
1: program ejemplo_con_nombre_muy_largo;
2: ent a;
3: ent b;
4: ent c;
5: real d;
6: real e;
7: proc suma;
8: val_ent x;
9: val_ent y;
10: ref_ent resul;
11: ent aux;
12: ent iteraciones;
13: aux := x;
14: resul := y;
15: if resul < 1000 goto 17;
16: goto 33;
17: iteraciones := 0;
18: _t1 := resul + 1;
19: resul := _t1;
```

```
20: if resul > 1000000 goto 26;
21: goto 22;
22: _t2 := aux - 1;
23: aux := _t2;
24: _t3 := iteraciones + 1;
25: iteraciones := _t3;
26: if aux = 0 goto 28;
27: goto 18;
28: if resul < 0 goto 30;
29: goto 31;
30: goto 33;
31: write iteraciones;
32: writeln;
33: read a;
34: read b;
35: if b = 0 goto ErrorDiv0;
36: _t4 := 1 / b;
37: d := _t4;
38: if a = 0 goto ErrorDiv0;
39: _t5 := ent2real a;
40: _t6 := 0.1e-1 / _t5;
41: e := _t6;
42: param_val a;
43: param_val b;
44: param_ref c;
45: call suma;
46: _t7 := ent2real c;
47: _t8 := _t7 * d;
48: _t9 := ent2real c;
49: _t10 := _t9 * _t8;
50: _t11 := _t10 + e;
51: _t12 := real2ent _t11;
52: c := _t12;
53: _t13 := c * c;
54: write _t13;
55: writeln;
56: halt;
ha finalizado BIEN…
```

PruebaBuena2

Entrada:

```
program programa_ejemplo2

int n1, n2, n3, n_total, var1;
float e, a;

   proc mult(int <= n1; int <= n2; int <=> n3)
   {
            if n1 > n2 {
            n3 = n1 * n2;
            }; # fin del if
```

```
            if n2 <= n1 {
            n3 = (n1 * n1) + n2;
            }; # fin del if

            while forever {
               [* Ejecuta la lectura y escritura y sale con el exit, solo para ver
                  que el exit funciona.
                *]
                  read(var1);
                  println(var1);
                  exit;
             }; # fin de while forever
   } # fin de proc mult

{
            read(n1); read(n2);
            e = 0.1e-1 / a;
            mult(n1 * 2, n2 + 1, n3); [* Operaciones en la llamada al procedimiento *]
            n_total = n3 * e;
            println(n_total);
}
```

Salida:

```
1: program programa_ejemplo2;
2: ent n1;
3: ent n2;
4: ent n3;
5: ent n_total;
6: ent var1;
7: real e;
8: real a;
9: proc mult;
10: val_ent n1;
11: val_ent n2;
12: ref_ent n3;
13: if n1 > n2 goto 15;
14: goto 17;
15: _t1 := n1 * n2;
16: n3 := _t1;
17: if n2 <= n1 goto 19;
18: goto 22;
19: _t2 := n1 * n1;
20: _t3 := _t2 + n2;
21: n3 := _t3;
22: read var1;
23: write var1;
24: writeln;
25: goto 27;
26: goto22;
27: read n1;
28: read n2;
```

```
29: if a = 0 goto ErrorDiv0;
30: _t4 := 0.1e-1 / a;
31: e := _t4;
32: _t5 := n1 * 2;
33: _t6 := n2 + 1;
34: param_val _t5;
35: param_val _t6;
36: param_ref n3;
37: call mult;
38: _t7 := ent2real n3;
39: _t8 := _t7 * e;
40: _t9 := real2ent _t8;
41: n_total := _t9;
42: write n_total;
43: writeln;
44: halt;
ha finalizado BIEN…
```

PruebaBuena3

Entrada:

```
program sumaPrimerosDiez
int acumulador, entrada;
{
        acumulador = 0;
        read(entrada);

        for (int i = 1; i <= 10; i = i + 1) {
                acumulador = acumulador + i;

                [* Sale del for si acumulador < 0 y la entrada es > 0 *]
                if (acumulador < 0 and entrada > 0) {
                        exit;
                };
        };
        println(acumulador);
}
```

Salida:

```
1: program sumaPrimerosDiez;
2: ent acumulador;
3: ent entrada;
4: acumulador := 0;
5: read entrada;
6: ent i;
7: i := 1;
8: if i <= 10 goto 10;
9: goto 20;
10: _t1 := i + 1;
11: _t2 := acumulador + i;
12: acumulador := _t2;
13: if acumulador < 0 goto 15;
```

```
14: goto 18;
15: if entrada > 0 goto 17;
16: goto 18;
17: goto 20;
18: i := _t1;
19: goto 8;
20: write acumulador;
21: writeln;
22: halt;
ha finalizado BIEN...
```

PruebaBuena4

Entrada:

```
program trianguloPascal
int rows, coef;
{
	read(rows);

	for(int i = 0; i < rows; i = i + 1) {

		for(int space = 1; space <= rows - i; space = space + 1) {
			println(0);
		};

		for(int j = 0; j <= i; j = j + 1){ [* FOR *]
			if (j == 0 or i == 0) { [* BOOLEANOS *]
				coef = 1;
			};

			if (j /= 0 and i /= 0) { [* BOOLEANOS *]
				coef = coef * ( i - j + 1 ) / j;
			};
			println(coef);
			println(0);
		};
		println(0);
	};
}
```

Salida:

```
1: program trianguloPascal;
2: ent rows;
3: ent coef;
4: read rows;
5: ent i;
6: i := 0;
7: if i < rows goto 9;
8: goto 51;
9: _t1 := i + 1;
10: ent space;
11: space := 1;
```

```
12: _t2 := rows - i;
13: if space <= _t2 goto 15;
14: goto 20;
15: _t3 := space + 1;
16: write 0;
17: writeln;
18: space := _t3;
19: goto 12;
20: ent j;
21: j := 0;
22: if j <= i goto 24;
23: goto 47;
24: _t4 := j + 1;
25: if j = 0 goto 29;
26: goto 27;
27: if i = 0 goto 29;
28: goto 30;
29: coef := 1;
30: if j /= 0 goto 32;
31: goto 41;
32: if i /= 0 goto 34;
33: goto 41;
34: _t5 := i - j;
35: _t6 := _t5 + 1;
36: _t7 := coef * _t6;
37: if j = 0 goto ErrorDiv0;
38: _t8 := _t7 / j;
39: _t9 := real2ent _t8;
40: coef := _t9;
41: write coef;
42: writeln;
43: write 0;
44: writeln;
45: j := _t4;
46: goto 22;
47: write 0;
48: writeln;
49: i := _t1;
50: goto 7;
51: halt;
ha finalizado BIEN…
```

PruebaBuena5

Entrada:

```
program pruebas5
int acumulador, sumaParcial;

proc sumaDos(int <= x; int <= y; int <=> resul){
		resul = x + y;
}
```

```
{
        acumulador = 0;

        for (int i = 1; i < 10; i = i + 2) {
            skip if (acumulador > 10); [* Simplemente para probar SKIP IF *]
            sumaDos(i, i + 1, sumaParcial); [* Llamada a procedimiento *]
            acumulador = acumulador + sumaParcial;
        };

        if (not acumulador > 10 and sumaParcial < 10) { [* BOOLEANOS *]
            println(sumaParcial);
        };
        # acumulador contiene el resultado final de la suma de los primeros 9 números
}
```

Salida:

```
1: program pruebas5;
2: ent acumulador;
3: ent sumaParcial;
4: proc sumaDos;
5: val_ent x;
6: val_ent y;
7: ref_ent resul;
8: _t1 := x + y;
9: resul := _t1;
10: acumulador := 0;
11: ent i;
12: i := 1;
13: if i < 10 goto 15;
14: goto 27;
15: _t2 := i + 2;
16: if acumulador > 10 goto 13;
17: goto 18;
18: _t3 := i + 1;
19: param_val i;
20: param_val _t3;
21: param_ref sumaParcial;
22: call sumaDos;
23: _t4 := acumulador + sumaParcial;
24: acumulador := _t4;
25: i := _t2;
26: goto 13;
27: if acumulador > 10 goto 33;
28: goto 29;
29: if sumaParcial < 10 goto 31;
30: goto 33;
31: write sumaParcial;
32: writeln;
33: halt;
ha finalizado BIEN...
```

PruebaMala1

Entrada:

```
program enRango

[* Este programa escribe 1 si un número está en el rango, y 0 en caso contrario.
   Error porque se asigna una expresión booleana a un entero.*]

int x, limiteInf, limiteSup;
int estaEnRango;
{
        read(x); read(limiteInf); read(limiteSup);
        estaEnRango = limiteInf >= x and x <= limiteSup;
        if estaEnRango then {
                println(1);
        };
        if not estaEnRango then {
                println(0);
        };
}
```

Salida:

```
line 10: Error semántico. No se puede asignara una variable de tipo bool a otra de tipo ent.
line 11: syntax error, unexpected TIDENTIFIER
ha finalizado MAL...
```

PruebaMala2

Entrada:

```
program programa_ejemplo2

int n1, n2, n3, n_total;

[*
    - prueba mala con "procedure" en vez de "proc"
*]

  procedure mult(int <= n1; int <= n2; int <=> n3)
  {
        if n1 > n2 {
                n3 = n1 * n2;
                exit;
        }; # fin del if

        if n2 <= n1 {
                n3 = (n1 * n1) + n2;
        }; # fin del if
   } # fin de proc mult

{
        read(n1); read(n2);
        e = 0.1e-1/a;
        # mult(n1, n2, n3);
```

```
		n_total = n3 * e;
		println(n_total);
}
```

Salida:

```
./parser <Pruebas/PruebaMala2.in
ha comenzado...

line 9: syntax error, unexpected TIDENTIFIER, expecting TLBRACE
ha finalizado MAL...
```

PruebaMala3

Entrada:

```
program sumaPrimerosDiez
[*
   Este programa calcula la suma de los primeros 10 números enteros, declarando la variable
   del for fuera.
*]

int acumulador;
int i;
{
		acumulador = 0;
		i = 1;
		for (i <= 10; i = i + 1) { # Se declara la variable que recorre el bucle fuera -> ERROR
			acumulador = acumulador + i;
		};
		println(acumulador);
}
```

Salida:

```
./parser <Pruebas/PruebaMala3.in
ha comenzado...

line 11: syntax error, unexpected TIDENTIFIER, expecting RFLOAT or RINTEGER
ha finalizado MAL...
```

PruebaMala4

Entrada:

```
program sumaPrimerosDiez
[*
   Este programa calcula la suma de los primeros 10 números enteros, declarando la variable
		del for fuera.
*]

int acumulador;
{
		acumulador = 0;
		for (int i; i <= 10; j = i + 1) { # No actualizamos la variable correcta... ERROR
			acumulador = acumulador + i;
		};
```

```
		println(acumulador);
}
```
Salida:
./parser <Pruebas/PruebaMala4.in
ha comenzado...

line 10: Error semántico. Has intentado utilizar la variable j antes de declararla.
line 12: Error semántico. Se debe actualizar la variable i no la variable j.
ha finalizado MAL...

PruebaMala5

Entrada:
```
program programa_ejemplo2

float a;
int e;

  proc mayor(float <= a; int <=> e)
  {
		if (a > e * 10) {
			e = a; [* ERROR: no coinciden los tipos *]
		};
  } # fin de proc mult

{
		read(a);
		read(e);
		mayor(e, a); [* ERROR: los tipos en la llamada no coinciden *]
}
```

Salida:
./parser <Pruebas/PruebaMala5.in
ha comenzado...

line 16: Error semántico. Los tipos real y ent no concuerdan.
ha finalizado MAL...

PruebaMala6

Entrada:
```
program booleanos
int a;
{
		println(a < 10);

		a = a > 10;

		e = a - 10;
}
```

Salida:

./parser <Pruebas/PruebaMala6.in
ha comenzado...

line 6: Error semántico. No se puede asignara una variable de tipo bool a otra de tipo ent.
line 8: Error semántico. Has intentado utilizar la variable e antes de declararla.
ha finalizado MAL...

PruebaMala7

Entrada:

```
program sumaPrimerosNueve
[*
   Este programa calcula la suma de los primeros 9 números enteros llamando al procedimiento
sumaDos.
*]

int acumulador, sumaParcial;

proc sumaDos(int <= x; int <= y; int <=> resul){
        resul = x + y;
}
{
        acumulador = 0;

        for (int i = 1; i < 10; i = i + 2) {
                sumaDos(i, i + 1, sumaParcial); [* Llamada a procedimiento *]
                acumulador = acumulador + sumaParcial;
        };

        if (not acumulador > 10 and sumaParcial < 10) { [* BOOLEANOS *]
                print(sumaParcial); [* Error Uso de print(), no existe proc *]
        };

        # acumulador contiene el resultado final de la suma de los primeros 9 números
}
```

Salida:

./parser <Pruebas/PruebaMala7.in
ha comenzado...

line 21: Error semántico. No se ha encontrado procedimiento print. Puede que el nombre o el número de parámetros sean incorrectos.
ha finalizado MAL…

PruebaMala8

Entrada:

```
program condicionesNumericas {

        [* No se puede colocar una expresión aritmética como condición. *]

        for (float i = 1; 8 + 7; i = i + 1.0) {
```

```
                println(i);
                if (i + 3) then {
                        println(0);
                };
                skip if 65 * 98;
        };
}
```

Salida:

```
./parser <Pruebas/PruebaMala8.in
ha comenzado...

line 5: Error semántico. La condición de parada debe ser de tipo booleano!
line 7: syntax error, unexpected TIDENTIFIER
ha finalizado MAL...
```