Práctica de Compilación

Mikel Dalmau y Diego Lorenzo

22 de Mayo de 2015

Resumen

El siguiente documento comprende todo el proceso de especificación lexica, semántica y traducción a lenguaje intermedio que es necesario llevar a cabo para crear un compilador. Así, también se ha ampliado la gramática del lenguaje para aceptar expresiones booleanas.

Contenido

Año: 2014 - 2015
Profesor(a): Nerea Ezeiza y Begona Losada
Teléfono:
E-mail: mdalmau002@ikasle.ehu.es

GIPUZKOAKO CAMPUSA
CAMPUS DE GUIPUZCOA
Paseo Manuel de Lardizabal, 1
20018 Donostia (Gipuzkoa)

1 Especificación Léxica y Autómata

TIPO DE TOKEN	DESCRIPCION	EJEMPLOS	E. REGULAR
Comentario	Comienza con la secuencia ($*$ y acaba en $*$) en medio cualquier cosa menos $*$).	($*$ ejemplo $*$)	$\backslash(\ \backslash* \ ([\hat{}*] \mid \backslash* + [\hat{}*)])^* \backslash* + \backslash)$
Palabra Reservada	programs, comienzo, procedimiento, fin, entero, real, variables, in, out, si, entonces, hacer, hasta, romper, escribir_linea, leer		
Identificador	Letra seguido de letras y dígitos o guiones bajos de 1 en 1, no puede terminar en guión.	id, Id, id1, id_1, id_1_2_a	$[a-zA-Z]\,(\_?\,[a-zA-Z0-9])^*$
Asignación	$=$		
Comparador	$== \ <= \ >= \ < \ > \ /=$		
Operador	$+ \ - \ * \ /$		
Separador	$; \ , \ (\ )$		
Blancos	Salto de línea, blanco, tabulador.		$[\ \backslash t \backslash n]$
Cte entera	Secuencia de dígitos.	1, 33344532, 0	$[0-9]+$
Cte real	Secuencia de dígitos con decimales y opcionalmente notación exponencial	0.1, 5.0, 45.23e+9, 45.23E-99, 1.234e2	$[0-9]+\backslash.\,[0-9]+$ $([eE]\,[+-]?\,[0-9]+)?$

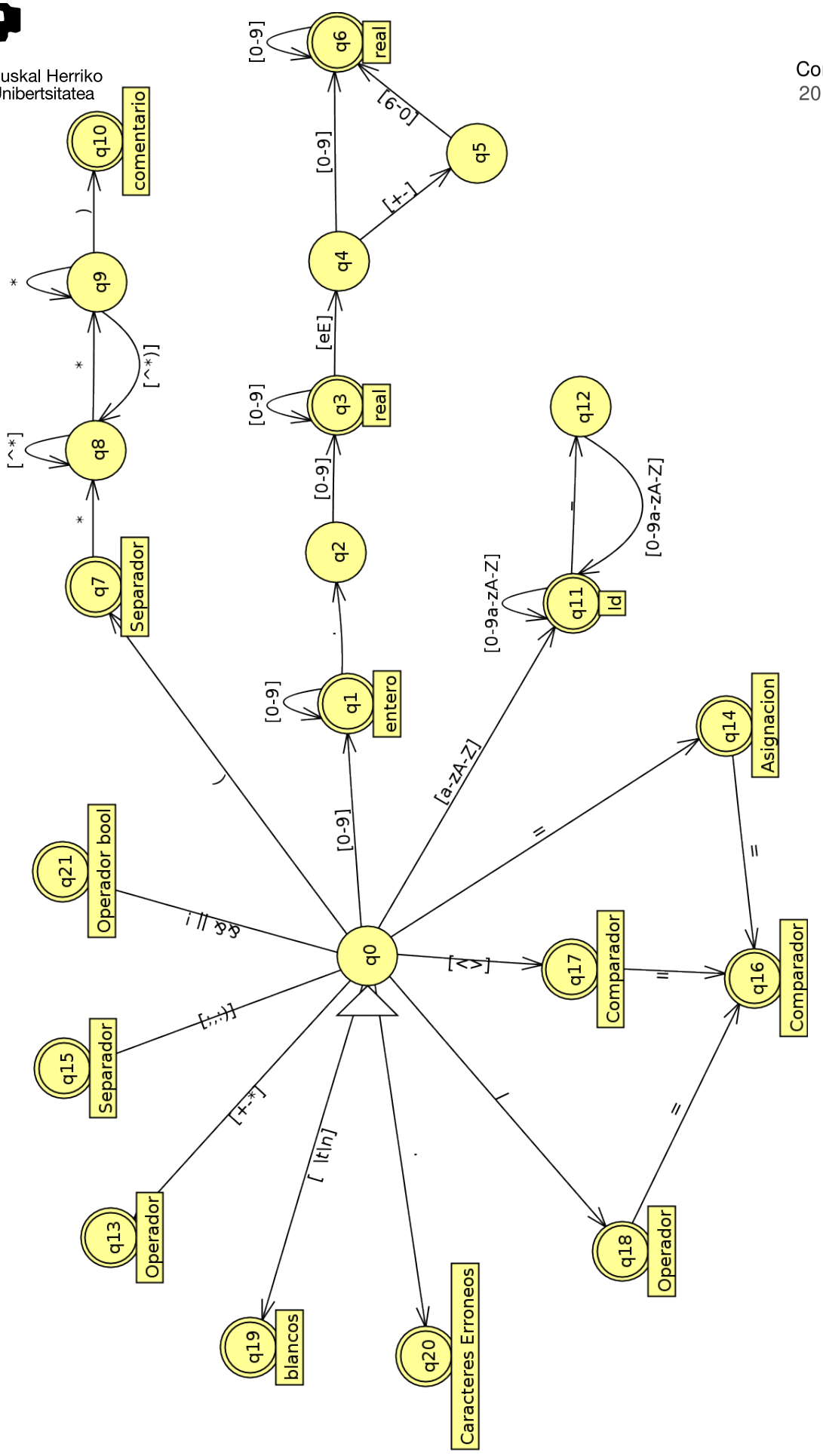

Figura 1: Autómata de la especificación Lésica

2 Definición sintáctica del lenguaje

Sea han añadido en color verde las produciones correspondientes a la ampliación de la gramática para el uso de booleanos booleanos.

programa → **programa id**
declaraciones
decl_de_subprogs
comienzo
lista_de_sentencias
fin ;

declaraciones → **variables** lista_de_ident : tipo ; declaraciones
| ξ

lista_de_ident → **id** resto_lista_id

resto_lista_id → , **id** resto_lista_id
| ξ

tipo → **entero** | **real**

decl_de_subprogs → decl_de_subprograma decl_de_subprogs
| ξ

decl_de_subprograma → **procedimiento id** argumentos declaraciones **comienzo** lista_de_sentencias **fin** ;

argumentos → (lista_de_param)
| ξ

lista_de_param → lista_de_ident : clase_par tipo resto_lis_de_param

clase_par → **in** | **out** | **in out**

resto_lis_de_param → ; lista_de_ident : clase_par tipo resto_lis_de_param
| ξ

lista_de_sentencias → sentencia lista_de_sentencias
| ξ

sentencia → variable = expresion ;
| **si** expresion **entonces** lista_de_sentencias **fin** ;
| **hacer** lista_de_sentencias **hasta** expresion **fin** ;
| **romper si** expresion ;
| **leer** (variable) ;
| **escribir_linea** (expresion) ;

variable → **id**

$$
\begin{aligned}
\text{expresion} \rightarrow\ & \text{expresion} == \text{expresion} \\
| \ & \text{expresion} > \text{expresion} \\
| \ & \text{expresion} < \text{expresion} \\
| \ & \text{expresion} \geq \text{expresion} \\
| \ & \text{expresion} \leq \text{expresion} \\
| \ & \text{expresion} \neq \text{expresion} \\
| \ & \text{expresion} + \text{expresion} \\
| \ & \text{expresion} - \text{expresion} \\
| \ & \text{expresion} * \text{expresion} \\
| \ & \text{expresion} / \text{expresion} \\
| \ & \text{expresion \&\& expresion} \\
| \ & \text{expresion} \,||\, \text{expresion} \\
| \ & \text{expresion ! expresion} \\
| \ & \textbf{id} \\
| \ & \textbf{num\_entero} \\
| \ & \textbf{num\_real} \\
| \ & \text{(expresion)}
\end{aligned}
$$

3 Atributos Definidos

- **programa:** *No terminal*, no tiene atributo, es el comienzo de la gramática.
- **declaraciones:** *No terminal*, no tiene atributos, sirve para definir en que lugar del programa se situarán las declaraciones de variables.
- **lista_de_ident:** *No terminal.*
  - listaId: Es una lista que contiene identificadores y se utiliza en el terminal de las declaraciones.
- **resto_lista_id:** *No terminal.*
  - listaId: Es una lista que contiene identificadores y se utiliza en el terminal de las declaraciones.
- **tipo:** *Terminal.*
  - tipoVar: Un cadena de caracteres que indique si se trata de un entero o un real que son los dos únicos tipos de variables que utilizamos.
- **decl_de_subprogs:** *No terminal*, no tiene atributos.
- **decl_de_subprograma:** *No terminal*, no tiene atributos.
- **argumentos:** *No terminal.*
- **lista_de_param:** *No terminal.*
- **clase_par:** *Terminal.*
  - clasePar: Un cadena de caracteres que representa que tipo de argumento es, de entrada, de salida o de ambas, *(in, out, in out)*.
- **resto_lis_de_param:** *No terminal.*
  - listaPar: Es una lista que por cada elemento contiene 3 datos, clase del parámetro *(in, out, in out)*, identificador y tipo del parámetro *(entero o real)*.
- **lista_de_sentencias:** *No terminal.*
  - exit: Lista de enteros que guarda la referencias a la línea final de las sentecias que son estructuras de control como hacer hasta, o romper si.
- **sentencia:** *No terminal.*
  - exit: Lista de enteros que guarda la referencias a la línea final de las sentecias que son estructuras de control como hacer hasta, o romper si.
- **variable:** *Terminal.*
  - nombre: Cadena de caracteres indicando el nombre de la variable.
- **expresión:** *Terminal.*
  - true: lista de enteros que contiene las referencias a las instrucciones que han sido evaluadas como verdaderas.
  - false: lista de enteros que contiene las referencias a las instrucciones que han sido evaluadas como falsas.
  - nombre: Cadena de caracteres de la expresión entera que luego se utiliza como parámetro de las funciones, completar, añadir_inst...
- **M:** *Terminal.*
  - ref: Entero que contiene la línea de la siguiente instrucción.

4 Abstracciones Funcionales

- **añadir_inst(string):** Añade una nueva instrucción que se forma a partir de una serie de cadenas de caracteres y se separan con el símbolo ‖ para indicar un espacio en blanco.
  En el ETDS se ha utilizado sobre todo esta abstracción, y sirve para traducir instrucciones a lenguaje intermedio.
- **añadir_declaraciones(lista , tipo):** Por cada identificador de la lista de entrada, añade la siguiente instrucción:
  - id : tipo
- **añadir(lista, nombre):** Añade el nombre a la lista y devuelve la lista actualizada.
- **inilista(nombre):** Crea una lista con el "nombre" y devuelve la lista recién creada.
- **completar(lista,entero):** Completa las instrucciones referenciadas por cada número de la lista y luego completa la instrucción referenciada por el segundo parámetro.
- **unir(lista_1,lista_2):** Devuelve la unión de las dos listas recibidas como parámetro.
- **añadir_parametros(listPar,listaId,string,string)**: Añade a la lista de parámetros cada identificador de la lista junto con su tipo y su clase.

5 ETDS

programa → **programa id** { añadir_inst(prog ‖ id.*nombre*); }
declaraciones
decl_de_subprogs
comienzo
lista_de_sentencias
fin ; { añadir_inst(halt ‖ ; }

declaraciones → **variables** lista_de_ident : tipo ;
{ añadir_declaraciones(lista_de_ident.*listaId* , tipo.*tipoVar*); } declaraciones
| ξ

lista_de_ident → **id** resto_lista_id
{ lista_de_ident.*listaId* := unir(resto_lista_id .*listaId* , tipo.*tipoVar*); }

resto_lista_id → , **id** resto_lista_id
{ resto_lista_id .*listaId* := unir(resto_lista_id .*listaId* , tipo.*tipoVar*); }
| ξ { resto_lista_id .*listaId* := lista_vacia(); }

tipo → **entero** { tipo.*tipoVar* := "entero"; }
| **real** { tipo.*tipoVar* := "real"; }

decl_de_subprogs → decl_de_subprograma decl_de_subprogs
| ξ

decl_de_subprograma → **procedimiento id** { añadir_inst(procedure ‖ id.*nombre*); }
argumentos declaraciones **comienzo** lista_de_sentencias **fin** ; { añadir_inst(endproc ‖ ;) }

argumentos → (lista_de_param)
| ξ

lista_de_param → lista_de_ident : clase_par tipo
{ añadir_parametros(lista_de_ident.*listaId*, clase_par.*clasePar*, tipo.*tipoVar*); } resto_lis_de_param

clase_par → **in** { clase_par.*clasePar* := "in"; }
|**out** { clase_par.*clasePar* := "out"; }
| **in out** { clase_par.*clasePar* := "in out"; }

resto_lis_de_param → ; lista_de_ident : clase_par tipo
{ añadir_parametros(lista_de_ident.*listaId*, clase_par.*clasePar*, tipo.*tipoVar*); } resto_lis_de_param
| ξ

lista_de_sentencias → sentencia lista_de_sentencias
{ lista_de_sentencias.*exit* := unir(sentencia.*exit* , lista_de_sentencias.*exit*;);}
| ξ
{ lista_de_sentencias.*exit* := lista_vacia();); }

sentencia → variable = expresion ;
{
añadir_inst(variable.*nombre* || := || expresion.*nombre* || ;);
sentencia.*exit* := lista_vacia();
}
| **si** expresion **entonces** M lista_de_sentencias M **fin** ;
{
completar(expresion.*true*, M1.*ref*);
completar(expresion.*false*, M2.*ref*);
sentencia.*exit* := lista_de_sentencias.*exit*;
}
| **hacer** M lista_de_sentencias **hasta** expresion M **fin** ;
{
completar(expresion.*true*, M1.*ref*);
completar(expresion.*false*, M2.*ref*);
completar(lista_de_sentencias.*exit*, M2.*ref*);
sentencia.*exit* := lista_vacia();
}
| **romper si** expresion M ;
{
completar(expresion.*false*, M.*ref*);
sentencia.*exit* := expresion.*true*;
}
| **leer** (variable) ;
{
añadir_inst(read || variable.*nombre* || ;);
sentencia.*exit* := lista_vacia();
}
| **escribir_linea** (expresion) ;
{
añadir_inst(writeln || expresion.*nombre* || ;);
sentencia.*exit* := lista_vacia();
}

variable → **id** { variable.*nombre* := id.*nombre*; }

expresion → expresion == expresion
{
expresion.*true* := inilista(obtenref());
expresion.*false* := inilista(obtenref() + 1);
añadir_inst(if || expresion$_1$.*nombre*|| == || expresion$_2$.*nombre*|| goto);
añadir_inst(goto);
}
| expresion > expresion
{
expresion.*true* := inilista(obtenref());
expresion.*false* := inilista(obtenref() + 1);
añadir_inst(if || expresion$_1$.*nombre*|| > || expresion$_2$.*nombre*|| goto);
añadir_inst(goto);
}
| expresion < expresion
{

expresion.*true* := inilista(obtenref());
expresion.*false* := inilista(obtenref() + 1);
añadir_inst(if ‖ expresion$_1$.*nombre*‖ < ‖ expresion$_2$.*nombre*‖ goto);
añadir_inst(goto);
}
| expresion ≥ expresion
{
expresion.*true* := inilista(obtenref());
expresion.*false* := inilista(obtenref() + 1);
añadir_inst(if ‖ expresion$_1$.*nombre*‖ ≥ ‖ expresion$_2$.*nombre*‖ goto);
añadir_inst(goto);
}
| expresion ≤ expresion
{
expresion.*true* := inilista(obtenref());
expresion.*false* := inilista(obtenref() + 1);
añadir_inst(if ‖ expresion$_1$.*nombre*‖ ≤ ‖ expresion$_2$.*nombre*‖ goto);
añadir_inst(goto);
}
| expresion ≠ expresion
{
expresion.*true* := inilista(obtenref());
expresion.*false* := inilista(obtenref() + 1);
añadir_inst(if‖ expresion$_1$.*nombre*‖ ≠ ‖ expresion$_2$.*nombre*‖ goto);
añadir_inst(goto);
}
| expresion + expresion
{
expresion := nuevo_id();
añadir_inst(expresion.*nombre* ‖ := ‖ expresion$_1$.*nombre*‖ + ‖ expresion$_2$.*nombre*);
}
| expresion − expresion
{
expresion := nuevo_id();
añadir_inst(expresion.*nombre* ‖ := ‖ expresion$_1$.*nombre*‖ − ‖ expresion$_2$.*nombre*);
}
| expresion ∗ expresion
{
expresion := nuevo_id();
añadir_inst(expresion.*nombre* ‖ := ‖ expresion$_1$.*nombre*‖ ∗ ‖ expresion$_2$.*nombre*);
}
| expresion / expresion
{
expresion := nuevo_id();
añadir_inst(expresion.*nombre* ‖ := ‖ expresion$_1$.*nombre*‖ / ‖ expresion$_2$.*nombre*);
}
| **id** { expresion.*nombre* := id.*nombre*; }
| **num_entero** { expresion.*nombre* := num_entero.*nombre*; }
| **num_real** { expresion.*nombre* := num_real.*nombre*; }

M → ξ
{ M.ref := obtenref(); }

6 ETDS con Booleanos

programa → **programa id** { añadir_inst(prog ‖ id.*nombre*); }
declaraciones
decl_de_subprogs
comienzo
lista_de_sentencias
fin ; { añadir_inst(halt ‖ ; }

declaraciones → **variables** lista_de_ident : tipo ;
{ añadir_declaraciones(lista_de_ident.*listaId* , tipo.*tipoVar*); } declaraciones
| ξ

lista_de_ident → **id** resto_lista_id
{ lista_de_ident.*listaId* := unir(resto_lista_id .*listaId* , tipo.*tipoVar*); }

resto_lista_id → , **id** resto_lista_id
{ resto_lista_id .*listaId* := unir(resto_lista_id .*listaId* , tipo.*tipoVar*); }
| ξ { resto_lista_id .*listaId* := lista_vacia(); }

tipo → **entero** { tipo.*tipoVar* := "entero"; }
| **real** { tipo.*tipoVar* := "real"; }

decl_de_subprogs → decl_de_subprograma decl_de_subprogs
| ξ

decl_de_subprograma → **procedimiento id** { añadir_inst(procedure ‖ id.*nombre*); }
argumentos declaraciones **comienzo** lista_de_sentencias **fin** ; { añadir_inst(endproc ‖ ;) }

argumentos → (lista_de_param)
| ξ

lista_de_param → lista_de_ident : clase_par tipo
{ añadir_parametros(lista_de_ident.*listaId*, clase_par.*clasePar*, tipo.*tipoVar*); } resto_lis_de_param

clase_par → **in** { clase_par.*clasePar* := "in"; }
|**out** { clase_par.*clasePar* := "out"; }
| **in out** { clase_par.*clasePar* := "in out"; }

resto_lis_de_param → ; lista_de_ident : clase_par tipo
{ añadir_parametros(lista_de_ident.*listaId*, clase_par.*clasePar*, tipo.*tipoVar*); } resto_lis_de_param
| ξ

lista_de_sentencias → sentencia lista_de_sentencias
{ lista_de_sentencias.*exit* := unir(sentencia.*exit* , lista_de_sentencias.*exit*;);}
| ξ
{ lista_de_sentencias.*exit* := lista_vacia();); }

sentencia → variable = expresion ;
{
añadir_inst(variable.*nombre* || := || expresion.*nombre* || ;);
sentencia.*exit* := lista_vacia();
}
| **si** expresion **entonces** M lista_de_sentencias M **fin** ;
{
completar(expresion.*true*, M1.*ref*);
completar(expresion.*false*, M2.*ref*);
sentencia.*exit* := lista_de_sentencias.*exit*;
}
| **hacer** M lista_de_sentencias **hasta** expresion M **fin** ;
{
completar(expresion.*true*, M1.*ref*);
completar(expresion.*false*, M2.*ref*);
completar(lista_de_sentencias.*exit*, M2.*ref*);
sentencia.*exit* := lista_vacia();
}
| **romper si** expresion M ;
{
completar(expresion.*false*, M.*ref*);
sentencia.*exit* := expresion.*true*;
}
| **leer** (variable) ;
{
añadir_inst(read || variable.*nombre* || ;);
sentencia.*exit* := lista_vacia();
}
| **escribir_linea** (expresion) ;
{
añadir_inst(writeln || expresion.*nombre* || ;);
sentencia.*exit* := lista_vacia();
}

variable → **id**
{ variable.*nombre* := id.*nombre*; }

expresion → expresion == expresion
{
expresion.*true* := inilista(obtenref());
expresion.*false* := inilista(obtenref() + 1);
añadir_inst(if || expresion_1.*nombre*|| == || expresion_2.*nombre*|| goto);
añadir_inst(goto);
}
| expresion > expresion
{
expresion.*true* := inilista(obtenref());
expresion.*false* := inilista(obtenref() + 1);
añadir_inst(if || expresion_1.*nombre*|| > || expresion_2.*nombre*|| goto);
añadir_inst(goto);
}
| expresion < expresion

{
expresion.*true* := inilista(obtenref());
expresion.*false* := inilista(obtenref() + 1);
añadir_inst(if ‖ expresion$_1$.*nombre*‖ < ‖ expresion$_2$.*nombre*‖ goto);
añadir_inst(goto);
}
| expresion ≥ expresion
{
expresion.*true* := inilista(obtenref());
expresion.*false* := inilista(obtenref() + 1);
añadir_inst(if ‖ expresion$_1$.*nombre*‖ ≥ ‖ expresion$_2$.*nombre*‖ goto);
añadir_inst(goto);
}
| expresion ≤ expresion
{
expresion.*true* := inilista(obtenref());
expresion.*false* := inilista(obtenref() + 1);
añadir_inst(if ‖ expresion$_1$.*nombre*‖ ≤ ‖ expresion$_2$.*nombre*‖ goto);
añadir_inst(goto);
}
| expresion ≠ expresion
{
expresion.*true* := inilista(obtenref());
expresion.*false* := inilista(obtenref() + 1);
añadir_inst(if‖ expresion$_1$.*nombre*‖ ≠ ‖ expresion$_2$.*nombre*‖ goto);
añadir_inst(goto);
}
| expresion + expresion
{
expresion := nuevo_id();
añadir_inst(expresion.*nombre* ‖ := ‖ expresion$_1$.*nombre*‖ + ‖ expresion$_2$.*nombre*);
}
| expresion − expresion
{
expresion := nuevo_id();
añadir_inst(expresion.*nombre* ‖ := ‖ expresion$_1$.*nombre*‖ − ‖ expresion$_2$.*nombre*);
}
| expresion ∗ expresion
{
expresion := nuevo_id();
añadir_inst(expresion.*nombre* ‖ := ‖ expresion$_1$.*nombre*‖ ∗ ‖ expresion$_2$.*nombre*);
}
| expresion / expresion
{
expresion := nuevo_id();
añadir_inst(expresion.*nombre* ‖ := ‖ expresion$_1$.*nombre*‖ / ‖ expresion$_2$.*nombre*);
}
| expresion && expresion
{
completar(expresion$_1$.*true*,M.*ref*)
expresion.*true* := expresion$_2$.*true*
expresion.*false* := unir(expresion$_1$.*false*,expresion$_2$.*false*)

}
| expresion || expresion
{
completar(expresion$_1$.*false*,M.*ref*)
expresion.*true* := unir(expresion$_1$.*true*,expresion$_2$.*true*)
expresion.*false* := expresion$_2$.*false*
}
| expresion **!** expresion
{
expresion.*true* := expresion$_1$.*false*
expresion.*false* := expresion$_1$.*true*
}
| **id** { expresion.*nombre* := id.*nombre*; }
| **num_entero** { expresion.*nombre* := num_entero.*nombre*; }
| **num_real** { expresion.*nombre* := num_real.*nombre*; }
| (expresion) {
expresion.*true* := expresion$_1$.*true*;
expresion.*false* := expresion$_1$.*false*;
}

M $\rightarrow \xi$
{ M.ref := obtenref(); }

7 Ejemplo de ejecución

```
(* En esta prueba se comprueban las declaraciones de subprogramas *)

programa ejemplo_con_nombre_muy_largo
variables a, b, c : entero;
variables d, e : real;

procedimiento suma(x, y : in entero; resultado : in out entero; otro : out real)
   variables aux : entero;
   comienzo
      hacer
         aux = a; c = b;
 aux = aux-1;
 c = c+1 ;
      hasta aux /= 0 fin ;
   fin;

comienzo
   leer(a); leer(b);
   d = 1/b;
   e = 0.1e-1/a;
   (* suma(a,b,c); esto solo para aquellos que
                   traten llamadas a procedimientos *)
   c = c*(c*d)+e;
   escribir_linea(c*c);
fin ;
```

```
./parser < ../Pruebas/PruebaBuena2.dat
ha comenzado...

1: prog ejemplo_con_nombre_muy_largo;
2: int a;
3: int b;
4: int c;
5: real d;
6: real e;
7: proc suma;
8: val_int x;
9: val_int y;
10: ref_int resultado;
11: ref_real otro;
12: int aux;
13: aux=a;
14: c=b;
15: _t1:=aux-1;
16: aux=_t1;
17: _t2:=c+1;
18: c=_t2;
19: if aux/=0 goto 13;
20: goto 21;
21: endproc;
22: read a;
23: read b;
24: _t3:=1/b;
25: d=_t3;
26: _t4:=0.1e-1/a;
27: e=_t4;
28: _t5:=c*d;
29: _t6:=c*_t5;
30: _t7:=_t6+e;
31: c=_t7;
32: _t8:=c*c;
33: write _t8;
34: writeln;
35: halt;
ha finalizado BIEN...
```

8 Seguimiento del trabajo

En muchos casos las actividades se han realizado en pareja por lo que esas horas en realidad computan el doble para la dedicación de tiempo al proyecto, aunque esto puede parecer poco eficiente ya que se podría realizar el doble de trabajo en el mismo tiempo si se hace por separado, es conveniente por otra parte contar con un compañero para comprender mejor el trabajo.

Tarea	Mikel	Diego	Total
Especifiación Léxica	30'	30'	1h
Autómata	1h	30'	1h 30'
programar tokens.l	1h 30'	1h 30'	3h
programar sintaxis	2h 30'	2h	4h 30'
Definir Atributos	0	1h 30'	1h 30'
Definir Abstracciones Funcionales	45'	1h	1h 45'
Definir ETDS	2h 30'	2h 15'	4h 45'
Programar ETDS	3h	1h	4h
Pruebas ETDS	0	1h	1h
Editar Documentación	3h	1h	4h
Total tras primera entrega	14h 45'	12h 15'	27h
Corregir ETDS	1h	1h	2h
Programar ETDS	5h 30'	4h	9h 30'
Total tras segunda entrega	21h 15'	17h 15'	38h 30'
Definir ETDS Booleanos	1h	1h	2h
Programar Booleanos	1h	1h	2h
Editar Documentación	3h	3h	6h
Total tras tercera entrega	25h 15'	21h 15'	46h 30'

9 Autoevaluación

Consideramos que hemos realizado una práctica correcta, bien desarrollada y bien documentada. Aunque no hemos podido alcanzar los objetivos adicionales que hubieramos querido creemos que hemos trabajado bien y estamos contentos con el resultado y sobre todo con lo aprendido. Valoramos nuestra páctica como un 8 sobre 10.

Bibliografía

- Alfred V.Aho, Ravi Sethi y Jeffrey D.Ullman, *Compiladores, principios técnicas y herramientas.*

10 Anexo

En las siguientes páginas adjuntamos el código más relevante de la práctica.

10.1 tokens.l

```
%option yylineno

%{
 #include <string>
 #include <iostream>
 #include <vector>
 #include <algorithm>
 using namespace std ;
 #include "Aux.hpp"
 #include "parser.hpp"

 #define TOKEN(t) yylval.str = new std::string(yytext, yyleng) ; return t

 extern "C" int yywrap() { return(1) ; }

%}

%%
programa                  TOKEN(RPROGRAM) ;
comienzo                  TOKEN(RBEGIN) ;
procedimiento           TOKEN(RPROCEDURE);
fin                       TOKEN(REND) ;

variables       TOKEN(RVAR);
entero          TOKEN(RINTEGER);
real            TOKEN(RFLOAT);

in            TOKEN(RIN);
out           TOKEN(ROUT);

si            TOKEN(RIF);
entonces          TOKEN(RTHEN);
romper          TOKEN(RBREAK);

hacer         TOKEN(RDO);
hasta         TOKEN(RUNTIL);

escribir_linea      TOKEN(RPUT_LINE);
leer            TOKEN(RREAD);

"=="       TOKEN(TCEQ);
">="       TOKEN(TCGE);
"<="       TOKEN(TCLE);
"/="       TOKEN(TCNE);
"<"      TOKEN(TCLT);
">"      TOKEN(TCGT);

"+"      TOKEN(TPLUS);
"-"      TOKEN(TMINUS);
"*"          TOKEN(TMUL);
"/"      TOKEN(TDIV);

"&&"       TOKEN(TAND);
"||"       TOKEN(TOR);
"!"      TOKEN(TNOT);

"="          TOKEN(TASSIG);

";"          TOKEN(TSEMIC);
","      TOKEN(TCOMMA);
":"      TOKEN(TCOLON);

"("      TOKEN(TLPAREN);
")"      TOKEN(TRPAREN);

\(\*([^*]|\*+[^*)])*\*+\)                       ;

[ \t\n]                                   ;
```

```
[a-zA-Z](_?[a-zA-Z0-9])*            TOKEN(TIDENTIFIER)  ;

[0-9]+\.[0-9]+([eE][\+\-]?[0-9]+)?    TOKEN(TFLOAT);
[0-9]+                              TOKEN(TINTEGER);

.                       { cout << "Token desconocido: " << yytext << endl; yyterminate();}
%%
```

10.2 parser.y

```
%error-verbose

%{
   #include <stdio.h>
   #include <iostream>
   #include <vector>
   #include <string>
   using namespace std;

   extern int yylex();
   extern int yylineno;
   extern char *yytext;
   void yyerror (const char *msg) {
     printf("line %d: %s at '%s'\n", yylineno, msg, yytext) ;
   }

   #include "Codigo.hpp"
   #include "Aux.hpp"


   expresionstruct makecomparison(std::string &s1, std::string &s2, std::string &s3) ;
   expresionstruct makearithmetic(std::string &s1, std::string &s2, std::string &s3) ;

   Codigo codigo;
%}

%union {
    string *str;
    vector<string> *list ;
    expresionstruct *expr ;
    int number ;
    vector<int> *vectorInt ;
}

%token <str> TIDENTIFIER
%token <str> TINTEGER TFLOAT
%token <str> TMUL TPLUS TMINUS TDIV
%token <str> TCGE TCLE TCLT TCGT TCNE TCEQ
%token <str> TAND TOR TNOT
%token <str> TASSIG
%token <str> TSEMIC TCOMMA TCOLON TLPAREN TRPAREN
%token <str> RPROGRAM RPROCEDURE RBEGIN REND
%token <str> RVAR RINTEGER RFLOAT
%token <str> RIN ROUT RREAD RWRITE RPUT_LINE
%token <str> RIF RTHEN RDO RUNTIL RBREAK

%type <number> M
%type <str> programa declaraciones clase_par tipo variable argumentos decl_de_subprogs decl_de_subprograma
%type <expr> expresion
%type <list> lista_de_ident resto_lista_id lista_de_param resto_lis_de_param
%type <vectorInt> lista_de_sentencias sentencia


%left TOR
%left TAND
%left TNOT
%nonassoc TCGE TCLE TCLT TCGT TCNE TCEQ
%left TMINUS TPLUS
%left TMUL TDIV

%start programa

%%

programa : RPROGRAM
     TIDENTIFIER  { codigo.anadirInstruccion("prog " + *\$2 + ";" ) ;}
```

```
     declaraciones
     decl_de_subprogs
     RBEGIN
     lista_de_sentencias
    {
      if(!\$7->empty())
        yyerror("Sentencia break fuera de estructura, hacer-hasta");
        exit(0);
    }
     REND TSEMIC { codigo.anadirInstruccion("halt;");
          codigo.escribir();}
;

declaraciones : RVAR lista_de_ident TCOLON tipo TSEMIC
    {codigo.anadirDeclaraciones(*\$2, *\$4); delete \$2; delete \$4 ;}  declaraciones
    |{}
;

lista_de_ident : TIDENTIFIER resto_lista_id
    {
      \$\$ = \$2 ;
           \$\$->insert(\$\$->begin(), *\$1);
    }
;

resto_lista_id : TCOMMA TIDENTIFIER resto_lista_id
     {
      \$\$ = \$3;
      \$\$->insert(\$\$->begin(), *\$2);
     }
    | { \$\$ = new vector<string>; }
;

tipo : RINTEGER { \$\$ = new string("int"); }
  | RFLOAT { \$\$ = new string("real"); }
;


decl_de_subprogs : decl_de_subprograma decl_de_subprogs {}
    |{}
;

decl_de_subprograma : RPROCEDURE TIDENTIFIER  {  codigo.anadirInstruccion("proc " + *\$2 + ";");}
      argumentos declaraciones
      RBEGIN lista_de_sentencias REND TSEMIC { codigo.anadirInstruccion("endproc;");}
;

argumentos : TLPAREN lista_de_param TRPAREN { }
    | { }
;

lista_de_param : lista_de_ident TCOLON clase_par tipo  { codigo.anadirParametros(*\$1,*\$3,*\$4); } resto_lis_de_par
;

clase_par : RIN { \$\$ = new string("in"); }
  | ROUT { \$\$ = new string("out"); }
  | RIN ROUT { \$\$ = new string("in out"); }
;

resto_lis_de_param : TSEMIC lista_de_ident TCOLON clase_par tipo {codigo.anadirParametros(*\$2,*\$4,*\$5);} resto_li
    |{}
;

lista_de_sentencias : sentencia lista_de_sentencias {\$\$ = codigo.unir(*\$1, *\$2);}
      | { \$\$ = new vector<int>; }
;

sentencia :  variable TASSIG expresion TSEMIC
  {
      codigo.anadirInstruccion(*\$1 + *\$2 + \$3->str + ";") ;
    delete \$1 ; delete \$3;
      \$\$ = new vector<int>;
  }

  | RIF expresion RTHEN M lista_de_sentencias M REND TSEMIC
  {
    codigo.completarInstrucciones(\$2->trues, \$4);
    codigo.completarInstrucciones(\$2->falses, \$6);
```

```
    \$\$ = \$5;
  }

  | RDO M lista_de_sentencias RUNTIL expresion M REND TSEMIC
  {
    codigo.completarInstrucciones(\$5->trues, \$2);
    codigo.completarInstrucciones(\$5->falses, \$6);
    codigo.completarInstrucciones(*\$3, \$6);
    \$\$ = new vector<int>;
  }

  | RBREAK RIF expresion M TSEMIC
  {
    codigo.completarInstrucciones(\$3->falses, \$4) ;
    \$\$ = new vector<int>;
      \$\$->assign(\$3->trues.begin(),\$3->trues.end());
  }

  | RREAD TLPAREN variable TRPAREN TSEMIC
  {
    codigo.anadirInstruccion("read " + *\$3 + ";") ;
    \$\$ = new vector<int>;
  }

  | RPUT_LINE TLPAREN expresion TRPAREN TSEMIC
  {
    codigo.anadirInstruccion("write " + \$3->str + ";") ;
    codigo.anadirInstruccion("writeln;") ;
    \$\$ = new vector<int>;
  }
;

variable : TIDENTIFIER { \$\$ = \$1 ; }
;

expresion :expresion TCEQ expresion{ \$\$ = new expresionstruct;
           *\$\$ = makecomparison(\$1->str,*\$2,\$3->str) ;
          delete \$1; delete \$3; }
   | expresion TCGT expresion{ \$\$ = new expresionstruct;
           *\$\$ = makecomparison(\$1->str,*\$2,\$3->str) ;
          delete \$1; delete \$3; }
   | expresion TCLT expresion{ \$\$ = new expresionstruct;
           *\$\$ = makecomparison(\$1->str,*\$2,\$3->str) ;
          delete \$1; delete \$3; }
   | expresion TCGE expresion{ \$\$ = new expresionstruct;
           *\$\$ = makecomparison(\$1->str,*\$2,\$3->str) ;
          delete \$1; delete \$3; }
   | expresion TCLE expresion{ \$\$ = new expresionstruct;
           *\$\$ = makecomparison(\$1->str,*\$2,\$3->str) ;
          delete \$1; delete \$3; }
   | expresion TCNE expresion{ \$\$ = new expresionstruct;
           *\$\$ = makecomparison(\$1->str,*\$2,\$3->str) ;
          delete \$1; delete \$3; }
   | expresion TPLUS expresion{ \$\$ = new expresionstruct;
           *\$\$ = makearithmetic(\$1->str,*\$2,\$3->str) ;
          delete \$1; delete \$3; }
   | expresion TMINUS expresion{ \$\$ = new expresionstruct;
           *\$\$ = makearithmetic(\$1->str,*\$2,\$3->str) ;
          delete \$1; delete \$3; }
   | expresion TMUL expresion{ \$\$ = new expresionstruct;
           *\$\$ = makearithmetic(\$1->str,*\$2,\$3->str) ;
          delete \$1; delete \$3; }
   | expresion TDIV expresion{ \$\$ = new expresionstruct;
           *\$\$ = makearithmetic(\$1->str,*\$2,\$3->str) ;
          delete \$1; delete \$3; }
   | expresion TAND M expresion
   {
    \$\$ = new expresionstruct;
    codigo.completarInstrucciones(\$1->trues, \$3);
    \$\$->trues = \$4->trues;
    \$\$->falses = *codigo.unir(\$1->falses, \$4->falses);
   }
   | expresion TOR M expresion
   {
    \$\$ = new expresionstruct;
    codigo.completarInstrucciones(\$1->falses, \$3);
    \$\$->trues = *codigo.unir(\$1->trues, \$4->trues);
    \$\$->falses = \$4->falses;
```

```
   }
   | TNOT expresion
   {
    \$\$ = new expresionstruct;
    \$\$->falses = \$2->trues;
    \$\$->trues = \$2->falses;
   }
   | TIDENTIFIER  { \$\$ = new expresionstruct; \$\$->str = *\$1; }
       | TINTEGER { \$\$ = new expresionstruct; \$\$->str = *\$1; }
       | TFLOAT { \$\$ = new expresionstruct; \$\$->str = *\$1; }
   | TLPAREN expresion TRPAREN { \$\$ = \$2; }

;

M : { \$\$ = codigo.obtenRef(); }
;

%%

expresionstruct makecomparison(std::string &s1, std::string &s2, std::string &s3) {
  expresionstruct tmp ;
  tmp.trues.push_back(codigo.obtenRef()) ;
  tmp.falses.push_back(codigo.obtenRef()+1) ;
  codigo.anadirInstruccion("if " + s1 + s2 + s3 + " goto") ;
  codigo.anadirInstruccion("goto") ;
  return tmp ;
}

expresionstruct makearithmetic(std::string &s1, std::string &s2, std::string &s3) {
  expresionstruct tmp ;
  tmp.str = codigo.nuevoId() ;
  codigo.anadirInstruccion(tmp.str + ":=" + s1 + s2 + s3 + ";") ;
  return tmp ;
}
```