# PRACTICA INDIVIDUAL JON PEREZ ETXEBARRIA:

## IMPLEMENTACÓN DE UN ETDS CON FLEX Y BISON

JPEREZ178@IKASLE.EHU.EUS
JONPEREZETXEBARRIA@GMAIL.COM

# Introduccion:

Con la implementación de esta práctica he intentado desarrollar un traductor principalmente sintáctico, pero con unas pequeñas comprobaciones semánticas de lo que vendría a ser un lenguaje de programación. Nuestro traductor es capaz de no solo analizar si la escritura y formato de la misma es la adecuada sino también de escribir por pantalla una traducción a código intermedio.

Entrando algo mas en detalles, hemos usado Flex y Bison para esta implementación, además de C++. Hemos Usado distintos tipos de documentos, cuyo uso explicare ahora, debido a que algunos documentos son dependientes o van estrictamente ligados a otros los mencionaremos en el apartado del documento principal.

**Tokens.l:** Este documento, junto a su versión .cpp, es donde quedan definidos todos los tokens que vamos a utilizar en mi lenguaje de programación. Estos tokens toman formas de expresión regular o de una palabra predefinida, se les dará un nombre como token y el .cpp les asignara un numero que se usara para la traducción mas adelante.

**Aux.hpp:** Los struct de C++ que se usan en en el parser.y están definidos aquí.

**Codigo.cpp:** en este documento y respectivo .hpp están definidos la grandísima mayoría de los métodos que usaremos para que la implementación no repita las mismas estructuras de codigo una y otra vez.

**Main.cpp:** Es el main, que en nuestro caso es uno muy sencillito que inicia la secuencia de traducción y hace varios prints para conocer el estado de la ejecución.

**Makefile:** llamaremos al make de este fichero que se encargara de indicar donde están los tokens y donde debe encontrar las estructuras de traducción que se hayan en el parser además de pasarle las distintas pruebas para que analice.

**Parse.y:** el parser.y y sus dependientes son los archivos donde esta definido el ETDS a seguir para la traducción adecuada del lenguaje además del algún método para faciulitar la implementación e instrucciones para escribir en pantalla el codigo intermedio apropiado.

# Autoevaluacion:

Como en toda autoevaluación es difícil mantener un enfoque objetivo ya que desde la perspectiva personal siempre tienes acceso a todas las escusas para los problemas tenidos y ves los obstáculos superados desde otra perspectiva y aun intentando compensar para ello puedes acabar en el otro lado de la parcialidad, por no mencionar que al no conocer exactamente cuales son los criterios de evaluación se dificulta aun mas la tarea. Una vez dicho esto creo que si puntuamos esta practica del 1 al 10 creo que es una practica que acabaría entre el 7 y el 8. Creo que es un punto intermedio dado el hecho que el 5 se obtenía con unas pocas estructuras básicas y en esta han sido implementadas alguna estructura de control extra un nuevo tipo de variable como el array e incluso un par de tipos de comprobaciones semánticas. Aun que incluso en la implementación de las estructuras se ha intentado que sean lo mas cercanas a como yo lo haría lo mas posible podría entender perfectamente que ese tipo de traducción no sea optima e incluso que para obtener la mayoría de los puntos hagan falta definir mas cosas dentro de los respectivos campos. Es difícil comentar mas haya ya que no se que forma tiene una practica perfecta y autoevaluar la mia sin los criterios de evaluaciones muy complicado.

# ETDS Practica:

## 1.Gramatica, tabla de tokens

| Tipo de Token | Patrón | Descripción | Lexemas |
|---|---|---|---|
| Operador | * | Carácter “*” | * |
| Operador | + | Carácter “+” | + |
| Operador | - | Carácter “-“ | - |
| Operador | / | Carácter “/” | / |
| Comparador | >= | Caracteres “>” y “=” concatenados | >= |
| Comparador | <= | Caracteres “<” y “=” concatenados | <= |
| Comparador | > | Carácter “>” | > |
| Comparador | < | Carácter “<” | < |
| Comparador | == | Los caracteres =,= | |

| | | | |
|---|---|---|---|
| | | concatenados | |
| Comparador | = | Carácter “=” | = |
| Distinto | /= | Caracteres ”/” y “=” concatenados | /= |
| Separador | ; | Carácter “;” | ; |
| Declaración de variable | Var | Las letras v,a,r concatendas | var |
| Asignación | : | Carácter “:” | : |
| Declaración de subprograma | procedure | Las letras p,r,o,c,e,d,u,r,e concatenadas | procedure |
| Tipo de parámetro | in | Las letras i,n concatenadas | in |
| Tipo de parámetro | out | Las letras o,u,t concatenadas | out |
| Tipo de parámetro | in out | Las letras i,n,o,u,t concatenadas | in out |
| Agrupador | ( | Carácter “(“ | ( |
| Agrupador | ) | Carácter “)” | ) |
| Agrupador | { | Carácter “{“ | { |
| Agrupador | } | Carácter "}" | } |
| Separador | , | Carácter "," | , |
| Condicional | if | la letra i seguida de f | if |
| Instruccion | then | las letras t,h,e,n concatenadas en ese orden. | then |
| Instruccion | while | las letras w,h,i,l,e concatenadas en ese orden. | while |
| Instruccion | do | las letras d,o concatenadas en ese orden. | do |
| Instruccion | until | las letras u,n,t,i,l concatenadas en ese | until |

| | | | |
|---|---|---|---|
| | | orden. | |
| Condicional | else | las letras e,l,s,e concatenadas en ese orden. | else |
| instrucción | skip | las letras s,k,i,p concatenadas en ese orden. | skip |
| instrucción | read | las letras r,e,a,d concatenadas en ese orden. | read |
| tipo de variable | integer | las letras i,n,t,e,g,e,r concatenadas en ese orden. | integer |
| tipo de variable | float | las letras f,l,o,a,t concatenadas en ese orden. | float |
| instrucción | println | las letras p,r,i,n,t,l,n concatenadas en ese orden. | println |
| Identificador | [a-zA-Z](_?[a-zA-Z0-9])* | una letra seguida o no de una barra baja seguida de una variable alfanumerica la cantidad de veces que se quiera | a_4A |
| real | [0-9]+\.[0-9]+([Ee][\+\-]?[0-9]+)? | un numero positivo por lo menos seguido de un punto seguido a su vez por otro numero y puede estar seguido o no de "Ee"+ o - y otro numero | 9.7 |
| entero | [0-9]+ | Una cantida de numeros entre 0 y 9 mayor que 1 | 1 |
| Comentario | \/*((*?[^/])\|([^*]\/))**\/ | empiezan por /* y acaban en */ .En medio puede haber cualquier secuencia de caracteres | /*h */ |

| | | | |
|---|---|---|---|
| | | excepto */ | |
| comentario de linea | \/\/.+\n | comentario de linea empieza por // seguido de cualquier cantidad de . | //a |
| tabulacion | [ \t\n] | tabulacion | a a |
| | | | |
| Tipo de variable | array | Las letras a,r,r,a,y concatenadas | array |
| Intruccion | endfor | Las letras e,n,d,f,o,r concatenadas | endfor |
| Instrucción | to | Las letras t,o concatenadas | to |
| Instrucción | for | Las f,o,r concatenadas | for |
| Intruccion | from | Las letras f,r,o,m concatenadas | from |
| Booleano | not | Las letras n,o,t concatenadas | not |
| Booleano | and | Las letras a,n,d concatenadas | and |
| Booleano | or | Las letras o,r concatenadas | or |
| Instruccion | endprogram | Las letras e,n,d,p,r,o,g,r,a,m concatenadas | endprogram |
| Instrucción | begin | Las letras b,e,g,i,n concatenadas | begin |
| Declaración de programa | program | Las letras p,r,o,g,r,a,m concatenadas | program |
| salto de linea | \r\n | salto de linea | a<br>a |

## 2.Tabla de Atributos definidos

M->ref : Referencia para conocer a que parte del código queremos ir en caso de que haya un salto en el mismo.

Expr-> expr: un tipo de estructura que contiene un string y dos vectores para almacenar referencia, E.true, E, false.

Tipo-> tipo: almacenar para analizar mas tarde.

Var-> varStruct: Un tipo de estructura que almacena con un string para almacenar el nombre de la variable y un vector donde en caso de ser una variable array almacenara los identificadores. Sus respectivos nombres son str y listaString.

Clase_Par->clase: contiene un string para poder saber que tipo de parámetro es.

Expr-> expr: un tipo de estructura que contiene un string y dos vectores para almacenar referencias y palabras clave. Sus respectivos nombres son str, trues y falses.

Lista de identificadores-> listaString: una lista para tener todos los identificadores guardados.

Resto de lista de identificadores-> listaString: una lista para tener todos los identificadores guardados

Sentencia-> listaNumeros: Vector de integer para almacenar referencias a las identificadores.

Lista de sentencias-> listaNumeros: Vector de integer para almacenar referencias a las identificadores.

## 3.Descripción de las abstracciones funcionales utilizadas en el ETDS

| | |
|---|---|
| Añadir-inst(string) ␣ código | Añade una línea de código especificando una instrucción. |
| completar(vector<int>,int) ␣ código | Completa las líneas de código en el vector con la referencia del numero entero |
| anadirDeclaraciones(var, tipo) -> codigo | Añade una línea de código en forma de declaración. |
| unir(vector<int>, vector<int>) -> parser.y | Une los dos vectores en uno y lo devuelve. |
| makearithmetic(string , string , string ) | Une 3 strings en unos solo |

| | |
|---|---|
| ->parse.y | creando un expressionstruct y asignandole a su string la union. |
| makecomparison(string &s1, string &s2, string &s3) ->parser.y | Une 3 strings en uno solo creando un expressionstruct y asignadole a su string su union |
| siguienteInstruccion() ->codigo | devuelve el tamaño de la lista de instrucciones mas 1. |
| nuevoId()->codigo | devuelve un string para usar como identificador. |
| escribir()->codigo | escribe las isntrucciones. |
| obtenRef()->codigo | devuelve el valor que siguienteInstruccion(). |

## 4.ETDS: Esquema de Traducción Dirigido por la Sintaxis

### SENTENCIA S:

**IF:**

S ␣if E then {M

{ añadirInstruccion( if E1 goto M1+2);

AñadirInstruccion(goto);

}

LS M};

{completar(M1.ref+1, M2,ref);

completar (E.true, $M_1$.ref);

completar (E.false, $M_2$.ref);

S.listaNumeros = LS;

}

**S** ␣ var = expr ;

{

codigo.anadirInstruccion(var, =, expr, ;)

S.listaNumeros = nueva lista();

}

**S** ␣ var{S.listaNumeros = nueva lista();

If(var.listaString.vacia()){

AñadirInstruccion(goto notAnArrayError);

S.listaNumeros = nueva lista();

}

}

**S** ␣ TFOR var TFROM expr TTO expr M{

If(!var.listaString.vacia()){

añadirInstruccion(goto notAnArrayError);

}

añadirInstruccion(var.str = expr1.str);

String t1 = new String;

añadirInstruccion( t1 = expr2.str);

añadirInstruccion(if var.str < t1 goto M1+4 _);

AñadirInstruccion(goto);

{completar(M1.ref+3, M2,ref+1);

}

TDO LS M{

añadirInstruccion(goto M1.red);

completar(M1.ref+3,M2.ref+1);

}TENDFOR

**WHILE:**

S ␣while M E{M LS M};

{completar (E.true, $M_2$.ref);

completar (E.false, $M_3$.ref + 1);

Añadir-inst (goto $M_1$.ref)

S.listaNumeros = nueva lista();

}

**DO UNTIL:**

**S** ⊔ do {M LS} until E else M{

AñadirInstruccion(if E1 goto M2+2);

AñadirInstruccion(goto m1.ref);

}

{ LS};

{completar (E.true, $M_2$.ref);

completar (E.false, $M_1$.ref);

S.listaNumeros = nueva lista();

S.listaNumeros = nueva lista();

}

**SKIP IF:**

**S** ⊔ skip if E M;

{ S.listaNumeros = nueva lista();

completar (E.false, M.ref);

S.listaNumeros = E.trues;

}

**READ**

**S** ⊔ read ( var );{

If(!var.listaString.vacia()){

añadirInstruccion(goto cantReadAnArrayError);

}

S.listaNumeros = nueva lista();

añadirInstruccion(read var.str);

}

**PRINT**

**S** ⊔ print ( expr );{

S.listaNumeros = nueva lista();

añadirInstruccion(write var.str);

}

**PROGRAM:**

**P** ▭ program identifier bloque;

{ anadirInstruccion("prog: " + identifer ) ;

{

codigo.anadirInstruccion("halt;");

codigo.escribir() ;

}

**BLOQUE:**

**B** ▭ declaraciones decl_sub_progs LS {}

**DECLARACIONES:**

**D** ▭ var lista_de_ident : tipo ; declaraciones

{

codigo.anadirDeclaraciones(lista_de_ident,tipo)

}

| ε

**LISTA_DE_IDENT:**

**LI** identifier resto_lista_id

{

LI.listaString = resto_lista_id;

Completar(identifier, LI.listaString);

}

**RESTO_LISTA_ID:**

**RI**->, identifier resto_lista_id

{

RI.listaString = resto_lista_id;

Completar(identifier, LI.listaString);

}

| ε{ RI.listaString = nueva Lista();

}

**TIPO:**

**T**->int{ T.tipo = int; }

| float{ T.tipo = float; }

**DECL_DE_SUBPROGS:**

**DSS**->decl_de_subprograma decl_de_subprogs | ε

{

}

**DECL_DE_SUBPROGRAMA:**

**DS**->procedure indentifier {

AñadirInstruccion( procedure identifier);

}

argumentos bloque ;

**ARGUMENTOS:**

**A**->( lista_de_param ) | ε

{

}

**LISTA_DE_PARAM:**

**LP**->ista_de_ident : clase_par tipo resto_lis_de_param

{ añadirInstruccion(lista_de_ident + clase_par + tipo);

}

**CLASE_PAR:**

**CP**-> in{ CP.clase = “val_ ” ; }

| out { CP.clase = “resul_ ” ; }

| inout{ CP.clase = “ref_ ” ; }

**RESTO_LIST_DE_PARAM:**

**RLP**-> ; lista_de_ident : clase_par tipo resto_lis_de_param

{ añadir instaruccion( clase_par tipo lista_de_ident);

}

**LISTA_DE_SENTENCIAS:**

**LS**-> sentencia lista_de_sentencias

{

unir(lista_de_sentencias, sentencia)

}

|ε{ LS.listaNumeros = nueva Lista(); }

**VAR:**

**V**-> identifier { V = new varStruct;

V.str = identifier;}

**V**-> array[] indentifier = [LS];{

V = new varStruct;

V.str = identifier;

V.list = LS;

AñadirInstruccion( array[] = [LS] );

}

**EXPR:**

**E**->identifier {}

|E or E{ completar (E1 .false, M.ref);

E.true := unir(E1 .true, E2 .true);

E.false := E2 .false

E{makearithmetic( E,or,E);

}

|E and E{ completar (E1 .true, M.ref);

E.true := E2 .true;

E.false := unir(E1 .false, E2 .false);

E{makearithmetic( E,and,E);

}

|not E{ E.true := E1 .false;

E.false := E1 .true;}

|integer{E.str= intger.tipo}

|float{E.str = float.tipo}

|E=E{makearithmetic( E,=,E)}

|E>=E{makecomparison(E,>=,E)}

|E<=E{makecomparison(E,<=,E)}

|E>E{makecomparison(E,>,E)}

|E<E{makecomparison(E,<,E)}

|E-E{makearithmetic(E,-,E)}

|E*E{makearithmetic(E,*,E)}

|E+E{makearithmetic(E,+,E)}

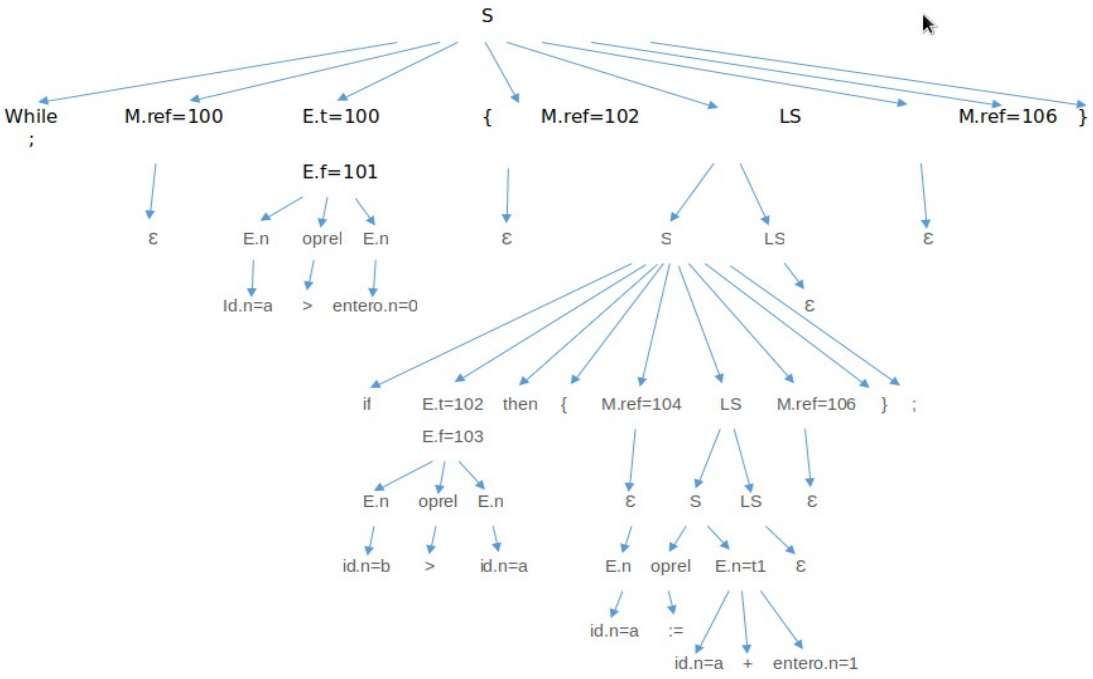
|E/=E{makearithmetic(E,/=,E)}

|E/E{makearithmetic(E,/,E)}

|(E){}

M: {M.ref = obtenRef();}

100. if a>0 goto 102




101. goto 107

102. if b>a goto 104

103. goto 106

104. t1:=a+1

105. a:= t1

106. goto 100

107. ___

# Mejoras Realizadas:

## 1.Comprobaciones semánticas implementadas

Se han implementado restricciones semánticas respecto a dos datos. La primera y mas sencilla es que cada vez que hay una división si el resultado de el divisor es 0 te llevara a un error de divisor 0.

Las otras restricciones tienen que ver con los arrays, como los arrays son un tipo de variable se comprueba que las variables usadas son las apropiadas. En sentencia se admite 'var' como sentencia, sin embargo, se comprueba que esa variable sea un array y por lo tanto la definición de un array se puede aceptar de manera independiente. Por otro lado, en sentencias como read o los bucles for se pide una variable, en estos casos haremos lo contrario, se comprobara que la variable que se pasa no sea un array.

## 2.Estructuras de control añadidas

La principal estructura extra añadida ha sido un bucle for que incluye la escritura de sentencias antes de terminar la definición de la estructura que después más adelante se completan. Escribe un código intermedio muy claro y fácilmente entendible que se vincula fácilmente con otro tipo de estructuras de control. Para la implementación de esta estructura se han usado nuevos tokens que han sido definidos en el archivo tokens.l y se han usado varias de las abstracción y métodos arriba definidos. Se puede comprobar en una de las pruebas lo fácilmente legible que es el codigo.

## 3.Nuevo tipo de variable

Se ha definido un nuevo tipo de variable siendo este un array para almacenar identificadores, se ha definido como var en el ETDS ya que facilitaba la implementación considerablemente a la hora de completar el resto del ETDS definido previamente, cabe la posibilidad de que si hubiera sido una estructura desde el inicio hubiera merecido la pena añadirlo de otra manera pero observando el estado del ETDS implementarlo como variable no solo era la manera mas sencilla de añadirlo a un ETDS ya existente si no que además me daba la posibilidad de hacer alguna comprobación semántica distintas sentencias y estructuras del árbol de traducción.

## 4.Booleanos

También se han añadido estructuras booleanas en como nueva estructura booleana y usadas dentro de “expr” después de ser analizadas quedaran guardas dentro del string de la estructura expresionstruct además de las correspondientes asignacionde las listas que corresponde.

## 5.Definida una nueva struct

Se ha definido tambien un nuevo tipo de struct, que contiene una lista y un string. Se le ha asignado a ‘var’ para poder ayudarme a implementar los arrays ademas usaremos su lista para varias comprobaciones semanticas asegurandonos asi de que la variable que pasamos tiene una lista vacia y que por tanto no es de tipo array.

Jon Perez Etxebarria