

# Elaboración de un Analizador Sintáctico con Bison

CARLOS AYALA T.<sup>1</sup>, DENNIS VEINTIMILLA A.<sup>1</sup>

1. Ingeniería en Sistemas Informáticos y de Computación, Facultad de Sistemas, Escuela Politécnica Nacional, Quito, Ecuador.

**Abstract**—The parser, which performs the analysis of input strings, is implemented through the Bison 'compiler-compilers' (compiler-compilers), in combination with the Flex rule setter. Work was done on the Ubuntu distribution of the Linux OS. In addition, a symbol table was used to check acceptable operations. By means of code will download the necessary programs before. Also with the commands of the terminal will be called to execute the files and the symbol table is called to check together with the .txt file if the written expressions are correct and admissible, but will appear the errors committed and the specific place (row and Column) In which the errors are found.

**Index Terms**—parsing, errors, Bison syntactic analyzer.

**Resumen**— El analizador sintáctico, encargado de realizar el análisis de cadenas de entrada es implementado a través del 'compilador-compiladores' Bison (compilador-compiladores), en combinación del definidor de reglas Flex. Se realizó el trabajo en la distribución Ubuntu del S.O Linux. Además, se utilizó una tabla de símbolos la cual será llamada para verificar las operaciones aceptables. Por medio de código se descargarán los programas necesarios descritos anteriormente. Además, también mediante comandos del terminal se llamarán a ejecutar los archivos y se llamarán a la tabla de símbolos para comprobar junto con el archivo .txt si las expresiones escritas son correctas y admisibles, sino aparecerán los errores cometidos y el lugar específico (fila y columna) en la que se encuentran los errores.

**Términos de Indexación**—parsing, Bison, analizador sintáctico.

## I. INTRODUCCIÓN

El parser o análisis sintáctico es el proceso de analizar cadenas de símbolos, ya sea en lenguaje natural o en lenguaje de máquina, conforme a las reglas de una gramática formal. Las estructuras de las cadenas de entrada consisten en una jerarquía de oraciones. Las más cortas son los símbolos básicos y las más largas son las sentencias.

El analizador debe ser capaz de manejar el número finito de posibles programas válidos que se pueden presentar a ella. La forma habitual de definir el lenguaje es especificar una

gramática. Una gramática es un conjunto de reglas un conjunto de reglas (o producciones) que especifica la sintaxis del lenguaje (es decir, qué es una oración válida en el lenguaje). Puede haber más de una gramática para un idioma dado. Además, es más fácil crear analizadores para algunas gramáticas que para otros. Afortunadamente, existe una clase de programas (a menudo llamada 'compilador-compiladores'), que puede tomar una gramática y generar un parser para ella en algún idioma.

Para este cometido se utilizó la herramienta, Bison, que permite generar analizadores sintácticos, es compatible con Yacc (otro generador de analizadores sintácticos) y convierte descripciones de gramáticas independientes del contexto a lenguaje C.

Para la elaboración de un compilador funcional es necesario la división del proceso en una serie de fases que variará con su complejidad. Por tanto, se debe unir el analizador léxico o el analizador semántico. Entonces es necesario de la herramienta Bison como la herramienta Flex.

Flex es el programa el cual define las reglas de reconocimiento de símbolos (Tokens), este programa, combinado con Bison, utiliza las definiciones de los tokens realizadas en el Bison para la comunicación entre ellos, los dos a grandes rasgos conforman el compilador.

## II. MATERIALES Y MÉTODOS

En primer lugar, para el desarrollo del analizador sintáctico, se debió conocer su estructura, esto se logró en base a ejemplos, además de conocer que la implementación de la herramienta Bison se la podía llevar a cabo en Windows y Linux. Para el desarrollo de este analizador y por la compatibilidad con las librerías que tiene Bison, se ocupó una Distro Linux (Ubuntu 16.04.01 LTS). [3] Para poder utilizar esta herramienta dentro de la Distro seleccionada, abrimos un terminal y colocamos el siguiente comando: `sudo apt-get install bison`. Para luego actualizar las librerías, para que no existan ningún conflicto al

momento de utilizar la herramienta Bison, esto con los siguientes comandos dentro de la terminal abierta: ***apt-get update*** y ***apt-get upgrade***.

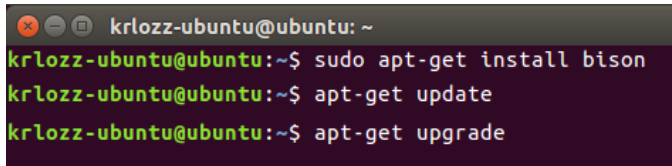


Fig. 1. Comandos de instalación de Bison dentro de Ubuntu 16.04.01 LTS

Una vez instalado empezamos a la realización del archivo en Bison, para lo cual debemos conocer la estructura que conforma un archivo en esta herramienta. [2] Donde se pueden identificar 3 partes claramente:

1. Declaraciones C y Declaraciones Bison
2. Sección de reglas gramaticales
3. Sección de código de usuario

Para la declaraciones en C, colocamos entre `%{ %}`, tipos y variables que se pretenden utilizar, además, todos los `include's` y `define's` necesarios.

```
%[
#include <stdio.h>
#include "tablaSimbolos.h"
extern int errors, lines, chars;
#define TABLE_FILE "tablaSimbolos"
#define ERROR 0
#define WARNING 1
#define NOTE 2
#define KEY_TYPE -100
FILE *yyin;
char *filename;
int yylex();
void yyerror();
int install(const char *lexeme, int type);
void install_keywords(char* keywords[], int n);
void install_id(char *name, int type);
void print_table(table_t table);
char *strbuild(int option, const char *str1, const char *str2);
void print_cursor();
void get_line(char *line);
#define YYDEBUG 1
%]
```

Fig. 2. Declaraciones en C

[4] Dentro de las declaraciones Bison colocamos los símbolos terminales o tokens y no terminales (%token), la precedencia de los operadores (%left, %right), tipos de datos de los valores semánticos de varios símbolos (%union, %type) y el símbolo inicial.

```
%token MAIN
%token IF ELSE DO WHILE FOR BREAK PRINT RETURN
%token INT_TYPE FLOAT_TYPE LETRA_TYPE STRING_TYPE BOOL_TYPE
%token CHAR STRING INTEGER FLOTANTE BOOLEANO
%token ID
%token MATH_INC MATH_DEC
%token EQL MENOR_QUE MAYOR_QUE AND OR NOT
%token KEYOP KEYCL ParetOP ParetCL BracketOP BracketCL
%token IGUAL

%union {
    char *lexeme;
    char *string;
    char *letra;
    int integer;
    float flotante;
    float booleano;
}

%type<lexeme> ID
%type<integer> INTEGER l_expr l_factor
%type<flotante> FLOTANTE g_expr g_term g_factor
%type<string> STRING t_expr
%type<letra> CHAR c_expr

%left KEYOP
%right KEYCL
%left ParetOP
%right ParetCL
```

Fig. 3. Declaraciones Bison

[4] En la sección de reglas gramaticales se colocará las producciones con las que se irán analizando para el analizador sintáctico.

```
program:
MAIN PareTOP PareTCL KEYOP comandos KEYCL metodo
|MAIN PareTOP declaration PareTCL KEYOP comandos KEYCL metodo
|MAIN PareTOP PareTCL KEYOP comandos metodo comandos KEYCL metodo
|MAIN PareTOP declaration PareTCL KEYOP comandos metodo comandos KEYCL metodo
| error {yerror("Formato de 'main' invalido", ERROR);}
;

metodo:
xempty
| lista_tipo ID PareTOP PareTCL KEYOP comandos KEYCL metodo
| lista_tipo ID PareTOP PareTCL KEYOP comandos metodo comandos KEYCL metodo
| lista_tipo ID PareTOP declaration PareTCL KEYOP comandos KEYCL metodo
| error {yerror("Formato de cuerpo invalido", ERROR);}
;

lista_tipo:
INT_TYPE
|FLOAT_TYPE
|STRING_TYPE
|LETRA_TYPE
|BOOL_TYPE
;
```

Fig. 4. Una parte de la sección de reglas gramaticales

Dentro de las reglas gramaticales se encontrarán las respectivas normas para el informe de errores, estas normas dependerán también de las declaraciones del archivo creado en flex con anterioridad.

```

t_attrs:
ID 'a' g_expr {if(get_entry($1)) {
    if(get_type($1) == INT_TYPE) {
        set_value($1, (int) $3);
    } else if(get_type($1) == FLOAT_TYPE) {
        set_value($1, (float) $3);
    } else if(get_type($1) == BOOL_TYPE) {
        if($3==>$3->i){set_value($1,$3);}
    } else if($3->is$3->i){set_value($1,$3);}
    } else {
        yyerror("tipos de datos incompatibles", WARNING);
    }
    else {char *str = (char *)strbuild(1, "declaracion de '%s' no encontrada", $1);
        yyerror(str, ERROR);
    }
ID 'a' t_expr {if(get_entry($1)) {
    if(get_type($1)==STRING_TYPE) {
        set_value($1, (char*)$3);
    } else {
        yyerror("tipos de datos incompatibles", WARNING);
    }
    else {char *str = (char *)strbuild(1, "declaracion de '%s' no encontrada", $1);
        yyerror(str, ERROR);
    }
ID 'a' c_expr {if(get_entry($1)) {
    if(get_type($1)==LETRA_TYPE) {
        set_value($1, (char*)$3);
    } else {
        yyerror("tipos de datos incompatibles", WARNING);
    }
    else {char *str = (char *)strbuild(1, "declaracion de '%s' no encontrada", $1);
        yyerror(str, ERROR);
    }
}

```

Fig. 5. Algunas normas para el informe de errores

[4] La sección de reglas gramaticales emplea el uso de dos funciones `yylex()` encargado de proporcionar tokens a Bison y `yyparse()` la cual realiza el análisis sintáctico.

Para el código generado, es necesario que todas las operaciones se realicen sobre registros. Lo cual se implementó un módulo en el generador de código que maneje el uso y la creación.

```

    \_expr EQ \_factor { $$-$1=$3; }
    \_expr AND \_factor { $$=$1&$3; }
    \_expr OR \_factor { $$=$1|$3; }
    \_expr MAYOR_QUE \_factor { $$=$1>$3; }
    \_expr MENOR_QUE \_factor { $$=$1<$3; }
    NOT \_expr { $$=$2; }
    \_factor
  ;

\_factor:
  ParetOP \_expr ParetOP { $$ = $2; }
  INTEGER { $$ = $1; }
  FLOTEANTE { $$ = $1; }
  ID {
    { if (get_entry($1)) { $$ = (int) get_value($1);
      else {char *str = (char *)strbuild(1, "declaracion de '%s' no encontrada", $1);
        yyyerror(str, ERROR);}}
  }
;

g_expr:
  g_expr '^' g_term { $$ = $1 + $3; }
  g_expr '*' g_term { $$ = $1 * $3; }
  g_term { $$ = $1; }

g_term:
  g_term '^' g_factor { $$ = $1 * $3; }
  g_term '/' g_factor { $$ = $1 / $3; }
  g_factor { $$ = $1; }

```

Fig. 6. Operaciones realizadas sobre registros

El enriquecimiento de la tabla de símbolos se encuentra dentro de la sección de código de usuario, para esto se empleó

una función la cual irá clasificando dependiendo de los tipos de símbolos almacenados. Esta función retornará tanto el tipo de símbolo como su valor, el orden de ingreso a la tabla dependerá de la sintaxis que se encontrará en el archivo input a ser compilado.

```
void printTable(table_t table) {
    FILE *f = fopen(TABLE_FILE, "w");
    int i;
    entry_t *cur;

    fprintf(f, "TABLA DE SÍMBOLOS\n");
    fprintf(f, "id\tentryes\n\n", table.t_size);

    fprintf(f, "-----\n");
    fprintf(f, " | TIPO | VALOR | \n");
    fprintf(f, "-----\n");

    for(i = 1; cur = table.t_head; cur != NULL; cur = cur->next, i++) {
        if(cur->type == INT_TYPE) {
            fprintf(f, " | %-5d | ENTERO | %s = %d\n", i, cur->lexeme, (int) cur->value);
        }
        else if(cur->type == FLOAT_TYPE) {
            fprintf(f, " | %-5d | FLOATANTE | %s = %f\n", i, cur->lexeme, (float) cur->value);
        }
        else if(cur->type == LETRA_TYPE) {
            fprintf(f, " | %-5d | CHAR | %s = %c\n", i, cur->lexeme, cur->value);
        }
        else if(cur->type == BOOL_TYPE) {
            fprintf(f, " | %-5d | BOOLEANO | %s = %f\n", i, cur->lexeme, cur->value);
        }
        else if(cur->type == STRING_TYPE) {
            fprintf(f, " | %-5d | STRING | %s = %f\n", i, cur->lexeme, cur->value);
        }
        else if(cur->type == KEY_TYPE) {
            fprintf(f, " | %-5d | CLAVE | %s = %f\n", i, cur->lexeme, cur->value);
        }
    }
}
```

Fig. 7. Función para el enriquecimiento de la tabla de símbolos

Además, en la sección de código de usuario, se encontrarán funciones las cuales nos permitirán el manejo del archivo input.txt a ser analizado, pues se verificará en primera instancia si el archivo está vacío y de no estarlo comprobará línea por línea con el analizador sintáctico.

Completada la estructura del archivo Bison, se obtendrá un archivo con extensión .y, este archivo lo debemos transformar a un archivo con extensión .c. Para esto abrimos un terminal de Ubuntu y colocaremos el siguiente comando: **bison -d -v AnaliSintactico.y**.

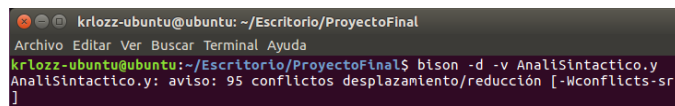


Fig. 8. Transformación de archivo de extensión .y a .c

[2] Este comando generará el respectivo archivo con extensión .c (AnaliSintactico.tab.c). El atributo -d generará el archivo \*.tab.h el cual servirá como cabecera para el archivo creado en Flex, crea tipo enumerado para los tokens que se usarán en Flex. El atributo -v generará el archivo \*.output el cual sirve para ver la gramática generada y depurarla si hay conflicto.



Fig. 9. Archivos generados con comando de Fig. 8.

Además, para el enriquecimiento de la tabla de símbolos es necesario crear dos programas en C. En las cuales siempre se iniciará la tabla como vacía y dependiendo de que no exista errores irá colocando token por token en la tabla.



Fig. 10. Programas para enriquecimiento de la tabla de símbolos.

Dentro del mismo terminal ya abierto, se procederá a crear un ejecutable colocando el siguiente comando: **gcc tablaSimbolos.c AnaliSintactico.tab.c lex.yy.c -lfl -ggdb -o <nombre del ejecutable>**.

### III. RESULTADOS Y DISCUSIÓN

Para la ejecución necesitaremos un archivo .txt donde tenga el lenguaje a analizar. Aquí colocaremos lenguaje de programación, que será analizado por el compilador creado por nosotros.



Fig. 1R. Archivo a ser analizado

Una vez generado el ejecutable, vamos a una terminal de Ubuntu y colocamos el siguiente comando: **./ejecutable <nombre del archivo .txt para ser analizado>**.

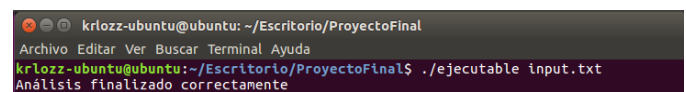


Fig. 2R. Ejecutar con el archivo seleccionado

Al colocar este comando el ejecutable compilará lo que se encuentre dentro del archivo .txt, comprobando si existe errores o no, de no existir errores generará un archivo donde se encontrará la tabla de símbolos.


Abrir 

TABLA DE SÍMBOLOS  
25 entries

-	TIPO	VALOR
1	ENTERO	tr = 0
2	BOOLEANO	pana = 0.000000
3	FLOTANTE	terwtw = 0.000000
4	ENTERO	trer = 0
5	ENTERO	lolo = 0
6	CHAR	as = -112.000000
7	BOOLEANO	vrđ = 0.000000
8	STRING	palabra = -16.000000
9	FLOTANTE	prueba4 = 1530000048128.000000
10	ENTERO	prueba3 = 0
11	ENTERO	prueba2 = 1
12	ENTERO	prueba = -1
13	FLOTANTE	hola = 0.000000
14	ENTERO	q = 0
15	PALABRA	float
16	PALABRA	int
17	PALABRA	return
18	PALABRA	print
19	PALABRA	break
20	PALABRA	for
21	PALABRA	while
22	PALABRA	do
23	PALABRA	else
24	PALABRA	if
25	PALABRA	main

Texto plano ▼

Fig. 3R. Tabla de símbolos generada

Uniendo el analizador sintáctico, con el analizador léxico implementado previamente se logra implementar un compilador funcional para una determinada gramática de lenguaje de programación. Se trata de la comprobación de la corrección del programa fuente, e incluye las fases correspondientes al Análisis Léxico, la cual consistió en la descomposición del programa fuente en componentes léxicos, el Análisis Sintáctico, el cual realiza la agrupación de los componentes léxicos en frases gramaticales, además de realizar un informe detallado de errores sintácticos.

El proyecto se encuentra en GitHub en el siguiente URL: <https://github.com/Krlozz/CompiladorFuncional>.

#### IV. CONCLUSIONES

La elaboración de un compilador funcional involucra la división del proceso en una serie de fases que variará con su complejidad. Generalmente se agrupan en dos tareas, el análisis del programa fuente y la síntesis del programa objeto. El objetivo de esta fase de elaboración del compilador fue la generación de la salida expresada en el lenguaje objeto y suele estar formado por una o varias combinaciones de fases de generación de código normalmente se trata de código intermedio o de código objeto y de la optimización de código en las que se busca obtener un código lo más eficiente posible).

El analizador sintáctico, también llamado parser, recibe como entrada los tokens que le pasa el Analizador Léxico (el analizador sintáctico no maneja directamente caracteres) y comprueba si esos tokens van llegando en el orden correcto (orden permitido por el lenguaje).

#### V. REFERENCIAS

[1] Tabango, "Compiladores - Flex y Bison", Es.slideshare.net, 2017. [Online]. Available:

<https://es.slideshare.net/TabangoSteven/compiladores-50328231>. [Accessed: 22- Feb- 2017]

[2] 2017. [Online]. Available:

<http://www.tamps.cinvestav.mx/~gtoscano/clases/LP/archivos/bison.pdf>. [Accessed: 22- Feb- 2017]

[3] "Bison- GNU Project - Free Software Foundation", Gnu.org, 2017. [Online]. Available:

<http://www.gnu.org/software/bison/>. [Accessed: 22- Feb- 2017]

[4] 2017. [Online]. Available:

<http://www.tamps.cinvestav.mx/~gtoscano/clases/LP/archivos/lexyacc.pdf>. [Accessed: 22- Feb- 2017]

#### VI. BIBLIOGRAFÍAS



**Carlos Antonio Ayala Tipán**, nació el 3 de octubre de 1995 en Quito – Ecuador, hijo de Marcelino Ayala y Narciza Tipán. Comenzó sus estudios en la Escuela Unidad Educativa Municipal Experimental “Antonio José de Sucre”, sus siguientes estudios los cursó en el Colegio Experimental e ISPED “Juan Montalvo”, y actualmente realiza sus estudios superiores en la Escuela

Politécnica Nacional, en donde cursa la carrera de Ingeniería en Sistemas Informáticos y de Computación, donde se encuentra en cuarto semestre. Sus hobbies favoritos son incursionar en tópicos de actualidad referente a su carrera, en sus tiempos libres le apasiona practicar algunos deportes tales como realizar bicicleta o el fútbol.



**Dennis Fernando Veintimilla Alvarado**

Nació el 23 de agosto de 1996 en Quito - Ecuador, hijo de Cristóbal Veintimilla y Marianita de Jesús, es el tercero de sus hermanos. Comenzó sus estudios en La Rioja, España en el

Colegio Nuestra Señora de la Vega. Estuvo en el “Instituto Ciudad de Haro” institución en la cual alcanzo el título de bachiller científico-tecnológico. En la universidad ingreso a la Escuela Politécnica Nacional para estudiar ingeniería en sistemas informáticos y de computación. Actualmente cursa el cuarto semestre de dicha carrera. Como pasatiempo le gusta jugar fútbol, salir con amigos y aprender cada vez más sobre las nuevas tecnologías.