Analizador Sintáctico con Bison

Alexis Guerrero, Dario Vargas

aguerrero_spp@hotmail.com, d_ario13@live.com

Escuela Politécnica Nacional

***Abstract*— For the construction of a compiler, inside the FRONT-END, it is prudent to develop the stage of interpretation of the symbols of the grammar; known like parsing later to the implementation of the lexicographical analysis with the tool Flex. To carry out this phase it is necessary to use the complementary tool to Flex, it's Bison.**

***Index Terms*—front-end, bison, syntactic analyzer.**

***Resumen*—Para la construcción de un compilador, dentro del FRONT-END, es prudente desarrollar la etapa de interpretación de los símbolos de la gramática; conocida como análisis sintáctico, posterior a la implementación del análisis lexicográfico con la herramienta Flex. Para llevar a cabo esta fase es necesario utilizar la herramienta complementaria a Flex, Bison.**

***Términos de Indexación*—front-end, bison, analizador sintáctico.**

I. INTRODUCCIÓN

El estudio y análisis de funcionamiento de compiladores es muy importante en la formación de esta rama de ingeniería, para lo cual se plantea construir un compilador por etapas y esta vez se desarrolla en analizador sintáctico con la herramienta Bison. Bison es un generador de analizadores sintácticos, su implementación es muy importante en la construcción de un traductor de lenguajes computacionales, en la etapa de Front-End; este software trabaja a la par con Flex para entender los símbolos correctamente entregados por el analizador lexicográfico. El proyecto fue desarrollado en Linux, por tener la licencia GNU a igual que Bison y Flex lo que ofrece una mayor compatibilidad.

II. MATERIALES Y MÉTODOS

Bison

Es un generador de "parsers", es decir analizadores sintácticos, de tipo LALR(1), genera código fuente C a partir de gramática independiente del contexto, y es distribuido bajo licencia GNU.

El proceso parte de un fichero de texto conteniendo una gramática incontextual de la especificación del analizador sintáctico, tras procesar ese fichero Bison genera código C; y se obtendrá un archivo *.tab.c. Este archivo contendrá la función yyparse(), misma que es llamada para que se ejecute el analizador sintáctico generado por Bison.

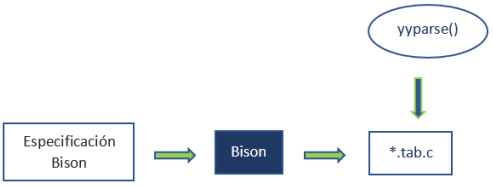

Imagen 1: Proceso del analizador sintáctico de Bison.

El fichero de especificación de Bison tiene 3 secciones:

```
Declaraciones C y Bison
%%
Reglas Gramaticales (Bison)
%%
Código de Usuario
```

Para declarar código C dentro de este fichero se lo debe realizar con los símbolos `%{ <código> %},` mientras que para realizar comentarios con los símbolos `/* <comentarios> */.`

Declaraciones: definición de los símbolos terminales (tokens).

```
%token <nombre_del_terminal>
```

Reglas Gramaticales: reglas de producción de una gramática libre de contexto, terminales y non-terminales.

```
<terminal>:   <non-terminal>   {<acción>}   |
              <non-terminal> {<acción>} ;
```

Código de Usuario: se coloca el código C que será devuelto a la salida, tal cual se lo declare.

Construcción

El desarrollo del analizador sintáctico se implementó en la distribución de Linux, Ubuntu, por ser software libre y de código abierto existe mayor compatibilidad con las librerías y herramientas de los softwares Flex y Bison.

La compilación, manejo de archivos e instalación de las herramientas se las llevó a cabo por medio del terminal de Linux, es decir, por consola, mientras que la configuración y escritura de archivos se realizó en el editor de texto gedit.

La instalación de Bison es un proceso muy simple se lo hace

como súper usuario con el comando: **apt-get install bison**.

```
root@ubuntu: /home/x-vier
root@ubuntu:/home/x-vier# apt-get install bison
Reading package lists... Done
Building dependency tree
```

Imagen 2: Instalación de Bison.

Ya instalado el software se procede a construir el fichero de especificación. Para escribir las declaraciones en lenguaje C es necesario incluir la librería de `<stdio.h>`, de este modo se reconocerán las funciones de C; así mismo, es necesario tener acceso a los símbolos de entrada almacenados, por lo cual se llama a la tabla de símbolos **"`tabalaSimbolos.h`"** que almacena y busca el tipo de retorno según las entradas.

```c
#ifndef SYMBOLTABLE_H
#define SYMBOLTABLE_H

/* Estructura para almacenar una entrada de tabla de símbolos */
struct entry_s {
        int type;               /* Tipo de lexema */
        char *lexeme;           /* String que representa el lexema */
        float value;            /* Valor numérico de la entrada */
        struct entry_s *next;   /* Puntero a la entrada siguiente */
};
typedef struct entry_s entry_t;

/* Estructura de la tabla de simbolos */
struct table_s {
        int t_size;             /* Tamaño de la tabla */
        entry_t *t_head;        /* Puntero a la primera entrada */
};
typedef struct table_s table_t;

/* Tabla de simbolos */
table_t table;

/* Inicializa la tabla de simbolos */
void init_table();

/* Crea una nueva entrada y devuelve un puntero */
entry_t *create_entry(int type, const char *lexeme, float value);

/* Pone la entrada al principio de la tabla de simbolos (binding) */
void put_entry(entry_t *new_entry);

/* Busca en la tabla de simbolos un lexema */
/* Retorna
        -NULL Si la entrada buscada no existe
        -Un puntero a la entrada encontrada */
```

Imagen 3: Tabla de Símbolos.

```c
%{
        /*DECLARACIONES c*/
        #include <stdio.h>
        #include "tablaSimbolos.h"
        extern int errors, lines, chars;

        // Destino del archivo que contiene la tabla de simbolos
        #define TABLE_FILE "tablasimbolos"

        // Declaracion de los mensajes
        #define ERROR 0
        #define WARNING 1
        #define NOTE 2

        // Tipo de llave para la tabla de símbolos
        #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
%}
```

Imagen 4: Declaraciones C

También dentro de la sección de declaraciones se procede a identificar y declarar los tokens con el respectivo nombre del terminal, según el lenguaje de Bison.

```c
/*DECLARACION DE BISON*/
%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
%left BracketOP
%right BracketCL
%token ASIGNACION
%left  COMA
```

Imagen 5: Declaraciones Bison

En la segunda sección correspondiente a las reglas gramaticales, se colocan los terminales con las producciones que derivan, teniendo en cuenta el lenguaje de Bison. Así mismo, cabe recalcar que, para realizar operaciones matemáticas o lógicas pertenecientes a las producciones es necesario utilizar lenguaje de bajo nivel para trabajar a nivel de los registros. Finalmente en esta sección implementará el informe de errores, mediante funciones de Bison y Flex como yyerror().

```
/* IF ELSE */
controlF:
        IF ParetOP l_expr ParetCL KEYOP comandos KEYCL controlElse
        ;

controlElse:
        %empty
        | ELSE KEYOP comandos KEYCL
        ;


/* FOR */
controlFor:
        FOR ParetOP ffirst FinCommand l_expr FinCommand fthird ParetCL KEYOP comandos KEYCL
        ;

ffirst:
        %empty
        | attrib_list
        ;

fthird:
        %empty
        | comando_simple COMA fthird
        | comando_simple
        ;


/* WHILE */
controlWhile:
        WHILE ParetOP l_expr ParetCL KEYOP comandos KEYCL
        ;
```

Imagen 6: Gramática de If Else, For y While

```
/* EXPRESIONES MATEMÁTICAS Y LÓGICAS */

t_expr:
        STRING  {$$=$1;}
        ;

c_expr:
        CHAR    {$$=$1;}
        ;


l_expr:
        l_expr EQL l_factor { $$=$1==$3; }
      | l_expr AND l_factor { $$=$1&&$3; }
      | l_expr OR l_factor { $$=$1||$3; }
      | l_expr MAYOR_QUE l_factor { $$=$1>$3; }
      | l_expr MENOR_QUE l_factor { $$=$1<$3; }
      | NOT l_expr { $$ = !$2; }
      | l_factor
      ;

l_factor:
        ParetOP l_expr ParetOP { $$ = $2; }
        | INTEGER { $$ = $1; }
        | FLOTANTE { $$ = $1; }
        | ID    {/* Si hay entrada obtiene su valor */
                if(get_entry($1)) {$$ = (int) get_value($1);}
                else {char *str = (char *)strbuild(1, "declaracion de '%s' no encontrada", $1);
                yyerror(str, ERROR);}}
        ;
```

Imagen 7: Expresiones lógicas, matemáticas e informe de errores con yyerror().

Finalmente, en la tercera y última sección perteneciente al código de usuario, se analiza los símbolos de entrada y se genera un archivo de salida con la respectiva tabla de símbolos. Generalmente de la escribe en lenguaje C junto con las funciones de Bison.

```
%%

int main( int argc, char **argv )
{
        char* keywords[] = {"main", "if", "else", "do", "while", "for", "break", "print",
                                                "return", "int", "float"};

        /* Comprueba si hay más archivos a leer */
        ++argv, --argc;
        if ( "ingreso.txt" > 0 ) {
                filename = strdup("ingreso.txt");
                yyin = fopen( "ingreso.txt", "r" );
        }
        else {
                filename = strdup("stdin");
                yyin = stdin;
        }

        /* Inicializa la tabla de símbolos */
        init_table();

        /* Instalar palabras clave en las primeras entradas de la tabla de símbolos */
        install_keywords(keywords, 11);

        /* Ejecuta el parser */
        yyparse();

        if(errors==0) {
                printf("Análisis finalizado correctamente\n");
        }
        else {
                printf("Se han encontrado %d errores\n", errors);
        }
```

Imagen 8: Código de usuario, análisis de la entrada de símbolos.

Ejecución

Para la ejecución de nuestro analizador sintáctico es necesario previamente compilar con Flex el analizador lexicográfico con el comando `flex analizadorLexico.l`, lo cual creara como salida un archivo `lex.yy.c`, con la información del scanner, este archivo está escrito en lenguaje C. Más adelante con el compilador de Bison ejecutamos el analizador sintáctico con las opciones de comando -yd para llamar a POSIX yacc, que es un programa que genera analizadores sintácticos como se muestra `bison -yd analizadorSintactico.y`, esto genera dos archivos `y.tab.h` y `y.tab.c`, una librería que debe ser llamada por el analizador lexicográfico y un archivo escrito en lenguaje C. Finalmente para crear el archivo ejecutable del compilador se lo hace con el compilador de C de Linux gcc, como muestra el comando `gcc y.tab.c lex.yy.c -lfl -o compilador.exe`; aquí se llaman a los archivos del analizador lexicográfico y sintáctico escritos en C; y como resultado tenemos un archivo ejecutable cuyo nombre es `compilador.exe` , y se especifica en el comando.

```
x-vier@ubuntu: ~/Desktop/Analizador Sintactico

x-vier@ubuntu:~/Desktop/Analizador Sintactico$ flex analizadorLexico.l
x-vier@ubuntu:~/Desktop/Analizador Sintactico$ bison -yd analizadorSintactico.y
analizadorSintactico.y: warning: 95 shift/reduce conflicts [-Wconflicts-sr]
x-vier@ubuntu:~/Desktop/Analizador Sintactico$ gcc y.tab.c lex.yy.c -lfl -o compilador.exe
```

Imagen 9: Ejecución del analizador lexicográfico y sintáctico.

IV. CONCLUSIONES

El analizador sintáctico ejecutado por Bison, trabaja en conjunto con el analizador lexicográfico ejecutado por Flex, ya que se necesita previamente el análisis de la entrada de los símbolos correctos para su posterior interpretación.

La aplicación de la teoría sobre la construcción de la gramática a partir del reconocimiento y declaración de los terminales y non-terminales es muy importante para crear la segunda parte del fichero de Bison, así mismo como el reconocimiento de las funciones de Bison.

Es importante manejar y crear la tabla de símbolos como una librería para facilitar su uso; así mismo como es importante reconocer su funcionamiento ya que en este caso trabaja como tabla de implementación y de tipos.

El proyecto se encuentra en GitHub en el siguiente URL: https://github.com/DarioXavier/Analizador-Sintactico-y-Tabla-de-Simbolos.git

V. REFERENCIAS

[1] F. J. C. Limón, «Galeon.com,» 2015. [En línea]. Available: http://10380054.galeon.com/u5.htm.

[2] V. Paxson, «TLDP,» Abril 1995. [En línea]. Available: http://es.tldp.org/Manuales-LuCAS/FLEX/flex-es-2.5.html#SEC2.

[3] Yacc, « The Open Group Base Specifications Issue 6,» IEEE Std 1003.1, 2004 Edition. [En línea]. Available: http://pubs.opengroup.org/onlinepubs/009604599/utilities/yacc.html

[4] D. J. B. Torralvo, «Universidad Politécnica de Madrid,» Junio 2014. [En línea]. Available: http://oa.upm.es/32288/1/PFC_JOSE_BARBERA_TORRALVO.pdf.

[5] S. Bejarano, «Compilar un programa usando flex y bison,» 2013. [En línea]. Available: https://www.youtube.com/watch?v=denLhiT2egw.