

Analizador Sintáctico y Tabla de Símbolos

David Morales and Paúl Cadena

Abstract—En el presente artículo se presentará de manera detallada la implementación de un analizador sintáctico. Mediante la herramienta Bison en el sistema operativo Linux. Para la implementación de este analizador también se utilizará el analizador léxico generado anteriormente. El analizador sintáctico es el encargado de realizar el análisis de las cadenas de entrada que se revisaron previamente en el analizador léxico. La implementación del analizador sintáctico se realizó dentro de un ambiente linux, en ese caso Ubuntu. Esto permite tener un mejor manejo de las herramientas adecuadas.

Keywords—Bison , parser , tabla de símbolos.

I. INTRODUCTION

El análisis sintáctico es la fase del compilador que se encarga de chequear el texto de entrada en base a una gramática dada. Y en caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce. En teoría, se supone que la salida del analizador sintáctico es alguna representación del árbol sintáctico que reconoce la secuencia de tokens suministrada por el analizador léxico.

Si un compilador tuviera que procesar sólo programas correctos, su diseño e implantación se simplificarían mucho. Pero los programadores a menudo escriben programas incorrectos, y un buen compilador debería ayudar al programador a identificar y localizar errores. Es más, considerar desde el principio el manejo de errores puede simplificar la estructura de un compilador y mejorar su respuesta a los errores. Los errores en la programación pueden ser de los siguientes tipos:

- Léxicos, producidos al escribir mal un identificador, una palabra clave o un operador.
- Sintácticos, por una expresión aritmética o paréntesis no equilibrados.
- Semánticos, como un operador aplicado a un operando incompatible.
- Lógicos, puede ser una llamada infinitamente recursiva.

El manejo de errores de sintaxis es el más complicado desde el punto de vista de la creación de compiladores. Nos interesa que cuando el compilador encuentre un error, se recupere y siga buscando errores. Por lo tanto el manejador de errores de un analizador sintáctico debe tener como objetivos:

- Indicar los errores de forma clara y precisa. Aclarar el tipo de error y su localización.
- Recuperarse del error, para poder seguir examinando la entrada.
- No ralentizar significativamente la compilación.

El analizador debe ser capaz de manejar infinito número de programas válidos, creados en base a una gramática dada.

Para la implementación de este analizador se hace uso de Bison, herramienta de gran utilidad y fácil uso dentro de un ambiente linux. Esta herramienta permite generar analizadores sintácticos, convierte descripciones de gramáticas a lenguaje C,

para que las instrucciones realizadas para la gramática puedan ser ejecutadas. En vista de que el analizador sintáctico es una fase más dentro del compilador, se debe unir con las etapas anteriores como son los analizadores: léxico y semántico.

II. TÉCNICAS Y HERRAMIENTAS

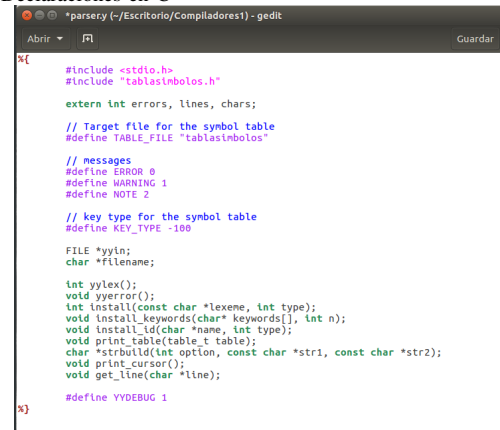
A. Bison

GNU bison es un programa generador de analizadores sintácticos de propósito general perteneciente al proyecto GNU disponible para prácticamente todos los sistemas operativos, se usa normalmente acompañado de flex aunque los analizadores léxicos se pueden también obtener de otras formas. Bison convierte la descripción formal de un lenguaje, escrita como una gramática libre de contexto LALR, en un programa en C, C++, o Java que realiza análisis sintáctico. Es utilizado para crear analizadores para muchos lenguajes, desde simples calculadoras hasta lenguajes complejos. Para utilizar Bison, es necesaria experiencia con el la sintaxis usada para describir gramáticas. GNU bison tiene compatibilidad con Yacc, todas las gramáticas bien escritas para Yacc, funcionan en Bison sin necesidad de ser modificadas. Cualquier persona que esté familiarizada con Yacc podría utilizar Bison sin problemas. Bison fue escrito en un principio por Robert Corbett; Richard Stallman lo hizo compatible con Yacc y Wilfred Hansen de la Carnegie Mellon University añadió soporte para literales multicaracter y otras características.[1]

Bison se comporta de 3 partes:

- **Sección de declaraciones:** Tenemos declaraciones de C y declaraciones de Bison[2]
Para la declaraciones en C, colocamos entre `%{ %}`, tipos y variables que se pretenden utilizar, además, todos los `include's` y `define's` necesarios.

Fig. 1. Declaraciones en C



```
%{
#include <stdio.h>
#include "tablasimbolos.h"

extern int errors, lines, chars;

// Target file for the symbol table
#define TABLE_FILE "tablasimbolos"

// messages
#define ERROR 0
#define WARNING 1
#define NOTE 2

// key type for the symbol table
#define KEY_TYPE -100

FILE *yyin;
char *filename;

int yytex();
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
}
```

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.

Fig. 2. Declaraciones en Bison

```

#define YYDEBUG 1

%}|
%start program
%token MAIN
%token IF ELSE DO WHILE FOR BREAK PRINT RE
%token INT_TYPE FLOAT_TYPE BOOL_TYPE
%token STRING INTEGER REAL BOOLEAN
%token ID
%token MATH_INC MATH_DEC
%token LOG_EQL LOG_LT LOG_GT LOG_AND LOG_OR

%union {
    char *lexeme;
    int integer;
    float real;
}

%type<lexeme> ID
%type<integer> INTEGER l_expr l_factor
%type<real> REAL g_expr g_term g_factor
%type<bool> BOOLEAN

%left '+' '-'
%left '*' '/'
%left MATH_INC MATH_DEC
%left LOG_EQL LOG_LT LOG_GT LOG_AND LOG_OR
%right '='
%right LOG_NOT

%lr-parser

%%

```

- **Sección de reglas gramaticales** En esta parte, se colocan las producciones que servirán de base para el analizador sintáctico. En esta sección se encuentran también las reglas para el informe de errores, dichas reglas hacen uso también del analizador léxico generado[2]

anteriormente. Se declaran los terminales: `%token`. Las reglas gramaticales como producciones y se permite la regla vacía.

Fig. 3. Parte de las reglas gramaticales

```

/*****
PROGRAM BODY
*****/
program : declaration ';' program
        | MAIN '(' ')' '{' commands '}' program_end
        | error {yyerror("formato de 'main' invalido", ERROR);};

program_end : %empty
            | ID '(' ')' '{' commands '}' program_end
            | error {yyerror("formato de cuerpo invalido", ERROR);};

/*****
COMMANDS
*****/
commands: command_list;

command_list: '{' command_list '}'
            | command command_list
            | %empty
            ;

command: cmd ';'
        | stmt
        | error { yyerror("instruccion errónea o ';' faltante", ERROR);};

stmt: if_stmt
    | for_stmt
    | while_stmt
    | dowhile_stmt
    ;

cmd: PRINT STRING
    | BREAK
    | RETURN
    | attrib
    | declaration
    ;

/*****

```

- **Sección de código de usuario** Se genera de acuerdo a la necesidad del programa. Estas instrucciones se copian literalmente en el archivo `tab.c`. En este proyecto se genera 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.

Fig. 4. Código de usuario

```

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

    // Checks if there are more files to be read
    ++argv, --argc;
    if ( argc > 0 ) {
        filename = strdup(argv[0]);
        yyin = fopen( argv[0], "r" );
    }
    else {
        filename = strdup("stdin");
        yyin = stdin;
    }

    // Initializes the symbol table
    int_table();

    // Install keywords to the first entries of the symbol table
    install_keywords(keywords, 11);

    // Executes the parser
    yyparse();

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

    // Prints the generated symbol table
    print_table(table);

    return 0;
}

// Inserts entry at the end of the symbol table
// -if it's already installed returns 0
int install(const char *lexeme, int type) {

```

Fig. 5. Tabla de Símbolos

```

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

    fprintf(f, "TABLA DE SÍMBOLOS\n");
    fprintf(f, "%d entries\n", table.t_size);

    fprintf(f, "+-----+-----+-----+\n");
    fprintf(f, "| - | TIPO | TOKEN = 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 | FLOTANTE | %s = %f\n", i, cur-
            >lexeme, (float) cur->value);
        }
        else if(cur->type == BOOL_TYPE) {
            fprintf(f, "| %-5d | BOOLEANO | %s = %f\n", i, cur-
            >lexeme, (bool) cur->value);
        }
        else if(cur->type == KEY_TYPE) {
            fprintf(f, "| %-5d | PALABRA | %s\n", i, cur->lexeme);
        }
    }

    fprintf(f, "+-----+-----+-----+\n");
}

void verror(const char *msg, int tvpe) {

```

B. Código para la ejecución

Archivo bash Se crea un script ejecutable(archivo bash) que nos permite ejecutar todos los programas a utilizar a la misma vez. Al inicio del script es necesario escribir lo siguiente: `#!/bin/bash`. Luego hacemos uso de `ECHO`, lo cual permite realizar alguna acción. Aquí también se utilizan

algunas funciones tales como:

- **flex scanner.l** Permite ejecutar el analizador léxico que devolverá un archivo en c para su ejecución.
- **bson -d -v parser.y** Este comando generará el respectivo archivo con extensión .c (parser.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.
- Dentro del mismo terminal, se procederá a crear un ejecutable colocando el siguiente comando: `gcc tablaSimbolos.c parser.tab.c lex.yy.c -lfl -ggdb`. El cual permite crear un ejecutable para analizar el programa de entrada.
- **.lexec programa.txt** Esta función es la que permite el análisis del programa que se quiere.

III. RESULTADOS

Para la ejecución se necesitará un archivo .txt escrito en lenguaje C. Este programa será analizado por el compilador. Ejecutamos el script.sh desde la consola para compilar el programa de entrada.

Fig. 6. Código en C

```

hola.txt (~/Escritorio/Sint[actico] - gedit
Abrir  Guardar

int i;
int f;
int h;
main() {
    i = 3;
    //comentario
    /*comen

    if ( i == f ) {
        i = 3;
    }
}

```

Luego de la compilación, se tiene como salida la tabla de símbolos en un archivo. txt en caso de que el programa no tenga errores. Caso contrario el compilador devolverá los errores tanto léxicos como sintácticos.

Fig. 7. Tabla de Símbolos

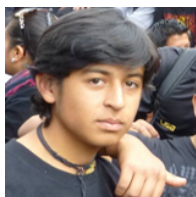
TABLA DE SÍMBOLOS		
14 entries		
-	TIPO	TOKEN = VALOR
1	ENTERO	i = 0
2	ENTERO	f = 0
3	ENTERO	i = 3
4	PALABRA	float
5	PALABRA	int
6	PALABRA	return
7	PALABRA	print
8	PALABRA	break
9	PALABRA	for
10	PALABRA	while
11	PALABRA	do
12	PALABRA	else
13	PALABRA	if
14	PALABRA	main

IV. CONCLUSIONES

Como se pudo apreciar, el compilador está compuesto por varias etapas que interactúan entre sí. De manera que permite la correcta ejecución. El analizador sintáctico recibe como entrada los tokens validados por el analizador léxico. Comprueba que lleguen en el orden correcto y que pertenezcan a la gramática previamente definida. Como salida devuelve el enriquecimiento de la tabla de símbolos.

REFERENCES

- [1] "Guía básica de Bison yFlex" *monografias.com*, 2014.[online]. Available <http://www.monografias.com/trabajos58/guia-bison-flex/guia-bison-flex.shtmlixzz4ZWHegLTB>[Accessed:2017-02-22]
- [2] "Bison", G. Toscano[online]. Available: <http://www.tamps.cinvestav.mx/gtoscano/clases/LP/archivos/bison.pdf>[Accessed:2017-02-23]



David Morales nació el 9 de mayo de 1995 en la parroquia de San José de Minas, Pichincha, Ecuador. Hijo de Blanca Pillajo y Jorge Morales. Realizó sus estudios de primaria en su pueblo natal. Los estudios secundarios los realizó en la ciudad de Quito en el colegio "Electrónico Pichincha", en el cual obtuvo el título de "Bachiller Técnico en Electrónica". Actualmente cursa el 4 de semestre de Ingeniería en Sistemas Informáticos y de Computación, en la EPN. En su tiempo incursiona en temas de interés referentes a su carrera de estudio, también es un

apasionado al fútbol y a la música; a la cual también le dedica gran parte de su tiempo.



Paúl Sebastián Cadena Estudiante de 20 años de edad de la universidad Escuela Politécnica Nacional, nacido en Atuntaqui-Ecuador en 1996, debido a su gusto por la computación actualmente es estudiante de la Carrera de Ingeniería en Sistemas. Graduado del colegio Sánchez y Cifuentes de la ciudad de Ibarra como Fisicomatemático. Es el segundo hijo de la familia, siendo el menor detrás de su hermana 2 años mayor que él.