



# Entrega final GLF

Grupo G



# Integrantes:

Manuel de Castro Caballero  
María Ruiz Molina  
Andrés Trigueros Vega

# Índice

- Lex - Andrés
- Yacc - María
- Ast - Manuel
- Tabla de símbolos - Manuel
- Representación gráfica AST- Manuel
- Ejemplo- Manuel

```

%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#include "autils.h"
#include "ac.syn.h"

#define KWORDSN 10

char *keyWords[KWORDSN] = {
    "cos",
    "div",
    "else",
    "if",
    "ln",
    "print",
    "read",
    "sin",
    "tan",
    "while",
};
unsigned keyCodes[KWORDSN] = {
    COS,
    DIV,
    ELSE,
    IF,
    LN,
    PRINT,
    READ,
    SIN,
    TAN,
    WHILE,
};

int yywrap(void) { return 1; }

extern char programName[];

static char *readStr();
static void addStr(char **s, unsigned long *len, char c);
%}

```

# LEX

LETTER	([_a-zA-Z])
DIGIT	([0-9])
NUMBER	({DIGIT}+)
HEX_DIGIT	([0-9a-fA-F])
HEX	(0x{HEX_DIGIT}+)
FLOAT	((({NUMBER}"."{DIGIT}*) ({DIGIT}*"."{NUMBER})))
ID	({LETTER}({LETTER} {DIGIT})*)
DEL_BL_A	(\{)
DEL_BL_C	(\})
WSPC	([ \t\f\r])
WSCPS	({WSPC}+)
OP_AR	([+*-/%=()^])
OP_LOG	("==" "!=" "&&" "  " "<" ">")
OP_BIT	("&" " " "~" "<<" ">>")
STR_START	(["])
LINE_COMM	("//".*\n)
COMM	("/*"([^\n/*])**"/")
%%	

# LEX

/\*

```
[\\n]  {      /*Salto de linea*/
            yylineno++;
            return yytext[0];
        }

{DEL_BL_A}    return DEL_BL_A;
{DEL_BL_C}    return DEL_BL_C;
{WSPC}      ;
{LINE_COMM}  {
                yylineno++;
                return '\\n';
            }

{COMM}  {      /*Cuenta las líneas que ocupa el comentario*/
                bool hasNewlines = false;

                for (int i = 0; i < strlen(yytext); i++)
                {
                    if (yytext[i] == '\\n')
                    {
                        hasNewlines = true;
                        yylineno++;
                    }
                }

                if (hasNewlines) return '\\n';
            }
        }
```

# LEX

```

{ID} {
    unsigned i = 0;
    int r = -1;

    /*Recorre el listado de palabras clave para ver si la introducida lo es*/
    while (i < KWORDSN && r < 0)
    {
        if ((r = strcmp(keyWords[i], yytext)) == 0) return keyCodes[i];
        ++i;
    }
    /*Si no es una palabra clave y lo devuelve como identificador*/
    mallocCheck(yyval.s.u.string, strlen(yytext) + 1);
    strcpy(yyval.s.u.string, yytext);
    yyval.s.type = ID_id;

    return ID;
}

{NUMBER} {
    /*Se lee un numero entero*/
    sscanf(yytext, "%ld", &yyval.s.u.int_value);
    yyval.s.type = INT_id;

    return INT;
}

{HEX} {
    sscanf(&yytext[2], "%lx", &yyval.s.u.int_value);
    yyval.s.type = INT_id;

    return INT;
}

{FLOAT} {
    /*Se lee un numero con decimales*/
    sscanf(yytext, "%lf", &yyval.s.u.real_value);
    yyval.s.type = REAL_id;

    return FLOAT;
}

{STR_START} {
    yyval.s.u.string = readStr();
    yyval.s.type = STR_id;
    return STR;
}

```

# LEX

```

{OP_AR} {
    return yytext[0];
}

{OP_BIT} {
    switch (yytext[0])
    {
        case '&':
            return BAND;
        case '|':
            return BOR;
        case '~':
            return BNOT;
        case '<':
            return SL;
        case '>':
            return SR;
    }
}

{OP_LOG} {
    switch (yytext[0])
    {
        case '=':
            return EQ;
        case '!':
            return NE;
        case '&':
            return LAND;
        case '|':
            return LOR;
        case '<':
            return LT;
        case '>':
            return GT;
    }
}

. {
    fprintf(stderr, "%s(%d): error -- Caracter inesperado %c\n", programName, yylineno, yytext[0]);
    exit(LEXICAL_ERROR);
}

```

# LEX



```

static void addStr(char **s, unsigned long *len, char c)
{
    char buf[2];
    buf[0] = c;
    buf[1] = '\0';

    if (strlen(*s) >= *len) {
        char *ss = (char *)malloc(sizeof(char) * (*len + 1025));
        strcpy(ss, *s);
        *s = ss;
        *len += 1024;
    }
    strcat(*s, buf);
}

static char *readStr()
{
    int c;
    char *buff;
    mallocCheck(buff, sizeof(char) * 257);
    unsigned long len = 256;

    buff[0] = '\0';

    do {
        c = input();
        if (c < ' ')
        {
            fprintf(stderr, "%s(%d): error -- Símbolo no esperado en cadena de caracteres: %d\n", programName, yylineno, c);
            exit(LEXICAL_ERROR);
        }
        if (c == '\n') break;
        if (c == '\\')
        {
            c = input();
            if (c != '\\') && c != '\n')
            {
                unput(c);
                c = '\\';
            }
        }
        addStr(&buff, &len, c);
    } while(1);
    return buff;
}

```

# LEX

```

%{

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "autls.h"
#include "ast.h"
#include "ac.syn.h"

int yylex();
int yyerror(char *error);
extern int yylineno;
extern FILE *yyin;

/* Árbol de sintaxis */
static ast_t *ast = NULL;

%}

/* Tipo/estructura de los tokens */
%union {
    struct syn_elem {
        unsigned char type;           /* Tipo de token */
        union {                       /* Valor útil del token */
            double real_value;        /* Como número real */
            long int_value;           /* Como número entero */
            char *string;             /* Como cadena de caracteres */
            struct ast_s *node;
        };
    } s;
}

```

# YACC

**%nonassoc** EQ NE LAND LOR LT GT

**%left** BAND BOR SL SR

**%left** '+' '-'

**%left** '\*' '/' '%' DIV

**%right** '^'

**%nonassoc** UNARY BNOT

**%token** <s> ID

**%term** PRINT READ SIN COS TAN LN

**%token** <s> INT

**%token** <s> FLOAT

**%token** <s> STR

**%term** WHILE IF ELSE DEL\_BL\_A DEL\_BL\_C

**%type** <s> PROGRAM PROGRAM\_ELEMENT BLOCK SENTENCE\_GROUP SENTENCE EXPR

**%%**

# YACC

```
PROGRAM --> PROGRAM_ELEMENT | PROGRAM PROGRAM_ELEMENT
PROGRAM_ELEMENT --> SENTENCE '\n' | '\n'
BLOCK --> DEL_BL_A SENTENCE_GROUP DEL_BL_C | '\n' DEL_BL_A SENTENCE_GROUP DEL_BL_C
SENTENCE_GROUP --> PROGRAM_ELEMENT | SENTENCE_GROUP PROGRAM_ELEMENT
SENTENCE --> BLOCK | ID '=' EXPR | PRINT ... | READ ... | WHILE '(' EXPR ')' SENTENCE | IF '(' EXPR ')' SENTENCE ELSE SENTENCE
| IF '(' EXPR ')' SENTENCE
EXPR --> EXPR EQ EXPR | ... | EXPR '+' EXPR | ... | '-' EXPR | ... | EXPR BAND EXPR | ... | FLOAT | ... | SIN '(' EXPR ')' | ... | LN '(' EXPR ')'
```

# YACC

```

/* Producciones de un programa */
PROGRAM
/* Un único bloque de programa */
: PROGRAM_ELEMENT
{
    ast = newRoot('r', ast, $1.u.node);
}
/* Varios bloques de programa */
| PROGRAM PROGRAM_ELEMENT
{
    ast = newRoot('r', ast, $2.u.node);
}
;

/* Elemento del programa */
PROGRAM_ELEMENT
/* Una sentencia */
: SENTENCE '\n'
{
    $$ = $1;
}
/* Sentencia vacía */
| '\n'
{
    $$ .type = AST_NODE_id;
    $$ .u.node = NULL;
}
;

/* Bloque de sentencias */
BLOCK
: DEL_BL_A SENTENCE_GROUP DEL_BL_C
{
    $$ = $2;
}
| '\n' DEL_BL_A SENTENCE_GROUP DEL_BL_C
{
    $$ = $3;
}
;

```

# YACC

# YACC

```
/* Grupo de sentencias */
SENTENCE_GROUP
/* Un único elemento de un programa */
: PROGRAM_ELEMENT
{
    $$ = $1;
}
/* Un grupo de sentencias y un elemento del programa (recursivo) */
| SENTENCE_GROUP PROGRAM_ELEMENT
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode('g', NULL, newRoot('r', $1.u.node, $2.u.node));
}
;

/* Sentencias */
SENTENCE
: BLOCK
{
    $$ = $1;
}
/* Asignación */
| ID '=' EXPR
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode('=', newLeafString(ID, $1.u.string), $3.u.node);
}
/* Impresión de un valor */
| PRINT EXPR
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode(PRINT, NULL, $2.u.node);
}
/* Impresión de una string */
| PRINT STR
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode(PRINT, newLeafString(STR, $2.u.string), NULL);
}
}
```

```

/* Impresión de una string y un valor */
| PRINT STR EXPR
{
    $$type = AST_NODE_id;
    $$u.node = newNode(PRINT, newLeafString(STR, $2.u.string), $3.u.node);
}
/* Lectura de un valor */
| READ ID
{
    $$type = AST_NODE_id;
    $$u.node = newNode(READ, NULL, newLeafString(ID, $2.u.string));
}
/* Lectura de un valor con mensaje de aviso */
| READ STR ID
{
    $$type = AST_NODE_id;
    $$u.node = newNode(READ, newLeafString(STR, $2.u.string), newLeafString(ID, $3.u.string));
}
/* Bucle while */
| WHILE '(' EXPR ')' SENTENCE
{
    $$type = AST_NODE_id;
    $$u.node = newNode(WHILE, $3.u.node, $5.u.node);
}
/* Condicional if-else */
| IF '(' EXPR ')' SENTENCE ELSE SENTENCE
{
    $$type = AST_NODE_id;
    $$u.node = newNode('g', NULL, newRoot('r', newRoot('r', NULL, newNode(IF, $3.u.node, $5.u.node)), newNode(ELSE, $3.u.node, $7.u.node)));
}
/* Condicional if */
| IF '(' EXPR ')' SENTENCE
{
    $$type = AST_NODE_id;
    $$u.node = newNode(IF, $3.u.node, $5.u.node);
};

```

# YACC

```

/* Expresiones */
EXPR
/* Igualdad lógica */
: EXPR EQ EXPR
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode(EQ, $1.u.node, $3.u.node);
}
/* Desigualdad lógica */
| EXPR NE EXPR
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode(NE, $1.u.node, $3.u.node);
}
/* And lógico */
| EXPR LAND EXPR
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode(LAND, $1.u.node, $3.u.node);
}
/* Or lógico */
| EXPR LOR EXPR
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode(LOR, $1.u.node, $3.u.node);
}
/* Menor que */
| EXPR LT EXPR
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode(LT, $1.u.node, $3.u.node);
}
/* Mayor que */
| EXPR GT EXPR
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode(GT, $1.u.node, $3.u.node);
}

```

# YACC



```

/* Suma aritmética */
| EXPR '+' EXPR
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode('+', $1.u.node, $3.u.node);
}
/* Resta aritmética */
| EXPR '-' EXPR
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode('-', $1.u.node, $3.u.node);
}
/* Multiplicación aritmética */
| EXPR '*' EXPR
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode('*', $1.u.node, $3.u.node);
}
/* División aritmética */
| EXPR '/' EXPR
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode('/', $1.u.node, $3.u.node);
}
/* Módulo */
| EXPR '%' EXPR
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode('%', $1.u.node, $3.u.node);
}
/* Potencia */
| EXPR '^' EXPR
{
    $$$.type = AST_NODE_id;
    $$$.u.node = newNode('^', $1.u.node, $3.u.node);
}

```

# YACC

```

/* Operación cociente */
|  EXPR DIV  EXPR
{
    $$ .type = AST_NODE_id;
    $$ .u.node = newNode(DIV, $1.u.node, $3.u.node);
}
/* Mantenimiento de signo */
|  '+' EXPR %prec UNARY
{
    $$ = $2;
}
/* Cambio de signo */
|  '-' EXPR %prec UNARY
{
    $$ .type = AST_NODE_id;
    $$ .u.node = newNode('-', NULL, $2.u.node);
}
|  '(' EXPR ')'
{
    $$ = $2;
}
/* Bit-and */
|  EXPR BAND EXPR
{
    $$ .type = AST_NODE_id;
    $$ .u.node = newNode(BAND, $1.u.node, $3.u.node);
}
/* Bit-or */
|  EXPR BOR EXPR
{
    $$ .type = AST_NODE_id;
    $$ .u.node = newNode(BOR, $1.u.node, $3.u.node);
}
/* Bit-not */
|  BNOT EXPR
{
    $$ .type = AST_NODE_id;
    $$ .u.node = newNode(BNOT, NULL, $2.u.node);
}

```

# YACC

```

/* Shift left */
| EXPR SL EXPR
{
    $$ .type = AST_NODE_id;
    $$ .u.node = newNode(SL, $1.u.node, $3.u.node);
}
/* Shift right */
| EXPR SR EXPR
{
    $$ .type = AST_NODE_id;
    $$ .u.node = newNode(SR, $1.u.node, $3.u.node);
}
/* Número real */
| FLOAT
{
    $$ .type = AST_NODE_id;
    $$ .u.node = newLeafNum(FLOAT, $1.u.real_value);
}
/* Número entero */
| INT
{
    $$ .type = AST_NODE_id;
    $$ .u.node = newLeafInt(INT, $1.u.int_value);
}
/* Variable */
| ID
{
    $$ .type = AST_NODE_id;
    $$ .u.node = newLeafString(ID, $1.u.string);
}
/* Seno */
| SIN '(' EXPR ')'
{
    $$ .type = AST_NODE_id;
    $$ .u.node = newNode(SIN, $3.u.node, NULL);
}
/* Coseno */
| COS '(' EXPR ')'
{
    $$ .type = AST_NODE_id;
    $$ .u.node = newNode(COS, $3.u.node, NULL);
}
/* Tangente */

```

# YACC

```
/* Tangente */  
| TAN '(' EXPR ')'  
{  
    $$ .type = AST_NODE_id;  
    $$ .u.node = newNode(TAN, $3 .u.node, NULL);  
}  
/* Logaritmo neperiano */  
| LN '(' EXPR ')'  
{  
    $$ .type = AST_NODE_id;  
    $$ .u.node = newNode(LN, $3 .u.node, NULL);  
}  
;
```

%%

# YACC

```

%%

char programName[256] = "";

/* Gestión de errores */
int yyerror(char *error)
{
    fprintf(stderr, "%s(%d): error -- %s\n", programName, yylineno, error);
    return 1;
}

int main(int argc, char *argv[])
{
    /* Comprobación de corrección de argumentos */
    if (argc < 2 || argc > 3 || (argc == 3 && strcmp("-t", argv[2]) != 0))
    {
        printf("Uso: ./ac <fichero> [-t]\n");
        exit(FILE_ERROR);
    }

    /* Nombre del fichero */
    strcpy(programName, argv[1]);

    /* Lectura del fichero */
    yyin = fopen(programName, "rb");
    if (yyin == NULL)
    {
        fprintf(stderr, "Error intentando abrir el fichero %s\n", programName);
        exit(FILE_ERROR);
    }

    /* Parseo */
    if (yyparse() != PARSE_SUCCESS)
    {
        fclose(yyin);
        fprintf(stderr, "Compilacion abortada.\n");
        exit(SYNTAX_ERROR);
    }

    fclose(yyin);

```

# YACC

```

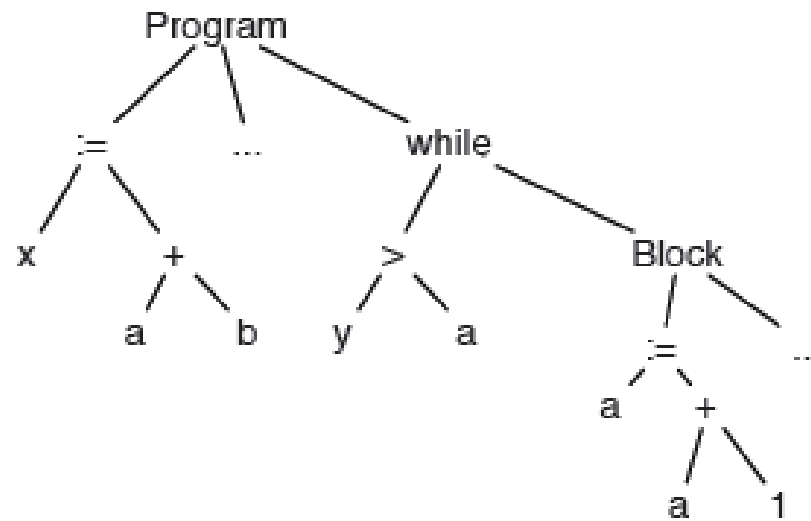
/* Recorrido del árbol */
if (ast != NULL)
{
    /* Impresión, si se ha utilizado la opción -t */
    if (argc == 3 && strcmp("-t", argv[2]) == 0)
    {
        char *treeFilename;
        mallocCheck(treeFilename, sizeof(char) * strlen(programName) + 6);
        strcpy(treeFilename, programName);
        treeFilename[strlen(programName)] = '.';
        treeFilename[strlen(programName) + 1] = 't';
        treeFilename[strlen(programName) + 2] = 'r';
        treeFilename[strlen(programName) + 3] = 'e';
        treeFilename[strlen(programName) + 4] = 'e';
        treeFilename[strlen(programName) + 5] = '\0';
        FILE *treeFile = fopen(treeFilename, "w");
        if (treeFile == NULL)
        {
            fprintf(stderr, "Error al exportar el AST.\n");
        }
        else
        {
            print_tree(treeFile, ast, 0);
        }
        fclose(treeFile);
    }
    /* Evaluación del árbol */
    process(ast);
}
else
{
    fprintf(stderr, "No hay nada que ejecutar.\n");
}

return 0;
}

```

# YACC

```
x := a + b;  
y := a * b;  
while (y > a) {  
    a := a + 1;  
    x := a + b  
}
```



AST

```

/* Creación de una hoja con valor de cadena de caracteres */
ast_t *newLeafString(unsigned tag, char *str)
{
    ast_t *res;
    mallocCheck(res, sizeof(ast_t));
    res->lineNum = (unsigned)yylineno - 1;
    res->tag = tag;
    res->u.str = str;
    return res;
}

/* Creación de una hoja con valor de número real */
ast_t *newLeafNum(unsigned tag, double dval)
{
    ast_t *res;
    mallocCheck(res, sizeof(ast_t));
    res->lineNum = (unsigned)yylineno - 1;
    res->tag = tag;
    res->u.real = dval;
    return res;
}

/* Creación de una hoja con valor de número entero */
ast_t *newLeafInt(unsigned tag, long val)
{
    ast_t *res;
    mallocCheck(res, sizeof(ast_t));
    res->lineNum = (unsigned)yylineno - 1;
    res->tag = tag;
    res->u.integer = val;
    return res;
}

```

# AST



```

/* Creación de un nodo no hoja */
ast_t *newNode(unsigned tag, ast_t *l, ast_t *r)
{
    ast_t *res;
    mallocCheck(res, sizeof(ast_t));
    res->lineNum = (unsigned)yylineno - 1;
    res->tag = tag;
    res->u.child.left = l;
    res->u.child.right = r;
    return res;
}

/* Creación-actualización de la raíz del árbol */
ast_t *newRoot(unsigned tag, ast_t *lst, ast_t *nd)
{
    if (lst == NULL) {
        if (nd == NULL) {
            return NULL;
        }
        return newNode(tag, nd, NULL);
    }
    if (nd == NULL) {
        return lst;
    }

    ast_t *tmp = lst;
    while (tmp->u.child.right != NULL) {
        tmp = tmp->u.child.right;
    }
    tmp->u.child.right = newNode(tag, nd, NULL);

    return lst;
}

```

# AST

```

/* Evaluar expresión del AST */
static symbol evaluateExpr(ast_t *node)
{
    symbol value; /* Valor de la expresión */
    symbol left; /* Valor del hijo izquierdo */
    symbol right; /* Valor del hijo derecho */

    double leftVal; /* Valor numérico del hijo izquierdo */
    double rightVal; /* Valor numérico del hijo derecho */
    switch (node->tag)
    {
        case EQ:
            if ((evaluateExpr(node->u.child.left)).value.real == (evaluateExpr(node->u.child.right)).value.real)
            {
                value.type = REAL_id;
                value.value.real = 1.0;
                return value;
            }
            else
            {
                value.type = REAL_id;
                value.value.real = 0.0;
                return value;
            }
        case NE:
            if ((evaluateExpr(node->u.child.left)).value.real != (evaluateExpr(node->u.child.right)).value.real)
            {
                value.type = REAL_id;
                value.value.real = 1.0;
                return value;
            }
            else
            {
                value.type = REAL_id;
                value.value.real = 0.0;
                return value;
            }
        case LAND:
            if ((evaluateExpr(node->u.child.left)).value.real && (evaluateExpr(node->u.child.right)).value.real)
            {
                value.type = REAL_id;

```

# AST

```

/* Evaluar nodo-sentencia del AST */
static void evaluateNode(ast_t *node)
{
    switch (node->tag)
    {
        case '=':
            edit(node->u.child.left->u.str, evaluateExpr(node->u.child.right));
            break;

        case PRINT:
            if (node->u.child.left == NULL)
            {
                symbol right = evaluateExpr(node->u.child.right);
                if (right.type == INT_id)
                {
                    printf("%ld\n", right.value.integer);
                }
                else
                {
                    printf("%g\n", right.value.real);
                }
            }
            else if (node->u.child.right == NULL)
            {
                printf("%s\n", node->u.child.left->u.str);
            }
            else
            {
                symbol right = evaluateExpr(node->u.child.right);
                if (right.type == INT_id)
                {
                    printf("%s%ld\n", node->u.child.left->u.str, right.value.integer);
                }
                else
                {
                    printf("%s%g\n", node->u.child.left->u.str, right.value.real);
                }
            }
            break;

        case READ:
            if (node->u.child.left == NULL)
            {
                symbol number;

```

# AST

```
/* Procesar un AST a partir de su raíz */  
void process(ast_t *root)  
{  
    while (root != NULL)  
    {  
        if (root->u.child.left != NULL)  
            evaluateNode(root->u.child.left);  
        root = root->u.child.right;  
    }  
}
```

# AST

```

#ifndef __AST_H__
#define __AST_H__

/* Tipo "nodo del ast" */
typedef struct ast_s {
    unsigned tag;
    unsigned lineNum;
    union {
        struct {
            struct ast_s *left, *right, *rightElse;
        } child;
        char *str;
        double real;
        long integer;
    } u;
} ast_t;

/* Funciones de ast.c */
ast_t *newLeafString(unsigned tag, char *str);
ast_t *newLeafNum(unsigned tag, double dval);
ast_t *newLeafInt(unsigned tag, long int);
ast_t *newNode(unsigned tag, ast_t *l, ast_t *r);
ast_t *newRoot(unsigned tag, ast_t *lst, ast_t *nd);

void process(ast_t *root);
void print_tree(FILE *f, ast_t *node, int space);

#endif

```

# AST

```
/* Función para obtener el valor de un símbolo de la tabla */
symbol get(char *id)
{
    int index = pos(id);
    if (index == SYMTAB_NOT_FOUND)
    {
        fprintf(stderr, "%s(%d) error -- identificador no encontrado: %s\n", programName, yylineno, id);
        exit(SYMTAB_NOT_FOUND);
    }
    return symTab[index].value;
}

/* Función para cambiar una entrada de la tabla de símbolos */
void edit(char *id, symbol value)
{
    if (symbols == 0) init();

    int index = pos(id);
    if (index == SYMTAB_NOT_FOUND)
    {
        symbols++;
        resize();

        index = freePos(id);
        symTab[index].id = id;
    }

    symTab[index].value = value;
}
```

# TABLA DE SÍMBOLOS

```

typedef struct {
    char *id;
    symbol value;
} entry;

static entry *symTab = NULL;

/* Rellenar la tabla con entradas vacías */
static void init()
{
    mallocCheck(symTab, sizeof(entry) * size);
    memset(symTab, 0, sizeof(entry) * (size_t)size);
}

/* Función de dispersión para un identificador (String) */
static int hash(char *s)
{
    int h = 0;
    int i = 0;
    for (int i = 0; i < strlen(s); i++)
    {
        h = 31*h + s[i++];
    }
    return abs(h);
}

/* Función para hallar la primera posición potencial de un identificador */
static int first(char *s)
{
    return hash(s) % size;
}

/* Función para hallar el "salto" para buscar un identificador en la tabla. */
static void step(char *s, int *pos)
{
    int step = (int)(hash(s) / size);
    if (step % 2 == 0) step++;
    *pos = (*pos + step) % size;
}

/* Función para obtener la posición de un identificador en la tabla */
static int pos(char *id)
{
    int pos = first(id);
    while (symTab[pos].id != NULL && strcmp(id, symTab[pos].id) != 0)

```

# TABLA DE SÍMBOLOS

```
/* Tipo "símbolo" para la tabla de símbolos
   (Par tipo de dato - valor del dato) */
typedef struct symbol_s {
    int type;
    union {
        double real;
        long integer;
    } value;
} symbol;

/* Declaración de funciones de symtab.c */
symbol get(char *id);
void edit(char *id, symbol value);
```

# TABLA DE SÍMBOLOS



```
* Imprimir recursivamente un árbol a partir del nodo actual */
old print_tree(FILE *f, ast_t *node, int space)

    if (node->u.child.left != NULL && node->tag != STR && node->tag != INT && node->tag != FLOAT && node->tag != ID)
        print_tree(f, node->u.child.left, space + 8);

    fprintf(f, "\n");
    for (int i = 0; i < space; i++)
        fprintf(f, " ");
    fprintf(f, "%s\n", translate(node));

    if (node->u.child.right != NULL && node->tag != STR && node->tag != INT && node->tag != FLOAT && node->tag != ID)
        print_tree(f, node->u.child.right, space + 8);
```

## REPRESENTACIÓN GRÁFICA DEL AST

```

|
    STR: Introduce una variable:
    READ
    VAR: x
ST
    STR: Introduce otra variable:
    READ
    VAR: y
ST
    VAR: z
    =
    VAR: x
    +
    VAR: y
ST
    STR: Su suma es:
    PRINT
    VAR: z
ST

```

## REPRESENTACIÓN GRÁFICA DEL AST

```
read "Introduce una variable: " x
read "Introduce otra variable: " y
z = x + y
print "Su suma es: " z
```

```
File Edit View Search Terminal Help
[2]+  Done                  gedit ejemplo.a
masha@Agamon:~/Desktop/Definitivo/glf-proyecto-master/src$ ./ac ejemplo.a -t
Introduce una variable: 4
Introduce otra variable: 2
Su suma es: 6
masha@Agamon:~/Desktop/Definitivo/glf-proyecto-master/src$
```

# EJEMPLO

FIN

