

**Centro de Enseñanza Técnica Industrial**

**Desarrollo de Software**

**Calculadora de conjuntos**

**Jesús Alberto Aréchiga Carrillo**

**22310439 6N**

**Profesor**

**Darío Yerenas**

**Junio 2025**

**Guadalajara, Jalisco**

Archivo de flex:

%%

"SetUnion"|"ClearSet"|"PrintSet"|"Showsets"|"Delete"|"Union"|"intersection"|"Sets"|"Set" {

printf("|CMD, %s|", yytext);

}

":=" { printf("|ASSIGN, %s|", yytext); }

"{" { printf("|LBRACE, %s|", yytext); }

"}" { printf("|RBRACE, %s|", yytext); }

"," { printf("|COMMA, %s|", yytext); }

";" { printf("|SEMICOLON, %s|", yytext); }

[A-Za-z][A-Za-z0-9]\* {

printf("|ID, %s|", yytext);

}

[0-9]+[A-Za-z0-9]\* {

printf("|ELEMENT, %s|", yytext);

}

[ \t]+ { /\* nada \*/ }

\n { putchar('\n'); }

. { /\* nada \*/ }

%%

int main(void) {

printf(">> Scanner iniciado. Escribe los comandos y pulsa Enter.\n");

yylex();

return 0;

}

Archivo en bison:

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\* ---- Estructura de datos para conjuntos ---- \*/

#define MAXSETS 1000

#define INITIAL\_CAPACITY 4

typedef struct {

char \*name;

char \*\*elements;

int count;

int capacity;

} Set;

/\* Arreglo global de conjuntos y contador \*/

static Set sets[MAXSETS];

static int setCount = 0;

/\* Puntero al conjunto que estamos definiendo en la regla SET … \*/

static Set \*currentSet = NULL;

/\* ---- Funciones ---- \*/

static Set \*findSet(const char \*name) {

for(int i = 0; i < setCount; i++)

if (strcmp(sets[i].name, name) == 0)

return &sets[i];

return NULL;

}

static Set \*createOrGetSet(const char \*name) {

Set \*s = findSet(name);

if (s) {

/\* Si ya existía, lo limpiamos para redefinirlo \*/

for(int i = 0; i < s->count; i++) free(s->elements[i]);

s->count = 0;

return s;

}

if (setCount >= MAXSETS) {

fprintf(stderr, "Error: número máximo de conjuntos alcanzado\n");

exit(1);

}

s = &sets[setCount++];

s->name = strdup(name);

s->capacity = INITIAL\_CAPACITY;

s->elements = malloc(s->capacity \* sizeof(char\*));

s->count = 0;

return s;

}

/\* agregar un elemento a un conjunto \*/

static void addElement(Set \*s, const char \*elem) {

if (!s) return;

/\* evitamos duplicados \*/

for(int i = 0; i < s->count; i++)

if (strcmp(s->elements[i], elem) == 0)

return;

if (s->count >= s->capacity) {

s->capacity \*= 2;

s->elements = realloc(s->elements, s->capacity \* sizeof(char\*));

}

s->elements[s->count++] = strdup(elem);

}

/\* eliminar un conjunto \*/

static void deleteSetFunc(const char \*name) {

for(int i = 0; i < setCount; i++) {

if (strcmp(sets[i].name, name) == 0) {

free(sets[i].name);

for(int j=0;j<sets[i].count;j++) free(sets[i].elements[j]);

free(sets[i].elements);

/\* compactamos el array \*/

for(int k=i;k<setCount-1;k++) sets[k]=sets[k+1];

setCount--;

return;

}

}

fprintf(stderr, "Warning: conjunto '%s' no existe (DeleteSet)\n", name);

}

/\* eliminar los elementos de un conjunto \*/

static void clearSetFunc(const char \*name) {

Set \*s = findSet(name);

if (s) {

for(int i=0;i<s->count;i++) free(s->elements[i]);

s->count = 0;

} else {

fprintf(stderr, "Warning: conjunto '%s' no existe (ClearSet)\n", name);

}

}

/\* imprimir un conjunto \*/

static void printSetFunc(const char \*name) {

Set \*s = findSet(name);

if (!s) {

fprintf(stderr, "Error: conjunto '%s' no existe (PrintSet)\n", name);

return;

}

printf("%s = { ", name);

for(int i=0;i<s->count;i++) {

printf("%s%s", s->elements[i], (i+1<s->count)?", ":" ");

}

printf("}\n");

}

/\* imprimir todos los conjuntos \*/

static void showAllSets() {

if (setCount == 0) {

printf("No hay conjuntos definidos.\n");

return;

}

for(int i=0;i<setCount;i++) {

printSetFunc(sets[i].name);

}

}

/\* union de conjunutos \*/

static void unionSetsFunc(const char \*n1, const char \*n2) {

Set \*a = findSet(n1), \*b = findSet(n2);

if (!a || !b) {

fprintf(stderr, "Error: conjuntos para unión no encontrados\n");

return;

}

printf("Union(%s,%s) = { ", n1, n2);

/\* imprimimos todos de 'a' y luego los de 'b' que no estén en 'a' \*/

int printed = 0;

for(int i=0;i<a->count;i++) {

if (printed++) printf(", ");

printf("%s", a->elements[i]);

}

for(int j=0;j<b->count;j++) {

int found = 0;

for(int i=0;i<a->count;i++)

if (strcmp(a->elements[i], b->elements[j])==0) { found=1; break; }

if (!found) {

if (printed++) printf(", ");

printf("%s", b->elements[j]);

}

}

printf(" }\n");

}

/\* interseccion de conjuntos \*/

static void intersectSetsFunc(const char \*n1, const char \*n2) {

Set \*a = findSet(n1), \*b = findSet(n2);

if (!a || !b) {

fprintf(stderr, "Error: conjuntos para intersección no encontrados\n");

return;

}

printf("Intersection(%s,%s) = { ", n1, n2);

int printed = 0;

for(int i=0;i<a->count;i++){

for(int j=0;j<b->count;j++){

if (strcmp(a->elements[i], b->elements[j])==0){

if (printed++) printf(", ");

printf("%s", a->elements[i]);

break;

}

}

}

printf(" }\n");

}

/\* Prototipos del parser \*/

extern int yylex(void);

extern int yylineno;

void yyerror(const char \*s) {

fprintf(stderr, "\*\* yyerror (linea %d): %s\n", yylineno, s);

}

%}

/\* Tokens \*/

%token SETUNION CLEARSET PRINTSET SHOWSETS DELETESET UNIONSET INTERSECTION SETS SET

%token ASSIGN LBRACE RBRACE COMMA SEMICOLON EXITCMD

%token ID ELEMENT

/\* Unión semántica \*/

%union {

char \*str;

}

/\* Asociamos tipos \*/

%type <str> ID ELEMENT

%%

program:

/\* vacío \*/

| program statement

;

statement:

instruction SEMICOLON

| EXITCMD SEMICOLON

{

exit(0);

}

| error SEMICOLON

{

yyerrok;

fprintf(stderr, "\*\* Error de sintaxis (línea %d). Ignoramos hasta ';'.\n", yylineno);

}

;

/\* ---- instrucciones ---- \*/

instruction:

/\* definición de un conjunto \*/

SET ID ASSIGN LBRACE

{

currentSet = createOrGetSet($2);

free($2);

}

element\_list RBRACE

{

printf(">> Conjunto '%s' definido.\n", currentSet->name);

currentSet = NULL;

}

| PRINTSET ID

{

printSetFunc($2);

free($2);

}

| SHOWSETS

{

showAllSets();

}

| CLEARSET ID

{

clearSetFunc($2);

free($2);

}

| DELETESET ID

{

deleteSetFunc($2);

free($2);

}

| SETUNION ID COMMA ID

{

unionSetsFunc($2, $4);

free($2); free($4);

}

| INTERSECTION ID COMMA ID

{

intersectSetsFunc($2, $4);

free($2); free($4);

}

;

/\* element\_list: va añadiendo elementos a currentSet \*/

element\_list:

ELEMENT

{

addElement(currentSet, $1);

free($1);

}

| element\_list COMMA ELEMENT

{

addElement(currentSet, $3);

free($3);

}

;

%%

int main(void) {

printf(">> Analizador iniciado. Termina instrucciones con ';', 'Exit;' para salir.\n\n");

yylineno = 1;

yyparse();

return 0;

}  
  
  
Este proyecto combina Flex y Bison para ofrecer un intérprete de línea de comandos que permite definir, manipular y mostrar “conjuntos” de cadenas en memoria. En la fase de escaneo (Flex), cada entrada del usuario se divide en tokens como SET, identificadores (ID), elementos (ELEMENT), llaves, comas y punto y coma. A cada token de tipo ID o ELEMENT se le asocia dinámicamente una cadena duplicada con strdup(), que luego el analizador sintáctico (Bison) utiliza en sus acciones semánticas. El parser reconoce construcciones como SET Nombre := { elem1, elem2, … };, PrintSet Nombre;, Showsets;, ClearSet Nombre;, DeleteSet Nombre;, SetUnion A, B; e Intersection A, B;, ejecutando la función C correspondiente para crear, vaciar, eliminar o combinar los conjuntos según las reglas de la gramática.

La estructura de datos central es un arreglo global sets[MAXSETS] de estructuras Set, cada una con un campo name (el nombre del conjunto), un puntero elements a un arreglo dinámico de cadenas, y dos enteros count y capacity para gestionar cuántos elementos hay y cuánto espacio hay reservado. Al definir un conjunto nuevo, se reserva memoria para su nombre y su array de elementos; al añadir cada elemento, se duplica la cadena y, de ser necesario, se agranda el array con realloc(). Las funciones de unión e intersección simplemente recorren los elementos de los conjuntos implicados y construyen la salida en pantalla. Finalmente, cada acción libera siempre la memoria temporal procedente del lexer (con free($n)) y, en operaciones de borrado o limpieza, también libera la memoria asociada al nombre y a los elementos del propio conjunto para evitar fugas.