UNIVERSIDAD
DE SANTIAGO
DE CHILE

Departamento de Matemática y Ciencia de la Computación

Tarea 1 - SAT Lineal

Miguel Olivares Morales
miguel.olivares@usach.cl

Benjamín Riveros Landeros
benjamin.riveros.l@usach.cl

Lógica Computacional - 22625
Licenciatura en Ciencia de la Computación

Semestre Otoño 2025

1 Introducción

El problema de determinar si las variables de una fórmula booleana pueden ser reemplazadas con valores **T** o **F** de tal forma que la fórmula de como resultado **T** se denomina problema de satisfacibilidad booleana o SAT. Si al evaluar la fórmula esta da como resultado **T**, entonces se dice que es satisfactoria.

2 Procedimiento

Las fórmulas que serán analizadas primero tendrán que ser codificadas según la siguiente gramatica:

$$\phi ::= p \mid (\neg\phi) \mid (\phi \wedge \phi)$$

Para esto usamos el siguiente esquema de traducción:

$T(p) = p$ $\qquad$ $T(\neg\phi) = \neg T(\phi)$

$T(\phi_1 \wedge \phi_2) = T(\phi_1) \wedge T(\phi_2)$ $\qquad$ $T(\phi_1 \vee \phi_2) = \neg(\neg T(\phi_1) \wedge \neg T(\phi_2))$

$T(\phi_1 \rightarrow \phi_2) = \neg(T(\phi_1) \wedge \neg T(\phi_2))$

Esto quiere decir que se analizarán fórmulas compuestas por proposiciones atómicas, negaciones de otras fórmulas y conjunciones de dos fórmulas.

Luego de codificar se tiene que transformar a su notación postfix o también llamada *notación polaca inversa* con la cual facilitará la creación de un *parse tree* para asignar valores **T** o **F** a cada nodo. Al tener el parse tree correspondiente a la fórmula que se evalúa asignamos **T** al nodo que encabeza el árbol. Esto implica asumir que la fórmula completa es verdadera y a partir de ello se puede extender esta asignación hacia los nodos hijos del árbol aplicando reglas semánticas de los conectores lógicos.

Si el nodo principal es una conjunción $\phi \wedge \psi$ entonces ϕ y ψ deben ser verdaderas. Por el contrario, si el nodo es una negación $\neg\phi$ quiere decir que la subfórmula ϕ es falsa. Este procedimiento se aplica recursivamente hasta llegar a los nodos hoja, los cuales corresponden a átomos proposicionales.

De esta forma se obtiene una asignación de valores de verdad que satisface la fórmula. En caso que las asignaciones conduzcan a una contradicción (por ejemplo, se tiene $p \equiv \mathbf{T}$ y $\neg p \equiv \mathbf{T}$) se descarta el camino recorrido o incluso puede significar que la fórmula es *insatisfacible*.

Adicionalmente para una mayor eficiencia en espacio y tiempo, detectar y reutilizar átomos proposicionales podemos construir en cambio un DAG (Directed Acyclic Graph).

2.1 Ejemplo

Dada la siguiente fórmula:

$$((p \rightarrow q) \wedge (\neg r \vee p))$$

El primer paso es aplicar la codificación mencionada anteriormente

$$\begin{aligned}
\phi &= ((p \to q) \land (\neg r \lor p)) \\
T(\phi) &= T(((p \to q) \land (\neg r \lor p))) \\
&= T(p \to q) \land T(\neg r \lor p) \\
&= \neg(T(p) \land \neg T(q)) \land \neg(\neg T(\neg r) \land \neg T(p)) \\
T(\phi) &= \neg(p \land \neg q) \land \neg(\neg\neg r \land \neg p)
\end{aligned}$$

El siguiente paso es transformar la fórmula codificada a su notación postfix, para esto hay que descomponer la fórmula en *tokens* de la siguiente manera:

$$[\neg, (, p, \land, \neg, q,), \land, \neg, (, \neg, \neg, r, \land, \neg, p,)]$$

Para transformar a su notación postfix se tiene que saber que se evalúan los operadores según la precedencia, en donde la negación tiene la mayor precedencia por lo que se evalúa primero luego de esta, la conjunción.

Token	**Acción**	**Salida**	**Stack**
$\neg$	Apilar operador		$\neg$
$($	Apilar paréntesis		$\neg$, $($
p	Agregar a salida	p	$\neg$, $($
$\land$	Apilar operador	p	$\neg$, $($, $\land$
$\neg$	Apilar operador	p	$\neg$, $($, $\land$, $\neg$
q	Agregar a salida	$p\ q$	$\neg$, $($, $\land$, $\neg$
$)$	Desapilar hasta $($	$p\ q\ \neg\ \land$	$\neg$
$\land$	Apilar operador	$p\ q\ \neg\ \land$	$\neg$, $\land$
$\neg$	Apilar operador	$p\ q\ \neg\ \land$	$\neg$, $\land$, $\neg$
$($	Apilar paréntesis	$p\ q\ \neg\ \land$	$\neg$, $\land$, $\neg$, $($
$\neg$	Apilar operador	$p\ q\ \neg\ \land$	$\neg$, $\land$, $\neg$, $($, $\neg$
$\neg$	Apilar operador	$p\ q\ \neg\ \land$	$\neg$, $\land$, $\neg$, $($, $\neg$, $\neg$
r	Agregar a salida	$p\ q\ \neg\ \land\ r$	$\neg$, $\land$, $\neg$, $($, $\neg$, $\neg$
$\land$	Apilar operador	$p\ q\ \neg\ \land\ r$	$\neg$, $\land$, $\neg$, $($, $\neg$, $\neg$, $\land$
$\neg$	Apilar operador	$p\ q\ \neg\ \land\ r$	$\neg$, $\land$, $\neg$, $($, $\neg$, $\neg$, $\land$, $\neg$
p	Agregar a salida	$p\ q\ \neg\ \land\ r\ p$	$\neg$, $\land$, $\neg$, $($, $\neg$, $\neg$, $\land$, $\neg$
$)$	Desapilar hasta $($	$p\ q\ \neg\ \land\ r\ p\ \neg\ \land\neg\neg$	$\neg$, $\land$, $\neg$
	Vaciar pila	$p\ q\ \neg\ \land\ r\ p\ \neg\ \land\neg\ \neg\ \neg\ \land\ \neg$	

3 Algoritmo

3.1 Codificación de fórmula

Algorithm 1: Algoritmo recursivo $T(\phi)$ para codificar una fórmula booleana ϕ

Input: ϕ
Output: $T(\phi)$
if ϕ *es un átomo proposicional* p **then**
  return p
end
if ϕ *es una negación* $(\neg\phi)$ **then**
  return $\neg T(\phi)$
end
if ϕ *es una conjunción* $(\phi_1 \wedge \phi_2)$ **then**
  return $T(\phi_1) \wedge T(\phi_2)$
end
if ϕ *es una disyunción* $(\phi_1 \vee \phi_2)$ **then**
  return $\neg(\neg T(\phi_1) \wedge \neg T(\phi_2))$
end
if ϕ *es una implicancia* $(\phi_1 \rightarrow \phi_2)$ **then**
  return $\neg(T(\phi_1) \wedge \neg T(\phi_2))$
end
if ϕ *tiene una fórmula no reconocida* **then**
  Reportar error: fórmula inválida
end

3.2 Conversión a Notación Postfija

Para transformar las fórmulas lógicas desde su forma infija a una notación postfija (también conocida como notación polaca inversa), se implementó un algoritmo inspirado en el *Shunting Yard Algorithm* de Edsger Dijkstra. Esta notación facilita el análisis y la evaluación de fórmulas lógicas, ya que elimina la ambigüedad del orden de operaciones y permite un procesamiento más eficiente.

Algorithm 2: Conversión a notación postfija (inspirado en Shunting Yard Algorithm)

Input: Fórmula codificada $T(\phi)$
Output: Fórmula $T(\phi)$ en notación postfija

```
Inicializar una pila vacía stack
Inicializar una cadena vacía output
for cada token en la fórmula de entrada do
    if el token es un operando then
        Agregar token a output
    end
    else if el token es un operador then
        while pila no vacía y tope operador con mayor o igual precedencia do
            Desapilar operador y agregarlo a output
        end
        Apilar el operador actual
    end
    else if el token es un paréntesis de apertura then
        Apilar el token
    end
    else if el token es un paréntesis de cierre then
        while tope de pila no es paréntesis de apertura do
            Desapilar operador y agregarlo a output
        end
        if pila vacía then
            Reportar error: paréntesis desbalanceados
        end
        else
            Desapilar paréntesis de apertura
        end
    end
end
while pila no vacía do
    if tope es paréntesis then
        Reportar error: paréntesis desbalanceados
    end
    else
        Desapilar operador y agregarlo a output
    end
end
return output
```

4 Implementación

4.1 Codificación de fórmula

Implementación de la codificación de fórmula en Lex

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include "logic.h"

    #define MAX_STACK 1024
    Node* stack[MAX_STACK];
    int top = 0;

    void push(Node* n){
        if (top < MAX_STACK){
            stack[top++] = n;
        }else{
            fprintf(stderr, "Stack overflow\n");
            exit(1);
        }
    }

    Node* pop() {
        if (top > 0) {
            return stack[--top];
        } else {
            fprintf(stderr, "Stack underflow\n");
            exit(1);
        }
    }

    void reset_stack() {
        top = 0;
    }
    %}

%option noyywrap

%%

[ \t\n]+              ;
"("                   ;
"\$"                  ;
")"                   ;

"\\rightarrow"         {
    if (top < 2) {
        fprintf(stderr, "Error: operador '\\rightarrow' requiere dos
```

```
                operandos\n");
46              exit(1);
47          }
48          Node* right = pop();
49          Node* left = pop();
50          Node* neg_right = create_not(right);
51          Node* and_node = create_op(AND, left, neg_right);
52          Node* impl_node = create_not(and_node);
53          push(impl_node);
54      }
55
56      "\\neg"                 {
57          Node* child = pop();
58          push(create_not(child));
59      }
60
61      "\\wedge"               {
62          Node* right = pop();
63          Node* left = pop();
64          push(create_op(AND, left, right));
65      }
66
67      "\\vee"                 {
68          if (top < 2) {
69              fprintf(stderr, "Error: operador '\\vee' requiere dos operandos
                    \n");
70              exit(1);
71          }
72          Node* right = pop();
73          Node* left = pop();
74          Node* neg_left = create_not(left);
75          Node* neg_right = create_not(right);
76          Node* and_node = create_op(AND, neg_left, neg_right);
77          Node* or_node = create_not(and_node);
78          push(or_node);
79      }
80
81      [a-zA-Z][a-zA-Z0-9]*(_[0-9]+|_\{[0-9]+\})?   {
82          push(create_var(yytext));
83      }
84
85      .                       { printf("Caracter no reconocido: %s\n", yytext);
            }
86
87      %%
```

4.2 Notación Postfix

Implementación de la conversión a notación postfix en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "postfix_converter.h"

#define MAX_STACK 100
#define MAX_OUTPUT 2048

Operator operators[] = {
    {"\\neg", 3, 1},
    {"\\wedge", 2, 0},
    {"\\vee", 1, 0},
    {"\\rightarrow", 0, 0},
    {NULL, 0, 0}
};

int precedence(const char* op) {
    for (int i = 0; operators[i].symbol; i++) {
        if (strcmp(op, operators[i].symbol) == 0)
        return operators[i].precedence;
    }
    return -1;
}

int is_right_associative(const char* op) {
    for (int i = 0; operators[i].symbol; i++) {
        if (strcmp(op, operators[i].symbol) == 0)
        return operators[i].right_associative;
    }
    return 0;
}

int is_operator(const char* token) {
    for (int i = 0; operators[i].symbol; i++) {
        if (strcmp(token, operators[i].symbol) == 0)
        return 1;
    }
    return 0;
}

char* convert_to_postfix(const char* input) {
    char output[MAX_OUTPUT] = "";
    const char* stack[MAX_STACK];
    int top = 0;

    char token[256];
```

```
48
49          for (int i = 0; input[i];) {
50              if (isspace(input[i]) || input[i] == '$') {
51                  i++;
52                  continue;
53              } else if (input[i] == '(' || input[i] == ')') {
54                  token[0] = input[i];
55                  token[1] = '\0';
56                  i++;
57              } else if (input[i] == '\\') {
58                  int j = 0;
59                  token[j++] = input[i++];
60                  while (isalpha(input[i])) token[j++] = input[i++];
61                  token[j] = '\0';
62              } else if (isalpha(input[i])) {
63                  int j = 0;
64                  token[j++] = input[i++];
65                  while (isalnum(input[i])) token[j++] = input[i++];
66                  if (input[i] == '_') {
67                      token[j++] = input[i++];
68                      if (input[i] == '{') {
69                              token[j++] = input[i++];
70                              while (isdigit(input[i])) token[j++] = input[i
                                    ++];
71                              if (input[i] == '}') token[j++] = input[i++];
72                      } else {
73                          while (isdigit(input[i])) token[j++] = input[i++];
74                      }
75                  }
76                  token[j] = '\0';
77              } else {
78                  token[0] = input[i++];
79                  token[1] = '\0';
80              }
81
82              if (is_operator(token)) {
83                  while (top > 0 && is_operator(stack[top - 1]) &&
84                  ((precedence(stack[top - 1]) > precedence(token)) ||
85                  (precedence(stack[top - 1]) == precedence(token) && !
                        is_right_associative(token)))) {
86                      strcat(output, stack[--top]);
87                      strcat(output, " ");
88                  }
89                  stack[top++] = strdup(token);
90              } else if (strcmp(token, "(") == 0) {
91                  stack[top++] = strdup(token);
92              } else if (strcmp(token, ")") == 0) {
93                  while (top > 0 && strcmp(stack[top - 1], "(") != 0) {
94                      strcat(output, stack[--top]);
95                      strcat(output, " ");
```

```
                }
                if (top == 0) {
                    fprintf(stderr, "Error: parentesis desbalanceados\n");
                    exit(1);
                }
                free((void*)stack[--top]);
            } else {
                strcat(output, token);
                strcat(output, " ");
            }
        }

        while (top > 0) {
            if (strcmp(stack[top - 1], "(") == 0) {
                fprintf(stderr, "Error: parentesis desbalanceados\n");
                exit(1);
            }
            strcat(output, stack[--top]);
            strcat(output, " ");
        }

        return strdup(output);
    }
```

4.3 Creación de arbol sintáctico

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "logic.h"

Node* create_var(const char* name) {
    Node* node = (Node*)malloc(sizeof(Node));
    if (!node) {
        fprintf(stderr, "Error al asignar memoria para un nodo VAR\n");
        exit(1);
    }
    node->type = VAR;
    node->var_name = strdup(name);
    node->left = node->right = NULL;
    return node;
}

Node* create_op(NodeType type, Node* left, Node* right) {
    Node* node = (Node*) malloc(sizeof(Node));
    if (!node) {
        fprintf(stderr, "Error al asignar memoria para un nodo de
            operacion\n");
        exit(1);
    }
    node->type = type;
    node->left = left;
    node->right = right;
    node->var_name = NULL;
    return node;
}

Node* create_not(Node* child) {
    Node* node = (Node*) malloc(sizeof(Node));
    if (!node) {
        fprintf(stderr, "Error al asignar memoria para un nodo NOT\n");
        exit(1);
    }
    node->type = NOT;
    node->left = child;
    node->right = NULL;
    node->var_name = NULL;
    return node;
}

Node* impl_free(Node* node) {
    if (!node) return NULL;

    if (node->type == IMPL) {
```

```
48              Node* left = impl_free(node->left);
49              Node* right = impl_free(node->right);
50              Node* neg_left = create_not(left);
51              return create_op(OR, neg_left, right);
52          } else if (node->type == AND || node->type == OR) {
53              Node* left = impl_free(node->left);
54              Node* right = impl_free(node->right);
55              return create_op(node->type, left, right);
56          } else if (node->type == NOT) {
57              Node* child = impl_free(node->left);
58              return create_not(child);
59          } else {
60              return create_var(node->var_name);
61          }
62      }
63
64      Node* copy_tree(Node* node) {
65          if (!node) return NULL;
66
67          Node* new_node = (Node*)malloc(sizeof(Node));
68          if (!new_node) {
69              fprintf(stderr, "Error al asignar memoria en copy_tree\n");
70              exit(1);
71          }
72          new_node->type = node->type;
73          new_node->var_name = node->var_name ? strdup(node->var_name) : NULL
                ;
74          new_node->left = copy_tree(node->left);
75          new_node->right = copy_tree(node->right);
76          return new_node;
77      }
78
79      Node* distribute(Node* a, Node* b) {
80          if (!a || !b) return NULL;
81
82          if (a->type == AND) {
83              return create_op(AND,
84              distribute(a->left, copy_tree(b)),
85              distribute(a->right, copy_tree(b)));
86          } else if (b->type == AND) {
87              return create_op(AND,
88              distribute(copy_tree(a), b->left),
89              distribute(copy_tree(a), b->right));
90          } else {
91              return create_op(OR, copy_tree(a), copy_tree(b));
92          }
93      }
94
95      Node* to_cnf(Node* node) {
96          if (!node) return NULL;
```

```
97
98          switch (node->type) {
99              case OR: {
100                 Node* left = to_cnf(node->left);
101                 Node* right = to_cnf(node->right);
102                 return distribute(left, right);
103             }
104             case AND: {
105                 return create_op(AND,
106                 to_cnf(node->left),
107                 to_cnf(node->right));
108             }
109             case NOT: {
110                 return create_not(to_cnf(node->left));
111             }
112             case VAR: {
113                 return create_var(node->var_name);
114             }
115             default:
116             return NULL;
117         }
118     }
119
120     void print_formula(Node* node) {
121         if (!node) return;
122
123         switch (node->type) {
124             case VAR:
125             printf("%s", node->var_name);
126             break;
127             case NOT:
128             printf("~");
129             print_formula(node->left);
130             break;
131             case AND:
132             printf("(");
133             print_formula(node->left);
134             printf(" & ");
135             print_formula(node->right);
136             printf(")");
137             break;
138             case OR:
139             printf("(");
140             print_formula(node->left);
141             printf(" | ");
142             print_formula(node->right);
143             printf(")");
144             break;
145             case IMPL:
146             printf("(");
```

```
147             print_formula(node->left);
148             printf(" -> ");
149             print_formula(node->right);
150             printf(")");
151             break;
152         }
153     }
154 
155     void free_tree(Node* node) {
156         if (!node) return;
157 
158         if (node->left) free_tree(node->left);
159         if (node->right) free_tree(node->right);
160 
161         if (node->var_name) {
162             free(node->var_name);
163             node->var_name = NULL;
164         }
165 
166         free(node);
167         node = NULL;
168     }
169 
170     static void mark_vars(Node* node, bool vars[26]) {
171         if (!node) return;
172 
173         if (node->type == VAR && node->var_name) {
174             char c = node->var_name[0];
175             if (c >= 'A' && c <= 'Z')
176             vars[c - 'A'] = true;
177         }
178 
179         mark_vars(node->left, vars);
180         mark_vars(node->right, vars);
181     }
182 
183     int get_num_vars(Node* node) {
184         bool vars[26] = {false};
185 
186         mark_vars(node, vars);
187 
188         int count = 0;
189         for (int i = 0; i < 26; i++)
190         if (vars[i]) count++;
191 
192         return count;
193     }
194 
195     bool eval_formula(Node* node, bool assignment[26]) {
196         if (!node) return false;
```

```
197
198         switch (node->type) {
199             case VAR:
200             return assignment[node->var_name[0] - 'A'];
201             case NOT:
202             return !eval_formula(node->left, assignment);
203             case AND:
204             return eval_formula(node->left, assignment) &&
205             eval_formula(node->right, assignment);
206             case OR:
207             return eval_formula(node->left, assignment) ||
208             eval_formula(node->right, assignment);
209             default:
210             return false;
211         }
212     }
213
214     bool solve_rec(Node* node, bool assignment[26], int idx, int max_vars)
        {
215         if (idx == max_vars)
216         return eval_formula(node, assignment);
217
218         assignment[idx] = false;
219         if (solve_rec(node, assignment, idx + 1, max_vars)) return true;
220
221         assignment[idx] = true;
222         if (solve_rec(node, assignment, idx + 1, max_vars)) return true;
223
224         return false;
225     }
226
227     bool solve_sat(Node* formula, int num_vars) {
228         bool assignment[26] = {false};
229         return solve_rec(formula, assignment, 0, num_vars);
230     }
```

4.4 Verificación de satisfacibilidad

```c
#include "logic.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

bool is_satisfied(Node* clause, bool* assignment, int num_vars);
bool backtrack(Node* formula, bool* assignment, int num_vars, int index);

bool is_satisfied(Node* clause, bool* assignment, int num_vars){

    if (!clause) return false;

    if (clause->type == VAR){
        int var_index = atoi(clause->var_name);
        return assignment[var_index];
    }
    else if (clause->type == NOT){
        return !assignment[atoi(clause->left->var_name)];
    }
    return false;
}

bool backtrack(Node* formula, bool* assignment, int num_vars, int index){
    if (index == num_vars){
        Node* current_clause = formula;
        while (current_clause){
            if (!is_satisfied(current_clause, assignment, num_vars)){
                return false;
            }
            current_clause = current_clause->right;
        }
        return true;
    }

    assignment[index] = true;
    if (backtrack(formula, assignment, num_vars, index + 1)) {
        return true;
    }

    assignment[index] = false;
    if (backtrack(formula, assignment, num_vars, index + 1)) {
        return true;
    }

    return false;
}
```

```
47
48      bool solve_sat(Node* formula, int num_vars){
49          bool* assignment = malloc(num_vars * sizeof(bool));
50          if (!assignment) {
51              fprintf(stderr, "Error al asignar memoria.\n");
52              return false;
53          }
54
55          for (int i = 0; i < num_vars; i++){
56              assignment[i] = false;
57          }
58
59          bool result = backtrack(formula, assignment, num_vars, 0);
60
61          free(assignment);
62          return result;
63      }
```

5 Compilación

Antes es necesario saber que para la implementación de SAT Lineal se usaron dos distribuciones distintas de Linux, Arch Linux y Fedora Linux 42 (Workstation Edition).
Para la compilación es necesario los siguientes requisitos:

5.1 Requisitos

- **gcc** - Compilador de C
  1. Arch Linux: `sudo pacman -S gcc`
  2. Fedora: `sudo dnf install gcc`
- **flex** - Generador de analizadores léxicos
  1. Arch Linux: `sudo pacman -S flex`
  2. Fedora: `sudo dnf install flex`
- **make** (Opcional) - Para automatizar la compilación
  1. Arch Linux: `sudo pacman -S make`
  2. Fedora: `sudo dnf install make`

5.2 Compilación

Hay dos métodos para compilar el programa:

1. Opción 1: Usando make

   ```
   make
   ```

   Esto genera el ejecutable llamado `tarea1.exe`

2. Opción 2: Manualmente (sin make)

   ```
   flex sat_cnf.l

   gcc -Wall -g -c sat_main.c logic.c postfix_converter.c sat_solver.c lex.yy.c

   gcc -Wall -g -o tarea1.exe sat_main.o logic.o postfix_converter.o lex.yy.o -lfl
   ```

   Esto genera el ejecutable llamado `tarea1.exe`

5.3 Ejecución

Se tendrá que preparar un archivo llamado "expresion.txt" con una o más fórmulas lógicas escritas en LaTeX, encerradas con "$" y con salto de linea entre cada una, por ejemplo:

`expresion.txt`

```
$p \wedge q$
$p \vee \neg q$
$p \wedge \neg r \rightarrow q$
```

Este formato de archivo es válido para el programa.

Luego de tener el archivo preparado se le envía como entrada estandar (stdin) de esta manera:

```
./tarea1.exe < expresion.txt
```

5.4 Limpieza

Finalmente para limpiar los archivos compilador y ejecutable:

1. Opción 1: Usando make

   ```
   make clean
   ```

2. Opción 2: Manualmente (sin make)

   ```
   rm -f *.o lex.yy.c tarea1.exe
   ```

6 Conclusiones

Hemos presentado la implementación de un algoritmo que determina si las variables de una fórmula booleana pueden ser reemplazadas con T o **F** de modo que la fórmula entregue resultado **T**, es decir si es satisfacible o no. La tarea fundamental es codificar una fórmula de lógica proposicional escrita en LaTeXde tal forma que los únicos operadores que se evalúen sean la negación y conjunción lo cual facilita la asignación de valores de verdadero o falso usando un árbol sintáctico.