Instituto Politécnico Nacional

Escuela Superior de Física y Matemáticas

T E S I S

Teoría de la computabilidad con C++ aplicada a un sistema de lógica proposicional

Que para obtener el grado de:

Licenciatura en Física y Matemáticas

Presenta:

Julio Solis Noguez

Director de tesis:

Dr. David J. Fernandez Bretón

Ciudad de México

Agradecimientos

A mis padres, Antonia Briceida Noguez Ortiz y Julio Solis Dominguez, por su apoyo constante e incondicional a lo largo de toda mi formación académica. Gracias por brindarme las herramientas, los valores y la motivación necesarios para culminar esta etapa. Su respaldo ha sido fundamental para la realización de esta tesis.

A mi tía, Sara Catalina Noguez Ortiz, por ser una presencia constante y un apoyo invaluable.

Finalmente, agradezco a mi asesor, David J. Fernández Bretón, por su paciencia y orientación a lo largo de este proceso.

CARTA CESIÓN DE DERECHOS AL IPN

Ciudad de México a 22 de Enero de 2026

El que suscribe:

C. Julio Solis Noguez con número de boleta 2018351991

Egresado del Programa Académico: Licenciatura en Física y Matemáticas

que se imparte en la Escuela Superior de Física y Matemáticas, manifiesta que es autor intelectual del presente trabajo de tesis titulado:

"Teoría de la computabilidad con C++ aplicada a un sistema de lógica proposicional" El cual fue desarrolado bajo la dirección de: Dr. David José Fernandez Bretón

Por lo anterior es mi interés manifestar que **SI** cedo los derechos del trabajo antes mencionado, al Instituto Politécnico Nacional, ello con el proósito de que lo ponga a disposición de la comunidad politécnica que requiera consultarlo con fines académicos y de investigación. Es importante aclarar que los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Dicho permiso puede ser solicitado a la siguiente dirección de correo electrónico: juliosolisnoguez@gmail.com. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Atentamente

Julio Solis Noguez

Nombre y firma del alumno

Secretaria Académica
Dirección de Educación Superior
Escuela Superior de Física y Matemáticas

90 Aniversario del Instituto Politécnico Nacional
90 Aniversario del CECyT 2 "Miguel Bernard"
30 Aniversario de la UPIITA
50 Aniversario del CICIMAR

Folio
SeAca-DES-ESFM-SAC-T-015-2026

Asunto
Autorización de impresión

Ciudad de México, 22 de enero de 2026

C. JULIO SOLIS NOGUEZ
PASANTE DE LA LICENCIATURA EN FÍSICA Y MATEMÁTICAS
PRESENTE

Sirva este medio para informarle que derivado de que ha concluido exitosamente el desarrollo de su trabajo, bajo la asesoría de la **DR. DAVID JOSÉ FERNANDO BRETÓN,** como su asesor de tesis, tengo a bien autorizar la versión final del trabajo de tesis profesional desarrollado por usted, denominado: **"Teoría de la computabilidad con C++ aplicada a un sistema de lógica proposicional".**

Es importante **integrar** en su trabajo, después de los agradecimientos, la carta de cesión de derechos del IPN, cuyo formato podrán descargar en la siguiente dirección electrónica: ***https://www.esfm.ipn.mx/estudiantes/titulacion.html***

Después de integrar la cesión de derechos, deberás añadir *este oficio de autorización de impresión*.

Posteriormente se deberá entregar a esta subdirección, la versión final digitalizada en formato PDF.

De manera adicional deberá mostrar en original para cotejo los documentos siguientes:

Acta de nacimiento, Certificado de estudios, Carta de Pasante, Carta de Liberación de Servicio Social y (CURP).

Además, debes traer **seis fotografías** tamaño **ovalo credencial** b/n papel mate **auto-adherible, fondo blanco, vestimenta formal** y comprobante de pago de acta de examen profesional.

El horario de recepción de documentación son los días lunes a viernes, en un horario de 11:00 a 14:00 h y de 16:00 a 19:00 h.
Finalmente, deberá hacer conocimiento de su fecha de realización de examen profesional, no debiendo ser esta menor a una semana posterior a la fecha de entrega y la cual deberá acordar previamente con su(s) asesor(es) y sinodales.

Sin otro particular, reciba un saludo cordial.

ATENTAMENTE
"La Técnica al Servicio de la Patria"

SEP
ESCUELA SUPERIOR DE FÍSICA Y MATEMÁTICAS
SUBDIRECCIÓN ACADÉMICA
I P N

M. en C. Erick Lee Guzmán
Subdirector Académico

ELG/mjc

2026 año de Margarita Maza

Índice general

Índice de figuras

Índice de cuadros

Índice de Códigos

Resumen

En este trabajo se presenta un estudio desde la construcción de un lenguaje formal hasta la computabilidad del mismo, no solo desde una visión teórica sino también a través de las herramientas tecnológicas.

Dentro de los primeros dos capítulos se encuentra la parte más teórica del trabajo, ya que se explica la construcción del sistema formal que se utiliza a lo largo del trabajo, empezando por la sintaxis hasta llegar a la semántica. Los siguientes dos capítulos presentan una revisión desde lo general hasta lo particular de la computabilidad del sistema formal que se construyó.

Introducción

En matemáticas, la computabilidad como área de estudio tiene un origen histórico que involucra a grandes figuras, como David Hilbert, quien sostenía que siempre debía existir un "algoritmo" capaz de resolver cualquier problema matemático. Sin embargo, fue en la década de 1930 cuando un joven Kurt Gödel, en su trabajo de doctorado, demostró que no todos los enunciados matemáticos son decidibles dentro de un sistema lógico formal, contradiciendo así las ideas del hasta entonces más prominente matemático de su época.

En tiempos más recientes, la lógica matemática ha abordado problemas de mayor complejidad, entre ellos los relacionados con la computabilidad. Existen numerosos y detallados estudios que recopilan el desarrollo de esta área, principalmente a lo largo del siglo XX. Por ello, el presente trabajo no pretende ser un estudio exhaustivo sobre la computabilidad, sino que ofrece una introducción básica al tema, con el objetivo de fundamentar los algoritmos implementados en el lenguaje de programación `C++` para este trabajo, los cuales buscan ilustrar de forma práctica algunas proposiciones que tradicionalmente se demuestran de forma teórica.

Este trabajo no es meramente un ejercicio técnico de programación, sino que busca presentar una alternativa práctica a las soluciones clásicas presentadas en textos académicos sobre computabilidad. En particular, se pone en práctica la propuesta del matemático inglés Alan Turing, quien ofreció una solución formal al problema de la computabilidad mediante sus conocidas máquinas de Turing.

Desde luego, desarrollar algoritmos que permitan demostrar proposiciones elementales en computabilidad dista mucho de construir un "solver"de problemas lógicos. De hecho, enfrentar un problema de tal magnitud mediante programación es una tarea titánica. Incluso en estos tiempos de "la inteligencia artificial", hacer que una máquina demuestre un teorema puede resultar, en algunos casos exitosa —especialmente con teoremas clásicos—, y en otros, un fracaso rotundo.

CAPÍTULO 1

Sistema formal $\mathcal{L}$

Primero se definirá el lenguaje con el que se va a trabajar, ya que usar lenguajes naturales como inglés, español o cualquier otro, es más contraproducente que benéfico, esto, porque las reglas que determinan si una secuencia de símbolos en algún alfabeto constituyen una oración gramaticalmente bien formada, son sumamente complejas y gran parte de esa complejidad es innecesaria para el trabajo que se desarrollará. Con esto en mente, las partes que formarán este lenguaje son las siguientes.

i. Alfabeto: Conjunto de símbolos.

ii. Expresiones: Cadenas de símbolos del alfabeto.

iii. Reglas: Esquemas que determinan cuando una expresión forma un enunciado en el lenguaje.

El lenguaje que se construirá a continuación, con las partes ya mencionadas está categorizado como un *Lenguaje Formal*.

1.1. Alfabeto del lenguaje del sistema formal $\mathcal{L}$

El alfabeto del lenguaje $\mathcal{L}$ estará compuesto por una cantidad infinita de símbolos, que serán representados con la literal A y un subíndice que tomará valores entre los números naturales, tal como se ve a continuación.

$$A_1, A_2, ...,$$

Los miembros del alfabeto serán identificados en adelante como **atómicos**, estarán numerados por los naturales, para dar la idea de infinitud que caracteriza al alfabeto.

Además de las expresiones **atómicas** en el alfabeto, se le sumarán los siguientes símbolos.

$$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$$

Los cuales recibirán el nombre de **conectivas lógicas**. La primera de las conectivas recibe el nombre de negación, la segunda conjunción, luego disyunción, implicación y por último doble implicación. Es importante diferenciar a la primera

conectiva, ya que la negación a diferencia de las otras es unaria, es decir sólo actúa sobre un operando, mientras las otras lo hacen sobre dos, por lo que son categorizadas como binarias.

También el alfabeto incluye a los símbolos de paréntesis, que tienen la función de agrupar expresiones, con la única finalidad de darle orden y legibilidad a lo que se escribe.

$$(,)$$

Como una nota importante, para este trabajo la ontología es irrelevante, ya que no se estudiará a los símbolos que componen el alfabeto desde lo que son y por qué lo son. Dentro del universo lingüístico del que se está hablando pueden ser cualquier cosa, como frutas, colores, conjuntos ó números, lo único importante a nivel de símbolos es su definición, como ya se hizo, pero también la forma, es decir, que no sea posible definir a ninguno de los símbolos concatenando a otros, por lo que la **conectiva lógica** $\vee$ no puede ser igual a escribir cualquier expresión como $\vee = A_{23} \longrightarrow \wedge))$.

Ya con un alfabeto, lo siguiente es crear **enunciados** en el lenguaje con los símbolos del alfabeto. La forma en la que se construirán es inductiva, a través de la aplicación recursiva de las **conectivas lógicas**.

Definición 1.1 (Enunciados en $\mathcal{L}$) *Un enunciado en el lenguaje $\mathcal{L}$, se define como:*

i. Cualquier expresión atómica.

ii. Si φ es un enunciado, entonces $\neg\varphi$ también lo es.

iii. Si φ y ψ son enunciados, entonces $\varphi * \psi$ también lo es, para $* \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$.

Cualquier enunciado que haya sido construido con la definición 1.1 recibe el nombre de **Fórmula Bien Formada**, que se abreviará como **FBF**. Para identificar si las expresiones son FBF o no en el lenguaje $\mathcal{L}$, se usará un programa en el lenguaje de programación `C++`, el cual recibirá una cadena con la expresión y retornará un mensaje que dice si la fórmula es bien formada o no.

Antes de iniciar con la implementación del código, es necesario introducir el concepto de **Notación Polaca Invertida**, que recibe el nombre por la persona que lo postuló, el filósofo polaco Jan Łukasiewicz en 1924 (Simons, 2023). Se necesita de esta notación, para eliminar los paréntesis en las expresiones, ya que a nivel computacional trabajar con ellos complica el código. Este tipo de notación es equivalente a leer un árbol en orden post-fijo, es decir, se tiene que recorrer el árbol rama por rama con una prioridad específica, la cual se puede resumir como:

i. Ve a la izquierda.

ii. Ve a la derecha.

iii. Ve a la raíz.

Siempre se tiene que recorrer el árbol con esta prioridad , sin cambiar o saltarse alguno de los puntos enlistados. Usando la expresión $((A_1 \wedge A_2) \vee A_3) \Rightarrow \neg A_4$, como ejemplo. En la figura 1.1 se observa el árbol que representa la manera como se construye dicha expresión.

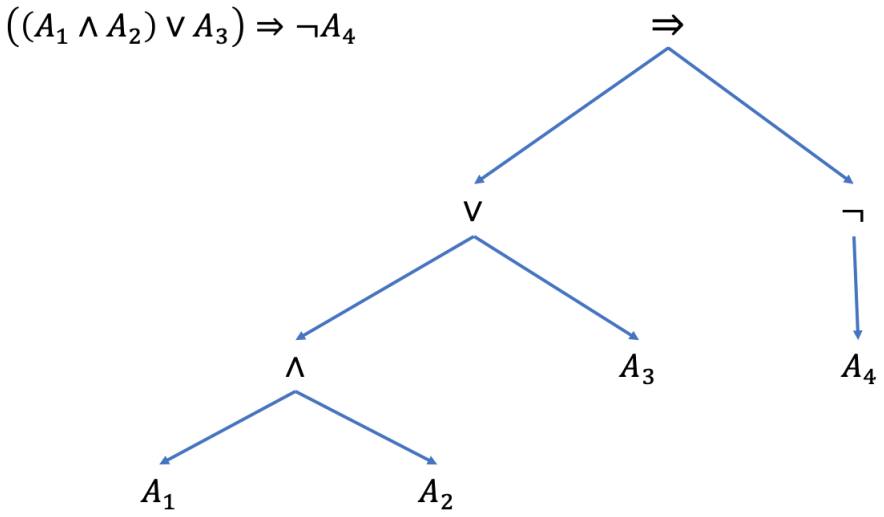

Figura 1.1: Árbol de la expresión $((A_1 \wedge A_2) \vee A_3) \Rightarrow \neg A_4$

Leyendo el árbol en orden post-fijo, el primer paso es recorrerlo rama por rama hacia la izquierda, hasta que ya no haya adónde moverse, llegando a la copa del árbol a la izquierda, que se muestra en la figura 1.2

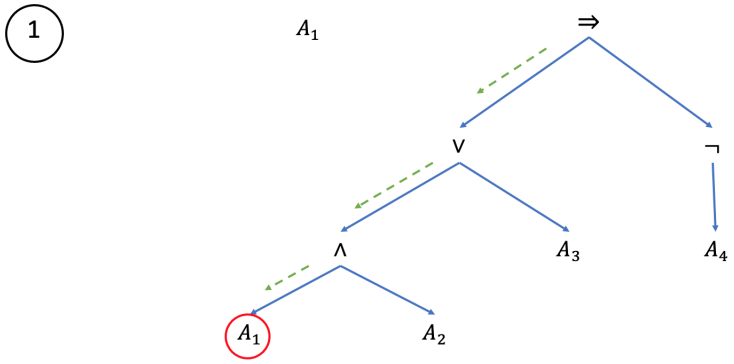

Figura 1.2: Primer paso del orden post-fijo.

El siguiente paso en prioridad es ir a la siguiente rama derecha, pero como no hay ninguna, entonces es necesario regresar a la raíz de la rama actual, una vez en ella, como sí hay una rama a la derecha, entonces se tiene que ir a esta, como se muestra en la figura 1.3.

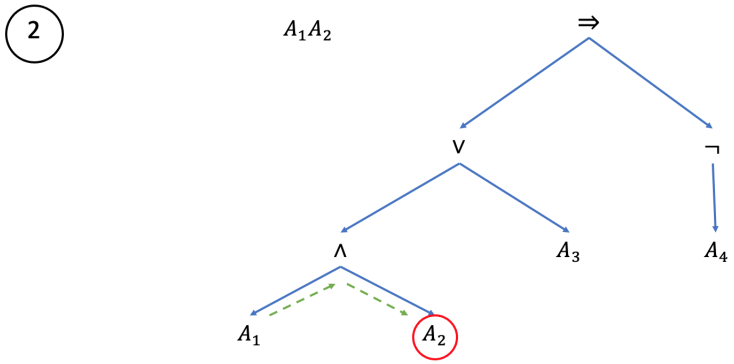

Figura 1.3: Segundo paso del orden post-fijo.

Como en la rama actual también se ha alcanzado la copa del árbol, sólo queda regresar a la raíz. Como la rama a la izquierda de la raíz ya fue visitada la raíz toma el valor actual, como se muestra en la figura 1.4.

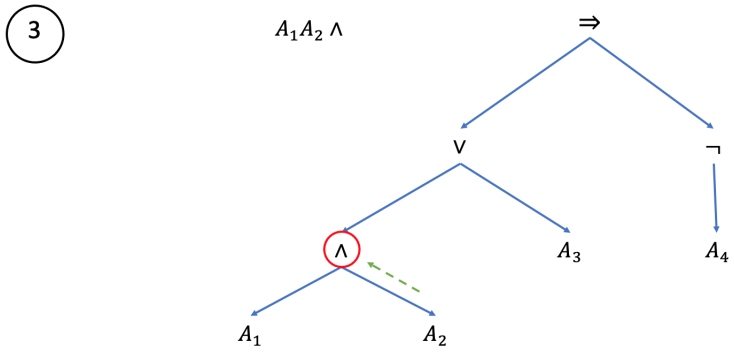

Figura 1.4: Tercer paso del orden post-fijo.

Como en el orden de prioridad ya no hay adónde moverse, sólo queda regresar a la raíz de la rama actual, pero como esta también tiene una rama a la derecha, por orden de prioridad primero se tiene que visitar esta, cambiando el valor actual a este, como se muestra en la figura 1.5.

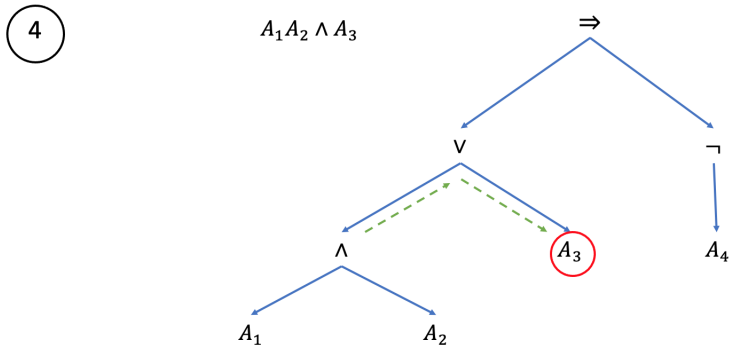

Figura 1.5: Cuarto paso del orden post-fijo.

Al ya no haber más ramas a la izquierda o derecha de la actual, sólo queda regresa a su raíz, de la cual ya se visitaron todos sus ramas a la izquierda y derecha, por lo que se fija el valor actual en esta rama, tal como se ve en la figura 1.6.

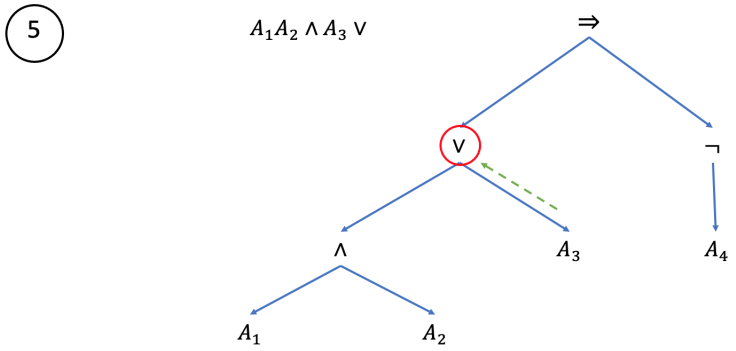

Figura 1.6: Quinto paso del orden post-fijo

De la rama actual sólo queda ir a su raíz, pero como todavía hay una rama sin visitar de esta, es necesario moverse hasta la última de sus ramas, llegando de nuevo a la copa del árbol, cambiando el valor actual al que se ve en la figura 1.7.

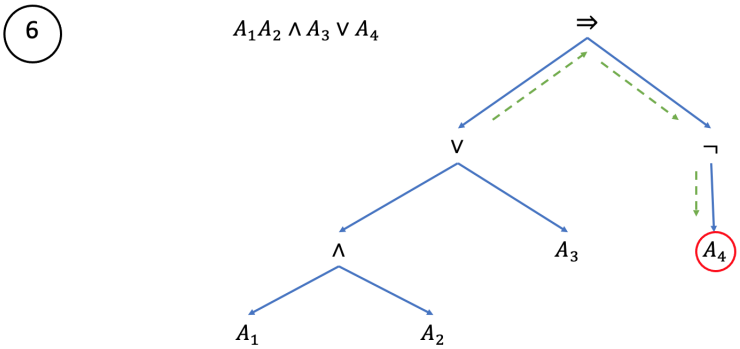

Figura 1.7: Sexto paso del orden post-fijo.

Como en el paso anterior se ha alcanzado la copa del árbol sólo se puede regresar a su raíz, logrando que el valor actual cambie al que se muestra en la figura 1.8.

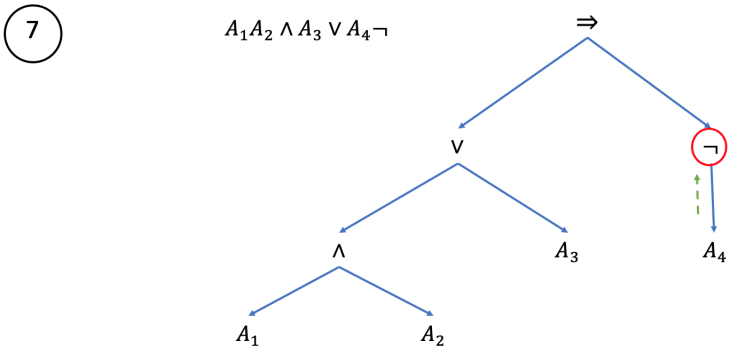

Figura 1.8: Séptimo paso del orden post-fijo.

Por último, sólo resta regresar a la raíz de la rama actual. Obteniendo la expresión $A_1A_2 \wedge A_3 \vee A_4\neg \Rightarrow$, que se puede observar en la figura 1.9.

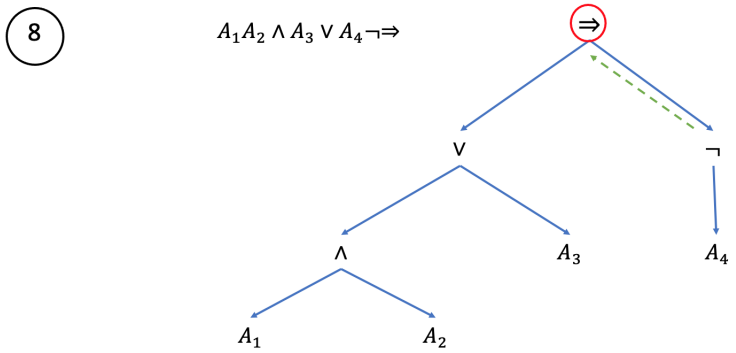

Figura 1.9: Octavo paso del orden post-fijo.

La expresión $A_1A_2 \wedge A_3 \vee A_4\neg \Rightarrow$, que se obtiene al final del proceso es el equivalente en notación polaca inversa de $((A_1 \wedge A_2) \vee A_3) \Rightarrow \neg A_4$, que fue elegida como ejemplo.

Aunque construir el árbol de una expresión es un proceso visual y fácil de entender, al programarlo no se puede ejecutar exactamente así, ya que la computadora sólo recibirá una cadena de texto que podrá leer caracter por caracter y no un diagrama con el que pueda interactuar. Para ejecutar la tarea de transformar una expresión en notación in-fija a una equivalente en post-fija, se utilizará una estructura de datos proveniente de la librería estándar del lenguaje de programación `C++`, que recibe el nombre de `Stack`, que en su traducción al español es conocida como `Pila`. Este tipo de estructura se caracteriza por ser **LIFO**, que es acrónimo de **Last in, First out**, lo que es ilustrativo, ya que el único elemento almacenado en ella que se puede consultar es el último apilado, prácticamente funciona como una pila de objetos en la vida real, sólo la puedes deshacer quitando primero lo que esté hasta arriba, de lo contrario, esta se derrumbaría.

Se usará la `pila` para almacenar las ramas que necesitan guardarse al hacer el recorrido en post-orden, ya que como se vio en el ejemplo, al hacer este tipo de lectura hay momentos en los que se tiene que seguir avanzando hasta llegar a una rama donde ya no es posible seguir avanzando y hay que volver a la anterior, a este tipo de procesos en computación se les conoce como `backtraking`, y este tipo de estructura de datos resulta primordial para hacerlos.

La `pila` que se va a utilizar en este proceso, será regida por el siguiente conjunto de reglas.

i. Si se encuentra una expresión atómica, entonces esta se imprime directamente.

ii. Si se encuentra un paréntesis izquierdo, entonces este se tiene que apilar.

iii. Si se encuentra un paréntesis derecho, se tienen que sacar todos los elementos de la pila e imprimirlos, hasta encontrar el primer paréntesis izquierdo.

iii. Si se encuentra algún operador lógico, entonces se saca e imprime todo lo que esté en la pila, hasta encontrar un operador con menor precedencia.

Como nota importante, en el tercer inciso de las instrucciones se menciona la precedencia de operadores, por lo que importa destacar que, así como en la aritmética, los operadores lógicos tienen una precedencia, que en orden de prioridad es la siguiente.

i. Negación ($\neg$).

ii. Disyunción ($\wedge$).

iii. Conjunción ($\vee$).

iv. Implicación ($\Rightarrow$) y Doble implicación ($\Leftrightarrow$).

El inciso iv. de la lista anterior es particular, ya que no se hará una diferencia en la jerarquía de los operadores implicación y doble implicación.

Una vez enlistadas las reglas que determinan cuándo apilar o no un carácter, de acuerdo a si es un paréntesis, operador o expresión atómica, se presenta en el código 1.1 que transforma expresiones de notación in-fija (normal), a notación post-fija (polaca inversa).

Es importante resaltar que para evitar los subíndices que tienen las expresiones atómicas en el lenguaje, el programa sólo utiliza las letras en mayúsculas del alfabeto inglés y para los operadores, las palabras en inglés **no**, **and**, **or**, **then** y **iff**, que representan a la negación, disyunción, conjunción, implicación y doble implicación respectivamente, ya que los símbolos que se han utilizado para representarlos no se encuentran en las distribuciones de teclados estandarizados por las normas `ISO` y `ANSI`.

Código 1.1: Función que transforma expresiones de notación in-fija a post-fija.

```
vector<string> infixtopostfix(vector<string> tokens)
{
    vector<string> postfix;
    stack<string> pila;
    for(int i = 0; i < tokens.size(); i++) {
        if(isLetter(tokens[i])) {
            postfix.push_back(tokens[i]);
        } else if (tokens[i] == "(") {
            pila.push(tokens[i]);
        } else if (tokens[i] == ")") {
            while (!pila.empty() && pila.top() != "(") {
                postfix.push_back(pila.top());
                pila.pop();
            }
            if (!pila.empty()) {
                pila.pop();
            } else {
                cout << "Error: Parentesis no balanceados." <<
                    endl;
                return postfix;
            }
        } else if(isOperator(tokens[i])) {
            while (!pila.empty() && isOperator(pila.top())) {
                if (logicPrecedence(pila.top()) >= \
                logicPrecedence(tokens[i])) {
                    postfix.push_back(pila.top());
                    pila.pop();
                } else {
                    break;
                }
            }
            pila.push(tokens[i]);
```

```
32          }
33      }
34      while (!pila.empty()) {
35          postfix.push_back(pila.top());
36          pila.pop();
37      }
38      return postfix;
39  }
```

La lógica detrás del código 1.1 es recorrer un `vector` de caracteres, entrada por entrada y ejecutar las instrucciones que se enlistaron anteriormente, haciendo uso de las estructuras de control de flujo `if`, `else if` y `else`. En el lenguaje de programación donde se desarrolló el código 1.1, la estructura de `pila` es parte de la librería estándar `stack`, por lo que sólo hace falta llamarla para, con la función `push`, poder apilar caracteres o `pop` para desapilarlos.

Anteriormente se dijo que la función `infixtopolish()` en el código 1.1, lee la estructura de datos propia de las librerías estandarizadas del lenguaje de programación `C++` llamada `vector`, el cual es un arreglo que almacena datos de forma consecutiva. Se recurrió a esta estructura de datos porque las cadenas de texto guardan a los espacios como un carácter en sí mismo, y en ocasiones la lógica del programa se puede ver afectada cuando el usuario introduce espacios de más o menos entre los caracteres alfanuméricos. Para pasar la cadena de texto a un vector sin espacios se creo la función `tokenizador()`, presentada en el código 1.2, que ejecuta el proceso que se puede ver en diagrama en la figura 1.10.

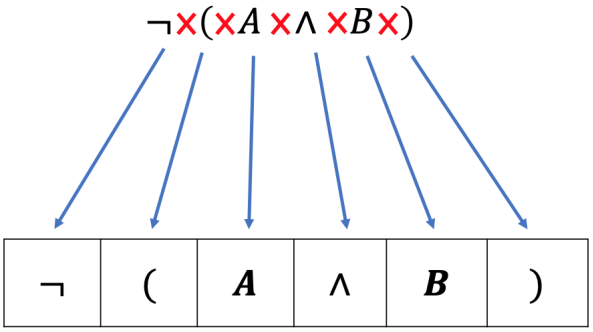

Figura 1.10: Diagrama de procesamiento de expresiones

Eliminar espacios en una cadena no es una tarea tan sencilla, pero para hacerlo el código 1.2 usa una técnica de procesamiento de cadenas de texto común, conocida como `tokenizing`, que consiste en separar a la cadena de texto en las sub-cadenas que el usuario necesite, para después almacenarlas en un `vector` o cualquier otro tipo de arreglo.

La condición que se implementa en el código 1.2 para sólo guardar paréntesis, operadores y expresiones atómicas en el proceso de `tokenizing`, se dio a través de `expresiones regulares`, en particular la siguiente

```
[A-Z]|no|and|or|then|iff|\\(|\\))
```

Donde:

i. `[A-Z]` detecta a los caracteres de las letras del alfabeto inglés, en mayúsculas.

ii. `|` es el operador lógico or.

iii. `and` detecta la sub-cadena ”and” dentro de la cadena de texto.

iv. `or` detecta la sub-cadena ”or” dentro de la cadena de texto.

v. `then` detecta la sub-cadena ”then” dentro de la cadena de texto.

vi. `\\(` detecta los paréntesis izquierdos.

vii. `\\)` detecta los paréntesis derechos.

De esta forma, la función `tokenizador()`, en el código 1.2 procesa la cadena de texto, para eliminar todos los espacios dentro de esta.

Código 1.2: Función que tokeniza una expresión.

```
vector<string> tokenizador(string exp)
{
    regex pattern("([A-Z]|no|and|or|then|iff|\\(|\\))");
    vector<string> tokens;
    sregex_iterator it(exp.begin(), exp.end(), pattern);
    sregex_iterator end;
    while (it != end) {
        tokens.push_back(it->str());
        ++it;
    }
    return tokens;
}
```

Para terminar con la explicación del código 1.1, se expone el código 1.3, con el cual se le termina de definir la sintaxis del lenguaje $\mathcal{L}$, en el programa desarrollado.

Código 1.3: Funciones auxiliares.

```
bool isOperator(string op)
{
    return (op == "no" || op == "and" || op == "or" \
            || op == "then" || op == "iff");
}
bool isLetter(string palabra)
{
    return (palabra.size() == 1 && isalpha(palabra[0]));
}
int logicPrecedence(string op)
{
    if (op == "no") {
        return 3;
    } else if (op == "and") {
        return 2;
    } else if (op == "or") {
        return 1;
    } else if ((op == "then") || (op == "iff")) {
        return 0;
    } else  {
        return -1;
    }
}
```

Una vez que se puede convertir una cadena de notación in-fija a post-fija, con la función `infixtopostfix()`, en el código 1.1, ya es más fácil evaluar si la expresión es o no una FBF. Para eso se hizo la función `isWFF()`, que se presenta en el código 1.4.

Código 1.4: Función que evalúa si una expresión es una FBF.

```
void isWFF(vector<string> exp)
{
    stack<bool> pila;
    for (string token : exp) {
        if (isLetter(token)) {
            pila.push(true);
        } else if (isOperator(token)) {
            if(token == "no") {
                if(pila.empty()) {
                    cout << "La expresion no es una formula \
                    bien formada" << endl;
                    return;
                }
                pila.pop();
                pila.push(true);
            } else {
                if (pila.size() < 2) {
                    cout << "La expresion no es una formula \
                    bien formada" << endl;
                    return;
                }
                pila.pop();
                pila.pop();
                pila.push(true);
            }
        }
    }
    if(pila.size()==1) {
        cout << "La expresion es una formula \
        bien formada" << endl;
        return;
    } else {
        cout << "La expresion no es es una formula \
        bien formada" << endl;
        return;
    }
}
```

La lógica detrás de la función `isWFF()`, en el código 1.4, es similar a la que se usó en `infixtopolish()`. La idea es ir evaluando las sub-expresiones que componen a la original, con ayuda de una `pila` y si en esta evaluación se cae en uno de los casos error, como paréntesis desequilibrados u operadores binarios que no tienen uno de los operandos, entonces la expresión no puede ser FBF. Para hacer esto, cada que se detecta una expresión atómica, se le asigna el valor booleano verdadero, si es un operador, entonces se pregunta si este es el operador negación, ya que si lo es, se pregunta si la pila tiene un elemento, en caso contrario la expresión no puede ser una FBF.

1.2. Validez de los enunciados del lenguaje

Una vez que se puede saber si un enunciado es una FBF o no, el siguiente paso es verificar la validez del enunciado, la cual recaerá completamente en las expresiones atómicas que lo componen, ya que son elementales en el sentido de que cualquiera puede ser o no válida, al igual que independiente, siempre que su validez no impone restricciones a la de ninguna otra. Para formalizar esto último, se enuncia la definición 1.2 a continuación.

Definición 1.2 (Interpretación) *Una interpretación en el lenguaje $\mathcal{L}$ es una función σ que asigna un valor en el conjunto $\{V, F\}$ a cualquier expresión atómica.*

A través de la función σ definida anteriormente, la validez de una expresión atómica A se define como la función ρ^σ tal que:

$$\rho^\sigma(A) = \sigma(A)\ ,\ \sigma(A) \in \{V, F\}$$

Para el caso de los enunciados que no son expresiones atómicos, pero sí FBF que contienen dos expresiones atómicas y una conectiva lógica, se definen las siguientes reglas semánticas, a través de funciones que les asignan valores en el conjunto $\{V, F\}$.

$$\varphi_\neg(A) = \begin{cases} F & si \quad \rho^\sigma(A) = V \\ V & si \quad \rho^\sigma(A) = F \end{cases}$$

$$\varphi_\wedge(A, B) = \begin{cases} V & si \qquad \rho^\sigma(A) = V \ y\ \rho^\sigma(B) = V \\ F & \text{en otro caso} \end{cases}$$

$$\varphi_\vee(A, B) = \begin{cases} F & si \qquad \rho^\sigma(A) = F \ y\ \rho^\sigma(B) = F \\ V & \text{en otro caso} \end{cases}$$

$$\varphi_\Leftrightarrow(A, B) = \begin{cases} V & si \qquad (\rho^\sigma(A) = V \ y\ \rho^\sigma(B) = V)\ o\ (\rho^\sigma(A) = F \ y\ \rho^\sigma(B) = F) \\ F & \text{en otro caso} \end{cases}$$

De esta forma, para poder establecer la validez de una FBF que tiene la estructura $\neg A$ ó $A * B$, con $* \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$, se extiende la definición de la función ρ^σ, a la siguiente.

$$\rho^\sigma(\neg A) = \varphi_\neg(\rho^\sigma(A))$$
$$\rho^\sigma(A * B) = \varphi_*(\rho^\sigma(A), \rho^\sigma(B))$$

Para verificar si una FBF A compuesta sólo por las expresiones atómicas $a_1, a_2, ..., a_n$ es válida, la extensión de la función ρ^σ ya no es tan sencilla como cuando se tienen enunciados con sólo dos expresiones atómicas. La herramienta más utilizada en la literatura del tema son las **tablas de verdad** que se definen de la siguiente forma.

Definición 1.3 (Tabla de verdad) *Una tabla de verdad es un arreglo rectangular, donde las primeras n columnas representan a las expresiones atómicas, que componen a un enunciado, las 2^n filas son todas las posibles interpretaciones que se les puede dar y las columnas restantes representan las evaluaciones de la fórmula para obtener el valor de verdad que tiene cada fila.*

De esta forma, bajo la definición 1.3, si un renglón es verdadero se puede concluir que bajo esa interpretación específica es verdadero y en caso contrario, no lo es.

Por ejemplo, la tabla de verdad para un tipo de FBF ya conocida, como $A \wedge B$, es la que se puede observar en la tabla 1.2

A	B	$A \wedge B$
V	V	V
V	F	F
F	V	F
F	F	F

Cuadro 1.1: Tabla de verdad de $A \wedge B$

.

Como se puede observar, la tabla 1.2 es equivalente a la función $f_\wedge$, pero presentada en un arreglo rectangular y no como una función. Pero si se tomara como ejemplo el enunciado $\neg(A_3 \wedge A_1) \Rightarrow (A_1 \vee A_2)$, que ya no tiene sólo a dos expresiones atómicas, la tabla de verdad se vería de la siguiente forma.

A_1	A_2	A_3	$A_3 \wedge A_1$	$\neg(A_3 \wedge A_1)$	$A_1 \vee A_2$	$\neg(A_3 \wedge A_1) \Rightarrow (A_2 \vee A_1)$
V	V	V	V	F	V	V
V	V	F	F	V	V	V
V	F	V	V	F	V	V
V	F	F	F	V	V	V
F	V	V	F	V	V	V
F	V	F	F	V	V	V
F	F	V	F	V	F	F
F	F	F	F	V	F	F

Cuadro 1.2: Tabla de verdad de $\neg(A_3 \wedge A_1) \Rightarrow (A_1 \vee A_2)$

.

Siguiendo la definición 1.3 se hizo un programa en el lenguaje de programación `C++`, que se presenta en el código 1.5. La lógica detrás de este, es generar un `vector` de valores booleanos, es decir ceros y unos, que tenga longitud igual a la cantidad de expresiones atómicas en el enunciado que se está evaluando.

El objetivo de este vector, es que cada valor represente a los elementos en el conjunto $\{V, F\}$, para asignárselos a las expresiones atómicas y de esta forma poder recorrer todas las posibles interpretaciones, o en esta lógica, las posibles combinaciones binarias de n-dígitos.

Para hacer la evaluación del enunciado en la interpretación dictada por el vector de valores booleanos, se creo la función`val()`, presentada en el código 1.5, que recibe como parámetro el enunciado ya en notación Polaca Inversa y el vector booleano y retorna un valor booleano que representa a algún elemento en el conjunto $\{V, F\}$.

Código 1.5: Función que calcula el valor de verdad de un enunciado.

```
void TruthTable(vector<string> exp, int numVar)
{
    vector<vector<bool> > truthTable;
    vector<bool> intrs(numVar, false);
    do {
        vector<bool> row = intrs;
        truthTable.push_back(row);
        bool result = val(exp, intrs);
        for (bool i : intrs) {
            if(i == false) {
                cout << "F ";
            } else {
                cout << "V ";
            }
        }
        if(result == false) {
            cout << "| F " << endl;
        } else {
            cout << "| V " << endl;
        }
        for (int i = 0; i < numVar; i++) {
            if (intrs[i]) {
                intrs [i] = false;
            } else {
                intrs [i] = true;
                break;
            }
        }
    } while (count(intrs.begin(), intrs.end(), false) < numVar);
}
```

La lógica que sigue la función `val()`, es similar a lo que ya se había hecho anteriormente en este trabajo. Crear una pila, en la cual se irán acumulando valores que se evaluarán conforme las reglas definidas lo indiquen. La forma en la que se hizo la evaluación, es recorriendo el vector que contiene al enunciado en notación Polaca Inversa y siempre que el valor sea una literal se le asignó el valor que le correspondía del arreglo booleano, en caso de que el valor fuera un conector lógico, entonces se operó con este, usando los valores apilados y luego se apiló el resultado y así hasta recorrer todo el arreglo del enunciado.

Código 1.6: Función que calcula el valor de verdad de un enunciado.

```
bool val(vector<string> exp, vector<bool> intrs)
{
    stack<bool> pila;
```

```
    for (string token : exp) {
        if (isLetter(token)) {
            int index = token[0] - 'A';
            pila.push(intrs[index]);
        } else if (isOperator(token)) {
            if(token == "no") {
                if(pila.empty()) {
                    cout << "Error: No hay suficientes \
                            operandos" << endl;
                    exit;
                }
                bool operando = pila.top();
                pila.pop();
                pila.push(!operando);
            } else {
                if (pila.size() < 2) {
                    cout << "Error: No hay suficientes \
                            operandos" << endl;
                    exit;
                }
                bool operando2 = pila.top();
                pila.pop();
                bool operando1 = pila.top();
                pila.pop();
                if(token == "and") {
                    pila.push(operando1 && operando2);
                } else if (token == "or") {
                    pila.push(operando1 || operando2);
                } else if (token == "then") {
                    pila.push(!operando1 || operando2);
                } else if (token == "iff") {
                    pila.push((!operando1 || operando2) && \
                              (!operando2 || operando1));
                }
            }
        }
    }
    return pila.top();
}
```

Para hacer la evaluación de los conectores lógicos, en el lenguaje de programación, los casos de la conjunción y disyunción fueron sencillos, ya que $\wedge$ y $\vee$ tienen equivalencias directas en el lenguaje de programación `C++`, que son los operadores && y ||, cuya función es operar valores booleanos y retornar el valor correspondiente a la operación, siguiendo las definiciones de las funciones $f_\wedge$ y $f_\vee$, previamente enunciadas en este trabajo. Sin embargo, los casos de la implicación y doble implicación no tienen una equivalencia directa, por lo que es necesario usar las equivalencias lógicas para la implicación y doble implicación $\neg A \vee B$ y $(\neg A \vee B) \wedge (\neg B \vee A)$, respectivamente.

Para poder usar las equivalencias lógicas anteriormente mencionadas, es necesario hacer una prueba formal para mostrar que efectivamente son equivalentes a las conectivas de implicación y doble implicación. La mejor forma que se tiene hasta el momento de hacer una prueba de este tipo es a través de las tablas de verdad, ya que si se logra mostrar que,

$$(A \Rightarrow B) \Leftrightarrow (\neg A \vee B)$$

Entonces se habrá mostrado que ambos enunciados son lógicamente equivalentes, lo mismo para el caso de la doble implicación, si se muestra que,

$$(A \Rightarrow B) \Leftrightarrow ((\neg A \vee B) \wedge (\neg B \vee A))$$

También se mostraría que ambos enunciados son equivalentes. Empezando por la primera de las equivalencias, la tabla de verdad, mostrada en la tabla 1.4, presenta la prueba de que $(A \Rightarrow B) \Leftrightarrow (\neg A \vee B)$.

A	B	$\neg A$	$\neg B$	$\neg A \vee B$	$A \Rightarrow B$	$(\neg A \vee B) \Leftrightarrow (A \Rightarrow B)$
V	V	F	F	V	V	V
V	F	F	V	F	F	V
F	V	V	F	V	V	V
F	F	V	V	V	V	V

Cuadro 1.3: Tabla de verdad de $(A \Rightarrow B) \Leftrightarrow (\neg A \vee B)$

Luego, la tabla 1.4 presenta la tabla de verdad que muestra la equivalencia entre los enunciados $(A \Rightarrow B) \Leftrightarrow ((\neg A \vee B) \wedge (\neg B \vee A))$.

A	B	$\neg A$	$\neg B$	$\neg A \vee B$	$\neg B \vee A$	$A \Leftrightarrow B$	$(\neg A \vee B)$ $\wedge$ $(\neg B \vee A)$	$(A \Leftrightarrow B)$ $\Leftrightarrow$ $((\neg A \vee B)$ $\wedge(\neg B \vee A))$
V	V	F	F	V	V	V	V	V
V	F	F	V	F	V	F	F	V
F	V	V	F	V	F	F	F	V
F	F	V	V	V	V	V	V	V

Cuadro 1.4: Tabla de verdad de $(A \Leftrightarrow B)$ y $((\neg A \vee B) \wedge (\neg B \vee A))$

Las tablas de verdad, aunque útiles, presentan un problema computacional grande, ya que aunque para casos de dos o tres expresiones atómicas, son de gran utilidad, cuando el número de estas empieza a crecer, tanto a mano como en el ordenador, la complejidad de los cálculos crece exponencialmente, para ser precisos, en el orden de 2^n. Con la intención de no trivializar este hecho, pero tampoco profundizar en la teoría de la complejidad, propia de las ciencias computacionales, se puede calcular esta cantidad, pensando que se tienen que iterar 2^n renglones, que contienen las diferentes interpretaciones de las expresiones atómicas.

1.3. El Argumento y su validez

Desde los trabajos de Aristóteles, donde señalaba que la verdad no puede ser anterior a las cosas, el estudio de la argumentación fue constituyente del área. Para introducir este tema primero se presentará en el siguiente ejemplo, clásico en la literatura del tema.

Todos los humanos son mortales,

Sócrates es un humano,

Por lo tanto Sócrates es mortal.

Los primeros dos enunciados, del argumento ejemplo, reciben el nombre de premisas y el último de conclusión. Para que el argumento sea válido, se debe mostrar que desde las premisas se pueden llegar a la conclusión a través de una serie de deducciones lógicas, que parten del hecho de que todos los humanos efectivamente son mortales, porque al final de sus vidas tendrán que morir, como regla natural, validando así a la primera premisa, luego Sócrates es un humano y por lo tanto morirá al final de su vida, como dicta la naturaleza, por último, como morirá, eso lo convierte en mortal.

Para formalizar la definición del concepto de argumento se presenta la definición 1.4, a continuación.

Definición 1.4 (Argumento) *Un argumento en el lenguaje $\mathcal{L}$ es un conjunto de enunciados pertenecientes al lenguaje, que recibe el nombre de premisas, y un enunciado destacado, llamado conclusión. Toda esta información se enuncia de forma simbólica de la siguiente forma.*

$$\frac{\Gamma}{\therefore \ \psi}$$

Donde Γ es el conjunto de premisas y ψ es la conclusión.

Para probar la validez de un argumento, la idea es comprobar si en todas las posibles interpretaciones que tienen tanto las premisas como la conclusión, siempre se mantienen válidas, esta es la idea que se refleja en la definición 1.5.

Definición 1.5 (Validez de un argumento) *Un argumento*

$$\frac{\Gamma}{\therefore \ \psi}$$

es válido, si para toda interpretación σ, se cumple que si $\rho^{\sigma}(\varphi) = V$ para toda $\varphi \in \Gamma$, entonces $\rho^{\sigma}(\psi) = V$. De forma sintética, la notación que se utiliza para expresar este hecho será la siguiente.

$$\Gamma \models \psi$$

*Que se lee como ψ es **consecuencia lógica** de Γ.*

Si para un enunciado compuesto con más de dos expresiones atómicas la complejidad de operar la tabla de verdad se volvía de orden exponencial, el esfuerzo de verificar interpretación por interpretación la validez del argumento se convierte en una tarea titánica, por lo que es necesario buscar nuevas formas para ejecutar las pruebas de validez. Esta nueva forma de revisar la validez de tanto enunciados como argumento, se le conoce como demostración formal y en la siguiente sección se presentará la definición de este concepto.

1.4. Demostración formal

Mostrar que un argumento es válido, siguiendo la definición 1.5 técnicamente es ejecutar múltiples tablas de verdad en paralelo, por lo que surge la necesidad de un nuevo concepto para ejecutar las pruebas de validez para los argumentos. A este nuevo concepto se le dará el nombre de demostración formal y se enuncia en la definición 1.6.

Definición 1.6 (Demostración formal) *Sea S un sistema formal. Si Γ es un conjunto de enunciados y ψ un enunciado, la demostración de que se puede concluir ψ a partir de Γ en S, es una secuencia finita de enunciados $\varphi_1, \varphi_2, ..., \varphi_n$, que satisfacen las siguientes propiedades:*

i. $\varphi_n = \psi$.

ii. Para todo $i \in \{1, 2, ..., n\}$ *ocurre alguno de los siguientes incisos:*

(1) $\varphi_i \in \Gamma$.

(2) φ_i *es un axioma de* S.

(3) φ_i *se sigue de alguna de las reglas de inferencia de* S*, aplicada a algunos enunciados entre* $\varphi_1, \varphi_2, ..., \varphi_{i-1}$

Para expresar de forma sintética lo anterior, la notación que se usará es la siguiente.

$$\Gamma \vdash \psi$$

Que simplemente se lee como, ψ *es* ***consecuencia deductiva*** *de* Γ *en* S.

El primero de los conceptos que salta a la vista, en la definición 1.6 es el de sistema formal, que se entenderá de aquí en adelante como en la definición 1.7.

Definición 1.7 (Sistema Formal) *Un sistema formal se conforma de:*

i. Un conjunto de enunciados del lenguaje, tales que siempre son válidos en el mismo, cuya función es tomarlos como premisas para futuros argumentos. A este tipo particular de enunciados se les llamará en adelante axiomas.

ii. Un conjunto de reglas de inferencia, que establecen cómo algunos enunciados se pueden inferir de otros.

Bajo la definición 1.7, la interpretación intuitiva de la definición 1.6 es que las demostraciones son una serie finita de enunciados, que pueden ser premisas, axiomas ó se siguen directamente de la aplicación de alguna regla de inferencia sobre cualquiera de los enunciados anteriores. En el caso particular del lenguaje formal que se está estudiando, las reglas de inferencia tradicionalmente utilizadas son las siguientes.

Reglas de Inferencia Lógica

1. Modus Ponens (M.P.)

$$\begin{array}{l} P \Rightarrow Q \\ P \\ \hline \therefore Q \end{array}$$

2. Modus Tollens (M.T.)

$$\begin{array}{l} P \Rightarrow Q \\ \neg Q \\ \hline \therefore \neg P \end{array}$$

3. Silogismo Hipotético (S.H.)

$$\begin{array}{l} P \Rightarrow Q \\ Q \Rightarrow R \\ \hline \therefore P \Rightarrow R \end{array}$$

4. Silogismo Disyuntivo (S.D.)

$$\begin{array}{l} P \vee Q \\ \neg P \\ \hline \therefore Q \end{array}$$

5. Dilema Constructivo (D.C.)

$$\begin{array}{l} (P \Rightarrow Q) \wedge (R \Rightarrow S) \\ P \vee R \\ \hline \therefore Q \vee S \end{array}$$

6. Dilema Destructivo (D.D.)

$$\begin{array}{l} (P \Rightarrow Q) \wedge (R \Rightarrow S) \\ \neg Q \vee \neg S \\ \hline \therefore \neg P \vee \neg R \end{array}$$

7. Simplificación (Simp.)

$$\begin{array}{l} P \wedge Q \\ \hline \therefore P \end{array}$$

8. Conjunción (Conj.)

$$\begin{array}{l} P \\ Q \\ \hline \therefore P \wedge Q \end{array}$$

9. Adición (Ad.)

$$\begin{array}{l} P \\ Q \\ \hline \therefore P \vee Q \end{array}$$

La selección de estas reglas de inferencia no es arbitraria, más bien, fueron seleccionadas porque gracias a su validez verificable, en tablas de verdad, permiten hacer deducciones lógicas bajo la composición de enunciados, premisas y axiomas.

De forma intuitiva, se pueden tomar las reglas de inferencia anteriores, como lo que en las ciencias computacionales se les conoce como macros, que son una serie de instrucciones que se agrupan en una sola estructura, para que sea ejecutada mediante una sola orden de ejecución, es decir, sólo basta con hacer referencia a alguna de las reglas de inferencia, en una demostración para justificar algún paso en la demostración.

Para este punto, se vuelve más claro que una demostración formal será una secuencia finita de enunciados que guardan una relación lógica entre ellos, los cuales pueden ser de muchas formas, pero siempre el último de estos tiene que ser la conclusión del argumento, para poder afirmar con certeza que es válido. Para ejemplificar, se enuncia el siguiente ejemplo.

> *Ya sea que se aumenten los impuestos o si aumentan los gastos, entonces el límite de la deuda se incrementa,*
>
> *Si los impuestos aumentan, entonces el costo de recaudar impuestos*

aumenta,

Si un aumento en los gastos implica que el gobierno toma prestado más dinero, entonces si se aumenta el límite de la deuda, entonces las tasas de interés aumentan,

Si los impuestos no aumentan y el costo de recaudar impuestos no aumenta, entonces si se aumenta el límite de la deuda, entonces el gobierno toma prestado más dinero,

El costo de recaudar impuestos no aumenta,

O bien las tasas de interés no aumentan o el gobierno no toma prestado más dinero,

Por lo tanto, o bien el límite de la deuda no se incrementa o los gastos no aumentan.

Para empezar con la demostración, el primer paso es traducir los enunciados que conforman las premisas y conclusión del argumento en símbolos del lenguaje formal.

Ya sea que se aumenten los impuestos o si aumentan los gastos, entonces el límite de la deuda se incrementa,

Ya sea que A_1 *o si* A_2*, entonces* A_3*,*

Si los impuestos aumentan, entonces el costo de recaudar impuestos aumenta,

Si A_1 *, entonces* A_4*,*

Si un aumento en los gastos implica que el gobierno toma prestado más dinero, entonces si se aumenta el límite de la deuda, entonces las tasas de interés aumentan,

Si A_2 *implica que el* A_5*, entonces si* A_3*, entonces* A_6*,*

Si los impuestos no aumentan y el costo de recaudar impuestos no aumenta, entonces si se aumenta el límite de la deuda, entonces el gobierno toma prestado más dinero,

Si no A_1 *y no* A_4*, entonces si* A_3*, entonces* A_5*,*

El costo de recaudar impuestos no aumenta,

No A_4*,*

O bien las tasas de interés no aumentan o el gobierno no toma prestado más dinero,

No A_6 *o no* A_5*,*

Por lo tanto, o bien el límite de la deuda no se incrementa o los gastos no aumentan.

Por lo tanto, no A_3 *o no* A_2*,*

Una vez que se ha concluido con la traducción de los enunciados que conforman las premisas y conclusión del argumento, resta traducir a símbolos los operadores que aparecen, en palabras, a lo largo del mismo.

Ya sea que se aumenten los impuestos o si aumentan los gastos, entonces el límite de la deuda se incrementa,

$A_1 \vee (A_2 \Rightarrow A_3)$,

Si los impuestos aumentan, entonces el costo de recaudar impuestos aumenta,

$A_1 \Rightarrow A_4$,

Si un aumento en los gastos implica que el gobierno toma prestado más dinero, entonces si se aumenta el límite de la deuda, entonces las tasas de interés aumentan,

$(A_2 \Rightarrow A_5) \Rightarrow (A_3 \Rightarrow A_6)$,

Si los impuestos no aumentan y el costo de recaudar impuestos no aumenta, entonces si se aumenta el límite de la deuda, entonces el gobierno toma prestado más dinero,

$(\neg A_1 \wedge \neg A_4) \Rightarrow (A_3 \Rightarrow A_5)$,

El costo de recaudar impuestos no aumenta,

$\neg A_4$,

O bien las tasas de interés no aumentan o el gobierno no toma prestado más dinero,

$\neg A_6 \vee \neg A_5$,

Por lo tanto, o bien el límite de la deuda no se incrementa o los gastos no aumentan.

$\therefore \neg A_3 \vee \neg A_2$

De esta forma, la traducción que tiene el argumento en símbolos del lenguaje formal es la siguiente.

$$\begin{array}{l} A_1 \vee (A_2 \Rightarrow A_3) \\ A_1 \Rightarrow A_4 \\ (A_2 \Rightarrow A_5) \Rightarrow (A_3 \Rightarrow A_6) \\ (\neg A_1 \wedge \neg A_4) \Rightarrow (A_3 \Rightarrow A_5) \\ \neg A_4 \\ \neg A_6 \vee \neg A_5 \\ \hline \therefore \neg A_3 \vee \neg A_2 \end{array}$$

Con el argumento expresado en términos del lenguaje formal, solo resta hacer su demostración.

Demostración:

(1) $A_1 \vee (A_2 \Rightarrow A_3)$	*(Premisa)*
(2) $A_1 \Rightarrow A_4$	*(Premisa)*
(3) $(A_2 \Rightarrow A_5) \Rightarrow (A_3 \Rightarrow A_6)$	*(Premisa)*
(4) $(\neg A_1 \wedge \neg A_4) \Rightarrow (A_3 \Rightarrow A_5)$	*(Premisa)*

(5) $\neg A_4$	*(Premisa)*
(6) $\neg A_6 \vee \neg A_5$	*(Premisa)*
(7) $\neg A_1$	*(M.T. ente 2 y 5)*
(8) $A_2 \Rightarrow A_3$	*(S.D. entre 1 y 7)*
(9) $\neg A_1 \wedge \neg A_4$	*(Conj. entre 5 y 7)*
(10) $A_3 \Rightarrow A_5$	*(M.P. entre 4 y 9)*
(11) $A_2 \Rightarrow A_5$	*(S.H. entre 8 y 10)*
(12) $A_3 \Rightarrow A_6$	*(M.P. entre 3 y 11)*
(13) $(A_3 \Rightarrow A_6) \wedge (A_2 \Rightarrow A_5)$	*(Conj. entre 11 y 12)*
(14) $\neg A_3 \vee \neg A_2$	*(D.D. entre 6 y 13)*

□

Por cuestiones de órden y legibilidad de la demostración, la estructura que se siguió en el ejemplo anterior, y en adelante, consiste en colocar las premisas del argumento en los primeros renglones, especificando, usualmente, a la derecha de las mismas, que lo son. Una vez que ya se enunciaron las premisas, en los siguientes renglones se deben escribir los resultados de aplicar, como mejor convenga, las diferentes reglas de inferencia, siempre especificando cuales fueron las que se aplicaron y a cuáles enunciados se les aplicaron. Por último, el enunciado que se encuentre al final de todos los demás tiene que ser la conclusión del argumento, mostrando así que se puede llegar a ella a través de deducciones lógicas bien definidas.

Aunque este proceso hasta cierto punto algorítmico de mostrar la validez de un argumento se siente lo suficientemente robusto como para con él mostrar la validez de cualquier argumento, no siempre las nueve reglas de inferencia son suficientes para hacerlo, como se observa a continuación.

> *Si trabajo, entonces gano dinero, pero si estoy inactivo, entonces me divierto.*
>
> *O trabajo o estoy inactivo.*
>
> *Sin embargo, si trabajo, entonces no me divierto, mientras que si estoy inactivo, entonces no gano dinero.*
>
> *Por lo tanto, me divierto si y solo si no gano dinero.*

Siguiendo un proceso similar al anterior, primero hay que traducir los enunciados que componen a las premisas y conclusión de la siguiente forma.

> *Si trabajo, entonces gano dinero, pero si estoy inactivo, entonces me divierto.*
>
> *Si* A_1*, entonces* A_2*, pero si* A_3*, entonces* A_4*.*
>
> *O trabajo o estoy inactivo.*
>
> *O* A_1 *o* A_3*.*
>
> *Sin embargo, si trabajo, entonces no me divierto, mientras que si estoy inactivo, entonces no gano dinero.*

Sin embargo, si A_1*, entonces no* A_4*, mientras que si* A_3*, entonces no* A_2*.*

Por lo tanto, me divierto si y solo si no gano dinero.

Por lo tanto, A_4 *si y solo si no* A_2*.*

Luego, traduciendo los operadores lógicos a los símbolos que los representan en el lenguaje, considerando al pero si y mientras que, como sinónimos del operador conjunción, se obtiene lo siguiente.

Si trabajo, entonces gano dinero, pero si estoy inactivo, entonces me divierto.

$(A_1 \Rightarrow A_2) \wedge (A_3 \Rightarrow A_4)$

O trabajo o estoy inactivo.

$A_1 \vee A_3$

Sin embargo, si trabajo, entonces no me divierto, mientras que si estoy inactivo, entonces no gano dinero.

$(A_1, \Rightarrow \neg\ A_4) \wedge (A_3 \Rightarrow \neg\ A_2)$

Por lo tanto, me divierto si y solo si no gano dinero.

$\therefore A_4 \Leftrightarrow \neg\ A_2$

Por lo que la forma simbólica del argumento sería la siguiente.

$$\begin{array}{l} (A_1 \Rightarrow A_2) \wedge (A_3 \Rightarrow A_4) \\ A_1 \vee A_3 \\ \underline{(A_1 \Rightarrow \neg A_4) \wedge (A_3 \Rightarrow \neg A_2)} \\ \therefore A_4 \Leftrightarrow \neg A_2 \end{array}$$

En este ejemplo, notar la parte problemática del enunciado es evidente, ya que en las nueve reglas de inferencia anteriores, no había ninguna que hiciera referencia al operador bicondicional, por lo que sería bueno contar con más reglas, que incluyan casos como el ya mencionado. Es en este punto, donde se introducen las **Reglas de Remplazo**, que reciben este nombre porque definen equivalencias de enunciados específicos, tal como se enuncia a continuación.

Reglas de Remplazo

1. Los teoremas de De Morgan (De M)

$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$
$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$

2. Conmutatividad (Conm.)

$P \wedge Q \equiv Q \wedge P$
$P \vee Q \equiv Q \vee P$

3. Asociatividad (Aso.)

$(P \wedge Q) \wedge R \equiv P \wedge (Q \wedge R)$
$(P \vee Q) \vee R \equiv P \vee (Q \vee R)$

4. Distributividad (Dist.)

$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$
$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$

5. Doble Negación (D.N.)

$\neg\neg P \equiv P$

6. Contraposición (Cont.)

$P \Rightarrow Q \equiv \neg Q \Rightarrow \neg P$

7. Implicación Material (I.M.)

$P \Rightarrow Q \equiv \neg Q \vee P$

8. Equivalencia Material (E.M.)

$P \Leftrightarrow Q \equiv (P \Leftrightarrow Q) \wedge (Q \Leftrightarrow P)$
$P \Leftrightarrow Q \equiv (P \wedge Q) \vee (\neg P \wedge \neg Q)$

9. Exportación (Exp.)

$(P \wedge Q) \Rightarrow R \equiv P \Rightarrow (Q \Rightarrow R)$

10. Tautología (Tau.)

$P \wedge P \equiv P$
$P \vee P \equiv P$

Como se dijo anteriormente, las **reglas de remplazo**, permiten en palabras llanas canjear un enunciado específico por otro lógicamente equivalente, que en principio facilita el trabajo al ejecutar una demostración. Para sintetizar esta idea en un sólo símbolo se usa el operador $\equiv$.

Una vez que se han enunciado las reglas de remplazo, la demostración del argumento que se usó para ejemplificar se puede construir de la siguiente manera.

Demostración:

(1) $(A_1 \Rightarrow A_2) \wedge (A_3 \Rightarrow A_4)$	*(Premisa)*
(2) $A_1 \vee A_3$	*(Premisa)*
(3) $(A_1 \Rightarrow \neg A_4) \wedge (A_3 \Rightarrow \neg A_2)$	*(Premisa)*
(4) $A_2 \vee A_4$	*(D.C. entre 1 y 2)*
(5) $\neg A_4 \vee \neg A_2$	*(D.C. entre 2 y 3)*

(6) $A_4 \Rightarrow \neg A_2$	*(I.M. en 5)*
(7) $\neg\neg A_2 \vee A_4$	*(D.N. en 4)*
(8) $\neg A_2 \Rightarrow A_4$	*(I.M. en 7)*
(9) $(A_4 \Rightarrow \neg A_2) \wedge (\neg A_2 \Rightarrow A_4)$	*(Conj. entre 6 y 8)*
(10) $A_4 \Leftrightarrow \neg A_2$	*(E.M. en 9)*

□

Concluyendo así con la demostración del argumento, haciendo evidente la necesidad de las reglas de remplazo antes enunciadas, ya que sin ellas no se hubiera podido aplicar la implicación material al quinto renglón , la doble negación en el cuarto y la equivalencia material en el noveno.

Aunque pareciera que con las reglas de inferencia y remplazo, es posible mostrar la validez de un argumento, de una forma relativamente fácil, simplemente aplicando iterativamente algunas de estas reglas, hasta llegar a la conclusión, la verdad es que no todos los argumentos en el lenguaje se pueden demostrar válidos, por lo que es necesario crear un método para identificarlos.

Para empezar a construir un método que permita demostrar cuando un argumento es inválido, es necesario recordar la definición 1.5, donde se enuncia que un argumento es válido siempre que bajo cualquier interpretación la valuación de los enunciados que lo conforman son siempre válidos. En términos de las tablas de verdad, todos los renglones de la última columna tienen que tener el valor V, del conjunto $\{V, F\}$, luego, sólo es necesario encontrar un renglón cuya entrada en la última columna tiene el valor F, probando así, que al menos existe una interpretación para la cual el enunciado no es válido. Para ilustrar este método se propone el siguiente ejemplo.

> *Si el senador vota en contra de la iniciativa, entonces se opondrá a penas más fuertes para evasores fiscales.*
>
> *Si el senador es un evasor fiscal, entonces se opondrá a penas más fuertes para evasores fiscales.*
>
> *Por lo tanto, si el senador vota en contra de la iniciativa, es un evasor fiscal.*

Siguiendo un proceso como el que se ha utilizado en los dos ejemplos anteriores, la traducción simbólica del ejemplo es la siguiente.

> *Si el senador vota en contra de la iniciativa, entonces se opondrá a penas más fuertes para evasores fiscales.*
>
> $A_1 \Rightarrow A_2$
>
> *Si el senador es un evasor fiscal, entonces se opondrá a penas más fuertes para evasores fiscales.*
>
> $A_3 \Rightarrow A_2$
>
> *Por lo tanto, si el senador vota en contra de la iniciativa, es un evasor fiscal.*
>
> $\therefore A_1 \Rightarrow A_3$

Demostración:

Se quiere mostrar que

$$\begin{array}{l} A_1 \Rightarrow A_2 \\ \underline{A_3 \Rightarrow A_2} \\ \therefore A_1 \Rightarrow A_3 \end{array}$$

De esta forma si se toman los valores $A_1 = V,\ A_2 = V,\ A_3 = F$ *se sigue que*

A_1	A_2	$A_1 \Rightarrow A_2$
V	V	V

A_2	A_3	$A_3 \Rightarrow A_2$
V	F	V

A_1	A_3	$A_1 \Rightarrow A_3$
V	F	F

De esta forma, bajo la interpretación $A_1 = V,\ A_2 = V,\ A_3 = F$*, el argumento se invalida.*

□

Antes de cerrar la sección sobre demostraciones, es necesario definir otros dos conceptos, que serán de utilidad en las siguientes dos secciones, tautología y teorema. Para comenzar la definición 1.8, formalizar el concepto de tautología

Definición 1.8 *Un enunciado* φ *es una tautología si y solo si* $\emptyset \models \varphi$*, que por simplicidad se escribe como* $\models \varphi$*.*

La definición 1.4 conceptualiza la idea de que, sin importar el conjunto de premisas que se escojan, bajo cualquier interpretación el enunciado es válido. Una idea similar se sigue en la definición de teorema, pero basada en el concepto de demostración formal.

Definición 1.9 *Un enunciado* φ *es un teorema si y solo si* $\emptyset \vdash \varphi$*, que por simplicidad se escribe como* $\vdash \varphi$*.*

De esta forma, un enunciado es un teorema, si para cualquier conjunto de premisas en el sistema formal existe una demostración de validez del enunciado.

1.5. Correctud lógica del sistema $\mathcal{L}$

El teorema de correctud lógica conforma parte fundamental del siguiente concepto a estudiar, que es el de completitud lógica. En este se demuestra que si todas las fórmulas bien formadas tienen una demostración, entonces son tautologías, lo cual se enuncia formalmente en el teorema 1.1.

Teorema 1.1 (Correctud Lógica) *Para toda φ FBF, en un sistema formal S.*

$$Si \vdash \varphi, \text{ entonces } \models \varphi.$$

El teorema 1.1 enuncia que, siempre que sea posible demostrar que una FBF es válida formalmente, entonces tiene que ser una tautología, pero si la situación se presenta al revés, es decir, si se tiene una tautología, entonces ¿siempre será posible encontrar una demostración?, la respuesta está incluida en el teorema de completitud lógica, que se presenta a continuación.

1.6. Completitud lógica del sistema $\mathcal{L}$

El teorema de completitud incluye la implicación del teorema de correctud lógica, y enuncia que esta es equivalente a mostrar que si una FBF es una tautología, entonces existe una demostración que la prueba. Formalmente, esto se presenta en el teorema 1.2

Teorema 1.2 (Completitud Lógica) *Para toda φ FBF, en un sistema formal S,*

$$\vdash \varphi \text{ si y solo si } \models \varphi.$$

El teorema 1.2 es sumamente importante, ya que muestra que existe una equivalencia entre probar que un argumento es válido bajo la definición 1.5 y hacer una demostración del argumento, con la definición 1.6.

El gran problema del sistema que se ha construido hasta el momento es que este no es completo, aunque sí es correcto, por lo que, para seguir con el trabajo se vuelve necesario buscar un lenguaje para el cual, definiendo una sintaxis y semántica precisas, sea posible mostrar que es completo, y sobre este objetivo se fundamenta el siguiente capítulo.

1.7. Alfabeto del lenguaje del sistema formal $\mathcal{L}^*$

Como el alfabeto del lenguaje del sistema formal $\mathcal{L}$, el que se utilizará para $\mathcal{L}*$ estará compuesto por una cantidad infinita de símbolos, que serán representados con la literal A y un subíndice que tomará valores entre los números naturales, tal como se ve a continuación.

$$A_1, A_2, ...,$$

Las únicas conectivas lógicas que se utilizarán para el lenguaje serán la negación y la implicación.

$$\neg, \Rightarrow$$

De igual forma, el alfabeto incluye los símbolos de paréntesis, para seguir dando orden y legibilidad a lo que se escribe.

$$(,)$$

1.8. Axiomas del sistema forma $\mathcal{L}^*$

Como parte del sistema deductivo con el que se va a trabajar, se definen los siguientes tres axiomas.

i. (A-1) $P \Rightarrow (Q \Rightarrow P)$

ii. (A-2) $(P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \Rightarrow Q) \Rightarrow (P \Rightarrow R))$

iii. (A-3) $(\neg P \Rightarrow \neg Q) \Rightarrow (Q \Rightarrow P)$

1.9. Modus Ponens: Regla de inferencia del sistema formal $\mathcal{L}^*$

Como regla de inferencia, sólo se usará Modus Ponens, asegurando que un enunciado condicional y su antecedente pueden mostrar la validez del consecuente, que en símbolos se expresa de la siguiente manera.

$$\begin{array}{l} P \Rightarrow Q \\ \underline{P \qquad} \\ Q \end{array}$$

Una vez que ya se ha dado una definición del lenguaje formal y el sistema deductivo que se va a utilizar, resta probar que hay una equivalencia entre los lenguajes $\mathcal{L}$ y $\mathcal{L}^*$, para lo cual se mostrará que se pueden deducir las reglas de inferencia definidas en el capítulo anterior.

1.10. Completitud del sistema $\mathcal{L}^*$

En el quehacer matemático, demostrar que los argumentos de la forma **Si** A**, entonces** B es algo recurrente, y poder ejecutar esta tarea se sustenta en el teorema de la deducción, que se presenta a continuación.

Teorema 1.3 (Teorema Deductivo) *Para cualquier conjunto* Γ *de fórmulas bien formadas, del lenguaje* $\mathcal{L}^*$ *y cualesquiera dos fórmulas bien formadas* A *y* B*,*

$$\Gamma \cup \{A\} \vdash B \text{ si y solo si } \Gamma \vdash (A \Rightarrow B).$$

Demostración:

Primero, mostrando la necesidad, se supone que $\Gamma \cup \{A\} \vdash B$*, entonces existe una demostración formal de* B *a partir de* $\Gamma \cup \{A\}$*, que consiste de* $\varphi_1, ..., \varphi_n$ *fórmulas bien formadas. De esta forma, procediendo por inducción sobre* $i \in \{1, ..., n\}$*.*

(1) Si $i = 1$*, la demostración solo consiste de una FBF* φ_1*, que bajo la definición 1.6 es un axioma, pertenece a* Γ *ó es* A*, por lo que* $\varphi_1 \in \Gamma \cup \{A\} \cup \{A - 1, A - 2, A - 3\}$*, obteniendo dos casos.*

(1.1) *Si* $\varphi_1 \in \Gamma \cup \{A-1, A-2, A-3\}$. *Al aplicarle Modus Ponens a la fórmula* $\varphi_1 \Rightarrow (A \Rightarrow \varphi_1)$, *que sigue el axioma* $A-1$, *se sigue que,*

$$\begin{array}{l} \varphi_1 \Rightarrow (A \Rightarrow \varphi_1) \\ \varphi_1 \\ \hline A \Rightarrow \varphi_1. \end{array}$$

(1.2) *Si* $\varphi_1 = A$, *entonces se tiene que probar que* $\Gamma \vdash (A \Rightarrow A)$.

i. *Usando el axioma A-2, con* $P = A$, $Q = (A \Rightarrow A)$, $R = A$, *se sigue que,*

$$(A \Rightarrow ((A \Rightarrow A) \Rightarrow A)) \Rightarrow ((A \Rightarrow (A \Rightarrow A)) \Rightarrow (A \Rightarrow A)).$$

ii. *Usando el axioma A-1, pero con* $P = A$ *y* $Q = (A \Rightarrow A)$.

$$A \Rightarrow ((A \Rightarrow A) \Rightarrow A).$$

iii. *Entonces, aplicando Modus Ponens a las dos fórmulas en los incisos anteriores, se sigue que,*

$$(A \Rightarrow (A \Rightarrow A)) \Rightarrow (A \Rightarrow A).$$

iv. *Luego, usando el axioma A-1, con* $P = Q = A$.

$$A \Rightarrow (A \Rightarrow A).$$

v. *De esta forma, aplicando Modus Ponens a los incisos iii y iv, se sigue que,*

$$A \Rightarrow A.$$

Por lo que queda demostrado que $\vdash (A \Rightarrow A)$ *y de esta forma que* $\Gamma \vdash (A \Rightarrow A)$.

De esta forma queda demostrado que $\Gamma \vdash (A \Rightarrow \varphi_1)$.

(2) *Como hipótesis inductiva, se supondrá que* $\Gamma \vdash (A \Rightarrow \varphi_j)$ *para todo* $1 < j < i$.

(3) *Suponiendo que* φ_i *pertenece a la demostración* $\varphi_1, ..., \varphi_n$. *De nuevo se tienen dos casos.*

(3.1) *Si* $\varphi_i \in \Gamma \cup \{A\} \cup \{A-1, A-2, A-3\}$, *la demostración sería análoga a la del inciso anterior.*

(3.2) *Si* φ_i *es la conclusión de la aplicación de Modus Ponens. Existen dos fórmulas bien formadas* φ_k *y* φ_t *tales que* $1 < k < t < i$, *para las cuales se cumple 1.1,*

$$\begin{array}{l} \varphi_t \\ \varphi_k \\ \hline \varphi_i \end{array} \tag{1.1}$$

Por lo que necesariamente $\varphi_t = \varphi_k \Rightarrow \varphi_i$.
Por otro lado, bajo la hipótesis inductiva se cumple 1.2,

$$\begin{array}{l} \Gamma \vdash (A \Rightarrow \varphi_t) \\ \Gamma \vdash (A \Rightarrow \varphi_k) \end{array} \tag{1.2}$$

Por lo tanto, se sigue 1.3,

$$\begin{array}{l} \Gamma \vdash (A \Rightarrow (\varphi_k \Rightarrow \varphi_i)) \\ \Gamma \vdash (A \Rightarrow \varphi_k) \end{array} \tag{1.3}$$

Tomando en cuenta la fórmula $(A \Rightarrow (\varphi_k \Rightarrow \varphi_i)) \Rightarrow ((A \Rightarrow \varphi_k) \Rightarrow (A \Rightarrow \varphi_i))$, *se sigue fácilmente que cumple con el esquema del axioma* $A-2$ *y por lo tanto,*

$$\Gamma \vdash (A \Rightarrow (\varphi_k \Rightarrow \varphi_i)) \Rightarrow ((A \Rightarrow \varphi_k) \Rightarrow (A \Rightarrow \varphi_i)). \tag{1.4}$$

De esta forma, aplicando Modus Ponens a las fórmulas 1.3 y 1.4 se sigue 1.5,

$$\frac{A \Rightarrow (\varphi_k \Rightarrow \varphi_i) \Rightarrow (A \Rightarrow \varphi_k) \Rightarrow (A \Rightarrow \varphi_i) \qquad A \Rightarrow (\varphi_k \Rightarrow \varphi_i)}{(A \Rightarrow \varphi_k) \Rightarrow (A \Rightarrow \varphi_i)} \tag{1.5}$$

Entonces se sigue 1.6,

$$\Gamma \vdash (A \Rightarrow \varphi_k) \Rightarrow (A \Rightarrow \varphi_i) \tag{1.6}$$

Luego, aplicando Modus Ponens a 1.2 y 1.6 se sigue 1.7,

$$\frac{(A \Rightarrow \varphi_k) \Rightarrow (A \Rightarrow \varphi_i) \qquad A \Rightarrow \varphi_k}{A \Rightarrow \varphi_i} \tag{1.7}$$

Así, se puede concluir 1.8

$$\Gamma \vdash (A \Rightarrow \varphi_i) \tag{1.8}$$

De esta forma, se puede concluir por inducción que $\Gamma \vdash (A \Rightarrow \varphi_i)$, *para todo* $i \in \{1, ..., n\}$. *En particular para* $i = n$, *se cumple que* $\Gamma \vdash (A \Rightarrow \varphi_n)$ *y como* $\varphi_n = B$, *entonces* $\Gamma \vdash (A \Rightarrow B)$.

Para mostrar la suficiencia, se supone que $\Gamma \vdash (A \Rightarrow B)$. *Luego, se sigue fácilmente que* $\Gamma \cup \{A\} \vdash (A \Rightarrow B)$ *y también que* $\Gamma \cup \{A\} \vdash A$ *y aplicando Modus Ponens a las fórmulas anteriores se obtiene una demostración de* B *a partir de* $\Gamma \cup \{A\}$, *concluyendo así la demostración.* □

Lema 1.3.1 *Dados* A, B *y* C *fórmulas bien formadas del lenguaje se cumple lo siguiente.*

i. $\neg\neg A \vdash A$

ii. $B \vdash \neg\neg B$

iii. $\{(A \Rightarrow B), (B \Rightarrow C)\} \vdash (A \Rightarrow C)$

iv. $\vdash (\neg A \Rightarrow (A \Rightarrow B))$

v. $(A \Rightarrow B) \vdash (\neg B \Rightarrow \neg A)$

vi. $\vdash (A \Rightarrow (\neg B \Rightarrow \neg(A \Rightarrow B)))$

vii. $\vdash ((A \Rightarrow B) \Rightarrow ((\neg A \Rightarrow B) \Rightarrow B))$

Demostración:

i. La demostración de $\neg\neg A \vdash A$ *es la siguiente:*

(1) $\neg\neg A$	*(Premisa)*
(2) $\neg\neg A \Rightarrow (\neg\neg\neg\neg A \Rightarrow \neg\neg A)$	*(A-1)*
(3) $\neg\neg\neg\neg A \Rightarrow \neg\neg A$	*(M.P entre 1 y 2)*
(4) $(\neg\neg\neg\neg A \Rightarrow \neg\neg A) \Rightarrow (\neg A \Rightarrow \neg\neg\neg A)$	*(A-3)*
(5) $\neg A \Rightarrow \neg\neg\neg A$	*(M.P entre 3 y 4)*
(6) $(\neg A \Rightarrow \neg\neg\neg A) \Rightarrow (\neg\neg A \Rightarrow A)$	*(A-3)*
(7) $\neg\neg A \Rightarrow A$	*(M.P entre 5 y 6)*
(8) A	*(M.P entre 7 y 1)*

ii. La demostración de $B \vdash \neg\neg B$ *es la siguiente:*

(1) B	*(Premisa)*
(2) $(\neg\neg\neg B \Rightarrow \neg B) \Rightarrow (B \Rightarrow \neg\neg B)$	*(A-3)*
(3) $\neg\neg\neg B \Rightarrow \neg B$	*(Inciso i, con* $A = \neg B$ *y teorema 1.3)*
(4) $B \Rightarrow \neg\neg B$	*(M.P entre 2 y 3)*
(5) $\neg\neg B$	*(M.P entre 1 y 4)*

iii. Antes de realizar la demostración de $\{(A \Rightarrow B), (B \Rightarrow C)\} \vdash (A \Rightarrow C)$*, se utilizará el teorema 1.3 para obtener la equivalencia* $\{(A \Rightarrow B), (B \Rightarrow C), A\} \vdash C$*, por lo que la demostración sería la siguiente:*

(1) $A \Rightarrow B$	*(Premisa)*
(2) $B \Rightarrow C$	*(Premisa)*
(3) A	*(Premisa)*
(4) B	*(M.P. entre 1 y 3)*
(5) C	*(M.P. entre 2 y 4)*

iv. Antes de ejecutar la demostración de $\vdash (\neg A \Rightarrow (A \Rightarrow B))$*, aplicándole dos veces el teorema 1.3, la demostración pasa de probar* $\vdash (\neg A \Rightarrow (A \Rightarrow B))$ *a* $\{\neg A, A\} \vdash B$*.*

(1) $\neg A$	*(Premisa)*

(2) A	*(Premisa)*
(3) $\neg A \Rightarrow (\neg B \Rightarrow \neg A)$	*(A-2)*
(4) $\neg B \Rightarrow \neg A$	*(M.P entre 1 y 3)*
(5) $(\neg B \Rightarrow \neg A) \Rightarrow (A \Rightarrow B)$	*(A-3)*
(6) $A \Rightarrow B$	*(M.P entre 4 y 5)*
(7) B	*(M.P entre 2 y 6)*

v. La demostración de $(A \Rightarrow B) \vdash (\neg B \Rightarrow \neg A)$ *es la siguiente:*

(1) $(A \Rightarrow B)$	*(Premisa)*
(2) $\neg\neg A \Rightarrow B$	*(Inciso iii con* $A = \neg\neg A$, $B = A$ *y* $C = B$*)*
(3) $B \Rightarrow \neg\neg B$	*(Aplicación de ii.)*
(4) $\neg\neg A \Rightarrow \neg\neg B$	*(Inciso iii con* $A = \neg\neg A$, $B = B$ *y* $C = \neg\neg B$*)*
(5) $(\neg\neg A \Rightarrow \neg\neg B) \Rightarrow (\neg B \Rightarrow \neg A)$	*(A-3 con* $A = \neg A$ *y* $B = \neg B$*)*
(6) $\neg B \Rightarrow \neg A$	*(M.P entre 5 y 6)*

vi. La demostración de $\vdash (A \Rightarrow (\neg B \Rightarrow \neg(A \Rightarrow B)))$ *se empieza después de aplicar dos veces el teorema 1.3 al enunciado, concluyendo que se debe mostrar* $\{A, \neg B\} \vdash \neg(A \Rightarrow B)$.

(1) A	*(Premisa)*
(2) $\neg B$	*(Premisa)*
(3) $A \Rightarrow ((A \Rightarrow B) \Rightarrow B)$	*(Inciso iii. con* $A = A$ *y* $B = (A \Rightarrow B) \Rightarrow B$*)*
(4) $((A \Rightarrow B) \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg(A \Rightarrow B))$	*(Inciso iii. con* $B = (A \Rightarrow B) \Rightarrow B$ *y* $C = \neg B \Rightarrow \neg(A \Rightarrow B)$*)*
(5) $A \Rightarrow (\neg B \Rightarrow \neg(A \Rightarrow B))$	*(Inciso iii. con 3 y 4)*
(6) $\neg B \Rightarrow \neg(A \Rightarrow B)$	*(Modus Ponens, 1 y 5)*
(7) $\neg(A \Rightarrow B)$	*(Modus Ponens, 2 y 6)*

vii. Antes de ejecutar la demostración de $\vdash ((A \Rightarrow B) \Rightarrow ((\neg A \Rightarrow B) \Rightarrow B))$, *es necesario un resultado intermedio, por lo que primero se probará que* $\vdash (P \Rightarrow (Q \Rightarrow R)) \Rightarrow (Q \Rightarrow (P \Rightarrow R))$. *Para hacer la prueba se aplica tres veces el teorema 1.3, por lo que la demostración sería la siguiente* $\{P \Rightarrow (Q \Rightarrow R), Q\} \vdash (P \Rightarrow R)$.

(1) $P \Rightarrow (Q \Rightarrow R)$	*(Premisa)*
(2) Q	*(Premisa)*
(3) $(P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \Rightarrow Q) \Rightarrow (P \Rightarrow R))$	*(A-2)*

(4) $Q \Rightarrow (P \Rightarrow Q)$ *(A-1 con $P = Q$ y Q=P)*
(5) $(P \Rightarrow Q) \Rightarrow (P \Rightarrow R)$ *(M.P entre 1 y 3)*
(6) $P \Rightarrow Q$ *(M.P entre 2 y 4)*
(7) $P \Rightarrow R$ *(M.P entre 5 y 6)*

Ya con este resultado intermedio se hará la prueba de $\{(A \Rightarrow B), (\neg A \Rightarrow B)\} \vdash B$, *después de aplicarle el teorema 1.3.*

(1) $A \Rightarrow B$ *(Premisa)*
(2) $\neg A \Rightarrow B$ *(Premisa)*
(3) $(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)$ *(Inciso v)*
(4) $\neg B \Rightarrow \neg A$ *(M.P entre 1 y 3)*
(5) $\neg A \Rightarrow (\neg B \Rightarrow \neg(\neg A \Rightarrow B))$ *(Inciso vi con $A = \neg A$)*
(6) $\neg B \Rightarrow (\neg A \Rightarrow \neg(\neg A \Rightarrow B))$ *(Resultado intermedio en 5)*
(7) $(\neg B \Rightarrow (\neg A \Rightarrow \neg(\neg A \Rightarrow B))) \Rightarrow$ *(A-2 con:*
$((\neg B \Rightarrow \neg A) \Rightarrow$ $P = \neg B, Q = \neg A,$
$(\neg B \Rightarrow \neg(\neg A \Rightarrow B)))$ $R = \neg(\neg A \Rightarrow B))$
(8) $(\neg B \Rightarrow \neg A) \Rightarrow (\neg B \Rightarrow \neg(\neg A \Rightarrow B))$ *(M.P entre 6 y 7)*
(9) $\neg B \Rightarrow \neg(\neg A \Rightarrow B)$ *(M.P entre 4 y 8)*
(10) $(\neg B \Rightarrow \neg(\neg A \Rightarrow B)) \Rightarrow$ *(A-3 con:*
$((\neg A \Rightarrow B) \Rightarrow B)$ $P = \neg B,$
$Q = (\neg A \Rightarrow B))$
(11) $(\neg A \Rightarrow B) \Rightarrow B$ *(M.P entre 9 y 10)*
(12) B *(M.P entre 2 y 11)*

□

Definición 1.10 *Sea A un enunciado del lenguaje $\mathcal{L}^*$ y $a_1, a_2, ..., a_n$ los enunciados atómicos que lo conforman. Sea υ una función con dominio en los enunciados atómicos y codominio en el conjunto $\{V, F\}$, la cual llamaremos asignación, de tal forma que $A', A_1, A_2, ..., A_n$ son como se muestra a continuación:*

$$A' = \begin{cases} A & si \quad \upsilon^*(A) = V \\ \\ \neg A & si \quad \upsilon^*(A) = F \end{cases}$$

$$A_i = \begin{cases} a_i & si \quad \upsilon(a_i) = V \\ \\ \neg a_i & si \quad \upsilon(a_i) = F \end{cases}$$

para todo $i \in \{1, 2, ..., n\}$.

La necesidad de la definición 1.10, es para enunciar el lema 1.3.2, que es crucial en la demostración de completitud. Pero antes de hacerlo, se presenta el lema 1.3.1 que contiene las demostraciones elementales para ejecutar la prueba del lema 1.3.2.

Lema 1.3.2 *Para cualquier fórmula bien formada A en el lenguaje $\mathcal{L}^*$ y cualquier asignación υ, que definan $A', A_1, A_2, ..., A_n$, como en la definición 1.10, entonces se cumple que:*

$$\{A_1, A_2, ..., A_n\} \vdash A'.$$

Demostración: *Procedemos por inducción sobre las conectivas lógicas que contiene la fórmula bien formada A.*

(1) Suponiendo que la fórmula A no tiene conectivas lógicas, tiene que ser atómica, por lo que se puede suponer que $A = a_1$, sin pérdida de generalidad. De esta forma, se consideran los siguientes dos casos.

(1.1) Cuando $\upsilon^*(A) = V$, entonces $A' = A = a_1$ y $A_1 = a_1$, por lo que se tendría que mostrar que la fórmula 1.9 es válida.

$$a_1 \vdash a_1 \tag{1.9}$$

Pero aplicando el teorema 1.3 a la fórmula 1.9, se sigue la fórmula 1.10, que ya se probó durante la demostración del teorema anterior.

$$\vdash (a_1 \Rightarrow a_1) \tag{1.10}$$

Por lo tanto la fórmula 1.9 es válida.

(1.2) Cuando $\upsilon^*(A) = F$, entonces $A' = \neg A = \neg a_1$ y $A_1 = \neg a_1$, por lo que se tendría que mostrar que la fórmula1.11 es válida.

$$\neg a_1 \vdash \neg a_1 \tag{1.11}$$

Que por la misma razón del caso anterior es válida.

(2) Suponiendo como hipótesis inductiva que se cumple el lema 1.3.2 para cualquier fórmula bien formada con k conectivas lógicas, tales que $k < n$, donde n es la cantidad de conectivas lógicas en A.

(3) Como paso inductivo se tendrían los siguientes casos.

(3.1) Suponiendo que $A = \neg B$, entonces se sigue que B contiene k conectivas lógicas, con $k < n$, por lo que bajo la hipótesis inductiva se cumple el lema 1.3.2, de esta forma existen $B', B_1, B_2, ..., B_m$ tales que se sostiene la fórmula 1.12

$$\{B_1, ..., B_m\} \vdash B' \tag{1.12}$$

Donde B' y $B_1, ..., B_m$ se definen bajo la definición 1.10. Ahora bien, como $B_1, B_2, ..., B_m$ provienen directamente de $b_1, b_2, ..., b_m$, que son las expresiones atómicas de B y a su vez la hipótesis es que $A = \neg B$,

entonces necesariamente $b_1, b_2, ..., b_m$ *son las expresiones atómicas de* A *y así resta demostrar la validez de la fórmula 1.13.*

$$\{B_1, ..., B_m\} \vdash A' \tag{1.13}$$

Para ejecutar esta demostración, es necesario revisar los siguientes casos.

(3.1.1) Si $\upsilon^*(B) = T$*, entonces* $B' = B$*, por lo que se tendría la fórmula 1.14.*

$$\{B_1, ..., B_m\} \vdash B \tag{1.14}$$

Por otro lado, se sigue que $\upsilon^*(A) = \upsilon^*(\neg B) = \neg\upsilon^*(B) = F$*, entonces* $A' = \neg A$*, obteniendo así la fórmula* $A' = \neg A = \neg\neg B$*. Ahora, como en la proposición 1.3.1 se demostró que la fórmula* $\vdash (B \Rightarrow \neg\neg B)$ *es válida, por lo que la fórmula 1.15 también lo es.*

$$\{B_1, ..., B_m\} \vdash (B \Rightarrow \neg\neg B) \tag{1.15}$$

Entonces, aplicando Modus Ponens (M.P) a las fórmulas 1.14 y 1.15, se sigue la fórmula 1.16.

$$\{B_1, ..., B_m\} \vdash \neg\neg B \tag{1.16}$$

Por lo que se puede concluir la fórmula 1.17.

$$\{B_1, ..., B_m\} \vdash A' \tag{1.17}$$

(3.1.2) Si $\upsilon^*(B) = F$*, entonces* $B' = \neg B$*, por lo que se tendría la fórmula 1.18.*

$$\{B_1, ..., B_m\} \vdash \neg B \tag{1.18}$$

Pero como bajo hipótesis $A = \neg B$*, entonces se sigue la fórmula 1.19.*

$$\{B_1, ..., B_m\} \vdash A' \tag{1.19}$$

(3.2) Suponiendo que $A = (B \Rightarrow C)$*, entonces existen* i, j *menores a* m *que es la cantidad de expresiones atómicas que componen a la fórmula* A*. De esta forma, existen* $B', B_1, B_2, ..., B_i$ *y* $C', C_1, C_2, ..., C_j$ *provenientes de la definición 1.10 sobre* B *y* C*, que al tener menos de* n *conectivas lógicas, configuran las fórmulas en 1.20, bajo la hipótesis de inducción.*

$$\begin{aligned} &\{B_1, ..., B_i\} \vdash B' \\ &\{C_1, ..., C_j\} \vdash C' \end{aligned} \tag{1.20}$$

Luego, como necesariamente las expresiones atómicas en B *y* C *están contenidas en* A*, entonces se sigue la fórmula 1.21*

$$\{A_1, A_2..., A_m\} \vdash B'$$
$$\{A_1, A_2..., A_m\} \vdash C' \tag{1.21}$$

Donde $A_1, A_2..., A_m$ provienen de las expresiones atómicas $a_1, a_2, ..., a_n$ de la fórmula A. De esta forma se tienen los siguientes casos.

(3.2.1) Si $v^*(B) = v^*(C) = V$, entonces se sigue que $B' = B$ y a su vez $C' = C$, de esta forma se sigue de la fórmula 1.21 se sigue la fórmula 1.22.

$$\{A_1, A_2..., A_m\} \vdash C \tag{1.22}$$

Luego, aplicando el axioma A-1, se puede seguir la fórmula 1.23

$$\{A_1, A_2..., A_m\} \vdash (C \Rightarrow (B \Rightarrow C)) \tag{1.23}$$

De esta forma, aplicando Modus Ponens a las fórmulas 1.22 y 1.23, se sigue la fórmula.

$$\{A_1, A_2..., A_m\} \vdash (B \Rightarrow C) \tag{1.24}$$

Luego, como bajo las hipótesis se sigue que $A = (B \Rightarrow C)$ y $A' = A$, entonces de la fórmula 1.24 se puede seguir 1.25

$$\{A_1, A_2..., A_m\} \vdash A' \tag{1.25}$$

(3.2.2) Si $v^*(B) = V$ y $v^*(C) = F$. Bajo estos supuestos se sigue que $B' = B$ y también que $C' = \neg C$, por lo tanto se sigue la fórmula 1.26

$$\{A_1, A_2..., A_m\} \vdash B$$
$$\{A_1, A_2..., A_m\} \vdash \neg C \tag{1.26}$$

Luego, usando el inciso v. del lema 1.3.1, se sigue que $\vdash (B \Rightarrow (\neg C \Rightarrow \neg(A \Rightarrow C)))$, por lo que se puede deducir la fórmula 1.27.

$$\{A_1, A_2..., A_m\} \vdash (B \Rightarrow (\neg C \Rightarrow \neg(A \Rightarrow C))) \tag{1.27}$$

De esta forma, aplicándole Modus Ponens a la fórmula 1.27, tanto con ambas de las fórmulas en 1.26 se puede concluir la fórmula 1.28.

$$\{A_1, A_2..., A_m\} \vdash (\neg(A \Rightarrow C)) \tag{1.28}$$

Por otro lado, bajo la hipótesis de que $v^*(B) = V$ y $v^*(C) = F$, se puede concluir que $v^*(B \Rightarrow C) = F$, por lo que $A' = \neg(B \Rightarrow C)$, logrando concluir la fórmula 1.29

$$\{A_1, A_2..., A_m\} \vdash A' \tag{1.29}$$

(3.2.3) Si $v^*(B) = F$, entonces se sigue que $B' = \neg B$, por lo que se sigue la fórmula 1.30.

$$\{A_1, A_2..., A_m\} \vdash \neg B \tag{1.30}$$

Por otro lado, usando el lema 1.3.1, al aplicar Modus Ponens entre la fórmula $\vdash (\neg A \Rightarrow (A \Rightarrow B)$ y la fórmula 1.30, se sigue 1.31.

$$\{A_1, A_2..., A_m\} \vdash (A \Rightarrow B) \tag{1.31}$$

Luego, sin importar el valor que tome $v^*(C)$, siempre que sigue que $v^*(B \Rightarrow C) = V$, entonces $A' = (B \Rightarrow C)$ y por lo tanto, de la fórmula 1.31 se sigue que 1.32.

$$\{A_1, A_2..., A_m\} \vdash A' \tag{1.32}$$

□

Teorema 1.4 (Teorema de Completitud) *Para cualquier fórmula bien formada A en el lenguaje $\mathcal{L}^*$,*

$$\vdash A \text{ si y solo si } \models A$$

Demostración:

Sea A una FBF, compuesta por las expresiones atómicas $a_1, a_2, ..., a_n$ y v una asignación de A, bajo la definición 1.10. Se define en 1.33 la restricción de v a la FBF A.

$$v_A : \{a_1, a_2, ..., a_n\} \to \{V, F\} \tag{1.33}$$

En base a 1.33 se define a en 1.34, la clase de asignaciones de A, restringidas a la misma.

$$V_A = \{v_A | v_A : \{a_1, a_2, ..., a_n\} \to \{V, F\}\} \tag{1.34}$$

Una vez se tienen las definiciones anteriores, primero se supondrá que $\models A$.

Por el lema 1.3.2, se sigue que para cualquier $v_A \in V_A$ existen $A_1, A_2, ..., A_n$ FBF tales que cumplen 1.35

$$A_1, A_2, ..., A_n \vdash A \tag{1.35}$$

De esta forma, específicamente al seleccionar $v_1, v_2 \in V_A$, con $v_1 \neq v_2$, tales que se cumple 1.36.

$$\begin{gathered} v_1|\{a_1, a_2, ..., a_{n-1}\} = v_2|\{a_1, a_2, ..., a_{n-1}\} \\ v_1(a_n) = V \\ v_2(a_n) = F \end{gathered} \tag{1.36}$$

Bajo el supuesto de que $\models A$, bajo la definición 1.10, como $v^*(A) = V$ se cumple que $A' = A$ y también como $v_1(a_n) = V$ es cierto que $A_n = a_n$, por lo que de 1.35 se deduce 1.37.

$$\{A_1, A_2, ..., a_n\} \vdash A \tag{1.37}$$

A partir de 1.37, por el teorema 1.3, se cumple 1.38.

$$\{A_1, A_2, ..., A_{n-1}\} \vdash (a_n \Rightarrow A) \tag{1.38}$$

Siguiendo un razonamiento similar, para el caso de que $v_2(a_n) = F$, se sigue 1.39.

$$\{A_1, A_2, ..., A_{n-1}\} \vdash (\neg a_n \Rightarrow A) \tag{1.39}$$

Luego, por el inciso vi del lema 1.3.1, se cumple 1.40.

$$\vdash ((a_n \Rightarrow A) \Rightarrow ((\neg a_n \Rightarrow A) \Rightarrow A)) \tag{1.40}$$

Por lo que se cumple 1.41.

$$\{A_1, A_2, ..., A_{n-1}\} \vdash ((a_n \Rightarrow A) \Rightarrow ((\neg a_n \Rightarrow A) \Rightarrow A)) \tag{1.41}$$

De esta forma, aplicando Modus Ponens, primero entre 1.41 y 1.38. Después, entre 1.41 y 1.39 se concluye 1.42

$$\{A_1, A_2, ..., A_{n-1}\} \vdash A \tag{1.42}$$

Aplicando este mismo proceso a las $n-1$ fórmulas en $\{A_1, A_2, ..., A_{n-1}\}$, se logra demostrar 1.43.

$$\vdash A \tag{1.43}$$

La suficiencia es la demostración de que el sistema es correcto, lo cual es cierto, por lo que la demostración queda completada. □

CAPÍTULO 2

Computabilidad del sistema formal $\mathcal{L}^*$

Existen muchas formas de definir el concepto de computabilidad, pero las primeras definiciones formales fueron las de Alonzo Church y Alan Turing en el año 1936, siendo la segunda la más trascendente por la formalización del concepto de computación a través de sus **Máquinas Universales** que ahora llamamos (Immerman, 2021).

Los trabajos de Church y Turing comenzaron con la intención de dar respuesta al "Entscheidungsproblem", que en español se traduciría como "el problema de decisión" y se refiere a encontrar un método efectivo para decidir si cualquier fórmula en un cálculo proposicional dado es demostrable. En términos de lo que ya se ha presentado en este trabajo, si dada cualquier fórmula existe un algoritmo que encuentra una demostración formal, ya sea de la fórmula o de su negación, utilizando únicamente las reglas del sistema. Para su prueba, Church y Turing usaron la lógica de primer orden, que consiste en una lógica proposicional estándar, como la que se ha construido hasta el momento en este trabajo, aunado con cuantificadores lógicos, como el "para todo", común en el lenguaje de las matemáticas.

El matemático alemán más influyente de su época, David Hilbert, y su par en la universidad de Göttingen, Wilhelm Ackermann, en 1928 señalaron al "Entscheidungsproblem" como "el principal problema de la lógica matemática" con la certeza de que una solución sistemática era posible (Copeland, 2024). Sin embargo, de manera unánime, con dos enfoques diferentes, tanto Church como Turing llegaron a la conclusión de que el problema era indecidible.

Alan Turing concibió las "máquinas de Turing" como la idealización de un dispositivo compuesto esencialmente por una cinta infinita donde se puede escribir y leer información. Esta cinta es manipulada por un cabezal que se mueve mecánicamente hacia adelante o hacia atrás para realizar dichas operaciones. La máquina cuenta también con un estado interno que cambia según las necesidades del proceso, así como con una lista de reglas que se aplican en función del estado actual y del símbolo que se esté leyendo en la cinta. Para construir una máquina de Turing de manera formal, se requiere un alfabeto bien definido y una cantidad finita de estados internos; una vez establecidos, la cinta se divide en cuadros que solo pueden contener uno de los símbolos del alfabeto.

2.1. La tesis de Church-Turing y el concepto de algoritmo

El trabajo que hicieron Church y Turing terminó convirtiéndose en lo que ahora se le da el nombre de **Tesis de Church-Turing**, que en el estado del arte hace referencia a un método sistemático o mecánico, que de cumplir su objetivo, se le considera efectivo, bajo las siguientes consideraciones:

i. El método se establece en términos de un número preciso de instrucciones exactas.

ii. El método alcanza su objetivo en una cantidad finita de pasos.

iii. El método puede ser ejecutado por cualquier humano.

iv. El método no depende de algún contexto y/o intuición previa para ser ejecutado.

Aunque la definición anterior es fácil de entender, también carece de rigor matemático, por esta razón Turing y Church, trabajando de forma independiente, propusieron conceptos formales para sustituir la noción informal de "método efectivo".

El trabajo de Turing consistió esencialmente en introducir el concepto de computabilidad a través de "máquinas universales" y Church el del "cálculo lambda". Aún cuando los enfoques fueron diferentes, resultaron ser equivalentes, al identificar el mismo conjunto de funciones matemáticas cuyas salidas pueden obtenerse mediante un método efectivo. En términos prácticos, se puede sostener que alguna función es computable si y sólo si pertenece al conjunto mencionado, lo que permite reemplazar afirmaciones informales como "hay un método efectivo para calcular la función" por una que diga "la función f pertenece al conjunto de funciones computables".

El trabajo de Turing sostiene que si existe un método efectivo para calcular los valores de una función matemática, entonces esa función puede ser computada por una "máquina de Turing". De esta forma, toda función computada por una "máquina de Turing" es también el resultado de un método efectivo, ya que un programa de máquina de Turing es en sí mismo un método efectivo, porque cualquier humano puede seguir sus instrucciones sin necesidad de ingenio. De esta forma, la discusión sobre métodos efectivos puede ser reemplazada por la discusión sobre la existencia o inexistencia de programas para máquinas de Turing.

Definido el concepto de método efectivo, un algoritmo puede entenderse como un método efectivo diseñado para resolver un problema específico. Este se caracteriza por una secuencia finita de instrucciones precisas que pueden ejecutarse sin necesidad de intuición o contexto previo.

En consecuencia, si un algoritmo es efectivo, entonces existe una máquina de Turing capaz de computarlo, y viceversa.

Definición de los estados de una máquina de Turing

Los estados de una máquina de Turing se definen a través de tuplas, cuyas entradas representan la información esencial para entenderlos, tal como se explica a continuación.

$$< A, B, C, D, E >$$

Donde:

i. A representa el estado actual en el que se encuentra la máquina de Turing.

ii. B representa el símbolo que actualmente está leyendo el cabezal de la máquina de Turing.

iii. C representa el símbolo por el que se va a reemplazar o no, el que actualmente está leyendo el cabezal.

iv. D representa la instrucción que se debe ejecutar después del cambio en el valor actual. La cual podría ser "R" para mover el cabezal a la derecha o "L" para moverlo hacía la izquierda.

v. E representa el estado final, después de la ejecución de las instrucciones.

Una tupla dentro de una máquina de Turing, bajo la definición anterior, en un alfabeto con unos y ceros, se vería como la siguiente.

$$< q_0, 1, 0, R, q_1 >$$

Donde q_0 es el estado inicial, para el cual se lee el carácter 1 y se reemplaza por 0, luego el cabezal se mueve a la derecha y termina en el estado q_1.

Por otro lado, si se quisiera pasar de un estado a otro sin generar alguna modificación, es necesario agregar un símbolo más, que al ponerlo en cualquier entrada de la tupla represente no hacer algún cambio, que será el guion bajo "_". Para ejemplificarlo, se presenta el siguiente ejemplo.

$$< q_0, 1, \_, R, q_1 >$$

Donde se parte de un estado q_0, hacia el siguiente leyendo el carácter 1 y moviéndose a la derecha de la cinta, que se representa con la letra mayúscula R, para así terminar en el estado q_1.

Al tener la posibilidad de escribir caracteres sobre la cinta, tener una operación inversa sería de gran utilidad. Para cambiar una letra por un espacio en blanco sobre la cinta, se representará este con el carácter "*", para que no exista ambigüedad. De esta forma, una tupla donde se quiera reemplazar un carácter dentro del alfabeto por un espacio en blanco en la cinta se vería como la siguiente.

$$< q_0, 1, *, R, q_1 >$$

Turing pensó en sus máquinas equipadas con una cinta infinita; sin embargo, el conjunto de instrucciones siempre es finito, sobre todo cuando no se consideran aquellos estados en los que se regresa de manera cíclica a ellos.

Para reforzar la definición, se ejemplifica a continuación la construcción de una máquina de Turing, pensando en que esta va a sumar uno a cualquier número en su representación binaria. Para comenzar con la solución de este problema, a continuación se plantean algunos casos, que permitirán llegar a la solución final.

i. Considerando la representación binaria del número 22, de derecha a izquierda, se sigue que sumarle 1 a este número sería equivalente a poner uno en el dígito menos significativo de la representación, como se observa en la figura 2.1.

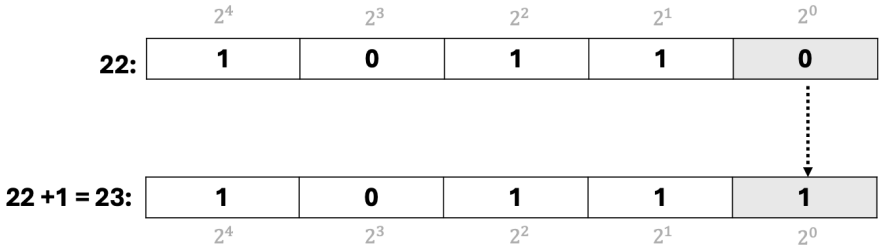

Figura 2.1: Suma de 1 al número 22, en binario

Con el caso presentado, es fácil deducir que para cualquier número par, sumarle uno es poner uno en el dígito menos significativo de su representación binaria.

ii. Ahora, si tomamos el número 23 como ejemplo, sumarle uno su representación binaria, se vería como en la figura 2.2.

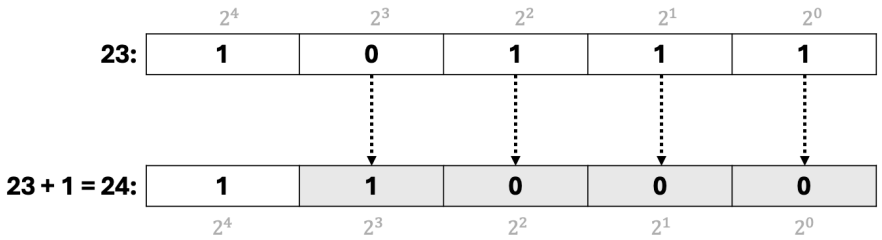

Figura 2.2: Suma de 1 al número 23, en binario

Este caso no es tan simple como el del inciso anterior, ya que no solo se tiene que poner uno en la cifra menos significativa, pero sí se puede ver un patrón, que se resume en los siguientes pasos.

- Siempre mover el cabezal hacia la cifra menos significativa.
- Después de alcanzar la cifra menos significativa, se tiene que recorrer el número hacia la izquierda hasta:
  - Encontrar uno, para cambiarlo por un cero y mover el cabezal una posición a la izquierda.
  - Encontrar cero, para cambiarlo por uno y detener el proceso.

iii. Existe un caso más, como es el del número 31, cuya representación sería como se presenta en la figura 2.3.

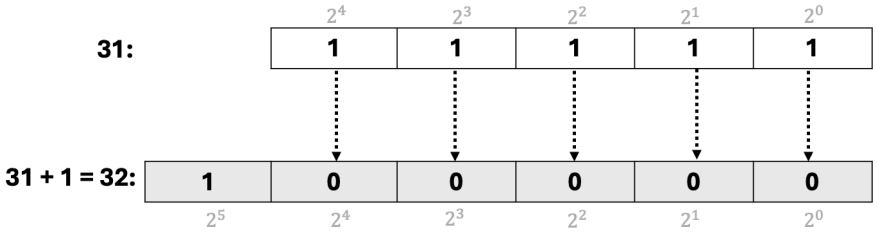

Figura 2.3: Suma de 1 al número 31, en binario.

Este caso es interesante, ya que siguiendo la lógica del inciso anterior, se avanzaría el cabezal hasta la cifra menos significativa y luego se empezaría a regresar cambiando los unos por cero, pero sin un límite en la cifra más significativa.

Finalmente, $< q_1, *, 1, \_, q_2 >$ para el caso de 31.

De forma generalizada, adaptando la representación de los números binarios a una que se pueda representar con las tuplas definidas para las máquinas de Turing, tal como se puede ver en la expresión 2.1.

$$*10111* \tag{2.1}$$

Las tuplas que constituyen la máquina de Turing que le suma el número 1 a cualquier otro en su representación binaria serían los siguientes.

$$\begin{aligned}
&< q_0, 1, _, R, q_0 > \\
&< q_0, 0, _, R, q_0 > \\
&< q_0, *, 1, L, q_1 > \\
&< q_1, 1, 0, L, q_1 > \\
&< q_1, 0, 1, _, q_2 >
\end{aligned}$$

De forma gráfica, una máquina de Turing se puede representar a través de diagramas similares a los que se utilizan para representar "grafos". Donde cada vértice representa un estado diferente y cada arista la operación que representa moverte de estado a estado. De esta forma, dado cualquier número, existe una descomposición única, como se muestra en la figura 2.4.

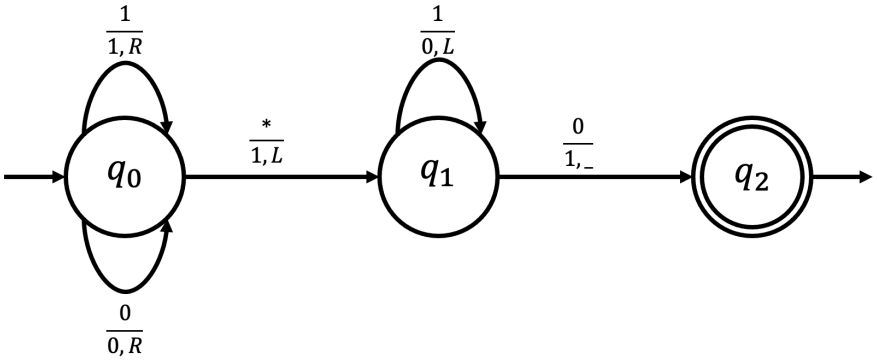

Figura 2.4: Diagrama de los estados de la máquina de Turing que suma 1 a cualquier número en binario.

El estado q_0 de la Figura 2.4, representa mover el cabezal hasta la cifra menos significativa. Cuando por fin alcanza el final del número que se representa con un $*$, regresa el cabezal una posición a la izquierda y entonces pasa al estado siguiente q_1, donde mientras siga leyendo unos los cambia por cero y se sigue moviendo a la izquierda, hasta alcanzar el primer cero y lo cambia por uno para así pasar al estado final q_2.

Formalización de la Máquina de Turing

Hasta el momento se han visto dos formas útiles de construir una máquina de Turing para un ejemplo específico, pero con el objetivo de generalizar el concepto se presenta la siguiente definición.

Definición 2.1 (Máquina de Turing) *Una Máquina de Turing es una siete tupla, como la siguiente.*

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$$

Donde, los conjuntos Q, Σ, Γ son finitos y cada símbolo en la tupla toma las siguientes definiciones:

i. Q *es el conjunto de estados*

ii. Σ *es el alfabeto, sin contener el símbolo* $*$.

iii. Γ *es el alfabeto, donde* $* \in \Gamma$ *y* $\Sigma \subseteq \Gamma$.

iv. $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$, *es la función de transición.*

v. $q_0 \in Q$ *es el estado inicial.*

vi. $q_{accept} \in Q$ *es el estado de aceptado.*

vii. $q_{reject} \in Q$ *es el estado de rechazado. Necesariamente* $q_{reject} \neq q_{accept}$.

La parte medular de la definición 2.1 es la función de transición *delta*, ya que le da estructura a la máquina de Turing, paso a paso. Esta función, aparte de su forma abstracta, se puede ver como el diagrama en la Figura 2.4, lo cual hace más sencillo la construcción de alguna máquina de Turing.

2.2. Codificación de cadenas de texto en números

Para poder deducir si una fórmula es computable o no, a través de una máquina de Turing es necesario un paso previo, que es la codificación de cadenas de texto en algo manejable por la máquina, por lo que se recurre a la codificación de texto a número.

Existen formas diferentes de hacer esto, siendo la de Gödel una de las más famosas, ya que la usó para construir la demostración de su primer teorema de incompletitud, donde establece que en cualquier sistema formal consistente, en el que se pueda realizar cierta aritmética elemental, hay enunciados que no pueden ser probados ni refutados en el mismo; es decir, el sistema es incompleto.

La primera parte del proceso de codificación que propuso Gödel es definir el número del símbolo en el sistema, lo cual se puede representar como $\#('s')$, donde s representa el símbolo al cual se le está asignando un número. Por ejemplo, tomando un sistema aritmético elemental como el de Peano, que fue el utilizado por Gödel, se le podría asignar una numeración como la siguiente:

$$\begin{array}{llll} \#('0') = 1 & \#('x') = 4 & \#(')') = 7 & \#('\forall') = 10 \\ \#('S') = 2 & \#('=') = 5 & \#('\rightarrow') = 8 & \\ \#('+') = 3 & \#('(') = 6 & \#('\neg') = 9 & \end{array}$$

Luego de tener los números de cada símbolo en el sistema, el siguiente paso requiere de un resultado de gran relevancia, "El Teorema Fundamental de la Aritmética", que enuncia la propiedad de todos los números naturales mayores a 1 de poder ser escritos de manera única como el producto de potencias de números primos. De esta forma, si se asigna una potencia definida a cada número primo y se multiplican entre ellos, siempre se encontrará de forma única un número natural, tal como se ve en 2.2

$$m = 2^{n_0} \times 3^{n_1} \times 5^{n_2} ... \times p_{k+1}^{n_{k+1}} \tag{2.2}$$

De esta forma se puede codificar cualquier enunciado de esta forma, ya que se pueden definir sucesiones de números que representen a los enunciados y las cuales indicarán las potencias que tendrán la secuencia ordenada de números primos que se multiplican, por ejemplo, en el enunciado 2.3

$$0 = 0 \tag{2.3}$$

Definiendo la sucesión $< 1, 5, 1 >$ con los números de cada uno de los símbolos en 2.3, se sigue que la codificación de 2.3 es la que se observa en 2.4.

$$2430 = 2 \times 243 \times 5 = 2^1 \times 3^5 \times 5^1 \tag{2.4}$$

De esta forma, como en el ejemplo, solo se necesita una sucesión construida con los números de los símbolos atómicos del sistema aritmético en el que se esté trabajando para mapear de forma única algún enunciado en el sistema a un número entero a través del producto de potencias de números primos.

Como ya se mencionó, la codificación de Gödel tiene una relevancia histórica por los "teoremas de incompletud de Gödel", sin embargo, para este trabajo se utilizará una distinta, que consiste en mapear cualquier enunciado en un lenguaje hacia un número binario.

Para mapear un enunciado en un lenguaje hacia un número binario, primero se le debe asignar una numeración a los símbolos atómicos del lenguaje. Definir una numeración que represente a todos los símbolos ha sido un reto por muchos años, sobre todo desde el advenimiento de la computación moderna.

Lograr un consenso global sobre qué código emplear para representar las letras de cada alfabeto, en la vasta diversidad de alfabetos del mundo, constituye una tarea monumental.

Dado que uno de los objetivos de este trabajo no es desarrollar un estándar propio para la codificación de las letras del alfabeto latino, se adoptará una ampliamente reconocida: el "American Standard Code for Information Interchange", más conocido como código "ASCII". De esta forma, se tendría el número de cada símbolo, o al menos de la mayoría, ya que en el alfabeto latino no existen el símbolo de para todo $\forall$ ó $\rightarrow$, como en el sistema aritmético de Peano.

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[END OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

Figura 2.5: Tabla con una lista de caracteres útiles en ASCII (Psychpsyo, 2022).

Definición 2.2 (Transformación de número entero a binario) *Dado un número entero no negativo n, su representación binaria $b_k, b_{k-1}, ..., b_1$ se puede expresar como:*

$$b_i = \left\lfloor \frac{n}{2^i} \right\rfloor \mod 2, \quad \text{para } i \in \{0, 1, 2, \ldots, k\}$$

Donde:

- $b_i \in \{0, 1\}$ *es el i-ésimo bit de la representación binaria, de derecha a izquierda.*
- k *es el mayor entero tal que* $2^k \leq n$.
- $\lfloor x \rfloor$ *denota la parte entera de* x.

Aunque la expresión requiera de una función y el uso de módulos, de forma simple, representar a un número entero en su forma binaria esencialmente consiste en representarlo como una cadena de ceros y unos. Por ejemplo, el número 21 decimal en binario se representa como 10101.

$b_0 = \lfloor \frac{21}{2^0} \rfloor \mod 2 = \lfloor \frac{21}{1} \rfloor \mod 2 = 21 \mod 2 = 1$	$b_0 = 1$
$b_1 = \lfloor \frac{21}{2^1} \rfloor \mod 2 = \lfloor \frac{21}{2} \rfloor \mod 2 = 10 \mod 2 = 0$	$b_1 = 0$
$b_2 = \lfloor \frac{21}{2^2} \rfloor \mod 2 = \lfloor \frac{21}{4} \rfloor \mod 2 = 5 \mod 2 = 1$	$b_2 = 1$
$b_3 = \lfloor \frac{21}{2^3} \rfloor \mod 2 = \lfloor \frac{21}{8} \rfloor \mod 2 = 2 \mod 2 = 0$	$b_3 = 0$
$b_4 = \lfloor \frac{21}{2^4} \rfloor \mod 2 = \lfloor \frac{21}{16} \rfloor \mod 2 = 1 \mod 2 = 1$	$b_4 = 1$

Cuadro 2.1: Proceso conversión del número 21 a binario.

Para hacer una transformación inversa, es decir, de un número binario se debe utilizar la definición 2.3.

Definición 2.3 (Transformación de número binario a entero) *Dado un número binario $b_k, b_{k-1}, ..., b_1$, donde b_i es un bit (0 ó 1) considerando que el bit más significativo está en la posición $n-1$ y el bit menos significativo está en la posición 0. De esta forma, la transformación que tendría a un número entero n se calcula con la siguiente ecuación.*

$$n = \sum_{i=0}^{k-1} b_{k-1-i} \cdot 2^{k-1-i}$$

De forma visual, se pueden usar diagramas como los que se presentaron en la sección de máquinas de Turing, para ubicar los bits que se mencionan en la fórmula.

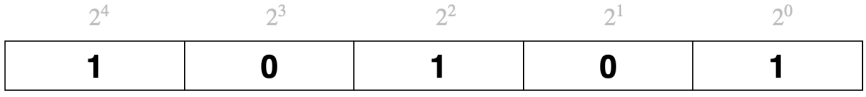

Figura 2.6: Representación en binario del número 21.

Empezando de la izquierda a la derecha en la representación gráfica del número 21 presentado en la figura 2.6, se tienen que sumar las potencias de dos que representan cada uno en la representación binaria, tal como se muestra en la ecuación 2.5.

$$2^0 + 2^2 + 2^4 = 1 + 4 + 16 = 21 \tag{2.5}$$

Contar con un método bien definido para transformar cadenas de texto en números enteros permite modelar las máquinas de Turing como funciones cuyo

dominio y codominio son subconjuntos de los enteros. Esto facilita su análisis y uso en la construcción de algoritmos. Por esta razón, el lenguaje binario constituye la base fundamental de la computación moderna.

2.3. Conjunto computable

Un conjunto computable es cualquier subconjunto de los números enteros, para el cual existe una función característica computable, es decir, un algoritmo que dado un número natural, siempre decide en tiempo finito si este pertenece o no al conjunto, lo que formalmente se enuncia en el teorema 2.1.

Teorema 2.1 (Conjunto computable) *Sea $A \subseteq \mathbb{N}$, es computable si y solo si existe una máquina de Turing M cuya función asociada $\chi_A : \mathbb{N} \longrightarrow \{0,1\}$ satisface:*

$$\chi_A(n) = \begin{cases} 1, & si\ n \in A \\ 0, & si\ n \notin A \end{cases} \tag{2.6}$$

2.4. Computabilidad del sistema formal $\mathcal{L}^*$

En el teorema 1.4 se mostró que el sistema formal $\mathcal{L}^*$ es completo. En esta sección, se mostrará que también es computable, pero para llegar a esta prueba son necesarios los lemas 2.1.1, 2.1.2 y 2.1.3, ya que enuncian que los conjuntos de fórmulas bien formadas, axiomas y enunciados derivados por la única regla de inferencia (Modus Ponens) son computables.

Gracias a las definiciones de máquinas de Turing y codificación que se dieron anteriormente, las demostraciones de los lemas enunciados a continuación se harán a través de algoritmos en el lenguaje `C++`, ya que esencialmente se basan en encontrar un algoritmo que muestre la pertenencia de las fórmulas bien formadas, axiomas, enunciados derivados por la única regla de inferencia (Modus Ponens) y demostraciones correctas en el sistema formal.

Es importante destacar que dentro de las demostraciones estará el código de los algoritmos principales, sin considerar todos los procesos que alimentan a estos; para poder ver y entender el flujo completo es necesario ver el apéndice del presente trabajo.

Lema 2.1.1 *El conjunto de fórmulas bien formadas del lenguaje $\mathcal{L}^*$ es computable.*

Demostración:

Para probar que el conjunto de las fórmulas bien formadas es computable se necesita mostrar que existe algún algoritmo que recibe como entrada cualquier cadena de símbolos y como salida determina si este pertenece al lenguaje $\mathcal{L}^*$. Este algoritmo ya se había mostrado en el capítulo 2 y se muestra en el código 2.1 a continuación.

Código 2.1: Función que evalúa si una expresión es una FBF.

```
void isWFF(vector<string> exp)
{
    stack<bool> pila;
    for (string token : exp) {
        if (isLetter(token)) {
            pila.push(true);
        } else if (isOperator(token)) {
            if(token == "no") {
                if(pila.empty()) {
                    cout << "La expresion no es una formula \
                    bien formada" << endl;
                    return;
                }
                pila.pop();
                pila.push(true);
            } else {
                if (pila.size() < 2) {
                    cout << "La expresion no es una formula \
                    bien formada" << endl;
                    return;
                }
                pila.pop();
                pila.pop();
                pila.push(true);
            }
        }
    }
    if(pila.size()==1) {
        cout << "La expresion es una formula \
        bien formada" << endl;
        return;
    } else {
        cout << "La expresion no es es una formula \
        bien formada" << endl;
        return;
    }
}
```

Como ya se mencionaba en el capítulo 2, antes de ejecutar el algoritmo en el código 2.1, es necesario un preprocesamiento de la cadena de texto para facilitar el trabajo de la computadora que lo ejecuta, sin perder la estructura original. Después del preprocesamiento se requiere de una estructura de datos conocida como pila para procesarla ya que su naturaleza de ser una estructura Last in, First Out (LIFO) facilita en lenguaje de máquina el procesamiento.

Cabe destacar que, aún cuando se está trabajando directamente con la estructura de datos `string`*, por detrás la manipulación de los datos es a nivel de bits y por eso la importancia de la transformación de texto a cadenas de bits que se explica en el capítulo 4. También, por esta razón, se decidió usar un lenguaje de programación de bajo nivel, como lo es* `C++`*, que es más cercano al lenguaje de máquina que otros, pero sin sacrificar la facilidad que representa tener y usar al menos las estructuras de datos más elementales.*

□

Lema 2.1.2 *El conjunto de fórmulas que son instancias de los axiomas* ($A -$

1), $(A-2)$ y $(A-3)$, del lenguaje $\mathcal{L}^*$, es computable.

Demostración:

Dado un enunciado, mostrar que este sigue la estructura de alguno de los esquemas de los axiomas consiste en fraccionar, en subfórmulas con un grado menos de complejidad, ya que en estas se puede detectar si el enunciado efectivamente sigue alguno de los esquemas o no. Si se toma el axioma (A-1) como ejemplo, se sigue que cualquier enunciado tiene la estructura del esquema si es de la forma $P \Rightarrow (Q \Rightarrow P)$, que al partirlo en sus subfórmulas nos permite destacar que el primero de ellos siempre tiene que ser igual al último. Para hacer este tipo de verificación a nivel de código, el proceso es similar a lo que se ha hecho anteriormente en este trabajo.

La verificación comienza por preprocesar la cadena de texto que guarda al enunciado y transformarla en una estructura de datos del tipo vector, lo que facilita su manipulación. Luego, a través de la función 2.2 se separa estratégicamente la expresión, ignorando todos los caracteres que están entre los paréntesis para detectar cuál es el operador lógico implicación más externo del enunciado y entonces hace una división de la cadena de texto.

Código 2.2: Función que separa un enunciado en dos por su implicación más externa.

```
void split_outer_then(const vector<string>& token_exp,
                      vector<string>& sub_exp_1,
                      vector<string>& sub_exp_2) {
    stack<char> parenthesis_stack;
    int split_index = -1;
    for (size_t i = 0; i < token_exp.size(); i++) {
        if (token_exp[i] == "(") {
            parenthesis_stack.push('(');
        } else if (token_exp[i] == ")") {
            if (!parenthesis_stack.empty()) {
                parenthesis_stack.pop();
            }
        } else if (token_exp[i] == "then" && \
                   parenthesis_stack.empty()) {
            split_index = i;
            break;
        }
    }
    if (split_index == -1) {
        sub_exp_1.push_back("None");
        sub_exp_2.push_back("None");
        return;
    }
    sub_exp_1.assign(token_exp.begin(), \
                     token_exp.begin() + split_index);
    sub_exp_2.assign(token_exp.begin() + split_index + 1, \
                     token_exp.end());

    no_external_parentheses(sub_exp_1);
    no_external_parentheses(sub_exp_2);
}
```

El proceso en el código 2.2 requiere de un sub-proceso auxiliar que manipule los paréntesis en la expresión, con la intención de quitar los paréntesis externos de los enunciados. Con esto se logra des-agrupar los enunciados resultantes después de hacer la separación por la implicación más externa, para eso se construyó la función 2.3.

Código 2.3: Función que remueve los paréntesis externos de un enunciado.

```
void no_external_parentheses(vector<string>& token_exp) {
    if (token_exp.size() >= 2 && token_exp.front() == "(" && \
            token_exp.back() == ")") {
        int balance = 0;
        bool is_wrapped = true;
        for (int i = 0; i < token_exp.size(); i++) {
            if (token_exp[i] == "(") balance++;
            if (token_exp[i] == ")") balance--;
            if (balance == 0 && i < token_exp.size() - 1) {
                is_wrapped = false;
                break;
            }
        }
        if (is_wrapped) {
            token_exp.erase(token_exp.begin());
            token_exp.pop_back();
        }
    }
}
```

□

Lema 2.1.3 *El conjunto de ternas donde la tercera se sigue por Modus Ponens de las otras dos, es computable.*

Demostración:

Con las pruebas de los dos axiomas anteriores se puede seguir hasta el momento que para el lenguaje $\mathcal{L}^*$ existen algoritmos los cuales nos dicen si un enunciado pertenece o no al lenguaje y también si es que este sigue el esquema de alguno de los axiomas (A-1), (A-2) y (A-3). Resta mostrar que dada una terna de enunciados existe un algoritmo que puede determinar si el tercer enunciado de la terna se sigue de los otros dos por Modus Ponens.

Como se puede notar en los lemas anteriores, se necesita de sub-algoritmos para generar el final que realice lo que se requiere. En el caso del código 2.4, se requiere del mismo preprocesamiento de las cadenas de texto para facilitar al lenguaje de programación su entendimiento. En general, la idea del algoritmo es separar el primer enunciado que recibe como entrada en dos, que representarán el antecedente y consecuente, el segundo enunciado que recibe como entrada sería la afirmación, con el cual se comparará el antecedente, la forma de hacer esto es primero tratar cada uno de los enunciados para que no existan caracteres que contaminen los enunciados y de esta forma se obtenga un falso negativo.

Código 2.4: Función que ejecuta Modus Ponens.

```
string modusPonens(string statement, string assertion)
```

```
{
    vector<string> statement_v = string_to_token(statement);
    vector<string> assertion_v = string_to_token(assertion);
    vector<string> antecedent_v;
    vector<string> consequent_v;
    split_outer_then(statement_v, antecedent_v, consequent_v);
    vector<string> postfix_antecedent_v = infix_to_postfix(
        antecedent_v);
    vector<string> postfix_consequent_v = infix_to_postfix(
        consequent_v);
    vector<string> postfix_assertion_v = infix_to_postfix(
        assertion_v);
    if (postfix_antecedent_v.size() != postfix_assertion_v.size()
        ) {
         return "";
    }
    vector<string>::iterator it1 = postfix_antecedent_v.begin();
    vector<string>::iterator it2 = postfix_assertion_v.begin();
    for(;it1 != postfix_antecedent_v.end() &&
         it2 != postfix_assertion_v.end();
         it1++, it2++) {
         if (*it1 != *it2) {
              return "";
         }
    }
    return postfix_to_infix(postfix_consequent_v);
}
```

□

Teorema 2.2 *El conjunto de demostraciones correctas en el sistema $\mathcal{L}^*$ es computable.*

Demostración:

Como se explicaba en el capítulo 2, una demostración formal es una secuencia finita de enunciados que son fórmulas bien formadas del lenguaje $\mathcal{L}^*$, son axiomas o en este caso se siguen de la única regla de inferencia Modus Ponens.

Desde el lema 2.1.1 hasta el 2.1.3 se han presentado algoritmos que permiten mostrar que tanto el conjunto de fórmulas bien formadas, como el conjunto de fórmulas que son instancias de los axiomas $(A-1), (A-2)$ y $(A-3)$ y también que el conjunto de ternas donde la tercera se sigue por Modus Ponens de las otras dos son computables. De esta forma, para mostrar que el conjunto de sucesiones finitas de fórmulas que son demostraciones formales es computable, habría que construir el algoritmo que, dada una sucesión de fórmulas de entrada, sistemáticamente verifica, para cada término de la sucesión, que este sea o bien una instancia de alguno de los axiomas A1, A2 o A3, o bien que se siga de dos términos previos por medio de la regla de Modus Ponens.

Ya en el Capítulo 2, se explicó una forma algorítmica de verificar si un enunciado es o no una fórmula bien formada, lo que ya codificado en el lenguaje de programación definido para este trabajo se codificó en los siguientes códigos y de forma global en apéndice.

El código 2.5, se presentan los pasos que se deben seguir para identificar si un enunciado sigue la estructura del Axioma 1 (A-1) del lenguaje $\mathcal{L}^*$.

Código 2.5: Código que verifica si un enuciado sigue la estructura de A-1

```
string og_exp_str = "Athen(BthenA)";

    // str(exp) -> vector(exp)
    vector<string> og_exp_vec = string_to_token(og_exp_str);

    cout << "Original Expression:" << endl;
    printVector(og_exp_vec);
    cout << endl;

    /*
        1st Split
        [exp1] A
        [exp2] BthenA
    */
    vector<string> exp1, exp2;

    split_outer_then(og_exp_vec, exp1, exp2);

    cout << "1st Split:" << endl;
    printVector(exp1);
    cout << endl;
    printVector(exp2);
    cout << endl;

    /*
        2nd Split
        [exp2_1] B
        [exp2_2] A
    */
    vector<string> exp2_1, exp2_2;
    split_outer_then(exp2, exp2_1, exp2_2);

    cout << "2nd Split:" << endl;
    printVector(exp2_1);
    cout << endl;
    printVector(exp2_2);
    cout << endl;

    vector<string> exp1_postfix = infix_to_postfix(exp1);
    vector<string> exp2_2_postfix = infix_to_postfix(exp2_2);

    if(exp1_postfix == exp2_2_postfix) {
        cout << "Sigue el esquema del Axioma 1" << endl;
    } else {
        cout << "No sigue el esquema del Axioma 1" << endl;
    }
```

En el código 2.6 se presentan los pasos que se deben seguir para identificar si un enunciado sigue la estructura del Axioma 2 (A-2) del lenguaje $\mathcal{L}^*$.

Código 2.6: Código que verifica si un enuciado sigue la estructura de A-2

```
string og_exp_str = "(Athen(BthenC))then((AthenB)then(AthenC))";

    // string(exp) -> vector(exp)
    vector<string> og_exp_vec = string_to_token(og_exp_str);

    cout << "Original Expression:" << endl;
```

```
    printVector(og_exp_vec);
    cout << endl;

    /*
        1st Split:
        [exp1] Athen(BthenC)
        [exp2] (AthenB)then(AthenC)
    */
    vector<string> exp1, exp2;

    split_outer_then(og_exp_vec, exp1, exp2);

    cout << "1st Split:" << endl;
    printVector(exp1);
    cout << endl;
    printVector(exp2);
    cout << endl;

    /*
        2nd Split:
        [exp1_1] A
        [exp1_2] BthenC
    */
   vector<string> exp1_1, exp1_2;

   split_outer_then(exp1, exp1_1, exp1_2);

   cout << "2nd Split:" << endl;
   printVector(exp1_1);
   cout << endl;
   printVector(exp1_2);
   cout << endl;

    /*
        3rd Split:
        [exp1_2_1] B
        [exp1_2_2] C
    */
   vector<string> exp1_2_1, exp1_2_2;

   split_outer_then(exp1_2, exp1_2_1, exp1_2_2);

   cout << "3rd Split:" << endl;
   printVector(exp1_2_1);
   cout << endl;
   printVector(exp1_2_2);
   cout << endl;

    /*
        4th Split:
        [exp2_1] AthenB
        [exp2_2] AthenC
    */
    vector<string> exp2_1, exp2_2;

    split_outer_then(exp2, exp2_1, exp2_2);

    cout << "4th Split:" << endl;
```

```
65      printVector(exp2_1);
66      cout << endl;
67      printVector(exp2_2);
68      cout << endl;
69
70      /*
71          5th Split:
72          [exp2_1_1] A
73          [exp2_1_2] B
74      */
75      vector<string> exp2_1_1, exp2_1_2;
76
77      split_outer_then(exp2_1, exp2_1_1, exp2_1_2);
78
79      cout << "5th Split:" << endl;
80      printVector(exp2_1_1);
81      cout << endl;
82      printVector(exp2_1_2);
83      cout << endl;
84
85      /*
86          6th Split:
87          [exp2_2_1] A
88          [exp2_2_2] C
89      */
90      vector<string> exp2_2_1, exp2_2_2;
91
92      split_outer_then(exp2_2, exp2_2_1, exp2_2_2);
93
94      cout << "6th Split:" << endl;
95      printVector(exp2_2_1);
96      cout << endl;
97      printVector(exp2_2_2);
98      cout << endl;
99
100     // Validation step
101     vector<string> exp1_1_postfix = infix_to_postfix(exp1_1);
102     vector<string> exp2_1_1_postfix = infix_to_postfix(exp2_1_1);
103     vector<string> exp2_2_1_postfix = infix_to_postfix(exp2_2_1);
104     vector<string> exp1_2_1_postfix = infix_to_postfix(exp1_2_1);
105     vector<string> exp2_1_2_postfix = infix_to_postfix(exp2_1_2);
106     vector<string> exp1_2_2_postfix = infix_to_postfix(exp1_2_2);
107     vector<string> exp2_2_2_postfix = infix_to_postfix(exp2_2_2);
108
109     // Partial conditions
110     bool cond1, cond2, cond3;
111     if ((exp1_1_postfix == exp2_1_1_postfix) && (exp2_1_1_postfix
            == exp2_2_1_postfix)) {
112         cond1 = true;
113     } else {
114         cond2 = false;
115     }
116
117     if(exp1_2_1_postfix == exp2_1_2_postfix) {
118         cond2 = true;
119     } else {
120         cond2 = false;
121     }
```

```
122
123     if(exp1_2_2_postfix == exp2_2_2_postfix) {
124         cond3 = true;
125     } else {
126         cond3 = false;
127     }
128
129     // Final condition
130     if(cond1 && cond2 && cond3) {
131         cout << "Sigue el esquema del Axioma 2\n";
132     } else {
133         cout << "No sigue el esquema del Axioma 2\n";
134     }
```

En el código 2.7 se presentan los pasos que se deben seguir para identificar si un enunciado sigue la estructura del Axioma 3 (A-3) del lenguaje $\mathcal{L}^*$.

Código 2.7: Código que verifica si un enuciado sigue la estructura de A-3

```
1
2       string og_exp_str = "(noAthennoB)then(BthenA)";
3
4     // string(exp) -> vector(exp)
5     vector<string> og_exp_vec = string_to_token(og_exp_str);
6
7     cout << "Original Expression:" << endl;
8     printVector(og_exp_vec);
9     cout << endl;
10
11    /*
12        1st Split:
13        [exp1] (no A)then(no B)
14        [exp2] B then A
15    */
16  vector<string> exp1, exp2;
17
18    split_outer_then(og_exp_vec, exp1, exp2);
19
20    cout << "1st Split:" << endl;
21    printVector(exp1);
22    cout << endl;
23    printVector(exp2);
24    cout << endl;
25
26    /*
27        2nd Split:
28        [exp1_1] no A
29        [exp1_2] no B
30    */
31  vector<string> exp1_1, exp1_2;
32
33  split_outer_then(exp1, exp1_1, exp1_2);
34
35  cout << "2nd Split:" << endl;
36  printVector(exp1_1);
37  cout << endl;
38  printVector(exp1_2);
39  cout << endl;
40
```

```
    /*
        3rd Split:
        [exp2_1] B
        [exp2_2] A
    */
    vector<string> exp2_1, exp2_2;

    split_outer_then(exp2, exp2_1, exp2_2);

    cout << "3rd Split:" << endl;
    printVector(exp2_1);
    cout << endl;
    printVector(exp2_2);
    cout << endl;


    if((exp1_1[1] == exp2_2[0]) && (exp1_2[1] == exp2_1[0])) {
        cout << "Sigue el esquema del Axioma 3" << endl;
    } else {
        cout << "No sigue el esquema del Axioma 3" << endl;
    }
```

□

2.5. Conjunto Computablemente Enumerable

El Teorema 2.2 estableció que el conjunto de las demostraciones correctas en $\mathcal{L}^*$ es un conjunto computable. Es tentador concluir que, en consecuencia, el conjunto de los enunciados que poseen al menos una de esas demostraciones (teoremas) también es computable. Sin embargo, esta inferencia es incorrecta. Para comprender por qué, es fundamental introducir el concepto de computablemente enumerable y contrastarlo con el de conjunto computable.

Definición 2.4 (Conjunto Computablemente Enumerable) *Un conjunto es computablemente enumerable si existe una máquina de Turing que al recibir cualquier enunciado se detiene únicamente sieste pertecen al conjunto. En caso contrario, la máquina puede no detenerse jamás.*

También una caracterización de conjunto computablemente enumerable que le da claridad a la definición 2.4, es que existe una Máquina de Turing la cual lista sus elementos, uno por uno, permitiendo que la ejecución continué de forma indefinida. Es crucial notar la diferencia con un conjunto computable, donde la máquina siempre se detiene, dando una respuesta "sí" o "no" en un tiempo finito.

Utilizando el hecho de que el conjunto de demostraciones correctas es computable, se puede construir una máquina de Turing que sistemáticamente enumere todas las secuencias finitas de símbolos. Para cada una, verifica en un tiempo finito si es o no una demostración válida. Cada vez que encuentra una, la máquina imprime el enunciado que está siendo demostrado. De este modo, se listarán todos y sólo los teoremas de $\mathcal{L}^*$. Este proceso no asegura que si un enunciado es un teorema, eventualmente lo encontraremos. Sin embargo, si un enunciado no es un teorema, el algoritmo de enumeración nunca dará esta certeza; simplemente

seguirá buscando sin terminar. No existe un procedimiento algorítmico general que, para cualquier enunciado pueda decidir en un tiempo finito si es un teorema o no.

En resumen, mientras que verificar una demostración concreta es un proceso computable, la tarea de decidir la existencia de una demostración no lo es. Por lo que el lenguaje $\mathcal{L}^*$ pertenece a la clase de los conjuntos computablemente enumerables y no a la de los computables.

APÉNDICE A

Códigos Completos

El presente apéndice contiene las versiones completas del código expuesto en este trabajo, incluyendo aquellas secciones indispensables para la ejecución integral y que fueron omitidas en el cuerpo principal para mantener el foco en las explicaciones.

A.1. Evaluación de Fórmulas Bien Formadas

El código que se presenta a continuación es una implementación diseñada para recibir y procesar una cadena de texto que representa una expresión. Su función es evaluar si dicha expresión cumple con las reglas que definen una fórmula bien formada (FBF).

Código A.1: Código completo que verifica si un enunciado es FBF.

```
#include <iostream>
#include <string>
#include <vector>
#include <sstream>
#include <regex>
#include <stack>
#include <algorithm>

using namespace std;

bool isOperator(string op)
{
    if(op=="no") {
        return true;
    } else if (op=="and") {
        return true;
    } else if (op=="or") {
        return true;
    } else if (op=="then") {
        return true;
    }  else {
        return false;
    }
}

bool isLetter(string palabra)
{
```

```
28          if(palabra.size()==1 && isalpha(palabra[0])) {
29              return true;
30          }
31          return false;
32      }
33
34      int logicPrecedence(string op) {
35          if (op=="no") {
36              return 3;
37          } else if (op=="and") {
38              return 2;
39          } else if (op=="or") {
40              return 1;
41          } else if (op=="then") {
42              return 0;
43          } else {
44              return -1; //No existe el operador
45          }
46      }
47
48      vector<string> tokenizador(string exp)
49      {
50          // Expresion regular para identificar letras y operadores
51          regex pattern("([A-Z]|no|and|or|then|\\(|\\))");
52
53          vector<string> tokens;
54          sregex_iterator it(exp.begin(), exp.end(), pattern);
55          sregex_iterator end;
56
57          while (it != end) {
58              tokens.push_back(it->str());
59              ++it;
60          }
61
62          return tokens;
63      }
64
65
66      vector<string> infixtopostfix(vector<string> tokens)
67      {
68          vector<string> postfix;
69          stack<string> pila;
70
71          for(int i = 0; i < tokens.size(); i++) {
72              if(isLetter(tokens[i])) {
73                  postfix.push_back(tokens[i]);
74              } else if (tokens[i] == "(") {
75                  pila.push(tokens[i]);
76              } else if (tokens[i] == ")") {
77                  while (!pila.empty() && pila.top() != "(") {
78                      postfix.push_back(pila.top());
79                      pila.pop();
80                  }
81                  if (!pila.empty()) {
82                      pila.pop();
83                  } else {
84                      cout << "Error: Parentesis no \
85                      balanceados." << endl;
```

```
86                      return postfix;
87                  }
88              } else if(isOperator(tokens[i])) {
89                  while (!pila.empty() && isOperator(pila.top())) {
90                      if (logicPrecedence(pila.top()) >= \
91                      logicPrecedence(tokens[i])) {
92                          postfix.push_back(pila.top());
93                          pila.pop();
94                      } else {
95                          break;
96                      }
97                  }
98                  pila.push(tokens[i]);
99              }
100         }
101
102         while (!pila.empty()) {
103             postfix.push_back(pila.top());
104             pila.pop();
105         }
106
107         return postfix;
108     }
109
110     void isWFF(vector<string> exp) {
111         stack<bool> pila;
112         for (string token : exp) {
113             if (isLetter(token)) {
114                 pila.push(true);
115             } else if (isOperator(token)) {
116                 if(token=="no") {
117                     if(pila.empty()) {
118                         cout << "La expresion no es una formula \
119                         bien formada" << endl;
120                         return;
121                     }
122                     pila.pop();
123                     pila.push(true);
124                 } else {
125                     if (pila.size()<2) {
126                         cout << "La expresion no es una formula \
127                         bien formada" << endl;
128                         return;
129                     }
130                     pila.pop();
131                     pila.pop();
132                     pila.push(true);
133                 }
134             }
135         }
136         if(pila.size()==1) {
137             cout << "La expresion es una formula bien formada" <<
                    endl;
138             return;
139         } else {
140             cout << "La expresion no es es una formula bien
                   formada" << endl;
141             return;
```

```
142         }
143     }
144
145     int main()
146     {
147         string input;
148         getline(cin, input);
149         vector<string> tokens=tokenizador(input);
150         vector<string> postfix=infixtopostfix(tokens);
151         cout << "La expresion a evaluar es: " << input << endl;
152         cout << "En notacion polaca es: ";
153         for(int i=0; i<postfix.size(); i++) {
154             cout << postfix[i] << " ";
155         }
156         cout << endl;
157
158         isWFF(postfix);
159
160         return 0;
161     }
```

A.2. Evaluación de Tablas de Verdad

El código a continuación es una implementación diseñada para evaluar las tablas de verdad de expresiones.

Código A.2: Código completo para evaluar tablas de verdad.

```
1     #include <iostream>
2     #include <string>
3     #include <vector>
4     #include <sstream>
5     #include <regex>
6     #include <stack>
7     #include <algorithm>
8
9     using namespace std;
10
11    bool isOperator(string op) {
12
13        return (op == "no" || op == "and" || op == "or" \
14                || op == "then" || op == "iff");
15
16    }
17
18    bool isLetter(string palabra) {
19
20        return (palabra.size() == 1 && isalpha(palabra[0]));
21
22    }
23
24    int logicPrecedence(string op) {
25
26        if (op == "no") {
27            return 3;
28        } else if (op == "and") {
```

```
29              return 2;
30          } else if (op == "or") {
31              return 1;
32          } else if ((op == "then") || (op == "iff")) {
33              return 0;
34          } else  {
35              return -1;
36          }
37
38      }
39
40      vector<string> tokenizador(string exp) {
41
42          regex pattern("([A-Z]|no|and|or|then|iff|\\(|\\))");
43
44          vector<string> tokens;
45          sregex_iterator it(exp.begin(), exp.end(), pattern);
46          sregex_iterator end;
47
48          while (it != end) {
49              tokens.push_back(it->str());
50              ++it;
51          }
52
53          return tokens;
54      }
55
56      vector<string> infixtopostfix(vector<string> tokens) {
57
58          vector<string> postfix;
59          stack<string> pila;
60
61          for (int i = 0; i < tokens.size(); i++) {
62              if (isLetter(tokens[i])) {
63                  postfix.push_back(tokens[i]);
64              } else if (tokens[i] == "(") {
65                  pila.push(tokens[i]);
66              } else if (tokens[i] == ")") {
67                  while (!pila.empty() && pila.top() != "(") {
68                      postfix.push_back(pila.top());
69                      pila.pop();
70                  }
71                  if (!pila.empty()) {
72                      pila.pop();
73                  } else {
74                      cout << "Error: Parentesis no balanceados."
                            << endl;
75                      return postfix;
76                  }
77              } else if (isOperator(tokens[i])) {
78                  while (!pila.empty() && isOperator(pila.top())) {
79                      if (logicPrecedence(pila.top()) >=
                            logicPrecedence(tokens[i])) {
80                          postfix.push_back(pila.top());
81                          pila.pop();
82                      } else {
83                          break;
84                      }
```

```
85                  }
86                  pila.push(tokens[i]);
87              }
88          }
89
90          while (!pila.empty()) {
91              postfix.push_back(pila.top());
92              pila.pop();
93          }
94
95          return postfix;
96      }
97
98      bool val(vector<string> exp, vector<bool> intrs) {
99
100         stack<bool> pila;
101
102         for (string token : exp) {
103             if (isLetter(token)) {
104                 int index = token[0] - 'A';
105                 pila.push(intrs[index]);
106             } else if (isOperator(token)) {
107                 if(token == "no") {
108                     if(pila.empty()) {
109                         cout << "Error: No hay suficientes
                                operandos" << endl;
110                         exit;
111                     }
112                     bool operando = pila.top();
113                     pila.pop();
114                     pila.push(!operando);
115                 } else {
116                     if (pila.size() < 2) {
117                         cout << "Error: No hay suficientes
                                operandos" << endl;
118                         exit;
119                     }
120                     bool operando2 = pila.top();
121                     pila.pop();
122                     bool operando1 = pila.top();
123                     pila.pop();
124                     if(token == "and") {
125                         pila.push(operando1 && operando2);
126                     } else if (token == "or") {
127                         pila.push(operando1 || operando2);
128                     } else if (token == "then") {
129                         pila.push(!operando1 || operando2);
130                     } else if (token == "iff") {
131                         pila.push((!operando1 || operando2) && \
132                                 (!operando2 || operando1));
133                     }
134                 }
135             }
136         }
137
138         return pila.top();
139
140     }
```

```
141
142     void TruthTable(vector<string> exp, int numVar) {
143
144         vector<vector<bool> > truthTable;
145         vector<bool> intrs(numVar, false);
146
147         do {
148             vector<bool> row = intrs;
149             truthTable.push_back(row);
150
151             bool result = val(exp, intrs);
152
153             for (bool i : intrs) {
154                 if(i == false) {
155                     cout << "F ";
156                 } else {
157                     cout << "V ";
158                 }
159             }
160
161             if(result == false) {
162                 cout << "| F " << endl;
163             } else {
164                 cout << "| V " << endl;
165             }
166
167             for (int i = 0; i < numVar; i++) {
168                 if (intrs[i]) {
169                     intrs [i] = false;
170                 } else {
171                     intrs [i] = true;
172                     break;
173                 }
174             }
175         } while (count(intrs.begin(), intrs.end(), false) <
                numVar);
176     }
177
178     int cont_var(vector<string> exp) {
179
180         int sum = 0;
181         vector<string>::iterator it;
182
183         for(it = exp.begin(); it != exp.end(); it++) {
184             if(isLetter(*it)) {
185                 sum++;
186             }
187         }
188
189         return sum;
190
191     }
192
193     int main() {
194         string input;
195
196         cout << "Ingrese la expresion logica: ";
197         getline(cin, input);
```

```
198
199         vector<string> tokens = tokenizador(input);
200
201         int numVar = cont_var(tokens);
202
203         vector<string> postfix=infixtopostfix(tokens);
204
205         cout << "Tabla de verdad para la expresion \"" << input
                << "\":" << endl;
206         cout << "----------" << endl;
207         TruthTable(postfix, numVar);
208
209         return 0;
210     }
```

A.3. Evaluación de Modus Ponens

El código a continuación es una implementación diseñada para ejecutar la regla de inferencia Modus Ponens entre dos expresiones dentro del lenguaje $\mathcal{L}^*$.

Código A.3: Código completo para ejecutar la regla de inferencia Modus Ponens.

```
1     #include <iostream>
2     #include <string>
3     #include <vector>
4     #include <sstream>
5     #include <regex>
6     #include <stack>
7     #include <algorithm>
8
9     using namespace std;
10
11    vector<string> string_to_token(string exp)
12    {
13        regex pattern("([A-Z]|no|then|\\(|\\))");
14
15        vector<string> tokens;
16        sregex_iterator it(exp.begin(), exp.end(), pattern);
17        sregex_iterator end;
18
19        while (it != end) {
20            tokens.push_back(it->str());
21            ++it;
22        }
23
24        return tokens;
25    }
26
27    bool isOperator(string op)
28    {
29        if (op == "no" || op == "then") {
30            return true;
31        } else {
32            return false;
33        }
34    }
```

```
35
36      int logicPrecedence(string op)
37      {
38          if (op == "no") {
39              return 1;
40          }  else if (op == "then") {
41              return 0;
42          } else {
43              return -1;
44          }
45      }
46
47      bool isAtomic(string exp)
48      {
49          if (exp.size()==1 && isalpha(exp[0])) {
50              return true;
51          }
52          return false;
53      }
54
55      vector<string> infix_to_postfix(vector<string> infix_str)
56      {
57          vector<string>::iterator str;
58          vector<string> postfix_str;
59          stack<string> pila;
60
61          for(str = infix_str.begin(); str != infix_str.end(); str
               ++) {
62              if (isAtomic(*str)) {
63                  postfix_str.push_back(*str);
64              } else if (*str == "(") {
65                  pila.push(*str);
66              } else if (*str == ")") {
67                  while (!pila.empty() && pila.top() != "(") {
68                      postfix_str.push_back(pila.top());
69                      pila.pop();
70                  }
71                  if (!pila.empty()) {
72                      pila.pop();
73                  } else {
74                      cout << "Error: Parentesis no balanceados."
                                << endl;
75                      return postfix_str;
76                  }
77              } else if (isOperator(*str)) {
78                  while (!pila.empty() && isOperator(pila.top())) {
79                      if (logicPrecedence(pila.top()) >=
                            logicPrecedence(*str)) {
80                          postfix_str.push_back(pila.top());
81                          pila.pop();
82                      } else {
83                          break;
84                      }
85                  }
86                  pila.push(*str);
87              }
88          }
89
```

```
90          while (!pila.empty()) {
91              postfix_str.push_back(pila.top());
92              pila.pop();
93          }
94
95          return postfix_str;
96      }
97
98      string postfix_to_infix(vector<string> postfix_str)
99      {
100         vector<string>::iterator str;
101         stack<string> infix_str;
102
103         for (str = postfix_str.begin(); str != postfix_str.end();
                str++) {
104             if (isOperator(*str)) {
105                 if (*str == "no") {
106                     if (infix_str.size() < 1) {
107                         cout << "Error: Formato postfix invalido
                                .." << endl;
108                         return "";
109                     }
110                     string op = infix_str.top();
111                     infix_str.pop();
112                     string expr = "(no " + op + ")";
113                     infix_str.push(expr);
114                 } else {
115                     if (infix_str.size() < 2) {
116                         cout << "Error: Formato postfix invalido
                                .." << endl;
117                         return "";
118                     }
119                     string op2 = infix_str.top();
120                     infix_str.pop();
121                     string op1 = infix_str.top();
122                     infix_str.pop();
123                     string expr = "(" + op1 + " " + *str + " " +
                            op2 + ")";
124                     infix_str.push(expr);
125                 }
126             } else {
127                 infix_str.push(*str);
128             }
129         }
130         return infix_str.top();
131     }
132
133     void no_external_parentheses(vector<string>& token_exp)
134     {
135         vector<string>::iterator str;
136         for (str = token_exp.begin(); str != token_exp.end();) {
137             if (*str == "(" && str == token_exp.begin()) {
138                 token_exp.erase(str);
139             } else if (*str == ")" && std::next(str) == token_exp
                    .end()) {
140                 token_exp.erase(str);
141             } else {
142                 ++str;
```

```
143             }
144         }
145     }
146 
147     void split_outer_then(vector<string>& token_exp,
148                           vector<string>& sub_exp_1,
149                           vector<string>& sub_exp_2)
150     {
151         vector<string>::iterator str;
152         vector<string>::iterator split_index = token_exp.end();
153         stack<string> parenthesis_stack;
154 
155         for (str = token_exp.begin(); str != token_exp.end(); ++
               str) {
156             if (*str == "(") {
157                 parenthesis_stack.push(*str);
158             } else if (*str == ")") {
159                 if (!parenthesis_stack.empty()) {
160                     parenthesis_stack.pop();
161                 }
162             } else if (*str == "then") {
163                 if (parenthesis_stack.empty()) {
164                     split_index = str;
165                     break;
166                 }
167             }
168         }
169 
170         if (split_index == token_exp.end()) {
171             sub_exp_1.push_back("None");
172             sub_exp_2.push_back("None");
173             return;
174         }
175 
176         vector<string>::iterator str1;
177 
178         for(str1 = token_exp.begin(); str1 != split_index; str1
               ++) {
179             sub_exp_1.push_back(*str1);
180         }
181 
182         no_external_parentheses(sub_exp_1);
183 
184         vector<string>::iterator str2;
185 
186         for(str2 = split_index + 1; str2 != token_exp.end(); str2
               ++) {
187             sub_exp_2.push_back(*str2);
188         }
189 
190         no_external_parentheses(sub_exp_2);
191     }
192 
193     string modusPonens(string statement, string assertion)
194     {
195         vector<string> statement_v = string_to_token(statement);
196         vector<string> assertion_v = string_to_token(assertion);
197 
```

```
198         vector<string> antecedent_v;
199         vector<string> consequent_v;
200 
201         split_outer_then(statement_v, antecedent_v, consequent_v)
                ;
202 
203         vector<string> postfix_antecedent_v = infix_to_postfix(
                antecedent_v);
204         vector<string> postfix_consequent_v = infix_to_postfix(
                consequent_v);
205         vector<string> postfix_assertion_v = infix_to_postfix(
                assertion_v);
206 
207         if (postfix_antecedent_v.size() != postfix_assertion_v.
                size()) {
208             return "";
209         }
210 
211         vector<string>::iterator it1 = postfix_antecedent_v.begin
                ();
212         vector<string>::iterator it2 = postfix_assertion_v.begin
                ();
213 
214         for(;it1 != postfix_antecedent_v.end() && it2 !=
                postfix_assertion_v.end();
215             it1++, it2++) {
216             if (*it1 != *it2) {
217                 return "";
218             }
219         }
220 
221         return postfix_to_infix(postfix_consequent_v);
222 
223     }
224 
225     int main() {
226         string statement, assertion;
227 
228         cout << "Statement: " << endl;
229         getline(cin, statement);
230 
231         cout << "Assertion: " << endl;
232         getline(cin, assertion);
233 
234         string consequent = modusPonens(statement, assertion);
235 
236         if (consequent == "") {
237             cout << "False\nConsequent:\n";
238         } else {
239             cout << "True \nConsequent: "<< consequent << endl;
240         }
241 
242         return 0;
243     }
```

A.4. Código completo del Axioma 1

El código a continuación es una implementación diseñada para evaluar cuando una expresión en el lenguaje formal sigue el esquema del Axioma 1 que estructura al lenguaje $\mathcal{L}^*$.

Código A.4: Código completo para evaular si una expresión cumple el Axioma 1.

```
#include <iostream>
#include <vector>
#include <string>
#include <stack>
#include <regex>

using namespace std;

vector<string> string_to_token(string exp) {
    regex pattern("([A-Z]|no|then|\\(|\\))");
    vector<string> tokens;
    sregex_iterator it(exp.begin(), exp.end(), pattern);
    sregex_iterator end;

    while (it != end) {
        tokens.push_back(it->str());
        ++it;
    }

    return tokens;
}

bool isOperator(string op)
{
    if (op == "no" || op == "then") {
        return true;
    } else {
        return false;
    }
}

int logicPrecedence(string op)
{
    if (op == "no") {
        return 1;
    }  else if (op == "then") {
        return 0;
    } else {
        return -1;
    }
}

bool isAtomic(string exp)
{
    if (exp.size()==1 && isalpha(exp[0])) {
        return true;
    }
    return false;
}

```

```
51      vector<string> infix_to_postfix(vector<string> infix_str)
52      {
53          vector<string>::iterator str;
54          vector<string> postfix_str;
55          stack<string> pila;
56
57          for(str = infix_str.begin(); str != infix_str.end(); str
               ++) {
58              if (isAtomic(*str)) {
59                  postfix_str.push_back(*str);
60              } else if (*str == "(") {
61                  pila.push(*str);
62              } else if (*str == ")") {
63                  while (!pila.empty() && pila.top() != "(") {
64                      postfix_str.push_back(pila.top());
65                      pila.pop();
66                  }
67                  if (!pila.empty()) {
68                      pila.pop();
69                  } else {
70                      cout << "Error: Parentesis no balanceados."
                           << endl;
71                      return postfix_str;
72                  }
73              } else if (isOperator(*str)) {
74                  while (!pila.empty() && isOperator(pila.top())) {
75                      if (logicPrecedence(pila.top()) >=
                           logicPrecedence(*str)) {
76                          postfix_str.push_back(pila.top());
77                          pila.pop();
78                      } else {
79                          break;
80                      }
81                  }
82                  pila.push(*str);
83              }
84          }
85
86          while (!pila.empty()) {
87              postfix_str.push_back(pila.top());
88              pila.pop();
89          }
90
91          return postfix_str;
92      }
93
94      void no_external_parentheses(vector<string>& token_exp) {
95          if (token_exp.size() >= 2 && token_exp.front() == "(" &&
               \
96                  token_exp.back() == ")") {
97              int balance = 0;
98              bool is_wrapped = true;
99
100             for (int i = 0; i < token_exp.size(); i++) {
101                 if (token_exp[i] == "(") balance++;
102                 if (token_exp[i] == ")") balance--;
103                 if (balance == 0 && i < token_exp.size() - 1) {
104                     is_wrapped = false;
```

```
                    break;
                }
            }

            if (is_wrapped) {
                token_exp.erase(token_exp.begin());
                token_exp.pop_back();
            }
        }
    }

    void split_outer_then(const vector<string>& token_exp,
                          vector<string>& sub_exp_1,
                          vector<string>& sub_exp_2) {
        stack<char> parenthesis_stack;
        int split_index = -1;

        for (size_t i = 0; i < token_exp.size(); i++) {
            if (token_exp[i] == "(") {
                parenthesis_stack.push('(');
            } else if (token_exp[i] == ")") {
                if (!parenthesis_stack.empty()) {
                    parenthesis_stack.pop();
                }
            } else if (token_exp[i] == "then" && \
                        parenthesis_stack.empty()) {
                split_index = i;
                break;
            }
        }

        if (split_index == -1) {
            sub_exp_1.push_back("None");
            sub_exp_2.push_back("None");
            return;
        }
        sub_exp_1.assign(token_exp.begin(), \
                         token_exp.begin() + split_index);
        sub_exp_2.assign(token_exp.begin() + split_index + 1, \
                         token_exp.end());

        no_external_parentheses(sub_exp_1);
        no_external_parentheses(sub_exp_2);
    }

    void printVector(vector<string> exp_vec)
    {
        int vec_size = exp_vec.size();
        cout << "{";
        for (int i=0; i<vec_size; i++) {
            if(i < vec_size - 1) {
                cout << exp_vec[i] << ", ";
            } else {
                cout << exp_vec[i] << "}";
            }
        }

        return;
```

```
163     }
164
165     int main() {
166         string og_exp_str = "Athen(BthenA)";
167
168         // str(exp) -> vector(exp)
169         vector<string> og_exp_vec = string_to_token(og_exp_str);
170
171         cout << "Original Expression:" << endl;
172         printVector(og_exp_vec);
173         cout << endl;
174
175         /*
176             1st Split
177             [exp1] A
178             [exp2] BthenA
179         */
180         vector<string> exp1, exp2;
181
182         split_outer_then(og_exp_vec, exp1, exp2);
183
184         cout << "1st Split:" << endl;
185         printVector(exp1);
186         cout << endl;
187         printVector(exp2);
188         cout << endl;
189
190         /*
191             2nd Split
192             [exp2_1] B
193             [exp2_2] A
194         */
195         vector<string> exp2_1, exp2_2;
196         split_outer_then(exp2, exp2_1, exp2_2);
197
198         cout << "2nd Split:" << endl;
199         printVector(exp2_1);
200         cout << endl;
201         printVector(exp2_2);
202         cout << endl;
203
204         vector<string> exp1_postfix = infix_to_postfix(exp1);
205         vector<string> exp2_2_postfix = infix_to_postfix(exp2_2);
206
207         if(exp1_postfix == exp2_2_postfix) {
208             cout << "Sigue el esquema del Axioma 1" << endl;
209         } else {
210             cout << "No sigue el esquema del Axioma 1" << endl;
211         }
212
213         return 0;
214     }
```

A.5. Código completo del Axioma 2

El código a continuación es una implementación diseñada para evaluar cuando una expresión en el lenguaje formal sigue el esquema del Axioma 2 que estructura al lenguaje $\mathcal{L}^*$.

Código A.5: Código completo para evaular si una expresión cumple el Axioma 2.

```
1
2     #include <iostream>
3     #include <vector>
4     #include <string>
5     #include <stack>
6     #include <regex>
7
8     using namespace std;
9
10    vector<string> string_to_token(string exp) {
11        regex pattern("([A-Z]|no|then|\\(|\\))");
12        vector<string> tokens;
13        sregex_iterator it(exp.begin(), exp.end(), pattern);
14        sregex_iterator end;
15
16        while (it != end) {
17            tokens.push_back(it->str());
18            ++it;
19        }
20
21        return tokens;
22    }
23
24    bool isOperator(string op)
25    {
26        if (op == "no" || op == "then") {
27            return true;
28        } else {
29            return false;
30        }
31    }
32
33    int logicPrecedence(string op)
34    {
35        if (op == "no") {
36            return 1;
37        }  else if (op == "then") {
38            return 0;
39        } else {
40            return -1;
41        }
42    }
43
44    bool isAtomic(string exp)
45    {
46        if (exp.size()==1 && isalpha(exp[0])) {
47            return true;
48        }
49        return false;
50    }
```

```
51
52      vector<string> infix_to_postfix(vector<string> infix_str)
53      {
54          vector<string>::iterator str;
55          vector<string> postfix_str;
56          stack<string> pila;
57
58          for(str = infix_str.begin(); str != infix_str.end(); str
               ++) {
59              if (isAtomic(*str)) {
60                  postfix_str.push_back(*str);
61              } else if (*str == "(") {
62                  pila.push(*str);
63              } else if (*str == ")") {
64                  while (!pila.empty() && pila.top() != "(") {
65                      postfix_str.push_back(pila.top());
66                      pila.pop();
67                  }
68                  if (!pila.empty()) {
69                      pila.pop();
70                  } else {
71                      cout << "Error: Parentesis no balanceados."
                           << endl;
72                      return postfix_str;
73                  }
74              } else if (isOperator(*str)) {
75                  while (!pila.empty() && isOperator(pila.top())) {
76                      if (logicPrecedence(pila.top()) >=
                           logicPrecedence(*str)) {
77                          postfix_str.push_back(pila.top());
78                          pila.pop();
79                      } else {
80                          break;
81                      }
82                  }
83                  pila.push(*str);
84              }
85          }
86
87          while (!pila.empty()) {
88              postfix_str.push_back(pila.top());
89              pila.pop();
90          }
91
92          return postfix_str;
93      }
94
95      void no_external_parentheses(vector<string>& token_exp) {
96          if (token_exp.size() >= 2 && token_exp.front() == "(" &&
               \
97                  token_exp.back() == ")") {
98              int balance = 0;
99              bool is_wrapped = true;
100
101             for (int i = 0; i < token_exp.size(); i++) {
102                 if (token_exp[i] == "(") balance++;
103                 if (token_exp[i] == ")") balance--;
104                 if (balance == 0 && i < token_exp.size() - 1) {
```

```
                    is_wrapped = false;
                    break;
                }
            }

            if (is_wrapped) {
                token_exp.erase(token_exp.begin());
                token_exp.pop_back();
            }
        }
    }

    void split_outer_then(const vector<string>& token_exp,
                          vector<string>& sub_exp_1,
                          vector<string>& sub_exp_2) {
        stack<char> parenthesis_stack;
        int split_index = -1;

        for (size_t i = 0; i < token_exp.size(); i++) {
            if (token_exp[i] == "(") {
                parenthesis_stack.push('(');
            } else if (token_exp[i] == ")") {
                if (!parenthesis_stack.empty()) {
                    parenthesis_stack.pop();
                }
            } else if (token_exp[i] == "then" && \
                        parenthesis_stack.empty()) {
                split_index = i;
                break;
            }
        }

        if (split_index == -1) {
            sub_exp_1.push_back("None");
            sub_exp_2.push_back("None");
            return;
        }
        sub_exp_1.assign(token_exp.begin(), \
                         token_exp.begin() + split_index);
        sub_exp_2.assign(token_exp.begin() + split_index + 1, \
                         token_exp.end());

        no_external_parentheses(sub_exp_1);
        no_external_parentheses(sub_exp_2);
    }

    void printVector(vector<string> exp_vec)
    {
        int vec_size = exp_vec.size();
        cout << "{";
        for (int i=0; i<vec_size; i++) {
            if(i < vec_size - 1) {
                cout << exp_vec[i] << ", ";
            } else {
                cout << exp_vec[i] << "}";
            }
        }

```

```
163         return;
164     }
165 
166     int main() {
167         string og_exp_str = "(Athen(BthenC))then((AthenB)then(
                AthenC))";
168 
169         // string(exp) -> vector(exp)
170         vector<string> og_exp_vec = string_to_token(og_exp_str);
171 
172         cout << "Original Expression:" << endl;
173         printVector(og_exp_vec);
174         cout << endl;
175 
176         /*
177             1st Split:
178             [exp1] Athen(BthenC)
179             [exp2] (AthenB)then(AthenC)
180         */
181         vector<string> exp1, exp2;
182 
183         split_outer_then(og_exp_vec, exp1, exp2);
184 
185         cout << "1st Split:" << endl;
186         printVector(exp1);
187         cout << endl;
188         printVector(exp2);
189         cout << endl;
190 
191         /*
192             2nd Split:
193             [exp1_1] A
194             [exp1_2] BthenC
195         */
196     vector<string> exp1_1, exp1_2;
197 
198     split_outer_then(exp1, exp1_1, exp1_2);
199 
200     cout << "2nd Split:" << endl;
201     printVector(exp1_1);
202     cout << endl;
203     printVector(exp1_2);
204     cout << endl;
205 
206         /*
207             3rd Split:
208             [exp1_2_1] B
209             [exp1_2_2] C
210         */
211     vector<string> exp1_2_1, exp1_2_2;
212 
213     split_outer_then(exp1_2, exp1_2_1, exp1_2_2);
214 
215     cout << "3rd Split:" << endl;
216     printVector(exp1_2_1);
217     cout << endl;
218     printVector(exp1_2_2);
219     cout << endl;
```

```
220
221         /*
222             4th Split:
223             [exp2_1] AthenB
224             [exp2_2] AthenC
225         */
226         vector<string> exp2_1, exp2_2;
227
228         split_outer_then(exp2, exp2_1, exp2_2);
229
230         cout << "4th Split:" << endl;
231         printVector(exp2_1);
232         cout << endl;
233         printVector(exp2_2);
234         cout << endl;
235
236         /*
237             5th Split:
238             [exp2_1_1] A
239             [exp2_1_2] B
240         */
241         vector<string> exp2_1_1, exp2_1_2;
242
243         split_outer_then(exp2_1, exp2_1_1, exp2_1_2);
244
245         cout << "5th Split:" << endl;
246         printVector(exp2_1_1);
247         cout << endl;
248         printVector(exp2_1_2);
249         cout << endl;
250
251         /*
252             6th Split:
253             [exp2_2_1] A
254             [exp2_2_2] C
255         */
256         vector<string> exp2_2_1, exp2_2_2;
257
258         split_outer_then(exp2_2, exp2_2_1, exp2_2_2);
259
260         cout << "6th Split:" << endl;
261         printVector(exp2_2_1);
262         cout << endl;
263         printVector(exp2_2_2);
264         cout << endl;
265
266         // Validation step
267         vector<string> exp1_1_postfix = infix_to_postfix(exp1_1);
268         vector<string> exp2_1_1_postfix = infix_to_postfix(
               exp2_1_1);
269         vector<string> exp2_2_1_postfix = infix_to_postfix(
               exp2_2_1);
270         vector<string> exp1_2_1_postfix = infix_to_postfix(
               exp1_2_1);
271         vector<string> exp2_1_2_postfix = infix_to_postfix(
               exp2_1_2);
272         vector<string> exp1_2_2_postfix = infix_to_postfix(
               exp1_2_2);
```

```
273         vector<string> exp2_2_2_postfix = infix_to_postfix(
                exp2_2_2);
274 
275         // Partial conditions
276         bool cond1, cond2, cond3;
277         if ((exp1_1_postfix == exp2_1_1_postfix) && (
                exp2_1_1_postfix == exp2_2_1_postfix)) {
278             cond1 = true;
279         } else {
280             cond2 = false;
281         }
282 
283         if(exp1_2_1_postfix == exp2_1_2_postfix) {
284             cond2 = true;
285         } else {
286             cond2 = false;
287         }
288 
289         if(exp1_2_2_postfix == exp2_2_2_postfix) {
290             cond3 = true;
291         } else {
292             cond3 = false;
293         }
294 
295         // Final condition
296         if(cond1 && cond2 && cond3) {
297             cout << "Sigue el esquema del Axioma 2\n";
298         } else {
299             cout << "No sigue el esquema del Axioma 2\n";
300         }
301 
302         return 0;
303     }
```

A.6. Código completo del Axioma 3

El código a continuación es una implementación diseñada para evaluar cuando una expresión en el lenguaje formal sigue el esquema del Axioma 3 que estructura al lenguaje $\mathcal{L}^*$.

Código A.6: Código completo para evaular si una expresión cumple el Axioma 3.

```
1     #include <iostream>
2     #include <vector>
3     #include <string>
4     #include <stack>
5     #include <regex>
6 
7     using namespace std;
8 
9     vector<string> string_to_token(string exp) {
10        regex pattern("([A-Z]|no|then|\\(|\\))");
11        vector<string> tokens;
12        sregex_iterator it(exp.begin(), exp.end(), pattern);
13        sregex_iterator end;
14 
```

```
15          while (it != end) {
16              tokens.push_back(it->str());
17              ++it;
18          }
19
20          return tokens;
21      }
22
23      bool isOperator(string op)
24      {
25          if (op == "no" || op == "then") {
26              return true;
27          } else {
28              return false;
29          }
30      }
31
32      int logicPrecedence(string op)
33      {
34          if (op == "no") {
35              return 1;
36          }  else if (op == "then") {
37              return 0;
38          } else {
39              return -1;
40          }
41      }
42
43      bool isAtomic(string exp)
44      {
45          if (exp.size()==1 && isalpha(exp[0])) {
46              return true;
47          }
48          return false;
49      }
50
51      vector<string> infix_to_postfix(vector<string> infix_str)
52      {
53          vector<string>::iterator str;
54          vector<string> postfix_str;
55          stack<string> pila;
56
57          for(str = infix_str.begin(); str != infix_str.end(); str
                ++) {
58              if (isAtomic(*str)) {
59                  postfix_str.push_back(*str);
60              } else if (*str == "(") {
61                  pila.push(*str);
62              } else if (*str == ")") {
63                  while (!pila.empty() && pila.top() != "(") {
64                      postfix_str.push_back(pila.top());
65                      pila.pop();
66                  }
67                  if (!pila.empty()) {
68                      pila.pop();
69                  } else {
70                      cout << "Error: Parentesis no balanceados."
                            << endl;
```

```
71                      return postfix_str;
72                  }
73              } else if (isOperator(*str)) {
74                  while (!pila.empty() && isOperator(pila.top())) {
75                      if (logicPrecedence(pila.top()) >=
                            logicPrecedence(*str)) {
76                          postfix_str.push_back(pila.top());
77                          pila.pop();
78                      } else {
79                          break;
80                      }
81                  }
82                  pila.push(*str);
83              }
84          }
85
86          while (!pila.empty()) {
87              postfix_str.push_back(pila.top());
88              pila.pop();
89          }
90
91          return postfix_str;
92      }
93
94      void no_external_parentheses(vector<string>& token_exp) {
95          if (token_exp.size() >= 2 && token_exp.front() == "(" &&
                \
96                  token_exp.back() == ")") {
97              int balance = 0;
98              bool is_wrapped = true;
99
100             for (int i = 0; i < token_exp.size(); i++) {
101                 if (token_exp[i] == "(") balance++;
102                 if (token_exp[i] == ")") balance--;
103                 if (balance == 0 && i < token_exp.size() - 1) {
104                     is_wrapped = false;
105                     break;
106                 }
107             }
108
109             if (is_wrapped) {
110                 token_exp.erase(token_exp.begin());
111                 token_exp.pop_back();
112             }
113         }
114     }
115
116     void split_outer_then(const vector<string>& token_exp,
117                           vector<string>& sub_exp_1,
118                           vector<string>& sub_exp_2) {
119         stack<char> parenthesis_stack;
120         int split_index = -1;
121
122         for (size_t i = 0; i < token_exp.size(); i++) {
123             if (token_exp[i] == "(") {
124                 parenthesis_stack.push('(');
125             } else if (token_exp[i] == ")") {
126                 if (!parenthesis_stack.empty()) {
```

```
                    parenthesis_stack.pop();
                }
            } else if (token_exp[i] == "then" && \
                        parenthesis_stack.empty()) {
                split_index = i;
                break;
            }
        }

        if (split_index == -1) {
            sub_exp_1.push_back("None");
            sub_exp_2.push_back("None");
            return;
        }
        sub_exp_1.assign(token_exp.begin(), \
                         token_exp.begin() + split_index);
        sub_exp_2.assign(token_exp.begin() + split_index + 1, \
                         token_exp.end());

        no_external_parentheses(sub_exp_1);
        no_external_parentheses(sub_exp_2);
    }

    void printVector(vector<string> exp_vec)
    {
        int vec_size = exp_vec.size();
        cout << "{";
        for (int i=0; i<vec_size; i++) {
            if(i < vec_size - 1) {
                cout << exp_vec[i] << ", ";
            } else {
                cout << exp_vec[i] << "}";
            }
        }

        return;
    }

    int main() {
        string og_exp_str = "(noAthennoB)then(BthenA)";

        // string(exp) -> vector(exp)
        vector<string> og_exp_vec = string_to_token(og_exp_str);

        cout << "Original Expression:" << endl;
        printVector(og_exp_vec);
        cout << endl;

        /*
            1st Split:
            [exp1] (no A)then(no B)
            [exp2] B then A
        */
    vector<string> exp1, exp2;

        split_outer_then(og_exp_vec, exp1, exp2);

        cout << "1st Split:" << endl;
```

```
185         printVector(exp1);
186         cout << endl;
187         printVector(exp2);
188         cout << endl;
189 
190         /*
191             2nd Split:
192             [exp1_1] no A
193             [exp1_2] no B
194         */
195     vector<string> exp1_1, exp1_2;
196 
197     split_outer_then(exp1, exp1_1, exp1_2);
198 
199     cout << "2nd Split:" << endl;
200     printVector(exp1_1);
201     cout << endl;
202     printVector(exp1_2);
203     cout << endl;
204 
205         /*
206             3rd Split:
207             [exp2_1] B
208             [exp2_2] A
209         */
210     vector<string> exp2_1, exp2_2;
211 
212     split_outer_then(exp2, exp2_1, exp2_2);
213 
214     cout << "3rd Split:" << endl;
215     printVector(exp2_1);
216     cout << endl;
217     printVector(exp2_2);
218     cout << endl;
219 
220     if((exp1_1[1] == exp2_2[0]) && (exp1_2[1] == exp2_1[0])) {
221             cout << "Sigue el esquema del Axioma 3" << endl;
222         } else {
223             cout << "No sigue el esquema del Axioma 3" << endl;
224         }
225 
226     return 0;
227     }
```

Bibliografía

Copeland, B. J. (2024). The Church–Turing Thesis [Consultado el 24 de octubre de 2022]. En *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/win2024/entries/church-turing/

Copi, I. M. (1979). *Symbolic Logic*. Macmillan Publishing.

Enderton, H. B. (2001). *A Mathematical Introduction to Logic*. Academic Press.

Immerman, N. (2021). Computability and Complexity [Consultado el 24 de octubre de 2022]. En *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/win2021/entries/computability/

Psychpsyo. (2022). ASCII-Table-wide [Consultado el 24 de octubre de 2022]. https://commons.wikimedia.org/wiki/File:ASCII-Table-wide.svg

Simons, P. (2023). Jan Łukasiewicz [Consultado el 24 de octubre de 2022]. En *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/spr2023/entries/lukasiewicz/

Weber, R. (2012). *Computability Theory*. American Mathematical Society.