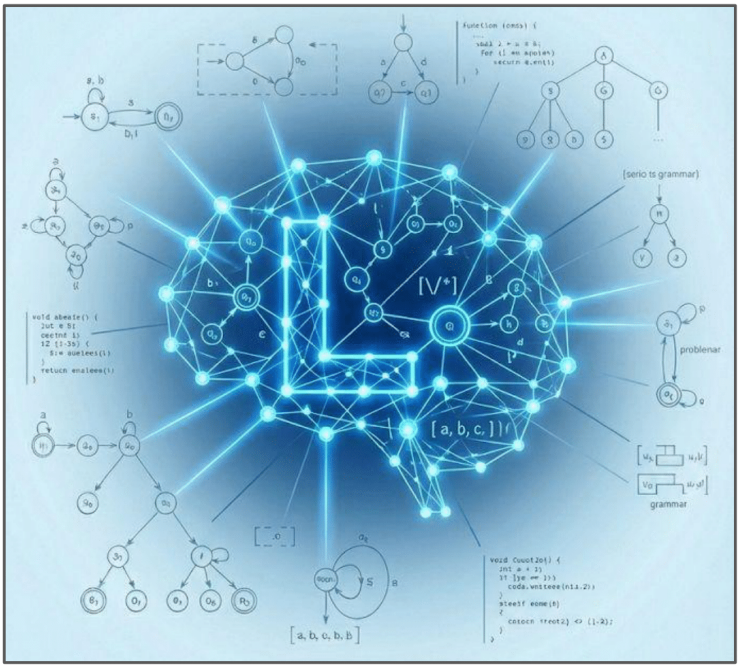
Teoría de Lenguajes Formales

Un enfoque basado en problemas



Carlos Andrés Flórez Villarraga
Ana María Tamayo Ocampo
Julián Esteban Gutiérrez Posada

2025

ISBN: 978-958-5108-76-9

Primera edición, [diciembre] de 2025

Inteligencia artificial: Los autores declaran ser responsables del contenido generado o asistido por herramientas de inteligencia artificial, incluyendo su originalidad, veracidad y licitud.

Teoría de Lenguajes Formales

Un enfoque basado en problemas

Carlos Andrés Flórez Villarraga
Ana María Tamayo Ocampo
Julián Esteban Gutiérrez Posada

Universidad del Quindío

Armenia - Quindío - Colombia
2025

Primera edición: Colombia, diciembre 2025

Teoría de Lenguajes Formales
Un enfoque basado en problemas

Sello editorial:

EIIZCOM S.A.S.(978-958-5108)

Carlos Andrés Flórez Villarraga
Ana María Tamayo Ocampo
Julián Esteban Gutiérrez Posada

Universidad del Quindío:
http://www.uniquindio.edu.co

Editado en: LaTeX 2_ε

Diciembre de 2025

Índice general

2 Capítulo 2
Lenguajes regulares

3 Capítulo 3
Lenguajes libres de contexto

Presentación

Los lenguajes formales constituyen una base fundamental de la ciencia de la computación, ya que permiten describir, analizar y comprender distintos procesos de comunicación y cómputo presentes en los sistemas informáticos. Este libro aborda los fundamentos de la Teoría de Lenguajes Formales desde un enfoque centrado en la resolución de problemas y articulado con los pilares del pensamiento computacional propuestos por Jeannette Wing [Wing, 2006], [Wing, 2010] y [Wing, 2024]: abstracción, descomposición, reconocimiento de patrones y codificación.

Cada capítulo parte de un problema contextualizado en situaciones propias de la computación. A partir de este, se desarrolla un proceso de análisis orientado a identificar la información relevante, las restricciones y los elementos esenciales del problema, favoreciendo así la abstracción. Luego, las tareas necesarias para resolverlo se dividen en componentes más manejables, lo que permite introducir gradualmente los conceptos y modelos formales requeridos.

El recorrido inicia con los conceptos básicos de la teoría, incluyendo lenguajes formales, alfabetos, cadenas y gramáticas como herramientas de modelación. La jerarquía de Chomsky se utiliza como eje para relacionar distintos niveles de complejidad lingüística con la capacidad computacional necesaria para procesarlos. Posteriormente, se estudian los lenguajes regulares mediante autómatas finitos y expresiones regulares, apoyándose en problemas de reconocimiento de patrones y validación de entradas, frecuentes en áreas como la programación y el análisis léxico.

Más adelante, se introducen los lenguajes libres de contexto a través de gramáticas independientes del contexto, árboles de derivación y autómatas de pila. Estos modelos se presentan a partir de problemas relacionados

con el análisis sintáctico, el diseño de mini lenguajes y la construcción de parsers. En este proceso, el reconocimiento de patrones permite establecer relaciones entre modelos previamente estudiados e identificar sus semejanzas y diferencias.

Finalmente, se incorporan modelos computacionales como las máquinas de Turing y el cálculo lambda, con el propósito de analizar distintas formas de computación, comparar el poder expresivo de los formalismos estudiados y discutir los límites teóricos de la computación.

Este enfoque integra teoría, pensamiento computacional y resolución de problemas, permitiendo que los lenguajes formales se estudien más allá de un conjunto de definiciones abstractas. De esta manera, se busca fortalecer la capacidad de modelar, analizar y diseñar soluciones computacionales en el contexto de la formación en ingeniería de sistemas.

Introducción

La Teoría de Lenguajes Formales es una de las áreas fundamentales de la ciencia de la computación y de la ingeniería de sistemas. Su estudio proporciona las bases necesarias para describir, analizar y diseñar los mecanismos mediante los cuales los sistemas computacionales procesan información. Compiladores, intérpretes, analizadores sintácticos, protocolos de comunicación, sistemas de verificación y diversas aplicaciones de inteligencia artificial utilizan, de una u otra forma, conceptos y modelos desarrollados en esta disciplina.

Sin embargo, para muchos estudiantes, el primer contacto con los lenguajes formales suele resultar abstracto y difícil de relacionar con problemas concretos de la computación. Con frecuencia, los conceptos se presentan como definiciones y formalismos aislados, sin que sea evidente su utilidad práctica. Este libro surge precisamente con el propósito de abordar la Teoría de Lenguajes Formales desde una perspectiva diferente, mostrando su aplicación en la resolución de problemas sin perder el rigor teórico que caracteriza al área.

El enfoque adoptado se basa en la resolución de problemas y se apoya en los pilares del pensamiento computacional propuestos por Jeannette Wing [Wing, 2006, Wing, 2010]: abstracción, descomposición, reconocimiento de patrones y codificación. En lugar de presentar los conceptos de manera aislada, cada capítulo inicia con una situación problema inspirada en contextos reales de la computación, cuya solución requiere la construcción progresiva de modelos formales.

Durante la fase de abstracción, el estudiante analiza el problema, identifica la información relevante y reconoce las restricciones que delimitan el conjunto de soluciones posibles. Este proceso conduce

naturalmente a conceptos como lenguaje formal, alfabeto y cadena.

Posteriormente, mediante la descomposición, se identifican los pasos necesarios para resolver el problema, enfatizando qué debe hacerse antes de abordar cómo hacerlo. Esto permite determinar qué conocimientos teóricos son necesarios e introducir de manera justificada los distintos formalismos que aparecen a lo largo del libro.

El reconocimiento de patrones permite identificar similitudes entre problemas, reutilizar soluciones previamente estudiadas y establecer relaciones entre diferentes modelos. Finalmente, la codificación no se limita a implementar una solución en un lenguaje de programación; también incluye la construcción de modelos formales, como autómatas, gramáticas, máquinas abstractas o pseudocódigo, que describen de manera precisa el comportamiento del sistema estudiado.

El libro inicia con los fundamentos de los lenguajes formales y la jerarquía de Chomsky, continúa con el estudio de los lenguajes regulares y los lenguajes libres de contexto, y finaliza con modelos de mayor poder computacional, como las gramáticas sensibles al contexto y las máquinas de Turing. Cada tema se introduce a partir de problemas relacionados con situaciones reales de la computación, como la validación de entradas, el reconocimiento de patrones, el análisis sintáctico y el estudio de los límites de la computación.

Este libro está dirigido a estudiantes de Ingeniería de Sistemas y Computación, especialmente en cursos de nivel intermedio. Su propósito es fortalecer tanto la comprensión de la Teoría de Lenguajes Formales como la capacidad de modelar problemas, desarrollar pensamiento computacional y utilizar la abstracción en el diseño de soluciones.

La intención es mostrar que los lenguajes formales no son únicamente una colección de definiciones teóricas, sino herramientas útiles para modelar, analizar y resolver problemas computacionales.

A lo largo del libro se incluirán notas como la siguiente para destacar conceptos importantes o aclarar aspectos que puedan generar dudas:

> Comentarios utilizados para resaltar conceptos importantes o proporcionar aclaraciones adicionales.

Capítulo 1

Fundamentos teóricos

Objetivo del capítulo

El objetivo de este capítulo es introducir los conceptos fundamentales de la Teoría de Lenguajes Formales (TLF), estableciendo el vocabulario matemático y computacional necesario para el estudio de gramáticas, autómatas y lenguajes computables en los capítulos posteriores.

Introducción a los lenguajes formales

La Teoría de Lenguajes Formales (TLF) es una de las áreas fundamentales de la informática teórica. Su objetivo es estudiar los lenguajes desde una perspectiva matemática, es decir, como conjuntos bien definidos de cadenas construidas a partir de símbolos pertenecientes a un alfabeto finito.

Aunque esta idea parece sencilla, constituye la base de múltiples áreas de la computación, como el diseño de compiladores, la verificación de software, el análisis de lenguajes de programación y algunos enfoques utilizados en inteligencia artificial.

A diferencia de lenguajes naturales como el español o el inglés, donde una misma expresión puede interpretarse de distintas maneras, los lenguajes formales se definen mediante reglas precisas y finitas. Esto permite analizarlos utilizando herramientas matemáticas rigurosas.

En este capítulo se introducen los conceptos básicos que servirán de fundamento para el resto del libro, así como la notación y el nivel de formalidad que se utilizarán en los capítulos siguientes.

Ejemplo

Pensemos en una frase como:

"El programa funciona correctamente"

Esta expresión tiene significado para una persona, pero no posee una estructura formal claramente definida. Por ello, una computadora tendría dificultades para analizarla de manera rigurosa.

En cambio, una expresión como:

$$a + b * c$$

sí puede interpretarse dentro de un lenguaje formal, siempre que existan reglas precisas que definan su estructura (por ejemplo, mediante una gramática). Gracias a estas reglas es posible analizar la cadena, identificar su estructura y verificar si está correctamente formada dentro del lenguaje definido.

Esto conduce a una pregunta fundamental:

¿Cómo especificar formalmente qué entradas son válidas en un sistema?

Motivación e importancia de los lenguajes formales

Los lenguajes formales surgen de la necesidad de describir estructuras sintácticas de manera precisa, no ambigua y verificable. En informática, esta necesidad aparece de forma natural en múltiples áreas, entre las cuales se destacan:

- **Lenguajes de programación**: su sintaxis tiene que estar definida formalmente para que un compilador pueda analizarla automáticamente.
- **Compiladores e intérpretes**: el análisis léxico y sintáctico se basa directamente en lenguajes formales.
- **Verificación y validación de sistemas**: muchos protocolos, especificaciones y modelos de comportamiento se expresan con lenguajes formales.
- **Teoría de la computación**: cuando se estudian los límites de lo que se puede computar, en el fondo se habla de qué lenguajes pueden reconocer distintos modelos abstractos.

Estas aplicaciones requieren definiciones exactas y operaciones matemáticamente bien definidas, lo que justifica el estudio formal de los lenguajes y de las operaciones sobre ellos, desarrolladas más adelante en este capítulo.

1.1. Segmentación institucional

Dentro de la gestión universitaria, los procesos de planificación académica y presupuestal suelen requerir que los estudiantes se agrupen según distintos criterios definidos por la institución. Estas clasificaciones no son mutuamente excluyentes; de hecho, un mismo estudiante puede pertenecer simultáneamente a varias categorías (por ejemplo, ser becario y también deportista).

Desde el punto de vista matemático, la población estudiantil puede modelarse como un conjunto universal U, donde cada clasificación representa un subconjunto de dicho conjunto. Esto permite aplicar operaciones básicas de teoría de conjuntos, como intersección, unión, diferencia y complemento, para analizar cómo se relacionan y superponen los distintos grupos.

Este tipo de análisis facilita la toma de decisiones institucionales y el estudio de la información académica de manera más estructurada.

1.1.1 Análisis / Abstracción

Después de analizar la situación problema anteriormente planteada, se puede identificar un problema relacionado con la forma en que se representan y analizan las diferentes clasificaciones de los estudiantes dentro de la universidad.

a. ¿Cuál es el problema a resolver?

El problema que se aborda consiste en cómo representar formalmente, o modelar, la manera en que una institución clasifica a sus estudiantes. La dificultad principal radica en que un mismo estudiante puede formar parte de varias categorías institucionales al mismo tiempo. Ejemplos de estos grupos son los programas de becas, las actividades deportivas o las iniciativas de investigación.

Para tratar este problema con herramientas matemáticas, se propone lo siguiente: considerar la población total de estudiantes como un conjunto universal U. A partir de ahí, cada criterio de clasificación se representa como un subconjunto de U. De esta forma, el solapamiento entre grupos queda expresado de manera natural mediante operaciones de teoría de conjuntos.

El objetivo que se persigue es examinar las relaciones entre dichos subconjuntos mediante operaciones básicas de teoría de conjuntos (como la unión, la intersección y la diferencia). Con estas operaciones se busca identificar, por un lado, a aquellos estudiantes que forman parte de múltiples clasificaciones a la vez y, por otro lado, a quienes pertenecen exclusivamente a una sola categoría específica.

Esta manera de proceder tiene dos ventajas. Primero, ofrece una representación formal de las estructuras de clasificación institucional. Segundo, facilita el análisis de información para apoyar la toma de decisiones tanto académicas como administrativas.

b. ¿Cuál es la información necesaria?

Para modelar la situación es necesario identificar los elementos fundamentales del problema:

- La totalidad de estudiantes de la institución, que se representa como el conjunto universal U.
- Los diferentes grupos o clasificaciones existentes (por ejemplo, becarios, deportistas, monitores, entre otros).
- Las características o criterios que determinan la pertenencia de un estudiante a cada grupo.

Con esta información es posible representar la población estudiantil y sus clasificaciones mediante conjuntos y subconjuntos.

1.1.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

El análisis del problema se apoya en los fundamentos de la teoría de conjuntos. La idea es sencilla: la población estudiantil se representa como

un conjunto universal, y cada clasificación institucional equivale a un subconjunto de ese conjunto.

Para resolver el problema, se plantea una descomposición lógica en los siguientes pasos:

1. Ingresar la cantidad total de estudiantes.
2. Ingresar los estudiantes de la institución.
3. Representar las distintas clasificaciones institucionales que sean de interés. Entre ellas se pueden mencionar, por ejemplo, a los estudiantes becarios, los estudiantes deportistas o cualquier otra categoría que resulte relevante para el estudio.
4. Identificar la población total de estudiantes de la institución.
5. Identificar los estudiantes que pertenecen simultáneamente a varias clasificaciones.
6. Determinar los estudiantes que pertenecen exclusivamente a una categoría específica.
7. Presentar los resultados obtenidos.

b. ¿Qué conocimiento adicional se requiere?

Para el diseño de la solución se requiere introducir los conceptos fundamentales de la **Teoría de Conjuntos**, junto con las **Operaciones entre Conjuntos**, los cuales constituyen herramientas matemáticas que permiten representar y analizar situaciones dentro de un contexto específico.

Teoría de Conjuntos Aplicada a la Segmentación

Dentro de un contexto específico, tal como se presenta en los fundamentos de las matemáticas discretas [Rosen, 2019].La teoría de conjuntos proporciona una herramienta formal para modelar poblaciones y clasificaciones dentro de una institución.

Conjunto Universal

Sea U el conjunto universal que representa la totalidad de estudiantes de la universidad:

$$U = \{x \mid x \text{ es estudiante matriculado}\}$$

Subconjuntos

Cada clasificación institucional se modela como un subconjunto de U:

$$B \subseteq U \quad \text{(Becarios)}$$

$$M \subseteq U \quad \text{(Monitores)}$$

$$D \subseteq U \quad \text{(Deportistas)}$$

$$R \subseteq U \quad \text{(Estudiantes en riesgo académico)}$$

Operaciones de Conjuntos

Las operaciones fundamentales permiten analizar la relación entre clasificaciones.

1. Intersección

La intersección entre dos conjuntos identifica estudiantes que pertenecen simultáneamente a ambas categorías.

$$B \cap D$$

Representa los estudiantes que son becarios y deportistas.

2. Unión

La unión reúne estudiantes que pertenecen al menos a una de las categorías.

$$B \cup R$$

Representa estudiantes que son becarios o están en riesgo académico.

3. Diferencia

La diferencia permite identificar poblaciones específicas excluyendo un grupo.

$$B - R$$

Representa becarios que no están en riesgo académico.

4. Complemento

El complemento identifica los estudiantes que no pertenecen a una clasificación.

Para el caso del riesgo académico, si se define R como el conjunto de estudiantes en riesgo y U como el conjunto universal, entonces el complemento $R^c = U - R$ representa justamente a los estudiantes que no están en riesgo.

5. Cardinalidad

La cardinalidad de un conjunto —denotada $|A|$— no es más que el número de elementos que contiene A. Es decir, una forma de medir su tamaño.

$$|B \cap D|$$

Indica cuántos estudiantes son simultáneamente becarios y deportistas.

Aplicación a la Toma de Decisiones

Con estas operaciones es posible hacer varias cosas de utilidad práctica. Por ejemplo:

- Identificar cuándo un mismo estudiante recibe beneficios de múltiples programas (superposición de beneficios).
- Detectar grupos de estudiantes que requieren atención prioritaria.
- Diseñar políticas enfocadas en necesidades específicas.
- Optimizar la forma en que se asigna el presupuesto.

De esta manera, la segmentación institucional puede verse como un sistema formal basado en conjuntos y en operaciones entre subconjuntos de un conjunto universal U.

c. ¿Qué se puede reutilizar de soluciones pasadas?

En este caso no hay soluciones previas que puedan reutilizarse, dado que aún no se ha abordado ningún problema similar con anterioridad. Por lo tanto, el modelo debe construirse desde cero.

d. Diseño de la solución para el problema

Desde la teoría de conjuntos, la población total de estudiantes puede representarse como un conjunto universal:

$$U = \{e_1, e_2, e_3, \ldots, e_n\}$$

donde cada elemento e_i representa a un estudiante de la institución.

Las distintas clasificaciones que maneja la institución se pueden modelar como subconjuntos de U. Por ejemplo:

$$B \subseteq U$$

En este caso, B representa al conjunto de estudiantes becarios.

$$D \subseteq U$$

Representa el conjunto de estudiantes deportistas.

A partir de estos subconjuntos es posible aplicar diversas operaciones de la teoría de conjuntos.

La intersección entre ambos conjuntos permite identificar los estudiantes que pertenecen simultáneamente a ambas clasificaciones:

$$B \cap D$$

La unión permite identificar todos los estudiantes que pertenecen al menos a una de las dos categorías:

$$B \cup D$$

Por su parte, la diferencia de conjuntos permite identificar aquellos estudiantes que pertenecen a una clasificación específica pero no a otra:

$$B - D$$

Este modelo permite representar formalmente las diferentes poblaciones estudiantiles y facilita el análisis de superposición entre grupos dentro de la institución.

Codificación

Esta sección muestra cómo traducir dichas representaciones a Python.

a. Algoritmo general de reconocimiento

Código 1.1: `main()`

```
4  def main():
5      cantidad=ingresar_cantidad("Ingrese la cantidad total de estudiantes: "
           )
6      U = ingresar_estudiantes(cantidad)
7      becarios = representar_clasificaciones("becarios",U)
8      deportistas = representar_clasificaciones("deportistas", U)
9      union = identificar_población_total(becarios, deportistas)
10
11     interseccion =
           identificar_estudiantes_que_pertenecen_simultaneamente_clasificaciones
           (becarios, deportistas)
12     diferencia = determinar_estudiantes_que_pertenecen_exclusivamente(
           becarios, deportistas)
13
14     mostrar_resultado("Unión", union)
15     mostrar_resultado("Intersección", interseccion)
16     mostrar_resultado("Diferencia (becarios - deportistas)", diferencia)
```

Propósito computacional: La función principal coordina la ejecución del programa. En ella se llaman las funciones encargadas de construir el conjunto universal, definir los subconjuntos de estudiantes y aplicar las operaciones de teoría de conjuntos. Finalmente, muestra los resultados obtenidos.

Interpretación conceptual: Dentro del problema descrito, esta función organiza las etapas del análisis institucional: modelar la población estudiantil y analizar las relaciones entre clasificaciones. Es, en cierto modo, una representación del flujo lógico del proceso.

Interpretación formal: En términos más formales, la función principal articula las distintas operaciones definidas sobre los conjuntos. Con ella se logra que las definiciones y operaciones de la teoría de conjuntos se apliquen paso a paso, de manera secuencial.

Código 1.2: `ingresar_cantidad()`

```
21 def ingresar_cantidad(mensaje):
22     cantidad = None
23 
24     while cantidad is None:
25         try:
26             valor = int(input(mensaje))
27             if valor >= 0:
28                 cantidad = valor
29             else:
30                 print("Ingrese un número mayor o igual a 0")
31         except ValueError:
32             print("Entrada inválida, ingrese un número entero")
33 
34     return cantidad
```

Propósito computacional: Esta función permite conocer la cantidad total de estudiantes del sistema. Para ello solicita al usuario la cantidad total de estudiantes.

Interpretación conceptual: ¿Qué representa esta función? En términos del problema, viene a ser la población total de estudiantes, es decir, el conjunto base sobre el cual se aplicarán las diferentes clasificaciones.

Interpretación formal: Desde un punto de vista más técnico, esta función no es más que la construcción completa del conjunto universal U.

Código 1.3: `ingresar_estudiantes()`

```
36 def ingresar_estudiantes(cantidad):
37     estudiantes = set()
38 
39 
40     for i in range(cantidad):
41         nombre = input("Ingrese el nombre del estudiante: ")
42         estudiantes.add(nombre)
43 
44     return estudiantes
```

Propósito computacional: Esta función permite construir el conjunto universal de estudiantes del sistema. Para ello solicita al usuario que registre cada nombre ingresado, almacenándolo dentro de una estructura de tipo conjunto.

Interpretación conceptual: En el contexto del problema, esta función representa la población total de estudiantes, dicho conjunto contiene todos los elementos que pueden pertenecer a uno o varios grupos dentro del análisis.

Interpretación formal: Visto desde la teoría de conjuntos, lo que hace esta función es construir el conjunto universal U. ¿Y qué es U? Pues el conjunto que contiene a todos los estudiantes que interesan en el sistema:

$$U = \{e_1, e_2, e_3, \ldots, e_n\}$$

En esa notación, cada e_i corresponde a un estudiante de la población que se está analizando.

Código 1.4: **representar_clasificaciones**

```
49 def representar_clasificaciones(nombre_grupo, universo):
50     subconjunto = set()
51     limite = len(universo)
52     cantidad = None
53 
54     while cantidad is None:
55         valor = ingresar_cantidad(f"Ingrese la cantidad de estudiantes en {
               nombre_grupo}: ")
56 
57         if valor <= limite:
58             cantidad = valor
59         else:
60             print(f"No puede ingresar más de {limite} estudiantes")
61 
62     for i in range(cantidad):
63         estudiante = input("Nombre del estudiante: ")
64 
65         if estudiante in universo:
66             subconjunto.add(estudiante)
67         else:
68             print("Este estudiante no pertenece al conjunto universal (U)")
69 
70     return subconjunto
```

Propósito computacional: Esta función permite crear subconjuntos de estudiantes que pertenecen a una clasificación específica, como becarios o deportistas. El usuario ingresa la cantidad de estudiantes del grupo y posteriormente registra cada uno de ellos.

Interpretación conceptual: En términos del problema, esta función representa las diferentes categorías institucionales que agrupan a ciertos estudiantes de acuerdo con características particulares. Cada grupo constituye una clasificación que forma parte del análisis institucional.

Interpretación formal: En teoría de conjuntos, cada clasificación se modela así: $A \subseteq U$, siendo A un grupo concreto de estudiantes, como los becarios o los deportistas.

Código 1.5: identificar_poblacion_total

```
76 def identificar_población_total(universo,conjunto1, conjunto2):
77     resultado = conjunto1.union(conjunto2)
78     es_valido = False
79
80     if len(resultado) <= len(universo):
81         es_valido = True
82     else:
83         print("\n Error: la unión excede el conjunto universal")
84         print("Revise los datos ingresados")
85
86     return resultado
```

Propósito computacional: Esta función recibe dos conjuntos y calcula su unión utilizando la operación `union()` de Python, generando un nuevo conjunto que contiene todos los elementos que pertenecen a cualquiera de los dos conjuntos; valida que la unión no exceda el conjunto universal

Interpretación conceptual: Dentro del contexto institucional, esta operación permite identificar el conjunto de estudiantes que pertenecen al menos a una de las clasificaciones consideradas, por ejemplo aquellos que son becarios o deportistas.

Interpretación formal: La unión de dos conjuntos A y B, escrita $A \cup B$, es el conjunto de elementos que están en A, en B, o en ambos. En notación de teoría de conjuntos:

$$A \cup B = \{x \mid x \in A \text{ o } x \in B\}$$

Código 1.6: identificar_estudiantes_que_pertenecen_simultaneamente

```
89 def identificar_estudiantes_que_pertenecen_simultaneamente_clasificaciones(
       conjunto1, conjunto2):
90     resultado = conjunto1.intersection(conjunto2)
91     return resultado
```

Propósito computacional: Esta función recibe dos conjuntos y calcula su intersección mediante la operación `intersection()` de Python, obteniendo un conjunto que contiene únicamente los elementos comunes a ambos conjuntos.

Interpretación conceptual: En el contexto del análisis institucional, esta operación permite identificar a los estudiantes que pertenecen simultáneamente a dos clasificaciones distintas, por ejemplo aquellos que son tanto becarios como deportistas.

Interpretación formal: En teoría de conjuntos, la intersección de dos conjuntos A y B se define así:

$$A \cap B = \{x \mid x \in A \text{ y } x \in B\}$$

Es decir, la intersección contiene únicamente a los elementos que están en A y en B al mismo tiempo.

Código 1.7: **determinar_estudiantes_que_pertenecen_exclusivamente**

```
93 def determinar_estudiantes_que_pertenecen_exclusivamente(conjunto1,
       conjunto2):
94     resultado = conjunto1.difference(conjunto2)
95     return resultado
```

Propósito computacional: Esta función calcula la diferencia entre dos conjuntos utilizando la operación `difference()` de Python, generando un conjunto con los elementos que pertenecen al primer conjunto pero no al segundo.

Interpretación conceptual: Dentro del problema que se está analizando, esta operación sirve para identificar a aquellos estudiantes que pertenecen a una clasificación específica, pero no a otra. Un ejemplo claro sería el caso de los estudiantes becarios que no participan en actividades deportivas.

Interpretación formal: En teoría de conjuntos, la diferencia entre dos conjuntos A y B se define de la siguiente manera:

$$A - B = \{x \mid x \in A \text{ y } x \notin B\}$$

Esto es, la diferencia está formada por los elementos que pertenecen a A, pero no pertenecen a B.

Código 1.8: mostrar_resultado

```
100 def mostrar_resultado(nombre_operacion, conjunto):
101     print("\n==============================")
102     print(f"Resultado de {nombre_operacion}:")
103 
104     if len(conjunto) == 0:
105         print("Conjunto vacío")
106     else:
107         for elemento in conjunto:
108             print(elemento)
```

Propósito computacional: Esta función se encarga de mostrar en pantalla los resultados de las operaciones sobre conjuntos, es decir, los elementos de cada conjunto resultante.

Interpretación conceptual: En el problema que se analiza, visualizar esos resultados permite entender con mayor claridad las relaciones entre los distintos grupos de estudiantes.

Interpretación formal: Desde una perspectiva formal, esta función no modifica los conjuntos ni realiza operaciones matemáticas sobre ellos; únicamente permite representar explícitamente el conjunto resultado de una operación previamente calculada.

Las siguientes secciones permiten evidenciar la aplicación práctica de los conceptos formales de alfabetos, palabras y lenguajes, ampliamente desarrollados en la literatura especializada [Lin and Rodger, 2009].

1.2. Sistema de autenticación bancaria

Una entidad bancaria se encuentra implementando un nuevo sistema de autenticación para transferencias electrónicas. Debido a requerimientos de interoperabilidad con sistemas heredados, el generador de códigos opera únicamente con símbolos pertenecientes a un conjunto previamente establecido.

Durante las pruebas piloto se identificaron fallos, ya que algunos módulos estaban aceptando y procesando caracteres no contemplados originalmente. Esta situación produjo inconsistencias entre el servidor central y la aplicación móvil, generando discrepancias en la validación de los códigos.

El escenario evidencia la importancia de contar con una definición clara y precisa de los elementos que intervienen en el sistema, especialmente en

entornos donde múltiples componentes interactúan y dependen de reglas estrictamente delimitadas.

1.2.1 Análisis / Abstracción

Después de analizar la situación problema anteriormente planteada, se puede identificar al menos un problema relacionado:

a. ¿Cuál es el problema a resolver?

Actualmente la entidad bancaria se encuentra implementando un nuevo sistema de autenticación para transferencias electrónicas. Ante esta situación, el equipo de ingeniería debe formalizar rigurosamente el conjunto de símbolos permitidos antes de diseñar el validador automático, considerando su impacto en la seguridad, la consistencia, posibles fallos y el control de entradas del sistema en lo referente a restricciones adecuadas, tales como errores de procesamiento, vulnerabilidades de seguridad o ambigüedades en la interpretación de los datos.

b. ¿Cuál es la información necesaria?

Para modelar el sistema de autenticación es necesario identificar los elementos fundamentales del problema:

- El conjunto de símbolos permitidos en el sistema, que se modela como un alfabeto Σ.
- Las entradas generadas o ingresadas por el usuario, que se representan como cadenas construidas a partir de Σ.
- La regla que determina si una entrada es válida se basa en si sus símbolos pertenecen o no al alfabeto que se haya definido previamente.

Con esta información es posible representar formalmente el sistema y analizar la validez de las entradas.

1.2.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

1. Identificar los símbolos permitidos.
2. Definir formalmente el conjunto alfabeto.
3. Solicitar la credencial
4. Verificar si cada símbolo de la credencial pertenece al conjunto definido, alfabeto.
5. Verificar si se acepta o rechaza la credencial.
6. Mostrar el resultado

b. ¿Qué conocimiento adicional se requiere?

Para diseñar la solución se requieren dos formalismos claves de la teoría de lenguajes formales: **Alfabetos**, proporciona el conjunto básico de símbolos a partir del cual se construyen todas las expresiones del lenguaje y los **símbolos**, unidades mínimas de información.

Alfabeto

Para entender qué es una cadena o un lenguaje formal, primero hay que hablar del alfabeto. Un alfabeto no es más que un conjunto finito y no vacío de símbolos. En la Teoría de Lenguajes Formales, es la materia prima: a partir de él se construyen todas las expresiones. Normalmente se denota con la letra Σ.

La cardinalidad de un alfabeto Σ se denota por $|\Sigma|$.
Ejemplos:

- Si $\Sigma = \{0, 1\}$, entonces $|\Sigma| = 2$. Este alfabeto es ampliamente utilizado en la representación de información binaria.
- Si $\Sigma = \{a, b, c\}$, entonces $|\Sigma| = 3$.
- En el diseño de lenguajes de programación, un alfabeto puede incluir letras, dígitos y símbolos especiales, por ejemplo:

$$\Sigma = \{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9, +, -, *, /\}$$

Alfabetos y su rol en los lenguajes formales

La elección de un alfabeto determina el universo de símbolos disponibles para la construcción de cadenas y, en consecuencia, el conjunto potencial de lenguajes que pueden definirse sobre dicho alfabeto. En capítulos posteriores se estudiará cómo, a partir de un alfabeto fijo, es posible especificar subconjuntos particulares de cadenas mediante gramáticas formales o modelos de cómputo. Es importante notar que un mismo lenguaje formal puede definirse únicamente con respecto a un alfabeto dado; cambiar el alfabeto implica modificar el conjunto de cadenas posibles y, por tanto, el contexto formal en el cual se estudia el lenguaje.

Símbolos

En el contexto del sistema de autenticación, los símbolos representan las unidades mínimas de información que pueden componer una credencial o código de acceso. Desde el punto de vista formal, cada símbolo es un elemento individual perteneciente a un conjunto finito denominado alfabeto Σ. En términos de ingeniería de sistemas y computación, estos símbolos pueden corresponder a letras, dígitos o caracteres especiales permitidos por las políticas de seguridad del sistema. La correcta identificación de los símbolos garantiza consistencia en el procesamiento de entradas y evita ambigüedades en la interpretación de los datos.

Restricción de dominio

La restricción de dominio consiste en delimitar explícitamente el conjunto de valores válidos que puede tomar una entrada dentro del sistema. En este caso, implica establecer que toda credencial debe estar compuesta exclusivamente por símbolos pertenecientes al alfabeto Σ. Esta restricción es fundamental en el diseño de software seguro, ya que reduce el espacio de posibles entradas, previene comportamientos inesperados y minimiza vulnerabilidades asociadas a datos no controlados. Formalmente, cualquier cadena válida debe cumplir la condición $w \in \Sigma^*$.

Validación estructural básica

La validación estructural básica corresponde al proceso mediante el cual el sistema verifica que la entrada cumpla con las reglas formales previamente definidas, antes de ejecutar cualquier procesamiento adicional. En este ejercicio, la validación implica comprobar que cada símbolo

de la cadena ingresada pertenece al alfabeto permitido y que la estructura general de la cadena se ajusta a los requisitos establecidos. Desde una perspectiva de arquitectura de software, esta validación debe implementarse en las primeras capas del sistema (por ejemplo, en el backend o en los servicios de control de entrada), garantizando coherencia entre la especificación formal y la implementación técnica.

c. ¿Qué se puede reutilizar de soluciones pasadas?

El análisis de este problema retoma algunos conceptos fundamentales de la teoría de conjuntos que ya se vieron antes. Uno de ellos es la representación de información mediante conjuntos, lo cual permite modelar el sistema de autenticación de forma estructurada.

En este nuevo contexto, el alfabeto Σ se puede ver como un conjunto finito de símbolos, y las cadenas como combinaciones de elementos de ese conjunto. La validación, por su parte, se relaciona directamente con la pertenencia: se verifica si los símbolos de una entrada están dentro del conjunto definido.

Así, nociones como conjunto, elemento y pertenencia se reutilizan, pero ahora aplicadas a un problema diferente. Eso permite reconocer patrones comunes en la forma de modelar y analizar distintos problemas computacionales.

d. Diseño de la solución para el problema

Primero se define el alfabeto de entrada, que en este caso es el conjunto de caracteres válidos, tiendo presente lo siguiente:

$$\text{Cadena compuesta únicamente por símbolos de } \Sigma$$

Adicionalmente, pueden definirse patrones más específicos como:

- Presencia de al menos un dígito.
- Presencia de al menos una letra mayúscula.
- Presencia de al menos un símbolo especial.
- Longitud mínima requerida.

Definición formal del alfabeto

Sea el sistema de autenticación que permite:

- Letras mayúsculas: $A, B, \ldots, Z$
- Letras minúsculas: $a, b, \ldots, z$
- Dígitos: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$
- Símbolos especiales: { @, #, $ }

Entonces el alfabeto permitido se define formalmente como:

$$\Sigma = \{A, \ldots, Z, a, \ldots, z, 0, \ldots, 9, @, \#, \$\}$$

Toda credencial válida debe pertenecer al conjunto: Σ

Importancia de la definición explícita del alfabeto

Definir formalmente Σ permite:

- Establecer una política clara de entrada.
- Reducir ambigüedades en el procesamiento.
- Garantizar coherencia entre cliente y servidor.
- Prevenir vulnerabilidades por caracteres no controlados.

Sin una definición formal, el sistema podría aceptar entradas inconsistentes o maliciosas.

1.2.3 Codificación

La representación del alfabeto se puede llevar a una implementación en código ejecutable. Esta sección muestra cómo traducir dichas representaciones a Python.

a. Algoritmo general de reconocimiento

El siguiente algoritmo de validación debe reflejar cada una de las etapas definidas, se plantea de acuerdo con la descomposición:

Código 1.9: Función `main()`

```
def main():
    # Paso 1
    simbolos = identificar_simbolos()

    # Paso 2
    sigma = definir_alfabeto(simbolos[0], simbolos[1], simbolos[2],
        simbolos[3])

    # Entrada
    credencial = pedir_credencial()

    # Paso 3
    es_valida = verificar_simbolos(credencial, sigma)

    # Paso 4
    resultado = validar_credencial(es_valida)

    mostrar_proceso(resultado)
```

Propósito computacional: Coordina la ejecución del programa llamando a cada función en el orden establecido por la descomposición del problema.

Interpretación conceptual: Lo que hace esta función es coordinar todo el proceso, desde el momento en que se definen los símbolos hasta que se entrega el resultado final.

Relación con el diseño algorítmico:

1. Identificar los símbolos permitidos.
2. Definir formalmente el alfabeto.
3. Verificar la pertenencia de la cadena.
4. Determinar si la entrada se acepta o se rechaza.

Este procedimiento garantiza lo siguiente:

- Que toda entrada cumpla con la restricción de dominio.
- Que la validación estructural se ejecute antes del cifrado.
- Que se reduzca el riesgo de vulnerabilidades provocadas por entradas no controladas.

Código 1.10: Funcion identificar_simbolos

```
23 def identificar_simbolos():
24     mayusculas = set("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
25     minusculas = set("abcdefghijklmnopqrstuvwxyz")
26     digitos = set("0123456789")
27     especiales = set("@#$")
28     resultado = (mayusculas, minusculas, digitos, especiales)
29
30     return resultado
```

Propósito computacional: Esta función construye cuatro conjuntos que agrupan los símbolos permitidos según su categoría: letras mayúsculas, letras minúsculas, dígitos y caracteres especiales. Luego devuelve esos conjuntos organizados dentro de una tupla, para que puedan usarse más adelante.

Interpretación conceptual: Lo que se hace aquí es una clasificación estructurada de los símbolos disponibles en el sistema, separándolos en subconjuntos homogéneos (es decir, cada subconjunto agrupa símbolos del mismo tipo).

Interpretación formal: Se definen subconjuntos del alfabeto:

$$M = \{A, \ldots, Z\}, \quad m = \{a, \ldots, z\}, \quad D = \{0, \ldots, 9\}, \quad E = \{@, \#, \$\}$$

Estos conjuntos serán utilizados posteriormente para construir el alfabeto total.

Código 1.11: Función definir_alfabeto

```
36 def definir_alfabeto(mayusculas, minusculas, digitos, especiales):
37     sigma = mayusculas.union(minusculas)
38     sigma = sigma.union(digitos)
39     sigma = sigma.union(especiales)
40
41     return sigma
```

Propósito computacional: Recibe los subconjuntos previamente definidos y construye un único conjunto que contiene todos los símbolos permitidos.

Interpretación conceptual: Lo que se logra con esta función es consolidar el conjunto total de símbolos autorizados dentro del sistema. En otras palabras, reúne en un solo lugar todos los símbolos que se consideran válidos.

Interpretación formal: Desde el punto de vista formal, la función implementa la operación de unión de conjuntos. La expresión sería la siguiente:

$$\Sigma = M \cup m \cup D \cup E$$

En esta ecuación, Σ representa el alfabeto formal del sistema, es decir, el conjunto total de símbolos permitidos.

Código 1.12: Función pedir_credencial

```
45 def pedir_credencial():
46     credencial = input("Ingrese la credencial: ")
47 
48     return credencial
```

Propósito computacional: Solicita al usuario el ingreso de una cadena desde el teclado y la retorna para su posterior procesamiento dentro del sistema.

Interpretación conceptual: Se captura la entrada del usuario, la cual representa la credencial que será evaluada para determinar si cumple con las condiciones del sistema de autenticación.

Interpretación formal: La función define una operación de entrada donde se obtiene una palabra $w \in \Sigma^*$, la cual será posteriormente evaluada respecto al alfabeto Σ y las condiciones de validez establecidas.

Código 1.13: Función verificar_simbolos

```
53 def verificar_simbolos(cadena, sigma, indice=0):
54     pertenece = True
55 
56     if len(cadena) == 0:
57         pertenece = False
58     else:
59         if indice < len(cadena):
60             if cadena[indice] not in sigma:
61                 pertenece = False
62             else:
63                 pertenece = verificar_simbolos(cadena, sigma, indice + 1)
64         else:
65             pertenece = True
66 
67     return pertenece
```

Propósito computacional: Evalúa de manera recursiva cada símbolo de la cadena ingresada, verificando si pertenece al alfabeto Σ. Además, valida que la cadena no sea vacía y retorna un valor booleano que indica si la cadena es válida o no.

Interpretación conceptual: La función recorre la cadena símbolo por símbolo. Si la cadena está vacía o algún símbolo no está en el conjunto permitido, se rechaza.

Interpretación formal: Se verifica que $w \in \Sigma^*$ cumpla $w \neq \epsilon$ y $\forall c \in w, c \in \Sigma$. El proceso es recursivo y evalúa cada posición hasta terminar la cadena.

Código 1.14: **Función validar_credencial**

```
72 def validar_credencial(es_valida):
73
74     if es_valida:
75         mensaje = "Acceso permitido"
76     else:
77         mensaje = "Acceso denegado"
78
79     return mensaje
```

Propósito computacional: Interpreta el resultado de la verificación y genera un mensaje final de aceptación o rechazo.

Interpretación conceptual: Transforma una condición lógica en una decisión del sistema.

Interpretación formal: Si $w \in L$, entonces la credencial es aceptada; en caso contrario, es rechazada.

Código 1.15: **Función mostrar_proceso()**

```
81 def mostrar_proceso(resultado):
82     print(resultado)
```

Propósito computacional: Esta función recibe como parámetro el resultado final del proceso de validación y lo muestra en pantalla mediante la instrucción `print`. Su objetivo es comunicar al usuario la decisión generada por el sistema.

Interpretación conceptual: Representa la etapa de salida del algoritmo. Mientras las funciones anteriores realizan operaciones de construcción y verificación, esta función cumple la tarea de presentar el resultado obtenido, haciendo visible la respuesta del sistema.

Interpretación formal: Desde el punto de vista abstracto, esta función no modifica el lenguaje ni interviene en la verificación de pertenencia. Su papel se limita a mostrar el resultado de la evaluación de la condición:

$$w \in L$$

donde L es el lenguaje definido previamente.

Es decir, actúa como un mecanismo de comunicación del resultado, sin modificar la lógica que determina si la entrada se acepta o se rechaza.

1.3. Análisis de comportamiento comercial

Una cadena de supermercados quiere analizar los patrones de compra y devolución que aparecen durante sus campañas promocionales. Cada transacción se codifica de manera simbólica y se guarda como una secuencia ordenada dentro de la base de datos.

Lo que la empresa busca es identificar rachas de devoluciones consecutivas, periodos de alta venta y secuencias atípicas que podrían evidenciar posibles situaciones de fraude.

1.3.1 Análisis / Abstracción

Después de analizar la situación problema anteriormente planteada, se puede identificar al menos un problema relacionado:

a. ¿Cuál es el problema a resolver?

La cadena de supermercados requiere modelar formalmente la jornada de transacciones. La idea es representar cada operación como una cadena sobre un alfabeto finito. Esto permite analizar la estructura secuencial y el orden de los símbolos, que son aspectos clave para identificar patrones significativos de compra y devolución.

Al preservar el orden de los registros, se pueden detectar tanto regularidades y comportamientos recurrentes como secuencias atípicas o sospechosas dentro de los datos almacenados. Así se obtiene una estrategia formal de análisis que ayuda a tomar decisiones y a controlar posibles anomalías.

Por el contrario, si no se hiciera este modelado, se podría perder información relevante. En concreto, si no se conserva el orden y la estructura de las transacciones, la interpretación de los patrones de comportamiento se vuelve más difícil o incluso incorrecta.

b. ¿Cuál es la información necesaria?

Para modelar la jornada de transacciones es necesario identificar los elementos fundamentales del problema:

- El conjunto de tipos de transacciones que pueden ocurrir (por ejemplo, compra, devolución, cancelación), los cuales se representan como un alfabeto Σ.
- Las jornadas de transacciones, que se modelan como cadenas formadas por secuencias ordenadas de símbolos del alfabeto.
- El orden en el que ocurren las transacciones, ya que este determina la estructura de la cadena y permite analizar patrones.

Con esta información es posible representar formalmente las secuencias de transacciones y analizar su comportamiento.

1.3.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

El análisis de las jornadas se apoya en la teoría de lenguajes formales. Cada jornada se representa como una cadena sobre un alfabeto finito que describe los tipos de operación del supermercado.

La descomposición lógica propuesta es la siguiente:

1. Definir el alfabeto de transacciones.
2. Solicitar Secuencia.

3. Verificar si cada símbolo de la secuencia pertenece al conjunto definido, alfabeto.

4. Analizar las secuencias.

5. Clasificar las secuencias.

6. Mostrar el resultado

Este enfoque garantiza que el análisis preserve la estructura secuencial de las transacciones y permita una detección sistemática de patrones significativos dentro del conjunto de registros almacenados.

b. ¿Qué conocimiento adicional se requiere?

Para diseñar la solución se requieren dos formalismos claves de la teoría de lenguajes formales: las **cadenas**, que describen las estructuras básicas, y la **Longitud de una cadena**, que hace referencia a la cantidad de símbolos.

Cadena

Con el alfabeto ya definido, el siguiente paso es construir las estructuras básicas que se generan a partir de él. A esas estructuras se les llama cadenas (o palabras), y son la base de los lenguajes formales.

Sea Σ un alfabeto. Una *cadena* (o palabra) sobre Σ es una secuencia finita de símbolos pertenecientes a Σ. Si w es una cadena formada por los símbolos $a_1, a_2, \ldots, a_n \in \Sigma$, se suele escribir: $w = a_1 a_2 \ldots a_n$.
Ejemplos:

- $0101 \in \{0, 1\}^*$
- $abc \in \{a, b, c\}^*$

Longitud de una cadena

La *longitud* de una cadena w, denotada por $|w|$, es el número de símbolos que la componen.

Si una cadena w está formada por n símbolos, es decir, $w = a_1a_2 \cdots a_n$, entonces su longitud se denota como $|w| = n$.
Algunos ejemplos:

- Si $w = 0101$, entonces $|w| = 4$.
- Si $w = abc$, entonces $|w| = 3$.
- Si $w = x_1x_2x_3$, entonces $|w| = 3$.

c. ¿Qué se puede reutilizar de soluciones pasadas?

Este problema retoma y amplía los conceptos trabajados en los ejercicios anteriores, tanto desde la teoría de conjuntos como desde el sistema de autenticación.

En primer lugar, se reutiliza la idea de conjunto para representar el alfabeto Σ, entendido como un conjunto finito de símbolos que describen los tipos de transacciones posibles.

Asimismo, se retoma el concepto de cadena, ya utilizado en el sistema de autenticación, donde una secuencia de símbolos representa una entrada. En este caso, dicha secuencia modela una jornada completa de transacciones.

La diferencia clave está en que ahora el orden de los símbolos se tiene en cuenta de forma explícita. Con esto ya no solo se puede verificar la validez de las entradas, sino también estudiar patrones y comportamientos dentro de las secuencias.

De esta manera, se evidencia la reutilización de conceptos fundamentales como conjunto, símbolo y cadena, aplicados en un contexto más complejo orientado al análisis de secuencias.

d. Diseño de la solución para el problema

La estrategia de resolución mantiene la misma estructura general que se usó en el ejercicio anterior.

En primer lugar, se define el alfabeto Σ que representa los tipos de transacción. Posteriormente, la secuencia almacenada se modela como una palabra $w \in \Sigma$ y se recorre símbolo por símbolo mediante un procedimiento iterativo.

La diferencia con el ejercicio previo no está en cómo está construido el algoritmo, sino en qué se revisa mientras se recorre la cadena. Antes bastaba con comprobar que cada símbolo estuviera en el alfabeto. Ahora, en cambio, se analizan las relaciones entre símbolos seguidos para detectar rachas o ciertos patrones de comportamiento.

Sea el alfabeto:

$$\Sigma = \{V, D\}$$

donde:

- V representa una venta.
- D representa una devolución.

Una jornada comercial se modela como una cadena:

$$w = a_1 a_2 \dots a_n, \quad a_i \in \Sigma$$

Para detectar devoluciones consecutivas se analiza la existencia de la subcadena:

$$DD$$

Formalmente, el lenguaje que contiene todas las jornadas con al menos dos devoluciones consecutivas es:

$$L = \{w \in \Sigma \mid DD \text{ es subcadena de } w\}$$

Así, el esquema general se mantiene:

1. Definir formalmente el alfabeto.
2. Validar la secuencia como palabra sobre Σ.
3. Recorrer la cadena de manera ordenada.
4. Evaluar una condición específica durante el recorrido.
5. Emitir una clasificación según el resultado obtenido.

Por lo tanto, se reutiliza el mismo patrón estructural de modelado y análisis sobre cadenas, y lo único que cambia es el criterio de evaluación.

1.3.3 Codificación

La representaciones del alfabeto y cadenas se puede llevar a una implementación en código ejecutable. Esta sección muestra cómo traducir dichas representaciones a Python.

a. Algoritmo general de reconocimiento

El siguiente algoritmo de validación debe reflejar cada una de las etapas definidas, se plantea de acuerdo con la descomposición:

Código 1.16: `Funcion main()`

```
1 def main():
2     sigma = definir_alfabeto()
3     secuencia = pedir_secuencia()
4     es_valida = verificar_simbolos(secuencia, sigma)
5     max_devoluciones = analizar_secuencia(secuencia)
6     resultado = clasificar(es_valida, max_devoluciones)
7     mostrar_resultado(resultado)
```

Propósito computacional: Orquesta la ejecución del programa, coordinando la definición del alfabeto, la captura de la secuencia, su validación, análisis y clasificación final.

Interpretación conceptual: Esta función organiza el flujo de ejecución del sistema y asegura que cada paso se haga en el orden correcto.

Interpretación formal: Define una secuencia de operaciones sobre $w \in \Sigma^*$: validación, análisis y clasificación. Produce una salida que describe el comportamiento de w en el sistema.

Código 1.17: `Funcion definir_alfabeto()`

```
12 def definir_alfabeto():
13     sigma = {"C", "D"}
14 
15     return sigma
```

Propósito computacional: Construye y devuelve el conjunto de símbolos permitidos dentro del sistema de transacciones.

Interpretación conceptual: Esta función establece los tipos de eventos que pueden registrarse en la base de datos durante una campaña promocional.

Interpretación formal: En términos formales, define el alfabeto del sistema de la siguiente manera: $\Sigma = \{C, D\}$, donde C representa una compra y D una devolución.

Código 1.18: Funcion pedir_secuencia

```
20 def pedir_secuencia():
21     secuencia = input("Ingrese la secuencia de transacciones: ")
22 
23     return secuencia
```

Propósito computacional: Solicita al usuario el ingreso de una secuencia de símbolos desde el teclado y la retorna para su procesamiento.

Interpretación conceptual: Se captura la entrada que representa una serie de transacciones que serán analizadas por el sistema.

Interpretación formal: Se obtiene una palabra $w \in \Sigma^*$, la cual será posteriormente evaluada respecto al alfabeto Σ y las condiciones de validez establecidas.

Código 1.19: Funcion verificar_simbolos

```
28 def verificar_simbolos(secuencia, sigma, indice=0):
29     es_valida = True
30 
31     if len(secuencia) == 0:
32         es_valida = False
33     else:
34         if indice < len(secuencia):
35             if secuencia[indice] not in sigma:
36                 es_valida = False
37             else:
38                 es_valida = verificar_simbolos(secuencia, sigma, indice +
                       1)
39         else:
40             es_valida = True
41 
42     return es_valida
```

Propósito computacional: Evalúa de forma recursiva cada símbolo de la secuencia ingresada, verificando si pertenece al alfabeto Σ. Además, valida que la secuencia no sea vacía.

Interpretación conceptual: La función recorre la cadena de forma secuencial y analiza cada símbolo para determinar si corresponde a una transacción válida. Si algún símbolo no pertenece al conjunto permitido, o si la secuencia está vacía, entonces se rechaza.

Interpretación formal: Desde el punto de vista formal, la función verifica que una palabra $w \in \Sigma^*$ cumpla dos condiciones: que no sea la cadena vacía ($w \neq \epsilon$) y que todos sus caracteres pertenezcan al alfabeto ($\forall c \in w, c \in \Sigma$). El análisis se realiza mediante un proceso recursivo que evalúa cada posición de la cadena.

Código 1.20: **Funcion analizar_secuencia**

```
47 def analizar_secuencia(secuencia, indice=0, contador=0, max_devoluciones=0)
       :
48     resultado = max_devoluciones
49
50     if indice < len(secuencia):
51
52         if secuencia[indice] == "D":
53             contador = contador + 1
54         else:
55             contador = 0
56
57         if contador > max_devoluciones:
58             max_devoluciones = contador
59
60         resultado = analizar_secuencia(secuencia, indice + 1, contador,
               max_devoluciones)
61
62     return resultado
```

Propósito computacional: Recorre la secuencia símbolo por símbolo y calcula la longitud máxima de rachas consecutivas de devoluciones.

Interpretación conceptual: Esta función permite identificar comportamientos repetitivos dentro de la jornada comercial. En particular, detecta devoluciones consecutivas que podrían indicar situaciones atípicas.

Interpretación formal: Dada una palabra

$$w = c_1 c_2 \dots c_n$$

la función analiza la relación entre símbolos consecutivos c_i y c_{i+1} para determinar la mayor subsecuencia contigua formada exclusivamente por el símbolo D.

Código 1.21: Funcion Clasificar

```
67 def clasificar(es_valida, max_devoluciones):
68     resultado = "Secuencia invalida"
69
70     if es_valida:
71         resultado = "Comportamiento regular"
72
73         if max_devoluciones >= 3:
74             resultado = "Secuencia atipica detectada"
75
76     return resultado
```

Propósito computacional: Esta función integra el proceso de validación y análisis para determinar cuál es la clasificación final de la secuencia.

Interpretación conceptual: Esta función traduce el análisis estructural de la cadena en una decisión interpretativa por parte del sistema, que puede ser ”regular” o ”atípico”.

Interpretación formal: En términos formales, el proceso funciona así: si la secuencia no pertenece a Σ ni a las combinaciones posibles a partir de él, entonces se clasifica como inválida. En caso de que sí pertenezca, se calcula la longitud máxima de las rachas de devoluciones y se compara con un umbral que se ha definido previamente.

Código 1.22: Funcion mostrar

```
81 def mostrar_resultado(mensaje):
82     print(mensaje)
```

Propósito computacional: Muestra en pantalla el resultado del análisis de la secuencia.

Interpretación conceptual: Comunica al usuario la clasificación obtenida para que interprete el comportamiento de la secuencia ingresada.

Interpretación formal: Realiza una operación de salida con el resultado de evaluar w, clasificada según las reglas de Σ^*.

1.4. Seguridad corporativa y longitud de contraseñas

En los entornos actuales de seguridad corporativa, las organizaciones dependen de mecanismos de autenticación sólidos para proteger activos críticos como bases de datos, información financiera, propiedad intelectual y credenciales de acceso remoto. Empresas tecnológicas como Microsoft, Google e IBM han documentado que una de las principales causas de incidentes de seguridad continúa siendo el uso de contraseñas débiles o configuradas de manera inadecuada.

1.4.1 Análisis / Abstracción

Después de analizar la situación problema anteriormente planteada, se puede identificar al menos un problema relacionado:

a. ¿Cuál es el problema a resolver?

Consiste en analizar formalmente la longitud de una cadena dentro de un sistema de autenticación y determinar cómo esta propiedad estructural incide en la seguridad del mecanismo de acceso.

En particular, se busca explicar qué representa la longitud de una cadena desde la teoría de lenguajes formales, es decir, como el número de símbolos que componen una cadena definida sobre un alfabeto finito, y examinar cómo dicha longitud influye en el número total de combinaciones posibles que pueden generarse dentro del sistema.

Asimismo, se pretende argumentar por qué la longitud constituye un criterio estructural independiente del contenido semántico de la contraseña, ya que depende exclusivamente de la cantidad de símbolos y no del significado que estos puedan tener. Finalmente, el problema implica proponer una forma coherente de integrar esta restricción de longitud dentro del mecanismo general de validación y control de acceso, garantizando consistencia formal y fortalecimiento del esquema de seguridad.

b. ¿Cuál es la información necesaria?

Para modelar el sistema de contraseñas es necesario identificar los elementos fundamentales:

- El conjunto de símbolos permitidos, que se representa como un alfabeto Σ.
- Las contraseñas del sistema, que se modelan como cadenas construidas a partir de Σ.
- La longitud de las cadenas, entendida como el número de símbolos que componen cada contraseña.

Con esta información es posible analizar las propiedades estructurales de las contraseñas y su impacto en la seguridad del sistema.

1.4.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

El análisis de las contraseñas se hace desde la teoría básica de lenguajes formales. Básicamente, una contraseña se trata como una cadena sobre un alfabeto finito, y lo que importa es que su longitud esté entre un mínimo y un máximo predefinidos.

Para resolver el problema, se propone la siguiente descomposición lógica:

1. Identificar simbolos, se establecen los conjuntos de símbolos que serán utilizados en el sistema.
2. Definir alfabeto, se construye el alfabeto Σ a partir de la unión de los símbolos definidos.
3. Pedir contrasena, se obtiene la cadena de entrada del usuario.
4. Pedir longitud mínima, se define el valor mínimo requerido para la longitud de la cadena.
5. Validar longitud permite verificar si la cadena cumple con la restricción de longitud.
6. Calcular combinaciones, se determina el número de combinaciones posibles en función del alfabeto y la longitud de la cadena.
7. Mostrar resultados, se presentan los resultados obtenidos.

b. ¿Qué conocimiento adicional se requiere?

Para diseñar la solución se requieren dos formalismos claves de la teoría de lenguajes formales: las **Conjunto de cadenas**, que describen las estructuras básicas, las **Operaciones con cadena**, que hace referencia a como se pueden construir nuevas cadenas con cadenas existentes, **Prefijos, sufijos y subcadenas** elementos en cadenas y **Lenguajes definidos por Propiedades**, permite verificar longitud, símbolos iniciales, finales, entre otros.

Conjunto de cadenas sobre un alfabeto

Sea Σ un alfabeto. A partir de él se define el conjunto de todas las cadenas finitas que se pueden formar con sus símbolos, y eso incluye también la cadena vacía.

Si $\Sigma = \{a, b\}$, entonces:El conjunto de cadenas sobre un alfabeto es =

$$\{\varepsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$$

El conjunto es infinito, incluso cuando el alfabeto Σ es finito.

Operaciones sobre cadenas

Una vez que se tienen cadenas definidas sobre un alfabeto, se pueden combinar y manipular con distintas operaciones formales. Estas operaciones permiten generar nuevas cadenas a partir de las ya existentes, y también son la base para definir operaciones más generales sobre lenguajes formales.

Concatenación

Sean $u, v \in \Sigma$. La *concatenación* de u y v, denotada por uv, es la cadena obtenida al escribir los símbolos de u seguidos, en el mismo orden, por los símbolos de v.

Si $u = a_1 a_2 \cdots a_m$ y $v = b_1 b_2 \cdots b_n$, entonces:

$$uv = a_1 a_2 \cdots a_m b_1 b_2 \cdots b_n$$

Propiedades de la concatenación

La operación de concatenación sobre cadenas cumple las siguientes propiedades:

- **Asociatividad**: Si se tiene las siguientes cadenas $u, v, w \in \Sigma$,

$$(uv)w = u(vw)$$

- **Elemento neutro**: Para toda cadena $w \in \Sigma$,

$$w\varepsilon = \varepsilon w = w$$

- **No conmutatividad**: En general, para $u, v \in \Sigma$,

$$uv \neq vu$$

Ejemplo

Sean $u = ab$ y $v = ba$. Entonces:

$$uv = abba, \qquad vu = baab$$

por lo que $uv \neq vu$.

Prefijos, sufijos y subcadenas

Definiciones

Prefijo: Sea $w \in \Sigma^*$. Una cadena $u \in \Sigma^*$ es un *prefijo* de w si existe $v \in \Sigma^*$ tal que $w = uv$.

Sufijo: Sea $w \in \Sigma^*$. Una cadena $u \in \Sigma^*$ es un sufijo de w si existe $v \in \Sigma^*$ tal que $w = vu$. Es decir, u es la parte final de w.

Subcadena:Sea $w \in \Sigma^*$. Una cadena $u \in \Sigma^*$ es una subcadena de w si existen $x, y \in \Sigma^*$ tales que $w = xuy$. En otras palabras, u es un segmento continuo dentro de w.

¿Cómo diferenciarlos visualmente?

Ejemplo Para $w = abab$:

- **Prefijos**: empiezan desde el principio — $\varepsilon, a, ab, aba, abab$
- **Sufijos**: terminan en el final — $\varepsilon, b, ab, bab, abab$
- **Subcadenas**: cualquier segmento continuo — a, b, ab, ba, aba

Los prefijos y los sufijos son casos particulares de subcadenas: un prefijo es una subcadena donde $x = \varepsilon$, y un sufijo es una subcadena donde $y = \varepsilon$.

Ejemplos con $w = xyza$, algunos prefijos son:

- ε.
- x.
- xy.
- xyz.
- $xyza$.

Algunos sufijos son:

- ε.
- a.
- za.
- yza.
- $xyza$.

Algunas subcadenas son:

- x.
- yz.
- z.
- xy.
- za.

Lenguajes definidos por propiedades

Muchos lenguajes formales se definen especificando una propiedad que deben cumplir sus cadenas.

Un lenguaje $L \subseteq \Sigma^*$ se define por una propiedad P cuando:

$$L = \{w \in \Sigma^* \mid P(w) \text{ es verdadera}\}$$

Donde $P(w)$ es una condición verificable sobre la cadena w.

Ejemplo

Algunos lenguajes definidos por propiedades sobre $\Sigma = \{0, 1\}$:

- **Longitud par**: $L_1 = \{w \in \{0, 1\}^* \mid |w| \text{ es par}\}$

- **Número igual de 0s y 1s**: $L_2 = \{w \in \{0,1\}^* \mid \#0(w) = \#1(w)\}$
- **Comienza con 0**: $L_3 = \{w \in \{0,1\}^* \mid w \text{ comienza con } 0\}$
- **Termina con 1**: $L_4 = \{w \in \{0,1\}^* \mid w \text{ termina con } 1\}$
- **Prefijo igual a sufijo**: $L_5 = \{w \in \{0,1\}^* \mid \text{el primer símbolo de } w = \text{el último}\}$

Nota Estas propiedades pueden verificarse localmente (longitud, primer/último símbolo) o globalmente (balance de símbolos). Muchas son reconocibles por autómatas finitos, base de la teoría de lenguajes regulares.

Lenguajes y su interpretación

Aunque los lenguajes formales carecen de significado semántico intrínseco, pueden interpretarse en distintos contextos computacionales. Por ejemplo, un lenguaje puede representar el conjunto de programas sintácticamente correctos de un lenguaje de programación, o el conjunto de expresiones aritméticas bien formadas.

El estudio de los lenguajes formales se centra en describir de manera precisa estos conjuntos, analizar sus propiedades y determinar los mecanismos necesarios para generarlos o reconocerlos, aspectos que serán abordados en los capítulos posteriores mediante gramáticas formales y autómatas.

c. ¿Qué se puede reutilizar de soluciones pasadas?

Este problema retoma y amplía los conceptos desarrollados en los ejercicios anteriores.

En primer lugar, se reutiliza la idea de conjunto, trabajada en la segmentación institucional, para representar el alfabeto Σ como un conjunto finito de símbolos permitidos.

Asimismo, se retoma el concepto de cadena, introducido en el sistema de autenticación, donde una secuencia de símbolos representa una entrada válida dentro de un sistema.

Lo nuevo en este caso es que se añade la longitud de la cadena como un factor a considerar. Gracias a eso se pueden analizar otras propiedades estructurales, por ejemplo cuántas combinaciones son posibles y cómo afecta eso a la seguridad.

De esta manera, se evidencia la reutilización de los conceptos de conjunto, símbolo y cadena, extendidos ahora hacia el análisis de propiedades cuantitativas en las cadenas.

d. Diseño de la solución para el problema

La estrategia de resolución conserva la estructura que se ha utilizado en los ejercicios anteriores:

Sea Σ un alfabeto finito tal que:

$$\Sigma = \{A, \ldots, Z, a, \ldots, z, 0, \ldots, 9, @, \#, \$\}$$

Las contraseñas del sistema se modelan como palabras:

$$w \in \Sigma^*$$

La longitud de una cadena se define como la función:

$$|w| : \Sigma^* \to \mathbb{N}$$

donde $|w|$ representa el número de símbolos que componen la cadena w.

Si $|\Sigma| = k$ y se fija una longitud n, el número total de combinaciones posibles es:

$$k^n$$

Como el crecimiento es exponencial con respecto a n, aumentar la longitud hace que el espacio de búsqueda crezca de manera significativa.

A partir de esto se establece una restricción de longitud mínima:

$$L = \{w \in \Sigma^* \mid |w| \geq n_{\text{mín}}\}$$

donde $n_{\text{mín}}$ es la longitud mínima que exige el sistema.

1.4.3 Codificación

La representaciones del alfabeto y cadenas se puede llevar a una implementación en código ejecutable. Esta sección muestra cómo traducir dichas representaciones a Python.

a. Algoritmo general de reconocimiento

El siguiente algoritmo de validación debe reflejar cada una de las etapas definidas, se plantea de acuerdo con la descomposición:

Código 1.23: `main()`

```
def main():
    simbolos = identificar_simbolos()
    sigma = definir_alfabeto(simbolos)
    contrasena = pedir_contrasena()
    n_min = pedir_longitud_minima()
    mensaje = validar_longitud(contrasena, n_min)
    total = calcular_combinaciones(contrasena, sigma, n_min)
    mostrar_resultados(mensaje, total)
```

Propósito computacional: Orquesta la ejecución del sistema mediante el llamado secuencial de las funciones definidas.

Interpretación conceptual: Esta función coordina el flujo general del proceso, que incluye: la definición del alfabeto, la captura de datos, la validación y la presentación de resultados.

Interpretación formal: En términos formales, define una secuencia de evaluaciones sobre una palabra $w \in \Sigma^*$. Para ello aplica funciones que verifican restricciones y calculan propiedades asociadas, todo sin realizar procesamiento interno adicional.

Código 1.24: `identificar_simbolos()`

```
def identificar_simbolos():
    mayusculas = set("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
    minusculas = set("abcdefghijklmnopqrstuvwxyz")
    digitos = set("0123456789")
    especiales = set("@#$")
    resultado = (mayusculas, minusculas, digitos, especiales)

    return resultado
```

Propósito computacional: Construir los subconjuntos de símbolos que serán utilizados para definir el alfabeto del sistema.

Interpretación conceptual: Esta función separa los tipos de caracteres permitidos en categorías estructurales: letras mayúsculas, letras minúsculas, dígitos y símbolos especiales. Esta clasificación ayuda a organizar el universo de símbolos antes de unificarlos formalmente.

Interpretación formal: Define subconjuntos finitos tales que:

$$M = \{A, \ldots, Z\}$$
$$m = \{a, \ldots, z\}$$
$$D = \{0, \ldots, 9\}$$
$$E = \{@, \#, \$\}$$

Estos conjuntos serán utilizados posteriormente para construir el alfabeto total.

Código 1.25: Función definir_alfabeto()

```
20 def definir_alfabeto(simbolos):
21     sigma = simbolos[0].union(simbolos[1])
22     sigma = sigma.union(simbolos[2])
23     sigma = sigma.union(simbolos[3])
24
25     return sigma
```

Propósito computacional: Unificar los subconjuntos de símbolos en un único conjunto denominado Σ.

Interpretación conceptual: Integra todas las categorías de símbolos permitidos en el sistema, estableciendo el universo estructural para las cadenas.

Interpretación formal: Construye el alfabeto como una unión de conjuntos: $\Sigma = M \cup m \cup D \cup E$, donde Σ es finito.

Código 1.26: Función pedir_contraseña()

```
27 def pedir_contrasena():
28     contrasena = input("Ingrese la contraseña: ")
29
30     return contrasena
```

Propósito computacional: Solicita al usuario una contraseña y la almacena para su posterior validación.

Interpretación conceptual: Permite capturar la contraseña que será evaluada según las reglas definidas por el sistema.

Interpretación formal: Se obtiene una cadena $w \in \Sigma^*$, la cual será analizada para verificar si cumple las condiciones establecidas.

Código 1.27: `Función pedir_longitud_minima()`

```
32 def pedir_longitud_minima():
33     n_min = int(input("Ingrese la longitud mínima: "))
34 
35     return n_min
```

Propósito computacional: Solicita al usuario el ingreso de un valor entero que representa la longitud mínima requerida para la validación de la cadena.

Interpretación conceptual: Se establece dinámicamente una restricción sobre el tamaño mínimo que debe cumplir la contraseña.

Interpretación formal: Se obtiene un valor $n_{min} \in \mathbb{N}$, el cual se utilizará para verificar la condición $|w| \geq n_{min}$.

Código 1.28: `Función validad_longitud()`

```
37 def validar_longitud(w, n_min):
38     resultado = ""
39 
40     if len(w) == 0:
41         resultado = "Cadena inválida: no puede estar vacía."
42     elif len(w) >= n_min:
43         resultado = "Longitud válida."
44     else:
45         resultado = "Longitud insuficiente."
46 
47     return resultado
```

Propósito computacional: Esta función evalúa si la cadena ingresada cumple con la longitud mínima requerida y si no está vacía. Luego retorna un mensaje que describe el resultado de esa verificación.

Interpretación conceptual: Lo que se hace aquí es verificar si la contraseña tiene un tamaño adecuado para ser considerada válida dentro del sistema.

Interpretación formal: Dada una palabra $w \in \Sigma^*$, la función evalúa:

Si $w = \epsilon$ (cadena vacía), se rechaza.

Si $|w| \geq n_{min}$, se acepta.

En caso contrario, se considera insuficiente.

Código 1.29: Función calcular_combinaciones()

```
50 def calcular_combinaciones(w, sigma, n_min):
51     total = 0
52
53     if len(w) >= n_min:
54         total = len(sigma) ** len(w)
55
56     return total
```

Propósito computacional: Determinar el número total de cadenas posibles de longitud fija n.

Interpretación conceptual: Esta función permite analizar el crecimiento del espacio de búsqueda del sistema de autenticación.

Interpretación formal: Si $|\Sigma| = k$ y $|w| = n$, entonces el número total de combinaciones posibles es k^n. La función, por lo tanto, implementa una relación de tipo exponencial.

Código 1.30: Función mostrar_proceso()

```
58 def mostrar_resultados(mensaje, total):
59     print(mensaje)
60     if total > 0:
61         print("Número total de combinaciones posibles:", total)
```

Propósito computacional: Esta función presenta en pantalla el resultado de la validación y, cuando corresponde, también muestra el número de combinaciones posibles.

Interpretación conceptual: Se le comunica al usuario el estado de su contraseña y el nivel de complejidad asociado a ella.

Interpretación formal: Dado un mensaje m y un valor $t \in \mathbb{N}$, la función muestra m en pantalla. Adicionalmente, si $t > 0$, también presenta el valor correspondiente al tamaño del conjunto de cadenas posibles.

1.5. Validación de formularios digitales

En el marco de la transformación digital institucional, la universidad ha implementado una plataforma web para gestionar procesos académicos como matrículas, solicitudes, certificaciones y actualización de datos personales.

El sistema permite que los estudiantes diligencien formularios electrónicos que incluyen campos obligatorios. Sin embargo, se ha detectado que algunos usuarios envían formularios con campos vacíos.

1.5.1 Análisis / Abstracción

Después de analizar la situación problema anteriormente planteada, se puede identificar al menos un problema relacionado:

a. ¿Cuál es el problema a resolver?

El problema consiste en garantizar la integridad estructural de los campos obligatorios dentro del sistema. Desde una perspectiva formal, cada campo se modela como una cadena de caracteres definida sobre un alfabeto determinado. Sin embargo, en algunos casos esa cadena puede estar vacía, es decir, no contener ningún símbolo.

La presencia de cadenas vacías en campos obligatorios afecta la consistencia de la información almacenada y puede generar problemas en los procesos administrativos que dependen de esos datos. Por lo tanto, el problema radica en establecer un criterio formal de validación que asegure que cada campo obligatorio contenga al menos un símbolo válido antes de ser aceptado y procesado por el sistema.

b. ¿Cuál es la información necesaria?

Para modelar el problema es necesario identificar los siguientes elementos:

- El alfabeto Σ que define los símbolos permitidos en los campos.
- Los campos del formulario, representados como cadenas w_i.
- La cadena vacía ε como caso particular de longitud cero.
- La distinción entre cadenas vacías y no vacías (Σ^+).

1.5.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

El análisis del diligenciamiento de formularios se aborda desde la teoría básica de lenguajes formales, considerando que cada campo ingresado por el usuario puede representarse como una cadena construida sobre un alfabeto Σ que describe los símbolos permitidos dentro del sistema (por ejemplo, letras, dígitos y algunos caracteres especiales). De esta manera, cada formulario puede entenderse como un conjunto de cadenas que representan la información suministrada por el estudiante, lo cual permite analizar formalmente si los campos obligatorios contienen información válida o si, por el contrario, corresponden a la cadena vacía.

Para resolver el problema, se propone la siguiente descomposición lógica:

1. Definir el alfabeto. Esto permite construir Σ, es decir, el conjunto que contiene los símbolos permitidos en los campos del formulario.

2. Ingresar formulario, para capturan los datos del usuario, organizados como un conjunto de cadenas.

3. Validar campos, verificat que cada campo no esté vacío y que sus símbolos pertenezcan al alfabeto definido.

4. Mostrar resultado, se presenta el estado del formulario según las validaciones realizadas.

b. ¿Qué conocimiento adicional se requiere?

Para diseñar la solución se requieren dos formalismos claves de la teoría de lenguajes formales: las **Potencia de una Cadena**, que describen su relación con una longitud,**Cadena vacía** su proceso dentro de los lenguajes vacío y unitario.

Potencia de una cadena

Sea $w \in \Sigma^*$ y $n \in \mathbb{N}$. La *potencia* n-ésima de la cadena w, que se denota como w^n, se define de manera recursiva de la siguiente forma:

$$w^0 = \varepsilon, \qquad w^n = w^{n-1}w \quad \text{para } n \geq 1$$

Ejemplo

Si $w = ab$, entonces: $w^1 = ab, \quad w^2 = abab, \quad w^3 = ababab$

Relación entre potencia y longitud

La potencia de una cadena está directamente relacionada con su longitud, tal como se expresa en la siguiente propiedad.

Sea $w \in \Sigma^*$ y $n \in \mathbb{N}$. Entonces: $|w^n| = n \cdot |w|$

La demostración se realiza por inducción sobre n.

Caso base: Para $n = 0$, se tiene $w^0 = \varepsilon$ y por tanto:

$$|w^0| = |\varepsilon| = 0 = 0 \cdot |w|$$

Paso inductivo: Supóngase que la propiedad se cumple para algún $n \geq 0$, es decir, $|w^n| = n \cdot |w|$.

Entonces: $|w^{n+1}| = |w^n w| = |w^n| + |w| = n \cdot |w| + |w| = (n+1) \cdot |w|$

Por lo tanto, la propiedad se cumple para todo $n \in \mathbb{N}$.

Cadena vacía

$$|\varepsilon| = 0$$

Esta cadena cumple un papel fundamental en la definición de operaciones sobre cadenas y lenguajes, ya que actúa como el elemento neutro de la concatenación.

Ejemplos:

1. La cadena ε sobre cualquier alfabeto Σ, que no contiene ningún símbolo y cuya longitud es $|\varepsilon| = 0$.
2. En un formulario web, el campo ”Comentarios” cuando está vacío antes de enviarlo representa justamente la cadena ε.
3. En las expresiones regulares, el patrón vacío es aquel que coincide con ”nada”, es decir, permite avanzar inmediatamente al siguiente carácter sin consumir ninguno.
4. Una variable de texto inicializada sin contenido: ‘string texto = ””;‘ (cadena ϵ).

Lenguaje vacío y lenguaje unitario

El lenguaje $\{\varepsilon\}$ es el lenguaje que contiene únicamente la cadena vacía y el lenguaje vacío, denotado por $\emptyset$, es el lenguaje que no contiene ninguna cadena.

Es fundamental distinguir entre ambos lenguajes, ya que:

$$\emptyset \neq \{\epsilon\}, \qquad |\emptyset| = 0, \qquad |\{\epsilon\}| = 1$$

c. ¿Qué se puede reutilizar de soluciones pasadas?

Este problema retoma conceptos fundamentales que ya se desarrollaron en ejercicios anteriores.

En primer lugar, se reutiliza la noción de conjunto para representar el alfabeto Σ, idea que ya había aparecido en el problema de segmentación institucional.

Asimismo, se retoma el concepto de cadena como secuencia de símbolos, trabajado en el sistema de autenticación.

También se retoma la validación por pertenencia: una cadena es válida si pertenece a un conjunto que cumple ciertas condiciones.

La novedad aquí es que se excluye explícitamente la cadena vacía. Así se puede distinguir entre Σ^* (que la incluye) y Σ^+ (que no), lo cual sirve como base para validar campos obligatorios.

d. Diseño de la solución para el problema

La estrategia de resolución conserva la estructura que se ha utilizado en los ejercicios anteriores:

Sea Σ un alfabeto finito que contiene los símbolos permitidos en los campos del formulario.

Cada campo obligatorio se modela como una cadena:

$$w \in \Sigma^*$$

Un campo es estructuralmente válido si:

$$w_i \in \Sigma^+$$

es decir, si pertenece al conjunto de cadenas no vacías formadas con símbolos de Σ.

El formulario completo se representa como:

$$F = (w_1, w_2, \ldots, w_n)$$

El conjunto de formularios válidos queda definido como:

$$L = \{F \mid \forall i,\ w_i \in \Sigma^+\}$$

Por tanto, un formulario es válido si todos sus campos contienen al menos un símbolo permitido y ninguno corresponde a la cadena vacía.

Codificación

Esta sección muestra cómo traducir dichas representaciones a Python.

a. Algoritmo general de reconocimiento

El siguiente algoritmo de validación debe reflejar cada una de las etapas definidas, se plantea de acuerdo con la descomposición:

Código 1.31: `main()`

```
1 def main():
2     sigma = definir_alfabeto()
3     formulario = ingresar_formulario()
4     resultado = validar_campos(formulario, sigma)
5     mostrar_resultado(resultado)
```

Propósito computacional: Coordina la ejecución del programa mediante el llamado de las funciones definidas.

Interpretación conceptual: Esta función organiza el flujo del sistema, pero sin hacer ningún procesamiento interno por sí misma.

Interpretación formal: Define una secuencia de evaluación sobre una tupla de palabras $(w_1, w_2, w_3) \in \Sigma^*$. Para ello, aplica funciones de validación sin incorporar lógica adicional.

Código 1.32: `definir_alfabeto()`

```
 9 def definir_alfabeto():
10     letras = set("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ")
11     digitos = set("0123456789")
12     caracteres = set("@_.")
13     espacios = set(" ")
14     sigma = letras.union(digitos)
15     sigma = sigma.union(caracteres)
16     sigma = sigma.union(espacios)
17 
18     return sigma
```

Propósito computacional: Define el alfabeto del sistema a partir de la unión de letras, dígitos, caracteres especiales y espacios permitidos.

Interpretación conceptual: Se construye el conjunto de símbolos válidos que pueden ser utilizados en los campos del formulario.

Interpretación formal: Se define el alfabeto Σ como: $\Sigma = L \cup D \cup C \cup E$, donde L es el conjunto de letras, D el conjunto de dígitos, C el conjunto de caracteres especiales y E el conjunto que contiene el espacio.

Código 1.33: `ingresar_formulario()`

```
23 def ingresar_formulario():
24     nombre = input("Ingrese nombre: ")
25     documento = input("Ingrese documento: ")
26     correo = input("Ingrese correo: ")
27     formulario = (nombre, documento, correo)
28 
29     return formulario
```

Propósito computacional: Esta función solicita al usuario que ingrese los datos del formulario y los agrupa en una estructura.

Interpretación conceptual: Se capturan múltiples entradas que representan los campos del formulario que se van a validar.

Interpretación formal: Como resultado, se obtienen palabras $w_1, w_2, w_3 \in \Sigma^*$, organizadas en una tupla (w_1, w_2, w_3).

Código 1.34: `validar_campos()`

```
34 def validar_campos(campos, sigma):
35     formulario_valido = "válido"
36 
37     for campo in campos:
38         if campo == "":
39             formulario_valido = "no es válido"
40 
41         for simbolo in campo:
42             if simbolo not in sigma:
43                 formulario_valido = "no es válido"
44 
45     return formulario_valido
```

Propósito computacional: Evalúa si los campos del formulario cumplen con las condiciones de no estar vacíos y contener únicamente símbolos válidos del alfabeto.

Interpretación conceptual: Se realiza una validación completa del formulario, revisando cada campo y cada símbolo dentro de ellos.

Interpretación formal: Dado un conjunto de palabras (w_1, w_2, w_3), se verifica que para cada w_i:

$w_i \neq \epsilon$

$\forall c \in w_i, ; c \in \Sigma$

Si alguna de estas condiciones no se cumple, el formulario no pertenece al conjunto de cadenas válidas.

Código 1.35: `mostrar_resultado()`

```
50 def mostrar_resultado(resultado):
51     print("Formulario", resultado)
```

Propósito computacional: Esta función muestra en pantalla el resultado de la validación del formulario.

Interpretación conceptual: Se le comunica al usuario si el formulario cumple o no con las condiciones que se han establecido.

Interpretación formal: Dado un valor r, se presenta un mensaje que indica si la tupla de cadenas pertenece o no al lenguaje válido.

1.6. Control vehicular automatizado

En el contexto de los sistemas inteligentes de transporte, el sistema nacional de tránsito ha implementado un esquema de control vehicular automatizado basado en cámaras con reconocimiento óptico de caracteres (OCR). Estas cámaras se encuentran instaladas en peajes, intersecciones, zonas de restricción y corredores de alta circulación. Su objetivo es monitorear el flujo vehicular, detectar infracciones y mejorar los mecanismos de control vial.

Cuando un vehículo atraviesa un punto de control, la cámara captura la imagen de la placa y el sistema OCR produce la cadena de caracteres correspondiente al número identificado. Sin embargo, debido a factores como iluminación deficiente, suciedad, ángulo de captura o velocidad del vehículo, pueden producirse errores en la lectura.

1.6.1 Análisis / Abstracción

Después de analizar la situación problema planteada, se puede identificar al menos un problema relacionado.

a. ¿Cuál es el problema a resolver?

El problema consiste en determinar cuándo una cadena generada por el sistema de reconocimiento óptico de placas puede considerarse válida dentro del esquema de identificación utilizado por el sistema de tránsito.

Cada lectura producida por el OCR corresponde a una secuencia de símbolos obtenida a partir de la imagen capturada por las cámaras. Sin embargo, factores como iluminación deficiente, velocidad del vehículo, ángulo de captura o interferencias visuales pueden producir errores en la lectura y generar cadenas que no corresponden a una placa real.

Por esta razón, es necesario definir criterios que permitan distinguir entre secuencias válidas e inconsistentes. El análisis se centra en la estructura de las cadenas, es decir, en el orden y tipo de símbolos que las componen, independientemente del vehículo al que pertenezcan.

En consecuencia, el problema se orienta a modelar formalmente las cadenas generadas por el sistema y verificar si cumplen el formato esperado para las placas vehiculares.

En términos prácticos, el sistema debe verificar que:

- Los caracteres correspondan únicamente a letras mayúsculas y dígitos.
- La longitud coincida con el estándar nacional establecido.
- Se cumpla la estructura definida para la placa vehicular.

Esto permite interpretar una norma administrativa como un lenguaje formal definido mediante reglas estructurales:

$$\text{Norma administrativa} \Rightarrow \text{Lenguaje formal}$$

b. ¿Cuál es la información necesaria?

Para modelar el problema es necesario identificar los siguientes elementos:

- El alfabeto Σ, compuesto por letras mayúsculas y dígitos.
- Las lecturas del sistema OCR, representadas como cadenas w.
- La longitud de las cadenas, determinada por el estándar de las placas.
- La estructura o patrón que deben cumplir las cadenas (por ejemplo, $LLLDDD$).

1.6.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

El análisis de las placas capturadas por el sistema de reconocimiento óptico de caracteres (OCR) se aborda desde la teoría básica de lenguajes formales. Para ello, se considera que cada lectura generada por el sistema puede representarse como una cadena de caracteres. Dicha cadena se construye sobre un alfabeto Σ que describe los símbolos que pueden aparecer en una placa vehicular, como letras y dígitos.

A partir de esta representación formal, es posible analizar si la cadena obtenida corresponde a una placa válida según las reglas que establece el sistema de tránsito. De esta forma, el problema puede modelarse como un proceso de verificación en el que se comprueba si la cadena pertenece o no al lenguaje que describe el formato permitido de las placas vehiculares.

Para resolver el problema, se propone la siguiente descomposición lógica:

1. Identificar simbolos, se definen los conjuntos de letras y dígitos que serán utilizarán en el sistema.
2. Definir alfabeto, se construye el alfabeto Σ como la unión de los símbolos definidos.
3. Leer placa, se obtiene la cadena ingresada por el sistema OCR.
4. Pertenece alfabeto, se verifica que todos los símbolos de la cadena pertenezcan al alfabeto.
5. Verificar formato placa, se valida que la cadena cumpla con la estructura definida para una placa.
6. Evaluar placa, se determina si la cadena corresponde a una placa válida.
7. Mostrar resultado, se presenta el resultado obtenido.

b. ¿Qué conocimiento adicional se requiere?

Para diseñar la solución del problema es necesario introducir algunos conceptos básicos de la teoría de lenguajes formales. En particular, se requiere comprender la noción de **Potencia de una Cadena, Clausuras Aplicadas a una cadena, Kleene y Positiva**, los cuales permiten representar formalmente las lecturas generadas por el sistema OCR.

Asimismo, es necesario utilizar el concepto de cadena vacía y las operaciones de verificación sobre cadenas para determinar si una lectura cumple con la estructura definida para las placas vehiculares. Estos elementos permiten modelar el problema de validación como la verificación de pertenencia de una cadena a un lenguaje que describe el formato permitido de las placas dentro del sistema de control vehicular.

Potencia de un lenguaje

Sea $L \subseteq \Sigma^*$. La *potencia* n-ésima de L, denotada por L^n, se define como:

$$L^0 = \{\varepsilon\}, \qquad L^n = L^{n-1}L \quad \text{para } n \geq 1$$

Ejemplo Si $L = \{a\}$, entonces:

$$L^2 = \{aa\}, \qquad L^3 = \{aaa\}$$

Clausura de Kleene y Positiva de una cadena

Cuando aplicamos clausuras a una cadena individual $w \in \Sigma^*$, estamos generando un **conjunto de cadenas** formado exclusivamente por las potencias de esa cadena.

Sea w una cadena. Se definen sus clausuras como:

- **Clausura Asterisco (w^*):** El conjunto de todas las potencias de w, incluyendo la potencia cero. $w^* = \{w^n \mid n \geq 0\} = \{\epsilon, w, ww, www, \dots\}$
- **Clausura Positiva (w^+):** El conjunto de todas las potencias de w a partir de la primera potencia. $w^+ = \{w^n \mid n \geq 1\} = \{w, ww, www, \dots\}$

Ejemplo: Para la cadena $w = 01$:

- $w^* = \{\epsilon, 01, 0101, 010101, \dots\}$
- $w^+ = \{01, 0101, 010101, \dots\}$

Identidad Universal: En ambos niveles se cumple la relación:

$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$

Comparación y Diferencias Críticas

Para evitar confusiones comunes en el estudio de la Teoría de Lenguajes Formales, a continuación se describen las diferencias fundamentales entre aplicar una clausura a un objeto individual (cadena) frente a un conjunto (lenguaje):

- **Variedad de los elementos:** Mientas que la clausura de una cadena (w^*) genera una secuencia *uniforme* que solo repite el mismo patrón rígido (ej. si $w = ab$, el conjunto es $\{ab, abab, ababab, \dots\}$), la clausura de un lenguaje (L^*) genera combinaciones *diversas*, mezclando distintos elementos del conjunto original (ej. si $L = \{a, b\}$, el lenguaje incluye $\{aa, ab, ba, bb, \dots\}$).
- **Crecimiento y Tamaño:** El crecimiento en una clausura de cadena es *lineal* respecto a la longitud de la cadena original $|w|$. En contraste, el crecimiento en la clausura de un lenguaje es *exponencial* en términos de combinatoria, dependiendo de la cantidad de cadenas distintas que conformen el lenguaje L.
- **Presencia de la cadena vacía (ϵ):** En el nivel de cadena, w^* siempre incluye a ϵ por la definición de la potencia w^0. En el nivel de lenguaje, L^* siempre incluye a ϵ porque el conjunto resultante siempre contiene a $L^0 = \{\epsilon\}$. Esta es la identidad fundamental que une ambos conceptos.
- **Restricciones estructurales:** La clausura de cadena es altamente predecible; nunca aparecerán símbolos en un orden que no estuviera en la cadena base. En la clausura de lenguaje, el orden se vuelve mucho más complejo debido a la permutación de los elementos del conjunto durante la concatenación.

c. ¿Qué se puede reutilizar de soluciones pasadas?

Este problema retoma y articula varios conceptos desarrollados en los ejercicios anteriores.

En primer lugar, se reutiliza la noción de conjunto para definir el alfabeto Σ, como se trabajó en el problema de segmentación institucional.

Asimismo, se retoma el concepto de cadena como secuencia de símbolos, utilizado en los sistemas de autenticación y en el análisis de transacciones.

También se reutiliza la idea de validación mediante pertenencia a un lenguaje, algo que ya se había aplicado en la verificación de formularios digitales.

En este caso, se incorpora un nuevo elemento: la estructura interna de la cadena, que permite no solo verificar los símbolos y la longitud, sino también la posición que cada tipo de símbolo debe ocupar dentro de la cadena.

De esta manera, se integran los conceptos de alfabeto, cadena, longitud y validación, extendiéndolos hacia el reconocimiento de patrones estructurales en las cadenas.

d. Diseño de la solución para el problema

Sea Σ el alfabeto que contiene los símbolos que pueden aparecer en una placa vehicular. En este caso se consideran dos tipos de símbolos:

$$L = \{A, B, C, \ldots, Z\}$$

$$D = \{0, 1, 2, \ldots, 9\}$$

Por lo tanto, el alfabeto del sistema se define como:

$$\Sigma = L \cup D$$

Cada lectura generada por el sistema de reconocimiento óptico de caracteres se modela como una cadena:

$$w \in \Sigma^*$$

Sin embargo, no todas las cadenas sobre Σ representan una placa válida. Las placas deben respetar un formato estructural previamente establecido. En este contexto, se considera el siguiente patrón estructural:

$$LLLDDD$$

donde las tres primeras posiciones corresponden a letras y las tres últimas a dígitos.

El conjunto de placas válidas puede definirse entonces como el lenguaje:

$$P = \{\, w \in \Sigma^* \mid w = l_1 l_2 l_3 d_1 d_2 d_3,\ l_i \in L,\ d_j \in D \,\}$$

De esta manera, una lectura producida por el sistema OCR será considerada válida si y solo si la cadena pertenece al lenguaje P, es decir, si cumple simultáneamente con el alfabeto permitido y con la estructura definida para las placas vehiculares.

Codificación

Esta sección muestra cómo traducir dichas representaciones a Python.

a. Algoritmo general de reconocimiento

El siguiente algoritmo de validación debe reflejar cada una de las etapas definidas, se plantea de acuerdo con la descomposición:

Código 1.36: `main()`

```
4  def main():
5      simbolos = identificar_simbolos()
6      sigma = definir_alfabeto(simbolos)
7      placa = leer_placa()
8      simbolos_validos = pertenece_alfabeto(placa, sigma)
9      formato_valido = verificar_formato_placa(placa, simbolos)
10     resultado = evaluar_placa(simbolos_validos, formato_valido)
11     mostrar_resultado(resultado)
```

Propósito computacional: Coordina la ejecución del sistema mediante el llamado de las funciones definidas.

Interpretación conceptual: Esta función organiza el flujo del proceso desde la definición de los símbolos hasta la evaluación final.

Interpretación formal: Define una secuencia de evaluación sobre una palabra $w \in \Sigma^*$. Para ello, aplica funciones que verifican condiciones de pertenencia y estructura, pero sin realizar ningún procesamiento interno adicional.

Código 1.37: `identificar_simbolos()`

```
15 def identificar_simbolos():
16     letras = set("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
17     digitos = set("0123456789")
18     simbolos = (letras, digitos)
19
20     return simbolos
```

Propósito computacional: Esta función define los conjuntos de símbolos correspondientes a letras mayúsculas y dígitos.

Interpretación conceptual: Se establecen las categorías de caracteres que serán utilizadas para construir el sistema de validación de placas.

Interpretación formal: Se definen dos subconjuntos L y D tales que $L, D \subset \Sigma$, donde L contiene las letras y D contiene los dígitos. La función retorna la tupla (L, D).

Código 1.38: `definir_alfabeto()`

```
25 def definir_alfabeto(simbolos):
26     sigma = simbolos[0].union(simbolos[1])
27 
28     return sigma
```

Propósito computacional: Esta función construye el alfabeto total a partir de la unión de los conjuntos de símbolos que se definieron previamente.

Interpretación conceptual: Se integran los diferentes tipos de caracteres en un único conjunto de símbolos válidos.

Interpretación formal: Formalmente, el alfabeto Σ se construye como la unión de los dos subconjuntos: $\Sigma = L \cup D$.

Código 1.39: `leer_placa()`

```
32 def leer_placa():
33     placa = input("Ingrese la cadena detectada por el sistema OCR: ")
34 
35     return placa
```

Propósito computacional: Esta función solicita al usuario que ingrese una cadena que representa la lectura de una placa.

Interpretación conceptual: Se captura la cadena detectada por el sistema OCR para que pueda ser analizada más adelante.

Interpretación formal: Se obtiene una palabra $w \in \Sigma^*$, la cual será evaluada respecto a condiciones de pertenencia y estructura.

Código 1.40: `pertenece_alfabeto()`

```
40 def pertenece_alfabeto(cadena, sigma):
41     pertenece = True
42 
43     for simbolo in cadena:
44 
45         if simbolo not in sigma:
46             pertenece = False
47 
48     return pertenece
```

Propósito computacional: Esta función verifica si todos los símbolos de la cadena pertenecen al alfabeto que se definió.

Interpretación conceptual: Se analiza si cada carácter de la placa es válido dentro del sistema.

Interpretación formal: Para una palabra $w \in \Sigma^*$, se verifica que:

$$\forall c \in w,\ c \in \Sigma$$

es decir, cada símbolo de la palabra pertenece al alfabeto definido.

Código 1.41: `verificar_formato_placa()`

```
52 def verificar_formato_placa(cadena, simbolos):
53     formato_valido = False
54     letras = simbolos[0]
55     digitos = simbolos[1]
56 
57     if len(cadena) == 6:
58         parte_letras = cadena[0:3]
59         parte_digitos = cadena[3:6]
60         letras_validas = True
61         digitos_validos = True
62 
63         for c in parte_letras:
64             if c not in letras:
65                 letras_validas = False
66 
67         for c in parte_digitos:
68             if c not in digitos:
69                 digitos_validos = False
70 
71         if letras_validas and digitos_validos:
72             formato_valido = True
73 
74     return formato_valido
```

Propósito computacional: Esta función evalúa si la cadena cumple con la estructura esperada para una placa.

Interpretación conceptual: Se valida que la cadena tenga una forma específica: tres letras seguidas de tres dígitos.

Interpretación formal: Dada una palabra $w \in \Sigma^*$, se verifican dos condiciones. Primero, que su longitud sea seis: $|w| = 6$. Segundo, que se pueda descomponer como $w = xyzuvw$, donde $x, y, z \in L$ (letras) y $u, v, w \in D$ (dígitos). En otras palabras, $w \in L^3D^3$.

Código 1.42: `evaluar_placa`

```
79 def evaluar_placa(simbolos_validos, formato_valido):
80     resultado = "Placa inválida"
81 
82     if simbolos_validos and formato_valido:
83         resultado = "Placa válida"
84 
85     return resultado
```

Propósito computacional: Esta función determina si la cadena corresponde a una placa válida, combinando los resultados de las validaciones previas.

Interpretación conceptual: Se integran dos condiciones: la pertenencia al alfabeto y el cumplimiento del formato esperado. Con base en ambas se toma una decisión final.

Interpretación formal: La palabra w es válida si cumple simultáneamente las siguientes condiciones:

$w \in \Sigma^*$

$w \in L^3D^3$

Es decir, $w \in L^3D^3 \subseteq \Sigma^*$.

Código 1.43: `mostrar_resultado()`

```
90 def mostrar_resultado(resultado):
91     print(resultado)
```

Propósito computacional: Esta función muestra en pantalla el resultado de la evaluación de la placa.

Interpretación conceptual: Se le comunica al usuario si la cadena que ingresó es válida o no.

Interpretación formal: Dado un valor r, se presenta un mensaje que indica si la palabra w pertenece o no al lenguaje definido.

1.7. Programación robótica

En una planta industrial moderna, muchas tareas son ejecutadas por robots programables que realizan movimientos básicos de forma automatizada. Estos robots participan en actividades como ensamblaje de piezas, traslado de componentes, clasificación de productos y manipulación de materiales dentro de la línea de producción.

Cada robot dispone de un conjunto limitado de acciones definidas por el sistema de control, como avanzar, retroceder, girar, tomar objetos o liberar piezas. A partir de estas acciones se construyen secuencias que representan operaciones más complejas dentro del proceso productivo.

Las secuencias ejecutadas por los robots se almacenan para registrar las operaciones realizadas, verificar el funcionamiento del sistema y analizar posteriormente el comportamiento del proceso automatizado. Sin embargo, durante la operación se han detectado registros inconsistentes: algunas secuencias contienen símbolos no válidos o no respetan la estructura esperada por el sistema de control. Esto puede generar errores de interpretación, afectar la trazabilidad e incluso producir fallos en la coordinación de las tareas.

Aunque el conjunto de acciones posibles es finito, las secuencias pueden repetirse múltiples veces y generar combinaciones muy diversas.

1.7.1 Análisis / Abstracción

a. ¿Cuál es el problema a resolver?

El problema consiste en modelar formalmente las secuencias de acciones ejecutadas por el sistema robótico y determinar si corresponden a secuencias válidas dentro del modelo de funcionamiento definido para el proceso industrial.

Cada jornada de operación puede representarse como una cadena formada por símbolos asociados a las acciones del robot. Estas cadenas deben analizarse para verificar que utilicen únicamente movimientos válidos y que respeten el orden esperado por el sistema de control.

Los ingenieros han observado que modificar el orden de las instrucciones produce comportamientos diferentes, incluso cuando se utilizan los mismos comandos. Por esta razón, resulta necesario estudiar formalmente la estructura de las secuencias generadas por el sistema y establecer mecanismos que permitan validar su consistencia.

b. ¿Cuál es la información necesaria?

Para modelar el comportamiento del sistema es necesario identificar los siguientes elementos:

- El conjunto de acciones básicas del robot.
- La representación simbólica de dichas acciones.
- El alfabeto Σ, definido como el conjunto de símbolos que representan los movimientos.
- Las secuencias de acciones, representadas como cadenas w.
- La posibilidad de repetición de acciones dentro de las secuencias.

1.7.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

El análisis de las secuencias de movimientos del robot se aborda desde la teoría básica de lenguajes formales, considerando que cada operación ejecutada por el robot puede representarse como una cadena construida sobre un alfabeto que describe los distintos tipos de movimientos disponibles en el sistema de control.

Para resolver el problema, se propone la siguiente descomposición lógica:

1. Identificar movimientos, se define el conjunto de símbolos que representan los movimientos del robot.
2. Definir alfabeto, se establece el alfabeto Σ del sistema.
3. Leer secuencia, se obtiene la cadena de movimientos ingresada por el usuario.

4. Pertenece alfabeto, se verifica que todos los símbolos de la secuencia pertenezcan al alfabeto definido.

5. Evaluar secuencia, se determina si la secuencia es válida.

6. Mostrar resultado, se presenta el resultado obtenido.

b. ¿Qué conocimiento adicional se requiere?

Para diseñar la solución del problema es necesario introducir algunos conceptos básicos de la teoría de lenguajes formales, los cuales permiten representar y analizar secuencias de símbolos de manera estructurada.

En particular, se requiere comprender las nociones de **Igual de Cadenas, Lenguajes Finitos e Infinitos, Lenguajes Formales y Operaciones con Lenguajes** y de esta manera poder representar el conjunto finito de movimientos elementales que el robot puede ejecutar, mientras que cada secuencia de movimientos registrada por el sistema puede modelarse como una cadena formada por símbolos pertenecientes a dicho alfabeto.

También se deben considerar nociones como la repetición de símbolos y la verificación de si una cadena pertenece o no a un lenguaje. Esto ayuda a determinar si una secuencia registrada es una combinación válida de movimientos del robot, o si por el contrario tiene símbolos o estructuras que no están dentro de las acciones permitidas por el sistema de control.

Igualdad de Cadenas

Se dice que dos cadenas u y v son iguales (y se escribe $u = v$) cuando tienen la misma longitud y en cada posición los símbolos coinciden. Esta definición es fundamental para el razonamiento formal y para las operaciones que se definen sobre cadenas. **Lenguajes finitos e infinitos**

Definición 1.1. Un lenguaje formal L es *finito* si contiene un número finito de cadenas. En caso contrario, se dice que L es *infinito*.

Ejemplos

- El lenguaje $L_1 = \{a, ab, bba\}$ es finito.
- El lenguaje $L_2 = \{a^n \mid n \geq 0\}$ es infinito.

Lenguajes formales

Una vez que se han definido las nociones de alfabeto y cadena, es posible introducir el concepto central de la Teoría de Lenguajes Formales: el lenguaje formal. De manera intuitiva, un lenguaje formal es un conjunto de cadenas construidas a partir de un alfabeto fijo, que además están sujetas a ciertas restricciones.

Un *lenguaje formal* sobre Σ es un subconjunto de Σ^*. Donde Σ es un alfabeto.

Es decir, un lenguaje formal L cumple:

$$L \subseteq \Sigma^*$$

Cada elemento de L es una cadena formada exclusivamente por símbolos del alfabeto Σ.

Ejemplo: Supóngase $\Sigma = \{a, b\}$. Entonces se pueden definir varios lenguajes formales sobre este alfabeto, como por ejemplo:

- $L_1 = \{\varepsilon\}$ que contiene únicamente la cadena vacía.
- $L_2 = \{a, aa, aaa\}$ es decir, cadenas de a's de longitud 1, 2 o 3.
- $L_3 = \{w \in \Sigma^* \mid |w| \text{ es par}\}$ o sea, todas las cadenas de longitud par.
- $L_4 = \Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, \dots\}$ que incluye todas las cadenas posibles.
- $L_5 = \emptyset$ el lenguaje vacío.

Operaciones sobre lenguajes

Así como es posible definir operaciones sobre cadenas individuales, también se pueden definir operaciones que permiten combinar lenguajes formales para construir nuevos lenguajes. Estas operaciones son fundamentales para el análisis estructural de los lenguajes y para la definición de clases de lenguajes reconocidas por distintos modelos de cómputo.

Sean $L_1, L_2 \subseteq \Sigma^*$ dos lenguajes formales definidos sobre un mismo alfabeto Σ.

Unión: La *unión* de dos lenguajes L_1 y L_2 se define como:

$$L_1 \cup L_2 = \{\, w \mid w \in L_1 \text{ o } \in L_2 \,\}$$

Ejemplo Si $L_1 = \{a, ab\}$ y $L_2 = \{b\}$, entonces:

$$L_1 \cup L_2 = \{a, ab, b\}$$

Intersección: La *intersección* de dos lenguajes L_1 y L_2 se define como:

$$L_1 \cap L_2 = \{\, w \mid w \in L_1 \text{ y } w \in L_2 \,\}$$

Ejemplo Si $L_1 = \{a, ab\}$ y $L_2 = \{ab, b\}$, entonces:

$$L_1 \cap L_2 = \{ab\}$$

Diferencia: La *diferencia* de dos lenguajes L_1 y L_2 se define como:

$$L_1 \setminus L_2 = \{\, w \mid w \in L_1 \text{ y } w \notin L_2 \,\}$$

Ejemplos

- Si $L_1 = \{a, ab\}$ y $L_2 = \{ab\}$, entonces:

$$L_1 \setminus L_2 = \{a\}$$

- Sea $\Sigma = \{0, 1\}$ y:

$$L_1 = \{w \in \{0,1\}^* \mid |w| \text{ es par}\}$$

$$L_1 = \{\varepsilon,\ 00,\ 01,\ 10,\ 11,\ 0000,\ 0001,\ 0010,\ 0011,\ 0100, \dots\}$$

$$L_2 = \{w \in \{0,1\}^* \mid w \text{ comienza con } 0\}$$

$$L_2 = \{0,\ 00,\ 01,\ 000,\ 001,\ 010,\ 011,\ 0000, \dots\}$$

Resultado: $L_1 \setminus L_2 = \{\varepsilon,\ 11,\ 101,\ 111,\ 1001,\ 1011, \dots\}$

- Si $L_1 = \{aa, aaa, aaaa\}$ y $L_2 = \{aaa\}$, entonces:

$$L_1 \setminus L_2 = \{aa, aaaa\}$$

Concatenación de lenguajes: La *concatenación* de dos lenguajes L_1 y L_2 se define como:

$$L_1 L_2 = \{\, uv \mid u \in L_1,\ v \in L_2 \,\}$$

Ejemplo Si $L_1 = \{a, ab\}$ y $L_2 = \{b\}$, entonces:

$$L_1 L_2 = \{ab, abb\}$$

c. ¿Qué se puede reutilizar de soluciones pasadas?

Este problema retoma y extiende los conceptos desarrollados en los ejercicios anteriores.

En primer lugar, se reutiliza la noción de conjunto para definir el alfabeto Σ, como se trabajó en el problema de segmentación institucional.

Asimismo, se retoma el concepto de cadena como secuencia de símbolos, utilizado en los sistemas de autenticación, en el análisis de transacciones y en el control vehicular.

También se reutiliza la idea de validación mediante pertenencia a un lenguaje, algo que ya se había aplicado en la verificación de formularios y en el reconocimiento de placas.

En este caso se incorpora un nuevo elemento: la repetición de acciones. Esto permite modelar secuencias de longitud variable dentro del sistema, y conduce al uso de Σ^* como representación de todas las posibles secuencias de movimientos del robot.

De esta forma, se integran conceptos como alfabeto, cadena, longitud y validación, y se extienden hacia la representación de procesos repetitivos en sistemas automatizados.

Todo esto evidencia un proceso de reconocimiento de patrones, donde las nociones de cadena y alfabeto se reutilizan para modelar secuencias más complejas, incorporando la repetición como una característica fundamental en sistemas automatizados.

d. Diseño de la solución para el problema

Para abordar el problema planteado es necesario modelar formalmente las secuencias de movimientos ejecutadas por el robot dentro de la planta industrial. En este contexto, cada movimiento básico puede representarse mediante un símbolo, y la combinación ordenada de dichos símbolos permite describir las diferentes acciones que el robot realiza durante una operación dentro de la línea de producción.

En primer lugar, se define el conjunto de movimientos elementales que el robot puede ejecutar. Cada movimiento se representa mediante un símbolo dentro de un alfabeto finito. Por ejemplo, se puede considerar el siguiente conjunto de movimientos:

$$M = \{A, R, G, T, S\}$$

Donde cada símbolo representa una acción básica del robot dentro del sistema de control, como avanzar, retroceder, girar, tomar un objeto o soltarlo. A partir de este conjunto de movimientos se define el alfabeto del sistema:

$$\Sigma = M$$

Este alfabeto contiene todos los símbolos válidos que pueden aparecer en una secuencia de acciones del robot.

Cada operación ejecutada por el robot puede modelarse entonces como una cadena w formada por símbolos pertenecientes a este alfabeto. En términos formales:

$$w \in \Sigma^*$$

donde Σ^* representa el conjunto de todas las cadenas finitas que pueden construirse utilizando los símbolos del alfabeto.

De esta manera, una secuencia como:

$$w = ATGAS$$

puede interpretarse como una serie ordenada de movimientos ejecutados por el robot dentro de la línea de producción.

El problema consiste en analizar estas cadenas y verificar que cada símbolo que las compone pertenece al alfabeto definido para los movimientos del robot. Si todos los símbolos de la secuencia pertenecen al alfabeto Σ, entonces la cadena puede considerarse una representación válida dentro del sistema de control.

Formalmente, el lenguaje que describe las secuencias válidas de movimientos del robot puede definirse como:

$$L = \{w \in \Sigma^* \mid \forall c \in w,\ c \in \Sigma\}$$

Este lenguaje contiene todas las secuencias de movimientos formadas únicamente por acciones válidas del robot. Cualquier cadena que contenga un símbolo que no pertenezca al alfabeto definido será considerada inválida dentro del sistema de control.

Codificación

Esta sección muestra cómo traducir dichas representaciones a Python.

a. Algoritmo general de reconocimiento

Código 1.44: `identificar_movimientos()`

```
4  def main():
5      movimientos = identificar_movimientos()
6      sigma = definir_alfabeto(movimientos)
7      secuencia = leer_secuencia()
8      simbolos_validos = pertenece_alfabeto(secuencia, sigma)
9      resultado = evaluar_secuencia(simbolos_validos)
10     mostrar_resultado(resultado)
```

Propósito computacional: Esta función coordina la ejecución del programa mediante el llamado de las funciones que la componen.

Interpretación conceptual: Organiza el flujo del sistema desde la definición de los movimientos hasta la validación de la secuencia.

Interpretación formal: Define una secuencia de evaluación sobre una palabra $w \in \Sigma^*$. Para ello, aplica funciones que verifican la pertenencia de sus símbolos al alfabeto, sin realizar ningún procesamiento interno adicional.

Código 1.45: `identificar_movimientos()`

```
14 def identificar_movimientos():
15     movimientos = set("ARGTS")
16
17     return movimientos
```

Propósito computacional: Esta función define el conjunto de símbolos que representan los posibles movimientos del robot.

Interpretación conceptual: Se establecen las acciones básicas que el robot puede ejecutar dentro del sistema.

Interpretación formal: Se define un conjunto $M \subset \Sigma$ tal que $M = \{A, R, G, T, S\}$, donde cada elemento representa uno de los movimientos permitidos. La función retorna este conjunto.

Código 1.46: `definir_alfabeto()`

```
22 def definir_alfabeto(movimientos):
23     sigma = movimientos
24 
25     return sigma
```

Propósito computacional: Asigna el conjunto de movimientos permitidos como alfabeto del sistema.

Interpretación conceptual: Establece los símbolos válidos que pueden utilizarse para representar movimientos dentro del sistema.

Interpretación formal: Se define el alfabeto $\Sigma = M$, donde M es el conjunto de movimientos permitidos.

Código 1.47: `leer_secuencia()`

```
30 def leer_secuencia():
31     secuencia = input("Ingrese la secuencia de movimientos del robot: ")
32 
33     return secuencia
```

Propósito computacional: Esta función solicita al usuario que ingrese una secuencia de movimientos.

Interpretación conceptual: Se captura la cadena que representa el recorrido o las acciones que ejecutó el robot.

Interpretación formal: Se obtiene una palabra $w \in \Sigma^*$, la cual será evaluada respecto a su validez.

Código 1.48: `pertenece_alfabeto()`

```
38 def pertenece_alfabeto(secuencia, sigma):
39     valido = True
40 
41     for c in secuencia:
42         if c not in sigma:
43             valido = False
44 
45     return valido
```

Propósito computacional: Verifica si todos los símbolos de la secuencia pertenecen al alfabeto definido.

Interpretación conceptual: Se analiza si cada movimiento registrado es válido dentro del conjunto de movimientos permitidos.

Interpretación formal: Para una palabra $w \in \Sigma^*$, se verifica que $\forall c \in w, c \in \Sigma$. Si algún símbolo no pertenece a Σ, la secuencia se considera inválida.

Código 1.49: `evaluar_secuencia()`

```
50 def evaluar_secuencia(simbolos_validos):
51     resultado = "Secuencia inválida"
52 
53     if simbolos_validos:
54         resultado = "Secuencia válida"
55 
56     return resultado
```

Propósito computacional: Esta función determina si la secuencia es válida en función del resultado de la verificación de los símbolos.

Interpretación conceptual: Se toma una decisión final sobre si la secuencia del robot es válida o no.

Interpretación formal: La palabra w es válida si cumple la condición $w \in \Sigma^*$, es decir, si todos sus símbolos pertenecen al alfabeto que se definió.

Código 1.50: `mostrar_resultado()`

```
61 def mostrar_resultado(mensaje):
62     print(mensaje)
63 
64 main()
```

Propósito computacional: Esta función muestra en pantalla el resultado de la evaluación de la secuencia.

Interpretación conceptual: Se le comunica al usuario si la secuencia que ingresó es válida o no.

Interpretación formal: Dado un valor r, se presenta un mensaje que indica si la palabra w pertenece o no al lenguaje definido.

1.8. Ejercicios

Ejercicios básicos

1. Sea $\Sigma = \{a, b\}$ y el lenguaje:

$$L = \{w \in \Sigma^* \mid w \text{ termina en } a\}$$

Determine si las siguientes cadenas pertenecen a L:

$$a,\ ba,\ abb,\ abba$$

2. Sea $\Sigma = \{0, 1\}$ y el lenguaje:

$$L = \{w \in \Sigma^* \mid |w| \text{ es par}\}$$

Determine cuáles de las siguientes cadenas pertenecen a L:

$$\varepsilon,\ 0,\ 01,\ 1010,\ 111$$

3. Sea $\Sigma = \{a, b\}$ y el lenguaje:

$$L = \{a^n b^n \mid n \geq 0\}$$

Indique cuáles de las siguientes cadenas pertenecen a L:

$$\varepsilon,\ ab,\ aabb,\ aaabbb,\ aabbb$$

4. Dado el alfabeto $\Sigma = \{x, y, z\}$, determine y liste los elementos de los siguientes conjuntos:

   a) Σ^1

   b) El subconjunto de cadenas pertenecientes a Σ^+ cuya longitud sea exactamente 2 ($|w| = 2$).

5. Sea el lenguaje $L = \{10, 01\}$. Indique si las siguientes cadenas pertenecen a L^+, justificando su respuesta mediante la descomposición en elementos de L:

   a) 100110

   b) 101

   c) ϵ (cadena vacía)

6. Para el lenguaje $L = \{a, b\}$, escriba explícitamente los primeros 5 elementos de L^* y los primeros 5 elementos de L^+, ordenándolos de menor a mayor longitud.

7. Sea $\Sigma = \{a, b\}$ y el lenguaje:

$$L = \{w \in \Sigma^* \mid w \text{ comienza con } a \text{ y termina con } b\}$$

Analice la pertenencia de las siguientes cadenas:

$$ab,\ aab,\ abb,\ baab$$

8. Sea $\Sigma = \{a, b\}$ y los lenguajes:

$$L_1 = \{w \mid w \text{ contiene exactamente dos símbolos } a\}$$

$$L_2 = \{w \mid w \text{ termina en } b\}$$

Determine si cada cadena pertenece a $L_1 \cap L_2$:

$$aab,\ aba,\ abb,\ aaab$$

9. Sea la cadena $w = abc$.

   a) Construya el conjunto explícito para la potencia w^2.

   b) Describa la estructura del conjunto infinito w^+.

   c) Explique formalmente por qué la cadena $aabbcc \notin w^+$, mientras que $abcabc \in w^+$.

10. Sea $L = \{0, 1\}$.

    a) Describa verbalmente el tipo de cadenas que representa el lenguaje L^+.

    b) Establezca la diferencia composicional entre el lenguaje generado por $\{0, 1\}^+$ y el conjunto generado por la clausura de la cadena $w = 01$ (es decir, w^+).

11. Sea la cadena $w = abab$. Identifique y liste todos los sufijos de w (según la Definición 1.5) que sean también elementos del lenguaje Σ^+ donde $\Sigma = \{a, b\}$.

12. Dado el alfabeto $\Sigma = \{a, b\}$ y el lenguaje $L = \{aa, b\}$, determine:

    a) Los elementos de las potencias L^2 y L^3.

b) Liste cinco cadenas de longitud par que pertenezcan a la clausura positiva L^+.

c) Explique, basándose en la definición de concatenación, por qué $aab \in L^+$ pero $aba \notin L^+$.

13. Sea $w = 101$.

a) Construya el conjunto explícito de la clausura positiva w^+.

b) Aplique la propiedad $|w^n| = n \cdot |w|$ (Definición 1.3) para determinar la longitud de la quinta cadena de este conjunto.

14. Sea $\Sigma = \{0, 1\}$ y el lenguaje $L = \{w \in \Sigma^+ \mid w$ comienza con 0 y termina con $1\}$.

a) Si tomamos una cadena $u \in L$, ¿pertenecen todas las cadenas de la clausura u^+ al lenguaje L? Justifique analizando los prefijos y sufijos de la concatenación uu.

15. Sea la cadena $x = ababab$. Identifique si x puede expresarse como un elemento de L^+ para los siguientes lenguajes:

a) $L = \{ab\}$

b) $L = \{a, b\}$

c) $L = \{aba, b\}$

16. Demuestre formalmente que para cualquier lenguaje L, se cumple la igualdad $L \cdot L^* = L^+$. Utilice esta identidad para explicar el resultado de la operación cuando $L = \emptyset$.

17. Sea $w \in \Sigma^+$. Demuestre que si una cadena u es subcadena de w, entonces u es necesariamente subcadena de cualquier elemento perteneciente a la clausura positiva w^+.

18. Sean $L_1 = \{a\}^+$ y $L_2 = \{aa\}^+$.

a) Describa el lenguaje resultante de la intersección $L_1 \cap L_2$.

b) Justifique si se cumple que $L_2 \subset L_1$.

Ejercicios de Codificación

Para cada ejercicio propuesto, desarrolle de manera completa el proceso de solución siguiendo las etapas trabajadas en este capítulo.

Ejercicio 1: Aviación - Sistema ACARS

El sistema ACARS (Aircraft Communications Addressing and Reporting System) es un estándar de aviación que permite la transmisión automática de mensajes digitales entre aviones y estaciones en tierra. A través de este sistema se intercambia información operativa, como rutas de vuelo, estado de la aeronave y reportes de posición. En algunos procedimientos, los mensajes llevan identificadores numéricos de ruta que deben tener una longitud fija. Esto es necesario para que los sistemas de comunicación los interpreten sin errores y para evitar ambigüedades en lo que se transmite. Desde la perspectiva de los lenguajes formales, estos identificadores pueden modelarse como cadenas formadas únicamente por dígitos decimales y cuya longitud es exactamente de cinco símbolos.

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$L_1 = \{w \in \Sigma^+ \mid |w| = 5\}$

Ejercicio 2: Salud - EHR/Genómica

En el campo de la bioinformática y de los sistemas de registros electrónicos de salud (EHR), las plataformas de análisis genómico manejan grandes cantidades de información generada por técnicas de secuenciación de ADN, como la secuenciación de nueva generación (NGS).

Las secuencias de ADN se representan mediante cuatro bases nitrogenadas: Adenina (A), Timina (T), Guanina (G) y Citosina (C). En ciertos procesos de análisis y control de calidad de los datos, es necesario verificar propiedades estructurales de las secuencias antes de realizar etapas posteriores como el alineamiento con genomas de referencia.

Un caso particular consiste en analizar secuencias donde las bases complementarias se encuentran balanceadas, es decir, el número de adeninas coincide con el número de timinas, y el número de guaninas coincide con el número de citosinas.

Este tipo de secuencias puede modelarse mediante un lenguaje formal definido sobre el alfabeto de las bases nitrogenadas.

$\Sigma = \{A, T, G, C\}$

$L_2 = \{w \in \Sigma^* \mid \#A(w) = \#T(w) \text{ y } \#G(w) = \#C(w)\}$

Ejercicio 3: Blockchain - Bitcoin/Ethereum

En sistemas de tecnología blockchain, como Bitcoin o Ethereum, las transacciones digitales incluyen diferentes campos codificados que permiten verificar la integridad de la información antes de que esta sea procesada por los nodos de la red.

En ciertos esquemas simplificados de validación, las cadenas binarias que representan identificadores o estructuras de datos pueden incluir prefijos o sufijos específicos que permiten detectar errores de transmisión o manipulación de la información antes del proceso de validación completa.

Desde el punto de vista de los lenguajes formales, este tipo de estructuras puede modelarse como cadenas binarias que comienzan con un símbolo específico y terminan con una secuencia determinada de bits.

$\Sigma = \{0, 1\}$

$L_3 = \{w \in \Sigma^* \mid w[0] = 0 \text{ y termina con } 11\}$

Ejercicio 4: IoT - Smart Grid Sensores

En las redes eléctricas inteligentes (Smart Grid), diversos sensores distribuidos a lo largo del sistema monitorean continuamente el estado de la red eléctrica. Dispositivos como las Phasor Measurement Units (PMU) registran y transmiten información sobre el comportamiento del voltaje y la frecuencia en intervalos de tiempo muy cortos.

En ciertos sistemas de monitoreo sencillos, los sensores reportan estados que se representan con símbolos discretos. Así, por ejemplo, la letra H podría usarse para indicar voltaje alto, mientras que L indicaría voltaje bajo.

Para ciertos procesos de verificación de datos, es necesario analizar secuencias de lecturas que cumplan condiciones estructurales específicas, como tener una longitud par y presentar un número impar de estados de voltaje alto.

Este tipo de secuencias puede representarse formalmente mediante un lenguaje definido sobre el alfabeto de estados del sensor.

$\Sigma = \{H, L\}$

$L_4 = \{w \in \Sigma^* \mid |w| \text{ es par y } \#H(w) \text{ es impar}\}$

1.9. Cierre del capítulo

En este capítulo se estudió el proceso de resolución de problemas computacionales a partir de diferentes contextos del mundo real. Cada situación fue analizada mediante etapas de abstracción, identificación de la información relevante, descomposición en pasos lógicos y diseño de soluciones apoyadas en modelos formales.

A lo largo del capítulo se introdujeron herramientas como la teoría de conjuntos, los lenguajes formales y la representación de cadenas sobre alfabetos. Estas herramientas permitieron modelar distintas situaciones de manera rigurosa y mostraron que problemas aparentemente diferentes pueden representarse mediante estructuras formales similares.

Las soluciones teóricas se complementaron con implementaciones en Python, evidenciando la relación entre el modelo formal y su aplicación práctica. De esta manera, el lector puede observar cómo los conceptos abstractos se transforman en algoritmos capaces de verificar condiciones, validar información y analizar datos dentro de un sistema.

Con ello, el capítulo establece las bases necesarias para estudiar modelos computacionales más avanzados, donde las estructuras formales y el reconocimiento de patrones son fundamentales en el diseño de soluciones eficientes.

En el próximo capítulo se analizará cómo automatizar el reconocimiento de ciertos lenguajes, introduciendo así el estudio de los lenguajes regulares y los autómatas finitos.

Capítulo 2

Lenguajes regulares

¿Cómo automatizar la validación de patrones simples de entrada?

En los sistemas informáticos constantemente se requiere verificar si una entrada cumple con un formato esperado. Por ejemplo, un sistema puede necesitar validar:

- identificadores de usuario, por ejemplo: `juan_perez23`.
- números enteros o reales, como `42`, `-7` o `3.1416`.
- códigos con formatos específicos, como códigos postales (`110111`) o códigos de productos (`PRD-4589`).
- secuencias simples de comandos, como `abrir`, `cerrar`, `leer` o `escribir`.
- cadenas que cumplen restricciones estructurales básicas, como `aabb` o `abba`, que contienen un número par de símbolos `a`.

En todos estos casos los patrones son sencillos y pueden describirse con un número finito de reglas. En esencia, el problema consiste en determinar, de forma automática y precisa, si una cadena de símbolos pertenece al conjunto de cadenas válidas. Para abordar estos problemas, este capítulo introduce los **lenguajes regulares** como marco teórico formal.

A lo largo del capítulo se plantean distintos problemas que motivan la introducción de los conceptos fundamentales: autómatas finitos deterministas y no deterministas, expresiones regulares, propiedades de clausura, limitaciones y técnicas de optimización. Cada problema se

resuelve siguiendo una misma metodología que integra análisis, diseño, codificación y pruebas.

Nota histórica

Los fundamentos de la teoría de autómatas y lenguajes regulares se establecieron en la década de 1950, durante los primeros años de desarrollo de la teoría de la computación.

Los orígenes conceptuales se remontan a 1943, cuando **Warren McCulloch** y **Walter Pitts** propusieron un modelo matemático de redes neuronales artificiales basado en unidades con dos estados [McCulloch and Pitts, 1943]. Este modelo sentó las bases formales sobre las que posteriormente se desarrollaría la teoría de autómatas finitos.

En 1956, **Stephen Kleene** introdujo las *expresiones regulares* y demostró que estas eran equivalentes en poder expresivo a los autómatas finitos, resultado conocido como el **Teorema de Kleene** [Kleene, 1956]. Este trabajo fue fundamental para establecer diferentes formalismos para describir el mismo conjunto de lenguajes.

Tres años después, en 1959, **Michael Rabin** y **Dana Scott** publicaron su trabajo seminal sobre autómatas finitos, formalizando tanto los autómatas deterministas como los no deterministas, y demostrando la equivalencia entre ambos modelos [Rabin and Scott, 1959]. Por estas contribuciones fundamentales a la teoría de la computación, Rabin y Scott recibieron el **Premio Turing** en 1976.

Estos desarrollos se enmarcan en el contexto más amplio de la investigación sobre los límites de la computación, iniciada por **Alan Turing** en la década de 1930 con su máquina universal. Los autómatas finitos representan el modelo más simple dentro de la jerarquía de Chomsky [Chomsky, 1956], proporcionando la base teórica para aplicaciones prácticas como el análisis léxico en compiladores, búsqueda de patrones y validación de formatos.

2.1. Validación de identificadores de usuario

La empresa TechSecure Solutions, especializada en seguridad informática, se encuentra desarrollando una plataforma de gestión de identidades para sus clientes corporativos. Como parte del proceso de registro, cada usuario debe crear un identificador único que cumpla con estrictas normas de formato. Estas normas garantizan que los identificadores sean legibles, consistentes y compatibles con los sistemas internos de autenticación.

El equipo de desarrollo ha identificado que los identificadores válidos deben comenzar con una letra minúscula, continuar con letras minúsculas o dígitos, y tener al menos tres caracteres de longitud. Cualquier entrada que no cumpla con estas reglas debe ser rechazada de manera inmediata, antes de que llegue a las capas internas del sistema.

Para abordar este problema se necesita un mecanismo automático capaz de decidir con precisión si un identificador proporcionado por el usuario es válido o no.

2.1.1 Análisis / Abstracción

Del escenario anterior se desprende un problema de validación automática de cadenas:

a. ¿Cuál es el problema a resolver?

TechSecure Solutions necesita un mecanismo formal que decida, sobre cada identificador ingresado, si cumple las reglas de formato establecidas. Sin ese mecanismo, entradas mal formadas alcanzan las capas internas del sistema de autenticación.

b. ¿Cuál es la información necesaria?

El validador trabaja sobre cada identificador de forma aislada, sin requerir información adicional del usuario:

Información necesaria	**Tipo de dato**
Cadena de entrada (identificador)	Texto (String)

El formato válido exige tres condiciones: comenzar con una letra minúscula, continuar con letras minúsculas o dígitos, y tener una longitud mínima de tres caracteres.

2.1.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

El procedimiento para validar un identificador de usuario puede realizarse de muchas formas, pero en este caso se propone un enfoque basado en el uso de expresiones regulares y autómatas finitos, que son herramientas formales adecuadas para describir y reconocer patrones de cadenas con formato específico.

- **ENTRADA:** Una cadena de caracteres proporcionada por el usuario como identificador.
- **PROCESAMIENTO:**
  1. Lectura de la cadena de entrada.
  2. Definición de la función de transición del AFD que reconoce el formato válido de identificadores.
  3. Validación del identificador mediante el AFD.
  4. Impresión del resultado de la validación.
- **SALIDA:** Una decisión binaria: `ACEPTADA` si el identificador cumple con el formato, o `RECHAZADA` si no lo hace.

b. ¿Qué conocimiento adicional se requiere?

Para diseñar la solución se requieren dos formalismos claves de la teoría de lenguajes formales: las **expresiones regulares**, que describen algebraicamente el patrón válido, y el **autómata finito determinista**, que implementa el reconocimiento símbolo a símbolo.

Expresiones regulares

Las **expresiones regulares** proporcionan una notación algebraica compacta para describir lenguajes regulares [Hopcroft et al., 2006]. Se emplean en editores de texto, lenguajes de programación y herramientas de procesamiento de cadenas.

Las expresiones regulares sobre un alfabeto Σ se definen inductivamente:

Casos base:

- $\emptyset$ es una expresión regular que denota el lenguaje vacío $\emptyset$
- ε es una expresión regular que denota el lenguaje $\{\varepsilon\}$ (solo la cadena vacía)
- Para cada $a \in \Sigma$, a es una expresión regular que denota $\{a\}$

Casos inductivos: Si r y s son expresiones regulares, entonces:

- $(r \cup s)$ denota $L(r) \cup L(s)$ (unión)
- $(r \cdot s)$ o simplemente (rs) denota $L(r) \cdot L(s)$ (concatenación)
- (r^*) denota $(L(r))^*$ (cerradura de Kleene: cero o más repeticiones)

Ejemplo 1: Expresiones regulares para lenguajes conocidos

Ejemplo 1.1: El lenguaje de todas las cadenas binarias se describe con:

$$(0 \cup 1)^*$$

Esta expresión denota Σ^* donde $\Sigma = \{0, 1\}$.

Ejemplo 1.2: Cadenas binarias que terminan en 01 (el mismo lenguaje que se estudiará mediante autómatas en los Ejemplos 2 y 3 de este mismo capítulo, presentados más adelante):

$$(0 \cup 1)^*01$$

Ejemplo 1.3: Cadenas sobre $\{a, b\}$ con cero o más a seguidas de cero o más b:

$$a^*b^*$$

Acepta: ε, a, b, aa, ab, bb, $aaab$, etc.
Rechaza: ba, aba, baa, etc.

Ejemplo 1.4: Cadenas sobre $\{a, b\}$ que terminan en abb:

$$(a \cup b)^* abb$$

Ejemplo 1.5: Cadenas sobre $\{a, b\}$ de longitud par:

$$((a \cup b)(a \cup b))^*$$

Acepta: ε, aa, ab, ba, bb, $aabb$, $abba$, etc.
Rechaza: a, b, aab, aba, bab, etc.

Autómata Finito Determinista (AFD)

Después de definir el lenguaje mediante una expresión regular, puede construirse un autómata finito determinista (AFD) equivalente. Este autómata procesa la entrada símbolo por símbolo y determina si la cadena pertenece o no al lenguaje.

Definición formal

Un *autómata finito determinista* (AFD) es una quíntupla [Sipser, 2012]

$$M = (Q, \Sigma, \delta, q_0, F)$$

donde:

- Q es un conjunto finito no vacío de **estados**,
- Σ es un conjunto finito no vacío, el **alfabeto de entrada**,
- $\delta : Q \times \Sigma \to Q$ es la **función de transición** (total),
- $q_0 \in Q$ es el **estado inicial**,
- $F \subseteq Q$ es el conjunto de **estados de aceptación** (o estados finales).

El autómata se denomina *determinista* porque, para cada combinación de estado y símbolo de entrada, existe una única transición posible. Esto elimina cualquier ambigüedad, ya que en todo momento se sabe con exactitud a qué estado debe pasar.

Interpretación operacional

Un AFD procesa una cadena $w = a_1 a_2 \cdots a_n \in \Sigma^*$ siguiendo estos pasos:

1. Comienza en el estado inicial q_0.
2. Lee los símbolos de w de izquierda a derecha.
3. Para cada símbolo a_i leído, transita del estado actual q al estado $\delta(q, a_i)$.
4. Al terminar de leer la cadena, **acepta** si el estado alcanzado pertenece a F; en caso contrario, **rechaza**.

Función de transición extendida

Para formalizar el procesamiento de cadenas completas se define la **función de transición extendida** $\hat{\delta} : Q \times \Sigma^* \to Q$ de manera inductiva:

$$\begin{aligned} \hat{\delta}(q, \varepsilon) &= q \\ \hat{\delta}(q, wa) &= \delta(\hat{\delta}(q, w), a) \quad \text{para } w \in \Sigma^*, a \in \Sigma \end{aligned}$$

Esta definición captura el procesamiento secuencial: $\hat{\delta}(q, wa)$ se calcula procesando primero w desde q y aplicando luego el símbolo a al estado resultante. Una cadena w es aceptada por el AFD M si y solo si $\hat{\delta}(q_0, w) \in F$.

Lenguaje reconocido

El **lenguaje reconocido** (o aceptado) por un AFD M es el conjunto de todas las cadenas que el autómata acepta:

$$L(M) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

El autómata M y el lenguaje $L(M)$ son objetos distintos: M es la quíntupla que define la máquina abstracta y $L(M)$ es el conjunto de cadenas que esa máquina acepta.

Equivalencia con expresiones regulares

Teorema de Kleene [Kleene, 1956]
Un lenguaje L es regular si y solo si existe una expresión regular que lo describe.

Este resultado establece que los autómatas finitos y las expresiones regulares son formalismos equivalentes: todo lenguaje descrito por una expresión regular puede reconocerse mediante un AFD, y todo lenguaje reconocido por un AFD puede describirse con una expresión regular. La construcción explícita de un AFN a partir de una expresión regular fue formalizada posteriormente por **Ken Thompson** [Thompson, 1968] y constituye la base de los motores de expresiones regulares utilizados en la práctica. Un método alternativo, basado en *derivadas* de expresiones regulares, fue propuesto por **Brzozowski** [Brzozowski, 1964] y permite construir directamente un AFD sin pasar por un AFN intermedio.

Ejemplo 2: AFD para cadenas que terminan en 01

Consideremos el lenguaje $L = \{w \in \{0,1\}^* \mid w \text{ termina en } 01\}$.

Diseñamos un AFD $M = (Q, \Sigma, \delta, q_0, F)$ donde:

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0, 1\}$
- q_0 es el estado inicial
- $F = \{q_2\}$

La función de transición δ se define mediante la siguiente tabla:

δ	0	1
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_0

Cada celda de la tabla indica a qué estado transita el autómata al leer el símbolo correspondiente desde el estado actual. Por ejemplo, desde q_0, al leer un 0, transita a q_1; al leer un 1, permanece en q_0.

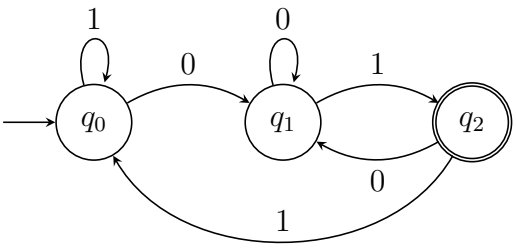
Representación mediante diagramas de transición

Los AFD suelen representarse mediante **diagramas de transición**, ya que permiten observar de forma gráfica los estados y las transiciones del autómata. Estos diagramas también facilitan el diseño y análisis del reconocimiento de cadenas.

Un diagrama de transición corresponde a un grafo dirigido donde:

- Cada **nodo** representa un estado.
- Cada **arista** dirigida representa una transición etiquetada con el símbolo que la activa.
- El **estado inicial** se indica mediante una flecha entrante sin estado de origen.
- Los **estados de aceptación** se representan con doble círculo.

El diagrama para el Ejemplo 2 es:



Interpretación de los estados:

- q_0: no se ha detectado el inicio del patrón "01" (último símbolo leído no fue 0).
- q_1: el último símbolo leído fue 0 (posible inicio del patrón).
- q_2: los últimos dos símbolos leídos fueron 01 (patrón detectado).

Verificación con una traza: Procesemos la cadena $w = 1101$:

Paso	Símbolo	Estado	Observación
0	—	q_0	Estado inicial
1	1	q_0	No es 0, permanece en q_0
2	1	q_0	No es 0, permanece en q_0
3	0	q_1	Lee 0, posible inicio de patrón
4	1	q_2	Lee 1, patrón "01" detectado

El estado final es $q_2 \in F$, por lo tanto la cadena es aceptada.

c. ¿Qué se puede reutilizar de soluciones pasadas?

Para resolver este problema es necesario conocer los conceptos del capítulo anterior: lenguajes, alfabetos, palabras y sus operaciones (concatenación, potencia, cerradura de Kleene). Sobre esa base se construye una estrategia general para validar cadenas con un formato específico.

d. ¿Cuál es el diseño de la solución?

Primero se define el alfabeto de entrada, que en este caso es el conjunto de caracteres válidos para los identificadores (letras minúsculas y dígitos). Luego, se construye una expresión regular que capture las reglas de formato especificadas. Finalmente, se diseña un AFD que implementa el reconocimiento de cadenas que cumplen con esa expresión regular, y se representa mediante un diagrama de transición para facilitar su comprensión y análisis.

Expresión regular $L_{\mathbf{id}}$

Sea $\Sigma = \{a, b, \ldots, z, 0, 1, \ldots, 9\}$, con:

- $L = \{a, b, \ldots, z\}$ (letras minúsculas)
- $D = \{0, 1, \ldots, 9\}$ (dígitos)

La expresión regular que describe los identificadores válidos (mínimo 3 caracteres, iniciando con letra) es:

$$L_{\text{id}} = L \cdot (L \cup D) \cdot (L \cup D)^{+}$$

que equivale a:

$$L_{\text{id}} = L \cdot (L \cup D)^{2} \cdot (L \cup D)^{*}$$

Autómata $M_{\mathbf{id}}$

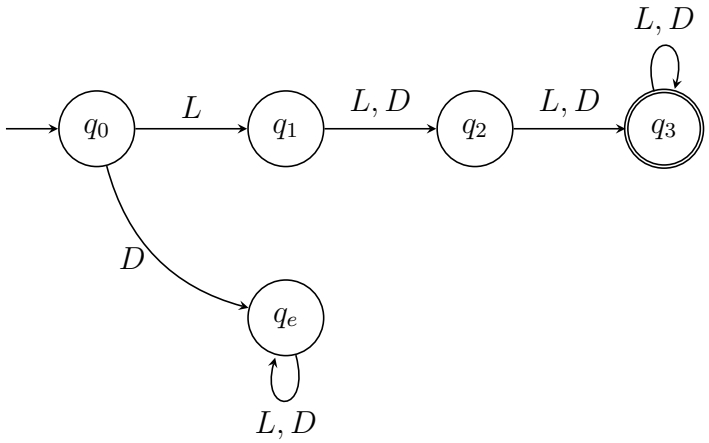
Definimos el autómata $M_{\text{id}} = (Q, \Sigma, \delta, q_0, F)$ donde:

- $Q = \{q_0, q_1, q_2, q_3, q_e\}$
- $\Sigma = \{L, D\}$ (representando letras y dígitos)
- q_0 es el estado inicial
- $F = \{q_3\}$ (identificadores válidos requieren al menos 3 caracteres)
- q_e es el estado de error (trampa)

La función $\delta : Q \times \Sigma \to Q$ se define mediante la siguiente tabla:

δ	L (letra)	D (dígito)
q_0	q_1	q_e
q_1	q_2	q_2
q_2	q_3	q_3
q_3	q_3	q_3
q_e	q_e	q_e

Diagrama de transición



Interpretación de los estados

- q_0: estado inicial, no se ha leído ningún símbolo
- q_1: se ha leído 1 letra (identificador válido requiere al menos 3 caracteres)
- q_2: se han leído 2 caracteres válidos (letra + letra/dígito)
- q_3: identificador válido (3 o más caracteres, comenzando con letra)
- q_e: error, la cadena comenzó con dígito (rechazado permanentemente)

2.1.3 Ejemplos de pruebas

a. Ejemplo 1

Cadena $w =$ “abc” (aceptada)

Paso	**Símbolo**	**Estado**
0	—	q_0
1	a (L)	q_1
2	b (L)	q_2
3	c (L)	q_3

Estado final: $q_3 \in F \Rightarrow$ **ACEPTADA**

b. Ejemplo 2

Cadena $w =$ “a2” (rechazada)

Paso	**Símbolo**	**Estado**
0	—	q_0
1	a (L)	q_1
2	2 (D)	q_2

Estado final: $q_2 \notin F \Rightarrow$ **RECHAZADA** (longitud insuficiente)

c. Ejemplo 3

Cadena $w =$ “2abc” (rechazada)

Paso	**Símbolo**	**Estado**
0	—	q_0
1	2 (D)	q_e
2	a (L)	q_e
3	b (L)	q_e
4	c (L)	q_e

Estado final: $q_e \notin F \Rightarrow$ **RECHAZADA** (comienza con dígito)

2.1.4 Codificación

Las representaciones del autómata (la quíntupla formal y el diagrama de transición) admiten una traducción directa a código ejecutable. Esta sección muestra esa traducción en Python.

a. Algoritmo general de reconocimiento

La siguiente función simula el funcionamiento de un AFD. A partir del estado inicial, la cadena se recorre símbolo por símbolo aplicando la función de transición, y al finalizar se verifica si el estado alcanzado pertenece al conjunto de estados de aceptación.

Código 2.1: **Función de clasificación**

```
1 def reconocer(cadena, estado_inicial, estados_finales, transicion):
2     estado = estado_inicial
3     for simbolo in cadena:
4         estado = transicion.get((estado, simbolo), "error")
5     return estado in estados_finales
```

El algoritmo es determinístico (cada transición es única) y se ejecuta en tiempo lineal respecto a la longitud de la cadena de entrada. El espacio requerido es constante, dado que solo se mantiene el estado actual durante el procesamiento.

Representación de la función de transición

La función de transición δ puede implementarse de diferentes maneras. Una de las más comunes es utilizar una **tabla bidimensional** indexada por estado y símbolo. Esta representación permite consultar transiciones rápidamente, aunque puede desperdiciar memoria cuando existen muchas combinaciones sin transición definida.

Otra alternativa consiste en usar un **diccionario** o **mapa hash** sobre pares (q, a). Esta opción suele ser más conveniente cuando el autómata tiene pocas transiciones respecto al número total de combinaciones posibles.

También es posible implementar la función mediante **sentencias condicionales**. Esta solución puede resultar sencilla en autómatas pequeños, pero a medida que aumenta el número de estados el código se vuelve más difícil de mantener.

b. Ejemplo de implementación para solución del problema

La solución se organiza en cinco componentes. La función `main()` se encarga de leer la cadena ingresada por el usuario, construir la función de transición, realizar la validación y mostrar el resultado.

Código 2.2: Función principal

```
def main():
    cadena = leer("Ingrese el identificador: ")
    transicion = definir_funcion_transicion()
    resultado = validar_identificador(cadena, transicion)
    imprimir_resultado(cadena, resultado)
```

Dos funciones auxiliares se ocupan de la consola: una lee el identificador, la otra reporta si fue aceptado.

Código 2.3: Funciones de entrada y salida

```
def leer(texto):
    return input(texto)

def imprimir_resultado(cadena, es_valido):
    if es_valido:
        print(f"'{cadena}': ACEPTADA")
    else:
        print(f"'{cadena}': RECHAZADA")
```

Antes de ejecutar el autómata es necesario verificar que cada símbolo de la cadena pertenezca al alfabeto $\Sigma = L \cup D$. Esa verificación corresponde a la función de clasificación.

Código 2.4: Función de clasificación

```
def clasificar(caracter):
    if caracter.isalpha() and caracter.islower():
        return "L"
    elif caracter.isdigit():
        return "D"
    else:
        return "error"
```

La validación inicia en q_0 y procesa la cadena símbolo por símbolo utilizando la función de transición δ. La cadena es aceptada cuando el estado final pertenece al conjunto F.

Código 2.5: Función de validación

```
def validar_identificador(cadena, transicion):
    estado = "q0"
    for caracter in cadena:
        simbolo = clasificar(caracter)
        estado = transicion.get((estado, simbolo), "qe")
    return estado == "q3"
```

Por último, δ se codifica como un diccionario anidado: cada estado se asocia a otro diccionario que mapea símbolos a estados destino.

Código 2.6: Función de transición

```
def definir_funcion_transicion():
    return {
        ("q0", "L"): "q1", ("q0", "D"): "qe",
        ("q1", "L"): "q2", ("q1", "D"): "q2",
        ("q2", "L"): "q3", ("q2", "D"): "q3",
        ("q3", "L"): "q3", ("q3", "D"): "q3",
        ("qe", "L"): "qe", ("qe", "D"): "qe",
    }
```

La ejecución del programa se inicia llamando a `main()` al final del archivo.

2.2. Reconocimiento de números decimales

FinData Analytics es una startup fintech que desarrolla herramientas de análisis financiero en tiempo real. Su plataforma recibe flujos continuos de datos numéricos provenientes de diversas fuentes: mercados bursátiles, sensores industriales y reportes contables automatizados. Un requisito crítico del sistema es la validación correcta de los valores numéricos recibidos, ya que un dato mal formateado puede provocar errores en los cálculos financieros y generar pérdidas significativas.

El equipo de ingeniería ha detectado que los datos numéricos pueden llegar en distintos formatos: números enteros como `123`, números decimales como `45.67`, o cadenas inválidas como `.123` o `12.34.56`. Se necesita un módulo de validación que determine de forma automática y eficiente si cada valor recibido corresponde a un número decimal válido.

2.2.1 Análisis / Abstracción

El requerimiento de FinData Analytics puede formalizarse como un problema de reconocimiento de cadenas numéricas:

a. ¿Cuál es el problema a resolver?

FinData Analytics requiere un módulo que determine si los valores recibidos en sus flujos de datos corresponden a números decimales válidos. La falta de esta validación puede producir errores en los cálculos financieros.

b. ¿Cuál es la información necesaria?

Cada valor del flujo de datos llega al módulo de validación como una cadena independiente:

Información necesaria	**Tipo de dato**
Cadena de entrada (valor numérico)	Texto (String)

Un número válido es, o bien una secuencia no vacía de dígitos (entero), o bien dígitos seguidos de un punto y más dígitos (decimal). Cadenas como `.123` o `12.34.56` quedan fuera de esa definición.

2.2.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **ENTRADA:** Una cadena de caracteres que representa un posible valor numérico.
- **PROCESAMIENTO:**
  1. Lectura de la cadena de entrada.
  2. Definición de la función de transición del AFD que reconoce números decimales válidos.
  3. Validación del número decimal mediante el AFD.
  4. Impresión del resultado de la validación.

- **SALIDA:** Una decisión binaria: `ACEPTADA` si la cadena es un número decimal válido, o `RECHAZADA` si no.

b. ¿Qué conocimiento adicional se requiere?

La solución utiliza los conceptos introducidos previamente, como las expresiones regulares y los AFD, e incorpora además el **autómata finito no determinista** (AFN) y la técnica de conversión de AFN a AFD.

El AFN permite diseños más simples y conceptuales, pero su ejecución directa resulta menos eficiente en la práctica. Por esta razón, la conversión a AFD produce el modelo determinista encargado de realizar la validación.

Autómata Finito No Determinista (AFN)

Definición formal

Un *autómata finito no determinista* (AFN) es una quíntupla

$$N = (Q, \Sigma, \delta, q_0, F)$$

donde los componentes son similares al AFD, excepto que la función de transición tiene la forma:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$$

donde $\mathcal{P}(Q)$ denota el conjunto potencia de Q.

Características del no determinismo

Un AFN se diferencia de un AFD en dos aspectos:

1. Múltiples transiciones: desde un estado, con un mismo símbolo, puede haber cero, una o más transiciones posibles.
2. Transiciones epsilon (ε): el autómata puede cambiar de estado sin consumir ningún símbolo de entrada.

Criterio de aceptación

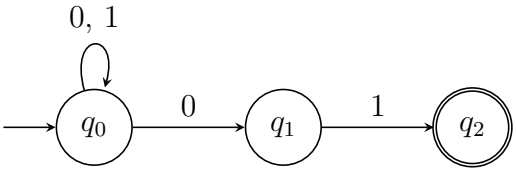
Una cadena w es aceptada por un AFN si existe al menos una secuencia de transiciones que, partiendo del estado inicial y procesando todos los símbolos de w, finalice en un estado de aceptación.

Este criterio es más flexible que el del AFD, ya que basta con encontrar un recorrido válido que acepte la cadena, incluso si otros recorridos terminan en rechazo.

Ejemplo 3: AFN para cadenas que terminan en 01

Retomemos el lenguaje $L = \{w \in \{0,1\}^* \mid w \text{ termina en } 01\}$ del Ejemplo 2.

El AFN equivalente se diseña a partir de la idea de *adivinar* cuándo comienza el patrón "01" final.



Interpretación:

- q_0: estado inicial, procesa cualquier símbolo sin avanzar en el patrón.
- Desde q_0, al leer 0, existen **dos opciones**:
  - permanecer en q_0 (ignorar este 0), o
  - transitar a q_1 (asumir que este 0 inicia el patrón final).
- q_1: se ha leído un 0 que podría ser el inicio del patrón final.
- q_2: se completó la secuencia "01".

Comparación con el AFD del Ejemplo 2:

- El AFN tiene 3 estados (igual que el AFD), pero es conceptualmente más simple.

- El AFN *adivina* no determinísticamente cuándo empieza el patrón final.
- El AFD debe *recordar* explícitamente el último símbolo leído.
- Ambos reconocen el mismo lenguaje L.

Equivalencia entre AFD y AFN

A pesar de la aparente mayor expresividad del AFN, ambos modelos reconocen la misma clase de lenguajes.

Teorema (Equivalencia AFD-AFN) [Rabin and Scott, 1959]**:** Para todo AFN N, existe un AFD M tal que $L(N) = L(M)$.

La demostración constructiva se basa en la construcción de subconjuntos [Hopcroft et al., 2006]: cada estado del AFD representa un conjunto de estados en los que podría estar el AFN. Esto permite usar AFN en la fase de diseño (más simples de construir) y convertirlos a AFD para la implementación (más eficientes de ejecutar).

Algoritmo de construcción de subconjuntos

Dado un AFN $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$, se construye un AFD $M = (Q_M, \Sigma, \delta_M, q_{0M}, F_M)$ donde:

- $Q_M = \mathcal{P}(Q_N)$ (el conjunto potencia de Q_N)
- $q_{0M} = \text{clausura-}\varepsilon(\{q_0\})$
- $F_M = \{S \subseteq Q_N \mid S \cap F_N \neq \emptyset\}$
- $\delta_M(S, a) = \bigcup_{q \in S} \text{clausura-}\varepsilon(\delta_N(q, a))$ para cada $S \in Q_M$ y $a \in \Sigma$

La clausura-ε de un conjunto S es el conjunto de estados alcanzables desde S usando solo transiciones ε.

Ejemplo 4: Conversión del AFN a AFD (cadenas que terminan en 01)

En este ejemplo se aplica la construcción de subconjuntos al AFN presentado en el Ejemplo 3, correspondiente al lenguaje de cadenas que terminan en "01". Como el autómata no posee transiciones ε, la clausura-ε de cualquier conjunto coincide con el mismo conjunto.

Paso 1: El AFD comienza en $\{q_0\}$.

Paso 2: Transiciones desde $\{q_0\}$:

- Con 0: desde q_0 se puede ir a q_0 o $q_1 \Rightarrow \delta_M(\{q_0\}, 0) = \{q_0, q_1\}$
- Con 1: desde q_0 solo se va a $q_0 \Rightarrow \delta_M(\{q_0\}, 1) = \{q_0\}$

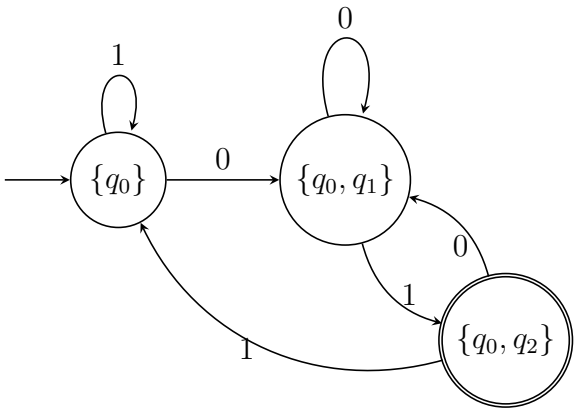
Paso 3: Transiciones desde $\{q_0, q_1\}$:

- Con 0: $\delta_M(\{q_0, q_1\}, 0) = \{q_0, q_1\}$ (el mismo estado)
- Con 1: desde q_0 se va a q_0; desde q_1 se va a $q_2 \Rightarrow \delta_M(\{q_0, q_1\}, 1) = \{q_0, q_2\}$

Paso 4: Transiciones desde $\{q_0, q_2\}$:

- Con 0: $\delta_M(\{q_0, q_2\}, 0) = \{q_0, q_1\}$ (ya existe)
- Con 1: $\delta_M(\{q_0, q_2\}, 1) = \{q_0\}$ (ya existe)

$F_M = \{\{q_0, q_2\}\}$ pues contiene a $q_2 \in F_N$. El AFD resultante:



c. ¿Qué se puede reutilizar de soluciones pasadas?

Para abordar este problema se requiere el conocimiento del capítulo anterior sobre lenguajes, alfabetos y operaciones de cadenas. Del problema anterior se reutilizan el patrón de diseño de autómatas con estados de error y el algoritmo general de reconocimiento.

d. ¿Cuál es el diseño de la solución?

Primero se define el alfabeto de entrada (dígitos y punto decimal), se construye la expresión regular que formaliza el patrón válido, se diseña un AFN que captura la estructura del lenguaje y finalmente se convierte a AFD mediante construcción de subconjuntos.

Expresión regular L_{num}

Sea $D = \{0, 1, \ldots, 9\}$ el conjunto de dígitos y $\{.\}$ el punto decimal. La expresión regular que describe los números válidos (enteros o con parte decimal) es:

$$L_{\text{num}} = D^+ \cup D^+ \cdot \{.\} \cdot D^+$$

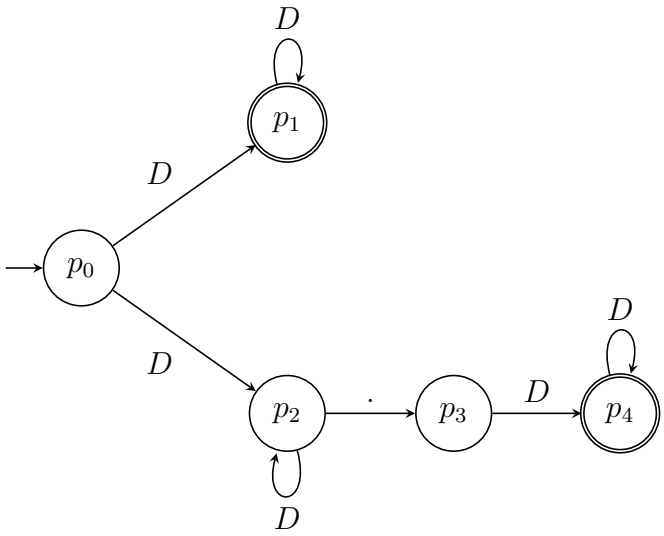
AFN de diseño para L_{num}

Después de leer el primer dígito, el autómata todavía no puede determinar si la cadena corresponde a un número entero (D^+) o a un número decimal ($D^+ \cdot \{.\} \cdot D^+$). Esta incertidumbre se modela mediante **no-determinismo**: desde el estado inicial, la lectura de un dígito permite continuar por dos posibles recorridos.

Se define el AFN $N = (P, \Sigma, \Delta, p_0, F_N)$ con $P = \{p_0, p_1, p_2, p_3, p_4\}$, $F_N = \{p_1, p_4\}$ y función de transición $\Delta : P \times \Sigma \to \mathcal{P}(P)$:

Δ	D (dígito)	. (punto)
p_0	$\{p_1, p_2\}$	$\emptyset$
p_1	$\{p_1\}$	$\emptyset$
p_2	$\{p_2\}$	$\{p_3\}$
p_3	$\{p_4\}$	$\emptyset$
p_4	$\{p_4\}$	$\emptyset$

La transición $\Delta(p_0, D) = \{p_1, p_2\}$ es el único punto no-determinista: p_1 inicia la rama de enteros (es de aceptación) y p_2 inicia la rama de decimales.



La conversión AFN → AFD mediante construcción de subconjuntos produce los siguientes estados (cada uno es un subconjunto de estados del AFN):

Estado AFD	**Subconjunto AFN**	D	.	**¿Acepta?**
q_0	$\{p_0\}$	q_1	q_e	No
q_1	$\{p_1, p_2\}$	q_1	q_2	Sí ($p_1 \in F_N$)
q_2	$\{p_3\}$	q_3	q_e	No
q_3	$\{p_4\}$	q_3	q_e	Sí ($p_4 \in F_N$)
q_e	$\emptyset$	q_e	q_e	No

Autómata M_{num}

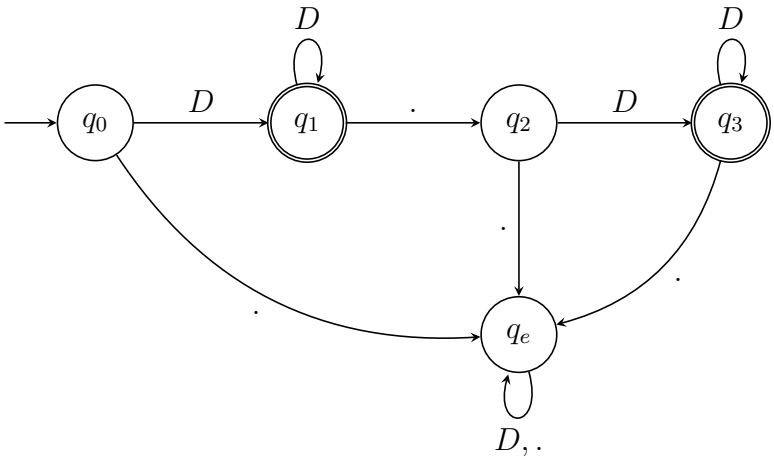
El AFD resultante de convertir el AFN anterior es $M_{\text{num}} = (Q, \Sigma, \delta, q_0, F)$ donde:

- $Q = \{q_0, q_1, q_2, q_3, q_e\}$
- $\Sigma = \{D, .\}$ (dígitos y punto decimal)
- q_0 es el estado inicial
- $F = \{q_1, q_3\}$ (enteros y decimales válidos)
- q_e es el estado de error

La función $\delta : Q \times \Sigma \to Q$ se define mediante la siguiente tabla:

δ	D (dígito)	. (punto)
q_0	q_1	q_e
q_1	q_1	q_2
q_2	q_3	q_e
q_3	q_3	q_e
q_e	q_e	q_e

Diagrama de transición



Interpretación de los estados

- q_0: estado inicial, no se ha leído ningún símbolo
- q_1: número entero válido (uno o más dígitos)
- q_2: punto decimal leído, esperando dígitos de la parte decimal
- q_3: número decimal válido (parte entera + punto + parte decimal)
- q_e: formato inválido (punto al inicio o múltiples puntos)

2.2.3 Ejemplos de pruebas

a. Ejemplo 1

Cadena $w =$ “123” (aceptada - entero)

Paso	**Símbolo**	**Estado**
0	—	q_0
1	1 (D)	q_1
2	2 (D)	q_1
3	3 (D)	q_1

Estado final: $q_1 \in F \Rightarrow$ **ACEPTADA**

b. Ejemplo 2

Cadena $w =$ “45.67” (aceptada - decimal)

Paso	**Símbolo**	**Estado**
0	—	q_0
1	4 (D)	q_1
2	5 (D)	q_1
3	.	q_2
4	6 (D)	q_3
5	7 (D)	q_3

Estado final: $q_3 \in F \Rightarrow$ **ACEPTADA**

c. Ejemplo 3

Cadena $w =$ “.123” (rechazada - comienza con punto)

Paso	**Símbolo**	**Estado**
0	—	q_0
1	.	q_e
2	1 (D)	q_e
3	2 (D)	q_e
4	3 (D)	q_e

Estado final: $q_e \notin F \Rightarrow$ **RECHAZADA**

2.2.4 Codificación

La implementación reutiliza los cinco componentes construidos en la sección anterior, ajustando los detalles propios del nuevo autómata: alfabeto $\{D, .\}$ y dos estados de aceptación, q_1 para enteros y q_3 para decimales.

El flujo comienza en la función `main`, que pide el valor, construye la tabla de transiciones y resuelve si la cadena es aceptada.

Código 2.7: Función principal

```
1 def main():
2     cadena = leer("Ingrese el numero decimal: ")
3     transicion = definir_funcion_transicion()
4     resultado = validar_decimal(cadena, transicion)
5     imprimir_resultado(cadena, resultado)
```

La interacción con la consola queda en dos funciones auxiliares: una para leer el dato, otra para imprimir el resultado.

Código 2.8: Funciones de entrada y salida

```
7 def leer(texto):
8     return input(texto)
9
10 def imprimir_resultado(cadena, es_valido):
11     if es_valido:
12         print(f"'{cadena}': ACEPTADA")
13     else:
14         print(f"'{cadena}': RECHAZADA")
```

Antes de invocar al autómata hay que reducir cada carácter de la cadena a uno de los dos símbolos del alfabeto, dígito o punto.

Código 2.9: Función de clasificación

```
16 def clasificar(caracter):
17     if caracter.isdigit():
18         return "D"
19     elif caracter == ".":
20         return "."
21     else:
22         return "error"
```

A diferencia del problema anterior, aquí hay dos rutas de aceptación: la cadena se acepta si termina en q_1 (entero) o en q_3 (decimal).

Código 2.10: Función de validación

```
def validar_decimal(cadena, transicion):
    estado = "q0"
    for caracter in cadena:
        simbolo = clasificar(caracter)
        estado = transicion.get((estado, simbolo), "qe")
    return estado in {"q1", "q3"}
```

Por último, δ se representa como un diccionario anidado de cinco entradas, una por estado, en el que cualquier transición no permitida conduce a q_e.

Código 2.11: Función de transición

```
def definir_funcion_transicion():
    return {
        ("q0", "D"): "q1", ("q0", "."): "qe",
        ("q1", "D"): "q1", ("q1", "."): "q2",
        ("q2", "D"): "q3", ("q2", "."): "qe",
        ("q3", "D"): "q3", ("q3", "."): "qe",
        ("qe", "D"): "qe", ("qe", "."): "qe",
    }
```

2.3. Validación de códigos de producto

LogiTrack Internacional es una empresa de logística que gestiona el inventario y distribución de productos para múltiples clientes corporativos en América Latina. Cada producto registrado en su sistema recibe un código único con un formato estricto: exactamente dos letras mayúsculas que identifican la categoría del producto, seguidas de un guión y exactamente cuatro dígitos que corresponden al número de serie dentro de esa categoría. Por ejemplo, el código `AB-1234` indica un producto de la categoría AB con número de serie 1234.

El departamento de tecnología de LogiTrack ha identificado que frecuentemente se ingresan códigos con formatos incorrectos: letras minúsculas, dígitos faltantes, guiones en posiciones incorrectas o caracteres adicionales. Estos errores generan inconsistencias en la base de datos y dificultan el seguimiento de los productos a lo largo de la cadena de distribución.

Se requiere un módulo de validación que determine automáticamente si un código de producto ingresado cumple con el formato establecido, garantizando así la integridad de los datos y facilitando el seguimiento eficiente de los productos. Adicionalmente, el área de sistemas ha informado que los códigos cuya categoría comience exactamente con las letras `XX` están reservados para uso interno del sistema: aunque tienen el formato correcto, *no deben aceptarse* como códigos de producto ordinarios.

2.3.1 Análisis / Abstracción

El escenario de LogiTrack combina una validación de formato con una restricción adicional sobre las categorías reservadas:

a. ¿Cuál es el problema a resolver?

LogiTrack Internacional requiere un validador automático que acepte los códigos con formato `LL-DDDD` (dos letras mayúsculas, guión, cuatro dígitos) y, simultáneamente, rechace aquellos cuya categoría sea `XX`, reservada para uso interno del sistema.

b. ¿Cuál es la información necesaria?

Cada producto registrado se identifica con un único código que se valida individualmente:

Información necesaria	**Tipo de dato**
Cadena de entrada (código de producto)	Texto (String)

El formato válido especifica exactamente dos letras mayúsculas, un guión y exactamente cuatro dígitos, como en `AB-1234`.

2.3.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **ENTRADA:** Una cadena de caracteres proporcionada como código de producto.

- **PROCESAMIENTO:**

  1. Lectura de la cadena de entrada.
  2. Definición de la función de transición del AFD que reconoce el formato `LL-DDDD`.
  3. Validación del código de producto mediante el AFD.
  4. Impresión del resultado de la validación.

- **SALIDA:** Una decisión binaria: `ACEPTADA` si la cadena cumple exactamente con el formato, o `RECHAZADA` si no.

b. ¿Qué conocimiento adicional se requiere?

Para diseñar la solución se aplican los formalismos ya presentados (expresiones regulares y AFD) y, adicionalmente, las **propiedades de clausura** de los lenguajes regulares. Estas propiedades permiten una construcción modular: se define primero el lenguaje de todos los códigos con formato correcto, luego el lenguaje de los códigos reservados, y se obtiene el lenguaje válido mediante complemento e intersección.

Propiedades de clausura

Los lenguajes regulares forman una familia *cerrada* bajo varias operaciones: aplicar estas operaciones a lenguajes regulares siempre produce otro lenguaje regular. Esta propiedad permite construir lenguajes complejos a partir de componentes más simples.

Teorema de clausura [Hopcroft et al., 2006, Sipser, 2012]**:** Si L_1 y L_2 son lenguajes regulares, entonces también lo son:

- $L_1 \cup L_2$ (unión)
- $L_1 \cap L_2$ (intersección)
- $L_1 \cdot L_2$ (concatenación)
- L_1^* (cerradura de Kleene)
- $\overline{L_1}$ (complemento: todas las cadenas que no están en L_1)
- $L_1 - L_2$ (diferencia: cadenas en L_1 pero no en L_2)

Las tres primeras operaciones son los constructores con los que se definen las expresiones regulares; el teorema garantiza que aplicarlas a lenguajes regulares produce un lenguaje regular.

Las operaciones restantes van más allá de la sintaxis de las expresiones regulares. La intersección requiere construir un AFD producto que simula ambos autómatas en paralelo. El complemento, en un AFD, se obtiene intercambiando estados de aceptación y rechazo; esta operación no funciona directamente sobre un AFN, que debe convertirse antes a AFD. La diferencia se expresa como $L_1 - L_2 = L_1 \cap \overline{L_2}$.

Aplicación al problema de LogiTrack

El problema requiere aceptar exactamente los códigos con formato correcto que no comiencen con `XX`. Esto se expresa mediante dos lenguajes regulares:

- $L_{\text{codigo}} = L^2 \cdot \{-\} \cdot D^4$: todos los códigos con formato válido (regular por clausura bajo concatenación).
- $L_{\text{reservado}} = \{X\} \cdot \{X\} \cdot \{-\} \cdot D^4$: los códigos reservados del sistema (también regular, por el mismo argumento).

El lenguaje objetivo se obtiene combinando ambos mediante complemento e intersección:

$$L_{\text{valido}} = L_{\text{codigo}} \cap \overline{L_{\text{reservado}}}$$

Por las propiedades de clausura, $\overline{L_{\text{reservado}}}$ es regular (complemento de un lenguaje regular) y, por lo tanto, L_{valido} también lo es (intersección de dos lenguajes regulares). Esto garantiza que existe un AFD que implementa la validación completa, incluyendo el rechazo de los códigos reservados.

c. ¿Qué se puede reutilizar de soluciones pasadas?

Para abordar este problema se requiere el conocimiento del capítulo anterior sobre lenguajes, alfabetos y operaciones de cadenas. De los problemas anteriores se reutilizan el patrón de diseño de autómatas con estados de error, el algoritmo general de reconocimiento y el procedimiento de conversión AFN $\rightarrow$ AFD.

d. ¿Cuál es el diseño de la solución?

La solución parte de la definición del alfabeto de entrada (letras mayúsculas, dígitos y guión) y de los lenguajes correspondientes a los códigos válidos y reservados. Posteriormente, utilizando las propiedades de clausura de los lenguajes regulares bajo complemento e intersección, se construye el lenguaje que contiene únicamente los códigos permitidos. Finalmente, dicho lenguaje se implementa mediante un AFD encargado de reconocer las cadenas válidas.

Expresiones regulares del problema

Sea $L = \{A, B, \ldots, Z\}$ (letras mayúsculas) y $D = \{0, 1, \ldots, 9\}$ (dígitos). Se definen tres lenguajes regulares:

$$L_{\text{codigo}} = L^2 \cdot \{-\} \cdot D^4$$
$$L_{\text{reservado}} = \{X\} \cdot \{X\} \cdot \{-\} \cdot D^4$$
$$L_{\text{valido}} = L_{\text{codigo}} \cap \overline{L_{\text{reservado}}}$$

L_{codigo} y $L_{\text{reservado}}$ son regulares por clausura bajo concatenación. $\overline{L_{\text{reservado}}}$ es regular por clausura bajo complemento, y L_{valido} es regular por clausura bajo intersección. Esto garantiza que existe un AFD que reconoce exactamente los códigos con formato correcto que no están reservados.

Autómata M_{codigo}

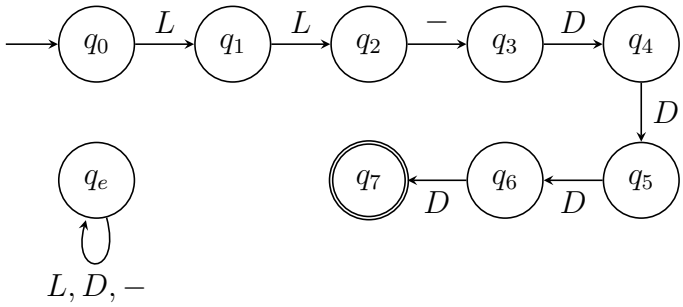
Definimos el autómata $M_{\text{codigo}} = (Q, \Sigma, \delta, q_0, F)$ donde:

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_e\}$
- $\Sigma = \{L, D, -\}$ (letras mayúsculas, dígitos, guión)
- q_0 es el estado inicial
- $F = \{q_7\}$ (código completo y válido)
- q_e es el estado de error (trampa)

La función $\delta : Q \times \Sigma \to Q$ se define mediante la siguiente tabla:

δ	L (letra)	D (dígito)	$-$ (guión)
q_0	q_1	q_e	q_e
q_1	q_2	q_e	q_e
q_2	q_e	q_e	q_3
q_3	q_e	q_4	q_e
q_4	q_e	q_5	q_e
q_5	q_e	q_6	q_e
q_6	q_e	q_7	q_e
q_7	q_e	q_e	q_e
q_e	q_e	q_e	q_e

Diagrama de transición



Se omiten las transiciones hacia q_e para mayor claridad.

Interpretación de los estados

- q_0: estado inicial, no se ha leído ningún símbolo
- q_1: primera letra mayúscula leída
- q_2: segunda letra mayúscula leída, esperando guión
- q_3: guión leído, esperando primer dígito
- q_4: primer dígito leído
- q_5: segundo dígito leído
- q_6: tercer dígito leído
- q_7: código completo y válido (exactamente cuatro dígitos)
- q_e: formato inválido (carácter inesperado o longitud incorrecta)

2.3.3 Ejemplos de pruebas

a. Ejemplo 1

Cadena $w =$ "AB-1234" (aceptada)

Paso	Símbolo	Estado
0	—	q_0
1	A (L)	q_1
2	B (L)	q_2
3	$-$	q_3
4	1 (D)	q_4
5	2 (D)	q_5
6	3 (D)	q_6
7	4 (D)	q_7

Estado final: $q_7 \in F \Rightarrow$ **ACEPTADA**

b. Ejemplo 2

Cadena $w =$ "AB-123" (rechazada - faltan dígitos)

Paso	Símbolo	Estado
0	—	q_0
1	A (L)	q_1
2	B (L)	q_2
3	$-$	q_3
4	1 (D)	q_4
5	2 (D)	q_5
6	3 (D)	q_6

Estado final: $q_6 \notin F \Rightarrow$ **RECHAZADA**

c. Ejemplo 3

Cadena $w =$ "XX-1234" (rechazada — código reservado)

M_{codigo} verifica únicamente el formato; como `XX-1234` tiene el formato `LL-DDDD` correcto, el autómata lo **acepta**:

Paso	Símbolo	Estado
0	—	q_0
1	X (L)	q_1
2	X (L)	q_2
3	$-$	q_3
4	1 (D)	q_4
5	2 (D)	q_5
6	3 (D)	q_6
7	4 (D)	q_7

Estado final: $q_7 \in F \Rightarrow M_{\text{codigo}}$ **ACEPTA** (formato correcto).

Sin embargo, la cadena pertenece a $L_{\text{reservado}} = \{X\} \cdot \{X\} \cdot \{-\} \cdot D^4$, por lo que $w \notin L_{\text{valido}} = L_{\text{codigo}} \cap \overline{L_{\text{reservado}}} \Rightarrow$ **RECHAZADA** como código ordinario.

2.3.4 Codificación

La implementación conserva la estructura de cinco componentes, aunque el alfabeto cambia a $\{L, D, -\}$ y el conjunto de estados de aceptación se reduce a $F = \{q_7\}$.

La función `main` mantiene el mismo flujo general de las secciones anteriores: leer el código ingresado, ejecutar el autómata y mostrar el resultado de la validación.

Código 2.12: Función principal

```
def main():
    cadena = leer("Ingrese el codigo de producto: ")
    transicion = definir_funcion_transicion()
    resultado = validar_codigo(cadena, transicion)
    imprimir_resultado(cadena, resultado)
```

La interacción con la consola se delega en dos funciones auxiliares: una lee el código, la otra lo reporta como aceptado o rechazado.

Código 2.13: Funciones de entrada y salida

```
def leer(texto):
    return input(texto)

def imprimir_resultado(cadena, es_valido):
    if es_valido:
        print(f"'{cadena}': ACEPTADA")
    else:
        print(f"'{cadena}': RECHAZADA")
```

Este problema añade complejidad al alfabeto: la clasificación distingue ahora tres tipos de símbolo (letra mayúscula, dígito o guión) en vez de los dos del ejercicio anterior.

Código 2.14: Función de clasificación

```
def clasificar(caracter):
    if caracter.isalpha() and caracter.isupper():
        return "L"
    elif caracter.isdigit():
        return "D"
    elif caracter == "-":
        return "-"
    else:
        return "error"
```

La validación es estricta: solo acepta cuando el autómata alcanza q_7, es decir, cuando se ha leído el patrón completo `LL-DDDD`.

Código 2.15: Función de validación

```
def validar_codigo(cadena, transicion):
    estado = "q0"
    for caracter in cadena:
        simbolo = clasificar(caracter)
        estado = transicion.get((estado, simbolo), "qe")
    return estado == "q7"
```

Por su parte, la tabla de transiciones cubre los nueve estados, encaminando hacia q_e cualquier desviación del patrón esperado.

Código 2.16: Función de transición

```
def definir_funcion_transicion():
    return {
        ("q0", "L"): "q1", ("q0", "D"): "qe", ("q0", "-"): "qe",
        ("q1", "L"): "q2", ("q1", "D"): "qe", ("q1", "-"): "qe",
        ("q2", "L"): "qe", ("q2", "D"): "qe", ("q2", "-"): "q3",
        ("q3", "L"): "qe", ("q3", "D"): "q4", ("q3", "-"): "qe",
        ("q4", "L"): "qe", ("q4", "D"): "q5", ("q4", "-"): "qe",
        ("q5", "L"): "qe", ("q5", "D"): "q6", ("q5", "-"): "qe",
        ("q6", "L"): "qe", ("q6", "D"): "q7", ("q6", "-"): "qe",
        ("q7", "L"): "qe", ("q7", "D"): "qe", ("q7", "-"): "qe",
        ("qe", "L"): "qe", ("qe", "D"): "qe", ("qe", "-"): "qe",
    }
```

El código implementa únicamente la validación del formato `LL-DDDD`. El rechazo de los códigos reservados requiere un paso adicional: verificar la pertenencia a $L_{\text{reservado}}$ y aplicar la diferencia $L_{\text{valido}} = L_{\text{codigo}} \setminus L_{\text{reservado}}$ definida en la sección de diseño.

2.4. Análisis léxico: reconocimiento de tokens

El grupo de investigación CompLab, adscrito al departamento de ciencias de la computación de una universidad, se encuentra desarrollando un compilador educativo para un subconjunto del lenguaje C. El objetivo del proyecto es crear una herramienta didáctica que permita a los estudiantes comprender las fases de compilación mediante la experimentación directa con código fuente.

La primera fase que el equipo debe implementar es el **analizador léxico** (también llamado *lexer* o *scanner*) [Aho et al., 2006], cuya función es transformar una secuencia de caracteres (el código fuente) en una secuencia de **tokens** o unidades léxicas con significado en el lenguaje de programación. Un token es la unidad mínima con significado sintáctico; por ejemplo, en la sentencia `if x == 42 + y;`, el analizador léxico debe identificar: la palabra clave `if`, el identificador `x`, el operador ==, el literal numérico `42`, el operador `+`, el identificador `y` y el delimitador `;`.

El lexer debe reconocer y clasificar múltiples tipos de tokens: palabras clave (`if`, `while`, `return`), identificadores definidos por el usuario, números enteros, operadores aritméticos y relacionales, y delimitadores. Distinguir la palabra clave `if` de un identificador como `iffy`, o el operador de asignación = del de comparación ==, es fundamental para el correcto análisis sintáctico posterior.

2.4.1 Análisis / Abstracción

El proyecto de CompLab plantea un problema de clasificación de subcadenas:

a. ¿Cuál es el problema a resolver?

CompLab requiere implementar el analizador léxico de su compilador educativo. El lexer procesa la cadena de código fuente y asigna a cada fragmento (lexema) su tipo de token correspondiente. La eficiencia exigida motiva el uso de autómatas minimizados.

b. ¿Cuál es la información necesaria?

Cada token reconocible está asociado a un patrón. Los cinco tipos considerados son:

Tipo de token	Ejemplos	Tipo de dato
Palabra clave (KEYWORD)	`if`, `while`, `return`	Texto (String)
Identificador (IDENTIFIER)	`x`, `suma`, `total1`	Texto (String)
Número entero (INTEGER)	`42`, `0`, `100`	Texto (String)
Operador (OPERATOR)	`+`, `-`, `=`, `==`	Texto (String)
Delimitador (DELIMITER)	`(`, `)`, `;`	Texto (String)

2.4.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **ENTRADA:** Una cadena de texto con código fuente (por ejemplo, `"if x == 42 + y;"`).
- **PROCESAMIENTO:**
  1. Lectura de la cadena de código fuente.
  2. Tokenización de la cadena en una lista de lexemas con su tipo de token.
  3. Impresión de la lista de tokens reconocidos.

- **SALIDA:** Lista de pares (lexema, tipo_token). Ejemplo: `[("if", KEYWORD), ("x", IDENTIFIER), ("==", OPERATOR), ("42", INTEGER), ("+", OPERATOR), ("y", IDENTIFIER), (";", DELIMITER)]`.

b. ¿Qué conocimiento adicional se requiere?

La construcción del analizador léxico requiere expresiones regulares para definir los patrones de cada tipo de token, además de la construcción y minimización de autómatas finitos para implementar su reconocimiento. Estos conceptos se han desarrollado a lo largo del capítulo, pero presentan también limitaciones que conviene caracterizar antes de diseñar el lexer.

Limitaciones: El lema de bombeo

No todos los lenguajes pueden ser reconocidos por autómatas finitos. El **lema de bombeo** [Bar-Hillel et al., 1961] establece una condición necesaria que todo lenguaje regular debe satisfacer. La condición no es suficiente, por lo que el lema sirve únicamente para demostrar que un lenguaje no es regular, nunca para demostrar lo contrario.

Lema de bombeo para lenguajes regulares:
Sea L un lenguaje regular. Entonces existe una constante $n \geq 1$ (llamada *longitud de bombeo*) tal que toda cadena $w \in L$ con $|w| \geq n$ puede escribirse como $w = xyz$, satisfaciendo:

1. $|xy| \leq n$ (la sección que se repite aparece dentro de los primeros n símbolos),
2. $|y| \geq 1$ (la parte que se repite no puede ser vacía),
3. Para todo $i \geq 0$, $xy^iz \in L$ (la cadena permanece en el lenguaje al repetir o eliminar y).

Intuición detrás del lema

Los autómatas finitos no pueden contar indefinidamente. Si un AFD tiene n estados y procesa una cadena con más de n símbolos, necesariamente algún estado debe repetirse. Esto genera un ciclo durante la ejecución del autómata.

El segmento y representa precisamente esa parte cíclica de la cadena, por lo que puede repetirse (bombearse) cualquier cantidad de veces sin que la cadena deje de ser aceptada.

Cómo usar el lema para demostrar que un lenguaje NO es regular:

1. Se supone (por contradicción) que L SÍ es regular
2. Se elige estratégicamente una cadena $w \in L$ que requiera contar o emparejar símbolos
3. Se muestra que *cualquier* forma de dividir $w = xyz$ y bombear y produce una cadena fuera del lenguaje
4. Esto contradice el lema de bombeo
5. Por lo tanto, la suposición inicial es falsa: L NO es regular

---

Ejemplo 5: Demostración de que el lenguaje de paréntesis balanceados no es regular

Lenguaje: $L = \{(^n)^n \mid n \geq 1\} = \{(), (()), ((())), \ldots\}$

Este lenguaje contiene cadenas formadas por n paréntesis de apertura seguidos de n paréntesis de cierre. Reconocer bloques balanceados de este tipo aparece naturalmente en el análisis léxico de CompLab, y el lema de bombeo explica por qué un AFD no puede resolver esa tarea.

Demostración por contradicción:

1. **Suposición:** Supongamos que L es regular.
2. **Aplicación del lema:** Por el lema de bombeo, existe una constante $n \geq 1$ tal que toda cadena en L con longitud $\geq n$ se puede dividir en tres partes $w = xyz$ cumpliendo las tres condiciones del lema.
3. **Elección de cadena:** Elegimos $w = (^n)^n$, es decir, n aperturas seguidas de n cierres. La cadena pertenece a L y cumple $|w| = 2n \geq n$.

---

4. **División según el lema:** w debe poder escribirse como $w = xyz$ donde $|xy| \leq n$, $|y| \geq 1$ y $xy^iz \in L$ para todo $i \geq 0$.

5. **Análisis de la división:** Como $|xy| \leq n$ y w comienza con n aperturas, las partes x e y caen completamente dentro de los primeros n paréntesis de apertura. Por lo tanto:

   $x = (^j$ (algunas aperturas, posiblemente ninguna)
   $y = (^k$ (al menos una apertura, pues $|y| \geq 1$)
   $z = (^{n-j-k})^n$ (las aperturas restantes más todos los cierres)

   donde $j \geq 0$, $k \geq 1$, y $j + k \leq n$.

6. **Bombeo con $i = 2$:** al repetir y dos veces se obtiene

   $$xy^2z = (^j \cdot ((^k)^2 \cdot (^{n-j-k})^n = (^{n+k})^n$$

   cuya longitud presenta $n + k$ aperturas y n cierres.

7. **Contradicción:** Como $k \geq 1$, la cadena xy^2z tiene más aperturas que cierres, luego $xy^2z \notin L$, contradiciendo la tercera condición del lema.

8. **Conclusión:** La suposición inicial es falsa: L no es regular. □

Ejemplo concreto con $n = 3$:

Consideremos $w = ((()))$. Una posible división es $x = (, y = ((, z =)))$:

- Eliminando y: $xy^0z = (\cdot\varepsilon\cdot))) = ()))$ (1 apertura, 3 cierres) $\Rightarrow \notin L$
- Sin bombear: $xy^1z = (\cdot((\cdot))) = ((()))$ (3 aperturas, 3 cierres) $\Rightarrow \in L$
- Bombeando una vez: $xy^2z = (\cdot((((\cdot))) = ((((()))$ (5 aperturas, 3 cierres) $\Rightarrow \notin L$

La restricción $|xy| \leq n$ obliga a que la parte bombeada pertenezca al bloque de aperturas, lo que rompe el balance de la cadena. Para el lexer de CompLab, esto significa que la verificación de paréntesis balanceados no puede resolverse únicamente mediante análisis léxico y debe tratarse durante el análisis sintáctico.

Implicación para el análisis léxico: El reconocimiento de estructuras anidadas, como paréntesis o llaves balanceadas, requiere mantener un conteo potencialmente ilimitado. Debido a esta limitación, los analizadores léxicos no son suficientes para resolver este tipo de problemas y es necesario utilizar gramáticas libres de contexto en la fase sintáctica.

Minimización de AFD

Para cada lenguaje regular existe un **AFD mínimo** que lo reconoce, es decir, uno con el menor número posible de estados. El primer algoritmo de minimización fue propuesto por Moore [Moore, 1956], y posteriormente Hopcroft desarrolló una versión más eficiente [Hopcroft, 1971]. La idea conceptual común a estos algoritmos es *refinar particiones* de estados según su comportamiento: estados que no pueden distinguirse mediante ninguna cadena de entrada se fusionan en uno solo.

Hablar de ese AFD mínimo como algo *único* exige antes una aclaración: dos autómatas pueden diferir solo en el nombre de sus estados y, aun así, ser el mismo a todos los efectos. Esa relación se llama isomorfismo. Dos AFD son *isomorfos* cuando existe una correspondencia uno a uno entre sus estados que conserva el estado inicial, los estados de aceptación y todas las transiciones. Por ejemplo, un AFD con estados $\{q_0, q_1, q_2\}$ y otro con $\{A, B, C\}$ son isomorfos si la asociación $q_0 \leftrightarrow A$, $q_1 \leftrightarrow B$, $q_2 \leftrightarrow C$ respeta esas tres condiciones; uno se obtiene del otro sin más que renombrar los estados.

El AFD mínimo de un lenguaje regular es único *salvo isomorfismo*. Este resultado se sustenta en el **teorema de Myhill-Nerode** [Myhill, 1957, Nerode, 1958], que caracteriza los lenguajes regulares mediante relaciones de equivalencia de índice finito sobre Σ^*.

Estados equivalentes

Definición (Estados equivalentes): Dos estados p y q de un AFD son **equivalentes**, denotado $p \equiv q$, si ninguna cadena permite distinguirlos por aceptación:

$$p \equiv q \iff \forall w \in \Sigma^* : \Big[\hat{\delta}(p, w) \in F \Leftrightarrow \hat{\delta}(q, w) \in F\Big].$$

Estados equivalentes pueden fusionarse en un solo estado sin alterar el lenguaje reconocido.

Algoritmo de minimización (refinamiento de particiones)

Se asume que el AFD es completo; en caso contrario, basta añadir un estado sumidero antes de aplicar el algoritmo.

1. **Paso 0:** Eliminar los estados inalcanzables desde el estado inicial.
2. **Paso 1:** Construir la partición inicial separando estados de aceptación y de no aceptación:
$$\Pi_0 = \{F,\ Q \setminus F\}.$$
3. **Paso 2:** Refinar la partición. La nueva partición Π_{i+1} se obtiene dividiendo cada bloque de Π_i de modo que dos estados p, q permanezcan en el mismo bloque *si y solo si*:
   - están en el mismo bloque de Π_i, y
   - para todo $a \in \Sigma$, los sucesores $\delta(p, a)$ y $\delta(q, a)$ están en el mismo bloque de Π_i.
4. **Paso 3:** Repetir el refinamiento hasta que $\Pi_{i+1} = \Pi_i$, es decir, hasta que la partición se estabilice.
5. **Paso 4:** Construir el AFD mínimo $A/\equiv$ a partir de la partición final Π:
   - los *estados* son los bloques de Π;
   - el *estado inicial* es el bloque que contiene a q_0;
   - los *estados de aceptación* son los bloques contenidos en F;
   - la *transición* se define como $\delta'([p], a) = [\delta(p, a)]$, que está bien definida porque todos los estados de un mismo bloque tienen sucesores dentro de un mismo bloque.

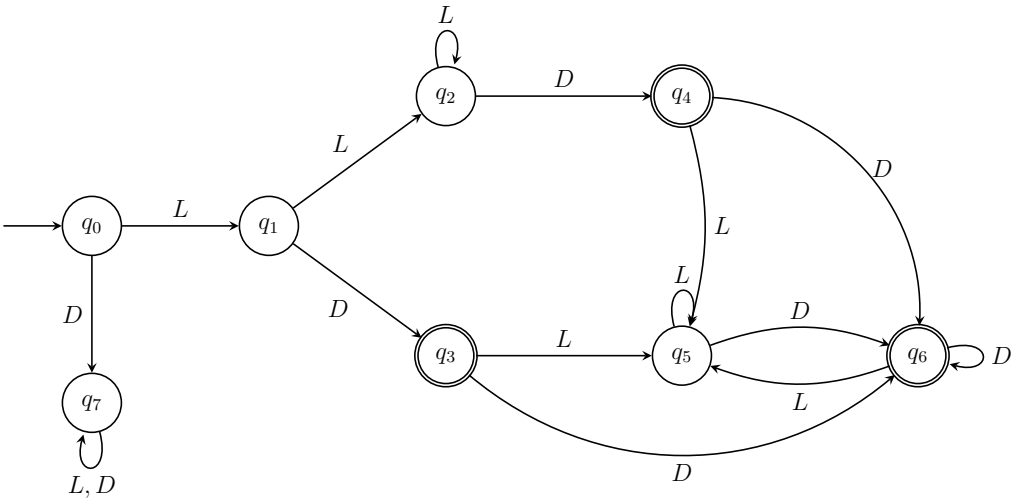
Ejemplo 6: Minimización del autómata para identificadores que terminan en dígito

Algunas convenciones de nomenclatura en lenguajes de ensamblador y sistemas heredados exigen que los identificadores comiencen con letra y terminen obligatoriamente en dígito. El lenguaje es:

$$L_{\text{id}} = L \cdot (L \cup D)^* \cdot D$$

Aplicaremos el algoritmo de minimización a un AFD que lo reconoce pero contiene varios estados redundantes producto de un diseño descuidado.

AFD inicial M (diseño con estados redundantes): Una primera versión del AFD que reconoce L_{id} podría tener la siguiente estructura, con estados que representan cada paso del proceso de lectura pero sin optimización:



Paso 0: Todos los estados son alcanzables desde q_0 (puede verificarse con un recorrido en anchura).

Paso 1: Partición inicial separando finales de no finales:

$$\Pi_0 = \Big\{ \{q_3, q_4, q_6\},\ \{q_0, q_1, q_2, q_5, q_7\} \Big\}$$

Paso 2 (iteración 1): Refinamos cada bloque examinando a qué bloque de Π_0 va cada estado con cada símbolo.

Bloque $\{q_0, q_1, q_2, q_5, q_7\}$*:*

- Con L: todos los destinos $(q_1, q_2, q_2, q_5, q_7)$ caen en el bloque de no finales.
- Con D: $\delta(q_0, D) = q_7$ (no final), $\delta(q_1, D) = q_3$ (final), $\delta(q_2, D) = q_4$ (final), $\delta(q_5, D) = q_6$ (final), $\delta(q_7, D) = q_7$ (no final).

Bajo D, los estados se reparten en dos grupos: los que van a finales $\{q_1, q_2, q_5\}$ y los que van a no finales $\{q_0, q_7\}$. El bloque se divide.

Bloque $\{q_3, q_4, q_6\}$*:* con L todos van a q_5 (no final); con D todos van a q_6 (final). No se separan.

$$\Pi_1 = \Big\{\{q_3, q_4, q_6\},\ \{q_1, q_2, q_5\},\ \{q_0, q_7\}\Big\}$$

Paso 2 (iteración 2): Refinamos los bloques de Π_1.

Bloque $\{q_0, q_7\}$*:*

- Con L: $\delta(q_0, L) = q_1 \in \{q_1, q_2, q_5\}$ pero $\delta(q_7, L) = q_7 \in \{q_0, q_7\}$. Van a bloques distintos: se separan.

Bloque $\{q_1, q_2, q_5\}$*:* con L los destinos q_2, q_2, q_5 están todos en $\{q_1, q_2, q_5\}$; con D los destinos q_3, q_4, q_6 están todos en $\{q_3, q_4, q_6\}$. No se separan.

Bloque $\{q_3, q_4, q_6\}$*:* idéntico al análisis de la iteración 1. No se separan.

$$\Pi_2 = \Big\{\{q_3, q_4, q_6\},\ \{q_1, q_2, q_5\},\ \{q_0\},\ \{q_7\}\Big\}$$

Paso 2 (iteración 3): Verificamos estabilidad de cada bloque con Π_2.

- $\{q_3, q_4, q_6\}$: con L todos van a $\{q_1, q_2, q_5\}$, con D todos van a $\{q_3, q_4, q_6\}$. Estable.
- $\{q_1, q_2, q_5\}$: con L todos a $\{q_1, q_2, q_5\}$, con D todos a $\{q_3, q_4, q_6\}$. Estable.
- $\{q_0\}$, $\{q_7\}$: bloques unitarios, trivialmente estables.

$$\Pi_3 = \Pi_2$$

Paso 3: La partición se estabiliza tras dos refinamientos efectivos; el algoritmo termina.

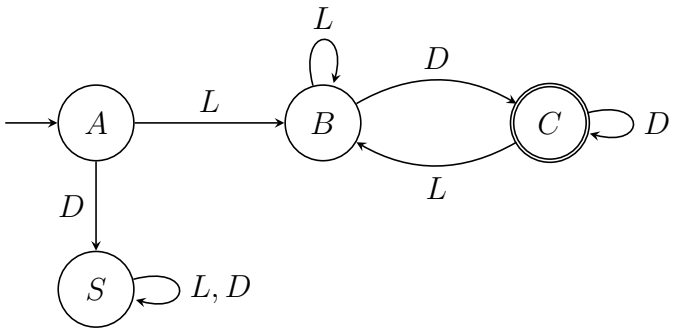
Paso 4 (construcción del AFD mínimo):

- Bloque $\{q_0\} \to$ estado A (inicial).
- Bloque $\{q_1, q_2, q_5\} \to$ estado B.
- Bloque $\{q_3, q_4, q_6\} \to$ estado C (final).
- Bloque $\{q_7\} \to$ estado S (sumidero).

Transiciones del cociente:

	L	D
A	B	S
B	B	C
C	B	C
S	S	S

AFD mínimo resultante:



El AFD pasó de 8 a 4 estados, una reducción del 50 %, y el algoritmo realizó dos refinamientos efectivos antes de estabilizarse.

Ventajas de la minimización. En la implementación de un analizador léxico:

- **Memoria:** la tabla de transiciones se reduce de $8 \times 2 = 16$ entradas a $4 \times 2 = 8$. En un analizador real, donde $|\Sigma|$ puede ser de cientos de caracteres, el ahorro escala proporcionalmente.

- **Mantenimiento:** cuatro estados con significado claro (inicial, "en construcción", "aceptable", sumidero) son más fáciles de verificar y modificar que ocho con distinciones artificiales.

- **Forma canónica:** como el AFD mínimo es único salvo isomorfismo, dos diseños distintos del mismo lenguaje pueden compararse por equivalencia minimizando ambos y verificando si las versiones reducidas tienen la misma estructura.

Tabla de evolución de particiones:

Iteración	**Partición**	**Observación**
Π_0	$\{\{q_3, q_4, q_6\},\ \{q_0, q_1, q_2, q_5, q_7\}\}$	Separación inicial
Π_1	$\{\{q_3, q_4, q_6\},\ \{q_1, q_2, q_5\},\ \{q_0, q_7\}\}$	Se aísla el grupo cuyo $\delta(\cdot, D)$ es final
Π_2	$\{\{q_3, q_4, q_6\},\ \{q_1, q_2, q_5\},\ \{q_0\},\ \{q_7\}\}$	q_0 y q_7 se distinguen porque q_7 es sumidero
Π_3	$\{\{q_3, q_4, q_6\},\ \{q_1, q_2, q_5\},\ \{q_0\},\ \{q_7\}\}$	Sin cambios; el algoritmo termina

c. ¿Qué se puede reutilizar de soluciones pasadas?

De los problemas anteriores se reutilizan las expresiones regulares para definir los patrones de cada tipo de token, la construcción de AFD para implementar su reconocimiento, las propiedades de clausura para combinar subpatrones y el procedimiento de conversión AFN $\rightarrow$ AFD.

d. ¿Cuál es el diseño de la solución?

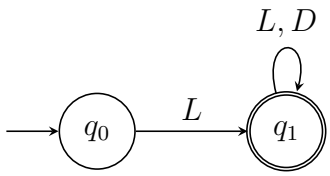
Para diseñar el analizador léxico, cada tipo de token se especifica con una expresión regular y se implementa como un AFD. La tabla siguiente resume las expresiones regulares del lenguaje simplificado, usando L para letras minúsculas y D para dígitos:

Tipo de token	Expresión regular	Descripción
KEYWORD	`if`, `while`, `return`	Palabras reservadas exactas
IDENTIFIER	$L \cdot (L \cup D)^*$	Inicia con letra
INTEGER	D^+	Uno o más dígitos
OPERATOR	$+, -, *, /, =, ==$	Operadores aritméticos y relacionales
DELIMITER	$(,), ;$	Separadores

A continuación se presenta el diagrama de transición de cada autómata. Se omiten los estados de error para mayor claridad; cualquier transición no mostrada lleva implícitamente a un estado de rechazo.

Autómata $M_{\mathbf{id}}$: identificadores

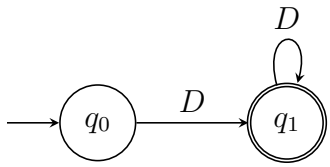
$L_{\text{id}} = L \cdot (L \cup D)^*$ (letra seguida de cero o más letras o dígitos)



q_0: inicial; q_1: al menos una letra leída (IDENTIFIER), acepta más letras o dígitos.

Autómata $M_{\mathbf{int}}$: números enteros

$L_{\text{int}} = D^+$ (uno o más dígitos)

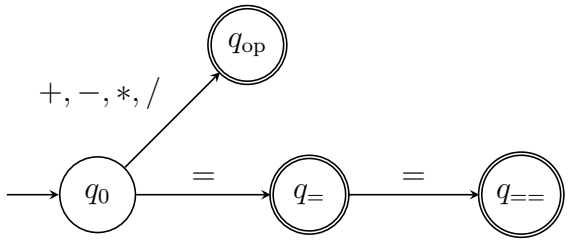


q_0: inicial; q_1: al menos un dígito leído (INTEGER), acepta más dígitos.

Autómata $M_{\mathbf{op}}$: operadores

$L_{\text{op}} = \{+\} \cup \{-\} \cup \{*\} \cup \{/\} \cup \{=\} \cup \{==\}$

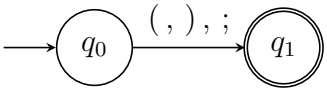
Este autómata es el más interesante: = y == comparten prefijo, por lo que al leer el primer = aún no se sabe si el token será = o ==.



q_0: inicial; q_{op}: operador simple (+, -, *, /); $q_=$: leído =, podría ser = o inicio de ==; $q_{==}$: leído exactamente ==.

Autómata M_{del}: delimitadores

$L_{\text{del}} = \{(\} \cup \{)\} \cup \{;\}$



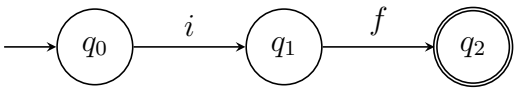
q_0: inicial; q_1: delimitador leído (DELIMITER). Siempre de un único carácter.

Autómata M_{if}: palabras clave

Las palabras clave deben reconocerse de manera exacta. Tomamos M_{if} como ejemplo; los autómatas para `while` y `return` siguen la misma construcción, con algunos estados intermedios más.

El alfabeto es $\Sigma = \{i, f, L', D\}$, donde L' representa las letras distintas de `i` y `f`.

$L_{\text{if}} = \{i\} \cdot \{f\}$



q_0: inicial; q_1: leído `i`, esperando `f`; q_2: leído exactamente `if` (KEYWORD).

El analizador léxico aplica estos autómatas en paralelo y, al detectar el lexema más largo posible, registra el token con el tipo correspondiente (principio de *maximal munch*).

2.4.3 Ejemplos de pruebas

a. Ejemplo 1

Cadena $w =$ "if" (aceptada — palabra clave)

Paso	**Símbolo**	**Estado**
0	—	q_0
1	i	q_1
2	f	q_2

Estado final: $q_2 \in F \Rightarrow$ **ACEPTADA** (KEYWORD)

b. Ejemplo 2

Cadena $w =$ "iffy" (rechazada — no es palabra clave)

Paso	**Símbolo**	**Estado**
0	—	q_0
1	i	q_1
2	f	q_2
3	f	q_e
4	y	q_e

Estado final: $q_e \notin F \Rightarrow$ **RECHAZADA** (no es palabra clave; M_{id} la reconocería como IDENTIFIER)

c. Ejemplo 3

Tokenización de la cadena `"if x == 42 + y;"`

El analizador léxico recorre la cadena carácter a carácter agrupando lexemas y consultando cada autómata. La secuencia de tokens resultante es:

Lexema	Tipo de token	Autómata
`if`	KEYWORD	M_{if}
`x`	IDENTIFIER	M_{id}
`==`	OPERATOR	M_{op}
`42`	INTEGER	M_{int}
`+`	OPERATOR	M_{op}
`y`	IDENTIFIER	M_{id}
`;`	DELIMITER	M_{del}

Resultado: 7 tokens identificados $\Rightarrow$ **ACEPTADA**

2.4.4 Codificación

A diferencia de los validadores anteriores, el analizador léxico reconoce múltiples tipos de tokens. El código se organiza en tres capas: los reconocedores por categoría, la función que clasifica un lexema completo, y el tokenizador que escanea la cadena fuente.

Primero se presenta la función principal que orquesta el flujo completo. Esta función coordina la lectura del código fuente, la tokenización y la impresión de los tokens identificados:

Código 2.17: Función principal

```
def main():
    entrada = leer("Ingrese el código a analizar: ")
    lista_tokens = tokenizar(entrada)
    imprimir_resultado(lista_tokens)
```

Las funciones auxiliares de entrada y salida encapsulan la lectura del código fuente desde la consola y la impresión formateada de los tokens reconocidos en una tabla con columnas para el lexema y su tipo:

Código 2.18: Funciones de entrada y salida

```
def leer(texto):
    return input(texto)

def imprimir_resultado(tokens):
    print(f"{'LEXEMA':<15} | {'TIPO'}")
    print("-" * 30)
    for lexema, tipo in tokens:
        print(f"{repr(lexema):<15} | {tipo}")
```

Se definen los conjuntos fijos de keywords, operadores y delimitadores que el analizador reconoce directamente:

Código 2.19: Constantes

```
KEYWORDS = {"if", "while", "return"}
OPERADORES = {"+", "-", "*", "/", "=", "=="}
DELIMITADORES = {"(", ")", ";"}
```

Cada tipo de token tiene su propia función extractora que, dada la cadena fuente y una posición inicial, consume los caracteres correspondientes y retorna el lexema, su tipo y la nueva posición. La función `extraer_palabra` distingue entre keywords e identificadores consultando el conjunto de palabras reservadas:

Código 2.20: Funciones extractoras

```
def extraer_palabra(codigo, i):
    """Extrae un identificador o una keyword."""
    inicio = i
    while i < len(codigo) and codigo[i].isalnum():
        i += 1
    lexema = codigo[inicio:i]
    tipo = "KEYWORD" if lexema in KEYWORDS else "IDENTIFIER"
    return lexema, tipo, i


def extraer_entero(codigo, i):
    """Extrae un número entero."""
    inicio = i
    while i < len(codigo) and codigo[i].isdigit():
        i += 1
    return codigo[inicio:i], "INTEGER", i


def extraer_operador(codigo, i):
    """Extrae operadores, priorizando los de dos caracteres (==)."""
    # Caso especial: Operador de dos caracteres
    if i + 1 < len(codigo) and codigo[i : i + 2] == "==":
        return "==", "OPERATOR", i + 2

    # Caso general: Operador de un carácter
    if codigo[i] in OPERADORES:
        return codigo[i], "OPERATOR", i + 1
    return None


def extraer_delimitador(codigo, i):
    """Extrae delimitadores de un solo carácter."""
    if codigo[i] in DELIMITADORES:
        return codigo[i], "DELIMITER", i + 1
    return None
```

El tokenizador recorre la cadena fuente carácter a carácter, delegando la extracción de cada token a la función extractora correspondiente según el tipo de carácter encontrado:

Código 2.21: `Tokenizador`

```
57 def tokenizar(codigo):
58     tokens = []
59     i = 0
60     n = len(codigo)
61 
62     while i < n:
63         char = codigo[i]
64 
65         # 1. Ignorar espacios
66         if char.isspace():
67             i += 1
68             continue
69 
70         # 2. Intentar extraer Palabra (ID o Keyword)
71         if char.isalpha():
72             lexema, tipo, i = extraer_palabra(codigo, i)
73             tokens.append((lexema, tipo))
74             continue
75 
76         # 3. Intentar extraer Entero
77         if char.isdigit():
78             lexema, tipo, i = extraer_entero(codigo, i)
79             tokens.append((lexema, tipo))
80             continue
81 
82         # 4. Intentar extraer Operador
83         resultado_op = extraer_operador(codigo, i)
84         if resultado_op:
85             lexema, tipo, i = resultado_op
86             tokens.append((lexema, tipo))
87             continue
88 
89         # 5. Intentar extraer Delimitador
90         resultado_del = extraer_delimitador(codigo, i)
91         if resultado_del:
92             lexema, tipo, i = resultado_del
93             tokens.append((lexema, tipo))
94             continue
95 
96         # 6. Carácter desconocido
97         print(f"Error léxico: Carácter '{char}' no reconocido en posición {
               i}")
98         i += 1
99 
100     return tokens
```

Para ejecutar el programa, se llama a `main()` al final del archivo, lo que desencadena la lectura del código fuente, su tokenización y la impresión de los resultados.

2.5. Validación de correos electrónicos (simplificado)

Conecta Digital es una organización no gubernamental que promueve la inclusión digital en comunidades rurales de América Latina. Como parte de su programa de alfabetización tecnológica, ha desarrollado una plataforma web donde los participantes pueden registrarse utilizando una dirección de correo electrónico. Sin embargo, muchos usuarios ingresan direcciones con errores de formato, lo que impide la comunicación posterior y genera frustración.

El equipo técnico de Conecta Digital necesita implementar un validador de correos electrónicos que verifique el formato básico de las direcciones ingresadas: una parte local compuesta por letras minúsculas o dígitos, seguida del símbolo arroba, un dominio compuesto por letras minúsculas, un punto y una extensión de exactamente dos o tres letras minúsculas. Este validador debe ser suficientemente simple para funcionar en dispositivos con recursos limitados y conexiones lentas.

Se trata de una versión simplificada del problema real de validación de correos electrónicos, cuyo estándar completo (RFC 5322) [Resnick, 2008] es considerablemente más complejo. No obstante, esta versión es suficiente para los propósitos de la organización.

2.5.1 Análisis / Abstracción

El requerimiento de Conecta Digital se reduce a un problema de validación de formato sobre una cadena:

a. ¿Cuál es el problema a resolver?

Conecta Digital requiere un validador que verifique el formato básico de las direcciones de correo ingresadas por los participantes del programa, condición necesaria para garantizar la comunicación posterior con los usuarios registrados.

b. ¿Cuál es la información necesaria?

El formulario de registro entrega al validador una sola dirección por usuario:

Información necesaria	Tipo de dato
Cadena de entrada (dirección de correo)	Texto (String)

2.5.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **ENTRADA:** Una cadena de caracteres proporcionada como dirección de correo electrónico.
- **PROCESAMIENTO:**
  1. Lectura de la cadena de entrada.
  2. Definición de la función de transición del AFD que reconoce el formato de correo electrónico.
  3. Validación del correo electrónico mediante el AFD.
  4. Impresión del resultado de la validación.
- **SALIDA:** Una decisión binaria: `ACEPTADA` si el correo cumple con el formato, o `RECHAZADA` si no lo hace.

b. ¿Qué conocimiento adicional se requiere?

Esta sección integra las herramientas desarrolladas a lo largo del capítulo, sin introducir nuevos formalismos. Las expresiones regulares describen el patrón del correo electrónico, el AFD implementa el reconocimiento símbolo a símbolo y las propiedades de clausura garantizan que la combinación de subpatrones (parte local, dominio, extensión) produce un lenguaje regular. La minimización del AFD, estudiada en la sección anterior, permite optimizar la implementación resultante.

c. ¿Qué se puede reutilizar de soluciones pasadas?

En este problema pueden reutilizarse varias de las técnicas desarrolladas anteriormente. Del primer ejercicio se retoma el algoritmo de reconocimiento mediante AFD. De la sección sobre LogiTrack se aprovecha la construcción modular por clausura, útil para combinar los subpatrones correspondientes a la parte local, el dominio y la extensión. Finalmente,

de la sección de análisis léxico se reutiliza la minimización de autómatas para reducir el número de estados de la implementación final.

d. ¿Cuál es el diseño de la solución?

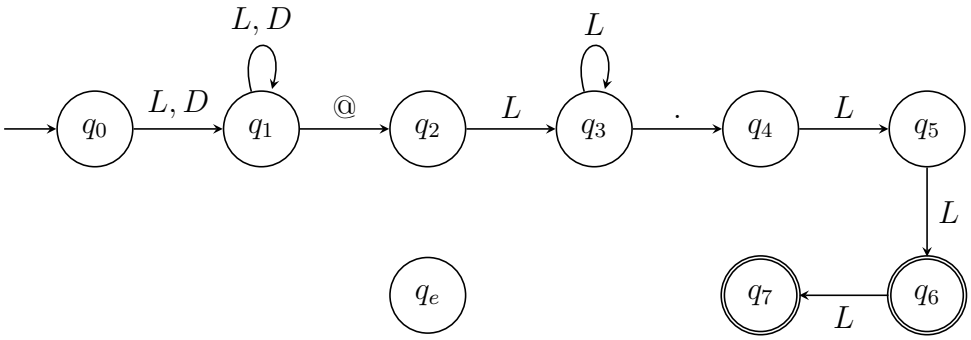
Expresión regular L_{email}

Sea $\Sigma = \{a, b, \ldots, z, 0, 1, \ldots, 9, @, .\}$, con L el conjunto de letras minúsculas y D el de dígitos. La expresión regular que describe los correos válidos en el formato simplificado es:

$$L_{\text{email}} = (L \cup D)^+ \cdot \{@\} \cdot L^+ \cdot \{.\} \cdot (L^2 \cup L^3)$$

donde L^2 y L^3 representan exactamente 2 o 3 letras para la extensión del dominio.

Autómata M_{email}



Se omiten las transiciones que llevan al estado de error q_e para claridad del diagrama. Cualquier entrada inválida debe llevar a q_e.

Interpretación de los estados

- q_0: estado inicial, esperando parte local
- q_1: leyendo parte local (usuario)
- q_2: @ leído, esperando dominio
- q_3: leyendo dominio

- q_4: punto leído, esperando primera letra de extensión
- q_5: 1 letra de extensión leída (aún no válido)
- q_6: 2 letras de extensión leídas (válido - ej: .co)
- q_7: 3 letras de extensión leídas (válido - ej: .com)
- q_e: formato inválido

Definición formal del AFD

Definimos el autómata $M_{\text{email}} = (Q, \Sigma, \delta, q_0, F)$ donde:

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_e\}$
- $\Sigma = \{L, D, @, .\}$ (letras, dígitos, arroba, punto)
- q_0 es el estado inicial
- $F = \{q_6, q_7\}$ (extensión de 2 o 3 letras)
- q_e es el estado de error

Función de transición

La función $\delta : Q \times \Sigma \to Q$ se define mediante la siguiente tabla:

δ	L (letra)	D (dígito)	@	. (punto)
q_0	q_1	q_1	q_e	q_e
q_1	q_1	q_1	q_2	q_e
q_2	q_3	q_e	q_e	q_e
q_3	q_3	q_e	q_e	q_4
q_4	q_5	q_e	q_e	q_e
q_5	q_6	q_e	q_e	q_e
q_6	q_7	q_e	q_e	q_e
q_7	q_e	q_e	q_e	q_e
q_e	q_e	q_e	q_e	q_e

2.5.3 Ejemplos de pruebas

a. Ejemplo 1

Cadena $w =$ “user@mail.com” (aceptada - extensión de 3 letras)

Paso	Símbolo	Estado
0	—	q_0
1	u (L)	q_1
2	s (L)	q_1
3	e (L)	q_1
4	r (L)	q_1
5	@	q_2
6	m (L)	q_3
7	a (L)	q_3
8	i (L)	q_3
9	l (L)	q_3
10	.	q_4
11	c (L)	q_5
12	o (L)	q_6
13	m (L)	q_7

Estado final: $q_7 \in F \Rightarrow$ **ACEPTADA**

b. Ejemplo 2

Cadena $w =$ “test@example.co” (aceptada - extensión de 2 letras)

Paso	Símbolo	Estado
0	—	q_0
1	t (L)	q_1
2	e (L)	q_1
3	s (L)	q_1
4	t (L)	q_1
5	@	q_2
6–12	(example)	q_3
13	.	q_4
14	c (L)	q_5
15	o (L)	q_6

Estado final: $q_6 \in F \Rightarrow$ **ACEPTADA**

c. Ejemplo 3

Cadena $w =$ “invalid.email” (rechazada - falta @)

Paso	Símbolo	Estado
0	—	q_0
1–7	(invalid)	q_1
8	.	q_e

Estado final: $q_e \notin F \Rightarrow$ **RECHAZADA**

2.5.4 Codificación

La estructura de cinco componentes vuelve a aparecer, esta vez sobre un alfabeto más rico (letras, dígitos, arroba y punto) y dos estados de aceptación, $F = \{q_6, q_7\}$, que distinguen las extensiones de dos y tres letras.

La función `main` se encarga de leer la dirección de correo electrónico ingresada, ejecutar el autómata correspondiente y mostrar si la cadena es aceptada o rechazada.

Código 2.22: Función principal

```
def main():
    cadena = leer("Ingrese el correo electronico: ")
    transicion = definir_funcion_transicion()
    resultado = validar_email(cadena, transicion)
    imprimir_resultado(cadena, resultado)
```

La capa de E/S concentra la interacción con la consola, sin novedades respecto a las secciones previas.

Código 2.23: Funciones de entrada y salida

```
def leer(texto):
    return input(texto)

def imprimir_resultado(cadena, es_valido):
    if es_valido:
        print(f"'{cadena}': ACEPTADA")
    else:
        print(f"'{cadena}': RECHAZADA")
```

A diferencia del ejercicio anterior, aquí intervienen cuatro categorías de símbolos en la clasificación: letra, dígito, arroba y punto.

Código 2.24: Función de clasificación

```
def clasificar(caracter):
    if caracter.isalpha() and caracter.islower():
        return "L"
    elif caracter.isdigit():
        return "D"
    elif caracter == "@":
        return "@"
    elif caracter == ".":
        return "."
    else:
        return "error"
```

La validación dispone de dos rutas de aceptación: el estado final puede ser q_6 (extensión de dos letras) o q_7 (extensión de tres letras).

Código 2.25: Función de validación

```
def validar_email(cadena, transicion):
    estado = "q0"
    for caracter in cadena:
        simbolo = clasificar(caracter)
        estado = transicion.get((estado, simbolo), "qe")
    return estado in ("q6", "q7")
```

Por último, la tabla cubre los nueve estados del autómata; cualquier combinación inválida conduce al estado de error.

Código 2.26: Función de transición

```
def definir_funcion_transicion():
    return {
        ("q0", "L"): "q1", ("q0", "D"): "q1", ("q0", "@"): "qe", ("q0", "."
            ): "qe",
        ("q1", "L"): "q1", ("q1", "D"): "q1", ("q1", "@"): "q2", ("q1", "."
            ): "qe",
        ("q2", "L"): "q3", ("q2", "D"): "qe", ("q2", "@"): "qe", ("q2", "."
            ): "qe",
        ("q3", "L"): "q3", ("q3", "D"): "qe", ("q3", "@"): "qe", ("q3", "."
            ): "q4",
        ("q4", "L"): "q5", ("q4", "D"): "qe", ("q4", "@"): "qe", ("q4", "."
            ): "qe",
        ("q5", "L"): "q6", ("q5", "D"): "qe", ("q5", "@"): "qe", ("q5", "."
            ): "qe",
        ("q6", "L"): "q7", ("q6", "D"): "qe", ("q6", "@"): "qe", ("q6", "."
            ): "qe",
        ("q7", "L"): "qe", ("q7", "D"): "qe", ("q7", "@"): "qe", ("q7", "."
            ): "qe",
        ("qe", "L"): "qe", ("qe", "D"): "qe", ("qe", "@"): "qe", ("qe", "."
            ): "qe",
    }
```

2.6. Ejercicios

2.6.1 Diseño de autómatas finitos

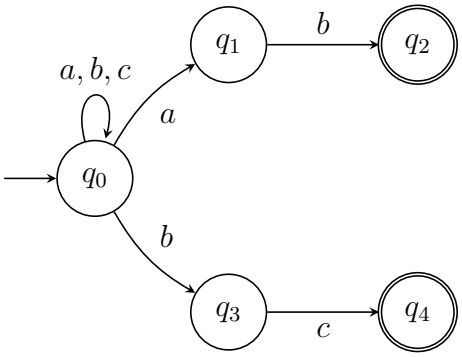
La empresa CyberGuard, especializada en ciberseguridad, necesita herramientas de reconocimiento de patrones para analizar tráfico de red. Como parte del proceso de capacitación de su equipo técnico, se requiere que los ingenieros demuestren competencia en el diseño de autómatas finitos deterministas para patrones básicos.

Se requiere un AFD sobre el alfabeto $\Sigma = \{a, b\}$ que acepte exactamente las cadenas que contienen un número par de letras `a`, sin importar la cantidad de letras `b` presentes.

2.6.2 Conversión de autómatas

Un sistema de análisis de protocolos de comunicación necesita detectar si una trama de datos, representada como una cadena sobre $\Sigma = \{a, b, c\}$, termina con la subcadena ab o con la subcadena bc. El equipo de ingeniería diseñó el siguiente AFN y ahora necesita convertirlo a un AFD para implementarlo eficientemente en hardware dedicado.

Aplique la construcción de subconjuntos para obtener un AFD equivalente. Diagrama del AFN:



Observación: el no determinismo aparece en q_0: al leer a, el autómata puede permanecer en q_0 o avanzar a q_1; al leer b, puede permanecer en q_0 o avanzar a q_3.

2.6.3 Especificación mediante expresiones regulares

En el desarrollo de herramientas de procesamiento de texto, las expresiones regulares son el medio principal para especificar patrones de búsqueda y validación. Un equipo de desarrollo de software necesita que sus programadores sean capaces de escribir expresiones regulares precisas para distintos tipos de patrones lingüísticos y numéricos.

Escriba expresiones regulares para los siguientes lenguajes:

a) Cadenas sobre $\{L, D, @\}$ (letras, dígitos y símbolo arroba) que contienen exactamente una ocurrencia del símbolo @. (Ejemplo simplificado de validación de la parte local de un correo electrónico.)

b) Números naturales sin ceros a la izquierda.

c) Comentarios de una línea en C (que comienzan con "//").

2.6.4 Límites de los lenguajes regulares

Un grupo de investigación en teoría de la computación está evaluando los límites de los modelos de reconocimiento basados en autómatas finitos.

Caracterizar qué lenguajes quedan fuera de su alcance permite elegir el formalismo adecuado para cada problema.

Demuestre usando el lema de bombeo que el siguiente lenguaje no es regular:

$$L = \{a^n b^m \mid n > m,\ n, m \geq 0\}$$

2.6.5 Comparación de formalismos

Un sistema de monitoreo de eventos clasifica registros de actividad usando secuencias de tres tipos de evento (a, b, c). El equipo de análisis necesita detectar aquellos registros que contienen la secuencia de eventos ac en alguna posición. Se requiere describir este patrón mediante los tres formalismos equivalentes estudiados en el capítulo.

Para el lenguaje $L = \{w \in \{a, b, c\}^* \mid w \text{ contiene la subcadena } ac\}$, proporcione:

a) Una expresión regular

b) Un AFN (aproveche el no determinismo: el autómata "adivina" en qué posición comienza el patrón ac)

c) Un AFD equivalente (obtenido mediante la construcción de subconjuntos)

2.6.6 Análisis de aceptación de cadenas

La verificación y validación de sistemas basados en autómatas requiere determinar si una cadena específica es aceptada o rechazada por un autómata dado. Este tipo de análisis se resuelve trazando el procesamiento paso a paso.

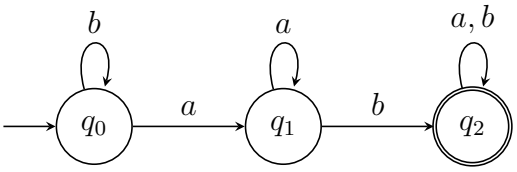
Considere el siguiente AFD $M = (Q, \Sigma, \delta, q_0, F)$ definido por:

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{a, b\}$
- $F = \{q_2\}$

Función de transición:

δ	a	b
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_2	q_2

Diagrama:



Determine si las siguientes afirmaciones son **verdaderas (V)** o **falsas (F)**:

a) La cadena vacía ε es aceptada por M.

b) La cadena ab es aceptada por M.

c) La cadena ba es aceptada por M.

d) La cadena aab es aceptada por M.

e) Todas las cadenas que contienen al menos una a seguida de al menos una b son aceptadas.

f) La cadena $bbbab$ es aceptada por M.

g) La cadena aba es aceptada por M.

h) El autómata acepta el lenguaje $L = \{w \in \{a, b\}^* \mid w$ contiene la subcadena $ab\}$.

2.6.7 Diseño de AFD a partir de descripción

Los siguientes tres ejercicios piden diseñar autómatas finitos a partir de descripciones en lenguaje natural. El reto está en traducir una especificación informal a una formal.

Diseñe un AFD para cada uno de los siguientes lenguajes. Proporcione:

- La definición formal de la quíntupla $(Q, \Sigma, \delta, q_0, F)$
- El diagrama de transición
- Una explicación breve de la estrategia utilizada

a) Cadenas sobre $\{a, b, c\}$ que empiezan con a y terminan con c. *Pista:* el AFD necesita rastrear si el primer símbolo fue a y cuál fue el último símbolo leído.

b) Cadenas sobre $\{a, b\}$ que no contienen dos a's consecutivas (no contienen la subcadena aa).

c) Cadenas sobre $\{a, b, c\}$ que no contienen la subcadena aa ni la subcadena bb (es decir, ningún símbolo a o b aparece dos veces consecutivas; el símbolo c puede repetirse libremente). *Pista:* el AFD debe recordar cuál fue el último símbolo leído.

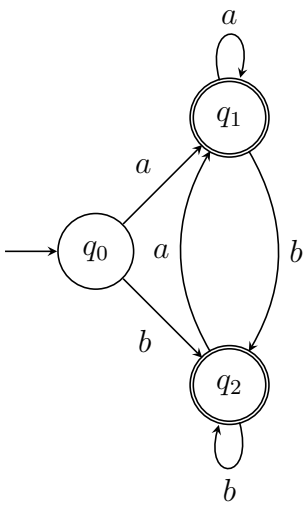
2.6.8 Descripción de lenguaje a partir de AFD

El problema inverso al diseño de autómatas consiste en analizar un AFD dado y determinar qué lenguaje reconoce. Esta habilidad permite verificar la corrección de los autómatas diseñados y comprender el comportamiento de sistemas existentes.

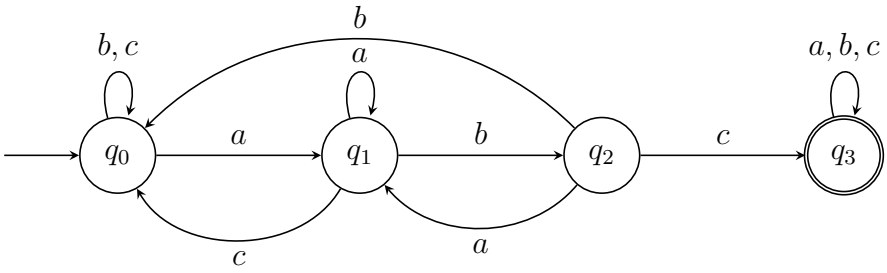
Para cada uno de los siguientes AFD, determine y describa formalmente el lenguaje que acepta. Proporcione:

- Una descripción en lenguaje natural
- La especificación formal usando notación de conjuntos
- Tres ejemplos de cadenas aceptadas y tres rechazadas

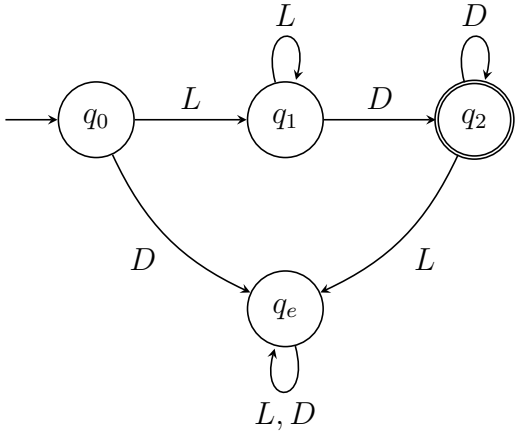
a) AFD M_1 con $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, $F = \{q_1, q_2\}$:



b) AFD M_2 con $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b, c\}$, $F = \{q_3\}$:



c) AFD M_3 con $Q = \{q_0, q_1, q_2, q_e\}$, $\Sigma = \{L, D\}$, $F = \{q_2\}$:



2.7. Cierre del capítulo

En este capítulo se presentaron los lenguajes regulares, los autómatas finitos y las expresiones regulares como herramientas para automatizar el reconocimiento de patrones simples. Estos modelos validan de forma eficiente una amplia clase de entradas presentes en sistemas reales, dentro de los límites expresivos que se hicieron explícitos a lo largo del capítulo.

Conceptos clave aprendidos

1. **Autómatas Finitos Deterministas (AFD):** Modelo computacional con un número finito de estados capaz de reconocer lenguajes regulares. Para cada símbolo de entrada existe una única transición posible. Se utilizan ampliamente en tareas de reconocimiento y validación de patrones.

2. **Autómatas Finitos No Deterministas (AFN):** Variante de los AFD que permite múltiples transiciones posibles y transiciones ε. Simplifica el diseño de autómatas, aunque posee el mismo poder computacional que un AFD.

3. **Equivalencia AFD-AFN:** Todo AFN puede transformarse en un AFD equivalente mediante la construcción de subconjuntos. Esto demuestra que el no determinismo no aumenta el poder computacional del modelo.

4. **Expresiones Regulares:** Forma compacta de describir lenguajes regulares. El teorema de Kleene establece que las expresiones regulares, los AFD y los AFN son formalismos equivalentes para representar esta clase de lenguajes.

5. **Propiedades de clausura:** Los lenguajes regulares son cerrados bajo unión, intersección, concatenación, cerradura de Kleene, complemento y diferencia. Estas propiedades permiten construir lenguajes más complejos a partir de otros más simples.

6. **Limitaciones (Lema de bombeo):** Los lenguajes regulares no pueden contar arbitrariamente ni emparejar símbolos. Lenguajes como $\{0^n1^n \mid n \geq 0\}$ no son regulares porque requieren una cantidad ilimitada de memoria para realizar el conteo.

7. **Minimización de AFD:** Para cada lenguaje regular existe un único AFD mínimo (salvo isomorfismo) con la menor cantidad de estados posible. El algoritmo de refinamiento de particiones permite identificar y fusionar estados equivalentes.

8. **Aplicación en compiladores:** Los autómatas finitos y las expresiones regulares se utilizan en el análisis léxico, etapa encargada de transformar el código fuente en tokens [Aho et al., 2006]. Herramientas como Lex/Flex [Lesk and Schmidt, 1975] automatizan este proceso a partir de especificaciones basadas en expresiones regulares.

Limitaciones y motivación para el siguiente capítulo

Los lenguajes regulares tienen limitaciones fundamentales:

- No pueden contar: $\{0^n 1^n \mid n \geq 0\}$ no es regular
- No pueden emparejar estructuras anidadas: paréntesis balanceados
- No tienen contexto: cada decisión depende solo del estado actual

Estas limitaciones motivan el estudio de lenguajes más expresivos. En el siguiente capítulo se estudiarán **lenguajes libres de contexto** y **autómatas de pila**, que añaden una pila como memoria auxiliar, permitiendo reconocer estructuras jerárquicas como expresiones aritméticas, paréntesis balanceados y la sintaxis de lenguajes de programación.

Capítulo 3

Lenguajes libres de contexto

¿Cómo analizar estructuras con jerarquía y anidamiento?

Muchos problemas en computación involucran estructuras donde los elementos no aparecen simplemente uno después de otro, sino que además se organizan de manera jerárquica o anidada. Algunos ejemplos comunes son:

- expresiones matemáticas con paréntesis,
- bloques de código delimitados por llaves,
- instrucciones anidadas,
- llamadas a funciones dentro de otras funciones.

En estos casos, no es suficiente verificar únicamente el formato de la entrada; también es necesario analizar la relación y el anidamiento entre sus elementos.

¿Cómo describir y analizar correctamente entradas que contienen dependencias estructurales y anidamiento, garantizando que la estructura sea válida?

Este capítulo aborda el análisis de estructuras jerárquicas y anidadas, problema que motiva el estudio de los lenguajes libres de contexto. A lo largo del capítulo, se presentan diversas situaciones problema que motivan la introducción progresiva de los conceptos fundamentales: gramáticas libres de contexto, notación BNF/EBNF, árboles de derivación,

ambigüedad, autómatas de pila, formas normales y propiedades de clausura. Cada problema se desarrolla siguiendo una metodología que integra análisis, diseño, pruebas y codificación.

Nota histórica

Los lenguajes libres de contexto tienen su origen en el trabajo de **Noam Chomsky** en la década de 1950. En 1956, Chomsky introdujo su famosa **jerarquía de gramáticas** para clasificar los lenguajes formales según su complejidad estructural [Chomsky, 1956], y en 1959 formalizó propiedades fundamentales de las gramáticas libres de contexto, incluyendo las formas normales [Chomsky, 1959]. Las gramáticas de tipo 2 en esta jerarquía corresponden a las gramáticas libres de contexto.

Paralelamente, **John Backus** y **Peter Naur** desarrollaron entre 1959 y 1960 la notación **BNF** (Backus-Naur Form) para describir la sintaxis del lenguaje de programación ALGOL 60 [Backus et al., 1960]. Esta notación, basada en gramáticas libres de contexto, pasó a ser ampliamente utilizada para especificar la sintaxis de lenguajes de programación.

El reconocimiento automático de estos lenguajes fue formalizado mediante los **autómatas de pila**, introducidos por **Anthony Oettinger** en 1961 [Oettinger, 1961], y la equivalencia entre gramáticas libres de contexto y autómatas de pila fue establecida por Chomsky en 1962 [Chomsky, 1962]. Posteriormente, en los años 1960s, surgieron algoritmos eficientes de parsing como el algoritmo CYK (Cocke-Younger-Kasami) [Kasami, 1966, Younger, 1967], el algoritmo de Earley para gramáticas arbitrarias [Earley, 1970] y los métodos de análisis ascendente de Knuth [Knuth, 1965] que dieron origen a los parsers LR utilizados en compiladores modernos.

3.1. Validación de delimitadores balanceados

La empresa DevTools está desarrollando un editor de código que debe verificar en tiempo real si los delimitadores escritos por los programadores están correctamente balanceados. Los delimitadores incluyen paréntesis `()`, corchetes `[]` y llaves {}.

El equipo ha identificado que cadenas como `([])`{} son válidas, mientras que `([)`} o `(()` no lo son. Se requiere un mecanismo formal para describir y verificar estas estructuras.

3.1.1 Análisis / Abstracción

a. ¿Cuál es el problema a resolver?

DevTools necesita un mecanismo formal para describir la estructura de cadenas con delimitadores anidados y verificar automáticamente si una cadena dada tiene todos sus delimitadores correctamente balanceados. Cada símbolo de apertura debe tener su correspondiente cierre, respetando el anidamiento.

b. ¿Cuál es la información necesaria?

Para poder resolver el problema es necesaria la siguiente información:

Información necesaria	Descripción
Cadena de delimitadores	El texto a verificar
Tipos de delimitadores	Paréntesis `()`, corchetes `[]`, llaves {}

La validez de la cadena depende de su estructura interna: cada delimitador de apertura debe tener su correspondiente cierre, respetando el anidamiento.

3.1.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **ENTRADA:** Una cadena de delimitadores.

- **PROCESAMIENTO:** Verificar que cada símbolo de apertura tenga su correspondiente cierre, respetando el orden de anidamiento.
- **SALIDA:** Indicar si los delimitadores están correctamente balanceados o no.

b. ¿Qué conocimiento adicional se requiere?

Para diseñar la solución se requiere un formalismo que permita describir lenguajes con estructura jerárquica y anidamiento. Los autómatas finitos y las expresiones regulares del capítulo anterior no son suficientes, pues no pueden "contar" de forma ilimitada. Se introduce entonces el concepto de **Gramática Libre de Contexto**.

Gramáticas Libres de Contexto (GLC)

Una *gramática libre de contexto* (GLC) es una cuádrupla [Hopcroft et al., 2006, Sipser, 2012]

$$G = (V, \Sigma, P, S)$$

donde:

- V es un conjunto finito no vacío de **variables** (o símbolos no terminales),
- Σ es un conjunto finito de **símbolos terminales**, con $V \cap \Sigma = \emptyset$,
- P es un conjunto finito de **producciones** (o reglas) de la forma $A \to \alpha$, donde $A \in V$ y $\alpha \in (V \cup \Sigma)^*$,
- $S \in V$ es el **símbolo inicial**.

Interpretación

- Las **variables** (o no terminales) representan categorías sintácticas o estructuras intermedias. Generalmente se escriben con letras mayúsculas: $A, B, S, \ldots$
- Los **símbolos terminales** corresponden a los símbolos que aparecen en las cadenas del lenguaje. Usualmente se representan con letras minúsculas o símbolos específicos: $a, b, c, +, (, \ldots$

- Una **producción** $A \to \alpha$ indica que la variable A puede reemplazarse por la cadena α.
- El proceso de generación inicia en el símbolo inicial S y continúa aplicando producciones hasta obtener una cadena formada únicamente por símbolos terminales.

Ejemplo 1: GLC para paréntesis balanceados

Como primer acercamiento, consideremos el subproblema más simple: solo paréntesis.

$L = \{w \in \{(,)\}^* \mid w \text{ tiene paréntesis balanceados}\}$

Este lenguaje incluye cadenas como: ε, (), (()), ()(), (()()), etc.

Podemos describir este lenguaje mediante la siguiente gramática:

$$S \quad \to \quad \varepsilon \mid (S) \mid SS$$

Componentes de la gramática:

- $V = \{S\}$ (una sola variable)
- $\Sigma = \{(,)\}$ (paréntesis de apertura y cierre)
- $P = \{S \to \varepsilon,\ S \to (S),\ S \to SS\}$
- S es el símbolo inicial

Interpretación de las producciones:

1. $S \to \varepsilon$: La cadena vacía tiene paréntesis balanceados
2. $S \to (S)$: Si w tiene paréntesis balanceados, entonces (w) también
3. $S \to SS$: Si w_1 y w_2 tienen paréntesis balanceados, entonces $w_1 w_2$ también

Derivación:

Para verificar que una cadena pertenece al lenguaje generado por una gramática, se utiliza un proceso llamado **derivación**.

Una **derivación** es el proceso de construir una cadena reemplazando paso a paso las variables utilizando las producciones de la gramática.

- En cada paso se sustituye una variable por el contenido de una de sus producciones.
- El proceso comienza con el símbolo inicial S y finaliza cuando la cadena contiene únicamente símbolos terminales.
- Se denota con la flecha $\Rightarrow$ (un paso) o $\Rightarrow^*$ (varios pasos).

Lenguaje generado:

Para definir formalmente el lenguaje generado por una gramática, se introduce el siguiente concepto:

El **lenguaje generado** por una gramática ($L(G)$) corresponde al conjunto de todas las cadenas formadas únicamente por símbolos terminales que pueden obtenerse desde el símbolo inicial mediante derivaciones.

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

Derivemos la cadena $(()())$ a partir de S:

$$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow (()S) \Rightarrow (()(S)) \Rightarrow (()())$$

En cada paso, seleccionamos una variable y la sustituimos según alguna producción:

1. $S \Rightarrow (S)$: aplicamos $S \to (S)$
2. $(S) \Rightarrow (SS)$: aplicamos $S \to SS$
3. $(SS) \Rightarrow ((S)S)$: aplicamos $S \to (S)$ al primer S
4. $((S)S) \Rightarrow (()S)$: aplicamos $S \to \varepsilon$
5. $(()S) \Rightarrow (()(S))$: aplicamos $S \to (S)$
6. $(()(S)) \Rightarrow (()())$: aplicamos $S \to \varepsilon$

Observaciones:

Cada paso de la derivación preserva la propiedad de paréntesis balanceados. Además, esta gramática es **recursiva**: S aparece en el lado derecho de las producciones $S \to (S)$ y $S \to SS$, lo que permite generar cadenas arbitrariamente largas.

La producción $S \to \varepsilon$ actúa como **caso base**: sin una producción no recursiva, sería imposible eliminar todas las variables y obtener una cadena compuesta únicamente de símbolos terminales.

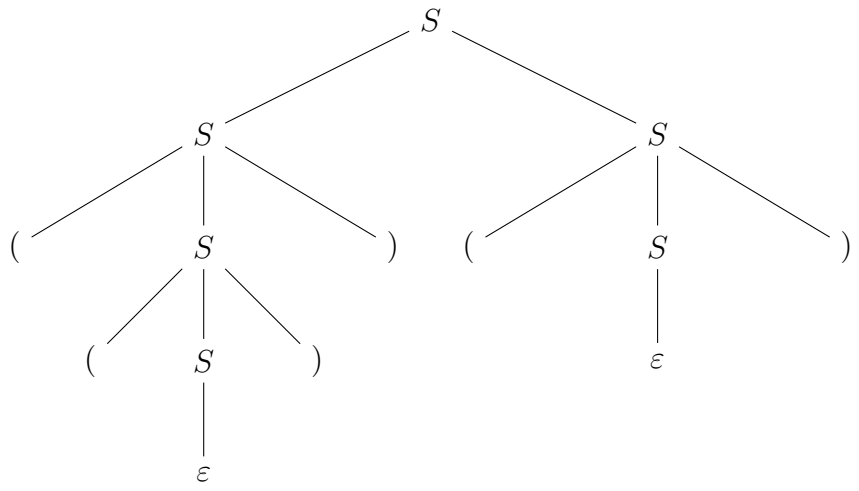
Árbol de derivación:

El proceso anterior puede representarse gráficamente mediante un **árbol de derivación** (o *parse tree*), que muestra cómo la cadena se organiza jerárquicamente según las reglas de la gramática.

Un **árbol de derivación** es una representación gráfica de una derivación con las siguientes propiedades:

- La **raíz** es el símbolo inicial S
- Los **nodos internos** son variables (no terminales)
- Las **hojas** son símbolos terminales o ε
- Cada nodo interno con etiqueta A y sus hijos representan la aplicación de una producción $A \to \alpha$
- La concatenación de las hojas (de izquierda a derecha) forma la cadena derivada

El árbol de derivación para la cadena (())() es:



Leyendo las hojas de izquierda a derecha (omitiendo ε): (())()

c. ¿Qué se puede reutilizar de soluciones pasadas?

Del capítulo anterior se reutiliza la idea general de un autómata que procesa una cadena símbolo a símbolo y decide si pertenece a un lenguaje. En este caso, se puede usar una pila como estructura de memoria auxiliar para "recordar" los delimitadores abiertos que aún no se han cerrado.

d. ¿Cuál es el diseño de la solución?

GLC para delimitadores balanceados $L_{\mathbf{del}}$

Extendiendo la gramática de paréntesis, la gramática que describe el lenguaje de delimitadores balanceados con paréntesis, corchetes y llaves es:

$$G_{\text{del}} = (\{S\}, \{(,), [,], \{, \}\}, P, S)$$

con producciones:

$$S \to \varepsilon \mid (S) \mid [S] \mid \{S\} \mid SS$$

Algoritmo de validación con pila

La estrategia de validación es:

- Por cada símbolo de apertura (`(`, `[`, {) que leemos: apilamos el símbolo
- Por cada símbolo de cierre (`)`, `]`, }) que leemos: verificamos que el tope de la pila sea el símbolo de apertura correspondiente y desapilamos (si la pila está vacía o no coincide, rechazamos)
- Al finalizar la entrada: si la pila está vacía, aceptamos

3.1.3 Ejemplos de pruebas

a. Ejemplo 1

Cadena: `([])`{}

- Lee `(`: apila. Pila: `[(`
- Lee `[`: apila. Pila: `[(, [`
- Lee `]`: tope es `[`, coincide. Desapila. Pila: `[(`
- Lee `)`: tope es `(`, coincide. Desapila. Pila: vacía
- Lee {: apila. Pila: `[{`
- Lee }: tope es {, coincide. Desapila. Pila: vacía

Resultado: `ACEPTADA` (pila vacía al final)

b. Ejemplo 2

Cadena: `([)`}

- Lee `(`: apila. Pila: `[(`
- Lee `[`: apila. Pila: `[(, [`
- Lee `)`: tope es `[`, no coincide con `)`

Resultado: `RECHAZADA` (cierre no corresponde con apertura)

c. Ejemplo 3

Cadena: `(({)}`

- Lee `(`: apila. Pila: `[(]`
- Lee `(`: apila. Pila: `[(, (]`
- Lee `{`: apila. Pila: `[(, (, {]`
- Lee `)`: tope es `{`, no coincide con `)`

Resultado: `RECHAZADA` (anidamiento incorrecto)

3.1.4 Codificación

a. Implementación para validación de delimitadores

La función principal sigue los pasos del diseño: lee la cadena, valida los delimitadores y muestra el resultado:

Código 3.1: Función principal

```
def main():
    cadena = leer("Ingrese los delimitadores: ")
    resultado = validar_delimitadores(cadena)
    imprimir_resultado(cadena, resultado)
```

Las funciones auxiliares de entrada y salida encapsulan la lectura desde la consola y la impresión del resultado (`ACEPTADA` o `RECHAZADA`):

Código 3.2: Funciones de entrada y salida

```
def leer(texto):
    return input(texto)

def imprimir_resultado(cadena, es_valido):
    if es_valido:
        print(f"'{cadena}': ACEPTADA")
    else:
        print(f"'{cadena}': RECHAZADA")
```

La función de validación implementa el algoritmo con pila: apila cada símbolo de apertura y, al encontrar un cierre, verifica que el tope de la pila sea la apertura correspondiente:

Código 3.3: Función de validación con pila

```
def validar_delimitadores(cadena):
    pila = []
    pares = {')': '(', ']': '[', '}': '{'}
    aperturas = set(pares.values())
    cierres = set(pares.keys())
    for simbolo in cadena:
        if simbolo in aperturas:
            pila.append(simbolo)
        elif simbolo in cierres:
            if len(pila) == 0 or pila[-1] != pares[simbolo]:
                return False
            pila.pop()
    return len(pila) == 0
```

b. Algoritmo general de reconocimiento con pila

El siguiente algoritmo genérico abstrae la lógica de validación para que pueda reutilizarse con distintos tipos de delimitadores. Recibe la cadena y funciones auxiliares que determinan qué símbolos son de apertura, cuáles de cierre y cómo deben coincidir:

Código 3.4: Función genérica de validación con pila

```
def validar_con_pila(cadena, es_apertura, es_cierre, coinciden):
    pila = []
    for simbolo in cadena:
        if es_apertura(simbolo):
            pila.append(simbolo)
        elif es_cierre(simbolo):
            if not pila or not coinciden(pila[-1], simbolo):
                return False
            pila.pop()
    return len(pila) == 0
```

3.2. Validación de expresiones matemáticas

Tras implementar la validación de delimitadores, el equipo de DevTools recibe una nueva solicitud: el editor debe validar que las expresiones matemáticas completas sean sintácticamente correctas. Ya no basta con verificar que los paréntesis estén balanceados; ahora hay que comprobar que operadores y operandos estén organizados de forma válida.

Por ejemplo, la expresión `(3 + (5 * 2))` es válida, pero `3 + * 2` es incorrecta porque contiene dos operadores consecutivos sin un valor entre ellos. De igual forma, `(3 + 5 * 2))` es incorrecta por un paréntesis sobrante. El validador con pila del problema anterior no detecta este tipo de errores; por ello, es necesario un enfoque que analice la estructura completa de la expresión.

3.2.1 Análisis / Abstracción

a. ¿Cuál es el problema a resolver?

DevTools necesita extender su editor para validar la estructura sintáctica completa de expresiones matemáticas con números, operadores (+, *) y paréntesis. El validador de delimitadores del problema anterior es insuficiente: expresiones como `3 + * 2` pasan la validación con pila (no tienen delimitadores desbalanceados) pero no son expresiones válidas. Se requiere un mecanismo que verifique simultáneamente el balanceo de paréntesis y la correcta alternancia entre operandos y operadores.

b. ¿Cuál es la información necesaria?

Cada expresión escrita en el editor se entrega completa al validador, junto con la lista de componentes que el lenguaje admite:

Información necesaria	Descripción
Expresión matemática	El texto a verificar
Componentes válidos	Números, operadores (+, *) y paréntesis

La validez depende de la estructura completa de la expresión: los operadores deben estar entre operandos, los paréntesis deben delimitar subexpresiones válidas, y la precedencia de operadores debe ser respetada.

3.2.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **ENTRADA:** Una cadena que representa una expresión matemática.
- **PROCESAMIENTO:** Verificar que la expresión tenga una estructura sintáctica válida: operandos y operadores correctamente alternados, paréntesis balanceados y precedencia respetada.
- **SALIDA:** Indicar si la expresión es sintácticamente válida o no.

b. ¿Qué conocimiento adicional se requiere?

Este problema requiere dos conceptos nuevos. Primero, una gramática con **múltiples variables** que capture la precedencia de operadores: la multiplicación debe evaluarse antes que la suma, lo que se modela usando variables distintas para expresiones (E), términos (T) y factores (F). Segundo, un mecanismo para convertir una gramática en un algoritmo de validación, lo que lleva al estudio del **orden de derivación** y al **descenso recursivo**.

Ejemplo 2: GLC para expresiones aritméticas simples

Consideremos el lenguaje de expresiones aritméticas sobre identificadores (id) con suma y multiplicación.

Ejemplos de cadenas válidas: id, $id + id$, $id * id + id$

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & id \mid (E) \end{array}$$

Componentes:

- $V = \{E, T, F\}$ (Expresión, Término, Factor)
- $\Sigma = \{id, +, *, (,)\}$
- $S = E$ (símbolo inicial)

Interpretación:

- E representa una expresión completa (puede contener sumas)
- T representa un término (puede contener multiplicaciones)
- F representa un factor (identificador o expresión entre paréntesis)
- Esta estructura refleja la precedencia de operadores: $*$ se evalúa antes que $+$

Ejemplo de derivación:

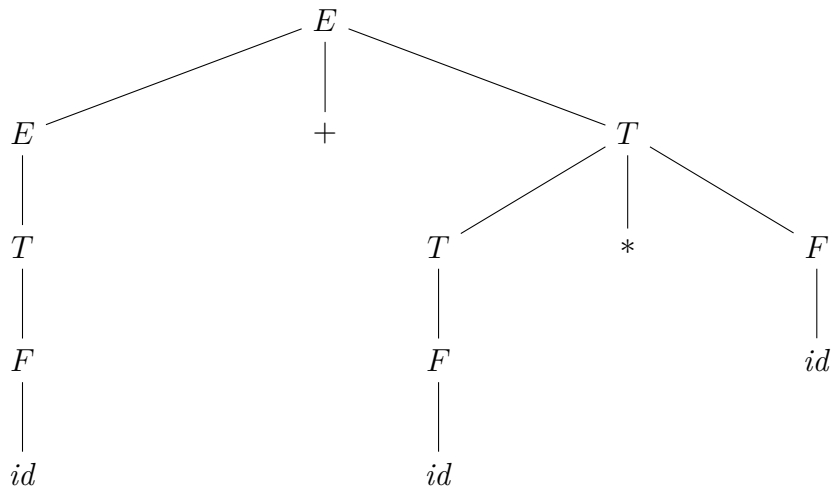
Derivemos la cadena $id + id * id$:

$$\begin{aligned} E &\Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow id + T \\ &\Rightarrow id + T * F \Rightarrow id + F * F \Rightarrow id + id * F \Rightarrow id + id * id \end{aligned}$$

Observe cómo la estructura de la gramática captura la precedencia: la multiplicación queda agrupada en T, mientras que la suma conecta expresiones E y términos T.

Árbol de derivación:

El árbol de derivación para la cadena $id + id * id$ es:



Observación: Este árbol refleja que la multiplicación se evalúa antes que la suma (mayor profundidad en el árbol = mayor precedencia).

c. ¿Qué se puede reutilizar de soluciones pasadas?

Del problema anterior se reutilizan los conceptos de gramática libre de contexto, derivación y árbol de derivación: la gramática de expresiones se define con la misma notación y las derivaciones siguen el mismo proceso. La estrategia de validación con pila, en cambio, no es suficiente aquí: una pila puede verificar el balanceo de paréntesis, pero no puede validar que los operadores estén correctamente ubicados entre operandos (por ejemplo, `3 + * 2` no tiene problemas de delimitadores pero es inválida).

d. ¿Cuál es el diseño de la solución?

GLC para expresiones aritméticas $L_{\mathbf{expr}}$

La gramática que describe el lenguaje de expresiones aritméticas con paréntesis, suma y multiplicación es:

$$G_{\text{expr}} = (\{E, T, F\}, \{id, +, *, (,)\}, P, E)$$

con producciones:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & id \mid (E) \end{array}$$

Esta gramática captura tanto el balanceo de paréntesis (mediante la producción $F \rightarrow (E)$) como la precedencia de operadores: la multiplicación se agrupa en T antes que la suma en E.

Orden de derivación

En el ejemplo anterior, las derivaciones se realizaban eligiendo libremente qué variable reemplazar en cada paso. Sin embargo, al diseñar un algoritmo de validación, es necesario fijar un orden sistemático. Se utilizan dos formas estándar de derivación:

- Una **derivación izquierda** reemplaza siempre el no terminal más a la izquierda en cada paso. Se denota $\Rightarrow_{\text{iz}}$.
- Una **derivación derecha** reemplaza siempre el no terminal más a la derecha en cada paso. Se denota $\Rightarrow_{\text{der}}$.

Ejemplo 3: Derivaciones izquierda y derecha

Consideremos una gramática simplificada de expresiones:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow id \end{aligned}$$

Para la cadena $id + id + id$:

Derivación izquierda:

$$\begin{aligned} E &\Rightarrow_{\text{iz}} E + T \\ &\Rightarrow_{\text{iz}} E + T + T \\ &\Rightarrow_{\text{iz}} T + T + T \\ &\Rightarrow_{\text{iz}} id + T + T \\ &\Rightarrow_{\text{iz}} id + id + T \\ &\Rightarrow_{\text{iz}} id + id + id \end{aligned}$$

Derivación derecha:

$$\begin{aligned} E &\Rightarrow_{\text{der}} E + T \\ &\Rightarrow_{\text{der}} E + id \\ &\Rightarrow_{\text{der}} E + T + id \\ &\Rightarrow_{\text{der}} E + id + id \\ &\Rightarrow_{\text{der}} T + id + id \\ &\Rightarrow_{\text{der}} id + id + id \end{aligned}$$

Ambas derivaciones generan la misma cadena, pero aplican las producciones en órdenes diferentes. En la derivación izquierda, en cada paso se expande la variable no terminal ubicada más a la izquierda.

Esto tiene implicaciones importantes en la implementación de analizadores sintácticos. Un algoritmo basado en derivación izquierda intentaría aplicar la producción E→E+T, produciendo nuevamente un E al inicio de la expresión, lo que puede repetirse indefinidamente. A este fenómeno se le conoce como **recursión izquierda**.

Algoritmo de validación por descenso recursivo

Un **parser por descenso recursivo** es un algoritmo que sigue una derivación izquierda: para cada variable, intenta aplicar sus producciones de izquierda a derecha. Como se observó en el ejemplo anterior, la gramática $E \to E + T$ es **recursiva por la izquierda**, lo que causa un ciclo infinito en este tipo de parser. Sin embargo, podemos reescribirla de forma equivalente eliminando la recursión izquierda mediante nuevas variables (E', T'):

$$\begin{array}{rcl} E & \to & T\,E' \\ E' & \to & +\,T\,E' \mid \varepsilon \\ T & \to & F\,T' \\ T' & \to & *\,F\,T' \mid \varepsilon \\ F & \to & id \mid (E) \end{array}$$

La idea es transformar la recursión izquierda ($E \to E + T$) en recursión derecha ($E' \to +\,T\,E'$), que el parser puede manejar sin entrar en ciclo. La producción $E' \to \varepsilon$ actúa como caso base, indicando que no hay más sumas. El mismo patrón se aplica a T y T' para la multiplicación.

Esta gramática puede implementarse mediante un **parser por descenso recursivo**, donde cada variable de la gramática se representa mediante una función encargada de procesar una parte de la entrada y llamar a las funciones asociadas a las demás variables.

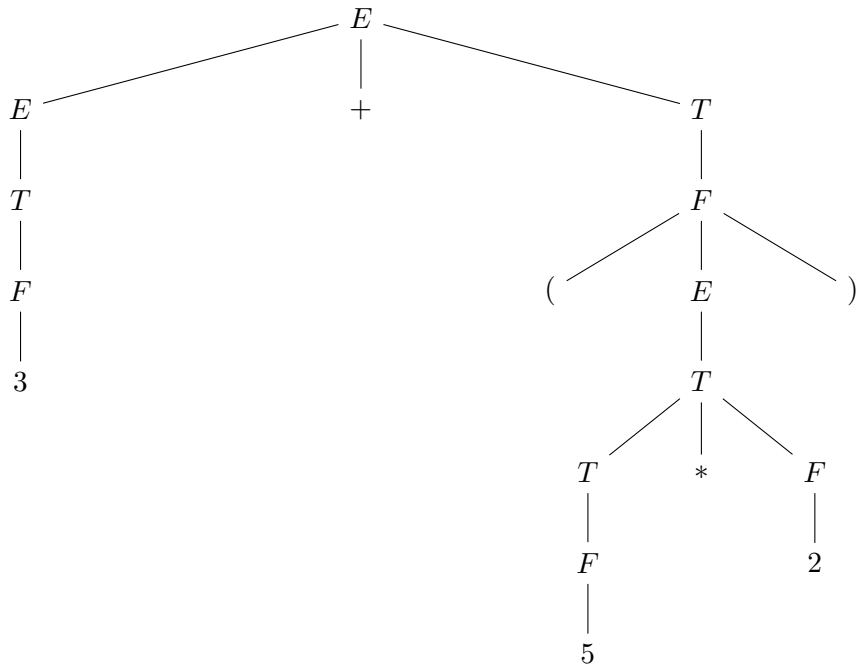
3.2.3 Ejemplos de pruebas

Los siguientes ejemplos verifican si las cadenas pertenecen al lenguaje generado por G_{expr}, mostrando la derivación y el árbol de derivación correspondiente. En los casos de rechazo, se muestra por qué no es posible completar la derivación.

a. Ejemplo 1

Cadena: $3 + (5 * 2)$ **Resultado:** `ACEPTADA`

Árbol de derivación:



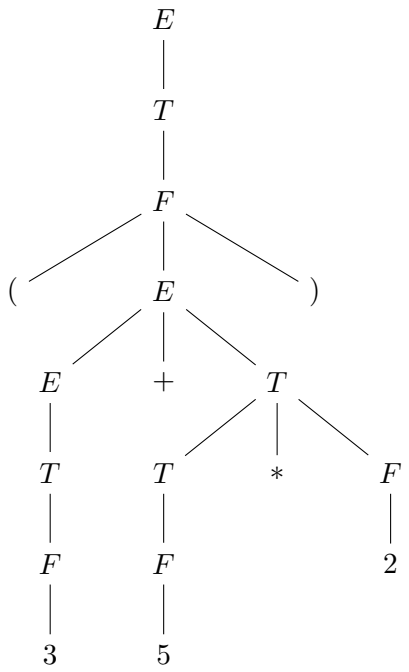
La cadena se deriva completamente desde E. El árbol muestra que la multiplicación $5*2$ queda agrupada dentro de T, y los paréntesis se generan mediante la producción $F \to (E)$.

b. Ejemplo 2

Cadena: $(3 + 5 * 2))$ **Resultado:** `RECHAZADA`

Árbol de derivación:

La subcadena $(3 + 5 * 2)$ es derivable. El árbol de derivación para esta parte es:

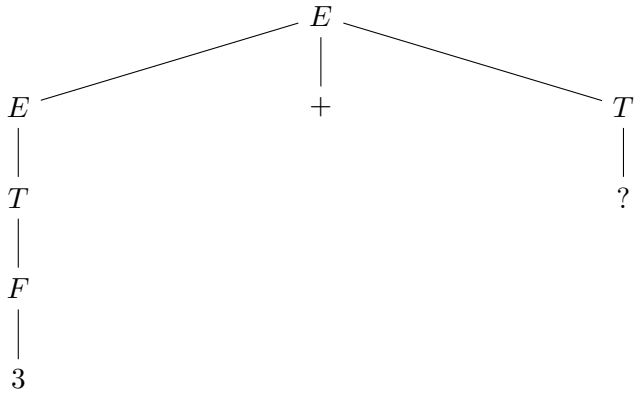


El árbol consume $(3 + 5 * 2)$ y la derivación se completa. Sin embargo, queda un segundo) en la entrada que no puede ser generado por ninguna producción de la gramática: no existe forma de derivar un) suelto después de una expresión completa. La cadena no pertenece a $L(G_{\text{expr}})$.

c. Ejemplo 3

Cadena: $3 + * 2$ **Resultado:** RECHAZADA

Intentemos construir el árbol de derivación:



Después de derivar 3+, el nodo T debe generar $*2$. Si aplicamos $T \to T * F$, el primer T debería derivar alguna cadena antes del $*$, pero no hay ningún operando en esa posición. Si aplicamos $T \to F$, entonces F debería derivar $*2$, pero F solo puede producir un identificador o (E). El árbol no se puede completar: el operador $*$ aparece donde la gramática exige un operando.

3.2.4 Codificación

a. Parser por descenso recursivo

La implementación refleja la estructura de la gramática sin recursión izquierda: cada variable (E, E', T, T', F) se representa mediante un método del parser.

Los métodos `expresion_prima()` y `termino_primo()` implementan la recursión derecha de E' y T', respectivamente. Cuando encuentran el operador correspondiente, procesan el siguiente operando y vuelven a llamarse de manera recursiva; en caso contrario, retornan `True`, representando la producción ε.

Finalmente, la función `main` se encarga de leer la expresión, validarla y mostrar el resultado obtenido.

Código 3.5: Función principal y E/S

```
def main():
    cadena = leer("Ingrese la expresion: ")
    resultado = validar_expresion(cadena)
    imprimir_resultado(cadena, resultado)

def leer(texto):
    return input(texto)

def imprimir_resultado(cadena, es_valido):
    if es_valido:
        print(f"'{cadena}': ACEPTADA")
    else:
        print(f"'{cadena}': RECHAZADA")
```

Antes de invocar al parser, la función `tokenizar` recorre la cadena carácter a carácter para separar números, operadores y paréntesis.

Código 3.6: Tokenizador y función de validación

```
def tokenizar(cadena):
    tokens = []
    i = 0
    while i < len(cadena):
        if cadena[i].isspace():
            i += 1
        elif cadena[i].isdigit():
            j = i
            while j < len(cadena) and cadena[j].isdigit():
                j += 1
            tokens.append(cadena[i:j])
            i = j
        elif cadena[i] in "+-*/()":
            tokens.append(cadena[i])
            i += 1
        else:
            return None
    return tokens

def validar_expresion(cadena):
    tokens = tokenizar(cadena)
    if tokens is None:
        return False
    parser = Parser(tokens)
    return parser.expresion() and parser.fin()
```

A continuación, las cinco funciones del parser, una por producción, intentan reconocer la estructura esperada y retornan `True` cuando lo logran.

Código 3.7: Clase Parser --- descenso recursivo

```
class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.pos = 0

    def actual(self):
        if self.pos < len(self.tokens):
            return self.tokens[self.pos]
        return None

    def consumir(self, esperado):
        if self.actual() == esperado:
            self.pos += 1
            return True
        return False

    def fin(self):
        return self.pos == len(self.tokens)

    # E -> T E'
    def expresion(self):
        if not self.termino():
            return False
        return self.expresion_prima()

    # E' -> + T E' | epsilon
    def expresion_prima(self):
        if self.actual() == '+':
            self.pos += 1
            if not self.termino():
                return False
            return self.expresion_prima()
        return True

    # T -> F T'
    def termino(self):
        if not self.factor():
            return False
        return self.termino_primo()

    # T' -> * F T' | epsilon
    def termino_primo(self):
        if self.actual() == '*':
            self.pos += 1
            if not self.factor():
                return False
            return self.termino_primo()
        return True

    # F -> id | ( E )
    def factor(self):
        if self.actual() == '(':
            self.pos += 1
            if not self.expresion():
                return False
            return self.consumir(')')
        elif self.actual() and self.actual()[0].isdigit():
            self.pos += 1
            return True
        return False
```

3.3. Validación de JSON

La empresa DataLink, dedicada al desarrollo de servicios web, está construyendo una API REST que recibe documentos JSON [Bray, 2017] de múltiples clientes. Antes de procesar los datos, el sistema debe validar que la estructura del JSON sea sintácticamente correcta: objetos con llaves balanceadas, arrays con corchetes, pares clave-valor separados por dos puntos, y elementos separados por comas.

El equipo necesita especificar formalmente la sintaxis de JSON para construir el validador.

3.3.1 Análisis / Abstracción

a. ¿Cuál es el problema a resolver?

DataLink necesita un mecanismo formal para especificar la sintaxis completa de documentos JSON y verificar automáticamente que un documento dado es sintácticamente correcto. Los documentos JSON tienen estructura recursiva: un valor puede ser un objeto que contiene otros valores, incluyendo otros objetos y arrays.

b. ¿Cuál es la información necesaria?

Para construir el validador hace falta tanto el documento concreto que llega a la API como el conjunto de construcciones que define la sintaxis JSON:

Información necesaria	Descripción
Documento JSON	El texto a verificar
Estructura de JSON	Objetos, arrays, pares clave-valor, valores primitivos

3.3.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **ENTRADA:** Un texto que representa un documento JSON.
- **PROCESAMIENTO:** Verificar que el documento cumple con las reglas sintácticas de JSON: objetos, arrays, pares clave-valor y valores primitivos correctamente formados y anidados.
- **SALIDA:** Indicar si el documento es sintácticamente válido o no.

b. ¿Qué conocimiento adicional se requiere?

Las gramáticas libres de contexto permiten describir estructuras complejas, pero su notación formal puede volverse extensa y difícil de leer en ejemplos grandes. Por esta razón, suelen utilizarse formas más prácticas de representación, como BNF y su extensión EBNF.

Notación BNF (Backus-Naur Form)

La **Backus-Naur Form** (BNF) es una notación formal utilizada para describir gramáticas libres de contexto [Backus et al., 1960], especialmente en la definición de lenguajes de programación. Su principal objetivo es representar las producciones gramaticales de una manera más clara y legible.

En la notación BNF estándar:

- Los **no terminales** se escriben entre ángulos: ⟨expresión⟩, ⟨término⟩
- Los **terminales** se escriben exactamente como aparecen: `+`, `*`, `id`, `if`
- El símbolo **::=** se usa en lugar de → para indicar producciones
- El símbolo | indica alternativas (o)
- Las producciones múltiples para un mismo no terminal se pueden combinar

Ejemplo 4: Expresiones aritméticas en BNF y EBNF

La gramática de expresiones aritméticas se puede escribir en BNF como:

```
<expresión> ::= <expresión> + <término> | <término>
<término>   ::= <término> * <factor> | <factor>
<factor>    ::= id | ( <expresión> )
```

Esto es equivalente a la GLC:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow id \mid (E) \end{aligned}$$

Hay una correspondencia directa entre las producciones en BNF y las reglas de la gramática libre de contexto, pero BNF es más legible para describir lenguajes complejos.

Nótese que esta gramática usa **recursión** para expresar repetición: `<expresión> ::= <expresión> + <término>` permite encadenar sumas, pero la recursión hace la gramática verbosa. Para estos casos existe una extensión más concisa.

Extensiones de BNF: EBNF

La **Extended BNF** (EBNF), formalizada por **Niklaus Wirth** [Wirth, 1977], introduce operadores adicionales para mayor expresividad:

- **Opcionalidad** $[\alpha]$: el elemento α es opcional (0 o 1 vez)
- **Repetición** $\{\alpha\}$: el elemento α se repite 0 o más veces
- **Agrupación** (α): agrupa elementos

Equivalencia entre notaciones

Concepto	**BNF**	**EBNF**
Opcional	$A ::= B \mid \varepsilon$	$A ::= [B]$
Repetición (0+)	$A ::= BA \mid \varepsilon$	$A ::= \{B\}$
Repetición (1+)	$A ::= B \mid AB$	$A ::= B\{B\}$
Alternativas	$A ::= B \mid C \mid D$	$A ::= B \mid C \mid D$

Nota importante: Aunque EBNF es más concisa, cualquier gramática en EBNF puede ser convertida a una gramática BNF estándar (y por tanto, a una GLC formal).

Con EBNF, la gramática de expresiones aritméticas se reescribe eliminando la recursión explícita:

```
<expresión> ::= <término> { "+" <término> }
<término>   ::= <factor> { "*" <factor> }
<factor>    ::= id | "(" <expresión> ")"
```

El operador {} captura directamente el patrón de repetición que en BNF requería producciones recursivas. Ambas gramáticas generan el mismo lenguaje.

Hasta ahora se ha usado BNF para describir expresiones aritméticas. Sin embargo, esta notación puede describir la sintaxis de formatos muy diferentes. Antes de aplicarla a JSON, veamos cómo se usa para especificar archivos de configuración, un formato que también se basa en pares clave-valor y bloques anidados.

Ejemplo 5: Parser para configuración

Se desea diseñar un lenguaje simple para archivos de configuración que soporte asignaciones, bloques y comentarios. La representación de su gramática en EBNF es:

```
<config> ::= { <elemento> }

<elemento> ::= <asignacion> | <seccion> | <comentario>

<asignacion> ::= <id> "=" <valor>

<seccion> ::= <id> "{" <config> "}"

<valor> ::= <numero> | <cadena> | <id>

<comentario> ::= "#" <texto> <nuevalinea>
```

Ejemplo de derivación:

Para el archivo de configuración:

```
servidor {
    puerto = 8080
}
```

Se tiene la siguiente derivación:

```
<config>
  ::= { <elemento> }
  ::= <elemento>
  ::= <seccion>
  ::= <id> "{" <config> "}"
  ::= servidor "{" <config> "}"
  ::= servidor "{" { <elemento> } "}"
  ::= servidor "{" <elemento> "}"
  ::= servidor "{" <asignacion> "}"
  ::= servidor "{" <id> "=" <valor> "}"
  ::= servidor "{" puerto "=" <numero> "}"
  ::= servidor "{" puerto "=" 8080 "}"
```

La similitud estructural con JSON es notable: ambos formatos usan pares clave-valor y permiten anidamiento. La gramática BNF para JSON seguirá un patrón similar, como se verá en el diseño de la solución.

c. ¿Qué se puede reutilizar de soluciones pasadas?

De los problemas anteriores se reutilizan los conceptos de gramática libre de contexto, derivación y árbol de derivación.

d. ¿Cuál es el diseño de la solución?

Gramática BNF para JSON

La sintaxis de JSON puede especificarse mediante la siguiente gramática en BNF:

```
<json> ::= <objeto> | <array> | <valor>

<objeto> ::= "{" <pares> "}" | "{" "}"

<pares> ::= <par> | <par> "," <pares>

<par> ::= <cadena> ":" <json>

<array> ::= "[" <elementos> "]" | "[" "]"

<elementos> ::= <json> | <json> "," <elementos>

<valor> ::= <numero> | <cadena> | "true" | "false" | "null"
```

Ejemplo de derivación:

Para el JSON: {"x": [1, 2]}

```
<json>
  ::= <objeto>
  ::= "{" <pares> "}"
  ::= "{" <par> "}"
  ::= "{" <cadena> ":" <json> "}"
  ::= "{" "x" ":" <array> "}"
  ::= "{" "x" ":" "[" <elementos> "]" "}"
  ::= "{" "x" ":" "[" <json> "," <elementos> "]" "}"
  ::= "{" "x" ":" "[" <numero> "," <json> "]" "}"
  ::= "{" "x" ":" "[" 1 "," <numero> "]" "}"
  ::= "{" "x" ":" "[" 1 "," 2 "]" "}"
```

Esta especificación formal documenta la sintaxis esperada, sirve de base para construir validadores que verifiquen la estructura y permite generar parsers automáticamente con herramientas como ANTLR.

3.3.3 Ejemplos de pruebas

a. Ejemplo 1

Documento: {"nombre": "Ana"}

Derivación paso a paso:

```
<json> ::= <objeto> ::= "{" <pares> "}"
  ::= "{" <par> "}" ::= "{" <cadena> ":" <json> "}"
  ::= "{" "nombre" ":" <valor> "}"
  ::= "{" "nombre" ":" <cadena> "}"
  ::= "{" "nombre" ":" "Ana" "}"
```

Resultado: ACEPTADA

b. Ejemplo 2

Documento: {"a": [1, {"b": 2}]}

Derivación paso a paso:

```
<json> ::= <objeto> ::= "{" <pares> "}"
  ::= "{" <par> "}" ::= "{" <cadena> ":" <json> "}"
  ::= "{" "a" ":" <array> "}"
  ::= "{" "a" ":" "[" <elementos> "]" "}"
  ::= "{" "a" ":" "[" <json> "," <elementos> "]" "}"
  ::= "{" "a" ":" "[" 1 "," <json> "]" "}"
  ::= "{" "a" ":" "[" 1 "," <objeto> "]" "}"
  ::= "{" "a" ":" "[" 1 "," "{" "b" ":" 2 "}" "]" "}"
```

Resultado: ACEPTADA (anidamiento de objetos dentro de arrays).

c. Ejemplo 3

Documento: {"nombre": "Ana", }

Intento de derivación:

```
<json> ::= <objeto> ::= "{" <pares> "}"
  ::= "{" <par> "," <pares> "}"
  ::= "{" <cadena> ":" <valor> "," <pares> "}"
  ::= "{" "nombre" ":" "Ana" "," <pares> "}"
  ::= "{" "nombre" ":" "Ana" "," ??? "}"
```

Después de la coma, la gramática espera un <par>, es decir, una cadena seguida de ":" y un valor. Sin embargo, se encuentra directamente el cierre }. Ninguna regla de producción permite derivar un par vacío, por lo que la derivación falla.

Resultado: RECHAZADA (coma final sin par clave-valor).

3.3.4 Codificación

a. Implementación en Python

La validación de JSON se implementa mediante un parser recursivo descendente, donde cada regla de la gramática BNF se traduce en una función del parser. La clase `ParserJSON` sigue esta correspondencia: cada método (`parsear_valor`, `parsear_objeto`, `parsear_array`, etc.) corresponde a un no terminal de la gramática.

`main` recibe el documento, instancia el parser e informa el resultado del análisis.

Código 3.8: Función principal

```
def main():
    texto = leer("Ingrese JSON: ")
    parser = ParserJSON(texto)
    resultado = parser.parsear()
    imprimir_resultado(texto, resultado)
```

Como en los ejercicios anteriores, dos funciones auxiliares concentran la interacción con la consola: lectura del texto JSON e impresión del fallo o aceptación.

Código 3.9: Funciones de entrada y salida

```
def leer(texto):
    return input(texto)

def imprimir_resultado(texto, es_valido):
    if es_valido:
        print(f"'{texto}': ACEPTADA (JSON valido)")
    else:
        print(f"'{texto}': RECHAZADA (JSON invalido)")
```

La clase `ParserJSON` comienza con su infraestructura básica: `saltar_espacios` ignora caracteres en blanco, `caracter_actual` consulta el siguiente carácter significativo y `consumir` verifica y avanza sobre un carácter esperado.

Código 3.10: `Clase ParserJSON: infraestructura`

```
16 class ParserJSON:
17     def __init__(self, texto):
18         self.texto = texto
19         self.pos = 0
20 
21     def saltar_espacios(self):
22         while self.pos < len(self.texto) and self.texto[self.pos] in ' \t\n
               \r':
23             self.pos += 1
24 
25     def caracter_actual(self):
26         self.saltar_espacios()
27         if self.pos < len(self.texto):
28             return self.texto[self.pos]
29         return None
30 
31     def consumir(self, esperado):
32         self.saltar_espacios()
33         if self.pos < len(self.texto) and self.texto[self.pos] == esperado:
34             self.pos += 1
35             return True
36         return False
```

El método `parsear` inicia el análisis y comprueba que se consuma toda la entrada; `parsear_valor` actúa como despachador, delegando a la función adecuada (objeto, array, cadena, número o literal) según el carácter actual.

Código 3.11: Clase ParserJSON: parsear y parsear_valor

```
38      def parsear(self):
39          resultado = self.parsear_valor()
40          self.saltar_espacios()
41          return resultado and self.pos == len(self.texto)
42
43      def parsear_valor(self):
44          c = self.caracter_actual()
45          if c == '{':
46              return self.parsear_objeto()
47          elif c == '[':
48              return self.parsear_array()
49          elif c == '"':
50              return self.parsear_cadena()
51          elif c is not None and (c.isdigit() or c == '-'):
52              return self.parsear_numero()
53          elif self.texto[self.pos:self.pos+4] == 'true':
54              self.pos += 4
55              return True
56          elif self.texto[self.pos:self.pos+5] == 'false':
57              self.pos += 5
58              return True
59          elif self.texto[self.pos:self.pos+4] == 'null':
60              self.pos += 4
61              return True
62          return False
```

Para las estructuras compuestas hay dos métodos: uno reconoce objetos como secuencias de pares clave-valor separados por comas, y otro hace lo análogo con arrays.

Código 3.12: Clase ParserJSON: objetos y arrays

```
64      def parsear_objeto(self):
65          if not self.consumir('{'):
66              return False
67          if self.caracter_actual() == '}':
68              self.pos += 1
69              return True
70          if not self.parsear_par():
71              return False
72          while self.consumir(','):
73              if not self.parsear_par():
74                  return False
75          return self.consumir('}')
76
77      def parsear_par(self):
78          if not self.parsear_cadena():
79              return False
80          if not self.consumir(':'):
81              return False
82          return self.parsear_valor()
83
84      def parsear_array(self):
85          if not self.consumir('['):
86              return False
87          if self.caracter_actual() == ']':
88              self.pos += 1
89              return True
90          if not self.parsear_valor():
91              return False
92          while self.consumir(','):
93              if not self.parsear_valor():
94                  return False
95          return self.consumir(']')
```

Por último, los valores primitivos quedan a cargo de dos métodos especializados: cadenas con comillas y caracteres escapados, y números con signo opcional.

Código 3.13: Clase ParserJSON: cadenas y números

```
97      def parsear_cadena(self):
98          if not self.consumir('"'):
99              return False
100         while self.pos < len(self.texto) and self.texto[self.pos] != '"':
101             if self.texto[self.pos] == '\\':
102                 self.pos += 1
103             self.pos += 1
104         return self.consumir('"')
105
106     def parsear_numero(self):
107         inicio = self.pos
108         if self.pos < len(self.texto) and self.texto[self.pos] == '-':
109             self.pos += 1
110         if self.pos >= len(self.texto) or not self.texto[self.pos].isdigit
                ():
111             return False
112         while self.pos < len(self.texto) and self.texto[self.pos].isdigit()
                :
113             self.pos += 1
114         return self.pos > inicio
```

3.4. Diseño de un mini-lenguaje

La startup EduCode está desarrollando un mini-lenguaje imperativo para enseñar programación a estudiantes de secundaria. El lenguaje debe soportar declaraciones de variables, asignaciones, ciclos `while`, condicionales `if-else` y expresiones aritméticas. El equipo necesita construir un **analizador sintáctico** (parser) que verifique que los programas escritos por los estudiantes tengan una estructura sintáctica correcta antes de intentar ejecutarlos.

A diferencia de los validadores vistos en las secciones anteriores (delimitadores, expresiones, JSON), este problema involucra múltiples tipos de sentencias y estructuras de control anidadas, lo que abre dos preguntas: ¿cómo especificar formalmente todas las construcciones del lenguaje? ¿Qué ocurre cuando una especificación permite múltiples interpretaciones de la misma entrada?

3.4.1 Análisis / Abstracción

a. ¿Cuál es el problema a resolver?

EduCode necesita un analizador sintáctico que reciba un programa escrito en el mini-lenguaje y determine si su estructura es válida. El analizador debe verificar que las declaraciones, asignaciones, ciclos y condicionales estén correctamente formados y anidados, rechazando programas con errores sintácticos antes de que se intenten ejecutar.

b. ¿Cuál es la información necesaria?

El analizador necesita tres referencias: el programa que escribe el estudiante, las construcciones que el mini-lenguaje permite y los elementos básicos sobre los que esas construcciones se forman:

Información necesaria	Descripción
Código fuente del programa	El texto a verificar
Construcciones del lenguaje	Declaraciones, asignaciones, ciclos, condicionales
Elementos básicos	Variables, números, operadores, delimitadores

Un programa válido del mini-lenguaje es una secuencia de sentencias, donde cada sentencia puede ser una declaración (`var x;`), una asignación (`x = 5;`), un ciclo (`while (...) {...}`) o un condicional (`if (...) {...} else {...}`).

3.4.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **ENTRADA:** El código fuente de un programa en el mini-lenguaje.
- **PROCESAMIENTO:** Verificar que el programa tenga una estructura sintáctica válida: declaraciones, asignaciones, ciclos y condicionales correctamente formados y anidados.
- **SALIDA:** Indicar si el programa es sintácticamente válido o no.

b. ¿Qué conocimiento adicional se requiere?

El diseño requiere conocer los tipos de analizadores sintácticos disponibles, el problema de la ambigüedad gramatical y la forma en que las gramáticas libres de contexto permiten definir la sintaxis de dominios muy diversos.

Tipos de parsers

En la sección anterior se implementó un parser recursivo descendente para validar JSON, uno de varios tipos de analizadores sintácticos disponibles. Los **parsers descendentes** (top-down) parten del símbolo inicial y expanden producciones hacia las hojas; los recursivos descendentes y los LL(k) son los ejemplos más comunes y los más fáciles de implementar manualmente. Los **parsers ascendentes** (bottom-up) recorren el camino inverso: parten de los terminales y reducen hacia el símbolo inicial. A esta familia pertenecen los LR(k) [Knuth, 1965], LALR y SLR usados por compiladores profesionales como Yacc y Bison [Johnson, 1975]. Finalmente, los **parsers universales** como CYK [Kasami, 1966, Younger, 1967] y Earley [Earley, 1970] aceptan cualquier gramática libre de contexto pero resultan menos eficientes para gramáticas comunes.

En este problema se utilizará nuevamente un parser recursivo

descendente, ya que cada regla de la gramática se traduce directamente en una función del programa.

El mini-lenguaje de EduCode debe combinar varias construcciones: declaraciones, asignaciones, estructuras de control y expresiones. Para apreciar cómo una gramática puede mezclar texto con estructuras de control anidadas, consideremos primero un dominio donde esto es central: los lenguajes de plantillas.

Ejemplo 6: Plantillas/Templates

Los lenguajes de plantillas (como Mustache, Jinja2, Handlebars) permiten generar HTML dinámico mezclando texto estático con variables y estructuras de control:

```
<template> ::= { <elem> }

<elem> ::= <texto> | <variable> | <bucle>

<variable> ::= "{{" <id> "}}"

<bucle> ::= "{% for" <id> "in" <id> "%}"
            { <elem> }
            "{% endfor %}"
```

Ejemplo de derivación:

Para la plantilla `<h1>{{ titulo }}</h1>`:

```
<template>
  ::= { <elem> }
  ::= <elem> <elem> <elem>
  ::= <texto> <elem> <elem>
  ::= "<h1>" <elem> <elem>
  ::= "<h1>" <variable> <elem>
  ::= "<h1>" "{{" <id> "}}" <elem>
  ::= "<h1>" "{{" titulo "}}" <elem>
  ::= "<h1>" "{{" titulo "}}" <texto>
  ::= "<h1>" "{{" titulo "}}" "</h1>"
```

Este ejemplo ilustra un patrón que aparecerá en el mini-lenguaje: la gramática mezcla elementos de distinto tipo (texto, variables, bucles) y permite anidamiento mediante repetición ({⟨elem⟩}). De manera similar, el mini-lenguaje necesitará combinar declaraciones, asignaciones y estructuras de control dentro de bloques anidados. Sin embargo, al diseñar gramáticas con múltiples alternativas y anidamiento, surge un problema importante: la ambigüedad.

Ambigüedad

El diseño de la gramática del mini-lenguaje debe garantizar que cada programa tenga una única interpretación posible.

Una gramática es **ambigua** [Aho et al., 2006] si existe al menos una cadena en $L(G)$ que tiene dos o más árboles de derivación distintos (o equivalentemente, dos derivaciones izquierdas distintas, o dos derivaciones derechas distintas).
Un lenguaje es **inherentemente ambiguo** si toda gramática que lo genera es ambigua. La existencia de lenguajes inherentemente ambiguos fue demostrada por **Parikh** [Parikh, 1966].

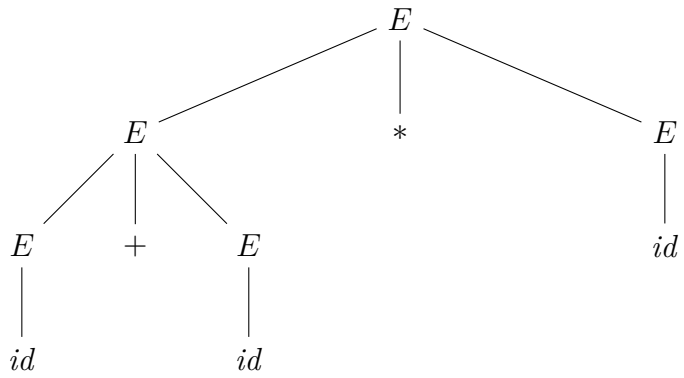
Ejemplo 7: Gramática ambigua para expresiones

Consideremos una gramática simple para expresiones:

$$E \to E + E \mid E * E \mid id$$

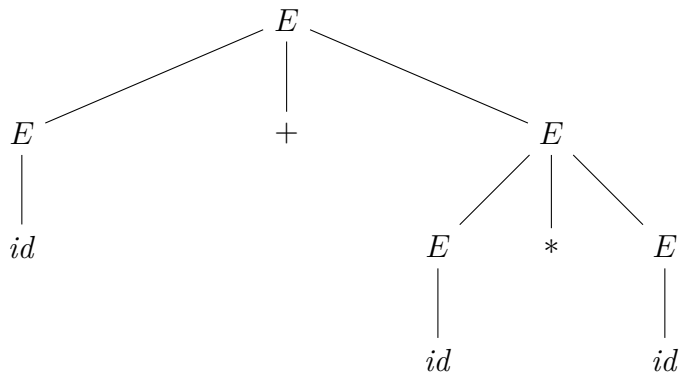
La cadena $id + id * id$ tiene dos árboles de derivación:

Árbol 1: Suma primero



Este árbol representa $(id + id) * id$

Árbol 2: Multiplicación primero



Este árbol representa $id + (id * id)$

Estos dos árboles representan diferentes interpretaciones semánticas. La gramática es **ambigua**.

Ejemplo 8: Problema del if-then-else colgante

Una de las ambigüedades más conocidas en lenguajes de programación:

```
<sentencia> ::= if <condición> then <sentencia>
        | if <condición> then <sentencia> else <sentencia>
        | otra-sentencia
```

Consideremos la cadena:

```
if c1 then if c2 then s1 else s2
```

Esta cadena tiene dos interpretaciones:

Interpretación 1: El `else` se asocia con el segundo `if`

```
if c1 then (if c2 then s1 else s2)
```

Interpretación 2: El `else` se asocia con el primer `if`

```
if c1 then (if c2 then s1) else s2
```

Solución: La mayoría de lenguajes resuelven esto con la regla: "el `else` se asocia con el `if` más cercano". En nuestro mini-lenguaje evitamos este problema al requerir llaves obligatorias en los bloques: `if (...) {...} else {...}`.

Estrategias para eliminar ambigüedad

1. **Reescribir la gramática**: Usar niveles de precedencia (como la gramática $E \to E + T \mid T$, $T \to T * F \mid F$, $F \to id \mid (E)$)
2. **Introducir nuevas variables**: Distinguir casos (sentencias cerradas vs abiertas)
3. **Requerir delimitadores explícitos**: Usar llaves obligatorias para bloques (como en nuestro mini-lenguaje)
4. **Reglas de desambiguación externas**: Especificar precedencia y asociatividad fuera de la gramática (usado en herramientas como Yacc/Bison)

c. ¿Qué se puede reutilizar de soluciones pasadas?

De las secciones anteriores se reutilizan varios elementos:

- La técnica de **parser recursivo descendente** (sección 2), donde cada no terminal de la gramática se implementa como una función.
- La **gramática para expresiones aritméticas** con niveles de precedencia, que se incorpora directamente como parte de las expresiones del mini-lenguaje.
- La **tokenización** de la entrada antes del análisis sintáctico.
- Las notaciones **BNF y EBNF** para especificar formalmente la gramática.

d. ¿Cuál es el diseño de la solución?

Gramática completa del mini-lenguaje

La gramática del mini-lenguaje en notación EBNF combina todas las construcciones necesarias: declaraciones, asignaciones, ciclos, condicionales y expresiones con precedencia de operadores.

```
<programa>    ::= { <sentencia> }

<sentencia>   ::= <declaracion> | <asignacion>
                | <while> | <if>

<declaracion> ::= "var" <id> ";"

<asignacion>  ::= <id> "=" <expresion> ";"

<while>       ::= "while" "(" <condicion> ")" <bloque>

<if>          ::= "if" "(" <condicion> ")" <bloque>
                  [ "else" <bloque> ]

<bloque>      ::= "{" { <sentencia> } "}"

<condicion>   ::= <expresion> <op-rel> <expresion>
```

```
<op-rel>       ::= ">" | "<" | "==" | "!="

<expresion>    ::= <termino> { ("+" | "-") <termino> }

<termino>      ::= <factor> { ("*" | "/") <factor> }

<factor>       ::= <id> | <numero> | "(" <expresion> ")"
```

Observaciones sobre el diseño:

- El uso de llaves obligatorias en `<bloque>` evita la ambigüedad del `if-else` colgante.
- La precedencia de operadores se refleja en la jerarquía `<expresion>` → `<termino>` → `<factor>`: la multiplicación y división (en `<termino>`) se evalúan antes que la suma y resta (en `<expresion>`).
- El `else` es opcional (["`else`" ⟨bloque⟩]), permitiendo condicionales con o sin rama alternativa.
- Un `<programa>` es una repetición de sentencias ({⟨sentencia⟩}), lo que permite programas de cualquier longitud.

El parser por descenso recursivo organiza el análisis mediante una función asociada a cada no terminal de la gramática: `programa()`, `sentencia()`, `declaracion()`, `asignacion()`, `sent_while()`, `sent_if()`, `bloque()`, `condicion()`, `expresion()`, `termino()` y `factor()`.

Ejemplo de derivación:

Para el programa:

```
var x;
x = 5;
while (x > 0) {
    x = x - 1;
}
```

Se tiene la siguiente derivación parcial:

```
<programa>
  ::= { <sentencia> }
  ::= <sentencia> <sentencia> <sentencia>
  ::= <declaracion> <sentencia> <sentencia>
  ::= "var" <id> ";" <sentencia> <sentencia>
  ::= "var" x ";" <asignacion> <sentencia>
  ::= "var" x ";" <id> "=" <expresion> ";" <sentencia>
  ::= "var" x ";" x "=" 5 ";" <sentencia>
  ::= "var" x ";" x "=" 5 ";" <while>
  ::= "var" x ";" x "=" 5 ";"
      "while" "(" <condicion> ")" <bloque>
  ::= "var" x ";" x "=" 5 ";"
      "while" "(" x ">" 0 ")" "{" <sentencia> "}"
  ::= "var" x ";" x "=" 5 ";"
      "while" "(" x ">" 0 ")" "{" <asignacion> "}"
  ::= "var" x ";" x "=" 5 ";"
      "while" "(" x ">" 0 ")"
      "{" x "=" <expresion> ";" "}"
  ::= "var" x ";" x "=" 5 ";"
      "while" "(" x ">" 0 ")"
      "{" x "=" x "-" 1 ";" "}"
```

3.4.3 Ejemplos de pruebas

a. Ejemplo 1

Programa:

```
var x; x = 10;
```

Tokens: `["var", "x", ";", "x", "=", "10", ";"]`

- `programa()` llama a `sentencia()`: detecta `"var"` → `declaracion()`
- `declaracion()`: consume `"var"`, `"x"`, `";"` → válido
- `sentencia()`: detecta identificador `"x"` → `asignacion()`
- `asignacion()`: consume `"x"`, `"="`, llama a `expresion()` → `termino()` → `factor()` → consume `"10"`, luego `";"` → válido
- No quedan tokens

Resultado: `ACEPTADA`

b. Ejemplo 2

Programa:

```
var i; i = 0; while (i < 5) { i = i + 1; }
```

- `declaracion()`: consume `var i ;` → válido
- `asignacion()`: consume `i = 0 ;` → válido
- `sent_while()`: consume `while (`, llama a `condicion()`
- `condicion()`: `expresion()` → `i`, operador `<`, `expresion()` → `5`, consume `)`
- `bloque()`: consume `{`, dentro: `asignacion()` consume `i = i + 1 ;`, consume `}`
- No quedan tokens

Resultado: `ACEPTADA`

c. Ejemplo 3

Programa con error sintáctico:

```
var x; x = ;
```

Tokens: `["var", "x", ";", "x", "=", ";"]`

- `declaracion()`: consume `var x ;` → válido
- `asignacion()`: consume `x`, `=`, llama a `expresion()` → `termino()` → `factor()`: el token actual es `";"`, que no es un número, identificador ni paréntesis
- Error: falta una expresión después de =

Resultado: `RECHAZADA`

d. Ejemplo 4

Programa con condicional:

```
var x; x = 10; if (x > 5) { x = 1; } else { x = 0; }
```

- `declaracion()`: consume `var x ;` → válido
- `asignacion()`: consume `x = 10 ;` → válido
- `sent_if()`: consume `if (`, `condicion()`: `x > 5`, consume `)`
- Primer `bloque()`: `{ x = 1 ; }` → válido
- Detecta `else`, segundo `bloque()`: `{ x = 0 ; }` → válido
- No quedan tokens

Resultado: `ACEPTADA`

3.4.4 Codificación

a. Implementación del analizador sintáctico

La implementación del parser para el mini-lenguaje imperativo refleja la gramática: cada no terminal corresponde a un método de la clase `Parser`. La función principal y la lectura del programa fuente abren el flujo.

Código 3.14: Función principal y entrada/salida

```
def main():
    programa = leer("Ingrese el programa: ")
    resultado = validar_programa(programa)
    imprimir_resultado(programa, resultado)

def leer(texto):
    return input(texto)

def imprimir_resultado(programa, es_valido):
    if es_valido:
        print("ACEPTADA")
    else:
        print("RECHAZADA")
```

Antes de analizar sintácticamente, la tokenización recorre el texto carácter a carácter para identificar palabras clave, identificadores, números, operadores de dos caracteres (==, !=) y símbolos individuales.

Código 3.15: Tokenización y validación

```
def tokenizar(texto):
    tokens = []
    i = 0
    while i < len(texto):
        if texto[i].isspace():
            i += 1
        elif texto[i].isalpha():
            j = i
            while j < len(texto) and (texto[j].isalnum() or texto[j] == '_'
                ):
                j += 1
            tokens.append(texto[i:j])
            i = j
        elif texto[i].isdigit():
            j = i
            while j < len(texto) and texto[j].isdigit():
                j += 1
            tokens.append(texto[i:j])
            i = j
        elif i + 1 < len(texto) and texto[i:i+2] in ("==", "!="):
            tokens.append(texto[i:i+2])
            i += 2
        elif texto[i] in "+-*/(){}=;<>":
            tokens.append(texto[i])
            i += 1
        else:
            return None
    return tokens

def validar_programa(texto):
    tokens = tokenizar(texto)
    if tokens is None:
        return False
    parser = Parser(tokens)
    return parser.programa() and parser.fin()
```

La clase `Parser` comienza definiendo las operaciones básicas necesarias para procesar los tokens. Los métodos `actual()`, `consumir()`, `fin()`, `es_numero()` y `es_id()` permiten consultar y avanzar sobre la secuencia de entrada.

Código 3.16: Clase Parser: infraestructura

```
50 class Parser:
51     def __init__(self, tokens):
52         self.tokens = tokens
53         self.pos = 0
54 
55     def actual(self):
56         if self.pos < len(self.tokens):
57             return self.tokens[self.pos]
58         return None
59 
60     def consumir(self, esperado):
61         if self.actual() == esperado:
62             self.pos += 1
63             return True
64         return False
65 
66     def fin(self):
67         return self.pos == len(self.tokens)
68 
69     def es_numero(self, token):
70         return token is not None and token[0].isdigit()
71 
72     def es_id(self, token):
73         return (token is not None and token[0].isalpha()
74                 and token not in ("var", "while", "if", "else"))
```

En el nivel superior de la gramática, `programa()` y `sentencia()` seleccionan la función adecuada según la palabra clave encontrada.

Código 3.17: Clase Parser: programa y sentencias

```
76      # <programa> ::= { <sentencia> }
77      def programa(self):
78          while self.actual() is not None:
79              if not self.sentencia():
80                  return False
81          return True
82
83      # <sentencia> ::= <declaracion> | <while> | <if> | <asignacion>
84      def sentencia(self):
85          if self.actual() == "var":
86              return self.declaracion()
87          elif self.actual() == "while":
88              return self.sent_while()
89          elif self.actual() == "if":
90              return self.sent_if()
91          elif self.es_id(self.actual()):
92              return self.asignacion()
93          return False
94
95      # <declaracion> ::= "var" <id> ";"
96      def declaracion(self):
97          if not self.consumir("var"):
98              return False
99          if not self.es_id(self.actual()):
100             return False
101         self.pos += 1
102         return self.consumir(";")
103
104     # <asignacion> ::= <id> "=" <expresion> ";"
105     def asignacion(self):
106         if not self.es_id(self.actual()):
107             return False
108         self.pos += 1
109         if not self.consumir("="):
110             return False
111         if not self.expresion():
112             return False
113         return self.consumir(";")
```

Las sentencias compuestas con anidamiento corresponden a tres métodos: `sent_while`, `sent_if` y `bloque`.

Código 3.18: Clase Parser: estructuras de control

```
115     # <while> ::= "while" "(" <condicion> ")" <bloque>
116     def sent_while(self):
117         if not self.consumir("while"):
118             return False
119         if not self.consumir("("):
120             return False
121         if not self.condicion():
122             return False
123         if not self.consumir(")"):
124             return False
125         return self.bloque()
126 
127     # <if> ::= "if" "(" <condicion> ")" <bloque> ["else" <bloque>]
128     def sent_if(self):
129         if not self.consumir("if"):
130             return False
131         if not self.consumir("("):
132             return False
133         if not self.condicion():
134             return False
135         if not self.consumir(")"):
136             return False
137         if not self.bloque():
138             return False
139         if self.actual() == "else":
140             self.pos += 1
141             if not self.bloque():
142                 return False
143         return True
144 
145     # <bloque> ::= "{" { <sentencia> } "}"
146     def bloque(self):
147         if not self.consumir("{"):
148             return False
149         while self.actual() is not None and self.actual() != "}":
150             if not self.sentencia():
151                 return False
152         return self.consumir("}")
```

Por último, condiciones y expresiones aritméticas se implementan con la jerarquía `expresion` → `termino` → `factor`, que codifica la precedencia de operadores en la estructura misma del código.

Código 3.19: Clase Parser: condiciones y expresiones

```
154     # <condicion> ::= <expresion> <op-rel> <expresion>
155     def condicion(self):
156         if not self.expresion():
157             return False
158         if self.actual() in (">", "<", "==", "!="):
159             self.pos += 1
160             return self.expresion()
161         return False
162 
163     # <expresion> ::= <termino> { ("+" | "-") <termino> }
164     def expresion(self):
165         if not self.termino():
166             return False
167         while self.actual() in ("+", "-"):
168             self.pos += 1
169             if not self.termino():
170                 return False
171         return True
172 
173     # <termino> ::= <factor> { ("*" | "/") <factor> }
174     def termino(self):
175         if not self.factor():
176             return False
177         while self.actual() in ("*", "/"):
178             self.pos += 1
179             if not self.factor():
180                 return False
181         return True
182 
183     # <factor> ::= <id> | <numero> | "(" <expresion> ")"
184     def factor(self):
185         if self.actual() == "(":
186             self.pos += 1
187             if not self.expresion():
188                 return False
189             return self.consumir(")")
190         elif self.es_numero(self.actual()):
191             self.pos += 1
192             return True
193         elif self.es_id(self.actual()):
194             self.pos += 1
195             return True
196         return False
```

3.5. Análisis y optimización de gramáticas

La empresa CompilerTech está desarrollando un sistema de optimización de parsers. El equipo ha notado que ciertas gramáticas, aunque correctas, tienen producciones con formatos muy variados (producciones vacías, unitarias, mixtas, de longitud arbitraria), lo que dificulta aplicar algoritmos de análisis eficientes. Se necesita un procedimiento que transforme cualquier gramática a una forma restringida y uniforme, sin cambiar el lenguaje que genera.

3.5.1 Análisis / Abstracción

a. ¿Cuál es el problema a resolver?

CompilerTech necesita un procedimiento algorítmico para convertir cualquier gramática libre de contexto a Forma Normal de Chomsky, preservando el lenguaje generado.

b. ¿Cuál es la información necesaria?

El procedimiento toma una gramática de partida y un conjunto de restricciones que la gramática resultante debe cumplir:

Información necesaria	Descripción
Gramática original	Las variables, terminales y reglas de producción
Formato deseado	Restricciones que debe cumplir la gramática resultante

3.5.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **ENTRADA:** Una gramática libre de contexto G.
- **PROCESAMIENTO:** Transformar la gramática a una forma restringida y uniforme, eliminando producciones problemáticas y estandarizando el formato de todas las reglas.

- **SALIDA:** Una gramática equivalente G' en forma estandarizada que genera el mismo lenguaje.

b. ¿Qué conocimiento adicional se requiere?

El diseño requiere la Forma Normal de Chomsky y el procedimiento algorítmico que convierte cualquier gramática a esa forma.

Forma Normal de Chomsky (FNC)

Las **formas normales** son restricciones sintácticas sobre las producciones de una gramática que simplifican el análisis teórico y práctico sin reducir el poder expresivo.

Una gramática está en **Forma Normal de Chomsky** [Chomsky, 1959, Hopcroft et al., 2006] si todas sus producciones tienen una de las siguientes formas:

- $A \to BC$ (dos variables)
- $A \to a$ (un terminal)
- $S \to \varepsilon$ (solo si $\varepsilon \in L(G)$, y S no aparece en el lado derecho de ninguna producción)

donde $A, B, C \in V$, $a \in \Sigma$, y S es el símbolo inicial.

Utilidad: La FNC es útil para:

- Diseñar algoritmos de parsing eficientes (algoritmo CYK)
- Demostrar teoremas sobre lenguajes libres de contexto
- Acotar la longitud de derivaciones

Ejemplo 9: Conversión paso a paso a FNC

Consideremos la siguiente gramática:

$$\begin{aligned} S &\rightarrow AB \mid a \\ A &\rightarrow aA \mid \varepsilon \\ B &\rightarrow bB \mid \varepsilon \end{aligned}$$

Esta gramática genera cadenas de la forma $a^n b^m$ donde $n, m \geq 0$.

Paso 1: Eliminar producciones ε

Identificamos las variables **anulables**: A y B pueden derivar ε. Como $S \rightarrow AB$ y ambas son anulables, S también es anulable (el lenguaje contiene ε).

Eliminamos las producciones ε y compensamos agregando versiones sin las variables anulables:

- $S \rightarrow AB$: agregamos $S \rightarrow A \mid B$ (cuando $B = \varepsilon$ o $A = \varepsilon$)
- $A \rightarrow aA$: agregamos $A \rightarrow a$ (cuando $A = \varepsilon$)
- $B \rightarrow bB$: agregamos $B \rightarrow b$ (cuando $B = \varepsilon$)

Resultado:

$$\begin{aligned} S &\rightarrow AB \mid A \mid B \mid a \mid \varepsilon \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid b \end{aligned}$$

Paso 2: Eliminar producciones unitarias

Las producciones unitarias son $S \rightarrow A$ y $S \rightarrow B$. Las reemplazamos por las producciones de A y B:

- $S \rightarrow A$ se convierte en $S \rightarrow aA \mid a$
- $S \rightarrow B$ se convierte en $S \rightarrow bB \mid b$

Resultado:

$$\begin{aligned} S &\rightarrow AB \mid aA \mid a \mid bB \mid b \mid \varepsilon \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid b \end{aligned}$$

Paso 3: Separar terminales en producciones mixtas

Las producciones aA y bB mezclan terminales con variables. Creamos variables auxiliares para los terminales:

$$T_a \to a \qquad T_b \to b$$

Reemplazamos aA por T_aA y bB por T_bB:

$$\begin{array}{rcl} S & \to & AB \mid T_aA \mid a \mid T_bB \mid b \mid \varepsilon \\ A & \to & T_aA \mid a \\ B & \to & T_bB \mid b \\ T_a & \to & a \\ T_b & \to & b \end{array}$$

Paso 4: Verificación

Todas las producciones cumplen con la FNC:

- $A \to BC$: dos variables ($S \to AB$, $S \to T_aA$, $S \to T_bB$, $A \to T_aA$, $B \to T_bB$)
- $A \to a$: un terminal ($S \to a$, $S \to b$, $A \to a$, $B \to b$, $T_a \to a$, $T_b \to b$)
- $S \to \varepsilon$: permitida solo para el símbolo inicial

Gramática final en FNC:

$$\begin{array}{rcl} S & \to & AB \mid T_aA \mid T_bB \mid a \mid b \mid \varepsilon \\ A & \to & T_aA \mid a \\ B & \to & T_bB \mid b \\ T_a & \to & a \\ T_b & \to & b \end{array}$$

Ahora todas las reglas cumplen estrictamente con la definición de Forma Normal de Chomsky.

Teorema: Para toda gramática libre de contexto G existe una gramática equivalente G' en Forma Normal de Chomsky.
Es decir: $L(G) = L(G')$
La conversión es algorítmica y siempre termina.

c. ¿Qué se puede reutilizar de soluciones pasadas?

De las secciones anteriores se reutiliza el concepto de gramática libre de contexto y derivaciones. Los pasos de conversión a FNC son algorítmicos y se aplican de manera sistemática a cualquier gramática.

d. ¿Cuál es el diseño de la solución?

Procedimiento de conversión a FNC

El procedimiento consta de los pasos presentados en el ejemplo anterior:

1. Eliminar producciones ε (identificar variables anulables y compensar)
2. Eliminar producciones unitarias ($A \to B$)
3. Separar terminales en producciones mixtas (crear variables auxiliares)
4. Binarizar producciones con más de dos variables ($A \to BCD$ se convierte en $A \to BZ_0$, $Z_0 \to CD$)

Nota: En la gramática del ejemplo, ninguna producción tiene más de dos símbolos en el lado derecho después de los pasos 1–3, por lo que el paso 4 no modifica la gramática. Sin embargo, es necesario para el caso general.

3.5.3 Ejemplos de pruebas

a. Ejemplo 1

Verificamos la conversión derivando la cadena aab en ambas gramáticas:

Gramática original:

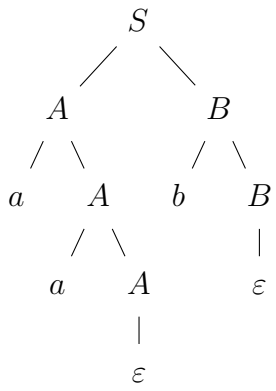
$$S \Rightarrow AB \Rightarrow aAB \Rightarrow aaB \Rightarrow aabB \Rightarrow aab$$

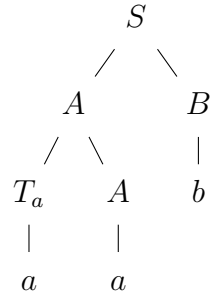
Gramática en FNC:

$$S \Rightarrow AB \Rightarrow T_aA \cdot B \Rightarrow aA \cdot B \Rightarrow aa \cdot B \Rightarrow aa \cdot b = aab$$

Ambas gramáticas generan la misma cadena. La diferencia se aprecia mejor en los árboles de derivación:

Gramática original **Gramática en FNC**





En el árbol original aparecen producciones ε (ramas que no generan nada) y nodos con cantidad variable de hijos. En el árbol en FNC, cada nodo interno tiene exactamente **dos hijos** (cuando produce variables) o **un hijo terminal** (cuando produce un símbolo del alfabeto). Esta estructura binaria uniforme es la que permite diseñar algoritmos de parsing eficientes como CYK.

b. Ejemplo 2

Verificamos con la cadena b:

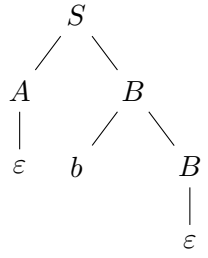
Gramática original:

$$S \Rightarrow AB \Rightarrow \varepsilon \cdot B \Rightarrow B \Rightarrow bB \Rightarrow b\varepsilon = b$$

Gramática en FNC:

$$S \Rightarrow b$$

Gramática original **Gramática en FNC**



En la gramática original se necesitan 5 pasos de derivación y ramas ε para generar una sola letra. En FNC, la producción $S \to b$ lo resuelve directamente.

c. Ejemplo 3

Verificamos con la cadena vacía ε:

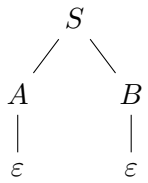
Gramática original:

$$S \Rightarrow AB \Rightarrow \varepsilon \cdot \varepsilon = \varepsilon$$

Gramática en FNC:

$$S \Rightarrow \varepsilon$$

Gramática original



Gramática en FNC

En la gramática original, la cadena vacía se genera a través de dos variables intermedias que derivan ε. En FNC, solo el símbolo inicial S puede producir ε directamente, eliminando las ramas innecesarias.

3.5.4 Codificación

a. Implementación en Python

La conversión a FNC se implementa mediante un algoritmo que aplica, en cuatro etapas, las transformaciones descritas en el diseño. La función principal coordina todo el proceso sobre una gramática representada mediante un diccionario.

Código 3.20: **Función principal**

```
1 def main():
2     gramatica = leer_gramatica()
3     fnc = convertir_a_fnc(gramatica, 'S')
4     imprimir_gramatica(fnc)
```

La gramática se representa como un diccionario que asocia cada variable con la lista de sus producciones; a su vez, cada producción corresponde a una lista de símbolos. El terminal especial `'eps'` representa ε.

Código 3.21: Definición de la gramática

```
def leer_gramatica():
    return {
        'S': [['A', 'B'], ['a']],
        'A': [['a', 'A'], ['eps']],
        'B': [['b', 'B'], ['eps']]
    }
```

Para visualizar el resultado intermedio o final se utiliza una función que recorre el diccionario y muestra cada variable junto con sus producciones separadas por `|`.

Código 3.22: Función de impresión

```
def imprimir_gramatica(gramatica):
    for var, prods in gramatica.items():
        derecha = ' | '.join(
            ' '.join(s for s in p) if p != ['eps'] else 'eps'
            for p in prods
        )
        print("\nGramatica en FNC:")
        print(f"  {var} -> {derecha}")
```

El primer paso consiste en identificar las variables anulables, es decir, aquellas capaces de derivar ε. Para ello se utiliza un algoritmo de punto fijo que continúa iterando hasta que no aparecen nuevas variables anulables.

Código 3.23: Identificación de variables anulables

```
def encontrar_anulables(gramatica):
    anulables = set()
    cambio = True
    while cambio:
        cambio = False
        for var, prods in gramatica.items():
            if var in anulables:
                continue
            for prod in prods:
                if prod == ['eps'] or all(
                    s in anulables for s in prod
                ):
                    anulables.add(var)
                    cambio = True
                    break
    return anulables
```

Una vez identificadas las anulables, se eliminan las producciones ε generando todas las variantes posibles, tanto con la presencia como con la ausencia de cada variable anulable. Si el símbolo inicial era anulable, la producción ε se conserva únicamente para él.

Código 3.24: Eliminación de producciones ε

```
def generar_variantes(prod, anulables):
    if not prod:
        return [[]]
    primero = prod[0]
    resto = generar_variantes(prod[1:], anulables)
    resultado = []
    for rv in resto:
        resultado.append([primero] + rv)
        if primero in anulables:
            resultado.append(rv)
    return resultado

def eliminar_epsilon(gramatica, simbolo_inicial):
    anulables = encontrar_anulables(gramatica)
    nueva = {}
    for var, prods in gramatica.items():
        nuevas_prods = []
        for prod in prods:
            if prod == ['eps']:
                continue
            for v in generar_variantes(prod, anulables):
                if v and v not in nuevas_prods:
                    nuevas_prods.append(v)
        nueva[var] = nuevas_prods
    if simbolo_inicial in anulables:
        if ['eps'] not in nueva[simbolo_inicial]:
            nueva[simbolo_inicial].append(['eps'])
    return nueva
```

El segundo paso elimina las producciones unitarias ($A \to B$) mediante un recorrido que reemplaza cada cadena unitaria por sus producciones finales.

Código 3.25: Eliminación de producciones unitarias

```
68 def eliminar_unitarias(gramatica):
69     nueva = {}
70     for var in gramatica:
71         nueva[var] = []
72         visitados = set()
73         cola = [var]
74         while cola:
75             actual = cola.pop(0)
76             if actual in visitados:
77                 continue
78             visitados.add(actual)
79             for prod in gramatica.get(actual, []):
80                 if len(prod) == 1 and prod[0] in gramatica:
81                     cola.append(prod[0])
82                 elif prod not in nueva[var]:
83                     nueva[var].append(prod)
84     return nueva
```

A continuación, los terminales que aparecen mezclados con variables se separan creando variables auxiliares para representarlos.

Código 3.26: Separación de terminales

```
86 def separar_terminales(gramatica):
87     terminales = set()
88     for prods in gramatica.values():
89         for prod in prods:
90             for s in prod:
91                 if s not in gramatica and s != 'eps':
92                     terminales.add(s)
93     auxiliares = {}
94     for t in sorted(terminales):
95         auxiliares[f'T_{t}'] = [[t]]
96     resultado = {}
97     for var, prods in gramatica.items():
98         nuevas_prods = []
99         for prod in prods:
100            if len(prod) <= 1:
101                nuevas_prods.append(prod)
102            else:
103                nueva_prod = []
104                for s in prod:
105                    if s in terminales:
106                        nueva_prod.append(f'T_{s}')
107                    else:
108                        nueva_prod.append(s)
109                nuevas_prods.append(nueva_prod)
110        resultado[var] = nuevas_prods
111    resultado.update(auxiliares)
112    return resultado
```

Por último, las producciones con más de dos símbolos se binarizan introduciendo variables auxiliares. La función `convertir_a_fnc()` ejecuta las cuatro etapas completas del proceso.

Código 3.27: Binarización y conversión completa

```
def binarizar(gramatica):
    resultado = {}
    nuevas_vars = {}
    contador = 0
    for var, prods in gramatica.items():
        nuevas_prods = []
        for prod in prods:
            if len(prod) <= 2:
                nuevas_prods.append(prod)
            else:
                simbolos = list(prod)
                while len(simbolos) > 2:
                    ultimo = simbolos.pop()
                    penultimo = simbolos.pop()
                    nueva_var = f'Z{contador}'
                    contador += 1
                    nuevas_vars[nueva_var] = [
                        [penultimo, ultimo]
                    ]
                    simbolos.append(nueva_var)
                nuevas_prods.append(simbolos)
        resultado[var] = nuevas_prods
    resultado.update(nuevas_vars)
    return resultado

def convertir_a_fnc(gramatica, simbolo_inicial):
    g = eliminar_epsilon(gramatica, simbolo_inicial)
    g = eliminar_unitarias(g)
    g = separar_terminales(g)
    g = binarizar(g)
    return g
```

3.6. Validación de XML/HTML

La empresa WebSecure, especializada en seguridad web, está implementando un validador de documentos XML/HTML [World Wide Web Consortium, 2008] que debe verificar que las etiquetas de apertura y cierre estén correctamente emparejadas y anidadas dentro del documento.

Por ejemplo:

- **Cadenas válidas:** `<html><body>Hola</body></html>`.
- **Cadenas inválidas:** `<html><body>Hola</html></body>`.

La última falla porque viola el principio de que el último elemento en abrirse debe ser el primero en cerrarse.

El desafío es que el verificador debe "recordar" qué etiquetas se han abierto y en qué orden, para poder comprobar que cada cierre corresponda a la apertura correcta.

3.6.1 Análisis / Abstracción

a. ¿Cuál es el problema a resolver?

WebSecure necesita un mecanismo para validar la estructura anidada de un documento XML/HTML, comprobando que cada etiqueta de apertura tenga su correspondiente cierre y que el anidamiento entre etiquetas sea correcto. Los autómatas finitos no pueden resolver este problema porque se requiere "recordar" información sobre un número ilimitado de etiquetas abiertas.

b. ¿Cuál es la información necesaria?

El validador opera sobre el documento como un todo, considerando además el conjunto de etiquetas que el formato admite:

Información necesaria	Descripción
Documento de entrada	El texto XML/HTML a verificar
Conjunto de etiquetas	Nombres válidos que pueden aparecer en el documento

3.6.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **ENTRADA:** Un documento XML/HTML.
- **PROCESAMIENTO:** Verificar que cada etiqueta de apertura tenga su correspondiente cierre, respetando el orden de anidamiento.
- **SALIDA:** Indicar si el documento está correctamente emparejado y anidado, o no.

b. ¿Qué conocimiento adicional se requiere?

Para diseñar la solución se requiere un modelo computacional más potente que los autómatas finitos: el **autómata de pila** (PDA). Este modelo extiende los autómatas finitos con una pila como memoria auxiliar.

Autómatas de Pila (PDA)

Los autómatas finitos no pueden reconocer lenguajes que requieren "contar" o "emparejar" símbolos de manera ilimitada, como los paréntesis balanceados o lenguajes de la forma $\{0^n 1^n \mid n \geq 0\}$, ya que carecen de memoria suficiente para almacenar una cantidad arbitraria de información.

La solución es agregar una **pila** (*stack*): una memoria **LIFO** (Last In, First Out) que permite almacenar y recuperar información temporalmente.

Un **autómata de pila** (PDA) es una séptupla [Sipser, 2012]:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

donde:

- Q es un conjunto finito de **estados**
- Σ es el **alfabeto de entrada**
- Γ es el **alfabeto de la pila**
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ es la **función de transición**
- $q_0 \in Q$ es el **estado inicial**
- $Z_0 \in \Gamma$ es el **símbolo inicial de la pila**
- $F \subseteq Q$ es el conjunto de **estados de aceptación**

Interpretación operacional

La función de transición $\delta(q, a, X)$ especifica:

- Estando en el estado q
- Leyendo el símbolo a de la entrada (o ε para transición sin consumir entrada)
- Con X en el tope de la pila

Se puede transitar a un nuevo estado p y reemplazar X en el tope de la pila por una cadena $\alpha \in \Gamma^*$:

- Si $\alpha = \varepsilon$: se desapila X (pop)
- Si $\alpha = X$: no cambia la pila
- Si $\alpha = YZ$: se reemplaza X por YZ (equivalente a pop X, push Z, push Y)

Criterios de aceptación

Existen dos criterios de aceptación equivalentes:

1. **Por estado final**: La cadena se acepta si, al terminar de leer la entrada, el autómata está en un estado de F

2. **Por pila vacía**: La cadena se acepta si, al terminar de leer la entrada, la pila está vacía

Ambos criterios son equivalentes en poder expresivo.

Equivalencia entre GLC y PDA

Teorema fundamental [Chomsky, 1962, Hopcroft et al., 2006]**:** Un lenguaje es libre de contexto si y solo si es aceptado por algún autómata de pila.
Más específicamente:

- Para toda gramática libre de contexto G, existe un PDA M tal que $L(G) = L(M)$
- Para todo PDA M, existe una gramática G tal que $L(M) = L(G)$

El resultado es paralelo al de los lenguajes regulares, donde AFD, AFN y expresiones regulares describen exactamente la misma clase.

Ejemplo 10: PDA para paréntesis balanceados

Diseñemos un PDA para el lenguaje:

$L = \{w \in \{(,)\}^* \mid w \text{ tiene paréntesis balanceados}\}$

Estrategia:

- Por cada `(` que leemos: apilamos un símbolo X
- Por cada `)` que leemos: desapilamos un símbolo X
- Al finalizar la entrada: si la pila solo contiene Z_0, aceptamos mediante transición ε

Definición formal:

$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ donde:

- $Q = \{q_0, q_f\}$

- $\Sigma = \{(,)\}$
- $\Gamma = \{Z_0, X\}$
- q_0 es el estado inicial
- Z_0 es el símbolo inicial de pila
- $F = \{q_f\}$

Función de transición:

$$\begin{aligned}
\delta(q_0, (, Z_0) &= \{(q_0, XZ_0)\} && \text{Primer (: apila } X \\
\delta(q_0, (, X) &= \{(q_0, XX)\} && \text{Más (: apila } X \\
\delta(q_0,), X) &= \{(q_0, \varepsilon)\} && \text{Cierra): desapila } X \\
\delta(q_0, \varepsilon, Z_0) &= \{(q_f, Z_0)\} && \text{Entrada terminada: acepta}
\end{aligned}$$

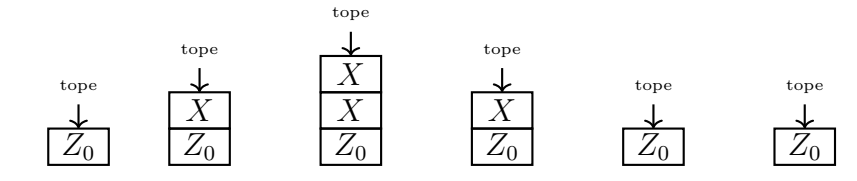
Nota: En las trazas siguientes, la pila se representa con el tope a la **izquierda**. Por ejemplo, XZ_0 significa que X está en el tope y Z_0 debajo.

Ejemplo de aceptación: Procesamos la cadena (()):

Paso	**Estado**	**Entrada**	**Pila**	**Transición aplicada**
0	q_0	(())	Z_0	Configuración inicial
1	q_0	())	XZ_0	Lee (sobre Z_0
2	q_0	))	XXZ_0	Lee (sobre X
3	q_0	)	XZ_0	Lee) sobre X, desapila
4	q_0	ε	Z_0	Lee) sobre X, desapila
5	q_f	ε	Z_0	Transición ε a estado final

Resultado: Cadena **aceptada** (alcanza estado final q_f con entrada consumida)

Visualización de la pila: La siguiente figura muestra cómo evoluciona la pila durante el procesamiento:



La pila crece cuando se leen paréntesis de apertura (( ) y decrece cuando se leen paréntesis de cierre ( )). Al final, queda solo Z_0, permitiendo la aceptación.

Ejemplo de rechazo 1: Procesamos la cadena ()) (más cierres que aperturas):

Paso	**Estado**	**Entrada**	**Pila**	**Transición aplicada**
0	q_0	())	Z_0	Configuración inicial
1	q_0	))	XZ_0	Lee (sobre Z_0
2	q_0	)	Z_0	Lee) sobre X, desapila
3	—	)	—	**No existe** $\delta(q_0,), Z_0)$

Resultado: Cadena **rechazada** (no hay transición para) con Z_0 en el tope)

Ejemplo de rechazo 2: Procesamos la cadena ((((más aperturas que cierres):

Paso	**Estado**	**Entrada**	**Pila**	**Transición aplicada**
0	q_0	(((	Z_0	Configuración inicial
1	q_0	((	XZ_0	Lee (sobre Z_0
2	q_0	(	XXZ_0	Lee (sobre X
3	q_0	)	$XXXZ_0$	Lee (sobre X
4	q_0	ε	XXZ_0	Lee) sobre X, desapila

Resultado: Cadena **rechazada** (entrada consumida pero pila contiene $XXZ_0 \neq Z_0$, no puede llegar a q_f)

Este PDA rechaza una cadena en dos casos:

1. **No existe transición válida:** Ocurre cuando hay más) que (en un prefijo
2. **Entrada consumida sin alcanzar estado final:** Ocurre cuando hay más (que) y la pila no queda con solo Z_0

Ejemplo 11: PDA para delimitadores múltiples

Una extensión natural del PDA para paréntesis balanceados consiste en aceptar también corchetes y llaves, manteniendo el anidamiento correcto entre los tres tipos. El lenguaje a reconocer es:

$$L = \{w \in \{(,),[,],\{,\}\}^* \mid w \text{ está balanceado y bien anidado}\}$$

Una gramática que lo describe es:

$$S \to \varepsilon \mid (S) \mid [S] \mid \{S\} \mid SS$$

Estrategia del PDA:

- Por cada símbolo de apertura leído (`(`, `[`, `{`): se apila el mismo símbolo.
- Por cada símbolo de cierre (`)`, `]`, `}`): se comprueba que el tope de la pila sea la apertura correspondiente y se desapila. Si no coincide, no existe transición y la cadena se rechaza.
- Al consumir toda la entrada, si la pila contiene únicamente Z_0 se acepta mediante una transición ε al estado final.

Definición formal: $M = (\{q_0, q_f\}, \Sigma, \Gamma, \delta, q_0, Z_0, \{q_f\})$ con $\Sigma = \{(,),[,],\{,\}\}$ y $\Gamma = \{(,[,\{,Z_0\}$.

La función de transición se define, para cada símbolo de apertura $a \in \{(,[,\{\}$ y su cierre correspondiente $\bar{a}$:

$$\begin{aligned}
\delta(q_0, a, X) &= \{(q_0, aX)\} \quad \text{para cualquier } X \in \Gamma \\
\delta(q_0, \bar{a}, a) &= \{(q_0, \varepsilon)\} \\
\delta(q_0, \varepsilon, Z_0) &= \{(q_f, Z_0)\}
\end{aligned}$$

Traza para `([{}])`:

Paso	Estado	Entrada	Pila	Acción
0	q_0	`([{}])`	Z_0	Configuración inicial
1	q_0	`[{}])`	`(` Z_0	Apila `(`
2	q_0	`{}])`	`[(` Z_0	Apila `[`
3	q_0	`}])`	`{[(` Z_0	Apila `{`
4	q_0	`])`	`[(` Z_0	Coincide con `{`, desapila
5	q_0	`)`	`(` Z_0	Coincide con `[`, desapila
6	q_0	ε	Z_0	Coincide con `(`, desapila
7	q_f	ε	Z_0	Transición ε a estado final

Resultado: cadena **aceptada**.

Ejemplo 12: PDA para $\{0^n 1^n \mid n \geq 0\}$

Diseñemos un PDA para el lenguaje:

$L = \{0^n 1^n \mid n \geq 0\}$

Este es el lenguaje clásico que no es regular pero sí es libre de contexto.

Estrategia:

- Por cada 0 que leemos: apilamos un símbolo X
- Por cada 1 que leemos: desapilamos un símbolo X
- Al finalizar la entrada: si la pila solo contiene Z_0, aceptamos mediante transición ε

Definición formal:

$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ donde:

- $Q = \{q_0, q_1, q_f\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{Z_0, X\}$
- q_0 es el estado inicial
- Z_0 es el símbolo inicial de pila
- $F = \{q_f\}$

Función de transición:

$\delta(q_0, 0, Z_0) = \{(q_0, XZ_0)\}$	Primer 0: apila X
$\delta(q_0, 0, X) = \{(q_0, XX)\}$	Más 0s: apila X
$\delta(q_0, 1, X) = \{(q_1, \varepsilon)\}$	Cambio a 1s: desapila
$\delta(q_1, 1, X) = \{(q_1, \varepsilon)\}$	Más 1s: desapila
$\delta(q_1, \varepsilon, Z_0) = \{(q_f, Z_0)\}$	Pila inicial: acepta

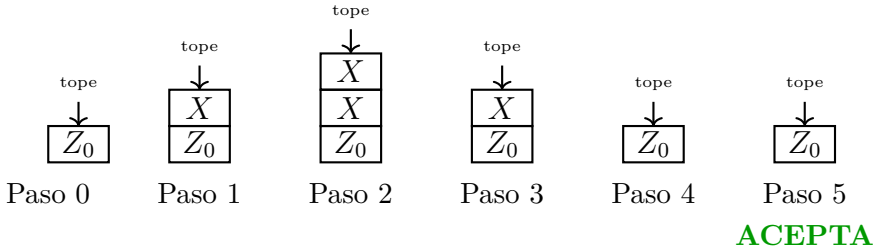
Nota: En las trazas siguientes, la pila se representa con el tope a la **izquierda**. Por ejemplo, XZ_0 significa que X está en el tope y Z_0 debajo.

Ejemplo de aceptación: Procesamos la cadena 0011:

Paso	Estado	Entrada	Pila	Transición aplicada
0	q_0	0011	Z_0	Configuración inicial
1	q_0	011	XZ_0	Lee 0 sobre Z_0
2	q_0	11	XXZ_0	Lee 0 sobre X
3	q_1	1	XZ_0	Lee 1 sobre X, desapila
4	q_1	ε	Z_0	Lee 1 sobre X, desapila
5	q_f	ε	Z_0	Transición ε a estado final

Resultado: Cadena **aceptada** (alcanza estado final q_f con entrada consumida)

Visualización de la pila: La siguiente figura muestra cómo evoluciona la pila durante el procesamiento de la cadena 0011:



La pila crece cuando se leen 0s (apilando Xs) y decrece cuando se leen 1s (desapilando Xs). Al final, queda solo Z_0, permitiendo la aceptación.

Ejemplo de rechazo 1: Procesamos la cadena 001 (más 0s que 1s):

Paso	Estado	Entrada	Pila	Transición aplicada
0	q_0	001	Z_0	Configuración inicial
1	q_0	01	XZ_0	Lee 0 sobre Z_0
2	q_0	1	XXZ_0	Lee 0 sobre X
3	q_1	ε	XZ_0	Lee 1 sobre X, desapila

Resultado: Cadena **rechazada** (entrada consumida pero pila contiene $XZ_0 \neq Z_0$, no puede llegar a q_f)

Ejemplo de rechazo 2: Procesamos la cadena 0111 (más 1s que 0s):

Paso	Estado	Entrada	Pila	Transición aplicada
0	q_0	0111	Z_0	Configuración inicial
1	q_0	111	XZ_0	Lee 0 sobre Z_0
2	q_1	11	Z_0	Lee 1 sobre X, desapila
3	—	11	—	**No existe** $\delta(q_1, 1, Z_0)$

Resultado: Cadena **rechazada** (no hay transición para 1 con Z_0 en el tope en estado q_1)

c. ¿Qué se puede reutilizar de soluciones pasadas?

Aunque este ejercicio podría resolverse directamente usando una GLC, el PDA resulta más adecuado para modelar la verificación de estructuras anidadas donde las etiquetas deben coincidir, como en XML o HTML. La pila permite manejar el anidamiento de manera directa, mientras que en una GLC sería necesario introducir reglas adicionales para garantizar la correspondencia entre aperturas y cierres.

Sin embargo, la GLC sigue siendo útil para describir la estructura general del lenguaje, y a partir de ella es posible construir el PDA mediante el procedimiento de conversión correspondiente.

d. ¿Cuál es el diseño de la solución?

GLC para XML/HTML

La estructura de documentos XML/HTML puede especificarse en EBNF:

Nota importante: Esta gramática captura la estructura sintáctica, pero la verificación de que el tag de apertura y cierre tengan el mismo nombre es una restricción semántica adicional.

```
<doc> ::= { <elem> }

<elem> ::= <tag> | <texto>

<tag> ::= "<" <id> ">" { <elem> } "</" <id> ">"

<texto> ::= <caracteres>
```

Ejemplo de derivación:

Para el documento: `<html><body>Hola</body></html>`

```
<doc>
  ::= { <elem> }
  ::= <elem>
  ::= <tag>
  ::= "<" <id> ">" { <elem> } "</" <id> ">"
  ::= "<html>" { <elem> } "</html>"
  ::= "<html>" <elem> "</html>"
  ::= "<html>" <tag> "</html>"
  ::= "<html>" "<body>" { <elem> } "</body>" "</html>"
  ::= "<html>" "<body>" <elem> "</body>" "</html>"
  ::= "<html>" "<body>" <texto> "</body>" "</html>"
  ::= "<html>" "<body>" Hola "</body>" "</html>"
```

PDA para XML/HTML

A diferencia de los delimitadores simples, donde un símbolo abre y otro cierra, las etiquetas XML llevan un nombre que debe emparejarse: la apertura `<html>` solo puede cerrarse con `</html>`. Esto exige que el alfabeto de pila Γ contenga los nombres completos de las etiquetas, no símbolos abstractos.

Sea T el conjunto de nombres válidos de etiqueta. El PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ que reconoce documentos XML bien formados se define con:

- $Q = \{q_0, q_f\}$
- $\Gamma = T \cup \{Z_0\}$
- $F = \{q_f\}$

La función de transición opera sobre etiquetas tomadas como un solo símbolo del flujo de entrada (un paso previo de extracción separa las etiquetas del texto). Para cada nombre $t \in T$:

$$\delta(q_0, \texttt{<}t\texttt{>}, X) = \{(q_0, tX)\} \quad \text{para cualquier } X \in \Gamma$$
$$\delta(q_0, \texttt{</}t\texttt{>}, t) = \{(q_0, \varepsilon)\}$$
$$\delta(q_0, \varepsilon, Z_0) = \{(q_f, Z_0)\}$$

La primera regla apila el nombre de la etiqueta cuando se lee una apertura. La segunda exige que el nombre del cierre coincida con el tope de la pila para poder desapilar; si no coinciden, no hay transición disponible y el documento se rechaza. La tercera lleva al estado de aceptación cuando la entrada se ha consumido y la pila ha vuelto a Z_0.

3.6.3 Ejemplos de pruebas

a. Ejemplo 1

Documento HTML bien anidado:

`<html><body>Hola</body></html>`.

El paso previo de extracción separa el texto y entrega la secuencia de etiquetas `<html>`, `<body>`, `</body>`, `</html>`.

- Lee `<html>`: apila `html`. Pila: `[html]`
- Lee `<body>`: apila `body`. Pila: `[html, body]`
- Lee `</body>`: tope = `body`, coincide. Desapila. Pila: `[html]`
- Lee `</html>`: tope = `html`, coincide. Desapila. Pila: vacía

Resultado: `ACEPTADA` (pila vacía al final)

b. Ejemplo 2

Documento HTML mal anidado:

`<html><body>Hola</html></body>`,

con etiquetas en orden cruzado. Etiquetas extraídas: `<html>`, `<body>`, `</html>`, `</body>`.

- Lee `<html>`: apila `html`. Pila: `[html]`
- Lee `<body>`: apila `body`. Pila: `[html, body]`
- Lee `</html>`: tope = `body`, **no coincide** con `html`

Resultado: `RECHAZADA` (etiqueta de cierre no corresponde a la apertura más reciente)

c. Ejemplo 3

Documento con etiqueta de cierre faltante:

`<html><body>Hola</body>`.

Etiquetas extraídas: `<html>`, `<body>`, `</body>`.

- Lee `<html>`: apila `html`. Pila: `[html]`
- Lee `<body>`: apila `body`. Pila: `[html, body]`
- Lee `</body>`: tope = `body`, coincide. Desapila. Pila: `[html]`
- Entrada consumida con `html` aún en la pila

Resultado: `RECHAZADA` (queda una etiqueta sin cerrar al final del documento)

3.6.4 Codificación

La implementación en Python sigue el diseño del PDA. El alfabeto de pila Γ se representa como una lista que almacena los nombres de las etiquetas abiertas. La operación principal consiste en comparar cada etiqueta de cierre con el elemento en el tope de la pila para decidir si se realiza el desapilado.

La función principal, junto con las funciones auxiliares de entrada y salida, coordina la ejecución del programa.

Código 3.28: Función principal y entrada/salida

```
def main():
    cadena = leer("Ingrese el documento: ")
    resultado = validar_documento(cadena)
    imprimir_resultado(cadena, resultado)

def leer(texto):
    return input(texto)

def imprimir_resultado(cadena, es_valido):
    if es_valido:
        print(f"'{cadena}': ACEPTADA")
    else:
        print(f"'{cadena}': RECHAZADA")
```

Antes de validar hay que recuperar las etiquetas como símbolos del alfabeto de entrada. La función `extraer_etiquetas` recorre el documento buscando contenido entre < y >, y devuelve cada etiqueta como una tupla `(tipo, nombre)` donde `tipo` es `"apertura"` o `"cierre"`.

Código 3.29: Extracción de etiquetas

```
def extraer_etiquetas(cadena):
    etiquetas = []
    i = 0
    while i < len(cadena):
        if cadena[i] == '<':
            j = cadena.find('>', i)
            if j == -1:
                return None
            contenido = cadena[i+1:j]
            if contenido.startswith('/'):
                etiquetas.append(("cierre", contenido[1:]))
            else:
                etiquetas.append(("apertura", contenido))
            i = j + 1
        else:
            i += 1
    return etiquetas
```

Con la secuencia de etiquetas en mano, la validación reproduce las tres reglas del PDA: $\delta(q_0, \texttt{<}t\texttt{>}, X) = \{(q_0, tX)\}$ apila el nombre, $\delta(q_0, \texttt{</}t\texttt{>}, t) = \{(q_0, \varepsilon)\}$ desapila si coincide con el tope, y $\delta(q_0, \varepsilon, Z_0) = \{(q_f, Z_0)\}$ acepta cuando la pila vuelve a Z_0 al consumir toda la entrada.

Código 3.30: Validación de documento

```
def validar_documento(cadena):
    etiquetas = extraer_etiquetas(cadena)
    if etiquetas is None:
        return False
    pila = []
    for tipo, nombre in etiquetas:
        if tipo == "apertura":
            pila.append(nombre)
        else:
            if len(pila) == 0 or pila[-1] != nombre:
                return False
            pila.pop()
    return len(pila) == 0

main()
```

3.7. Ejercicios

3.7.1 Diseño de gramáticas libres de contexto

Una empresa de verificación formal necesita que sus ingenieros demuestren competencia en el diseño de gramáticas libres de contexto. Se requiere diseñar una GLC para cada uno de los siguientes lenguajes:

a) $L_1 = \{a^n b^n \mid n \geq 0\}$

b) $L_2 = \{w \in \{a, b\}^* \mid w = w^R\}$ (palíndromos sobre $\{a, b\}$)

c) $L_3 = \{a^i b^j c^k \mid i = j \text{ o } j = k\}$

d) $L_4 = \{w \in \{a, b\}^* \mid \#_a(w) = 2 \cdot \#_b(w)\}$ (cadenas con el doble de a's que b's)

3.7.2 Gramática para expresiones booleanas

Un equipo de desarrollo de un motor de reglas de negocio necesita diseñar una gramática para expresiones booleanas que incluya:

- Operadores lógicos: AND, OR, NOT
- Valores: true, false, variables
- Paréntesis para agrupación
- Precedencia: NOT > AND > OR

Ejemplos de cadenas válidas: `(a AND b) OR c`, `NOT (x OR y)`, `true AND NOT false`

a) Escriba la gramática asegurando la precedencia correcta de operadores

b) Proporcione una derivación izquierda para: `NOT a AND b OR c`

c) Dibuje el árbol de derivación y verifique que refleja la precedencia

d) ¿Es su gramática ambigua? Justifique su respuesta

3.7.3 Análisis de ambigüedad en lenguajes naturales

Un laboratorio de procesamiento de lenguaje natural está investigando cómo las gramáticas libres de contexto modelan ambigüedades del español. Considere la siguiente gramática simplificada:

$$\begin{aligned} S &\to SN\,SV \\ SN &\to Det\,N \mid Det\,N\,Prep\,SN \\ SV &\to V \mid V\,SN \\ Det &\to \text{el} \mid \text{la} \\ N &\to \text{niño} \mid \text{niña} \mid \text{telescopio} \\ V &\to \text{vio} \\ Prep &\to \text{con} \end{aligned}$$

a) Demuestre que la oración "el niño vio la niña con el telescopio" es ambigua mostrando dos árboles de derivación distintos

b) Explique el significado diferente que representa cada árbol

c) Proponga una modificación a la gramática que elimine la ambigüedad especificando que la frase preposicional modifica al verbo, no al sustantivo

3.7.4 Transformaciones de gramáticas

Un sistema de optimización de parsers necesita convertir gramáticas a Forma Normal de Chomsky. Dada la gramática:

$$\begin{aligned} S &\to aSb \mid A \\ A &\to bA \mid B \\ B &\to a \mid \varepsilon \end{aligned}$$

a) Identifique todas las producciones ε (variables anulables)

b) Elimine las producciones ε obteniendo una gramática equivalente

c) Convierta la gramática resultante a Forma Normal de Chomsky

d) Verifique que ambas gramáticas generan el mismo lenguaje con ejemplos

3.7.5 Autómatas de pila

Una empresa de verificación de protocolos necesita implementar autómatas de pila para validar formatos de mensajes. Diseñe un PDA que reconozca cada uno de los siguientes lenguajes:

a) $L_1 = \{0^n 1^{2n} \mid n \geq 0\}$ (el doble de 1's que de 0's)

b) $L_2 = \{ww^R \mid w \in \{a, b\}^*\}$ (cadenas duplicadas en reversa)

c) $L_3 = \{a^i b^j \mid i \leq j \leq 2i\}$

Para cada PDA, especifique formalmente:

- Estados Q
- Alfabetos Σ y Γ
- Función de transición δ
- Estado inicial q_0 y símbolo inicial de pila Z_0
- Criterio de aceptación (estado final o pila vacía)

3.7.6 Gramática para Expresiones Simbólicas (LISP)

Una empresa que mantiene un intérprete de Lisp necesita formalizar la sintaxis del lenguaje. Diseñe una gramática BNF (o EBNF) para representar expresiones simbólicas (S-Expressions) que soporte:

- **Átomos**: Pueden ser identificadores (letras) o números.
- **Listas**: Secuencias de cero o más expresiones separadas por espacios y encerradas entre paréntesis.
- **Anidamiento**: Las listas pueden contener átomos u otras listas arbitrariamente.

Ejemplos válidos:

- `atom`
- `(1 2 3)`

- `(+ 1 2)`
- `(defun cuadrado (x) (* x x))`
- `((a) b (c (d)))`

Asegúrese de que su gramática permita el anidamiento recursivo infinito y maneje correctamente la lista vacía `()`.

3.8. Cierre del capítulo

En este capítulo se presentaron las herramientas para analizar estructuras jerárquicas, un problema recurrente en ciencias de la computación que abarca desde la validación de paréntesis balanceados hasta el análisis sintáctico de programas completos.

Conceptos clave aprendidos

1. **Gramáticas Libres de Contexto (GLC):** Formalismos que describen lenguajes mediante reglas recursivas de la forma $A \to \alpha$. Permiten representar estructuras anidadas y jerárquicas que no pueden expresarse con lenguajes regulares.

2. **Notación BNF/EBNF:** Forma estándar utilizada para especificar gramáticas de lenguajes de programación. Se basa en las GLC, pero emplea una sintaxis más clara y compacta.

3. **Derivaciones y árboles de derivación:** Herramientas que permiten generar cadenas desde el símbolo inicial y representar gráficamente su estructura sintáctica.

4. **Ambigüedad gramatical:** Una gramática es ambigua si una misma cadena puede generar varios árboles de derivación. En los lenguajes de programación, la ambigüedad debe evitarse para asegurar una interpretación única del código.

5. **Formas Normales:** Restricciones sobre las producciones gramaticales, como la Forma Normal de Chomsky, que facilitan el análisis teórico y el diseño de algoritmos de parsing sin modificar el lenguaje generado.

6. **Autómatas de Pila (PDA):** Extensión de los autómatas finitos que incorpora una pila LIFO como mecanismo de memoria. Esto permite reconocer lenguajes libres de contexto. Los PDA y las GLC poseen el mismo poder expresivo.

7. **Propiedades de clausura:** Los LLC son cerrados bajo unión, concatenación y clausura de Kleene, pero *no* bajo intersección ni complemento, a diferencia de los lenguajes regulares.

8. **Análisis sintáctico (Parsing):** Proceso mediante el cual se construye la estructura jerárquica de un programa. Puede implementarse mediante técnicas como el parser recursivo descendente estudiado en este capítulo.

Limitaciones de los lenguajes libres de contexto

Si bien los lenguajes libres de contexto son significativamente más potentes que los regulares, existen lenguajes que requieren una memoria más compleja que una pila o que implican múltiples comparaciones simultáneas. Estos lenguajes **no son libres de contexto**.

Ejemplos clásicos de lenguajes que **NO** son LLC:

- $L = \{a^n b^n c^n \mid n \geq 0\}$: Requiere contar tres grupos de símbolos simultáneamente. Un autómata de pila puede verificar que $\#a = \#b$ o que $\#b = \#c$, pero no ambas condiciones a la vez.
- $L = \{ww \mid w \in \{a, b\}^*\}$: Requiere verificar la duplicación exacta de una cadena arbitraria. La pila no permite comparar posición por posición sin destruir la información almacenada.

Propiedades de clausura

Decimos que una clase de lenguajes es **cerrada** bajo una operación si, al aplicar esa operación a lenguajes de la clase, el resultado siempre pertenece a la misma clase. A diferencia de los lenguajes regulares (cerrados bajo todas las operaciones básicas), los LLC presentan diferencias importantes:

Operación	**¿Es cerrado?**	**Observación**
Unión ($\cup$)	**SÍ**	La unión de dos LLC es un LLC.
Concatenación ($\cdot$)	**SÍ**	La concatenación de dos LLC es un LLC.
Clausura de Kleene ($*$)	**SÍ**	L^* es LLC si L es LLC.
Intersección ($\cap$)	**NO**	La intersección de dos LLC **no** garantiza ser un LLC.
Complemento ($\overline{L}$)	**NO**	El complemento de un LLC **no** garantiza ser un LLC.

¿Por qué la intersección no es cerrada? Consideremos los lenguajes

$$L_1 = \{a^n b^n c^m \mid n, m \geq 0\}$$

y

$$L_2 = \{a^m b^n c^n \mid m, n \geq 0\}.$$

Ambos son LLC. Sin embargo,

$$L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\},$$

y este lenguaje no es libre de contexto.

¿Por qué el complemento no es cerrado? Esto puede demostrarse utilizando las leyes de De Morgan:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}.$$

Si los LLC fueran cerrados bajo complemento, entonces también serían cerrados bajo intersección, ya que la unión sí es cerrada. Sin embargo, el ejemplo anterior muestra que esto no ocurre.

Caso especial importante: La intersección entre un LLC y un lenguaje *regular* siempre produce otro LLC. Este resultado es especialmente útil en compiladores, donde el análisis léxico (regular) se combina con el análisis sintáctico (libre de contexto).

Motivación para el siguiente capítulo

Las limitaciones anteriores nos llevan naturalmente a preguntarnos:

¿Existe un modelo computacional que no tenga estas restricciones?

¿Cuáles son los límites absolutos de lo que puede ser computado?

En el siguiente capítulo estudiaremos las **Máquinas de Turing** y el **cálculo lambda**, dos modelos que marcan el límite teórico de lo que una máquina puede hacer. Analizaremos qué problemas pueden resolverse mediante un algoritmo y cuáles, por su propia naturaleza, no admiten solución alguna, estableciendo los fundamentos de la teoría de la computación.

Capítulo 4

Máquinas de Turing y Cálculo Lambda

¿Cuáles son los límites fundamentales de lo que puede ser computado?

En los capítulos anteriores se estudiaron modelos computacionales con capacidades limitadas. Los autómatas finitos permiten reconocer patrones simples, pero no pueden contar cantidades arbitrarias, mientras que los autómatas de pila pueden manejar estructuras anidadas, aunque siguen teniendo restricciones importantes.

Sin embargo, existen problemas que requieren un modelo de computación más general, por ejemplo:

- verificar que distintas secciones de un archivo contengan la misma cantidad de registros,
- determinar si un programa finalizará su ejecución o entrará en un ciclo infinito,
- describir cálculos completos utilizando únicamente funciones.

Este capítulo introduce dos de los modelos más importantes de la teoría de la computación: las **Máquinas de Turing**, basadas en una cinta de memoria potencialmente infinita, y el **Cálculo Lambda**, un modelo fundamentado en la aplicación de funciones. Aunque fueron desarrollados de manera independiente, ambos poseen el mismo poder computacional, idea que sustenta la **Tesis de Church-Turing**.

A lo largo del capítulo se presentarán distintos problemas que motivan el estudio progresivo de estos conceptos.

Nota histórica

En 1936, dos matemáticos publicaron trabajos independientes que definirían los fundamentos de la computación:
Alan Turing introdujo la "Máquina de Turing" en su artículo *"On Computable Numbers, with an Application to the Entscheidungsproblem"* [Turing, 1936]. Su objetivo era formalizar la noción de "procedimiento mecánico" para demostrar que el problema de decisión planteado por Hilbert y Ackermann [Hilbert and Ackermann, 1928] era irresoluble.
Alonzo Church, simultáneamente, desarrolló el **Cálculo Lambda**, un sistema formal basado en funciones [Church, 1936]. Church también demostró la indecidibilidad del problema de Hilbert usando su formalismo.
Aunque fueron desarrollados de manera independiente, ambos modelos resultaron ser **equivalentes en poder expresivo**. Esta coincidencia condujo a la **Tesis de Church-Turing**: todo lo intuitivamente computable puede ser computado por estos modelos.

4.1. Reconocimiento de cadenas con dependencia múltiple

Una empresa de análisis de datos, **DataVerify**, se especializa en el desarrollo de herramientas de control de calidad para grandes volúmenes de información. Se requiere un software que permita validar automáticamente la integridad estructural de archivos de datos enviados por sus clientes. Estos archivos están organizados en múltiples secciones, y una de las validaciones críticas consiste en garantizar que tres secciones específicas del archivo contengan exactamente la misma cantidad de registros. Si esta condición no se cumple, el archivo debe ser rechazado antes de ser procesado. Se ha determinado que un autómata de pila puede verificar que dos secciones coincidan en cantidad, pero no tres simultáneamente.

¿Qué tipo de modelo computacional puede manejar este tipo de dependencias?

4.1.1 Análisis / Abstracción

a. ¿Cuál es el problema a resolver?

Dado un archivo o cadena con estructura de la forma $a^n b^n c^n$ (tres bloques con exactamente la misma cantidad de símbolos), determinar si la cadena es válida.

b. ¿Cuál es la información necesaria?

Para poder resolver el problema es necesaria la siguiente información:

Información necesaria	Descripción
Cadena de entrada	El texto a verificar
Restricción	Tres bloques con exactamente la misma cantidad de símbolos

Las limitaciones conocidas son: los autómatas finitos tienen memoria fija (estados) y los autómatas de pila tienen memoria auxiliar con acceso restringido (solo la cima). Se necesita un modelo que supere estas restricciones.

4.1.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **ENTRADA:** Una cadena con tres bloques de símbolos (por ejemplo, a's, b's y c's).
- **PROCESAMIENTO:** Verificar que los tres bloques tengan exactamente la misma cantidad de símbolos.
- **SALIDA:** Indicar si la cadena cumple con la restricción de igualdad triple o no.

b. ¿Qué conocimiento adicional se requiere?

Resolver este problema requiere un modelo computacional más general que los autómatas de pila: la **Máquina de Turing**, la cual dispone de

una cinta de memoria potencialmente infinita y permite leer y escribir en distintas posiciones, a diferencia del acceso limitado a la cima de una pila.

Máquinas de Turing: idea intuitiva

Antes de presentar la definición formal, resulta útil entender intuitivamente cómo funciona una Máquina de Turing.

Imaginemos una persona sentada frente a una cinta de papel infinitamente larga, dividida en casillas. La persona tiene un lápiz y puede:

1. **Leer** el símbolo escrito en la casilla actual.
2. **Escribir** un símbolo nuevo (borrando el anterior).
3. **Moverse** una casilla a la izquierda o a la derecha.
4. **Recordar** en qué "modo" o "estado mental" se encuentra.

La persona sigue un conjunto fijo de instrucciones del tipo: *"Si estoy en el estado X y veo el símbolo Y, entonces escribo Z, me muevo a la derecha, y paso al estado W"*. Aunque el modelo parece sencillo, esta idea permite describir cualquier proceso computacional.

Componentes principales

Una Máquina de Turing (MT) está formada por los siguientes componentes:

- **Cinta**: secuencia infinita de celdas, cada una con un símbolo. Al inicio contiene la entrada, y el resto de posiciones se consideran vacías. Estas posiciones vacías suelen representarse mediante el símbolo especial $\sqcup$, denominado símbolo blanco (*blank*), distinto de cualquier símbolo del alfabeto de entrada.
- **Cabeza**: apunta a una celda de la cinta. Puede leer su contenido, escribir un nuevo símbolo y moverse una posición a la izquierda (L) o a la derecha (R).
- **Estados**: describen la configuración interna de la máquina. Se distinguen un estado inicial, un estado de aceptación y un estado de rechazo.

- **Reglas de transición**: indican qué acción realizar en función del estado actual y del símbolo leído.

Definición formal

Ahora traducimos la descripción intuitiva a notación matemática. Cada componente que describimos informalmente tiene su contraparte formal.

Una *Máquina de Turing* (MT) es una séptupla [Sipser, 2012] $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ donde:

- Q: conjunto finito de **estados**
- Σ: **alfabeto de entrada** (sin el símbolo blanco)
- Γ: **alfabeto de cinta** (incluye Σ y el blanco $\sqcup$)
- δ: **función de transición** (las reglas)
- q_0: **estado inicial**
- q_{accept}, q_{reject}: estados de **aceptación** y **rechazo**

La función de transición δ es el "cerebro" de la máquina: contiene todas las reglas. Su tipo formal es:

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

Esto se lee: "δ toma un estado y un símbolo de cinta, y devuelve un nuevo estado, un símbolo a escribir, y una dirección".

Por ejemplo, si tenemos:

$$\delta(q, a) = (q', b, D)$$

significa: *"estando en estado q y leyendo a, escribir b, moverse en dirección D, y pasar al estado q' "*.

Notación formal	Significado intuitivo
Q	Los "modos" o "estados mentales" de la persona
Σ	Los símbolos que puede tener la entrada
Γ	Todos los símbolos que pueden aparecer en la cinta
δ	La tabla de instrucciones ("si veo X, hago Y")
q_0	El modo inicial al comenzar
q_{accept}	El modo de "terminé y la respuesta es sí"
q_{reject}	El modo de "terminé y la respuesta es no"

¿Cómo funciona una MT?

El proceso de ejecución es:

1. La entrada se escribe en la cinta; el resto son blancos ($\sqcup$).
2. La cabeza se posiciona en el primer símbolo; el estado es q_0.
3. En cada paso, la máquina consulta δ y ejecuta la transición.
4. La máquina se **detiene** cuando llega a q_{accept} o q_{reject}.
5. Si nunca llega a estos estados, la máquina **no termina** (ciclo infinito).

Notación de configuraciones

Para mostrar cómo funciona una MT paso a paso, usamos una notación llamada **configuración**. Una configuración describe el estado completo de la máquina en un instante: qué hay en la cinta, dónde está la cabeza, y en qué estado se encuentra.

Convención: Escribimos el contenido de la cinta con el estado insertado *justo antes* del símbolo que la cabeza está leyendo. Por ejemplo:

```
X0q1 11
```

significa:

- La cinta contiene `X011`
- La máquina está en el estado q_1
- La cabeza está posicionada sobre el primer `1` (el símbolo inmediatamente después del estado)

Visualmente, sería así:

X	0	**1**	1	⊔	⊔

↑

Estado: q_1

Una **traza** es una secuencia de configuraciones conectadas por flechas ($\rightarrow$), mostrando cómo la máquina evoluciona paso a paso. Usamos _ para representar el blanco ⊔ en las trazas.

Ejemplo 1: MT para verificar cadenas que empiezan con 0

Empecemos con un ejemplo sencillo. Diseñemos una MT que acepte cadenas sobre $\{0, 1\}$ que comiencen con 0.

Estados: q_0 (inicial), q_{accept}, q_{reject}

Reglas:

- En q_0, si leo 0: aceptar.
- En q_0, si leo 1 o blanco: rechazar.

Transiciones:

Estado, Símbolo	**Transición**	**Significado**
q_0, 0	q_{accept}, 0, R	Empieza con 0, aceptar
q_0, 1	q_{reject}, 1, R	Empieza con 1, rechazar
q_0, ⊔	q_{reject}, ⊔, R	Cadena vacía, rechazar

Cadena aceptada: $w = 0110$

```
q0 0110  ->  0q_accept 110  ->  ACEPTAR
```

La máquina inicia en q_0 y lee el símbolo 0. Según la regla $\delta(q_0, 0) = (q_{accept}, 0, R)$, escribe 0, se mueve a la derecha y pasa al estado de aceptación.

Cadena rechazada: $w = 1001$

```
q0 1001  ->  1q_reject 001  ->  RECHAZAR
```

Ejemplo 2: MT para $\{0^n 1^n \mid n \geq 1\}$

Este lenguaje (igual cantidad de 0's seguidos de 1's) **no es regular**, pero sí es libre de contexto. Una MT puede reconocerlo marcando símbolos de la cinta.

Estrategia intuitiva:

1. Buscar el primer 0 y "tacharlo" (marcarlo con X).
2. Ir a la derecha hasta encontrar el primer 1 y "tacharlo" (marcarlo con Y).
3. Volver al inicio y repetir.
4. Si al final todos están tachados (solo quedan X's e Y's), aceptar.

¿Por qué funciona? Cada vez que tachamos un 0, tachamos exactamente un 1. Si la cantidad de 0's y 1's es igual, al final todos estarán tachados.

Estados y su propósito:

- q_0: Buscar el siguiente 0 no marcado
- q_1: Ir a la derecha buscando un 1 no marcado

- q_2: Volver a la izquierda al inicio
- q_3: Verificar que solo queden marcas

Función de transición:

Estado, Símbolo	Transición	Descripción
q_0, 0	q_1, X, R	Marcar 0, buscar 1
q_0, Y	q_3, Y, R	No hay más 0's, verificar
q_1, 0	q_1, 0, R	Saltar 0's
q_1, Y	q_1, Y, R	Saltar Y's
q_1, 1	q_2, Y, L	Marcar 1, volver
q_2, 0	q_2, 0, L	Retroceder
q_2, Y	q_2, Y, L	Retroceder
q_2, X	q_0, X, R	Llegué al inicio, repetir
q_3, Y	q_3, Y, R	Saltar Y's
q_3, ⊔	q_{accept}, ⊔, R	Todo marcado, aceptar

Traza para $w = 0011$:

```
q0 0011  ->  Xq1 011  ->  X0q1 11  ->  Xq2 0Y1
         ->  q2 X0Y1  ->  Xq0 0Y1  ->  XXq1 Y1
         ->  XXYq1 1  ->  XXq2 YY  ->  Xq2 XYY
         ->  XXq0 YY  ->  XXYq3 Y  ->  XXYYq3 _
         ->  ACEPTAR
```

Explicación de los primeros pasos:

1. `q0 0011`: Estado inicial, cabeza sobre el primer 0.
2. `Xq1 011`: Marcamos el 0 con X, avanzamos, buscamos un 1.
3. `X0q1 11`: Saltamos el segundo 0, seguimos buscando.
4. `Xq2 0Y1`: Encontramos el 1, lo marcamos con Y, **retrocedemos**.
5. `q2 X0Y1`: Seguimos retrocediendo hasta encontrar X.
6. `Xq0 0Y1`: Encontramos X, avanzamos y repetimos el ciclo.

c. ¿Qué se puede reutilizar de soluciones pasadas?

La estructura de lectura, validación e impresión de resultados es la misma que en capítulos anteriores. Además, la estrategia de "marcar símbolos ya procesados" es análoga a la técnica de "empujar y sacar de la pila" de los PDA, pero con más libertad de acceso.

d. ¿Cuál es el diseño de la solución?

El lenguaje que describe este problema es

$$\{a^n b^n c^n \mid n \geq 1\}.$$

Este lenguaje **no es libre de contexto**, por lo que no puede ser reconocido mediante un autómata de pila. Un PDA dispone de una única pila y, después de contar los símbolos a y verificar la misma cantidad de símbolos b, ya no conserva suficiente información para comprobar la cantidad de símbolos c.

Una MT, en cambio, puede recorrer la cinta varias veces y marcar posiciones previamente visitadas, utilizando la propia cinta como memoria auxiliar.

Estrategia intuitiva:

1. Buscar la primera a no marcada y marcarla con X.
2. Ir a la derecha hasta encontrar la primera b no marcada y marcarla con Y.
3. Continuar a la derecha hasta encontrar la primera c no marcada y marcarla con Z.
4. Volver al inicio y repetir.
5. Cuando no queden a's sin marcar, verificar que tampoco queden b's ni c's sin marcar.

Estados y su propósito:

- q_0: Buscar la siguiente a no marcada

- q_1: Ir a la derecha buscando una b no marcada
- q_2: Ir a la derecha buscando una c no marcada
- q_3: Volver a la izquierda al inicio
- q_4: Verificar que solo queden marcas

Función de transición:

Estado, Símbolo	Transición	Descripción
q_0, a	q_1, X, R	Marcar a, buscar b
q_0, Y	q_4, Y, R	No hay más a's, verificar
q_1, a	q_1, a, R	Saltar a's
q_1, Y	q_1, Y, R	Saltar Y's
q_1, b	q_2, Y, R	Marcar b, buscar c
q_2, b	q_2, b, R	Saltar b's
q_2, Z	q_2, Z, R	Saltar Z's
q_2, c	q_3, Z, L	Marcar c, volver
q_3, a	q_3, a, L	Retroceder sobre a
q_3, b	q_3, b, L	Retroceder sobre b
q_3, Y	q_3, Y, L	Retroceder sobre Y
q_3, Z	q_3, Z, L	Retroceder sobre Z
q_3, X	q_0, X, R	Llegué al inicio, repetir
q_4, Y	q_4, Y, R	Saltar Y's
q_4, Z	q_4, Z, R	Saltar Z's
q_4, ⊔	q_{accept}, ⊔, R	Todo marcado, aceptar

4.1.3 Ejemplos de pruebas

a. Ejemplo 1

Entrada: $w = abc$ **Esperado**: ACEPTADA

```
q0 abc   ->  Xq1 bc   ->  XYq2 c   ->  Xq3 YZ
         ->  q3 XYZ   ->  Xq0 YZ   ->  XYq4 Z
         ->  XYZq4 _  ->  ACEPTAR
```

La cadena tiene una a, una b y una c. La MT marca la a como X, la b como Y y la c como Z en un solo ciclo. Al volver, no encuentra más a's y verifica que solo hay marcas. Resultado correcto.

b. Ejemplo 2

Entrada: $w = aabbcc$ **Esperado**: ACEPTADA

```
q0 aabbcc  ->  Xq1 abbcc  ->  Xaq1 bbcc
           ->  XaYq2 bcc  ->  XaYbq2 cc
           ->  XaYq3 bZc  ->  Xaq3 YbZc
           ->  Xq3 aYbZc  ->  q3 XaYbZc
           ->  Xq0 aYbZc  ->  XXq1 YbZc
           ->  XXYq1 bZc  ->  XXYYq2 Zc
           ->  XXYYZq2 c  ->  XXYYq3 ZZ
           ->  XXYq3 YZZ  ->  XXq3 YYZZ
           ->  Xq3 XYYZZ  ->  XXq0 YYZZ
           ->  XXYq4 YZZ  ->  XXYYq4 ZZ
           ->  XXYYZq4 Z  ->  XXYYZZq4 _  ->  ACEPTAR
```

La MT realiza dos ciclos completos de marcado, emparejando cada a con una b y una c. Al final, la cinta contiene XXYYZZ y la máquina acepta.

c. Ejemplo 3

Entrada: $w = aabcc$ **Esperado**: RECHAZADA

```
q0 aabcc  ->  Xq1 abcc  ->  Xaq1 bcc
          ->  XaYq2 cc  ->  Xaq3 YZc
          ->  Xq3 aYZc  ->  q3 XaYZc
          ->  Xq0 aYZc  ->  XXq1 YZc
          ->  XXYq1 Zc  ->  RECHAZAR
```

En el primer ciclo, la MT marca una a, una b y una c sin problemas. En el segundo ciclo, marca la segunda a como X y avanza buscando una b sin marcar. Salta la Y (marca de la primera b), pero encuentra una Z (marca de la primera c). No existe transición para (q_1, Z) porque en estado q_1 la máquina solo espera a's, Y's o b: al no encontrar ninguna b sin marcar, la máquina se detiene y rechaza. Esto demuestra que $aabcc \notin \{a^n b^n c^n\}$.

4.1.4 Codificación

a. Simulador genérico de Máquina de Turing

La clase `MaquinaTuring` almacena las transiciones en un diccionario donde cada clave corresponde a un par `(estado, símbolo)`, y cada valor indica el nuevo estado, el símbolo que debe escribirse y la dirección del movimiento de la cabeza.

El método `ejecutar()` procesa la cinta transición por transición hasta alcanzar un estado de aceptación, un estado de rechazo o un límite máximo de pasos.

b. Simulador genérico de Máquina de Turing

La clase `MaquinaTuring` recibe las transiciones como un diccionario donde cada clave es una tupla `(estado, símbolo)` y cada valor es una tupla `(nuevo_estado, símbolo_a_escribir, dirección)`. El método `ejecutar` simula la máquina paso a paso hasta alcanzar un estado de aceptación, rechazo, o un límite de pasos.

Código 4.1: Clase MaquinaTuring --- simulador genérico

```
class MaquinaTuring:
    def __init__(self, transiciones, q_inicial, q_aceptar, q_rechazar):
        self.transiciones = transiciones
        self.q_inicial = q_inicial
        self.q_aceptar = q_aceptar
        self.q_rechazar = q_rechazar

    def ejecutar(self, entrada, max_pasos=10000):
        cinta = list(entrada) if entrada else ['_']
        posicion = 0
        estado = self.q_inicial
        pasos = 0
        while pasos < max_pasos:
            if estado == self.q_aceptar:
                return True, ''.join(cinta).rstrip('_'), pasos
            if estado == self.q_rechazar:
                return False, ''.join(cinta).rstrip('_'), pasos
            # Leer símbolo actual (blanco si está fuera de la cinta)
            simbolo = cinta[posicion] if posicion < len(cinta) else '_'
            clave = (estado, simbolo)
            # Sin transición definida: rechazar
            if clave not in self.transiciones:
                return False, ''.join(cinta).rstrip('_'), pasos
            nuevo_estado, escribir, direccion = self.transiciones[clave]
            # Extender cinta si la cabeza está más allá del final
            while posicion >= len(cinta):
                cinta.append('_')
            cinta[posicion] = escribir
            if direccion == 'R':
                posicion += 1
            elif direccion == 'L' and posicion > 0:
                posicion -= 1
            estado = nuevo_estado
            pasos += 1
        return None, ''.join(cinta).rstrip('_'), pasos
```

c. Implementación de la MT para $a^n b^n c^n$

La implementación se apoya en el simulador genérico, particularizándolo con la tabla de transiciones diseñada para este lenguaje.

La función `main` articula el flujo: lee la cadena, instancia la MT con sus transiciones y reporta el resultado junto con el número de pasos consumidos.

Código 4.2: Función principal

```
def main():
    cadena = leer("Ingrese la cadena (a's, b's y c's): ")
    mt = crear_mt_anbncn()
    resultado, cinta, pasos = mt.ejecutar(cadena)
    imprimir_resultado(cadena, resultado, pasos)
```

Dos funciones auxiliares atienden la consola: una recoge la cadena del usuario y la otra imprime el resultado.

Código 4.3: Funciones de entrada y salida

```
def leer(texto):
    return input(texto)

def imprimir_resultado(cadena, es_aceptada, pasos):
    if es_aceptada:
        print(f"'{cadena}': ACEPTADA ({pasos} pasos)")
    else:
        print(f"'{cadena}': RECHAZADA ({pasos} pasos)")
```

Por último, la función de creación traduce a código todas las transiciones del diseño.

Código 4.4: Creación de la MT para $a^n b^n c^n$

```
def crear_mt_anbncn():
    transiciones = {
        ('q0', 'a'): ('q1', 'X', 'R'),
        ('q0', 'Y'): ('q4', 'Y', 'R'),
        ('q1', 'a'): ('q1', 'a', 'R'),
        ('q1', 'Y'): ('q1', 'Y', 'R'),
        ('q1', 'b'): ('q2', 'Y', 'R'),
        ('q2', 'b'): ('q2', 'b', 'R'),
        ('q2', 'Z'): ('q2', 'Z', 'R'),
        ('q2', 'c'): ('q3', 'Z', 'L'),
        ('q3', 'a'): ('q3', 'a', 'L'),
        ('q3', 'b'): ('q3', 'b', 'L'),
        ('q3', 'Y'): ('q3', 'Y', 'L'),
        ('q3', 'Z'): ('q3', 'Z', 'L'),
        ('q3', 'X'): ('q0', 'X', 'R'),
        ('q4', 'Y'): ('q4', 'Y', 'R'),
        ('q4', 'Z'): ('q4', 'Z', 'R'),
        ('q4', '_'): ('qa', '_', 'R'),
    }
    return MaquinaTuring(transiciones, 'q0', 'qa', 'qr')
```

4.2. Equivalencia de modelos de cómputo

La empresa **ChipArch**, especializada en diseño de procesadores para aplicaciones específicas, necesita determinar si construir procesadores con múltiples unidades de memoria aporta mayor poder computacional que un procesador con una sola unidad. El equipo de arquitectura quiere saber si estas variantes más complejas pueden resolver problemas que el modelo básico no puede, o si solo ofrecen ventajas de eficiencia.

4.2.1 Análisis / Abstracción

a. ¿Cuál es el problema a resolver?

ChipArch necesita determinar si las variantes de Máquinas de Turing (múltiples cintas, no determinismo, cinta bidireccional) pueden resolver problemas que la MT básica de una cinta no puede. De la respuesta depende si la inversión en hardware más complejo se justifica por mayor poder computacional o solo por mayor eficiencia.

b. ¿Cuál es la información necesaria?

Para responder a la pregunta de ChipArch hay que fijar un punto de referencia, las extensiones bajo evaluación y el criterio que decide la comparación:

Información necesaria	Descripción
Modelo básico	El procesador con una sola unidad de memoria
Variantes a evaluar	Procesadores con múltiples unidades u otras extensiones
Criterio de comparación	Si resuelven los mismos problemas o no

4.2.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **ENTRADA:** Las variantes de procesador a comparar con el modelo básico.

- **PROCESAMIENTO:** Determinar si cada variante puede resolver problemas que el modelo básico no puede, o si solo ofrece ventajas de diseño.
- **SALIDA:** Conclusión sobre si las variantes añaden poder computacional o solo conveniencia.

b. ¿Qué conocimiento adicional se requiere?

Variantes de Máquinas de Turing

El modelo básico de Máquina de Turing utiliza una sola cinta y funciona de manera determinista. Sin embargo, existen otras variantes que modifican algunas de sus características.

MT con múltiples cintas. En lugar de una única cinta, la máquina dispone de k cintas, cada una con su propia cabeza de lectura y escritura. En cada paso puede leer los símbolos de todas las cintas y moverse de manera independiente sobre cada una de ellas.

Esta variante facilita la construcción de algoritmos, ya que permite separar distintas tareas entre las cintas. Por ejemplo, una cinta puede almacenar la entrada, otra utilizarse como memoria de trabajo y otra contener el resultado.

Ejemplo intuitivo: para sumar dos números, el primer operando podría ubicarse en la cinta 1, el segundo en la cinta 2 y el resultado escribirse en la cinta 3. Con una sola cinta, la máquina tendría que desplazarse continuamente entre los operandos y la zona de escritura.

MT no determinista. En una MT no determinista, la función de transición puede definir varias acciones posibles para un mismo estado y símbolo leído. La máquina acepta la entrada si al menos una secuencia de decisiones conduce a un estado de aceptación.

Cinta infinita en ambas direcciones. En el modelo clásico, la cinta es infinita únicamente hacia la derecha. Algunas variantes permiten extenderla infinitamente en ambas direcciones.

El teorema de equivalencia.

Teorema [Hopcroft et al., 2006, Sipser, 2012, Hennie and Stearns, 1966]**:** Todas las variantes mencionadas son **equivalentes en poder computacional** a la MT básica de una cinta determinista. Reconocen la misma clase de lenguajes.

El teorema tiene tres consecuencias inmediatas: las variantes son más *convenientes* o *eficientes*, pero no más *poderosas*; cualquier problema que resuelva una MT de 10 cintas también puede resolverlo una MT de 1 cinta; y el concepto de "computable" resulta estable, sin depender de los detalles del modelo. Esto permite usar el modelo más conveniente para cada situación sin ganar ni perder poder computacional.

c. ¿Qué se puede reutilizar de soluciones pasadas?

De la sección anterior se reutiliza el simulador genérico de MT y la estrategia de marcado. La implementación de $\{a^n b^n c^n\}$ demuestra que, aunque el diseño con múltiples cintas es más intuitivo, el modelo básico de una cinta es suficiente para resolver el mismo problema.

d. ¿Cuál es el diseño de la solución?

Para responder la pregunta de ChipArch, se diseña una MT de **2 cintas** para el mismo lenguaje $\{a^n b^n c^n\}$ de la sección anterior, y se compara con la MT de 1 cinta ya diseñada.

MT de 2 cintas para $\{a^n b^n c^n\}$

La segunda cinta actúa como memoria auxiliar independiente. La estrategia es lineal: contar las a's en la cinta 2, verificar que haya la misma cantidad de b's, y luego la misma cantidad de c's.

Estrategia:

1. **Fase 1** ($q_i \to q_0$): Escribir un centinela `$` al inicio de la cinta 2.

2. **Fase 2** (q_0): Para cada a en la cinta 1, escribir una X en la cinta 2. Ambas cabezas avanzan a la derecha.

3. **Fase 3** (q_1): Al encontrar la primera b, la cinta 2 retrocede. Para cada b en la cinta 1, la cinta 2 retrocede una posición sobre las X's. Al llegar al centinela \$, se confirma que $\#b = \#a$.

4. **Fase 4** (q_2): Para cada c en la cinta 1, la cinta 2 avanza sobre las X's. Al terminar, si ambas cintas están en blanco simultáneamente, se confirma que $\#c = \#a$ y se acepta.

Función de transición (2 cintas):

En cada transición se indica: estado actual, (símbolo cinta 1, símbolo cinta 2) → nuevo estado, (escritura cinta 1, escritura cinta 2), (dirección cinta 1, dirección cinta 2). La dirección S significa "quedarse".

Estado, Símbolos	**Transición**	**Descripción**
q_i, (a, ⊔)	q_0, (a, \$), (S, R)	Escribir centinela
q_0, (a, ⊔)	q_0, (a, X), (R, R)	Contar a: escribir X
q_0, (b, ⊔)	q_1, (b, ⊔), (S, L)	Fin de a's, retroceder cinta 2
q_1, (b, X)	q_1, (b, X), (R, L)	Verificar b: retroceder sobre X
q_1, (c, \$)	q_2, (c, \$), (S, R)	$\#b = \#a$ confirmado
q_1, (b, \$)	q_r, (b, \$), (S, S)	Más b's que a's: rechazar
q_1, (c, X)	q_r, (c, X), (S, S)	Menos b's que a's: rechazar
q_2, (c, X)	q_2, (c, X), (R, R)	Verificar c: avanzar sobre X
q_2, (⊔, ⊔)	q_a, (⊔, ⊔), (S, S)	$\#c = \#a$ confirmado: aceptar
q_2, (c, ⊔)	q_r, (c, ⊔), (S, S)	Más c's que a's: rechazar
q_2, (⊔, X)	q_r, (⊔, X), (S, S)	Menos c's que a's: rechazar

Comparación: 1 cinta vs. 2 cintas

Característica	**1 cinta**	**2 cintas**
Transiciones	16	11
Estados (sin q_a, q_r)	5	4
Recorrido de la cinta	Múltiples pasadas	Una sola pasada
Lenguaje reconocido	$\{a^n b^n c^n\}$	$\{a^n b^n c^n\}$

Ambas MT reconocen el mismo lenguaje. La versión con 2 cintas resulta más sencilla de diseñar y suele ser más eficiente, pero no puede resolver problemas que una MT de 1 cinta no pueda resolver. Esto muestra que agregar más recursos puede simplificar la computación y reducir el tiempo de ejecución, aunque sin aumentar el poder computacional del modelo.

4.2.3 Ejemplos de pruebas

Se ejecutan las mismas cadenas en ambas MT para verificar que producen el mismo resultado.

a. Ejemplo 1

Entrada: $w = abc$ **Esperado**: ACEPTADA en ambas MT

MT de 1 cinta (8 pasos): Marca a con X, busca b y la marca con Y, busca c y la marca con Z. Vuelve al inicio, no encuentra más a's y verifica que todo esté marcado.

MT de 2 cintas (7 pasos): Escribe centinela, copia la a como X en cinta 2. Verifica la b retrocediendo al centinela. Verifica la c avanzando sobre la X. Ambas cintas terminan en blanco: acepta.

b. Ejemplo 2

Entrada: $w = aaabbbccc$ **Esperado**: ACEPTADA en ambas MT

MT de 1 cinta (46 pasos): Realiza tres ciclos completos de marcado, recorriendo la cinta de extremo a extremo en cada uno.

MT de 2 cintas (13 pasos): Escribe 3 X's en la cinta 2, retrocede 3 posiciones al verificar las b's, avanza 3 posiciones al verificar las c's. Una sola pasada.

La diferencia de pasos (46 vs. 13) crece con el tamaño de la entrada, pero el resultado es siempre el mismo.

c. Ejemplo 3

Entrada: $w = aabcc$ **Esperado**: RECHAZADA en ambas MT

MT de 1 cinta (9 pasos): Marca la primera a y la primera b. En el segundo ciclo, marca la segunda a pero no encuentra una segunda b. Rechaza.

MT de 2 cintas (6 pasos): Escribe 2 X's en cinta 2. Al verificar la única b, retrocede una posición pero encuentra X en vez de \$, lo que indica menos b's que a's. Rechaza.

4.2.4 Codificación

a. Simulador de MT de múltiples cintas

La clase `MaquinaTuringMulticinta` amplía el simulador básico para trabajar con varias cintas. Cada transición recibe una tupla de símbolos (uno por cinta) y produce las correspondientes tuplas de escritura y movimiento. La dirección `S` indica que la cabeza permanece en la misma posición.

La primera parte de la clase implementa la inicialización y las operaciones necesarias para leer el contenido de las cintas:

Código 4.5: Clase `MaquinaTuringMulticinta`

```
class MaquinaTuringMulticinta:
    def __init__(self, transiciones, q_inicial, q_aceptar, q_rechazar,
        num_cintas):
        self.transiciones = transiciones
        self.q_inicial = q_inicial
        self.q_aceptar = q_aceptar
        self.q_rechazar = q_rechazar
        self.num_cintas = num_cintas

    def ejecutar(self, entrada, max_pasos=10000):
        # Cinta 1: entrada; cintas restantes: vacías (solo blancos)
        cintas = [list(entrada) if entrada else ['_']]
        for _ in range(self.num_cintas - 1):
            cintas.append(['_'])
        posiciones = [0] * self.num_cintas
        estado = self.q_inicial
        pasos = 0
        while pasos < max_pasos:
            if estado == self.q_aceptar:
                return True, self._cintas_str(cintas), pasos
            if estado == self.q_rechazar:
                return False, self._cintas_str(cintas), pasos
            # Leer el símbolo actual de cada cinta
            simbolos = tuple(self._leer(cintas[i], posiciones[i])
                             for i in range(self.num_cintas))
            clave = (estado, simbolos)
            # Sin transición definida: rechazar
            if clave not in self.transiciones:
                return False, self._cintas_str(cintas), pasos
            nuevo_estado, escrituras, direcciones = self.transiciones[clave
                ]
            for i in range(self.num_cintas):
                self._escribir(cintas[i], posiciones[i], escrituras[i])
                posiciones[i] = self._mover(posiciones[i], direcciones[i],
                    cintas[i])
            estado = nuevo_estado
            pasos += 1
        return None, self._cintas_str(cintas), pasos
```

La segunda parte desarrolla el ciclo principal de ejecución junto con las operaciones de escritura y desplazamiento de cada cabeza:

Código 4.6: Clase MaquinaTuringMulticinta

```
37      def _leer(self, cinta, pos):
38          return cinta[pos] if pos < len(cinta) else '_'
39
40      def _escribir(self, cinta, pos, simbolo):
41          while pos >= len(cinta):
42              cinta.append('_')
43          cinta[pos] = simbolo
44
45      def _mover(self, pos, direccion, cinta):
46          if direccion == 'R':
47              return pos + 1
48          elif direccion == 'L' and pos > 0:
49              return pos - 1
50          return pos  # 'S' (Stay) o L en posición 0
51
52      def _cintas_str(self, cintas):
53          return [''.join(c).rstrip('_') for c in cintas]
```

b. Comparación de MT de 1 cinta y 2 cintas

El programa ejecuta ambas MT sobre la misma cadena y compara tanto el resultado como la cantidad de pasos realizados. La función `main()` se encarga de leer la cadena y llamar a la rutina de comparación.

Código 4.7: Función principal y comparación

```
3
4  def main():
5      cadena = leer("Ingrese la cadena (a's, b's y c's): ")
6      mt1 = crear_mt_una_cinta()
7      r1, _, p1 = mt1.ejecutar(cadena)
8      mt2 = crear_mt_dos_cintas()
9      r2, _, p2 = mt2.ejecutar(cadena)
10     imprimir_comparacion(cadena, r1, p1, r2, p2)
11
12 def leer(texto):
13     return input(texto)
14
15 def imprimir_comparacion(cadena, r1, p1, r2, p2):
16     v1 = "ACEPTADA" if r1 else "RECHAZADA"
17     v2 = "ACEPTADA" if r2 else "RECHAZADA"
18     print(f"Cadena: '{cadena}'")
19     print(f"  1 cinta:  {v1} ({p1} pasos)")
20     print(f"  2 cintas: {v2} ({p2} pasos)")
21     print(f"  Mismo resultado: {r1 == r2}")
```

La MT de 2 cintas se define mediante 11 transiciones organizadas en cuatro etapas: colocación del centinela, conteo de símbolos a, verificación de símbolos b y verificación de símbolos c.

Código 4.8: Creación de la MT de 2 cintas

```
23 def crear_mt_dos_cintas():
24     t = {
25         ('qi', ('a', '_')): ('q0', ('a', '$'), ('S', 'R')),
26         ('q0', ('a', '_')): ('q0', ('a', 'X'), ('R', 'R')),
27         ('q0', ('b', '_')): ('q1', ('b', '_'), ('S', 'L')),
28         ('q1', ('b', 'X')): ('q1', ('b', 'X'), ('R', 'L')),
29         ('q1', ('c', '$')): ('q2', ('c', '$'), ('S', 'R')),
30         ('q1', ('b', '$')): ('qr', ('b', '$'), ('S', 'S')),
31         ('q1', ('c', 'X')): ('qr', ('c', 'X'), ('S', 'S')),
32         ('q2', ('c', 'X')): ('q2', ('c', 'X'), ('R', 'R')),
33         ('q2', ('_', '_')): ('qa', ('_', '_'), ('S', 'S')),
34         ('q2', ('c', '_')): ('qr', ('c', '_'), ('S', 'S')),
35         ('q2', ('_', 'X')): ('qr', ('_', 'X'), ('S', 'S')),
36     }
37     return MaquinaTuringMulticinta(t, 'qi', 'qa', 'qr', 2)
```

La MT de 1 cinta utiliza las 16 transiciones introducidas en la sección anterior:

Código 4.9: Creación de la MT de 1 cinta

```
39 def crear_mt_una_cinta():
40     transiciones = {
41         ('q0', 'a'): ('q1', 'X', 'R'),
42         ('q0', 'Y'): ('q4', 'Y', 'R'),
43         ('q1', 'a'): ('q1', 'a', 'R'),
44         ('q1', 'Y'): ('q1', 'Y', 'R'),
45         ('q1', 'b'): ('q2', 'Y', 'R'),
46         ('q2', 'b'): ('q2', 'b', 'R'),
47         ('q2', 'Z'): ('q2', 'Z', 'R'),
48         ('q2', 'c'): ('q3', 'Z', 'L'),
49         ('q3', 'a'): ('q3', 'a', 'L'),
50         ('q3', 'b'): ('q3', 'b', 'L'),
51         ('q3', 'Y'): ('q3', 'Y', 'L'),
52         ('q3', 'Z'): ('q3', 'Z', 'L'),
53         ('q3', 'X'): ('q0', 'X', 'R'),
54         ('q4', 'Y'): ('q4', 'Y', 'R'),
55         ('q4', 'Z'): ('q4', 'Z', 'R'),
56         ('q4', '_'): ('qa', '_', 'R'),
57     }
58     return MaquinaTuring(transiciones, 'q0', 'qa', 'qr')
```

4.3. Los límites de la computación

La empresa **AutoCheck** desarrolla herramientas de análisis estático para equipos de desarrollo. Sus clientes solicitan una funcionalidad ambiciosa: un módulo que determine automáticamente si cualquier fragmento de código enviado terminará su ejecución o entrará en un ciclo infinito. El equipo necesita comprender si esta funcionalidad es teóricamente posible antes de invertir recursos en su desarrollo.

4.3.1 Análisis / Abstracción

a. ¿Cuál es el problema a resolver?

Dado un programa arbitrario y una entrada, determinar si el programa terminará su ejecución para esa entrada, o si entrará en un ciclo infinito. Más ampliamente: ¿cuáles son los límites fundamentales de lo que un algoritmo puede decidir?

b. ¿Cuál es la información necesaria?

A diferencia de los problemas anteriores, aquí los datos no son una cadena sino una pieza de software junto con el contexto en el que se ejecutaría:

Información necesaria	Descripción
Programa a analizar	El código que se quiere verificar
Entrada del programa	Los datos con los que se ejecutaría
Pregunta a responder	Si el programa terminará o no para esa entrada

4.3.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **ENTRADA:** Un programa y una entrada para ese programa.
- **PROCESAMIENTO:** Determinar si el programa terminará su ejecución para esa entrada.

- **SALIDA:** Indicar si el programa termina o no, o concluir que el problema no tiene solución general.

b. ¿Qué conocimiento adicional se requiere?

Lenguajes decidibles y reconocibles

Las MT tienen una característica peculiar que los modelos anteriores no tenían: pueden entrar en un **ciclo infinito** y nunca detenerse. Esto genera una distinción importante.

Analogía: el verificador y el buscador

- **Un verificador:** "¿Está el número 42 en esta lista ordenada?" El programa recorre la lista y siempre termina con una respuesta: "sí" o "no". Este comportamiento corresponde a un problema **decidible**.
- **Un buscador:** "¿Existe una entrada que haga fallar este programa?" Si el sistema encuentra un caso que produce el error, puede responder: "sí, aquí está". Sin embargo, si no encuentra ninguno, no necesariamente puede concluir que no existe; podría seguir buscando indefinidamente. Este comportamiento se relaciona con la **reconocibilidad**.

Definiciones formales

Un lenguaje L es **Turing-reconocible** (o **recursivamente enumerable**) si existe una MT que:

- acepta toda cadena $w \in L$,
- y para toda cadena $w \notin L$, puede rechazarla o continuar ejecutándose indefinidamente.

Un lenguaje L es **Turing-decidible** (o recursivo) [Sipser, 2012] si existe una MT que:

- Acepta toda cadena $w \in L$ (dice "sí" correctamente).
- Rechaza toda cadena $w \notin L$ (dice "no" correctamente).
- **Siempre termina** (nunca se queda pensando para siempre).

La distinción entre decidible y reconocible tiene consecuencias prácticas. Para un problema **decidible** se puede escribir un programa que siempre da una respuesta. Para un problema solo **reconocible** se puede escribir un programa que responde "sí" cuando la respuesta es positiva, pero ante una respuesta negativa el programa podría no terminar nunca; en la práctica, no se sabe si está tardando mucho o si no terminará jamás.

Ejemplo 3: Un lenguaje reconocible pero no decidible

Consideremos el problema de determinar si un programa termina su ejecución o entra en un ciclo infinito. Analicemos los siguientes ejemplos:

```
# Programa A: termina inmediatamente
def programa_A():
    return 42

# Programa B: nunca termina
def programa_B():
    while True:
        pass

# Programa C: ¿termina para cualquier valor?
def programa_C(n):
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
    return n
```

El programa A termina inmediatamente, mientras que el programa B entra en un ciclo infinito. En cambio, el comportamiento del programa C está relacionado con la conjetura de Collatz, un problema que aún

permanece abierto: no se ha demostrado si el proceso termina para todos los valores de n.

El lenguaje de los programas que terminan:

Sea $L_{termina} = \{\text{todos los programas que eventualmente terminan}\}$.

¿Es reconocible? Sí. Podemos escribir un "verificador" que recibe un programa, lo ejecuta paso a paso, y si termina, dice "sí". Para el programa A, nuestro verificador rápidamente dirá "sí".

¿Es decidible? No. El problema aparece con programas como B y C:

- Para el programa B, nuestro verificador se quedará ejecutándolo *para siempre.*
- Para el programa C, ¿cuánto tiempo debemos esperar? No hay forma de saber cuándo "rendirse".

Jerarquía de lenguajes

La siguiente tabla resume las clases de lenguajes y sus reconocedores:

Clase	**Reconocedor**	**Ejemplo**
Regular	AFD, AFN	a^*b^*
Libre de contexto	PDA	a^nb^n
Decidible	MT (siempre termina)	$a^nb^nc^n$
Reconocible (no decidible)	MT (puede no terminar)	$L_{termina}$

Estas clases forman una jerarquía:

$$\text{Regular} \subset \text{Libre de contexto} \subset \text{Decidible} \subset \text{Reconocible}.$$

La Tesis de Church-Turing

En la década de 1930, los matemáticos intentaban responder una pregunta fundamental: *¿qué significa que un problema pueda resolverse mediante un procedimiento mecánico?*

En 1936, distintos investigadores desarrollaron modelos formales para describir la computación:

- **Alan Turing** introdujo las Máquinas de Turing.
- **Alonzo Church** desarrolló el cálculo lambda.
- **Kurt Gödel**, junto con Jacques Herbrand, definió las funciones recursivas, formalizadas posteriormente por Kleene [Kleene, 1936].
- **Emil Post** propuso un modelo combinatorio finito equivalente [Post, 1936].

Estos formalismos resultaron ser **matemáticamente equivalentes**. La equivalencia entre las Máquinas de Turing y el cálculo lambda fue demostrada por el propio Turing [Turing, 1937].

Tesis de Church-Turing [Turing, 1936, Church, 1936]**:** Todo proceso que intuitivamente consideramos "computable" o "algorítmico" puede ser realizado por una Máquina de Turing.

La palabra "tesis" se utiliza de manera intencional, ya que este enunciado no corresponde a un teorema demostrable en sentido estricto. La razón es que involucra la noción de "computable de manera intuitiva", la cual no posee una definición matemática formal. En términos generales, la tesis afirma que todo problema que pueda resolverse mediante un procedimiento efectivo puede ser computado por una máquina de Turing.

Existen varias razones que apoyan esta idea. A lo largo del tiempo se han propuesto distintos modelos de computación y, aunque fueron desarrollados de manera independiente, todos resultan equivalentes en poder computacional. Además, cualquier algoritmo conocido puede implementarse mediante una máquina de Turing. Incluso cuando el modelo se modifica o se introducen variantes más complejas, la capacidad de cómputo no cambia realmente. Hasta hoy tampoco se ha encontrado un modelo razonable de computación capaz de resolver problemas no computables por una máquina de Turing.

La tesis tiene consecuencias fundamentales en informática. Lenguajes como Python, Java, C o Haskell poseen el mismo poder computacional que una máquina de Turing. Por ello, las diferencias entre estos lenguajes no están en lo que pueden calcular, sino en aspectos como el rendimiento, la facilidad de programación o el paradigma utilizado.

De manera similar, si se demuestra que una máquina de Turing no puede resolver un problema, entonces ningún programa podrá resolverlo, independientemente del lenguaje empleado. La computabilidad tampoco

depende del hardware específico: un computador moderno o incluso uno cuántico puede ejecutar ciertos cálculos de forma más eficiente, pero no resolver problemas que estén fuera de los límites de la computación Turing.

Indecidibilidad y el problema de la parada

¿Pueden las MT resolver **todo** problema bien definido? La respuesta es **no**.

El ejemplo más famoso es el **Problema de la Parada** (*Halting Problem*) [Turing, 1936]:

> **Problema de la parada:** Dado un programa y una entrada, ¿terminará el programa cuando se ejecute con esa entrada?

¿Por qué es indecidible?

No existe ningún programa que resuelva este problema. La demostración usa un argumento de *autorreferencia*, similar a paradojas clásicas.

Consideremos la oración "Esta oración es falsa".

- Si es **verdadera**, entonces lo que dice es cierto, así que es falsa. ¡Contradicción!
- Si es **falsa**, entonces lo que dice es incorrecto, así que es verdadera. ¡Contradicción!

La demostración de Turing se basa en construir un programa hipotético que, al analizarse a sí mismo, produce una contradicción: si termina, no debería terminar; y si no termina, debería hacerlo.

Esta contradicción prueba que el programa H que resuelve el problema de la parada no puede existir.

Las consecuencias prácticas son directas: no puede existir un "depurador perfecto" que siempre detecte si un programa terminará. Las herramientas de análisis pueden detectar *algunos* ciclos infinitos obvios, pero siempre habrá programas para los cuales no podrán decidir.

Otros problemas indecidibles:

- ¿Acepta este programa alguna entrada?
- ¿Rechaza este programa todas las entradas posibles?

- ¿Hacen dos programas lo mismo para todas las entradas?
- ¿Cumple este código su especificación?

En general, casi cualquier pregunta interesante sobre el *comportamiento* de un programa es indecidible. Este hecho se formaliza mediante el **Teorema de Rice** [Rice, 1953]: toda propiedad no trivial del lenguaje reconocido por una Máquina de Turing es indecidible.

c. ¿Qué se puede reutilizar de soluciones pasadas?

La jerarquía de lenguajes presentada en este capítulo continúa la progresión estudiada anteriormente:

$$\text{Regular} \subset \text{LLC} \subset \text{Decidible} \subset \text{Reconocible}.$$

Cada nivel incluye a los anteriores y permite describir problemas que los modelos previos no podían resolver.

d. ¿Cuál es el diseño de la solución?

El módulo solicitado por los clientes de AutoCheck, un verificador universal capaz de determinar si *cualquier* fragmento de código terminará, **no puede existir**. El problema de la parada demuestra que no existe un algoritmo capaz de responder esta pregunta para todos los programas posibles. Esta no es una limitación tecnológica ni un problema de eficiencia, sino una restricción fundamental de la computación.

Solución práctica: verificación acotada

Aunque el problema general es indecidible, una versión restringida **sí puede resolverse**: dado un programa y un límite máximo de pasos, es posible determinar si el programa termina dentro de ese límite ejecutándolo paso a paso.

Estrategia del verificador acotado:

1. Recibir el programa P, su entrada x y un límite de pasos N.
2. Ejecutar $P(x)$ paso a paso, manteniendo un contador de ejecuciones.

3. Si P finaliza antes de alcanzar N pasos, reportar "**TERMINA**" junto con el resultado.
4. Si P alcanza el límite sin finalizar, reportar "**INDETERMINADO**".

¿Por qué funciona? El verificador siempre termina después de ejecutar, como máximo, N pasos, por lo que el procedimiento es decidible. Sin embargo, si el programa supera ese límite, no es posible saber si finalizará más adelante o si continuará ejecutándose indefinidamente. Esta es precisamente la dificultad asociada al problema de la parada.

Para AutoCheck se propone implementar este verificador acotado como una herramienta configurable de análisis. Cada cliente puede definir el límite de pasos según el tipo de programa que desee evaluar.

El sistema puede producir tres resultados:

- **TERMINA**: el programa finaliza dentro del límite establecido.
- **INDETERMINADO**: el programa supera el límite sin finalizar, por lo que no es posible concluir si terminará o no.
- **ERROR**: ocurre una falla durante la ejecución del programa.

4.3.3 Ejemplos de pruebas

Se prueba el verificador acotado con tres programas que ilustran los diferentes escenarios. El límite de pasos se establece en 1000.

a. Ejemplo 1

Programa: `factorial(5)` **Límite**: 1000 pasos

Resultado: TERMINA en 5 pasos (resultado: 120).

El factorial de 5 realiza exactamente 5 multiplicaciones. El verificador detecta la terminación rápidamente y reporta el resultado con confianza.

b. Ejemplo 2

Programa: `collatz(27)` **Límite**: 1000 pasos

Resultado: TERMINA en 111 pasos.

La secuencia de Collatz para $n = 27$ es notoriamente larga: alcanza un valor máximo de 9232 antes de descender a 1. Aunque nadie ha demostrado que la conjetura de Collatz sea verdadera para todo n, para esta entrada particular el verificador confirma la terminación dentro del límite.

c. Ejemplo 3

Programa: `collatz(9780657630)` **Límite**: 1000 pasos

Resultado: INDETERMINADO (superó los 1000 pasos).

Para este valor, la secuencia de Collatz necesita más de 1000 pasos para llegar a 1. Por ello, el verificador no puede concluir si el programa terminará o no dentro del límite establecido.

Este ejemplo muestra una de las dificultades fundamentales del problema de la parada: un programa puede terminar después de una gran cantidad de pasos, pero el verificador no puede saberlo sin continuar la ejecución. Por esta razón, ningún límite finito permite identificar correctamente todos los programas que terminan.

4.3.4 Codificación

a. Verificador acotado de terminación

Si bien el problema de la parada es indecidible en general, una versión acotada sí es decidible: dado un programa y un límite de pasos, podemos verificar si el programa termina dentro de ese límite simplemente ejecutándolo.

`main` ofrece un menú al usuario para seleccionar un programa, ingresar un valor y definir el límite de pasos.

Código 4.10: Función principal

```
def main():
    programa = leer("Programa (factorial, collatz): ")
    n = int(leer("Ingrese el valor de entrada: "))
    max_pasos = int(leer("Limite de pasos: "))
    if programa == "factorial":
        resultado, pasos = ejecutar_acotado(factorial, n, max_pasos)
    elif programa == "collatz":
        resultado, pasos = ejecutar_acotado(collatz, n, max_pasos)
    else:
        print("Programa no reconocido")
        return
    imprimir_resultado(programa, n, resultado, pasos)
```

Las funciones de entrada y salida gestionan la interacción con la consola y dan formato a los tres posibles resultados: `TERMINA`, `INDETERMINADO` o `ERROR`.

Código 4.11: Funciones de entrada y salida

```
def leer(texto):
    return input(texto)

def imprimir_resultado(programa, n, resultado, pasos):
    if resultado is not None:
        print(f"{programa}({n}): termino en {pasos} pasos (resultado: {
            resultado})")
    else:
        print(f"{programa}({n}): excedio {pasos} pasos (no sabemos si
            termina)")
```

b. Ejecución acotada y programas de ejemplo

La función `ejecutar_acotado` envuelve la ejecución de un programa con un límite de pasos. Si el programa termina antes del límite, reporta el resultado; si excede el límite, no puede concluir si terminará eventualmente:

Código 4.12: Ejecución acotada y excepción de límite

```
def ejecutar_acotado(programa, entrada, max_pasos):
    try:
        resultado, pasos = programa(entrada, max_pasos)
        return resultado, pasos
    except PasosExcedidos as e:
        return None, e.pasos

class PasosExcedidos(Exception):
    def __init__(self, pasos):
        self.pasos = pasos
```

Las funciones `factorial` y `collatz` implementan dos programas de ejemplo. El factorial siempre termina, mientras que la conjetura de Collatz ilustra un caso donde no sabemos si termina para todo valor de entrada:

Código 4.13: Programas de ejemplo con contador de pasos

```
def factorial(n, max_pasos):
    resultado = 1
    pasos = 0
    for i in range(1, n + 1):
        resultado *= i
        pasos += 1
        if pasos > max_pasos:
            raise PasosExcedidos(pasos)
    return resultado, pasos

def collatz(n, max_pasos):
    pasos = 0
    original = n
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        pasos += 1
        if pasos > max_pasos:
            raise PasosExcedidos(pasos)
    return original, pasos
```

4.4. Computación basada en funciones

La empresa **FuncTech** explora nuevos paradigmas de programación para sus productos de procesamiento de datos. El equipo ha notado que muchos errores provienen del manejo de estado mutable y quiere adoptar un enfoque puramente funcional. Surge la pregunta: ¿es posible expresar toda la computación usando únicamente funciones, sin variables mutables, sin memoria explícita y sin pasos secuenciales? ¿Tendría este enfoque el mismo poder que los modelos basados en máquinas?

4.4.1 Análisis / Abstracción

a. ¿Cuál es el problema a resolver?

Entender cómo representar procesos computacionales utilizando únicamente funciones. En particular, cómo expresar datos (números, booleanos) y operaciones (suma, multiplicación) sin recurrir a tipos de datos nativos, sino solamente mediante definición y aplicación de funciones.

b. ¿Cuál es la información necesaria?

Antes de evaluar el enfoque puramente funcional conviene delimitar qué se puede usar, qué hay que representar y qué pregunta se quiere responder:

Información necesaria	**Descripción**
Operaciones permitidas	Únicamente definir y aplicar funciones
Datos a representar	Números, booleanos, operaciones aritméticas
Pregunta a responder	Si este enfoque tiene el mismo poder que los modelos con máquinas

4.4.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **ENTRADA:** Un conjunto de operaciones a realizar (aritmética, lógica).

- **PROCESAMIENTO:** Expresar datos y operaciones usando únicamente funciones, sin tipos de datos nativos ni estado mutable.
- **SALIDA:** Demostrar que el enfoque funcional puede representar la misma computación que los modelos basados en máquinas.

b. ¿Qué conocimiento adicional se requiere?

Los tres elementos del cálculo lambda

Las expresiones (llamadas **λ-términos**) se construyen con solo tres elementos [Barendregt, 1984, Church, 1936]:

- **Variables:** $x, y, z, \ldots$ (nombres para valores)
- **Abstracción:** $\lambda x.E$ (definir una función que recibe x y retorna E, donde E es el *cuerpo* de la función)
- **Aplicación:** $E_1\, E_2$ (aplicar la función E_1 al argumento E_2)

Ejemplos:

- $\lambda x.x$ — La función identidad. Recibe x y retorna x sin cambios.
- $\lambda x.\lambda y.x$ — Recibe dos argumentos, retorna el primero.
- $\lambda x.\lambda y.y$ — Recibe dos argumentos, retorna el segundo.

Nota sobre funciones con varios argumentos: en cálculo lambda, toda función recibe un único argumento. Por ello, una función de dos argumentos se representa como una función que recibe el primero y retorna otra encargada de recibir el segundo. Este proceso se conoce como **currificación**.

Por ejemplo, $\lambda x.\lambda y.x$ funciona así:

1. Recibe x y retorna la función $\lambda y.x$
2. Esta función recibe y y retorna x (ignorando y)

La β-reducción

La única operación del cálculo lambda es la **β-reducción** [Barendregt, 1984]: cuando aplicamos una función a un argumento, sustituimos el parámetro por el argumento en el cuerpo de la función.

Regla de β-reducción:

$$(\lambda x.\text{cuerpo})\,\text{argumento} \quad \to \quad \text{cuerpo}[x := \text{argumento}]$$

Donde $[x := \text{argumento}]$ significa "reemplazar cada x por el argumento".

Ejemplo 4: Función identidad

$$(\lambda x.x)\,5 \quad \to \quad 5$$

Paso a paso: La función es $\lambda x.x$ (el cuerpo es simplemente x). El argumento es 5. Sustituimos x por 5 en el cuerpo: $x[x := 5] = 5$.

Ejemplo 5: Función aplicada a otra función

$$(\lambda x.x)\,(\lambda y.y) \quad \to \quad \lambda y.y$$

Paso a paso: El argumento ahora es otra función $(\lambda y.y)$. Sustituimos x por $(\lambda y.y)$ en el cuerpo x, obteniendo $(\lambda y.y)$.

Ejemplo 6: Aplicaciones anidadas (dos argumentos)

Evaluemos $(\lambda x.\lambda y.x)\,a\,b$ paso a paso:

---

1. **Expresión inicial:** $(\lambda x.\lambda y.x)\, a\, b$
2. **Primera reducción:** Aplicamos $(\lambda x.\lambda y.x)$ al argumento a.
   - Función: $\lambda x.\lambda y.x$ (cuerpo: $\lambda y.x$)
   - Argumento: a
   - Sustituimos x por a: $(\lambda y.x)[x := a] = \lambda y.a$

   Resultado: $(\lambda y.a)\, b$
3. **Segunda reducción:** Aplicamos $(\lambda y.a)$ al argumento b.
   - Función: $\lambda y.a$ (cuerpo: a)
   - Argumento: b
   - Sustituimos y por b: $a[y := b] = a$ (la y no aparece en a, así que no cambia nada)

   Resultado: a

Resumen:

$$(\lambda x.\lambda y.x)\, a\, b \quad \to \quad (\lambda y.a)\, b \quad \to \quad a$$

Esta función "selecciona el primer argumento": recibe a y b, retorna a.

Comparación con Python

La sintaxis del cálculo lambda inspiró directamente las funciones anónimas de muchos lenguajes:

Cálculo Lambda	**Python**
$\lambda x.x$	`lambda x: x`
$\lambda x.\lambda y.x$	`lambda x: lambda y: x`
$(\lambda x.x)\, 5$	`(lambda x: x)(5)`

c. ¿Qué se puede reutilizar de soluciones pasadas?

La organización general de los programas presentada en capítulos anteriores, como el uso de funciones auxiliares, lectura de entrada y generación de resultados, puede reutilizarse para implementar y experimentar con conceptos de cálculo lambda en Python.

d. ¿Cuál es el diseño de la solución?

La respuesta para FuncTech es afirmativa: toda computación puede representarse utilizando únicamente funciones. Además, este enfoque posee el mismo poder computacional que los modelos basados en máquinas.

Para demostrar esta idea, la solución se desarrolla en tres etapas.

Paso 1: Representar datos como funciones (Numerales de Church)

Aunque el cálculo lambda solo tiene variables, funciones y aplicación, es posible representar números, booleanos, listas y cualquier estructura de datos usando *únicamente funciones*.

El número n se representa como una función que aplica otra función n veces; en otras palabras, el número indica cuántas veces debe hacerse algo.

Esta codificación fue introducida por Church [Church, 1941].

Numerales de Church:

- $\mathbf{0} = \lambda f.\lambda x.x$ (aplica f cero veces, retorna x sin cambios)
- $\mathbf{1} = \lambda f.\lambda x.f\,x$ (aplica f una vez)
- $\mathbf{2} = \lambda f.\lambda x.f\,(f\,x)$ (aplica f dos veces)
- $\mathbf{3} = \lambda f.\lambda x.f\,(f\,(f\,x))$ (aplica f tres veces)

Intuición: Imaginemos que f es "dar un paso" y x es "el punto de partida". Entonces:

- **0** significa "no dar ningún paso" (permanecer en x)
- **2** significa "dar dos pasos" (aplicar f dos veces desde x)

Verificando que 2 representa el número 2:

Recordemos que $\mathbf{2} = \lambda f.\lambda x.f\,(f\,x)$: recibe una función f y un valor x, y aplica f dos veces. Dado que f y x pueden ser *cualquier* función y valor, podemos elegir un caso concreto para verificar. Si elegimos $f = \lambda n.n + 1$ (la función que suma 1 a su argumento, en Python `lambda x: x + 1`) y $x = 0$, aplicamos dos β-reducciones:

$$\begin{aligned}
\mathbf{2}\,(\lambda n.n+1)\,0 &= (\lambda f.\lambda x.f\,(f\,x))\,(\lambda n.n+1)\,0 \\
&\to_\beta (\lambda x.(\lambda n.n+1)\,((\lambda n.n+1)\,x))\,0 && [\text{sustituir } f] \\
&\to_\beta (\lambda n.n+1)\,((\lambda n.n+1)\,0) && [\text{sustituir } x \text{ por } 0] \\
&\to_\beta (\lambda n.n+1)\,1 && [\text{reducir: } 0+1=1] \\
&\to_\beta 2 && [\text{reducir: } 1+1=2]
\end{aligned}$$

Cada $\to_\beta$ es una β-reducción: se sustituye el parámetro por el argumento en el cuerpo de la función. El resultado final es 2, confirmando que el numeral de Church **2** representa correctamente el número 2.

Paso 2: Definir operaciones como funciones puras

La función sucesor (SUCC):

Podemos definir operaciones aritméticas. El sucesor de n es $n+1$:

$$\text{SUCC} = \lambda n.\lambda f.\lambda x.f\,(n\,f\,x)$$

Intuición: Dado un numeral n que aplica f n veces, SUCC retorna un numeral que aplica f una vez más.

Calculando SUCC(1):

$$\begin{aligned}
\text{SUCC}\,\mathbf{1} &= (\lambda n.\lambda f.\lambda x.f\,(n\,f\,x))\,\mathbf{1} \\
&\to \lambda f.\lambda x.f\,(\mathbf{1}\,f\,x) \\
&= \lambda f.\lambda x.f\,((\lambda f.\lambda x.f\,x)\,f\,x) \\
&\to \lambda f.\lambda x.f\,(f\,x) \\
&= \mathbf{2}
\end{aligned}$$

El sucesor de **1** es **2**, como esperábamos.

Suma y multiplicación:

$$\text{SUMA} = \lambda m.\lambda n.\lambda f.\lambda x.m\,f\,(n\,f\,x)$$

$$\text{MULT} = \lambda m.\lambda n.\lambda f.m\,(n\,f)$$

Intuición de SUMA: Primero aplicar f n veces (obteniendo $n\,f\,x$), luego aplicar f m veces más.

Intuición de MULT: Aplicar "aplicar f n veces" m veces, lo que da $m \times n$ aplicaciones de f.

Paso 3: Equivalencia con las Máquinas de Turing

El cálculo lambda y las máquinas de Turing abordan la computación desde perspectivas distintas. La máquina de Turing modela el cálculo mediante estados y transiciones sobre una cinta, mientras que el cálculo lambda se fundamenta en la definición y aplicación de funciones.

Aunque ambos enfoques son muy diferentes, tienen el mismo poder computacional, idea central de la tesis de Church-Turing [Turing, 1937].

Aspecto	**Máquina de Turing**	**Cálculo Lambda**
Paradigma	Imperativo	Funcional
Idea principal	Estados y transiciones	Funciones y aplicación
Memoria	Cinta explícita	Sustitución de expresiones
Forma de cómputo	Ejecución paso a paso	Reducción de expresiones

El cálculo lambda se considera la base teórica de muchos lenguajes funcionales [Roy, 2007]. Conceptos como funciones anónimas, funciones de orden superior y currificación aparecen hoy en lenguajes modernos como Python, JavaScript y Haskell.

Para FuncTech, esto significa que adoptar un enfoque funcional no reduce las capacidades de los sistemas desarrollados por la empresa. Los mismos problemas que pueden resolverse con modelos imperativos también pueden describirse mediante funciones puras.

4.4.3 Ejemplos de pruebas

Se verifica que los numerales de Church representan correctamente los números y que las operaciones aritméticas producen resultados correctos mediante β-reducciones.

a. Ejemplo 1

Verificación de numerales: Aplicando cada numeral a $f = \lambda n.n + 1$ y $x = 0$:

$$\mathbf{0}\,(\lambda n.n+1)\,0 = (\lambda f.\lambda x.x)\,(\lambda n.n+1)\,0 \to_\beta 0$$
$$\mathbf{1}\,(\lambda n.n+1)\,0 = (\lambda f.\lambda x.f\,x)\,(\lambda n.n+1)\,0 \to_\beta (0+1) = 1$$
$$\mathbf{2}\,(\lambda n.n+1)\,0 = (\lambda f.\lambda x.f\,(f\,x))\,(\lambda n.n+1)\,0 \to_\beta (0+1)+1 = 2$$

Cada numeral aplica "sumar 1" el número esperado de veces. **0** no aplica la función (retorna 0 directamente), **1** la aplica una vez ($0 \to 1$), **2** la aplica dos veces ($0 \to 1 \to 2$). Los datos se representan correctamente como funciones.

b. Ejemplo 2

Verificación de SUCC: SUCC(**2**) debe dar **3**.

$$\begin{aligned}
\text{SUCC}\,\mathbf{2} &= (\lambda n.\lambda f.\lambda x.f\,(n\,f\,x))\,\mathbf{2} \\
&\to_\beta \lambda f.\lambda x.f\,(\mathbf{2}\,f\,x) && [\text{sustituir } n \text{ por } \mathbf{2}] \\
&= \lambda f.\lambda x.f\,(f\,(f\,x)) && [\mathbf{2}\,f\,x = f(f(x))] \\
&= \mathbf{3}
\end{aligned}$$

SUCC toma el numeral **2** (que aplica f dos veces) y retorna un numeral que aplica f tres veces.

c. Ejemplo 3

Verificación de SUMA: SUMA(**1**)(**2**) debe dar **3**.

$$\begin{aligned}
\text{SUMA}\,\mathbf{1}\,\mathbf{2} &= (\lambda m.\lambda n.\lambda f.\lambda x.m\,f\,(n\,f\,x))\,\mathbf{1}\,\mathbf{2} \\
&\to_\beta \lambda f.\lambda x.\mathbf{1}\,f\,(\mathbf{2}\,f\,x) && [\text{sustituir } m \text{ y } n] \\
&= \lambda f.\lambda x.\mathbf{1}\,f\,(f\,(f\,x)) && [\mathbf{2}\,f\,x = f(f(x))] \\
&= \lambda f.\lambda x.f\,(f\,(f\,x)) && [\mathbf{1}\,f\,y = f(y)] \\
&= \mathbf{3}
\end{aligned}$$

2 aplica f dos veces a x, y luego **1** aplica f una vez más: $1 + 2 = 3$ aplicaciones en total.

d. Ejemplo 4

Verificación de MULT: MULT(**2**)(**3**) debe dar **6**.

$$\begin{aligned} \text{MULT}\,\mathbf{2}\,\mathbf{3} &= (\lambda m.\lambda n.\lambda f.m\,(n\,f))\,\mathbf{2}\,\mathbf{3} \\ &\to_\beta \lambda f.\mathbf{2}\,(\mathbf{3}\,f) && [\text{sustituir } m \text{ y } n] \end{aligned}$$

Sea $g = \mathbf{3}\,f = \lambda x.f(f(f(x)))$, es decir, g aplica f tres veces. Entonces:

$$\begin{aligned} \lambda f.\mathbf{2}\,g &= \lambda f.\lambda x.g\,(g\,x) && [\mathbf{2} \text{ aplica } g \text{ dos veces}] \\ &= \lambda f.\lambda x.g\,(f(f(f(x)))) && [g \text{ interno: 3 aplicaciones de } f] \\ &= \lambda f.\lambda x.f(f(f(f(f(f(x)))))) && [g \text{ externo: 3 aplicaciones más}] \\ &= \mathbf{6} \end{aligned}$$

Aplicar "f tres veces" (g) dos veces (**2**) produce $2 \times 3 = 6$ aplicaciones de f. La aritmética completa funciona sin variables mutables ni tipos de datos nativos, confirmando la viabilidad del enfoque funcional para FuncTech.

4.4.4 Codificación

a. Numerales de Church

El primer programa construye el numeral de Church correspondiente al número ingresado por el usuario.

Código 4.14: Función principal y entrada/salida

```
def main():
    n = int(leer("Ingrese un numero (0-9): "))
    numeral = construir_numeral(n)
    resultado = church_to_int(numeral)
    imprimir_resultado(n, resultado)

def leer(texto):
    return input(texto)

def imprimir_resultado(n, valor):
    print(f"El numeral de Church para {n} es: {valor}")
```

La función `church_to_int()` convierte un numeral de Church a entero aplicando la operación "sumar 1" n veces sobre el valor inicial 0. Por su parte, `construir_numeral()` genera cualquier numeral aplicando repetidamente SUCC a partir de CERO.

Código 4.15: Conversión, construcción y numerales base

```
13 def church_to_int(n):
14     return n(lambda x: x + 1)(0)
15
16 def construir_numeral(n):
17     numeral = CERO
18     for _ in range(n):
19         numeral = SUCC(numeral)
20     return numeral
21
22 CERO = lambda f: lambda x: x
23 SUCC = lambda n: lambda f: lambda x: f(n(f)(x))
```

b. Operaciones aritméticas con numerales de Church

El segundo programa incorpora operaciones aritméticas sobre numerales de Church. El usuario puede seleccionar una operación (SUCC, SUMA o MULT) e ingresar los operandos correspondientes para obtener el resultado evaluado.

La función `main()` recibe la operación seleccionada y los operandos, y ejecuta el cálculo correspondiente.

Código 4.16: Función principal

```
def main():
    operacion = leer("Operacion (SUCC, SUMA, MULT): ")
    if operacion == "SUCC":
        n = int(leer("Ingrese un numero: "))
        numeral = construir_numeral(n)
        resultado = church_to_int(SUCC(numeral))
        imprimir_resultado(f"SUCC({n})", resultado)
    elif operacion == "SUMA":
        a = int(leer("Primer numero: "))
        b = int(leer("Segundo numero: "))
        resultado = church_to_int(SUMA(construir_numeral(a))(
            construir_numeral(b)))
        imprimir_resultado(f"SUMA({a})({b})", resultado)
    elif operacion == "MULT":
        a = int(leer("Primer numero: "))
        b = int(leer("Segundo numero: "))
        resultado = church_to_int(MULT(construir_numeral(a))(
            construir_numeral(b)))
        imprimir_resultado(f"MULT({a})({b})", resultado)
    else:
        print("Operacion no reconocida")
```

La entrada de datos y la visualización del resultado se realizan mediante funciones auxiliares encargadas de interactuar con la consola.

Código 4.17: Funciones de entrada y salida

```
def leer(texto):
    return input(texto)

def imprimir_resultado(expresion, valor):
    print(f"{expresion} = {valor}")
```

Las definiciones aritméticas reutilizan las funciones de conversión y construcción del programa anterior, añadiendo las codificaciones de SUMA y MULT como λ-términos.

Código 4.18: Conversión, construcción y operaciones

```
27 def church_to_int(n):
28     return n(lambda x: x + 1)(0)
29
30 def construir_numeral(n):
31     numeral = CERO
32     for _ in range(n):
33         numeral = SUCC(numeral)
34     return numeral
35
36 CERO = lambda f: lambda x: x
37 SUCC = lambda n: lambda f: lambda x: f(n(f)(x))
38 SUMA = lambda m: lambda n: lambda f: lambda x: m(f)(n(f)(x))
39 MULT = lambda m: lambda n: lambda f: m(n(f))
```

4.5. Ejercicios

4.5.1 Trazas de Máquinas de Turing

Para la MT que reconoce $\{0^n 1^n \mid n \geq 1\}$ (Ejemplo 2 del capítulo), trace la ejecución completa para las siguientes cadenas:

a) $w = 01$

b) $w = 000111$

c) $w = 0111$ (debe rechazar)

4.5.2 Diseño de Máquinas de Turing

Diseñe una MT que reconozca el lenguaje $L = \{w \in \{a, b\}^* \mid w \text{ termina en } ab\}$.

4.5.3 Clasificación de lenguajes

Clasifique cada lenguaje como regular, libre de contexto (no regular), o decidible (no LLC). Justifique brevemente.

a) $L_1 = \{a^n b^m \mid n < m\}$

b) $L_2 = \{a^n b^n c^n \mid n \geq 0\}$

c) $L_3 = \{ww^R \mid w \in \{a, b\}^*\}$

4.5.4 Clasificación y diseño: palíndromos binarios

Considere el lenguaje de los palíndromos binarios $L_{pal} = \{w \in \{0, 1\}^* \mid w = w^R\}$.

a) Clasifique L_{pal} dentro de la jerarquía de lenguajes. ¿Es regular? ¿Es libre de contexto? ¿Es decidible? Justifique cada respuesta.

   Pista: Para descartar regularidad, use el lema de bombeo con la cadena 0^p10^p. Para confirmar que es LLC, proponga una gramática libre de contexto.

b) Diseñe una MT que reconozca L_{pal}. La estrategia sugerida es: borrar el primer símbolo (recordando si es 0 o 1), ir al final y verificar que el último coincida, borrarlo y volver al inicio. Repetir hasta que la cadena quede vacía (aceptar) o haya discrepancia (rechazar).

c) Trace la ejecución de su MT para las cadenas $w = 1001$ (debe aceptar) y $w = 1010$ (debe rechazar).

d) Implemente su MT usando el simulador genérico `MaquinaTuring` y verifique con las cadenas: `1001`, `10101`, `1010`, `0`, `11011`.

4.5.5 β-reducción en cálculo lambda

Reduzca las siguientes expresiones paso a paso:

a) $(\lambda x.x)\,(\lambda y.y)$

b) $(\lambda x.\lambda y.x)\,a\,b$

c) $(\lambda f.f\,(f\,x))\,(\lambda y.y)$

4.5.6 Análisis de decidibilidad

Determine si cada problema es decidible o indecidible. Justifique.

a) Dado un AFD M, ¿acepta M alguna cadena?

b) Dada una GLC G, ¿genera G al menos una cadena?

c) Dado un programa P y una entrada x, ¿termina $P(x)$ en menos de 1000 pasos?

d) Dados dos programas P_1 y P_2, ¿producen la misma salida para toda entrada?

4.5.7 Equivalencia de variantes de MT

Explique intuitivamente cómo una MT de una cinta puede simular una MT de dos cintas.

4.5.8 Reflexión sobre los límites de la computación

a) ¿Implica la indecidibilidad del problema de la parada que nunca podremos verificar programas? Explique qué pueden y qué no pueden hacer las herramientas de verificación.

b) Si la Tesis de Church-Turing es correcta, ¿qué implicaciones tiene para la inteligencia artificial? ¿Puede una IA resolver problemas que una MT no puede?

4.5.9 Implementación práctica

Usando el simulador de Máquina de Turing proporcionado en la sección de codificación:

a) Implemente las transiciones de la MT para $\{a^n b^n c^n \mid n \geq 1\}$ usando el diccionario de transiciones.

b) Pruebe con las cadenas: `abc`, `aabbcc`, `aaabbbccc`, `aabbc`, `abcc`.

c) Modifique el simulador para que muestre la traza paso a paso (estado actual, contenido de la cinta, posición de la cabeza).

4.6. Cierre del capítulo

En este capítulo se presentaron las Máquinas de Turing y el cálculo lambda como modelos que delimitan los límites de la computación. Ambos formalizan la noción de "computable" y muestran que existen problemas bien definidos sin solución algorítmica.

Conceptos clave aprendidos

1. **Máquina de Turing (MT)**: Modelo de computación basado en una cinta potencialmente infinita, una cabeza de lectura/escritura y un conjunto finito de estados. Permite reconocer lenguajes más complejos que los libres de contexto, como $\{a^n b^n c^n \mid n \geq 0\}$.

2. **Cálculo Lambda**: Modelo funcional basado únicamente en definición y aplicación de funciones. Su operación fundamental es la β-reducción (sustitución), base teórica de muchos lenguajes funcionales modernos.

3. **Tesis de Church-Turing**: Idea de que todo proceso efectivamente computable puede realizarse mediante una MT o expresarse en cálculo lambda. No corresponde a un teorema formal, aunque distintos modelos desarrollados de manera independiente han resultado equivalentes en poder computacional.

4. **Decidibilidad**: Un problema es decidible si existe un algoritmo que siempre termina y produce una respuesta correcta. Ejemplos: determinar si una cadena pertenece a $a^n b^n c^n$ o verificar si un AFD acepta alguna cadena.

5. **Indecidibilidad**: Existen problemas bien definidos que no admiten una solución algorítmica general. El problema de la parada es el ejemplo más representativo: no existe un programa capaz de determinar si un programa arbitrario terminará.

6. **Equivalencia de modelos**: La MT, el cálculo lambda, las funciones recursivas y otros formalismos poseen el mismo poder computacional. Aunque surgieron de manera independiente, todos describen la misma noción de computación efectiva.

Capítulo 5

Aplicaciones modernas

5.1. Introducción

En los sistemas computacionales actuales, los lenguajes formales pocas veces son visibles para el usuario final. Sin embargo, hacen parte de la estructura fundamental de muchas herramientas y servicios digitales utilizados diariamente. Compiladores, entornos de desarrollo, sistemas críticos, infraestructuras distribuidas y aplicaciones de inteligencia artificial dependen de modelos formales para garantizar un comportamiento correcto y confiable.

A lo largo de este libro se han estudiado los modelos fundamentales de la teoría de lenguajes formales y de la computación: lenguajes regulares, lenguajes libres de contexto, autómatas finitos, autómatas de pila, máquinas de Turing y cálculo lambda. Cada uno de ellos permitió responder una pregunta central:

> *¿Qué puede reconocerse, generarse o decidirse mediante un sistema computacional?*

No obstante, en el ejercicio profesional el ingeniero no diseña directamente un autómata aislado ni redacta una gramática como fin en sí mismo. Lo que hace es tomar decisiones de arquitectura, definir restricciones estructurales, modelar comportamientos y automatizar procesos. Todas estas decisiones dependen, de manera explícita o implícita, del poder expresivo y de las limitaciones de los modelos formales estudiados.

Comprender esta relación permite identificar qué aspectos de un sistema pueden automatizarse, cuáles pueden verificarse de manera rigurosa y dónde aparecen los límites inherentes a la computabilidad. La ingeniería moderna no se presenta como una colección de modelos teóricos aislados, sino como arquitecturas complejas que integran múltiples niveles de análisis, validación y transformación de información.

Este capítulo busca cerrar el libro mostrando cómo los lenguajes formales sirven como base para el diseño de sistemas reales. Más que un conjunto de conceptos abstractos, estos modelos permiten construir software con mayor precisión, seguridad y capacidad de automatización.

5.2. Validación de tramas en un sistema de frenado autónomo

Un fabricante de vehículos autónomos está desarrollando un sistema de frenado de emergencia que funciona a partir de la comunicación entre varios sensores (lídar, radar y cámara) y una unidad central. Cada uno de estos sensores envía tramas binarias de 8 bits.

Durante las pruebas se detectó que algunas tramas, aunque tenían la longitud correcta, contenían inconsistencias que provocaban activaciones erróneas del sistema.

5.2.1 Análisis / Abstracción

a. ¿Cuál es el problema a resolver?

Se necesita validar automáticamente que una trama de 8 bits sea correcta (estructura y coherencia de datos) antes de ser procesada.

También hay que definir formalmente el lenguaje de tramas válidas y garantizar su verificación en tiempo real.

b. ¿Cuál es la información necesaria?

Elemento	Descripción
Alfabeto	$\Sigma = \{0, 1\}$
Longitud	8 bits
Tipo de obstáculo	Bits 1–3 (001, 010, 011)
Distancia	Bits 4–6 (0 a 7)
Código de verificación	Bits 7–8 (paridad de los primeros 6 bits)

5.2.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **Definir** el alfabeto binario $\Sigma = 0, 1$.
- **Especificar** los subconjuntos válidos.
- **Verificar** la longitud Σ^8 y la paridad.
- **Mostrar** el resultado

b. ¿Qué conocimiento adicional se requiere?

Resolver este problema implica entender cómo los lenguajes formales modelan estructuras válidas y cómo los autómatas permiten validarlas en tiempo real.

Lenguajes regulares

> Un lenguaje regular es un conjunto de cadenas que se puede describir con expresiones regulares o reconocer con un autómata finito. En concreto, un autómata finito determinista (AFD) puede reconocerlo.

Para este problema, el conjunto de tramas válidas es:

$$L \subseteq \{0, 1\}^8$$

Interpretación:

Esto significa que cada trama es una cadena de longitud fija (8 bits), pero no todas las cadenas de 8 bits son válidas. L solo contiene aquellas que cumplen ciertas restricciones, tanto de estructura como de consistencia.

Estructura de la trama

Campo	Bits	Descripción
Tipo	1–3	Tipo de obstáculo
Distancia	4–6	Distancia codificada
Verificación	7–8	Paridad

Autómatas finitos deterministas (AFD)

Un AFD es una máquina abstracta que lee una cadena símbolo a símbolo y decide si la cadena está en el lenguaje.

Componentes:

- Estados (Q)
- Alfabeto (Σ)
- Función de transición (δ)
- Estado inicial (q_0)
- Estados de aceptación (F)

Validación semántica

Aquí hay una dependencia entre campos, algo que no ocurre en validaciones simples

- El código de verificación depende de los primeros 6 bits.
- Esto introduce una relación funcional dentro del lenguaje.

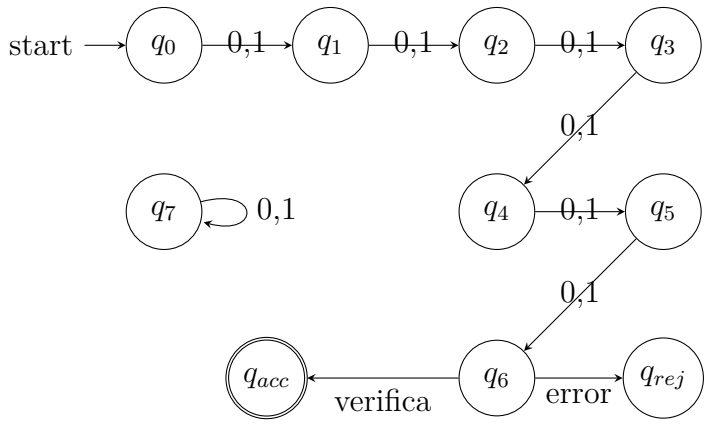
Cálculo de paridad

Paridad	Código esperado
Par	00
Impar	01

Importancia:

- Permite detectar errores en transmisión.
- Garantiza consistencia de datos.

Diagrama de estados del autómata



c. ¿Qué se puede reutilizar de soluciones pasadas?

En este problema se reutilizan varios conceptos fundamentales desarrollados en capítulos anteriores:

- **Alfabetos y lenguajes formales**: se retoma la definición de un lenguaje como subconjunto de Σ^*, en este caso restringido a Σ^8.
- **Expresiones regulares**: permiten describir patrones de cadenas con estructura fija, útiles para modelar formatos de tramas.
- **Autómatas finitos deterministas (AFD)**: se emplea la idea de un reconocedor que procesa la cadena símbolo a símbolo y decide su aceptación.
- **Validación de cadenas**: se reutiliza el enfoque de verificar pertenencia al lenguaje mediante condiciones estructurales.

Estos elementos permiten construir un sistema de validación eficiente para cadenas de longitud fija, como las tramas del sistema de frenado.

d. Diseño de la solución para el problema

La solución propuesta consiste en construir un modelo formal que, usando criterios sintácticos y semánticos, pueda diferenciar las tramas válidas de las inválidas. El lenguaje de tramas se define como un subconjunto de cadenas binarias de longitud fija, en el que no solo interesa la forma de la cadena, sino también cómo se relacionan sus partes internamente.

Se contemplan dos niveles de validación: uno estructural (la forma de la cadena) y otro de consistencia (relaciones entre distintas partes de la trama). Así se pueden modelar situaciones reales de sistemas críticos, donde la validez de la información depende de varias condiciones que interactúan entre sí.

En cuanto a lo computacional, el modelo funciona como un reconocedor que evalúa cadenas y decide si pertenecen al lenguaje, imitando el comportamiento de un autómata pero con verificaciones adicionales.

5.2.3 Ejemplos de pruebas

a. Ejemplo 1

Cadena: 00101000

- Tipo válido
- Paridad correcta

Resultado: ACEPTADA

b. Ejemplo 2

Cadena: 00101001

- Código incorrecto

Resultado: RECHAZADA

Otros ejemplos:

Trama	Resultado	Motivo
00011100	RECHAZADA	Tipo inválido y verificación incorrecta
00102A00	RECHAZADA	Símbolos fuera del alfabeto $\Sigma = \{0, 1\}$
01010	RECHAZADA	Longitud incorrecta (menor a 8 bits)

5.2.4 Codificación

Una vez definido el lenguaje formal de tramas válidas y construido el autómata finito determinista que lo reconoce, es posible traducir estos elementos a una implementación computacional. Esta codificación permite simular el comportamiento del reconocedor formal en un entorno de programación, facilitando la validación automática de tramas en un sistema real. Esta sección muestra cómo traducir dichas representaciones a Python.

a. Algoritmo general de reconocimiento

Integración del sistema

Hay una función principal que se encarga de coordinar todos los componentes. Conceptualmente, esa función es como el sistema completo de reconocimiento, porque reúne la definición del alfabeto, la captura de la entrada, el procesamiento y la salida.

Esta forma de organizar el sistema (modular) es una buena práctica tanto en programación como en el diseño formal, ya que separa bien las responsabilidades y hace que el sistema sea más fácil de entender, verificar y mantener.

Código 5.1: `main()`

```
1 def main():
2     alfabeto = definir_alfabeto()
3     tramas = pedir_tramas()
4     resultados = validar_tramas(tramas, alfabeto)
5     mostrar_resultados(resultados)
```

Definición del alfabeto

Aquí el alfabeto es $\{0, 1\}$. Una función define explícitamente este conjunto de símbolos permitidos, lo que luego permite verificar que cada símbolo de la trama pertenezca al alfabeto y así asegurar la validez sintáctica básica.

Código 5.2: `definir_alfabeto()`

```
10 def definir_alfabeto():
11     alfabeto = {'0', '1'}
12
13     return alfabeto
```

Captura de entrada

El proceso de captura de entrada corresponde a la obtención de cadenas que serán evaluadas respecto al lenguaje definido. En términos formales, estas cadenas son candidatos a pertenecer al lenguaje $L \subseteq \Sigma^8$. Desde la perspectiva de implementación, esta etapa se encarga de recolectar las tramas ingresadas por el usuario, simulando la recepción de datos desde sensores en un sistema embebido. Cada cadena capturada será posteriormente procesada por el validador, que actúa como un reconocedor del lenguaje.

Código 5.3: `pedir_tramas()`

```
18 def pedir_tramas():
19     tramas = []
20     cantidad = int(input("Ingrese la cantidad de tramas a validar: "))
21
22     for i in range(cantidad):
23         trama = input(f"Ingrese la trama {i+1}: ")
24         tramas.append(trama)
25
26     return tramas
```

Validación mediante un reconocedor

La validación de las tramas corresponde al proceso de decidir si una cadena pertenece o no al lenguaje L. Esta tarea es realizada por un autómata finito determinista, el cual procesa la cadena símbolo a símbolo y alcanza un estado de aceptación o rechazo. En la implementación,

este comportamiento se traduce en una función de validación que aplica secuencialmente las restricciones definidas: verificación del alfabeto, longitud de la cadena, validez de los campos y consistencia del código de verificación. Aunque no se implementa explícitamente la estructura del autómata, la lógica condicional refleja su funcionamiento, donde cada validación puede interpretarse como una transición hacia un estado de aceptación o rechazo.

Código 5.4: `validar_trama()`

```
31 def validar_tramas(tramas, alfabeto):
32     resultados = []
33 
34     for trama in tramas:
35         resultado = "Valida"
36         contiene_error = any(bit not in alfabeto for bit in trama)
37 
38         if contiene_error:
39             resultado = "Contiene simbolos invalidos"
40         else:
41 
42             if len(trama) != 8:
43                 resultado = "Longitud invalida"
44             else:
45                 tipo = trama[:3]
46                 distancia = trama[3:6]
47                 verificacion = trama[6:]
48                 tipos_validos = {"001", "010", "011"}
49 
50                 if tipo not in tipos_validos:
51                     resultado = "Tipo invalido"
52                 else:
53                     suma = sum(int(b) for b in trama[:6])
54                     esperado = "00" if suma % 2 == 0 else "01"
55 
56                     if verificacion != esperado:
57                         resultado = "Error en verificacion"
58 
59         resultados.append((trama, resultado))
60 
61     return resultados
```

Presentación de resultados

Una vez evaluada la pertenencia de cada cadena al lenguaje, es necesario comunicar el resultado del proceso. En términos formales, esto equivale a indicar si la cadena es aceptada o rechazada por el reconocedor.

Esta etapa recorre las tramas procesadas y muestra el resultado de la validación, permitiendo visualizar el comportamiento del sistema y verificar que la implementación coincida con la especificación formal del lenguaje.

Código 5.5: `mostrar_resultados()`

```
66 def mostrar_resultados(resultados):
67     for trama, resultado in resultados:
68         print(f"Trama: {trama} -> {resultado}")
```

5.3. Análisis sintáctico de consultas médicas en inteligencia artificial

En el contexto de la transformación digital en el sector salud, un equipo de investigación está desarrollando un sistema de apoyo al diagnóstico médico basado en inteligencia artificial. Este sistema tiene como objetivo asistir a los profesionales de la salud en la interpretación de consultas clínicas escritas en lenguaje natural, permitiendo extraer información estructurada que pueda ser utilizada para la toma de decisiones, la consulta de bases de conocimiento y la generación de posibles diagnósticos.

Las consultas médicas suelen ser lingüísticamente muy complejas, porque en una misma oración pueden aparecer varios síntomas, localizaciones, intensidades y relaciones semánticas. Por ejemplo, una frase como ”paciente presenta dolor torácico con irradiación a brazo izquierdo y disnea de esfuerzo” incluye varias entidades clínicas que están relacionadas entre sí. Para interpretarla correctamente no basta con entender las palabras una por una; también hay que captar la estructura sintáctica que las organiza.

Al probar el sistema, se vio que los modelos basados solo en redes neuronales tenían problemas con este tipo de estructuras complejas. En concreto, se detectaron dificultades para interpretar oraciones con varios síntomas coordinados, negaciones, modificadores y estructuras anidadas. Estas limitaciones se deben principalmente a que no hay una representación explícita y jerárquica del proceso.

5.3.1 Análisis / Abstracción

a. ¿Cuál es el problema a resolver?

Es necesario modelar las consultas médicas de manera formal, con el fin de mejorar su interpretación y poder procesar estructuras lingüísticas complejas. Para ello se requiere una representación que capture la jerarquía sintáctica de las oraciones.

b. ¿Cuál es la información necesaria?

Elemento	Descripción
Terminales	Palabras médicas
Variables	Categorías sintácticas
Reglas	Producciones
Estructura	Árbol de derivación

5.3.2 Diseño / Descomposición

a. ¿Cuáles son los pasos lógicos necesarios para resolverlo?

- **Definir** el alfabeto de terminales.
- **Ingresar** las consultas.
- **Validar** las consultas
- **Mostrar** el resultado según la consulta ingrada.

b. ¿Qué conocimiento adicional se requiere?

Gramáticas libres de contexto

Una GLC describe lenguajes jerárquicos mediante reglas de producción.

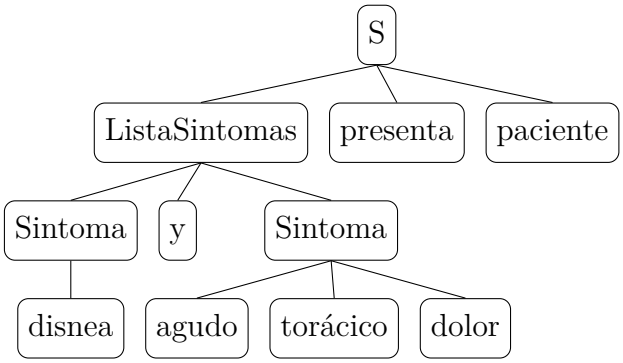
Estructura del lenguaje médico

Categoría	Ejemplos
Síntomas	dolor, fiebre
Localización	torácico, abdominal
Intensidad	leve, severo
Negación	no, sin

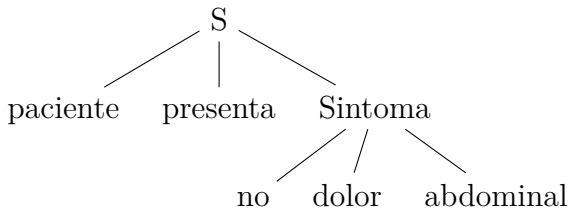
Gramática del sistema

$$S \rightarrow \text{"paciente" "presenta" } Lista$$

Árbol de derivación principal



Árbol con negación



c. ¿Qué se puede reutilizar de soluciones pasadas?

Este problema se apoya en varios conceptos clave de lenguajes con estructura jerárquica:

- **Gramáticas libres de contexto (GLC)**: sirven para modelar las estructuras recursivas y jerárquicas que aparecen en el lenguaje natural.
- **Derivaciones y árboles sintácticos**: se usan para representar cómo se organiza internamente una cadena.
- **Reconocimiento de cadenas**: se retoma la noción de comprobar si una cadena pertenece a un lenguaje definido formalmente.

- **Procesamiento simbólico**: se aprovecha la división de las cadenas en tokens para poder analizarlas mejor.

Estos elementos permiten integrar enfoques simbólicos dentro de sistemas modernos de inteligencia artificial.

d. Diseño de la solución para el problema

La solución propuesta incorpora un modelo estructural del lenguaje que representa de forma explícita cómo se organizan internamente las consultas médicas. En vez de ver las cadenas como simples secuencias de palabras, se utiliza una jerarquía que refleja las relaciones sintácticas entre los diferentes elementos.

Este enfoque combina lenguajes formales con técnicas actuales de inteligencia artificial, añadiendo una parte simbólica que complementa a los modelos de aprendizaje automático. La gramática sirve como un marco que limita y ordena las posibles interpretaciones de la entrada, ayudando a extraer la información relevante.

El sistema final no solo determina si una cadena es válida, sino que también entrega una estructura que pueden aprovechar otros modelos, mejorando así la interpretabilidad y la solidez del análisis.

5.3.3 Ejemplos de pruebas

a. Ejemplo 1

Entrada: paciente presenta fiebre

Resultado: ACEPTADA

b. Ejemplo 2

Entrada: paciente dolor

Resultado: RECHAZADA

Otros Ejemplo:

Consulta	Resultado	Motivo
paciente presenta fiebre y tos	ACEPTADA	Uso correcto de conector
paciente presenta disnea con dolor	ACEPTADA	Secuencia válida de síntomas
paciente dolor torácico	RECHAZADA	Falta palabra clave “presenta”
presenta dolor	RECHAZADA	Falta sujeto “paciente”
paciente presenta xyz	RECHAZADA	Palabra no reconocida en el lenguaje
paciente presenta	RECHAZADA	Estructura incompleta

Importancia

- Mejora interpretación
- Reduce ambigüedad
- Aumenta precisión

5.3.4 Codificación

Una vez definida la gramática independiente de contexto que modela las consultas médicas, es posible implementar un analizador sintáctico simplificado que verifique si una cadena cumple con la estructura definida. Esta implementación simula el proceso de reconocimiento de una GIC, permitiendo validar la forma estructural de las consultas.

El sistema se organiza en funciones que reflejan la descomposición del problema: definición de los elementos del lenguaje, captura de entrada, validación estructural y presentación de resultados.

a. Algoritmo general de reconocimiento

Integración del sistema

La función principal coordina la ejecución de todos los componentes definidos.

Código 5.6: `main()`

```
def main():
    lenguaje = definir_lenguaje()
    consultas = pedir_consultas()
    resultados = validar_consultas(consultas, lenguaje)
    mostrar_resultados(resultados)
```

Definición del lenguaje

En este caso, el lenguaje está compuesto por palabras agrupadas en categorías como síntomas, conectores y modificadores. Estos elementos corresponden a los símbolos terminales de la gramática.

La implementación representa estos conjuntos como estructuras de datos para validar si las palabras de entrada pertenecen al lenguaje.

Código 5.7: `definir_lenguaje()`

```
def definir_lenguaje():
    lenguaje = {
        "inicio": ["paciente", "presenta"],
        "sintomas": ["dolor", "fiebre", "tos", "disnea"],
        "conectores": ["y", "con"]
    }
    return lenguaje
```

Captura de entrada

La entrada corresponde a consultas médicas expresadas como cadenas de texto. Estas cadenas serán evaluadas respecto a la estructura definida por la gramática.

La captura de datos simula el ingreso de información clínica al sistema.

Código 5.8: `pedir_consultas()`

```
def pedir_consultas():
    consultas = []
    n = int(input("Cantidad de consultas: "))

    for i in range(n):
        consulta = input(f"Consulta {i+1}: ")
        consultas.append(consulta.lower())

    return consultas
```

Validación estructural

La validación consiste en verificar si la secuencia de palabras sigue una estructura válida según la gramática. Aunque no se implementa un parser completo, se simula la verificación mediante reglas que capturan patrones válidos.

Este proceso representa una aproximación al reconocimiento de lenguajes libres de contexto.

Código 5.9: `validar_consulta()`

```
34 def validar_consultas(consultas, lenguaje):
35     resultados = []
36 
37     for consulta in consultas:
38         resultado = "Valida"
39         palabras = consulta.split()
40 
41         if len(palabras) < 3:
42             resultado = "Estructura incompleta"
43         else:
44 
45             if palabras[0] != "paciente" or palabras[1] != "presenta":
46                 resultado = "Inicio invalido"
47             else:
48 
49                 for palabra in palabras[2:]:
50                     if (palabra not in lenguaje["sintomas"] and
51                         palabra not in lenguaje["conectores"]):
52                         resultado = "Palabras no reconocidas"
53 
54         resultados.append((consulta, resultado))
55 
56     return resultados
```

Presentación de resultados

Este paso permite visualizar el comportamiento del sistema y verificar que la implementación corresponde correctamente a la especificación formal del lenguaje.

Código 5.10: `mostrar_resultados()`

```
61 def mostrar_resultados(resultados):
62     for consulta, resultado in resultados:
63         print(f"{consulta} -> {resultado}")
```

5.4. Verificación de un controlador de semáforos inteligente

En el marco de las ciudades inteligentes, una administración municipal ha implementado un sistema de semáforos inteligentes cuyo objetivo es optimizar el flujo vehicular y mejorar la seguridad vial mediante el uso de sensores y algoritmos de control adaptativo. Este sistema recibe información en tiempo real sobre el tráfico, la presencia de peatones y la aproximación de vehículos de emergencia, y ajusta dinámicamente los ciclos de los semáforos en función de dichas condiciones.

El controlador central del sistema tiene que manejar varias entradas y asegurarse de que las combinaciones de luces (rojo, amarillo y verde) en cada cruce respeten reglas muy estrictas de seguridad y eficiencia. Pero durante las pruebas de integración aparecieron comportamientos problemáticos, como que se encendieran luces verdes al mismo tiempo en direcciones que chocan entre sí, o que ciertos estados se quedaran activos de manera indefinida. Esto puede provocar riesgos de colisión o congestión vehicular.

5.4.1 Análisis / Abstracción

a. ¿Cuál es el problema a resolver?

Verificar que un sistema de semáforos cumple propiedades de seguridad y vivacidad mediante model checking.

b. ¿Cuál es la información necesaria?

Elemento	Descripción
Estados	Configuración del sistema
Eventos	Entradas del sistema
Transiciones	Cambios de estado
Propiedades	Fórmulas LTL

5.4.2 Diseño / Descomposición

- **Definir** el conjunto de estados del sistema.
- **Solicitar** el eventos.
- **Validar** la secuencia.
- **Mostrar** la validación de la secuencia ingresada.

a. ¿Qué conocimiento adicional se requiere?

Para resolverlo, se necesita entender el modelado de sistemas como autómatas y la verificación de propiedades con lógica temporal.

Autómatas como modelos de sistemas

Un sistema con finitas configuraciones se representa como un autómata finito (sistema reactivo que responde a eventos).

$$M = (Q, \Sigma, \delta, q_0)$$

Estados del sistema

Estado	Descripción
NSV	Norte-Sur verde
NSA	Norte-Sur amarillo
EOV	Este-Oeste verde
EOA	Este-Oeste amarillo

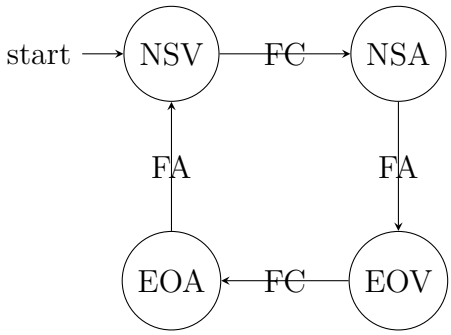


Diagrama de estados del sistema

Lógica temporal lineal (LTL)

La **lógica temporal** permite expresar propiedades sobre el comportamiento de un sistema en el tiempo.

- $\Box\neg(\text{NSV} \wedge \text{EOV})$
- $\Box(\text{BPNS} \rightarrow \Diamond\text{PNS})$

Operadores:

Símbolo	Significado
$\Box$	Siempre
$\Diamond$	Eventualmente
$\bigcirc$	Siguiente Estado

Tipos de propiedades

Tipo	Ejemplo
Seguridad	Nunca ocurre colisión
Vivacidad	Toda solicitud es atendida

Ejemplo de propiedad

$$\Box\neg colision$$

Interpretación:

En todos los estados posibles, no debe ocurrir colisión.

Model checking

El **model checking** es una técnica automática que verifica si un modelo cumple una propiedad.

Model checking

- Exploración exhaustiva del sistema
- Verificación automática
- Generación de contraejemplos

Ventajas:

- Detecta errores automáticamente
- Explora todos los estados posibles
- Genera contraejemplos

b. ¿Qué se puede reutilizar de soluciones pasadas?

Este problema retoma varios conceptos ya vistos en el libro:

- **Autómatas finitos**: sirven para modelar sistemas con una cantidad limitada de estados.
- **Lenguajes reconocidos**: el comportamiento del sistema se ve como secuencias de eventos que el autómata acepta.
- **Validación de secuencias**: se vuelve a usar la idea de recorrer cadenas símbolo por símbolo con transiciones.
- **Modelado de sistemas**: se amplía el uso de autómatas, de los lenguajes a los sistemas reactivos.

Estos conceptos permiten abordar problemas de verificación formal en sistemas reales.

c. Diseño de la solución para el problema

La solución propuesta modela el comportamiento del sistema como un conjunto finito de estados y transiciones. Esto permite representar formalmente todas las configuraciones posibles del controlador. Así, el problema de validación se convierte en un análisis de comportamiento, en el que se examinan las secuencias de eventos que el sistema puede procesar.

La idea detrás del diseño es que un sistema reactivo se puede ver como un lenguaje de secuencias válidas, reconocido por un autómata. Con ese modelo se pueden analizar propiedades del sistema, no solo mirando si acepta o no ciertas secuencias, sino también estudiando su comportamiento en el tiempo.

Esto es la base de la verificación automática: el modelo formal permite encontrar errores que las pruebas convencionales no detectan fácilmente, y ofrece garantías más sólidas de que el sistema funciona correctamente.

5.4.3 Ejemplos de pruebas

a. Ejemplo 1: atención a solicitud peatonal

- Secuencia: `BPNS, FC`
- Evolución:

$$NSV \xrightarrow{BPNS} PNS \xrightarrow{FC} NSV$$

- **Evaluación:**
  - Se verifica la propiedad de vivacidad:

$$\square(BPNS \rightarrow \Diamond PNS)$$

  - El sistema responde correctamente a la solicitud del peatón.

Resultado: CORRECTO

b. Ejemplo 2: atención a evento de emergencia

- Secuencia: `EMGEO, FC`
- Evolución:

$$NSV \xrightarrow{EMGEO} EMGEO \xrightarrow{FC} EOV$$

- **Evaluación:**
  - Se valida la prioridad de eventos de emergencia.
  - Se cumple la propiedad de vivacidad asociada a emergencias.

Resultado: CORRECTO

Otros ejemplos:

Secuencia de eventos	Resultado	Motivo
FC, FA, FC, FA	ACEPTADA	Ciclo normal completo
FC, FA	ACEPTADA	Transiciones válidas del autómata
FC, FC	RECHAZADA	Transición no permitida desde NSA
FA	RECHAZADA	Evento inválido desde estado inicial
XYZ	RECHAZADA	Evento no pertenece al alfabeto
FC, FA, XYZ	RECHAZADA	Evento inválido en la secuencia

Importancia

- Garantía formal
- Detección de errores
- Seguridad en sistemas críticos

5.4.4 Codificación

El controlador de semáforos se implementa como un autómata finito: los estados son configuraciones del sistema, y las transiciones responden a eventos.

Esto permite simular el sistema y verificar si las secuencias de eventos llevan a estados válidos.

a. Algoritmo general de reconocimiento

Integración del sistema

La función principal coordina la ejecución de todos los componentes definidos.

Código 5.11: `main()`

```
def main():
    automata = definir_automata()
    eventos = pedir_eventos()
    resultado = validar_secuencia(eventos, automata)
    mostrar_resultado(resultado, eventos)
```

Definición del autómata

El sistema se modela con un autómata finito determinista: estados = configuraciones de semáforos, transiciones = eventos del entorno.

La implementación representa la función de transición con estructuras de datos.

Código 5.12: `definir_automata()`

```
def definir_automata():
    automata = {
        ("NSV", "FC"): "NSA",
        ("NSA", "FA"): "EOV",
        ("EOV", "FC"): "EOA",
        ("EOA", "FA"): "NSV"
    }
    return automata
```

Captura de eventos

Los eventos (sensores, solicitudes, etc.) son las entradas del sistema y se procesan secuencialmente para definir su comportamiento.

Código 5.13: `pedir_eventos()`

```
22 def pedir_eventos():
23     eventos = []
24     n = int(input("Cantidad de eventos: "))
25 
26     for i in range(n):
27         evento = input(f"Evento {i+1}: ")
28         eventos.append(evento)
29 
30     return eventos
```

Simulación del autómata

Validar es recorrer el autómata con la secuencia de eventos. Si falta una transición válida en algún paso, se rechaza.

Así se simula el reconocimiento de secuencias válidas del sistema.

Código 5.14: `validar_secuencia()`

```
35 def validar_secuencia(eventos, automata):
36     estado = "NSV"
37     resultado = "Secuencia valida"
38 
39     for evento in eventos:
40 
41         if (estado, evento) in automata:
42             estado = automata[(estado, evento)]
43         else:
44             resultado = "Secuencia invalida"
45 
46     return resultado
```

Presentación de resultados

Este paso permite visualizar el comportamiento del sistema y verificar que la implementación corresponde correctamente a la especificación formal del lenguaje.

Código 5.15: `mostrar_resultados()`

```
51 def mostrar_resultado(resultado, eventos):
52     print(f"Eventos: {eventos} -> {resultado}")
```

5.5. Cierre del capítulo

A lo largo de este capítulo se mostró cómo los lenguajes formales dejan de ser únicamente un tema teórico y se convierten en herramientas fundamentales para construir, analizar y verificar sistemas complejos.

Los tres escenarios estudiados (sistemas críticos, inteligencia artificial y verificación automática) reflejan una necesidad común en la ingeniería moderna: utilizar modelos formales para garantizar confiabilidad, interpretabilidad y corrección.

En el caso de los sistemas críticos, la validación de tramas en el sistema de frenado autónomo muestra cómo incluso estructuras aparentemente simples requieren una caracterización formal precisa. La distinción entre validez sintáctica y semántica resulta esencial, y los autómatas finitos permiten implementar mecanismos de validación eficientes bajo restricciones estrictas de tiempo y recursos. Este ejemplo evidencia la importancia de los lenguajes regulares en sistemas embebidos y de control.

Por otra parte, el análisis sintáctico de consultas médicas ilustra la integración entre modelos formales e inteligencia artificial. Las gramáticas libres de contexto permiten representar la estructura jerárquica del lenguaje y construir árboles de derivación útiles para tareas como desambiguación, identificación de relaciones semánticas y reducción del espacio de búsqueda. Este enfoque muestra que los modelos formales continúan siendo relevantes en problemas modernos de procesamiento del lenguaje natural.

Finalmente, la verificación del controlador de semáforos introduce un nivel más avanzado de análisis, donde no solo se modela el comportamiento del sistema, sino que también se verifican propiedades temporales sobre todos sus posibles estados. La lógica temporal lineal y el model checking permiten comprobar propiedades de seguridad y vivacidad, superando las limitaciones de las pruebas tradicionales y proporcionando garantías formales sobre el funcionamiento del sistema.

Estos ejemplos también muestran una progresión en la capacidad expresiva de los modelos formales: los lenguajes regulares describen patrones locales, las gramáticas libres de contexto permiten representar estructuras jerárquicas y la lógica temporal hace posible razonar sobre comportamientos dinámicos. Esta evolución permite abordar problemas progresivamente más complejos.

Una idea central del capítulo es que los lenguajes formales proporcionan

herramientas para diseñar sistemas más confiables. Permiten especificar comportamientos de manera precisa, detectar errores tempranamente, verificar propiedades críticas e integrar modelos simbólicos con técnicas de aprendizaje automático.

En un contexto donde los sistemas computacionales son cada vez más complejos, los lenguajes formales se mantienen como una herramienta esencial para razonar rigurosamente sobre el comportamiento de software y hardware.

Al finalizar este libro, la idea principal es entender que los conceptos estudiados no son únicamente teoría, sino herramientas para diseñar soluciones más confiables y estructuradas. El dominio de estos fundamentos permite comprender cómo funcionan los sistemas y desarrollar software con criterios formales de corrección.

Conclusiones

La Teoría de Lenguajes Formales es una de las bases fundamentales de la ciencia de la computación, no solo por su importancia teórica, sino porque aparece en muchos problemas reales, desde compiladores hasta sistemas de verificación. A lo largo de este libro se estudió un conjunto de modelos que permiten describir lenguajes, analizar estructuras y comprender qué problemas pueden resolverse mediante algoritmos.

Los temas se presentaron a partir de problemas concretos y no únicamente desde definiciones formales. La intención fue que cada concepto surgiera como respuesta a una necesidad específica, mostrando que estos modelos no son construcciones aisladas, sino herramientas útiles para representar y resolver problemas.

También se buscó relacionar los contenidos con ideas de pensamiento computacional. La abstracción permite identificar los elementos esenciales de un problema; la descomposición ayuda a dividirlo en partes manejables; el reconocimiento de patrones muestra similitudes entre distintos escenarios; y la codificación se entiende como la construcción de modelos formales capaces de representar soluciones.

En los lenguajes regulares se trabajó con problemas donde basta reconocer patrones simples mediante autómatas finitos, como ocurre en el análisis léxico de compiladores.

Posteriormente, los lenguajes libres de contexto introdujeron estructuras más complejas, especialmente aquellas con jerarquía y anidamiento. En este caso ya no basta con procesar símbolos secuencialmente, sino que es necesario comprender la estructura interna de las cadenas, aspecto fundamental en el análisis sintáctico y en los lenguajes de programación.

Más adelante se estudiaron las Máquinas de Turing y el cálculo lambda, modelos que permiten abordar la computación en un sentido más general. A partir de ellos surge una pregunta fundamental: qué problemas pueden resolverse mediante algoritmos y cuáles no. Esto conduce a una de las ideas centrales de la teoría de la computación: existen problemas que no admiten solución computable, independientemente del lenguaje o la máquina utilizada.

Otro aspecto importante es que muchos modelos distintos poseen el mismo poder computacional. Esto muestra que los límites de lo computable no dependen del lenguaje de programación ni del hardware, sino de la naturaleza del problema.

En la parte final del libro se mostró que estos conceptos tienen aplicaciones directas en compiladores, procesamiento de lenguaje natural, verificación de sistemas e inteligencia artificial. En todos estos contextos, los modelos formales ayudan a construir sistemas más claros, verificables y confiables.

El desarrollo de los ejercicios también permitió fortalecer una metodología basada en descomposición, modelado formal y codificación. Las soluciones no se limitaron a implementar programas, sino que incluyeron la definición de alfabetos, gramáticas, autómatas y mecanismos de validación, conectando constantemente la teoría con la práctica.

En conjunto, la intención del libro no fue solo presentar definiciones, sino promover una manera estructurada de analizar problemas, donde la abstracción y la formalización sirvan como herramientas para diseñar mejores soluciones.

Finalmente, la invitación es a continuar explorando áreas como compiladores, lenguajes de programación y teoría de la computación. En todas ellas, los conceptos estudiados siguen siendo una base esencial para comprender cómo funcionan los sistemas computacionales modernos.

Bibliografía

[Aho et al., 2006] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, 2 edition.

[Backus et al., 1960] Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., van Wijngaarden, A., and Woodger, M. (1960). Report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299–314.

[Bar-Hillel et al., 1961] Bar-Hillel, Y., Perles, M., and Shamir, E. (1961). On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14(2):143–172.

[Barendregt, 1984] Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, revised edition.

[Bray, 2017] Bray, T. (2017). The javascript object notation (json) data interchange format. RFC 8259, IETF. `https://www.rfc-editor.org/rfc/rfc8259`. [Online; consultado Enero-2025].

[Brzozowski, 1964] Brzozowski, J. A. (1964). Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494.

[Chomsky, 1956] Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124.

[Chomsky, 1959] Chomsky, N. (1959). On certain formal properties of grammars. *Information and Control*, 2(2):137–167.

[Chomsky, 1962] Chomsky, N. (1962). Context-free grammars and pushdown storage. In *Quarterly Progress Report No. 65*, pages 187–194. MIT Research Laboratory of Electronics, Cambridge, MA.

[Church, 1936] Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363.

[Church, 1941] Church, A. (1941). *The Calculi of Lambda-Conversion*. Number 6 in Annals of Mathematics Studies. Princeton University Press, Princeton, NJ.

[CreativeCommons, 2025] CreativeCommons (2025). Creative commons - attribution-noncommercial-noderivatives 4.0 international. `http://creativecommons.org/licenses/by-nc-nd/4.0/`. [Online; consultado Enero-2025].

[Earley, 1970] Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

[GNU, 2007] GNU (2007). Gnu general public license. `https://www.gnu.org/licenses/gpl-3.0.html`. [Online; consultado Enero-2025].

[Hennie and Stearns, 1966] Hennie, F. C. and Stearns, R. E. (1966). Two-tape simulation of multitape turing machines. *Journal of the ACM*, 13(4):533–546.

[Hilbert and Ackermann, 1928] Hilbert, D. and Ackermann, W. (1928). *Grundzüge der theoretischen Logik*. Springer, Berlin.

[Hopcroft, 1971] Hopcroft, J. E. (1971). An $n \log n$ algorithm for minimizing states in a finite automaton. *Theory of Machines and Computations*, pages 189–196.

[Hopcroft et al., 2006] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Boston, MA, 3 edition.

[Johnson, 1975] Johnson, S. C. (1975). Yacc: Yet another compiler-compiler. Technical Report Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ.

[Kasami, 1966] Kasami, T. (1966). An efficient recognition and syntax-analysis algorithm for context-free languages. *Scientific Report AFCRL-65-758*.

[Kleene, 1936] Kleene, S. C. (1936). General recursive functions of natural numbers. *Mathematische Annalen*, 112(1):727–742.

[Kleene, 1956] Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In Shannon, C. E. and McCarthy, J., editors, *Automata Studies*, number 34 in Annals of Mathematics Studies, pages 3–41. Princeton University Press, Princeton, NJ.

[Knuth, 1965] Knuth, D. E. (1965). On the translation of languages from left to right. *Information and Control*, 8(6):607–639.

[Lesk and Schmidt, 1975] Lesk, M. E. and Schmidt, E. (1975). Lex—a lexical analyzer generator. Technical Report Computing Science Technical Report 39, Bell Laboratories, Murray Hill, NJ.

[Lin and Rodger, 2009] Lin, S.-C. and Rodger, S. H. (2009). *An Introduction to Formal Languages and Automata*. Jones and Bartlett Learning, Burlington, MA, 4 edition.

[McCulloch and Pitts, 1943] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5(4):115–133.

[Moore, 1956] Moore, E. F. (1956). Gedanken-experiments on sequential machines. In Shannon, C. E. and McCarthy, J., editors, *Automata Studies*, number 34 in Annals of Mathematics Studies, pages 129–153. Princeton University Press, Princeton, NJ.

[Myhill, 1957] Myhill, J. (1957). Finite automata and the representation of events. Technical Report WADD TR-57-624, Wright Air Development Center, Ohio.

[Nerode, 1958] Nerode, A. (1958). Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544.

[Oettinger, 1961] Oettinger, A. G. (1961). Automatic syntactic analysis and the pushdown store. *Proceedings of Symposia in Applied Mathematics*, 12:104–129.

[Parikh, 1966] Parikh, R. J. (1966). On context-free languages. *Journal of the ACM*, 13(4):570–581.

[Post, 1936] Post, E. L. (1936). Finite combinatory processes—formulation 1. *The Journal of Symbolic Logic*, 1(3):103–105.

[Rabin and Scott, 1959] Rabin, M. O. and Scott, D. (1959). Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125.

[Resnick, 2008] Resnick, P. (2008). Internet message format. RFC 5322, IETF. https://www.rfc-editor.org/rfc/rfc5322. [Online; consultado Enero-2025].

[Rice, 1953] Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366.

[Rosen, 2019] Rosen, K. H. (2019). *Discrete Mathematics and Its Applications*. McGraw-Hill, New York, NY, 8 edition.

[Roy, 2007] Roy, P. V. (2007). Programming paradigms for dummies: What every programmer should know. https://www.dmi.unict.it/barba/PROG-LANG/PROGRAMMI-TESTI/READING-MATERIAL/VanRoyChapter.pdf. [Online; consultado Enero-2025].

[Sipser, 2012] Sipser, M. (2012). *Introduction to the Theory of Computation*. Cengage Learning, Boston, MA, 3 edition.

[Thompson, 1968] Thompson, K. (1968). Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422.

[Turing, 1936] Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265.

[Turing, 1937] Turing, A. M. (1937). Computability and λ-definability. *The Journal of Symbolic Logic*, 2(4):153–163.

[Wing, 2006] Wing, J. M. (2006). Computational thinking. https://www.cs.cmu.edu/~15110-s13/Wing06-ct.pdf. [Online; consultado Enero-2025].

[Wing, 2010] Wing, J. M. (2010). Computational thinking: What and why? https://www.cs.cmu.edu/%7ECompThink/resources/TheLinkWing.pdf. [Online; consultado Enero-2025].

[Wing, 2024] Wing, J. M. (2024). Jeannette wing. https://en.wikipedia.org/wiki/Jeannette_Wing. [Online; consultado Enero-2025].

[Wirth, 1977] Wirth, N. (1977). What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823.

[World Wide Web Consortium, 2008] World Wide Web Consortium (2008). Extensible markup language (xml) 1.0 (fifth edition). W3C Recommendation. `https://www.w3.org/TR/xml/`. [Online; consultado Enero-2025].

[Younger, 1967] Younger, D. H. (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208.

Índice alfabético