Informática teórica. Elementos propedéuticos. Raúl Gómez Marín y Andrés Sicard Ramírez, publicado por el Fondo Editorial de la Universidad EAFIT, 2002.

Nuestro texto está agotado. La versión que usted está leyendo corresponde a una reimpresión para la *Web* en la cual hemos realizado algunas correcciones y algunos cambios en la diagramación.

Agradecemos a nuestro colega René Alejandro Londoño Cano por sus observaciones y correcciones a una versión preliminar de esta reimpresión digital.

Si el lector desea realizar algún comentario (sugerencia, corrección, etc.) con relación a los contenidos del texto, lo puede hacer al correo electrónico asr@eafit.edu.co.

Última actualización: 8 de febrero de 2025

Correcciones:

Pág. 204: El problema del isomorfismo de grafos no es un problema NP-completo (es un problema NPI). Por lo tanto, lo afirmado por el teorema 6.60 es incorrecto.

A Lisa y Andrea, por un tiempo que era nuestro

Índice general

Índice de figuras

Índice de tablas

Prefacio

Si escribir un texto, cualquiera sea éste, es una faena difícil, escribirlo a varias voces todavía lo es más. Con todo, la dificultad es dínamo de reto, de provocación, de osadía incluso, máxime si se trata de dos profesores cuya manera de ocuparse del saber no es extraña a una búsqueda permanente de momentos o encuentros afortunados que propicien el compartir saberes y construir perspectivas confluyentes de trabajo, de suerte que éstas emanen del tramado de sus experiencias didácticas, lógicas, teóricas y humanas. La obra que aquí ofrecemos es el producto de un haz de confluencias, realizado a partir de encuentros afortunados que nos han permitido compartir nuestros conocimientos y experimentaciones como profesores de los diversos cursos del área de Matemáticas Especiales, cursos que hemos impartido en la Facultad de Ingeniería de Sistemas de la Universidad EAFIT, cuya sede se encuentra en la ciudad de Medellín, Colombia. Es así como nos hemos dado a la tarea de darle un cierto corpus didáctico y teórico a nuestras experimentaciones en este campo específico.

En efecto, no sin cierto rigor y esmero en la formalización, hemos querido reunir diversos temas que se han desarrollado en varios campos del conocimiento, temas que hoy se agrupan en lo que se podría llamar Informática Teórica. Anotemos además, que buena parte de los distintos "objetos" que definen el corpus teórico de esta obra emergen del oficio del saber lógico-matemático. Y en lo que a nosotros hace relación, aquéllos se yerguen esencialmente de una arraigada inquietud de saber, compartir lo sabido, y reconocer que todo saber sabido es siempre precario, inquietud que nosotros consideramos de orden ético-estético. Ello, primero que todo, porque esta inquietud está en el núcleo de lo humano y debe ser el "motor inmóvil" de nuestra manera de existir y, en segundo lugar, porque esta inquietud y en parte estos "objetos", no han cesado de trabajar en nosotros y hacernos trabajar, marcando de este modo la que mal que bien podríamos llamar "dinámica del curso" de Matemáticas Especiales III, de la citada facultad.

No es del todo fácil develar los propósitos de un autor. Cuando se escribe, cabe la pregunta, ¿qué se pone realmente en disposición y a disposición del otro? Nosotros no sabríamos arriesgar una respuesta. No obstante, digamos que creemos poner en disposición un cierto corpus de nociones —cuya filiación marcamos meticulosamente—, un cierto tramado, analítico y a la vez sintético, constituido por objetos teóricos y demostraciones hechas muy a nuestra manera, así como objetos y consideraciones que forman parte del quehacer

formativo de nuestros estudiantes y profesores (y de manera muy específica de aquellos que transitan por las Facultades de Ingeniería de Sistemas de algunas universidades de nuestro entorno). Así contorneada la respuesta, no sobra destacar una variable importante en nuestras pretensiones. Queremos contribuir a llenar un cierto vacío bibliográfico en nuestro medio. Ciertamente existe una buena bibliografía, pero sus enfoques y contenidos están esencialmente orientados a cursos de postgrado. Pocos textos hay en el ámbito universitario que tengan una orientación hacia una propedéutica acorde con las condiciones y pretensiones de nuestros pregrados. En realidad son escasos los textos que tengan una orientación hacia estos niveles y que reúnan y traten adecuadamente y con cierta didaxis los temas aquí presentados. Cabría destacar en esta especie de vacío el texto de Lógica y Calculabilidad, escrito por el profesor colombiano Xavier Caicedo Ferrer; los demás, referenciados en nuestras notas bibliográficas, sin dejar de ser importantes, mantienen una dimensión bastante intuitiva y poco formalizada para lo que en nuestro medio, al menos en nuestra visión de las cosas, se busca con este género de cursos de pregrado.

Las referencias contextuales anteriores no pretenden escamotear los valores de los otros textos que circulan en nuestro medio, ni mucho menos abrogarnos una cuota de cabal originalidad. No, lejos estamos de tal intención. Los puntos de vista que aquí desarrollamos están bastante difundidos, como puede constatarse en el esfuerzo que hemos realizado a lo largo de este texto, por señalar la fuente específica desde la cual asumimos una determinada mirada, definición, demostración o ejemplo. Allí, en ese juego de despeje, de reconocimiento y asunción queda nuestro esfuerzo de originalidad. Ciertamente hay un poco de originalidad, pero no queremos entrar en ese juego de imaginarios; más bien, respetuosamente dejamos este asunto a cargo de los lectores avisados en estos campos. Lo que en rigor el texto busca, no sin deliberada intención, es generar un trazado entre intuición y formalización, así, como contornear o solamente diagramatizar ciertos temas; diagramatizar en el sentido de introducir iconos de relación que permitan, en el caso del lector, engendrar una visión amplia y comprensiva del tema, y ello no sin ciertas precisiones de detalle, técnicas formales, cuando así lo amerite la perspectiva que deseamos proponer. Con esta última alusión también queremos aclarar que ciertos temas; quizá la mayoría no pretende ir más allá de los límites que el texto configura, dada justamente su pretensión de ampliar los campos de formación en pregrado a la vez que otorgarles un cierto límite. Así las cosas, presentamos un texto que recoge y ordena metódicamente ciertos temas que habitualmente no se consideran o integran en los currículos de Ingeniería de Sistemas en nuestro medio; y otros temas que son formulados clásica e intuitivamente en otros textos, se desarrollan en el nuestro en función de una metodología que con un cierto grado de síntesis y de desarrollo, creemos, abre vías de lecturas que acompañarán el proceso de formación y desarrollo (más científico y riguroso) de un Ingeniero de Sistemas.

Finalmente, conviene afirmar que al avanzar en la elaboración de este texto, no cesamos de hallar problemas y vías alternas de evolución en su producción. Al fin de cuentas, nos percatamos de que el sentido que jalona una escritura no debe estar alejado de poder pensarla como una vía múltiple, es decir, como una vía que, al vislumbrar el fin, no puede

menos de pretextarlo. Nuestra sentida apuesta es, entonces, por un convidar al lector a "ver" en esta obra un trabajo que no cesa de rehacerse, esto es, a ver en ella una obra abierta. Abierta en diversos sentidos y grados. Abierta en ella misma, abierta al lector, campo abierto de problemas que ella contiene y que, quizás, contribuye a generar. Por último, para cerrar con un mandato ético de nuestro proceder, queremos seȟalar que toda argumentación, imprecisión, problemas inherentes a la producción de este libro y que los cuales permanecen sin resolver, son enteramente nuestra responsabilidad. Como diría el pensador chileno Humberto Maturana: "Nosotros nos hacemos cargo de lo dicho, en la esperanza de que el lector se haga cargo del sentido construido."

Para mantener una comunicación constante con nuestros lectores—con base en los medios informáticos actuales—el texto cuenta con una página *web*, la cual puede ser accesada desde la dirección `www1.eafit.edu.co/asicard`. Esta página mantendrá información actualizada respecto a correcciones, modificaciones o actualizaciones al texto. Por otra parte, si el lector desea realizar algún comentario (sugerencia, corrección, etc.) con relación a los contenidos del texto, lo puede hacer al correo electrónico `asicard@eafit.edu.co`.

No queremos cerrar este prefacio sin ofrecer algunas notas de agradecimiento. A nuestros colegas, Juan Carlos Agudelo Agudelo, Orlando García Jaimes y Hugo Guarín Vásquez, quienes al seguir una versión preliminar de este texto en sus cursos, formularon algunas sugerencias al mismo. A nuestro estudiante Carlos Andrés Ardila por la elaboración del ejemplo 2.57 (pág. 87). A nuestros revisores por sus correcciones y sugerencias. Al fondo editorial de la Universidad EAFIT y a su directora Leticia Bernal. A nuestros estudiantes, quienes, semestre a semestre, por espacio de seis años, nos acompañaron en esta labor, bien en calidad de escuchas concernidos, bien en calidad de escuchas silenciosos. A la Universidad EAFIT, de una forma u otra nuestra alma mater, por concedernos el tiempo y el espacio requeridos para sacar adelante este pequeño sueño hoy vuelto realidad. Y, de contera, a otras presencias, aquí no nombradas, pero cuyos nombres sabemos con certeza que nos acompañan en nuestra memoria.

Los autores.
Universidad EAFIT, Medellín
15 de febrero del 2001

Capítulo 0

Introducción

A continuación presentamos un texto que contiene una cierta estructura didáctica y teórica. Ciertamente hay cambios en el orden canónico usual. No obstante, hay que decir que la experiencia en la enseñanza de estos temas nos ha revelado lo fructífero de tal ordenamiento.

Hemos organizado ciertos aspectos formales básicos para la computación en una estructura que contempla seis capítulos. En el primero nos ocuparemos de los elementos que consideramos básicos para una compresión de la teoría de las máquinas de Turing. El segundo, por su parte, lo consagraremos al estudio de la teoría de las funciones parciales recursivas. En el tercero nos adentraremos en el estudio de aspectos básicos de las teorías de lenguajes y gramáticas formales. El cuarto, en íntima conexión con el anterior, nos permitirá acercanos a la teoría de los autómatas de estado finito. El quinto, lo consagremos al estudio de la teoría de los autómatas de pila. Finalmente, cerraremos el texto con un capítulo consagrado a presentar los elementos básicos de la teoría de la complejidad algorítmica. La figura 1 ilustra claramente la estructura del texto, esto es, nos representa las relaciones y depencias establecidas entre los contenidos del texto.

A continuación presentamos algunas consideraciones que contribuyan a revelar la importancia y pertinencia de estos temas en la formación de una buena parte de ingenieros y tecnólogos, la misma que realizaremos para cada uno de los capítulos del texto.

Capítulo 1: Computabilidad

La teoría de la computabilidad clásica, tal como la conocemos hoy en día, fue establecida en la década de los 30's por los trabajos fundacionales realizados por Kurt Gödel, Jacques Herbrand y Stephen Kleene en funciones recursivas, Alonso Church y Stephen Kleene en funciones λ-definibles, y Alan Turing y Emil Post en funciones computables. Entre estos trabajos se destaca el realizado por Turing, debido a que no es necesario realizar un gran esfuerzo para observar la compenetración existente entre, por un lado, la idea intuitiva de

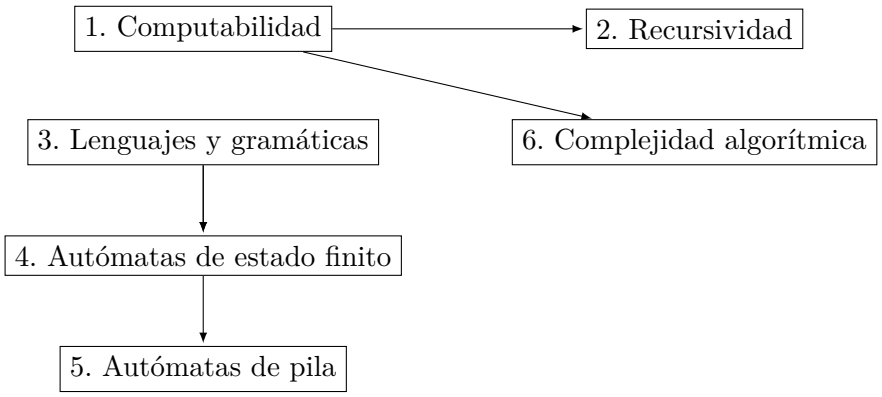


Figura 1: Relaciones y dependencia entre capítulos.

procedimiento calculable (es decir, aquel procedimiento que se pueda llevar a buen término siguiendo un conjunto de pasos establecidos) y, por otro lado, la formalización realizada por Turing para dar cuenta de esta clase de procedimientos, justamente lo que hoy denominamos máquinas de Turing.

El capítulo 1 aborda el estudio de la teoría de la computabilidad desde la perspectiva de las máquinas de Turing, obteniendo así la "teoría de la Turing-computabilidad". Esta teoría, además de aportar conceptos muy importantes al estudio de las propiedades metamáticas de los sistemas formales, se constituyó (al igual que otras teorías) de la mano del propio Turing (y por supuesto de la mano de muchos otros) en la fundamentación formal de la Ciencias de la Computación. Inicialmente identificamos la noción de algoritmo con la noción de máquina de Turing.

Una vez establecida una definición formal de procedimiento computable como el que es computable por una máquina de Turing, podemos clasificar los objetos (números, funciones o procesos) como computables o no computables. Dentro de estos procesos no computables se destaca el problema de la parada de una máquina de Turing, el cual consiste en determinar si una máquina de Turing se detendrá o no con una entrada seleccionada de un conjunto posible de ellas. La insolubilidad del problema de la parada es fraseable, en términos de la ciencias de la computación, afirmando la imposiblidad de construir un depurador de programas que detecte si un programa podría o no entrar en un ciclo infinito.

Debido a que el conjunto de instrucciones de una máquina de Turing está compuesto por instrucciones muy simples, la "programación" de una máquina de Turing es usualmente un proceso muy tedioso, pudiéndose incluso comparar con la programación en lenguaje ensamblador. Para soslayar esta dificultad, Turing estableció el concepto de m-función. Este concepto no es más que el predecesor formal de los llamados macros en los lenguajes ensambladores o de los llamados procedimientos en los lenguajes de programación de mayor nivel.

La noción de máquina de Turing universal es la noción formal subyacente a un computador, en el sentido que éste puede ser pensado como una máquina que ejecuta un algoritmo para ejecutar algoritmos. La máquina de Turing universal se constituye, pues, en el modelo formal de nuestras actuales máquinas de cómputo; de allí que se afirme que cualquier computador es una máquina de Turing con las limitaciones físicas que aquélla no tiene, limitaciones físicas que hacen referencia a la capacidad de memoria no acotada de una máquina de Turing universal.

En fin, en este capítulo iniciaremos un recorrido por la teoría de la computabilidad, recorrido que no dejará de gozara de un cierto nivel de formalización y abstracción.

Capítulo 2: Recursividad

Una vez hayamos terminado el recorrido con el concepto de computabilidad (en el capítulo 1), retomaremos este concepto en el capítulo 2, pero esta vez no en el sentido de Turing, sino en el sentido de Gödel, Herbrand y Kleene. El concepto de computabilidad es una idea bastante importante. ¿Por qué? Justamente porque existen operaciones, incluso aritméticas, que no son computables. En este capítulo probaremos por ejemplo, que existen funciones numéricas que no son computables.

El concepto de computabilidad es realmente un concepto abstracto, es decir, va más allá de cualquier realización concreta, en el sentido de que existen diversas formas de abordar la idea de computabilidad (λ-cálculo, funciones recursivas parciales, máquinas de Turing, etc.)

En este capítulo abordaremos la teoría de la computabilidad desde la perspectiva de la teoría de las funciones recursivas parciales, justamente para trabajar el concepto de computabilidad de una manera más matemática, lo cual nos dará cierta independencia de formalismos concretos y nos permitirá, mediante la teoría de conjuntos, obtener ciertos resultados sorprendentes sobre lo computable y lo no computable. Mostraremos, por ejemplo, la equivalencia entre las funciones recursivas parciales y las funciones Turing-computables (lo cual nos permitirá otra formalización de la noción de algoritmo); probaremos igualmente que el conjunto de las funciones computables es del orden infinito enumerable, mientras que el conjunto de las no computables es del orden infinito no enumerable. Una pregunta o problema nos hará ciertamente trabajar bastante: se trata de saber si todo lo computable en sentido intuitivo es computable en el sentido formal, esto es, en el sentido de una función recursiva parcial. Esta es, justamente, una de las formas como podemos expresar la tesis de Church-Turing.

Es importante resaltar que, desde la perpectiva de las ciencias de la computación, la teoría de las funciones recursivas se constituye en la teoría formal subyacente al paradigma de programación denominado “programación funcional”, así lenguajes de programación tales como *LISP* y *Mathematica* obtienen sus fundamentos formales de esta teoría.

En fin, en este capítulo haremos un recorrido axiomático y constructivo, de un cierto rigor, que nos ofrecerá de nuevo posibilidades de adentrarnos aún más en el fascinante

universo de los problemas de la computabilidad.

Capítulo 3: Lenguajes y gramáticas

Los lenguajes admiten una clasificación en lenguajes formales y lenguajes naturales. Una diferencia importante existente entre ellos, estriba en que en los primeros es posible desarrollar un trabajo en sintaxis y semántica mucho más riguroso que en los segundos. Ejemplos de lenguajes formales lo constituyen la lógica, la matemática y los lenguajes de programación.

En los lenguajes formales infinitos tendremos uno de los primeros aspectos importantes para considerar; se trata de la posibilidad de generación de un lenguaje por un conjunto finito de reglas, denominadas "reglas de producción". Este conjunto finito de reglas de producción constituye lo que se conoce como la gramática del lenguaje. Noam Chomsky estableció una taxonomía para aquellos lenguajes formales susceptibles de ser generados por una gramática, con base en una clasificación de las reglas de producción. Trabajos posteriores establecieron relaciones del tipo lenguaje-reconocedor, asignando a cada tipo de lenguaje un tipo de reconocedor. Así dichas las cosas, dedicaremos este capítulo al desarrollo de la teoría de los lenguajes y de las gramáticas formales.

Uno de los aspectos más importantes de este capítulo, reside en el hecho de la clase de lenguajes formales denominados "lenguajes independientes del contexto". Esta clase de lenguajes es de suma importancia en la ciencias de la computación, debido a que justamente, gran parte de los lenguajes de programación actuales como *C++*, *Java*, *Fortran*, entre otros, pertenecen a esta clase y por lo tanto comparten propiedades formales, la más importante de la cuales es quizás, que son reconocidos por máquinas teóricas denominadas "autómatas de pila". Estos autómatas serán estudiados en el capítulo 5.

Una vez conocida la grámatica de un lenguaje es posible estudiar algunas propiedades y características de la misma, y por ende, algunas propiedades y características del lenguaje genererado por ésta. Dentro de estas propiedades se destaca la propiedad de ambigüedad o no del lenguaje, propiedad que se revela de gran importancia debido a sus efectos nocivos en la manipulación autómatica del mismo, es decir, en la generación, interpretación o el reconocimiento del mismo.

En fin, este capítulo tiene como objetivo fundamental el estudio formal y riguroso de los lenguajes formales a partir de sus generadores, o sea, a partir de las gramáticas formales.

Capítulo 4: Autómatas de estado finito

La palabra 'autómata' tiene diversos sentidos. Puede evocar, por ejemplo, un dispositivo que simula los movimientos de un ser vivo. En su referencia a la informática, lo esencial de esta palabra consiste en permitirnos pensar en una simulación de los procesos para

manipular información, cuyo paradigma típico es, obviamente, el computador, en la medida que éste puede ser visto como una máquina que manipula símbolos.

Este capítulo lo consagraremos a la teoría de autómatas de estado finito, también llamada teoría algebraica de máquinas. Si entendemos por un autómata una máquina secuencial (esto es, que opera sobre secuencias de símbolos), entonces el objetivo aquí es formalizar esta idea mediante elementos de la teoría de conjuntos y del álgebra abstracta.

La teoría de autómatas de estado finito que vamos a desarrollar nos dota de conocimientos y métodos para los problemas de análisis y de síntesis de las máquinas secuenciales. Un aspecto muy importante que desarrollaremos en este capítulo es el concerniente al problema de la decidibilidad, via autómatas de estado finito, de un lenguaje y sus conexiones con las gramáticas formales. Puede pensarse, por ejemplo, en un autómata reconocedor para un cierto lenguaje. Pero esto último no siempre es posible garantizarlo para todo lenguaje formal, por ello en este capítulo estudiaremos bajo qué condiciones podemos garantizar el que un lenguaje dado sea aceptado por un reconocedor finito.

Aunque sabemos que desde la teoría de la computabilidad los autómatas de estado finito son modelos más limitados que las máquinas de Turing o las funciones recursivas, su importancia reside en que desde la perspectiva de las ciencias de la computación estos autómatas son usuados para modelar procesos cuya complejidad admite un modelo más simple que el modelo general de computabilidad. Ejemplos de tales procesos los hallamos en la especificación de ciertas partes de los lenguajes de programación y en la modelación de ciertas subetapas de algunas de las etapas de Ingeniería del *software*.

En fin, este capítulo tiene como objetivo fundamental presentar los elementos básicos de la teoría de autómatas de estado finito, desarrollando algunos ejemplos que nos permitan ver su utilidad en campos diversos, especialemte los relacionados con la informática y la computabilidad.

Capítulo 5: Autómatas de pila

Una clase de máquinas formales más potentes que los autómatas de estado finito, pero menos potente que las máquinas de Turing, la consituye la clase de los autómatas de pila. Intuitivamente expresado, podemos decir que esta diferencia está sustentada en la capacidad de memoria de las tres máquinas formales mencionadas; así, una máquina de Turing tiene memoria no acotada y sin restricciones de acceso, un autómata de pila también tiene memoria no acotada pero el acceso a ella es restringido a las manipulaciones que se puedan realizar sobre la estructura de datos denominada “pila”, y un autómata de estado finito tiene memoria finita.

Como mencionamos en la sección correspondiente al capítulo 3, los autómatas de pila son los reconocedores de los lenguajes independientes del contexto (y de este tipo son la mayoría de los lenguajes de programación actuales). Por lo tanto, no es de extrañar que los autómatas de pila sean uno de los principales elementos formales subyacente al área de

las ciencias de la computación denominada "teoría de la compilación". Es más, usualmente cuando compilamos un programa estamos ejecutando una implementación de un autómata de pila diseñado para reconocer palabras (algoritmos en este caso) que pertencen al lenguaje de programación en cuestión.

De nuevo, un aspecto muy importante que desarrollaremos en este capítulo es el relacionado con el problema de la decidibilidad, en este caso desde la perspectiva de los autómatas de pila, estableciendo qué lenguajes pueden o no ser reconocidos por estos autómatas.

En fin, este capítulo será una primera aproximación formal a la teoría de los autómatas de pila, teoría que como mencionamos está en los cimientos de nuestros actuales compiladores.

Capítulo 6: Complejidad algorítmica

La teoría de la computabilidad establece qué procesos pueden o no ser computados. Para ellos la teoría de la complejidad algorítmica establece clasificaciones de acuerdo con los recursos necesarios (tiempo y memoria) para computar los procesos computables. Este capítulo se consagrará entonces al desarrollo de los elementos básicos de la teoría de la complejidad algorítmica.

Para las ciencias de la computación no sólo es necesario saber si un problema es o no computable, sino que, además, es necesario establecer la factibilidad de ejecutar dicha computación (es decir, la posibilidad real de su ejecución), pues si la computación requiere un período de tiempo excesivo (quizás años o siglos) o si requiere una cantidad de memoria imposible de suministrar (desde un punto de vista práctico), la computación no se puede realizar. De otra parte, el estudio y desarrollo de algoritmos cada vez mejores está sustentado, entre otros aspectos, en la reducción de los recursos requeridos en su ejecución.

Una clasificación establecida *de facto*, para los diferentes problemas computables, es la que clasifica los problemas en tratables e intratables. Intuitivamente expresado esto último, problemas tratables, en cuanto a algún recurso, son problemas que tienen una demanda del recurso en cuestión, expresable como una función polinómica del tamaño de la entrada al problema, y problemas intratables son aquellos para los cuales esta función es exponencial.

Un aspecto de fundamental importancia para la teoría de la complejidad algorítmica lo constituye el modo de computación (determinista o no determinista). Mientras que en la teorías de la Turing-computabilidad, de los autómatas de estado finito y de los autómatas de pila no existe diferencia fundamental en cuanto a si ellos son o no deterministas, desde la perspectiva de la complejidad algorítmica el operar en modo determinista o el operar en modo no determinista puede establecer la diferencia entre que un problema sea tratable (complejidad polinómica) o no tratable (complejidad exponencial). De hecho, uno de los problemas matemáticos actuales de mayor trascedencia consiste en decidir si para problemas de complejidad temporal polinómica existe o no diferencia en cuanto al modo de computación empleado.

En fin, en este capítulo haremos una introducción a esa área de la informática teórica que goza de dos características importantes: por una lado, la complejidad algorítmica es uno de los campos de investigación de mayor dinámica desde el punto de vista teórico, y por otro lado, es uno de los campos teóricos con mayor potencial para ofrecer posibilidad de obtener resultados aplicados.

Capítulo 1

Computabilidad

En la actualidad contamos con una definición muy precisa del concepto de *algoritmo*, palabra que procede del nombre del matemático persa del siglo IX, Abu Ja'far Mohammed ibd Mâsa al-Khowâriz; pero la situación no era la misma a principios de este siglo.

Como consecuencia del descubrimiento (a finales del siglo pasado y a comienzos de éste) de las paradojas o antinomias en la teoría de conjuntos y la situación que esto generó (situación muy importante dado el papel de teoría base que desempeña la teoría de conjuntos en la matemática), se abordó un problema más amplio que comprendía la fundamentación de la matemática y la lógica. En 1900, en el Congreso Internacional de Matemáticos realizado en París, Francia, Hilbert propuso 23 problemas que deberían marcar el desarrollo de las matemáticas en los años siguientes. El décimo de estos problemas era: ¿es posible encontrar un procedimiento mecánico para calcular la solución de una clase particular de ecuaciones (ecuaciones diofánticas, es decir, ecuaciones algebraicas con una o más incógnitas, de coeficientes enteros y de las que interesa únicamente las soluciones enteras)? El mismo Hilbert generalizó el problema y lo presentó en 1928 en el Congreso Internacional de Matemáticos realizado en Bolonia, Italia; esta generalización es conocida actualmente como *el problema de la decisión (Entscheidungsproblem)*, (algunos autores mencionan que el problema de la decisión es una generalización del segundo problema planteado por Hilbert en el congreso de 1900; este problema consistía en demostrar que los axiomas de la aritmética eran consistentes entre sí) y su enunciado es: ¿existe algún método o procedimiento mecánico, que pueda *en principio*, decidir (resolver) *todas* las preguntas (problemas) matemáticos? Expresado en términos más "formalistas" el problema de la decisión consistía en encontrar un *método efectivo general* para determinar si un fórmula era o no verdadera en un sistema formal dado.

Alan Mathison Turing demostró el carácter irresoluble del problema de la decisión, por un camino diferente al empleado por Kurt Gödel, presentado en su famoso teorema sobre la incompletitud de los sistemas formales. Turing sintió la necesidad de contar con una noción más precisa del concepto de *procedimiento efectivo de decisión* para el cual concibió

la noción de *computabilidad*. Creemos que la idea de que el procedimiento de decisión es un procedimiento mecánico, es decir, puede ser ejecutado por un ente sin inteligencia, siempre y cuando tenga acceso a las instrucciones indicadas por él (éstas sí creadas con inteligencia) llevó a Turing a concebir la idea de una máquina abstracta para describir este concepto. Esta máquina abstracta se conoce actualmente como la *máquina de Turing* y corresponde a la noción formal del concepto de algoritmo. Turing replanteó el problema de la decisión en términos de sus máquinas y demostró que este problema no tenía solución.

1.1. Descripción informal de la máquina de Turing

Alan Mathison Turing construyó una máquina abstracta, o si se prefiere, una máquina matemática conocida en nuestros días como *máquina de Turing*, en la cual capturó la noción de *algoritmo*.

Un algoritmo recibe algunos datos de entrada, los procesa y genera unos resultados. Usualmente, cuando ejecutamos un algoritmo a mano, utilizamos el papel como mecanismo de entrada-salida del algoritmo. Veamos inicialmente cuál es el mecanismo de entrada-salida utilizado por la máquina de Turing.

La máquina de Turing utiliza una *cinta infinita* como mecanismo de entrada-salida. La cinta está dividida en celdas las cuales pueden contener sólo un *símbolo* de un *alfabeto finito de símbolos* indivisibles de la máquina (el símbolo vacío se representa por $\square$ y por convención hace parte de este alfabeto). En lenguaje técnico decimos que la cinta es una cinta unidimensional bi-infinita (infinita en ambos extremos).

En cada instante *discreto* de tiempo, la máquina, "observa" una celda de la cinta y tiene la capacidad de leer-escribir un símbolo sobre ésta. Adicionalmente la máquina tiene la capacidad de *desplazarse* sobre la cinta, es decir, se puede desplazar una celda a la derecha, o una celda a la izquierda, o no desplazarse con respecto a su posición actual. Aunque hemos mencionado que es la máquina la que se desplaza sobre la cinta, no existe tampoco ningún inconveniente, desde el punto de vista conceptual, el considerar que es la cinta la que se desplaza sobre la máquina. Nuestra convención estipula que es la máquina la que se desplaza sobre la cinta.

Además del *alfabeto* (conjunto finito de símbolos indivisibles), la máquina tiene también un *conjunto finito de estados*, los cuales representan, como su nombre lo indica, el estado en el que está la máquina en algún instante discreto de tiempo.

Definimos una *situación* de la máquina, como una dupla formada por un *estado* (perteneciente al conjunto finito de estados) y un *símbolo* (perteneciente al alfabeto). Para cada instante de tiempo, la *situación actual* de la máquina está determinada por el *estado actual* en el cual se encuentra y por el *símbolo* que está en la celda, es decir, aquélla sobre la cual la máquina está situada (la acción de leer el símbolo de la celda es automática). Es necesario entonces, antes de comenzar la ejecución de la máquina, definir su *situación inicial* (estado inicial, posición inicial sobre la cinta).

El elemento más importante de una máquina de Turing lo constituye el *conjunto finito de instrucciones*. Este conjunto representa el comportamiento de la máquina. Cada *instrucción* está compuesta por dos partes; la primera indica *cuándo* se debe ejecutar la instrucción, es decir, en qué situación (estado, símbolo); la segunda parte indica *qué hace* la instrucción, lo cual consiste en: escribir un símbolo (en la posición actual), ejecutar un moviento sobre la cinta y colocar la máquina en un nuevo estado. En otras palabras, una instrucción está compuesta por cinco elementos, a saber: *estado-actual, símbolo-leer, símbolo-escribir, movimiento, estado-siguiente*; donde los dos primeros elementos indican *cuándo* y los tres restantes *qué hacer*.

Veamos cómo opera una máquina de Turing: La máquina busca en su conjunto de instrucciones una instrucción que concuerde (la primera parte) con la situación actual (estado actual, símbolo actual) de la máquina. Si la encuentra, la ejecuta, escribiendo el símbolo, realizando un movimiento y pasando al estado indicado por la instrucción hallada. En este momento la máquina se encuentra en una nueva situación actual y repite el proceso. Si por el contrario la máquina no encuentra una instrucción que concuerde con la situación actual, la máquina se detiene y se da por finalizada su ejecución. Éste es un contexto adecuado para la siguiente pregunta: ¿Es posible que la máquina tenga más de una instrucción para una situación en particular? De acuerdo con la respuesta obtenemos dos clases diferentes de máquina de Turing, a saber: Las *máquinas de Turing determinísticas*, las cuales no permiten definir más de una instrucción para una situaciónen particular, y las *máquinas de Turing no determinísticas*, que sí permiten tal cosa.

1.2. Descripción formal de la máquina de Turing

Definimos formalmente una máquina de Turing determinista mediante la estructura matemática

$$\mathsf{MT} = \langle Q, \Sigma, M, I \rangle,$$

donde:

(i) $Q = \{q_0, q_1, q_2, \ldots, q_n\}$ es un conjunto finito de estados de la máquina ($Q \neq \emptyset$).

(ii) $\Sigma = \{s_0, s_1, s_2, \ldots, s_m\}$ es un alfabeto o conjunto finito de símbolos de entrada-salida. Adoptamos por convención que $s_0 = \square$ (símbolo vacío) ($\Sigma - \{s_0\} \neq \emptyset$).

(iii) $M = \{L, R, N\}$ es el conjunto de movimientos (L: izquierda, R: derecha, N: no movimiento).

(iv) I es una función definida de un subconjunto $Q \times \Sigma$ en $\Sigma \times M \times Q$.

Se debe notar que la función I está definida sobre un subconjunto de $Q \times \Sigma$, puesto que no es necesario que exista una instrucción para cada una de las situaciones (estado, símbolo) teóricas (el conjunto formado por $Q \times \Sigma$) en las que puede estar la máquina. Además,

el concepto de función refleja la característica de las máquinas de Turing determinísticas. La función I también puede ser definida como un conjunto *finito* de instrucciones $I = \{i_0, i_1, i_2, \ldots, i_p\}$, donde cada i_j es una quíntupla de la forma: $q_m\ s_m\ s_n\ m\ q_n$, donde $q_m, q_n \in Q$; $s_m, s_n \in \Sigma$; $m \in M$.

No obstante la distinción establecida, nuestro trabajo en este capítulo hará referencia principalmente a máquinas de Turing deterministas; por la tanto, la palabra "deterministas" se omitirá desde este momento.

Es importante anotar que la información inicial que contiene la cinta, esto es, el estado inicial y la posición inicial de la máquina sobre ésta, será suministrada "informalmente" antes de comenzar la ejecución de la máquina.

Si una máquina de Turing MT se encuentra en la situación actual (q_m, s_m) y encuentra una instrucción $i_j : q_m, s_m, s_n, m, q_n$, entonces MT cambia s_m por s_n; realiza el movimiento indicado por m y pasa al estado q_n (puede ocurrir que $q_m = q_n$ o $s_m = s_n$); en un caso contrario, la máquina finaliza su ejecución.

Pasemos ahora a considerar algunos ejemplos de máquinas de Turing:

Ejemplo 1.1. Sea $\mathsf{MT} = \langle Q, \Sigma, M, I\rangle$ una máquina de Turing definida por:
$Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$ (recordamos que, por convención el símbolo vacío $\square$ pertenece al alfabeto), $I = \{i_0, i_1, i_2, i_3\}$ donde:

i_0:	q_0	$\square$	0	R	q_1
i_1:	q_1	$\square$	$\square$	R	q_2
i_2:	q_2	$\square$	1	R	q_3
i_3:	q_3	$\square$	$\square$	R	q_0

Antes de comenzar la ejecución de MT necesitamos definir la secuencia de símbolos iniciales en la cinta, el estado inicial en el cual se encuentra la máquina y la posición inicial de MT en dicha cinta. Inicialmente la cinta se encuentra vacía y la máquina está en alguna posición de la cinta (representada por la celda a la cual apunta el símbolo $\uparrow$); además, la máquina se encuentra en su estado inicial q_0. Al inicio tenemos:
estado actual: q_0

...	$\uparrow$					...

Comencemos la ejecución: la máquina busca en su conjunto de instrucciones una que comience por la situación actual $(q_0, \square)$ y encuentra la instrucción "$q_0\ \square\ 0\ R\ q_1$" (instrucción i_0), entonces, la máquina procede a cambiar $\square$ por 0, se mueve una posición a la derecha y pasa al estado q_1. Después de ejecutar la primera instrucción tenemos:
estado actual: q_1

...	0	$\uparrow$				...

Ahora la máquina está en la situación actual $(q_1, \square)$; como a esta situación le corresponde la instrucción "$q_1\ \square\ \square\ R\ q_2$" (instrucción i_1), entonces la máquina se mueve a la derecha y pasa al estado q_2 (porque la instrucción indica que cambie $\square$ por $\square$ lo que se traduce en no escribir nada). Ahora tenemos:
estado actual: q_2

...	0		↑			...

Como la situación actual $(q_2, \square)$ corresponde la instrucción "$q_2\ \square\ 1\ R\ q_3$" (instrucción i_2), entonces la máquina cambia $\square$ por 1, se mueve a la derecha y pasa al estado q_3. Ahora tenemos:
estado actual: q_3

...	0		1	↑		...

Ahora, para la situación actual $(q_3, \square)$ corresponde la instrucción "$q_3\ \square\ \square\ R\ q_0$" (instrucción i_3), la cual le indica a la máquina que se mueva a la derecha y pase al estado q_0, es decir:
estado actual: q_0

...	0		1		↑	...

En este momento la máquina se encuentra de nuevo en la situación actual $(q_0, \square)$ y necesariamente repite el ciclo indefinidamente.

La tabla 1.1 muestra el comportamiento de la máquina MT para la ejecución de los siete primeros "pasos".

Paso	Estado Actual	Instrucción										
0	q_0		...	↑								...
1	q_1	i_0	...	0	↑							...
2	q_2	i_1	...	0		↑						...
3	q_3	i_2	...	0		1	↑					...
4	q_0	i_3	...	0		1		↑				...
5	q_1	i_0	...	0		1		0	↑			...
6	q_2	i_1	...	0		1		0		↑		...
7	q_3	i_2	...	0		1		0		1	↑	...

Tabla 1.1: Simulación máquina MT.

Observación 1.2. Note que particularmente esta máquina no se detiene nunca. Además utiliza la convención propuesta por Turing de "trabajar" en celdas intercaladas, con el propósito de utilizar los espacios entre las celdas ocupadas para colocar "marcas" temporales que se necesiten durante la ejecución. Pero señalemos que no todas las máquinas tienen estas características.

Ejemplo 1.3. A continuación vamos a construir una máquina de Turing que calcula la suma de dos números naturales representados en el código "palitos".

Uno de los aspectos que se debe tener en cuenta en el diseño de una máquina de Turing es la forma como están representados en la cinta los datos de entrada. Supongamos que deseamos realizar la suma de 1 y 2.

Para codificar los números naturales $\mathbb{N}$ en el alfabeto de "palitos", estableceremos la siguiente convención:

$$\begin{aligned} 0 &= / \\ 1 &= // \\ 2 &= /// \\ 3 &= //// \\ &\vdots \\ n = /^{n+1} &= \underbrace{///\ldots/}_{n+1\ veces} \end{aligned}$$

Una representación de la diversas representaciones que podríamos tener viene dada por la tabla 1.2. En este caso los números que se van a sumar están separados por un espacio en blanco.

...	/	/		/	/	/	...

Tabla 1.2: Representación entrada de datos (1).

Otra representación viene dada por la tabla 1.3. En este caso los números que se van a sumar están separados por un signo '+'.

...	/	/	+	/	/	/	...

Tabla 1.3: Representación entrada de datos (2).

Otra posible representación viene dada por la tabla 1.4. En este caso los números que se van sumar están separados por n espacios en blanco.

...	/	/	...	/	/	/	...

Tabla 1.4: Representación entrada de datos (3).

Seleccionemos la primera representación (tabla 1.2) para construir nuestra máquina sumadora de palitos.

Otro de los aspectos que se debe considerar en el diseño de una máquina de Turing, es la forma o convención elegida para representar en la cinta la respuesta generada por la máquina.

Es este caso, vamos a suponer que la salida de la máquina está dada por el número de "palitos" en la cinta. Entonces para nuestro ejemplo de la suma de $1 + 2$ la salida estará representada como lo indica la tabla 1.5.

...	/	/	/	...

Tabla 1.5: Representación salida de los datos.

Precisemos ahora algunos detalles concernientes a la máquina en cuestión. Digamos de entrada que nuestra máquina está definida por $\mathsf{MT} = \langle Q, \Sigma, M, I\rangle$ donde, el conjunto de estados internos de la máquina es $Q = \{q_0, q_1, q_2, q_3, q_4, stop\}$.

Antes de continuar, hablemos un poco acerca de este estado *stop* en nuestra máquina. En la construcción de una gran parte de máquinas, se espera que éstas finalicen (a diferencia de nuestro primer ejemplo). Es necesario conocer si dicha finalización fue exitosa o no, es decir, si la máquina finalizó porque estaba programada para ello, o finalizó porque no encontró una instrucción para la situación actual en la que se encontraba. Para resolver esta incertidumbre se acostumbra dotar a la máquina de un estado de parada exitosa, normalmente llamado *stop*, el cual, cuando es alcanzado por la máquina, finaliza su ejecución (pues no existe ninguna instrucción que contemple a *stop* como su primer componente). Ciertamente, para el diseñador de la máquina esta es una parada exitosa. Observe el lector que el uso de esta convención no modifica para nada lo dicho hasta el momento. Hechas las anteriores consideraciones, culminemos el diseño de nuestra máquina.

Inicialmente la cinta contiene dos números naturales representados en el código "palitos", separados por un espacio en blanco. La máquina se encuentra situada sobre el primer palito de izquierda a derecha y el estado inicial es q_0.
$\Sigma = \{|\}$ (alfabeto de entrada-salida), $I = \{i_0, i_1, i_2, i_3, i_4, i_5, i_6\}$ (conjunto de instrucciones), donde:

i_0:	q_0	|	|	R	q_0	;recorre el primer número hasta el espacio en blanco
i_1:	q_0	□	|	R	q_1	
i_2:	q_1	|	|	R	q_1	;recorre el segundo número hasta el espacio en blanco
i_3:	q_1	□	□	L	q_2	
i_4:	q_2	|	□	L	q_3	
i_5:	q_3	|	□	L	q_4	
i_6:	q_4	|	□	N	*stop*	;la máquina se detiene en el estado *stop*

1.3. Funciones Turing-computables

Precisemos que, en este contexto, el conjunto de los números naturales $\mathbb{N}$ está definido por los números enteros no negativos, es decir $\mathbb{N} = \{0, 1, 2, \dots\}$. Señalemos igualmente que la teoría de la computabilidad opera sobre funciones de la forma $f : \mathbb{N}^n \to \mathbb{N}$, es decir, funciones $f(x_1, x_2, \dots, x_n) = y$ (la función f asigna a un n-tupla $(x_1, x_2, \dots, x_n)$ un único número natural y). Como un caso particular se tienen las funciones de la forma $f : \mathbb{N} \to \mathbb{N}$; esto es, funciones tales que $f(x) = y$ (la función f asigna a un número natural x un único número natural y). Estas funciones son denominadas funciones numérico-teóricas.

A lo largo de nuestra experiencia como estudiantes de matemáticas, hemos encontrado la idea de que una función es una relación que asocia a todos y cada uno de los elementos de su dominio una y sólo una imagen. No obstante, es importante para nuestro trabajo en la teoría de la computabilidad, que hagamos una distinción en el conjunto de las funciones numérico-teóricas, a saber, funciones totales y funciones parciales.

Definición 1.4 (Función total)**.** Sea $f(x_1, \dots, x_n)$ una función n-ádica. Si f está definida para toda $(x_1, \dots, x_n) \in \mathbb{N}^n$, entonces diremos que f es una función total.

Definición 1.5 (Función parcial)**.** Sea $f(x_1, \dots, x_n)$ una función n-ádica. Si f no está definida para al menos una $(x_1, \dots, x_n) \in \mathbb{N}^n$, entonces diremos que f es una función parcial.

Para poder asociar a una máquina de Turing MT una función f_{MT} y así, poder hablar de funciones Turing-computables, es necesario contar con una descripción completa de la ejecución de la máquina, es decir, necesitamos conocer la evolución temporal de la máquina, y para ello, necesitamos conocer los símbolos sobre la cinta, la posición actual sobre la cinta, la instrucción que se ejecuta y los cambios efectuados por la ejecución de dicha instrucción.

Las definiciones que presentamos a continuación, nos permitirán identificar los elementos antes mencionados.

Definición 1.6 (Sucesión)**.** Sea $\mathsf{MT} = \langle Q, \Sigma, M, I \rangle$ una máquina de Turing. Una sucesión es una palabra finita (posiblemente vacía) construida sobre el alfabeto formado por $Q \cup \Sigma \cup M$.

Sea MT una máquina de Turing tal que: $Q = \{q_1, q_2, \dots, q_n\}$, $\Sigma = \{s_0, s_1, \dots, s_m\}$ y $M = \{L, R, N\}$. Algunas posibles sucesiones son: Λ (la palabra vacía), $\alpha \equiv q_1 q_1 L L s_2 s_3$, $\beta \equiv NNNs_0 q_1$ y $\gamma \equiv s_2 s_3 L R q_4$.

Observe el lector que, según la definición anterior, el conjunto de instrucciones I puede ser definido como un conjunto de sucesiones con unas características particulares, en cuanto al número de símbolos (longitud) en cada sucesión y en cuanto al dominio al cual deben pertenecer estos símbolos.

Definición 1.7 (Descripción instantánea)**.** Sea $\mathsf{MT} = \langle Q, \Sigma, M, I \rangle$ una máquina de Turing. Una descripción instantánea α de MT, es una sucesión tal que:

1. Contiene exactamente un símbolo $q_i \in Q$.

2. No contiene ningún símbolo $m \in M$.

3. El símbolo q_i en α no es el último símbolo de izquierda a derecha.

4. Todos los s_i que ocurren en α pertenecen a Σ.

Definición 1.8 (Sucesión de la cinta)**.** Una sucesión que consta únicamente de símbolos $s_i \in \Sigma$ es llamada una sucesión de la cinta. Las sucesiones de la cinta se representarán por los símbolos P y Q.

Las definiciones de descripción instantánea y sucesión de la cinta, junto con las instrucciones de una máquina de Turing, nos permiten formalizar la evolución temporal de la misma. Para tal propósito presentamos las siguientes definiciones.

Definición 1.9 (Cambio descripción instantánea)**.** Sea una máquina de Turing MT y sean α y β descripciones instantáneas de MT. El paso de la descripción instantánea α a la descripción instantánea β se representa por $\alpha \xrightarrow{\mathsf{MT}} \beta$, lo cual significa que alguna de las siguientes condiciones se satisface:

1. Existen sucesiones P y Q (sucesiones de la cinta, posiblemente vacías) tales que:

$$\alpha \equiv P\, q_i\, s_j\, Q$$
$$\beta \equiv P\, q_j\, s_k\, Q$$

   y MT contiene una instrucción $i_a : q_i\, s_j\, s_k\, N\, q_j$.

2. Existen sucesiones P y Q (sucesiones de la cinta, posiblemente vacías) tales que:

$$\alpha \equiv P\, q_i\, s_j\, s_k\, Q$$
$$\beta \equiv P\, s_l\, q_j\, s_k\, Q$$

   y MT contiene una instrucción $i_a : q_i\, s_j\, s_l\, R\, q_j$.

3. Existen sucesiones P y Q (sucesiones de la cinta, posiblemente vacías) tales que:

$$\alpha \equiv P\, q_i\, s_j\, Q$$
$$\beta \equiv P\, s_l\, q_j\, s_0\, Q$$

   donde $s_0 = \square$ y MT contiene una instrucción $i_a : q_i\, s_j\, s_l\, R\, q_j$.

4. Existen sucesiones P y Q (sucesiones de la cinta, posiblemente vacías) tales que:

$$\alpha \equiv P\, s_j\, q_i\, s_k\, Q$$
$$\beta \equiv P\, q_j\, s_j\, s_l\, Q$$

   y MT contiene una instrucción $i_a : q_i\, s_k\, s_l\, L\, q_j$.

5. Existen sucesiones P y Q (sucesiones de la cinta, posiblemente vacías) tales que:

$$\alpha \equiv P\, q_i\, s_j\, Q$$
$$\beta \equiv P\, q_j\, s_0\, s_l\, Q$$

donde $s_0 = \square$ y MT contiene una instrucción $i_a : q_i\, s_j\, s_l\, L\, q_j$.

Definición 1.10 (Descripción terminal)**.** Sea MT un máquina de Turing. Una descripción instantánea α es llamada una descricipción terminal, con respecto a MT, si no existe una descripción instantánea β tal que: $\alpha \xrightarrow{\mathsf{MT}} \beta$.

Definición 1.11 (Computación de una máquina de Turing)**.** Una computación de una máquina de Turing MT es una secuencia finita de descripciones instantáneas $\alpha_1, \ldots, \alpha_n$, tal que: $\alpha_i \xrightarrow{\mathsf{MT}} \alpha_{i+1}$, para $1 \leq i < n-1$, y tal que, α_n es una descripción terminal con respecto a MT. En este caso escribimos: $\alpha_n = \mathsf{Res}_{\mathsf{MT}}(\alpha_1)$ y llamamos a α_n el resultado de α_1 con respecto a MT.

Ejemplo 1.12. Para la máquina MT presentada en el ejemplo 1.3, tenemos la siguiente computación:

$$\begin{aligned}
& \alpha_1 \equiv q_0||\square||| \\
\rightarrow\ & \alpha_2 \equiv |q_0|\square||| \\
\rightarrow\ & \alpha_3 \equiv ||q_0\square||| \\
\rightarrow\ & \alpha_4 \equiv |||q_1||| \\
\rightarrow\ & \alpha_5 \equiv ||||q_1|| \\
\rightarrow\ & \alpha_6 \equiv |||||q_1| \\
\rightarrow\ & \alpha_7 \equiv ||||||q_1\square \\
\rightarrow\ & \alpha_8 \equiv |||||q_2|\square \\
\rightarrow\ & \alpha_9 \equiv ||||q_3|\square\square \\
\rightarrow\ & \alpha_{10} \equiv |||q_4|\square\square\square \\
\rightarrow\ & \alpha_{11} \equiv |||stop\square\square\square\square.
\end{aligned}$$

Entonces, $\alpha_1 1 = \mathsf{Res}_{\mathsf{MT}}(\alpha_1)$.

Una vez formalizada la evolución temporal de una máquina de Turing y el resultado de ejecutar dicha máquina a partir de una descripción instantánea inicial, procedemos a dotar del carácter funcional el resultado obtenido en la ejecución de la máquina. Para ello utilizaremos una codificación similar a la presentada en el ejemplo 1.3.

Definición 1.13 (Codificación números naturales)**.** Cada número $n \in \mathbb{N}$ se asocia con una sucesión $\overrightarrow{n}$ (en la cinta), donde:

$$\overrightarrow{n} = |^{n+1} = \underbrace{|\ldots|}_{n+1\ veces}.$$

Definición 1.14 (Codificación n-tuplas de números naturales)**.** Cada n-tupla de números naturales $(n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$ se asocia una sucesión $(\overrightarrow{n_1, n_2, \ldots, n_k})$ (en la cinta) donde:

$$(\overrightarrow{n_1, n_2, \ldots, n_k}) = \overrightarrow{n_1}\square\overrightarrow{n_2}\square \ldots \square\overrightarrow{n_k} = \overrightarrow{n_1}s_0\overrightarrow{n_2}s_0 \ldots s_0\overrightarrow{n_k}.$$

Por ejemplo, la terna $(0, 1, 2)$ se asocia con la sucesión en la cinta:

$$\begin{aligned}(\overrightarrow{0, 1, 2}) &= |\square||\square|||\\ &= |s_0||s_0|||.\end{aligned}$$

Definición 1.15 (Números de | en una sucesión)**.** Sea γ una sucesión. El número de | en γ lo representamos por $< \gamma >$. Por ejemplo, $< |\square||\square||| >= 6$.

Definición 1.16 (Función asociada a una máquina de Turing)**.** Sea MT una máquina de Turing. Por cada $n \in \mathbb{N}$ se asocia con la máquina MT una función $f^{(n)}_{\mathsf{MT}}(x_1, x_2, \ldots, x_n)$, definida por:

Para cada n-tupla $(m_1, m_2, \ldots, m_n)$ se selecciona una descripción instantánea

$$\alpha_1 \equiv q_1(\overrightarrow{m_1, m_2, \ldots, m_n})$$

y se distinguen dos casos:

1. Existe una computación de MT, $\alpha_1, \alpha_2, \ldots, \alpha_p$. En este caso, tenemos que:

$$\begin{aligned}f^{(n)}_{\mathsf{MT}}(m_1, m_2, \ldots, m_n) &=< \alpha_p >\\ &=< \mathsf{Res}_{\mathsf{MT}}(\alpha_1) > .\end{aligned}$$

2. No existe una computación de MT, $\alpha_1, \alpha_2, \ldots, \alpha_p$. En este caso, tenemos que la función

$$f^{(n)}_{\mathsf{MT}}(m_1, m_2, \ldots, m_n)$$

no está definida.

Definición 1.17 (Función parcial Turing-computable)**.** Sea $f(x_1, \ldots, x_n)$ una función parcial. La función $f(x_1, , \ldots, x_n)$ es una función parcial Turing-computable si y sólo si, existe una máquina de Turing MT, tal que:

$$f(x_1, \ldots, x_n) = f^{(n)}_{\mathsf{MT}}(x_1, \ldots, x_n).$$

En tal caso se dice que MT computa a la función f.

Definición 1.18 (Función total Turing-computable)**.** Sea $f(x_1, x_2, \ldots, x_n)$ una función total. $f(x_1, x_2, \ldots, x_n)$ es una función total Turing-computable si y sólo si, existe una máquina de Turing MT, tal que:

$$f(x_1, x_2, \ldots, x_n) = f^{(n)}_{M}(x_1, x_2, \ldots, x_n).$$

Se dice, igualmente, que MT computa a la función f.

Ejemplo 1.19. La función $f(x) = 0$, donde $x \in \mathbb{N}$ es una función Turing-computable. Para comprobarlo es necesario construir una máquina de Turing MT, tal que: $f(x) = f^{(1)}_{\mathsf{MT}}(x)$. Sea MT la máquina de Turing dada por:

i_1: q_1 | $\square$ R q_1
i_2: q_1 $\square$ $\square$ R *stop*

Así, para $x = 2$ comenzado en, $\alpha_1 \equiv q_1(\overrightarrow{x})$, tenemos la siguiente computación:

$$\begin{aligned} & \alpha_1 \equiv q_1/// \\ \rightarrow \quad & \alpha_2 \equiv \square q_1// \\ \rightarrow \quad & \alpha_3 \equiv \square\square q_1/ \\ \rightarrow \quad & \alpha_4 \equiv \square\square\square q_1\square \\ \rightarrow \quad & \alpha_5 \equiv \square\square\square\square q_1\square. \end{aligned}$$

Entonces,

$$\begin{aligned} f^{(1)}_{\mathsf{MT}}(2) &=< \mathsf{Res}_{\mathsf{MT}}(\alpha_1) > \\ &=< \mathsf{Res}_{\mathsf{MT}}(q_1///) > \\ &=< \alpha_5 > \\ &=< \square\square\square\square q_1\square > \\ &= 0 \\ &= f(2). \end{aligned}$$

La demostración de que $f(x) = f^{(1)}_{\mathsf{MT}}(x)$ se presenta en el lema 2.49.

Ejemplo 1.20. Mostrar que la función $f(x) = x + 1$ es una función Turing-computable. Construyamos una máquina de Turing MT tal que $f(x) = f^{(1)}_{\mathsf{MT}}(x)$. Sea MT la máquina de Turing dada por: i_1: q_1 $\square$ $\square$ N q_1.

Así, para $x = 5$ y comenzado en, $\alpha_1 \equiv q_1(\overrightarrow{x})$, la computación estaría dada por:

$$\alpha_1 \equiv q_1//////.$$

Es decir, la máquina no realiza ningún paso de computación, por lo tanto:

$$\begin{aligned} f^{(1)}_{\mathsf{MT}}(5) &=< \mathsf{Res}_{\mathsf{MT}}(\alpha_1) > \\ &=< \mathsf{Res}_{\mathsf{MT}}(q_1//////) > \\ &=< \alpha_1 > \\ &=< q_1////// > \\ &= 6 \\ &= f(x). \end{aligned}$$

Una demostración de que $f(x) = f^{(1)}_{\mathsf{MT}}(x)$ se presenta en el lema 2.50.

1.4. M-funciones

En la actualidad, el comportamiento de cualquier máquina de Turing se acostumbra expresarlo en la notación de instrucciones. Cada instrucción está constituida por una quíntupla:

$$q_i \; s_j \; s_k \; m \; q_l$$

donde:

(i) q_i es el estado actual,

(ii) s_j es el símbolo que se lee,

(iii) s_k es el símbolo que se escribe,

(iv) m es el movimiento que se realiza,

(v) q_l es el estado final.

Aunque la notación anterior fue propuesta por Turing en su artículo fundacional sobre las máquinas de Turing, las máquinas construidas por él en dicho artículo (en particular la máquina universal de Turing), no están descritas en la "notación de instrucciones"; por el contrario, están descritas en una notación que permite simplificar considerablemente el número de instrucciones necesarias para describir el comportamiento de una máquina; dicha notación tiene como elemento principal el concepto de *m-función*.

Existen ciertos tipos de tareas tales como copiar una determinada secuencia de símbolos, buscar un determinado símbolo en la cinta, imprimir una determinada cadena de símbolos al final de la cinta, entre otras, que son de uso muy frecuente en la construcción de máquinas de Turing de cierta complejidad. La idea entonces es contruir una máquina de Turing *génerica* para cada una de estas tareas, de tal forma que pueda ser invocada por otras máquinas de Turing que necesiten realizar dicha tarea. El concepto de m-función corresponde a estas máquinas de Turing génericas.

Presentamos a continuación algunos elementos necesarios para poder formalizar (parcialmente) y ejemplificar el uso de las m-funciones. Además, se indicarán los procedimientos requeridos para expresar en la notación de instrucciones cualquier m-función, lo cual nos permitirá paladear la simplicidad-complejidad ofrecida por dichas m-funciones.

Como mencionamos con anterioridad, la descripción de algunas de las máquinas propuestas por Turing se realiza con base en las m-funciones y en una notación no tradicional (diferente a la notación de instrucciones). Esta notación no tradicional se presenta por medio de los siguientes ejemplos:

Ejemplo 1.21. Es frecuente, en Alan Turing, el uso de una notación especial para escribir o borrar un símbolo (tal como lo indican las tablas 1.6 y 1.7).

Ejemplo 1.22. Turing también utilizaba una notación que permitía en la columna de operaciones varios símbolos (tal como lo indica la tabla 1.8).

En la notación actual, la tabla 1.8 se podría escribir como lo indica la tabla 1.9.

Estado Inicial	Símbolo	Operación	Estado Final
B	E		B

Tabla 1.6: Notación no tradicional: eliminación de un símbolo.

Estado Inicial	Símbolo	Operación	Estado Final
B		$P\alpha$	B

Tabla 1.7: Notación no tradicional: escritura del símbolo α sobre la cinta.

Ejemplo 1.23. Un ejemplo adicional del uso de varios símbolos en la columna de operaciones se indica en la tabla 1.10.

En la notación actual, la tabla 1.10 se podría escribir como lo indica la tabla 1.11.

Este ejemplo permite entrever la dificultad que se presenta al traducir la notación utilizada por Alan Turing a la notación de instrucciones. El doble movimiento indicado por la operación "$R, R, P1$", para ser escrito en la notación tradicional, exige conocer el conjunto de símbolos que pueden estar sobre la cinta. Este conjunto de símbolos se denota por el símbolo '$Cualquiera$'.

Ejemplo 1.24. La notación no tradicional permite escribir en una sola instrucción el número de operaciones que se desee (como lo permite observar la tabla 1.12).

En este caso, a partir del estado B, sin importar sobre cuál símbolo del alfabeto se encuentre la máquina, ésta imprime el símbolo e, se mueve a la derecha, imprime de nuevo el símbolo e, se mueve a la derecha, imprime el símbolo 0, se mueve dos casillas a la derecha, imprime el símbolo 0, finalmente se mueve dos casillas a la izquierda y pasa al estado C (mirar ejercicio 1.7).

Definición 1.25 (m-función)**.** Una m-función es una máquina de Turing que permite el uso de parámetros para los estados y los símbolos que la conforman.

El uso de parámetros permite que la m-función sea llamada por una máquina de Turing, instanciando éstos de acuerdo con sus necesidades específicas. El uso y la combinación de estas máquinas de Turing parametrizadas o m-funciones simplifica la construcción de las máquinas de Turing.

Definición 1.26 (Expansión m-función)**.** El proceso de la traducción final de una m-función a la notación de instrucciones (instrucciones de la forma: $q_i\ s_j\ s_k\ m\ q_l$), exige conocer el valor de los parámetros con los cuales es invocada, y además conocer el alfabeto de la máquina de la cual la m-función hace parte. Aunque esta traducción no es posible de realizarla con base en la definición de la m-función, sí es posible hacer una traducción parcial a la "nomenclatura de instrucciones" que facilite el proceso de traducción final, una vez se conozca la información (valor de los parámetros y alfabeto de la máquina) requerida para ello; este proceso de traducción parcial lo denominamos la *expansión* de la m-función.

Estado Inicial	Símbolo	Operación	Estado Final
B	Ninguno	$P0, L$	B

Tabla 1.8: Notación no tradicional: varios símbolos en la columna de operaciones (1).

Estado Actual	Símbolo Leer	Símbolo Escribir	Movimiento	Estado Siguiente
B	$\square$	0	L	B

Tabla 1.9: Escritura en la notación actual para la tabla 1.8.

Los ejemplos 1.27, 1.28 y 1.29 operan bajo la hipótesis de que existe un símbolo e como el símbolo más a la izquierda en la cinta; además suponen que los símbolos principales (denotados con s_i) están intercalados con símbolos auxiliares (denotados con m_i), tal como lo indica la tabla 1.13.

Ejemplo 1.27. Dado que exista un símbolo e como el símbolo más a la izquierda de la cinta, la m-función $\mathbf{F}(S, B, \alpha)$ busca el símbolo α; si lo encuentra, pasa al estado S, si no lo encuentra, pasa al estado B. La m-función $\mathbf{F}(S, B, \alpha)$ está dada por la tabla 1.14.

Es frecuente que una m-función—en este caso la m-función $\mathbf{F}(S, B, \alpha)$—esté compuesta por varias m-funciones: para este caso, $\mathbf{F_1}(S, B, \alpha)$ y $\mathbf{F_2}(S, B, \alpha)$. De las definiciones de las m-funciones $\mathbf{F_1}(S, B, \alpha)$ y $\mathbf{F}(S, B, \alpha)$ se observa que la definición de $\mathbf{F_1}(S, B, \alpha)$ presentada en la tabla 1.15, indica el comportamiento para el símbolo 'Ninguno'. Por el contrario, la definición para $\mathbf{F}(S, B, \alpha)$ presentada en la tabla 1.16, no lo indica. En este caso el comportamiento del símbolo 'Ninguno' está implícito en el comportamiento para el "símbolo" 'No e'. Señalemos además que la expansión de la m-función $\mathbf{F}(S, B, \alpha)$ se presenta en la tabla 1.17.

Ejemplo 1.28. Dado que exista un símbolo e como el símbolo más a la izquierda de la cinta, la m-función $\mathbf{PE}(S, \beta)$ imprime el símbolo β al final de la secuencia de símbolos y pasa al estado S. La m-función $\mathbf{PE}(S, \beta)$ está dada por la tabla 1.18.

Para simplicar la expansión de la m-función $\mathbf{F}(\mathbf{PE_1}(S, \beta), S, e)$ se supondrá que el símbolo 'e' está en la cinta. Note el lector que la expansión de la m-función $\mathbf{PE}(S, \beta)$ se presenta en la tabla 1.19.

Ejemplo 1.29. Con base en que exista un símbolo e como el símbolo más a la izquierda de la cinta, la m-función $\mathbf{PE_2}(S, \alpha, \beta)$ imprime los símbolos α y β al final de la secuencia de símbolos y entonces pasa al estado S. Note además que la m-función $\mathbf{PE_2}(S, \alpha, \beta)$ está dada por la tabla 1.20 (mirar ejercicio 1.8).

Estado Inicial	Símbolo	Operación	Estado Final
B	0	$R, R, P1$	B

Tabla 1.10: Notación no tradicional: Varios símbolos en la columna de operaciones (2).

Estado Actual	Símbolo Leer	Símbolo Escribir	Movimiento	Estado Siguiente
q_i	0	0	R	q_{i+1}
q_{i+1}	Cualquiera	Cualquiera	R	q_{i+2}
q_{i+2}	Cualquiera	1	N	q_i

Tabla 1.11: Escritura en la notación actual para la tabla 1.10.

1.5. Codificación de las máquinas de Turing

1.5.1. Codificación de Turing

Para desarrollar la solución al caso general del *problema de la decisión*, Turing utilizó una enumeración muy particular de sus máquinas. Esta enumeración tiene como base un mecanismo de codificación de las mismas. Por otro lado, la codificación de las máquinas es necesaria para definir un procedimiento que sea capaz de ejecutar cualquier máquina de Turing (en la sección correspondiente a la máquina de Turing universal desarrollaremos esta idea).

Presentemos entonces la codificación y la enumeración de máquinas propuesta por Turing. De acuerdo con la definición formal de una máquina de Turing tenemos el siguiente "esquema" para una instrucción: $q_i\ s_j\ s_k\ m\ q_l$; donde $i, j, k, l \geq 0$ y $m \in M = \{L, R, N\}$. Con base en esta representación, reemplazamos cada instrucción de acuerdo con la siguiente codificación:
q_i: Se reemplaza por una D seguida de A repetida i veces
s_j: Se reemplaza por una D seguida de C repetida j veces

Acto seguido, todas las instrucciones de la máquina se reescriben utilizando este código y se separan por un punto y coma (;).

Definición 1.30 (Descripción estándar)**.** Cuando describimos el conjunto de instrucciones de nuestra máquina utilizando este sistema de codificación, decimos que la máquina está representada en su descripción estándar. Las instrucciones de la descripción estándar estarán construidas con símbolos del alfabeto $\Sigma = \{A, C, D, R, L, N, ;\}$.

De forma similar, asociamos a cada símbolo del alfabeto Σ un número natural mediante

EI	S	Operación	EF
B		$Pe, R, Pe, R, P0, R, R, P0, L, L$	C

Tabla 1.12: Notación no tradicional: múltiples operaciones.

$\ldots$		e		s_i	m_i	s_i		s_i	m_i	s_i		s_i		$\ldots$

Tabla 1.13: Estado de la cinta para los ejemplos 1.27, 1.28 y 1.29.

la siguiente función $f : \Sigma \to \mathbb{N}$; donde:

$$f(s) = \begin{cases} 1, & \text{si } s = A; \\ 2, & \text{si } s = C; \\ 3, & \text{si } s = D; \\ 4, & \text{si } s = L; \\ 5, & \text{si } s = R; \\ 6, & \text{si } s = N; \\ 7, & \text{si } s = ;. \end{cases}$$

Luego reemplazamos cada símbolo de la descripción estándar de una máquina de Turing por el valor numérico que tiene asociado.

Definición 1.31 (Número de descripción)**.** El número así obtenido es denominado el *número de descripción* de la máquina codificada.

Ejemplo 1.32. Hallemos la descripción estándar y el número de descripción para la máquina MT:

i_1:	q_0	$\square$	/	R	q_1
i_2:	q_1	/	/	R	q_1
i_3:	q_1	$\square$	$\square$	L	q_2
i_4:	q_2	/	$\square$	N	*stop*

Renombremos los símbolos utilizados por MT así: $\square$ por s_0, y / por s_1. Además renombremos el estado *stop* con el símbolo q_3. Ahora podemos escribir de nuevo las instrucciones y obtener la descripción estándar de cada una de ellas como lo indica la tabla 1.21.

Realizado este proceso de codificación, tendríamos la siguiente descripción estándar para la máquina MT:

$$DDDCRDA; DADCDCRDA; DADDLDAA; DAADCDNDAAA;$$

y su número de descripción sería, entonces:

$$3332531731323253173133431173113236311117.$$

Estado Inicial	Símbolo	Operación	Estado Final
$\mathbf{F}(S, B, \alpha)$	e	L	$\mathbf{F_1}(S, B, \alpha)$
	No e	L	$\mathbf{F}(S, B, \alpha)$
$\mathbf{F_1}(S, B, \alpha)$	α		S
	No α	R	$\mathbf{F_1}(S, B, \alpha)$
	Ninguno	R	$\mathbf{F_2}(S, B, \alpha)$
$\mathbf{F_2}(S, B, \alpha)$	α		S
	No α	R	$\mathbf{F_1}(S, B, \alpha)$
	Ninguno	R	B

Tabla 1.14: M-función $\mathbf{F}(S, B, \alpha)$.

Estado Inicial	Símbolo	Operación	Estado Final
$\mathbf{F_1}(S, B, \alpha)$	α		B
	No α	R	$\mathbf{F_1}(S, B, \alpha)$
	Ninguno	R	$\mathbf{F_2}(S, B, \alpha)$

Tabla 1.15: M-función $\mathbf{F_1}(S, B, \alpha)$.

Podemos observar que para cada máquina de Turing existe un número natural que la representa, mas no todo número natural representa una máquina de Turing. Además, de acuerdo con la enumeración anterior, es posible demostrar que el conjunto de las máquinas de Turing es enumerable (y en particular infinito), y naturalmente lo es el conjunto de los algoritmos que se pueden construir sobre algún lenguaje. Esto debe ser así, si somos consistentes con la idea de que cualquier algoritmo puede ser representado por una máquina de Turing.

1.5.2. Codificación de Gödel

Existen otros procedimientos de codificación para las máquinas de Turing diferentes al propuesto por Turing. Presentamos entonces, uno de estos procedemientos, basado en los números de Gödel.

Para la demostración de su famoso teorema de incompletitud de la aritmética, Kurt Gödel, utilizó la idea hoy conocida como codificación numérica o códigos de números. La potencia de esta técnica produjo su estandarización en el campo de la lógica matemática y teorías afines. En otros términos, Gödel se sirvió de una codificación numérica de las cadenas o palabras del lenguaje de la aritmética. Es decir, mediante una previa ordenación de los símbolos del alfabeto, logró generar una técnica para codificar éste y en general, cualquier lenguaje cuyo alfabeto sea enumerable. Gödel nos mostró, en detalle, cómo codificar las reglas de inferencia de un sistema formal, así como el conjunto de sus axiomas y teoremas.

Estado Inicial	Símbolo	Operación	Estado Final
$\mathbf{F}(S, B, \alpha)$	e	L	$\mathbf{F_1}(S, B, \alpha)$
	No e	L	$\mathbf{F}(S, B, \alpha)$

Tabla 1.16: M-función $\mathbf{F}(S, B, \alpha)$.

EA[a]	SL[b]	SE[c]	M[d]	ES[e]	Explicación
q_i	e	e	L	q_{i+1}	Expansión m-función $\mathbf{F}(S, B, \alpha)$
q_i	No e	No e	L	q_i	
q_i	$\square$	$\square$	L	q_i	
q_{i+1}	α	α	N	S	Expansión m-función $\mathbf{F_1}(S, B, \alpha)$
q_{i+1}	No α	$No\ \alpha$	R	q_{i+1}	
q_{i+1}	$\square$	$\square$	R	q_{i+2}	
q_{i+2}	α	α	N	S	Expansión m-función $\mathbf{F_2}(S, B, \alpha)$
q_{i+2}	No α	No α	R	q_{i+1}	
q_{i+2}	$\square$	$\square$	R	B	

[a]EA: Estado Actual
[b]SL: Símbolo Leer
[c]SE: ímbolo Escribir
[d]M: Movimiento
[e]ES: Estado Siguiente

Tabla 1.17: Expansión m-función $\mathbf{F}(S, B, \alpha)$.

El objetivo de la presente sección se limitará a presentar, no lo que Gödel hizo, sino lo esencial de su técnica de codificación y su aplicación a las máquinas de Turing.

Definición 1.33 (Codificación de Gödel)**.** Supongamos que $\Sigma = \{b_1, b_2, \dots\}$ es un alfabeto enumerable y sean Σ^* y Σ^n los conjuntos de palabras y n-tuplas construibles sobre el alfabeto Σ. Llamaremos codificación de Gödel, del conjunto Σ^n, a toda función inyectiva $f : \Sigma^n \to \mathbb{N}$. Igualmente, la función $f : \Sigma^* \to \mathbb{N}$ la llamaremos la codificación de Gödel para Σ^*.

El procedimiento o técnica de codificación (o aritmetización) de Gödel está sustentado en el teorema fundamental de la aritmética. Este procedimiento lo podemos dividir en dos partes, a saber:

Primeramente, definimos una función, digamos NG, sobre el conjunto Σ (el alfabeto); posteriormente, definimos una extensión de esta función (o del proceso), apoyándonos en el teorema fundamental de la aritmética. Así, para el caso del alfabeto $\Sigma = \{b_1, b_2, \dots\}$, tenemos

1. $\mathsf{NG}(b_i) = i$.

Estado Inicial	Símbolo	Operación	Estado Final
$\mathbf{PE}(S,\beta)$			$\mathbf{F}(\mathbf{PE_1}(S,\beta),S,e)$
$\mathbf{PE_1}(S,\beta)$	Cualquiera	R,R	$\mathbf{PE_1}(S,\beta)$
$\mathbf{PE_1}(S,\beta)$	Ninguno	$P\beta$	S

Tabla 1.18: M-función $\mathbf{PE}(S,\beta)$.

EA	SL	SE	M	ES	Explicación
q_i	e	e	N	q_{i+1}	Expansión m-función $\mathbf{PE}(S,\beta)$
q_i	No e	No e	L	q_i	
q_i	$\square$	$\square$	L	q_i	
q_{i+1}	C[a]	C	R	q_{i+2}	
q_{i+1}	$\square$	β	N	S	
q_{i+2}	C	C	R	q_{i+1}	
q_{i+2}	$\square$	$\square$	R	q_{i+1}	

[a]C: Cualquiera

Tabla 1.19: Expansión m-función $\mathbf{PE}(S,\beta)$.

2. Si $\alpha \in \Sigma^*$ y $\alpha \equiv s_1 s_2 \dots s_k$, sea $\mathsf{Pn}(k)$ la función que calcula el k-ésimo primo, entonces

$$\begin{aligned}\mathsf{NG}(\alpha) &= \mathsf{Pn}(1)^{\mathsf{NG}(s_1)} \cdot \mathsf{Pn}(2)^{\mathsf{NG}(s_2)} \dots \mathsf{Pn}(k)^{\mathsf{NG}(s_k)} \\ &= \prod_{i=1}^{k} \mathsf{Pn}(i)^{\mathsf{NG}(s_i)}\end{aligned}$$

La función NG, transforma la palabra $\alpha \in \Sigma^*$ en un número natural $\mathsf{NG}(\alpha)$. Este número se conoce como el número de Gödel para α, según la enumeración o codificación NG.

Ejemplo 1.34.

1. Consideremos el alfabeto Σ cuyos elementos son los símbolos de la siguiente lista:

$$+, \cdot, (,), =, 0, 1, \dots, 9, a, b, x_1, x_2, \dots, x_i, \dots$$

Estado Inicial	Símbolo	Operación	Estado Final
$\mathbf{PE_2}(S, \alpha, \beta)$			$\mathbf{PE}(\mathbf{PE}(S, \beta), \alpha)$

Tabla 1.20: M-función $\mathbf{PE_2}(S, \alpha, \beta)$.

	Instrucción original	Instrucción renombrada	Instrucción codificada
i_1:	q_0 □ / R q_1	q_0 s_0 s_1 R q_1	$DDDCRDA$
i_2:	q_1 / / R q_1	q_1 s_1 s_1 R q_1	$DADCDCRDA$
i_3:	q_1 □ □ L q_2	q_1 s_0 s_0 L q_2	$DADDLDAA$
i_4:	q_2 / □ N *stop*	q_2 s_1 s_0 N q_3	$DAADCDNDAAA$

Tabla 1.21: Descripción estándar de instrucciones.

2. Definimos la función NG, que codifica los símbolos de Σ, como sigue:

$$\mathsf{NG}(s) = \begin{cases} 1, & \text{si } s = +; \\ 2, & \text{si } s = \cdot; \\ 3, & \text{si } s = (; \\ 4, & \text{si } s =); \\ 5, & \text{si } s = =; \\ 6, & \text{si } s = 0; \\ 7, & \text{si } s = 1; \\ \vdots & \\ 15, & \text{si } s = 9; \\ 16, & \text{si } s = a; \\ 17, & \text{si } s = b; \\ 18, & \text{si } s = x_1; \\ \vdots & \end{cases}$$

3. ¿Cuál es el código o número de Gödel de la expresión α?

$$\begin{aligned} \alpha &\equiv s_1 s_2 s_3 s_4 s_5 s_6 \\ &\equiv 2 \cdot 5 = 10. \end{aligned}$$

De acuerdo con esta codificación o enumeración la expresión α tendría el siguiente

número de Gödel:

$$\begin{aligned}\mathsf{NG}(2\cdot 5=10) &= \prod_{i=6}^{k}\mathsf{Pn}(i)^{\mathsf{NG}(s_i)}\\ &= 2^{\mathsf{NG}(2)}\cdot 3^{\mathsf{NG}(\cdot)}\cdot 5^{\mathsf{NG}(5)}\cdot 7^{\mathsf{NG}(=)}\cdot 11^{\mathsf{NG}(1)}\cdot 13^{\mathsf{NG}(0)}\\ &= 2^8\cdot 3^2\cdot 5^{11}\cdot 7^5\cdot 11^7\cdot 13^6\\ &= 177849083895509989462500000000.\end{aligned}$$

Como la función NG es inyectiva, podemos garantizar la unicidad del código y la existencia de códigos diferentes para palabras diferentes de Σ^*. Igualmente, podemos saber si dado un $n\in\mathbb{N}$ existe una palabra de Σ^* codificada por dicho número. Y en caso de que exista, podemos determinar tal palabra; para lograrlo es suficiente hallar la factorización prima de n en $\mathbb{N}$.

Ejemplo 1.35. Para la función de codificación presentada en el ejemplo 1.34, ¿Existe una palabra de Σ^*, cuyo código sea $n=2250$?
Solución:

1. Factorizamos a n, esto es, $n=2^1\cdot 3^2\cdot 5^3$.
2. Hallamos, si existen, $s_1,s_2,s_3\in\Sigma$, tales que $\mathsf{NG}(s_1)=1$; $\mathsf{NG}(s_2)=2$ y $\mathsf{NG}(s_3)=3$. Luego $s_1=+$, $s_2=\cdot$ y $s_3=($
3. Luego, $\alpha\equiv +\cdot($

Definición 1.36 (Codificación de Gödel: instrucción)**.** El alfabeto Σ para una instrucción de una máquina de Turing está dado por:

$$\Sigma=\{R,L,N,s_0,\ldots,s_m,q_0,\ldots,q_n\}.$$

Definimos la función NG que codifica los símbolos de Σ por:

$$\mathsf{NG}(a)=\begin{cases}3, & \text{si } a=R;\\ 5, & \text{si } a=L;\\ 7, & \text{si } a=N;\\ 9, & \text{si } a=s_0;\\ 11, & \text{si } a=q_0;\\ 13, & \text{si } a=s_1;\\ 15, & \text{si } a=q_1;\\ \vdots & \\ 4i+9, & \text{si } s=s_i;\\ 4i+11, & \text{si } s=q_i.\end{cases}$$

El esquema para un instrucción de una máquina de Turing es: $q_i\ s_j\ s_k\ m\ q_l$, donde $i, j, k, l \geq 0$ y $m \in M = \{L, R, N\}$. Si definimos los símbolos a_1, a_2, a_3, a_4, a_5 por: $a_1 = q_i$, $a_2 = s_j$, $a_3 = s_k$, $a_4 = m$ y $a_5 = q_l$, entonces de acuerdo con la codificación anterior para el alfabeto Σ, el número de Gödel asociado con una instrucción, está dado por:

$$\mathsf{NG}(i) = \prod_{k=1}^{5} \mathsf{Pn}(k)^{\mathsf{NG}(a_k)}.$$

Ejemplo 1.37. Sean las instrucciones:

i_1: q_0 s_0 s_1 R q_1
i_2: q_1 s_1 s_1 R q_1

Su número de Gödel está dado por:

$$\begin{aligned}\mathsf{NG}(i_1) &= \prod_{k=1}^{5} \mathsf{Pn}(k)^{\mathsf{NG}(a_k)} \\ &= 2^{\mathsf{NG}(q_0)} \cdot 3^{\mathsf{NG}(s_0)} \cdot 5^{\mathsf{NG}(s_1)} \cdot 7^{\mathsf{NG}(R)} \cdot 11^{\mathsf{NG}(q_1)} \\ &= 2^{11} \cdot 3^9 \cdot 5^{13} \cdot 7^3 \cdot 11^{15} \\ &= 705043151787065817777975000000000000.\end{aligned}$$

$$\begin{aligned}\mathsf{NG}(i_2) &= \prod_{k=1}^{5} \mathsf{Pn}(k)^{\mathsf{NG}(a_k)} \\ &= 2^{\mathsf{NG}(q_1)} \cdot 3^{\mathsf{NG}(s_1)} \cdot 5^{\mathsf{NG}(s_1)} \cdot 7^{\mathsf{NG}(R)} \cdot 11^{\mathsf{NG}(q_1)} \\ &= 2^{15} \cdot 3^{13} \cdot 5^{13} \cdot 7^3 \cdot 11^{15} \\ &= 913735924716037299840255600000000000000.\end{aligned}$$

Definición 1.38 (Codificación de Gödel: máquina de Turing)**.** Sea MT una máquina de Turing compuesta por un conjunto de instrucciones $\{i_1, i_2, \ldots, i_n\}$. El número de Gödel para MT está dado por:

$$\mathsf{NG}(\mathsf{MT}) = \prod_{k=1}^{n} \mathsf{Pn}(k)^{\mathsf{NG}(i_k)}.$$

Ejemplo 1.39. Sea MT una máquina de Turing compuesta por las instrucciones:

i_1: q_0 s_0 s_1 R q_1
i_2: q_1 s_1 s_1 R q_1

El número de Gödel de la máquina de Turign MT, está dado por:

$$\begin{aligned}\mathsf{NG}(\mathsf{MT}) &= \prod_{k=1}^{2} \mathsf{Pn}(k)^{\mathsf{NG}(i_k)} \\ &= 2^{\mathsf{NG}(i_1)} \times 3^{\mathsf{NG}(i_2)} \\ &= 2^{2^{11}\cdot 3^{9}\cdot 5^{13}\cdot 7^{3}\cdot 11^{15}} \times 3^{2^{15}\cdot 3^{13}\cdot 5^{13}\cdot 7^{3}\cdot 11^{15}} \\ &= 2^{705043151787065817777975000000000000} \times 3^{913735924716037299840255600000000000000}.\end{aligned}$$

1.6. Máquina universal de Turing

Alan Mathison Turing no sólo definió formalmente el concepto de algoritmo por medio de sus máquinas de Turing, sino que además (en el mismo artículo donde describió la noción de máquina de Turing), construyó el modelo teórico de nuestros computadores actuales por medio de una máquina abstracta conocida en nuestros días como *máquina universal de Turing*. Posteriormente a estos trabajos, Turing dirigió su interés hacia el área de la Inteligencia Artificial. En esta área se destaca su test para verificar si un computador goza de la propiedad de "inteligencia", test hoy conocido como la *verificación de Turing*.

La máquina universal de Turing es una máquina de Turing muy particular. Esta máquina recibe como entrada una máquina de Turing y una secuencia de datos iniciales; así, la máquina universal de Turing realiza la ejecución de la máquina de Turing (dada como entrada) con la secuencia de datos iniciales. Expresaremos, en notación funcional, esta noción como sigue: sean U, la máquina universal de Turing, MT una máquina de Turing, α una secuencia de datos iniciales y β una secuencia de datos de salida, entonces:

$$\mathsf{U}(\mathsf{MT}, \alpha) = \beta \quad \text{si y sólo si} \quad \mathsf{MT}(\alpha) = \beta.$$

En el lenguaje informático lo expresamos por: un computador (máquina universal de Turing) ejecuta un algoritmo (máquina de Turing) con unos datos de entrada (secuencia de datos iniciales) y genera unos datos de salida (secuencia de datos salida).

Como mencionamos anteriormente, la máquina universal de Turing es una máquina de Turing muy particular: antes de comenzar su ejecución, los datos de entrada (máquina de Turing, secuencia de datos iniciales) deben estar en la cinta. La máquina de Turing actúa como "primer" dato de entrada, y debe estar expresada en su descripción estándar; a su derecha se escribe la secuencia de datos iniciales, como lo indica la tabla 1.22.

...	Descripción estándar de una MT		secuencia datos iniciales	...

Tabla 1.22: Entrada para la máquina universal de Turing.

La máquina universal de Turing está compuesta por un conjunto de instrucciones que "saben" cómo "ejecutar" la máquina de Turing (expresada en su descripción estándar) con

la secuencia de datos iniciales. La descripción exacta del comportamiento de la máquina universal de Turing es bastante compleja (por ello remitimos al lector interesado a la bibliografía).

En resumen podemos sintetizar lo dicho como sigue: la máquina universal de Turing debe "capturar" la situación actual (estado actual, símbolo actual) de la máquina de Turing. Entonces, debe ir a buscar una instrucción para esta situación actual (en la sección de la descripción estándar de la máquina); si la encuentra, debe realizar lo que esta instrucción indica (en la sección de la secuencia de datos iniciales), almacenando cuál era la posición de la máquina (que está ejecutando) sobre los mismos; luego va de nuevo y busca una instrucción para la nueva situación actual y así continúa su proceso hasta que la máquina de Turing se detenga (si éste fuese el caso).

1.7. El problema de la parada

Antes de realizar la presentación formal del problema de la parada, consideremos las siguientes definiciones y "propiedades" fundamentales para nuestro trabajo.

(i) Una función $f(x)$ es efectivamente calculable si y sólo si existe un algoritmo que la calcula.

(ii) Dado el hecho de que los algoritmos se presentan como constructos de un lenguaje en particular, podemos pensar en una ordenación de los mismos. Primero, tendríamos los algoritmos de longitud uno (si existen), luego los algoritmos de longitud dos, y, así sucesivamente. Para los algoritmos de longitud n, podemos realizar un ordenamiento lexicográfico, ordenamiento con el cual obtendríamos finalmente una enumeración de todos los algoritmos. Denotemos esta enumeración por: $A_1, A_2, \ldots, A_n, \ldots$.

(iii) Sea $f_i(x)$, la función efectivamente calculable, calculada por el algoritmo A_i.

(iv) Definamos una nueva función $g(x) = f_x(x) + 1$, para toda x. Podemos observar que $g(x)$ es una función efectivamente calculable, dado que se obtiene sumando uno al algoritmo usado para calcular la función $f_x(x)$. Este procedimiento es algorítmico.

(v) Dado que $g(x)$ es una función efectivamente calculable, debe existir una función $f_i(x)$ tal que, $f_i(x) = g(x)$. Es decir, debe existir un algoritmo A_i que calcula la función $g(x)$.

De otra parte, tenemos el siguiente teorema.

Teorema 1.40. *No existe $f_i(x)$ tal que, $f_i(x) = g(x)$. Es decir, $g(x)$ es una función que no es efectivamente calculable.*

Demostración.

1. $g(x) = f_x(x) + 1$, para todo x.
2. Sea $f_i(x) = g(x) = f_x(x) + 1$ para todo x.
3. Para $x = i$ tenemos $f_i(i) = g(i) = f_i(i) + 1$, lo cual es una contradicción.
4. Por lo tanto, no existe $f_i(x)$, tal que $f_i(x) = g(x)$. □

Por una parte, las propiedades y razonamientos implicados en los numerales (i) a (v) nos indican que $g(x)$ es una función efectivamente calculable, pero el teorema 1.40 nos indica que $g(x)$ no es una función efectivamente calculable. ¿Cuál de los razonamientos es incorrecto?

El razonamiento incorrecto es el realizado en los numerales (i) a (v). En particular el numeral (ii) supone la existencia del ordenamiento de los posibles algoritmos. Dado varios algoritmos de longitud n, no vemos inconveniente en realizar un orden lexicográfico sobre ellos. Entonces, ¿cuál es el error en este punto? El error consiste en suponer que contamos con un criterio de decidibilidad para los algoritmos, es decir, que dada una secuencia de palabras, se pueda identificar si éstas forman o no un algoritmo. La noción intuitiva de algoritmo nos exige la ejecución de un número finito de instrucciones en un tiempo finito. Con respecto al número finito de instrucciones, no existe inconveniente en su detección. El problema consiste en que no contamos con un procedimiento efectivo que nos permita decidir si un conjunto de instrucciones finitas se detendrá o no a partir de una secuencia de datos iniciales y, al no contar con este procedimiento efectivo, entonces no podemos identificar el conjunto de instrucciones como un algoritmo (se detiene) o como un no algoritmo (no se detiene). Este problema de determinar si un conjunto de instrucciones se detiene o no, es conocido como el problema de la parada.

El *problema de la decisión*, en su caso general, fue replanteado por Turing en los siguientes términos: ¿Es posible construir una máquina de Turing capaz de responder si una máquina de Turing se detendrá o no para una secuencia de datos iniciales determinada? Expresado en otros términos:

Sean, MT una máquina de Turing y α una secuencia de datos iniciales. Existe una máquina de Turing P, tal que:

$$\mathsf{P}(\mathsf{MT}, \alpha) = \begin{cases} 1, & \text{si } \mathsf{MT} \text{ se detiene con } \alpha; \\ 0, & \text{si } \mathsf{MT} \text{ no se detiene con } \alpha. \end{cases}$$

Esta formulación (de Turing) se conoce como *el problema de la parada.*

El lector observará que para ciertas máquinas de Turing y para ciertas secuencias de datos iniciales es posible determinar si la máquina se detendrá o no. Por ejemplo, para la máquina MT construida en el ejemplo 1.1 y la secuencia de datos iniciales vacía, la máquina no se detiene. Por el contrario, la máquina MT construida en el ejemplo 1.3 y para una secuencia de datos iniciales constituida por dos números naturales, expresados en el código de "palitos", sí se detiene. Pero estas, son soluciones al problema de la parada para un caso

particular, y nos interesa la solución al problema de la parada para su caso general, es decir: ¿es posible construir una máquina de Turing P que resuelva el problema de la parada, para cualquier máquina de Turing y para cualquier secuencia de datos iniciales?

Teorema 1.41. *Si* MT *es una máquina de Turing y* α *es una secuencia de datos iniciales, entonces no existe una máquina de Turing* P*, tal que:*

$$\mathsf{P}(\mathsf{MT}, \alpha) = \begin{cases} 1, & \textit{si } \mathsf{MT} \textit{ se detiene con } \alpha; \\ 0, & \textit{si } \mathsf{MT} \textit{ no se detiene con } \alpha. \end{cases}$$

Demostración. La demostración procederá por *reduction ad absurdum*. Supongamos que existe P, entonces podemos definir una nueva máquina de Turing H por:

$$\mathsf{H}(\alpha) = \begin{cases} 1, & \text{si } \mathsf{P}(\mathsf{MT}_\alpha, \alpha) = 0; \\ \text{no se detiene}, & \text{si } \mathsf{P}(\mathsf{MT}_\alpha, \alpha) = 1. \end{cases}$$

Como H es una máquina de Turing le corresponde un número de descripción; denominemos este número MT_β, e invoquemos a H con el parámetro β. Entonces, $\mathsf{H}(\beta) = 1$ si $\mathsf{P}(\mathsf{MT}_\beta, \beta) = 0$. De acuerdo con la definición de P, $\mathsf{P}(\mathsf{MT}_\beta, \beta) = 0$, lo cual significa que la máquina MT_β con la entrada β no de detiene. Pero la máquina MT_β es la máquina H y estamos afirmando que $\mathsf{H}(\beta) = \mathsf{MT}_\beta(\beta) = 1$, es decir, que la máquina MT_β con la entrada β se detiene, y esto es una contradicción. Un razonamiento similar puede ser construido para, $\mathsf{H}(\beta) = 0$ no se detiene. Podemos entonces concluir que no existe la máquina $\mathsf{P}(\mathsf{MT}, \alpha)$. □

El teorema anterior expresado en términos del problema de la decisión nos permite afirmar el problema de la decisión; el caso general, no tiene solución.

El problema de la parada no es el único problema que no puede ser resuelto por una máquina de Turing. Como un ejemplo adicional presentamos el problema conocido como *el problema del castor afanoso (the busy beaver problem).*

Ejemplo 1.42. Sea $\Sigma(n)$ el número máximo de "unos" que puede escribir una máquina de Turing con n estados antes de detenerse (sin contar con el estado donde se detiene), comenzando con la cinta en blanco, con un alfabeto dado por $\{|\}$ y en la cual sólo se permiten dos movimientos, a izquierda o a derecha. Por otra parte, la función $\mathsf{S}(n)$ representa el número máximo de movimientos realizados por una máquina de Turing con las características anteriores.

Hallar el valor de la función $\Sigma(n)$ es conocido como el problema del castor afanoso. La función $\Sigma(n)$ y la función $\mathsf{S}(n)$ fueron sugeridas por Tibor Rado en 1962. La tabla 1.23 muestra los valores conocidos para las funciones $\Sigma(n)$ y $\mathsf{S}(n)$.

A manera de ejemplo, la máquina de Turing que calcula $\Sigma(3)$ está dada por las siguientes instrucciones:

n	$\Sigma(n)$	$S(n)$
1	1	1
2	4	6
3	6	21
4	13	107
5	$\geq$ 4,098	$\geq$ 47,176,870
6	$\geq$ 95,524,079	$\geq$ 8,690,333,381,690,951

Tabla 1.23: $\Sigma(n)$ y $\mathsf{S}(n)$ para $1 \leq n \leq 6$.

i_1:	q_1	$\square$	|	R	q_2
i_2:	q_1	|	|	L	q_3
i_3:	q_2	$\square$	|	L	q_1
i_4:	q_2	|	|	R	q_2
i_5:	q_3	$\square$	|	L	q_2
i_6:	q_3	|	|	R	*stop*

El número de máquinas de Turing de n estados con las características descritas para el problema del castor afanoso es $(4(n+1))^{2n}$, dado que para cada estado de no parada hay dos posibles transiciones; entonces hay $2n$ transiciones y cada transición tiene: 2 posibilidades de escribir un símbolo $(\square, |)$, dos posibilidades de movimiento (L, R) y $(n+1)$ posibles estados para ir (contando el estado de parada).

Solucionar el problema del castor afanoso para un n en particular es muy difícil por dos razones: la primera es que el espacio de búsqueda es muy extenso ($(4(n+1))^{2n}$ máquinas) y la segunda, algunas de estas máquinas pueden no parar y detectar esto puede ser muy difícil.

Para el caso general, el problema del castor afanoso en imposible de solucionar tal como lo indica el siguiente teorema.

Teorema 1.43. *La función $\Sigma(n)$ no es Turing-computable.*

Demostración. La idea de la demostración es demostrar que si $f(x)$ es una función Turing-computable, existe x_0 tal que $\Sigma(x) > f(x)$ para $x \geq x_0$.

1. Sea $f(x)$ una función Turing-computable.
2. La función $g(x) = \sum_{0 \leq i \leq x} (f(i) + i^2)$ es una función Turing-computable.
3. Sea M_g una máquina de Turing de n estados que calcula la función g. Supongamos que M_g comienza con un bloque de x unos y finaliza con un bloque de $g(x)$ unos, separados al menos por una celda del bloque inicial.

4. Sea M una máquina que realiza tres cosas: (a) Inicialmente escribe x unos a partir de una cinta en blanco, lo cual puede hacerse con x estados; (b) Ejecuta la máquina M_g; por lo tanto en la cinta hay un bloque de x unos y un bloque de $g(x)$ unos, lo cual puede hacerse con $x + n$ estados; (c) Ejecuta de nuevo la máquina M_g, por lo tanto en la cinta hay un bloque de x unos, un bloque de $g(x)$ unos y un bloque de $g(g(x))$ unos, lo cual puede hacerse con $x + 2n$ estados.

5. Sabemos que $\Sigma(x + 2n) \geq x + g(x) + g(g(x))$ porque la máquina que calcula la función Σ debe escribir por lo menos tantos unos como la máquina M.

6. Existe c_1 tal que $x^2 > x + 2n$, para todo $x \geq c_1$ y de la definición de $g(x)$ tenemos que $g(x) \geq x^2$, luego $g(x) > x + 2n$, para todo $x \geq c_1$.

7. De la definición de $g(x)$ tenemos que $g(x) > g(y)$ si $x > y$, luego $g(g(x)) > g(x + 2n)$, para todo $x \geq c_1$.

8. Luego, para todo $x \geq c_1$,

$$\begin{aligned} \Sigma(x + 2n) &\geq x + g(x) + g(g(x)) \\ &> g(g(x)) \\ &> g(x + 2n) \\ &> f(x + 2n). \end{aligned}$$

9. Dado que la función f es arbitraria y la función Σ eventualmente es mayor que la función f, se sigue que la función $\Sigma(n)$ no es Turing-computable. □

Como consecuencia del teorema anterior, se sigue que la función $\mathsf{S}(n)$ no es Turing-computable.

Teorema 1.44. *La función $\mathsf{S}(n)$ no es Turing-computable.*

Demostración.

1. Sea M_Σ una máquina de Turing de n estados que escribe $\Sigma(n)$ unos antes de detenerse.

2. La máquina M_Σ debe realizar por lo menos $\Sigma(n)$ movimientos.

3. Entonces $\mathsf{S}(n) \geq \Sigma(n)$.

4. Como $\Sigma(n)$ es eventualmente más grande que cualquier función Turing-computable, entonces $\mathsf{S}(n)$ también lo es.

5. Luego, $\mathsf{S}(n)$ no es una función Turing-computable. □

1.8. Ejercicios

Ejercicio 1.1. Sea $\Sigma = \{a, b, c, d\}$ un alfabeto y α una palabra de Σ^* sobre la cinta. Diseñe una máquina de Turing, tal que:

1. Elimine la palabra α sobre la cinta.
2. Elimine todos los símbolos a de la palabra α.
3. Elimine todos los símbolos a y el segundo símbolo b de la palabra α.
4. Elimine todos los símbolos a y el último símbolo b de la palabra α.
5. Invierta la palabra α.

Ejercicio 1.2. Dado un símbolo $s \in \Sigma$ sobre la cinta. Diseñe una máquina de Turing que encuentre éste, sin importar en qué posición inicial sobre la cinta, comienza su ejecución.

Ejercicio 1.3. Dos máquinas de Turing son equivalentes si para la misma entrada α sobre la cinta, se obtiene la misma salida β sobre la cinta. Demuestre que para toda máquina de Turing MT existe una máquina de Turing equivalente MT' que nunca visita las celdas a la izquierda de la celda desde la cual comienza la máquina MT su ejecución.

Ejercicio 1.4. Diseñe máquinas de Turing que realicen las conversiones necesarias entre los códigos binario, decimal y "palitos".

Ejercicio 1.5. Sean $x_1, x_2 \in \mathbb{N}$, demuestre que la función $f(x_1, x_2) = x_1 + x_2$ es una función Turing-computable.

Ejercicio 1.6. Diseñe una máquina de Turing MT tal que $f_{\mathsf{MT}} = f$, donde la función f está dada por:

1. $f(x_1, x_2, x_3) = x_2$.
2. $f(x_1, x_2, x_3, x_4) = x_4$.
3. $f(x) = 2x$.
4. $f(x, y) = \mathsf{máx}(x, y)$.
5. $f(x, y) = \mathsf{mín}(x, y)$.
6. $f(x, y) = \mathsf{mcd}(x, y)$ (use el algoritmo de Euclides para el mcd).
7. $f(x, y) = xy$.

Ejercicio 1.7. Traducir la tabla 1.12 a la "nomenclatura de instrucciones".

Ejercicio 1.8. Realizar la expansión de la m-función $\mathbf{PE_2}(S, \alpha, \beta)$ presentada en el ejemplo 1.29.

Ejercicio 1.9. Construya las siguientes m-funciones:

1. $\mathbf{L}(S)$: Se desplaza una celda a la izquierda y pasa al estado S.
2. $\mathbf{F}'(S, B, \alpha)$: Similar a la m-función $\mathbf{F}(S, B, \alpha)$, pero antes de pasar al estado S se desplaza a la izquierda.
3. $\mathbf{CP}(S, A, C, \alpha, \beta, \gamma)$: A partir de un símbolo inicial a la izquierda γ, compara las dos primeras ocurrencias de los símbolos α y β. Si alguno de los dos símbolos no está, pasa al estado C; si los símbolos son diferentes, pasa al estado A y si los dos símbolos son iguales, pasa al estado S.

Ejercicio 1.10. Decimos que una máquina de Turing acepta una palabra α si comenzando con α sobre la cinta, la máquina se detiene con la cinta en blanco. Construya una máquina de Turing que acepte únicamente palabras de la forma 10, 1100, 111000, ..., 1^n0^n, ..., para $n \geq 1$.

Ejercicio 1.11. Proponga una definición de una máquina de Turing que satisfaga las siguientes condiciones:

1. No determinística.
2. Probabilística.
3. Que opere sobre dos cintas.
4. Que opere sobre n cintas.
5. Que opere con dos cabezas de lectura-escritura.
6. Que opere con n cabezas de lectura-escritura.
7. 2-dimensional.
8. n-dimensional.

1.9. Notas bibliográficas

El artículo fundacional sobre las máquinas de Turing es [Turing 1936-1937]. Por su parte, Davis [1982], Hopcroft [1984], Kleene [1974], Mendelson [2015], Minsky [1967], Penrose [1991], Penrose [1996], Sicard Ramírez [1996], Sieg [1997] y Soare [1996], entre otros, ofrecen algunos elementos contextuales para la noción de máquina de Turing. La definición para las

funciones Turing-computables fue adaptada de [Davis 1982]. La codificación de las máquinas de Turing y la definición de m-función fueron adaptadas de [Sicard Ramírez 1996], el cual a su vez es una adaptación de [Turing 1936-1937]. Con relación a la máquina universal de Turing, Penrose [1991] presenta una construcción alternativa a la propuesta por Turing y Sicard [1997] ofrece algunos elementos para construir ésta, a partir de las m-funciones propuestas por Turing. La situación presentada como antesala al problema de la parada fue tomada de [Davis 1982]. El problema del castor afanoso, presentado en el ejemplo 1.42 fue tomado de [Shallit 1998; Marxen y Buntrock 1990].

Capítulo 2

Recursividad

En esta sección consagraremos nuestros esfuerzos al desarrollo de una teoría intuitiva, teoría que construiremos para una cierta clase de predicados y funciones de la teoría de números. El concepto de función recursiva primitiva, los conceptos de función parcial y función recursiva general o función μ-recursiva, así como las herramientas teóricas que presentaremos en esta sección, constituyen, a bien decir, elementos de mucha importancia tanto en la lógica como en la informática teórica.

En los inicios de la década de los treinta, en busca de la noción de procedimiento efectivo, se introduce la teoría de las funciones recursivas. En ese entonces se hace de la noción de función recursiva general un eje central. En particular Kurt Gödel utiliza una subclase específica de esta noción (las funciones primitivas recursivas) para codificar la teoría de números; Gödel constituye esta noción como estrategia clave para la construcción de la demostración de su teorema de incompletitud.

Anotemos inicialmente que para infinitas funciones numérico-teóricas no es posible hallar una expresión algebraica que defina su regla de asociación. Es justamente este aspecto el que le confiere a la recursión toda su potencia métodica.

2.1. Funciones y relaciones numérico-teóricas

Definición 2.1 (Relaciones numérico-teóricas)**.** Sea $R(x_1, x_2, \ldots, x_n)$ una relación n-ádica tal que los argumentos o evaluaciones de $x_1, x_2, \ldots, x_n$ sean números naturales. Tal relación la denominaremos relación numérico-teórica, es decir, R es una relación numérico-teórica, si y sólo si, $R \subseteq \mathbb{N}^n$, con $n \geq 1$.

Ejemplo 2.2. Las siguientes son relaciones numérico-teóricas:

$$R_1 = \left\{ (x, y, z) \mid x^2 - y = z, \text{ para } x, y, z \in \mathbb{N} \right\},$$
$$R_2 = \left\{ (x_1, x_2, x_3, x_4, x_5) \in \mathbb{N}^5 \mid x_1 - x_3 = 0 \text{ y } 2x_4 = x_5 \right\}.$$

Definición 2.3 (Funciones numérico-teóricas)**.** Una función numérico-teórica es una función definida en $\mathbb{N}^n$ y evaluada en $\mathbb{N}$, es decir, f es una función numérico-teórica, si y sólo si, $f : \mathbb{N}^n \to \mathbb{N}$. La clase de las funciones numérico-teóricas F, la definimos como:

$$\mathsf{F} = \bigcup_{n<\mathbb{N}} \mathsf{F}_n,$$

donde

$$\begin{aligned}
\mathsf{F}_0 &= \{\, f \mid f : \mathbb{N} \to \mathbb{N} \,\}, \\
\mathsf{F}_1 &= \left\{\, f \mid f : \mathbb{N}^2 \to \mathbb{N} \,\right\}, \\
&\vdots \\
\mathsf{F}_n &= \left\{\, f \mid f : \mathbb{N}^{n+1} \to \mathbb{N} \,\right\}, \\
&\vdots
\end{aligned}$$

Ejemplo 2.4. Las siguientes funciones, son funciones numérico-teóricas:

1. $f : \mathbb{N}^3 \to \mathbb{N}$, donde $f(x, y, z) = 2x + y + z$.
2. $f : \mathbb{N}^4 \to \mathbb{N}$, donde $f(x_1, x_2, x_3, x_4) = (x_4)^2 - x_1$.

2.2. Funciones primitivas recursivas

Nuestro objetivo en esta sección estará centrado en la construcción de una subclase de funciones numérico-teóricas que llamaremos funciones primitivas recursivas (FPR).

Antes de introducirnos en la teoría de la sección, es útil tener encuenta que para definir una función o un predicado por recursión, procedemos mediante el siguiente esquema: digamos que para f,

$$\begin{aligned}
f(0) &= k, \\
f(n+1) &= g(n, f(n)),
\end{aligned}$$

donde, k es un número natural fijo y g es una función numérico-teórica.

Siguiendo el procedimiento de definición por construcción, veremos que las funciones que deseamos reconocer como primitivas recursivas, son justamente aquellas que constructivamente generamos mediante el uso de esquemas de construcción aplicados, un número finito de veces, sobre un conjunto de funciones de base (o axiomas) que, de entrada, declaramos como FPR.

Para definir, constructivamente, el conjunto de las FPR emplearemos el siguiente procedimiento.

Inicialmente definimos un conjunto de axiomas o condiciones límite. Las siguientes funciones, llamadas funciones de base, serán nuestras condiciones límite. Por ello las declaramos axiomáticamente como FPR.

- $\mathsf{z}(x) = 0$, la función cero.
- $\mathsf{s}(x) = x + 1$, la función sucesor.
- $\mathsf{I}_k^n(x_1, x_2, \ldots, x_n) = x_k$, el esquema de funciones k-ésima proyección.

Después dotamos al sistema de dos esquemas de construcción, con el propósito de construir nuevas funciones a partir de funciones ya construidas. Los esquemas de construcción son:

- Esquema de composición o de sustitución.

  Este esquema lo definimos mediante la siguiente expresión:

  $$f(x_1, x_2, \ldots, x_n) = g(h_1(x_1, x_2, \ldots, x_n), \ldots, h_m(x_1, x_2, \ldots, x_n)),$$

  donde, $g : \mathbb{N}^m \to \mathbb{N}$ y cada $h_i : \mathbb{N}^n \to \mathbb{N}$ son funciones bien definidas.

  Es decir, si la función $g(x_1, x_2, \ldots, x_m)$ y cada función $h_i(x_1, x_2, \ldots, x_n)$ son FPR, entonces, la función $f(x_1, x_2, \ldots, x_n)$, definida por el esquema anterior, es una FPR.

  El valor de f, para una interpretación de $x_1, x_2, \ldots, x_n$, se obtiene evaluando a g en los valores obtenidos para las h_i con la interpretación de las x_i.

  Si una función proviene de la aplicación del esquema de composición, decimos que f está definida por composición de las funciones g y $h_1, h_2, \ldots, h_m$.

- Esquema de recurrencia primitiva.

  Este esquema lo definimos mediante la siguiente expresión:

  $$\begin{aligned} f(x_1, x_2, \ldots, x_n, 0) &= g(x_1, x_2, \ldots, x_n), \\ f(x_1, x_2, \ldots, x_n, y+1) &= h(x_1, x_2, \ldots, x_n, y, f(x_1, x_2, \ldots, x_n, y)), \end{aligned}$$

  donde, $g : \mathbb{N}^n \to \mathbb{N}$ y $h : \mathbb{N}^{n+2} \to \mathbb{N}$ son funciones bien construidas. Es decir, si las funciones $g(x_1, x_2, \ldots, x_n)$ y $h(x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2})$ son FPR, entonces, la función $f(x_1, x_2, \ldots, x_n, y)$, definida por el esquema anterior, es una FPR.

  En este caso, diremos que f está definida o construida por recurrencia primitiva, mediante las funciones g y h.

  Si no existe ambigüedad respecto al contexto de trabajo, simplificaremos la notación introducciendo el símbolo $\vec{x}_n$ en lugar de la n-tupla $x_1, x_2, \ldots, x_n$; así, el esquema de recurrencia primitiva, quedaría expresado por:

  $$\begin{aligned} f(\vec{x}_n, 0) &= g(\vec{x}_n), \\ f(\vec{x}_n, y+1) &= h(\vec{x}_n, y, f(\vec{x}_n, y)). \end{aligned}$$

Definición 2.5 (Función primitiva recursiva (FPR))**.** Una función numérico-teórica f se dice primitiva recursiva, si y sólo si f es una función de base, o, f se puede obtener a partir de las funciones de base mediante la aplicación de un número finito de veces del esquema de composición y/o del esquema de definición por recurrencia primitiva. Es decir, f es FPR si y sólo si existe una sucesión de funciones $f_1, f_2, \ldots, f_n$ tales que:

1. $f_n = f$.
2. Para cada $i \leq n$, f_i es una función de base, o f_i se obtiene de sucesiones anteriores, mediante aplicación finita del esquema de composición y/o del esquema de defínción por recurrencia primitiva.

2.3. Construcción de funciones recursivas primitivas

En esta sección presentaremos una serie de funciones recursivas primitivas que ilustran el proceso de construcción de la teoría y que constituyen, por asi, decirlo, las herramientas básicas de la teoría.

1. Las siguientes funciones son FPR
   a) $\mathsf{id}(x) = x$, ya que, $\mathsf{id}(x) = \mathsf{I}_1^1(x)$.
   b) $\mathsf{c}_k(x) = k$, ya que, $\mathsf{c}_k(x) = \underbrace{\mathsf{s}(\mathsf{s}(\mathsf{s}(\ldots(\mathsf{z}(x)))))}_{k \text{ veces}}$.
   c) $f(x) = x + 2$, ya que, $f(x) = \mathsf{s}(\mathsf{s}(x))$.
2. La adición, $\mathsf{add}(x, y) = x + y$

   Si deseamos adicionarle al número x el número y, podemos pensarlo como

   $$(\ldots(((x + 0) + 1) + 1) + \cdots + 1),$$

   lo que nos sugiere una construcción por recursividad. Así:

   $$\begin{aligned} \mathsf{add}(x, 0) &= x, \\ \mathsf{add}(x, y + 1) &= \mathsf{add}(\mathsf{add}(x, y), 1). \end{aligned}$$

   Para llevarla al esquema de definición por recurrencia primitiva, lo expresamos por:

   $$\begin{aligned} \mathsf{add}(x, 0) &= \mathsf{I}_1^1(x), \\ \mathsf{add}(x, y + 1) &= \mathsf{s}\left(\mathsf{I}_3^3(x, y, \mathsf{add}(x, y))\right), \end{aligned}$$

   la función add, así definida, es FPR, puesto que queda definida por el esquema de definición por recurrencia primitiva aplicado sobre funciones primitivas recursivas.

3. Multiplicación y potenciación

   Si observamos los primeros y segundos miembros de las ecuaciones dadas a continuación, podremos constatar que las funciones son FPR, ya que están construidas por recurrencia primitiva sobre funciones primitivas recursivas.

   a) $\mathsf{mult}(x, y) = x \cdot y$

   $$\begin{aligned} \mathsf{mult}(x, 0) &= \mathsf{z}(0), \\ \mathsf{mult}(x, y+1) &= h(x, y, \mathsf{mult}(x, y)), \end{aligned}$$

   donde, $h(x, y, z) = \mathsf{add}\left(\mathsf{I}_3^3(x, y, z), \mathsf{I}_1^3(x, y, z)\right)$. Luego $\mathsf{mult}(x, y) = x \cdot y$ es una FPR.

   b) $\mathsf{pow}(x, y) = x^y$

   $$\begin{aligned} \mathsf{pow}(x, 0) &= x^0 = \mathsf{s}(\mathsf{z}(x)), \\ \mathsf{pow}(x, y+1) &= x^{y+1} = x^y \cdot x = h(x, y, \mathsf{pow}(x, y)), \end{aligned}$$

   donde, $h(x, y, z) = \mathsf{mult}\left(\mathsf{I}_3^3(x, y, z), \mathsf{I}_1^3(x, y, z)\right)$. Luego $\mathsf{pow}(x, y) = x^y$ es una FPR.

   Observación 2.6. De acuerdo con esta definición $0^0 = 1$.

4. Función factorial, $f(x) = x!$

   $$\begin{aligned} f(0) &= 0! = 1 = \mathsf{s}(0), \\ f(x+1) &= (x+1)! \\ &= x!(x+1) \\ &= \mathsf{mult}(x+1, f(x)) \\ &= \mathsf{mult}(\mathsf{s}(x), f(x)). \end{aligned}$$

   Entonces, $f(x) = x!$ es FPR. Hemos obviado expresar a $f(x+1)$ mediante una función de tres variables, en aras de la simplificación.

5. Función permutación de variables

   Sea $f(x_1, x_2, \ldots, x_n) = g\left(x_{G(1)}, x_{G(2)}, \ldots, x_{G(n)}\right)$, entonces, si la función g es recursiva primitiva, la función f también lo es.

   Sea $f(x_1, x_2, \ldots, x_n) = g\left(\mathsf{I}_{G(1)}^n, \mathsf{I}_{G(2)}^n, \ldots, \mathsf{I}_{G(n)}^n\right)$. Si por ejemplo,

   $$G = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 2 & 4 \end{pmatrix},$$

   entonces $f(x_1, x_2, x_3, x_4) = g(x_3, x_1, x_2, x_4)$.

6. Función predecesor

   La función predecesor, denotada por pred, asocia con cada número natural su predecesor, salvo con cero al que le asocia cero. Definimos la función pred, por recurrencia primitiva, como sigue:

$$\begin{aligned}\mathsf{pred}(0) &= 0,\\ \mathsf{pred}(x+1) &= \mathsf{I}_1^2(x, \mathsf{pred}(x)).\end{aligned}$$

7. Funciones diferencia truncada y diferencia absoluta

   a) La función diferencia truncada

$$\dot{-}(x,y) = \begin{cases} x-y, & \text{si } x \geq y;\\ 0, & \text{si } x < y.\end{cases}$$

   Podemos expresar esta función mediante el esquema de definición por recurrencia primitiva sobre FPR, así

$$\begin{aligned}\dot{-}(x,0) &= \mathsf{id}(x),\\ \dot{-}(x,y+1) &= \mathsf{pred}\left(\mathsf{I}_3^3(x,y,\dot{-}(x,y))\right).\end{aligned}$$

   Por ejemplo calculemos, $\dot{-}(4,2) = 2$.

$$\begin{aligned}\dot{-}(4,2) &= \dot{-}(4,1+1)\\ &= \mathsf{pred}\left(\mathsf{I}_3^3(4,1,\dot{-}(4,1))\right)\\ &= \mathsf{pred}(\dot{-}(4,1))\\ &= \mathsf{pred}\left(\mathsf{pred}\left(\mathsf{I}_3^3(4,0,\dot{-}(4,0))\right)\right)\\ &= \mathsf{pred}(\mathsf{pred}(\dot{-}(4,0)))\\ &= \mathsf{pred}(\mathsf{pred}(4))\\ &= \mathsf{pred}(3)\\ &= 2.\end{aligned}$$

   b) La función diferencia absoluta

$$|x-y| = \begin{cases} x \dot{-} y, & \text{si } x \geq y;\\ y \dot{-} x, & \text{si } y \geq x.\end{cases}$$

   Observemos que $|x-y|$, es FPR ya que $|x-y| = \mathsf{add}(\dot{-}(x,y), \dot{-}(y,x))$.

8. La función cero test $\overline{\mathsf{sg}}$ y su complementaria, la función cero test inversa sg

   La función cero test está definida por:

$$\overline{\mathsf{sg}}(x) = \begin{cases} 1, & \text{si } x = 0; \\ 0, & \text{si } x \neq 0. \end{cases}$$

   La función $\overline{\mathsf{sg}}$ es recursiva primitiva ya que

$$\begin{aligned} \overline{\mathsf{sg}}(0) &= \mathsf{s}(0), \\ \overline{\mathsf{sg}}(x+1) &= \mathsf{z}\left(\mathsf{I}_1^2(x, \overline{\mathsf{sg}}(x))\right). \end{aligned}$$

   Así, $\overline{\mathsf{sg}}(0) = 1$, $\overline{\mathsf{sg}}(1) = \mathsf{z}(0) = 0$, $\overline{\mathsf{sg}}(2) = \mathsf{z}(1) = 0$.

   La función cero test inversa sg está definida por:

$$\mathsf{sg}(x) = \begin{cases} 1, & \text{si } x \neq 0; \\ 0, & \text{si } x = 0. \end{cases}$$

   La función sg es FPR, porque:

$$\begin{aligned} \mathsf{sg}(x) &= 1 \dot{-} \overline{\mathsf{sg}}(x) \\ &= \mathsf{c}_1(x) \dot{-} \overline{\mathsf{sg}}(x) \end{aligned}$$

9. La función paridad

   La función paridad está definida por

$$\mathsf{par}(x) = \begin{cases} 1, & \text{si } x \text{ es par}; \\ 0, & \text{si } x \text{ es impar}. \end{cases}$$

   La función $\mathsf{par}(x)$ es FPR porque:

$$\begin{aligned} \mathsf{par}(0) &= 1 = \mathsf{c}_1(x), \\ \mathsf{par}(x+1) &= \overline{\mathsf{sg}}\left(\mathsf{I}_2^2(x, \mathsf{par}(x))\right). \end{aligned}$$

2.4. Predicados primitivos recursivos

Definición 2.7 (Predicado primitivo recursivo)**.** Un predicado $P(\vec{x})$ se dice primitivo recursivo (PPR), si y sólo si la función característica de $P(\vec{x})$ es primitiva recursiva. La función C_P, designará la función característica de $P(\vec{x})$, donde,

$$C_P(\vec{x}) = \begin{cases} 0, & \text{si } P(\vec{x}); \\ 1, & \text{si } \neg P(\vec{x}). \end{cases}$$

Ejemplo 2.8. Los siguientes son ejemplos de predicados primitivos recursivos.

1. El predicado $P(x, y)$: $x = y$, es PPR.

$$C_P(x, y) = \mathsf{sg}(|x - y|) = \begin{cases} 0, & \text{si } x = y; \\ 1, & \text{si } x \neq y. \end{cases}$$

2. El predicado: *ser menor que*; $<(x, y)$ o $x < y$, es PPR.

$$C_<(x, y) = \overline{\mathsf{sg}}(y \dot{-} x) = \begin{cases} 0, & \text{si } x < y; \\ 1, & \text{si } x \geq y. \end{cases}$$

Teorema 2.9. *Si $P(\vec{x})$ y $Q(\vec{x})$ son predicados primitivos recursivos, entonces los predicados: $\neg P(\vec{x})$, $P(\vec{x}) \vee Q(\vec{x})$ y $P\vec{x}) \wedge Q(\vec{x})$, son predicados primitivos recursivos.*

Demostración.

$$\begin{aligned} C_{P \vee Q} &= C_P \cdot C_Q, \\ C_{P \wedge Q} &= (C_P + C_Q) \dot{-} C_P C_Q, \\ C_{\neg P} &= 1 \dot{-} C_P. \end{aligned}$$

□

Definición 2.10 (Cuantificadores acotados)**.** Si $P(\vec{x}, y)$ es un predicado de la teoría de números, podemos definir nuevos predicados mediante cuantificación acotada. Para todo entero n tenemos:

$$\begin{aligned} \forall_{y \leq n}(P(\vec{x}, y)) &\stackrel{\text{def}}{=} P(\vec{x}, 0) \wedge P(\vec{x}, 1) \wedge \cdots \wedge P(\vec{x}, n) \\ &\stackrel{\text{def}}{=} \forall y(y \leq n \Rightarrow P(\vec{x}, y)); \end{aligned}$$

$$\begin{aligned} \exists_{y \leq n}(P(\vec{x}, y)) &\stackrel{\text{def}}{=} P(\vec{x}, 0) \vee P(\vec{x}, 1) \vee \cdots \vee P(\vec{x}, n) \\ &\stackrel{\text{def}}{=} \exists y(y \leq n \wedge P(\vec{x}, y)). \end{aligned}$$

Teorema 2.11. *Si $P(\vec{x}, y)$ es un predicado primitivo recursivo y definimos*

$$\begin{aligned} R(\vec{x}) &= \exists_{y \leq n}(P(\vec{x}, y)), \\ S(\vec{x}) &= \forall_{y \leq n}(P(\vec{x}, y)), \end{aligned}$$

entonces $R(\vec{x})$ y $S(\vec{x})$ son predicados primitivos recursivos.

Demostración.

1. $C_P(\vec{x}, y)$ es FPR.

2. Sea $C_R(\vec{x}) = \prod_{i=0}^{n} C_P(\vec{x}, i)$.
3. Sea $C_S(\vec{x}) = \mathsf{sg}\,(\sum_{i=0}^{n} C_P(\vec{x}, i))$.
4. Las funciones $C_R(\vec{x})$ y $C_S(\vec{x})$ son FPR. □

Teorema 2.12. *Sea $P(\vec{x}, y)$ un PPR y sea $f(\vec{x})$ una FPR. Si definimos*

$$R(\vec{x}) = \exists_{y \leq f(\vec{x})}(P(\vec{x}, y)),$$
$$S(\vec{x}) = \forall_{y \leq f(\vec{x})}(P(\vec{x}, y)),$$

entonces $R(\vec{x})$ y $S(\vec{x})$ son predicados primitivos recursivos.

Demostración.

1. $C_P(\vec{x}, y)$ es FPR.
2. Sea $C_R(\vec{x}) = \prod_{i=0}^{f(\vec{x})} C_P(\vec{x}, i)$.
3. C_R es FPR.
4. Sea $C_S(\vec{x}) = \mathsf{sg}\left(\sum_{i=0}^{f(\vec{x})} C_P(\vec{x}, i)\right)$.
5. C_S es FPR. □

2.5. Funciones definidas mediante condiciones

Las funciones definidas por condiciones son bastante familiares en matemáticas. Una instancia posible es la siguiente: Sea $P(x)$ el predicado x *es par*, entonces:

$$g(x) = \begin{cases} 2x, & \text{si } P(x); \\ 3x & \text{si } \neg P(x). \end{cases}$$

Observemos que si expresamos a $\neg P(x)$ como $P_1(x)$, entonces, $P(x)$ y $P_1(x)$ definen conjuntos que forman una partición de $\mathbb{N}$. Además, estos dos predicados son PPR. De allí que g está definida mediante condiciones y funciones que son recursivas primitivas. El siguiente teorema generaliza este procedimiento y nos garantiza que la función así definida es recursiva primitiva.

Teorema 2.13. *Si $g_1(\vec{x})$, $g_2(\vec{x})$, ..., $g_m(\vec{x})$; son m funciones FPR, y $P_1(\vec{x})$, $P_2(\vec{x})$, ..., $P_m(\vec{x})$; son m predicados PPR, tales que los predicados $P_1, P_2, \ldots, P_m$ forman una partición de $\mathbb{N}^n$, entonces la siguiente función es recursiva primitiva:*

$$f(\vec{x}) = \begin{cases} g_1(\vec{x}), & \textit{si } P_1(\vec{x}); \\ g_2(\vec{x}), & \textit{si } P_2(\vec{x}); \\ \vdots & \\ g_m(\vec{x}), & \textit{si } P_m(\vec{x}). \end{cases}$$

Demostración.

1. Sea C_{P_i} la función característica del predicado P_i.
2. Cada $\vec{x}$ satisface un y sólo un predicado P_k, para $1 \leq k \leq m$.
3. Luego, $f(\vec{x}) = g_k(\vec{x})\overline{\mathsf{sg}}(C_{P_k}(\vec{x})) = g_k(\vec{x})$, para algún k tal que $P_k(\vec{x})$.
4. Luego, para cualquier $\vec{x} \in \mathbb{N}^n$ se tiene que
$$f(\vec{x}) = \sum_{i=1}^{m} g_i(\vec{x})\overline{\mathsf{sg}}(C_{P_i}(\vec{x})).$$
5. La afirmación de que f es FPR está sustentada en el teorema siguiente. □

El siguiente teorema nos proporciona herramientas para construir FPR mediante iteraciones acotadas.

Teorema 2.14. *Si $g(\vec{x}, y)$ es una FPR, entonces las siguientes funciones son FPR.*

1. $f(\vec{x}, y) = \sum_{i=0}^{y} g(\vec{x}, i)$.
2. $h(\vec{x}, y) = \prod_{i=0}^{y} g(\vec{x}, i)$.

Demostración.

1. Función $f(\vec{x}, y)$
   a) $f(\vec{x}, 0) = g(\vec{x}, 0)$.
   b) $f(\vec{x}, n+1) = \sum_{i=0}^{n+1} g(\vec{x}, i) = g(\vec{x}, n+1) + \sum_{i=0}^{n} g(\vec{x}, i)$.
   c) $f(\vec{x}, n+1) = \mathsf{add}(g(\vec{x}, n+1), f(\vec{x}, n))$.
   d) f es FPR.
2. Función $h(\vec{x}, y)$
   Ejercicio 2.11. □

2.6. Funciones recursivas

Definición 2.15 (Función regular)**.** La función $g(\vec{x}, y)$ es una función regular si g verifica la condición: $\forall \vec{x}\, \exists y(g(\vec{x}, y) = 0)$.

Ejemplo 2.16. Sea $P(y)$ el predicado: *y es un número primo*. Sea $f(x, y)$ la siguiente función:
$$f(x, y) = \begin{cases} 0, & \text{si } y > x \text{ y } P(y); \\ 1, & \text{si } y \leq x \text{ o } \neg P(y). \end{cases}$$

Verifiquemos que $f(x, y)$ es regular.

1. $\forall x(x \in \mathbb{N} \Rightarrow \exists y(y \in \mathbb{N} \wedge y > x))$.
2. Dado un $x_0 \in \mathbb{N}$, existen infinitos números primos y, tales que $y > x_0$.
3. Luego, $\forall x \exists y(f(x, y) = 0)$.

Ejemplo 2.17. La función $f(x, y) = x \dot{-} y$ es regular, ya que:

1. $\forall x(x \in \mathbb{N} \Rightarrow \exists y(y \in \mathbb{N} \wedge y \geq x))$.
2. Dado un $x_0 \in \mathbb{N}$, existen infinitos y_0, tales que $x_0 \dot{-} y_0$.
3. Luego, $\forall x \exists y(f(x, y) = 0)$.

Definición 2.18 (Operador de minimalización μ)**.** Si $f(\vec{x}, y)$ es una función, entonces definimos el operador μ (no restringido) por: $g(\vec{x}) = \mu_y(f(\vec{x}, y) = 0)$, que leemos: g asocia a $\vec{x}$, el *menor* y tal que $f(\vec{x}, y) = 0$.

Si la función $f(\vec{x}, y)$ es una función regular, entonces la función $g(\vec{x})$ es una función total, pero si $f(\vec{x}, y)$ no es una función regular, entonces el operador de minimalización $\mu_y(f(\vec{x}, y) = 0)$ no está definido para toda $\vec{x}$, y por lo tanto, $g(\vec{x})$ es una función parcial.

Ejemplo 2.19.

1. La función $f(x, y, z) = (x \dot{-} z) + (y \dot{-} z)$ es regular para la variable z (ejercicio 2.13).
2. Podemos construir una función $g(x, y)$ por minimalización como sigue: $g(x, y) = \mu_z(f(x, y, z) = 0)$.
3. Obsérvese que $g(x, y) = \mathsf{máx}(x, y)$.

Ejemplo 2.20. Dada la función regular $f(x, y) = x \dot{-} y$, podemos definir, por minimalización (operador μ), la función $g(x) = \mu_z(f(x, z) = 0)$.

Para construir la clase de las funciones recursivas totales (llamadas funciones recursivas), introduciremos un tercer esquema de contrucción de funciones (mediante el operador μ, o sea, mediante la operación de minimalización).

Definición 2.21 (Funciones recursivas)**.** La clase de las funciones recursivas es la menor clase de funciones que contiene a las funciones de base y que es estable bajo las operaciones:

R1. Composición.

R2. Recurrencia primitiva.

R3. El operador μ (es decir bajo la operación de minimalización), esto es,

$$f(\vec{x}) = \mu_y(g(\vec{x}, y) = 0), \text{ si } g \text{ es regular.}$$

Observemos que para obtener la clase de funciones recursivas hemos añadido a la definición de las FPR el esquema de minimalización.

En otras palabras, una función $f(\vec{x})$ es recursiva si y sólo si:

1. Es una función de base, o
2. Puede producirse a partir de la clase de funciones de base mediante un número finito de aplicaciones de los esquemas de composición y/o de recurrencia primitiva y/o de minimalización.

Para logra mayor claridad indicaremos el esquema de minimalización, sobre todo si tenemos en cuenta el dicho esquema u operación es quien introduce un corte de distinción entre las FPR y aquellas funciones que son recursivas pero que no son primitivas recursivas.

Esquema de minimalización: sea $f(\vec{x}, y)$ una función recursiva y regular. Entonces la función,

$$g(\vec{x}) = \mu_y(f(\vec{x}, y) = 0),$$

es una función recursiva.

Ejemplo 2.22. La función $\mathsf{máx}(x, y)$ es recursiva.

1. $f(x, y, z) = (x \dot{-} z) + (y \dot{-} z)$ es una función recursiva.
2. $f(x, y, z)$ es una función recursiva (cuando $z = \mathsf{máx}(x, y)$, $f(x, y, z) = 0$).
3. $h(x, y) = \mu_z(f(x, y, z) = 0)$.
4. Pero, $h(x, y) = \mathsf{máx}(x, y)$. Luego, $\mathsf{máx}(x, y)$ es recursiva.

Observemos que al minimalizar una función, eventualmente se puede obtener una FPR. Así, $\mathsf{máx}(x, y) = (x \dot{-} y) + y$, la cual es una FPR.

De acuedo con la distinción que introduce el operador μ, sabemos que existen funciones recursivas que no son funciones primitivas recursivas. Presentamos sin demostración un ejemplo de una de tales funciones.

Ejemplo 2.23. La función de Ackermann, denotada por $\mathsf{ack}(x, y)$, es una función recursiva que no es una función primitiva recursiva. La función está definida por:

$$\begin{aligned} \mathsf{ack}(0, y) &= y + 1, \\ \mathsf{ack}(x + 1, 0) &= \mathsf{ack}(x, 1), \\ \mathsf{ack}(x + 1, y + 1) &= \mathsf{ack}(x, \mathsf{ack}(x + 1, y)). \end{aligned}$$

Como ejemplo calculemos $\mathsf{ack}(2, 3)$ en el apéndice A.

Aunque no es difícil implementar la función de Ackermann, debido a su rápido crecimiento, es imposible calcularla para valores relativamente pequeños de x y de y. Así por ejemplo, $\mathsf{ack}(4, 2)$ es un número con 19,728 dígitos. Por otra parte, parece ser que el número de llamadas a la función de Ackermann para calcular un valor inicial, crece más rápido que la misma función, tal como lo ilustra la tabla 2.1.

$\mathsf{ack}(x, y)$	Valor	Número de llamadas
$\mathsf{ack}(3, 0)$	5	15
$\mathsf{ack}(3, 1)$	13	106
$\mathsf{ack}(3, 2)$	29	541
$\mathsf{ack}(3, 3)$	61	2,432
$\mathsf{ack}(3, 4)$	125	10,307
$\mathsf{ack}(3, 5)$	253	42,438
$\mathsf{ack}(3, 6)$	509	172,233
$\mathsf{ack}(3, 7)$	1,021	693,964
$\mathsf{ack}(3, 8)$	2,045	2,785,999

Tabla 2.1: Número de llamadas a la función de Ackermann.

2.7. Funciones recursivas parciales

En el contexto de las funciones recursivas es necesario también que hagamos la distinción entre función total (definición 1.4) y función parcial (definición 1.5). De hecho, ya hemos trabajado con las funciones parciales en el contexto de las máquinas de Turing, donde permitimos que las reglas de construcción de la máquina nos condujeran a un computar que nunca termina, o en otros términos, la función asociada con la máquina queda indefinida para algunas entradas, esto es, no originando así ningun valor de salida o imagen de la función. El interés de nuestra anterior distinción reside en el hecho de que podemos transladarla al contexto de las funciones recursivas.

Consideremos el caso de una función recursiva $h(x, y)$ que sea una función total, pero que no sea necesariamente una función regular. Si aplicamos el operador de minimalización a la función h, entonces, para ciertos valores de x, la función generada por minimalización no estará definida, y por ende la función h será una función parcial. Es decir, la función $f(x) = \mu_y(h(x, y) = 0)$ no estará definida para toda $x \in \mathbb{N}$. La función $f(x)$ la podemos expresar como sigue:

$$f(x) = \begin{cases} \mu_y(h(x, y) = 0), & \text{si un tal } y \text{ existe;} \\ \text{indefinida}, & \text{si no existe tal } y. \end{cases}$$

Ejemplo 2.24.

1. $f(x, y) = (x + y) \dot{-} 17$, es una función recursiva total y no regular.

2. $g(x) = \mu_y(f(x, y) = 0)$, es una función recursiva parcial.

3. Observamos que para $x \geq 18$, tendríamos $g(x)$ indefinida. Por ejemplo, para $x = 18$, $g(18) =$ indefinida, ya que, $(18 + 0) \dot{-} 17 \neq 0$, $(18 + 1) \dot{-} 17 \neq 0$, etc.

Entonces diremos que al aplicar el operador de minimalización a una función recursiva total, obtendremos, en general, una función recursiva parcial.

Definición 2.25 (Funciones recursivas parciales)**.** La clase de las funciones recursivas parciales es la menor clase de funciones que contiene a las funciones de base y que es estable bajo las operaciones composición, recurrencia primitiva y el operador μ (es decir bajo la operación de minimalización $f(\vec{x}) = \mu_y(g(\vec{x}, y) = 0)$, si tal y existe, de lo contrario, indefinida).

Cerremos la presente sección estableciendo la siguiente cadena de inclusiones propias entre las diversas clases de funciones que hemos estudiado. Denotemos por F la clase de funciones numérico-teóricas, por F_0 la clase de funciones recursivas parciales, por F_1 la clase de funciones recursivas y por F_2 la clase de funciones primitivas recursivas. Entonces se puede establer la cadena $\mathsf{F} \supset \mathsf{F}_0 \supset \mathsf{F}_1 \supset \mathsf{F}_2$. La relaciones de inclusión en general se deducen del hecho mismo de las definiciones de cada clase. Ahora:

1. $\mathsf{F}_2 \subset \mathsf{F}_1$, puesto que existen funciones recursivas que no son primitivas recursivas. Es decir, existen funciones computables que no son primitivas recursivas. Para ello, basta probar que si $f_0, f_1, \ldots, f_k, \ldots$ es una enumeración de F_2, entonces la función $g(n) = f_n(n)$ no está en la clase F_2, luego no es FPR.

2. $\mathsf{F}_1 \subset \mathsf{F}_0$, es decir, toda función recursiva es recursiva parcial, pero existen funciones recursivas parciales que non recursivas totales. En general, existen funciones recursivas parciales, estrictamente parciales que no pueden llevarse a una función recursiva total. Si se considera, por ejemplo, una enumeración de todas las funciones recursivas parciales de una variable: $g_0, g_1, \ldots$, entonces la función $h(n) = g_n(n)$ (para el caso en que g_n este definida, e indefinida cuando no lo esté) no puede ser llevada a una función recursiva total.

3. $\mathsf{F}_0 \subset \mathsf{F}$. Un argumento de cardinalidad lo prueba inmediatamente, ya que $\overline{\overline{\mathsf{F}_0}} = \aleph_0$ y $\overline{\overline{\mathsf{F}}} > \aleph_0$.

2.8. Funciones definidas por minimalización acotada

Definición 2.26 (Operador de minimalización acotada)**.** Cuando acotamos el alcance del operador de minimalización (μ) obtenemos una nueva operación que llamaremos minimalización acotada. Escribimos, para un entero n y una función $g(\vec{x}, y)$ la expresión $\mu_{y \leq n}(g(\vec{x}, y) = 0)$, para señalar o simbolizar al menor y tal que sea menor o igual que n, y tal que $g(\vec{x}, y) = 0$.

Sin que $g(\vec{x}, y)$ sea necesariamente una función regular, podemos siempre definir el

operador de minimalización acotada, así:

$$h(\vec{x}) = \mu_{y\leq n}(g(\vec{x},y)=0) = \begin{cases} \text{al menor } y \leq n, \text{ tal que } g(\vec{x},y)=0, & \text{si tal } y \text{ existe,} \\ 0, & \text{si no existe tal } y. \end{cases}$$

Si la función g es recursiva, entonces para un n especificado se calcula:

1. $g(\vec{x},0)$, $g(\vec{x},1)$, $\ldots$, $g(\vec{x},n)$.
2. Si para algún $k \leq n$, $g(\vec{x},k)=0$, entonces, $\mu_{y\leq n}(g(\vec{x},y)=0)=k$, donde k fue el primer número tal que $g(\vec{x},k)=0$.
3. Si no existe ningún cero en la sucesión $g(\vec{x},0)$, $g(\vec{x},1)$, $\ldots$, $g(\vec{x},n)$, entonces, $\mu_{y\leq n}(g(\vec{x},y)=0)=0$.

Teorema 2.27. *Si $g(\vec{x},y)$ es una FPR, entonces la función $f(\vec{x})$ definida por: $f(\vec{x}) = \mu_{y\leq m}(g(\vec{x},y)=0)$, es una FPR.*

Demostración.

1. Para demostrar este resultado introducimos la función $h(\vec{x},m)$ definida por el siguiente esquema de recursión:

$$\begin{aligned} h(\vec{x},0) &= 0; \\ h(\vec{x},m+1) &= \begin{cases} h(\vec{x},m), & \text{si } h(\vec{x},m)\neq 0; \\ m+1; & \text{si } h(\vec{x},m)=0 \text{ y } g(\vec{x},m+1)=0; \\ 0, & \text{si } h(\vec{x},m)=0 \text{ y } g(\vec{x},m+1)\neq 0. \end{cases} \end{aligned}$$

2. Si utilizamos la función de identidad $\mathsf{id}(x) = \mathsf{I}^1_1(x)$, es posible demostrar que la función $h(\vec{x},m)$ es FPR.
3. Se puede verificar que
$$f(\vec{x}) = h(\vec{x},m),$$
por lo tanto, $f(\vec{x})$ es FPR. □

Teorema 2.28. *Si $g(\vec{x},y)$ y $h(\vec{x})$ son FPR, entonces la función $f(\vec{x})$ definida por: $f(\vec{x}) = \mu_{y\leq h(\vec{x})}(g(\vec{x},y)=0)$, es una FPR.*

Demostración.

1. Se prueba que $j(\vec{x},n) = \mu_{y\leq n}(g(\vec{x},y)=0)$ es FPR.
2. Hacemos $f(\vec{x}) = j(\vec{x},n)$. □

Ejemplo 2.29. La función $p(n) = (n+1)$-énesimo primo es FPR.

1. Definimos la función $p(n)$ como sigue:

$$\begin{aligned} p(0) &= 2, \\ p(n+1) &= \mu_{y \leq h(n,p(n))}[y > p(n) \text{ y } p_1(y)]. \end{aligned}$$

2. Donde $h(x,k) = (x+1)^{k+1}$ es FPR.

3. Donde $p_1(y)$ es el predicado: "y es un número primo".

4. La función

$$f(x,y) = \begin{cases} 0, & \text{si } y > p(x) \text{ y } p_1(y); \\ 1, & \text{si no es el caso,} \end{cases}$$

es FPR.

5. Luego, $p(n+1) = \mu_{y \leq h(n,p(n))}(f(n,y) = 0)$ es FPR.

2.9. Conjuntos recursivos

La noción de recursividad puede aplicarse igualmente a los conjuntos y, especificamente a conjuntos numéricos. La importancia de la noción de conjunto recursivo está vinculada a los problemas de decidibilidad. Presentaremos tal noción en el contexto de subconjuntos de $\mathbb{N}^n$.

Definición 2.30 (Conjunto (primitivo) recursivo)**.** Dado $A \subseteq \mathbb{N}^n$, diremos que el conjunto A es (primitivo) recursivo, si y sólo si, existe una función (primitiva) recursiva f tal que:

$$f(x_1, x_2, \ldots, x_n) = \begin{cases} 0, & \text{si } (x_1, x_2, \ldots, x_n) \in A; \\ 1, & \text{si } (x_1, x_2, \ldots, x_n) \notin A. \end{cases}$$

Esto es, la función caractecterística de la relación: "$\in A$", es una función (primitiva) recursiva. Notemos que la función f decide la pertenencia o no de una cierta n-tupla al conjunto A. En este sentido, la función f es la función característica del conjunto A. Por ello decimos con frecuencia que f es la función decisión para el conjunto A.

Ejemplo 2.31. El conjunto $\mathbb{N}^n$ es recursivo. La función $f(\vec{x}) = 0$ es recursiva.

Ejemplo 2.32. Todo conjunto finito A es recursivo (primitivo recursivo). Si $A = \{a_1, \ldots, a_n\}$, entonces la función característica de A es FPR. Así,

$$C_A(x) = \prod_{i=1}^{n} \mathsf{sg}\left(|x - a_i|\right).$$

Ejemplo 2.33. El conjunto de los números impares es recursivo. Para comprobarlo se demuestra que la función:

$$f(x) = \begin{cases} 0, & \text{si } x \text{ es impar;} \\ 1, & \text{si } x \text{ es par;} \end{cases}$$

es recursiva. Así: $f(0) = 1$ y $f(x+1) = \overline{\mathsf{sg}}\left(\mathsf{I}_2^2(x, f(x))\right)$.

Ejemplo 2.34. El conjunto de los números primos es recursivo. Esto es, si designamos con A el conjunto de los números primos, entonces la función

$$f(n) = \begin{cases} 0, & \text{si } n \text{ es primo,} \\ 1, & \text{si } n \text{ es compuesto;} \end{cases}$$

es recursiva, lo cual es verdadero ya que $f(n) = |d(n) - \mathsf{s}(\mathsf{s}(\mathsf{z}(n)))|$; donde $d(n)$ es el número de divisores de n, luego f es FPR.

Decíamos que f es una función de decisión. Efectivamente, la noción de conjunto recursivo es una formulación de la noción intuitiva de conjunto decidible. Intuitivamente hablando, decimos que un conjunto A es decidible, si y sólo si, existe un algoritmo o procedimiento α que nos permita decidir acerca de si un determinado elemento x, del universo de referencia de A, pertenece o no al conjunto A.

Ejemplo 2.35. Para el conjunto de los números naturales, si A es el conjunto de los números pares; ¿es n un elemento de A?

Efectivamente, sabemos que existe un algoritmo que proporciona una respuesta. Así, dado cualquier $n \in \mathbb{N}$

1. Divida n por 2.
2. Hallar el residuo r.
3. Si $r = 0$, entonces $n \in A$. Si $r \neq 0$, entonces $n \notin A$.

Entonces, el conjunto $A = \{\, x \mid x \text{ es par} \,\}$ es recursivo.

Ejemplo 2.36. El conjunto P de los números primos es decidible. Es decir, dado un $n \in \mathbb{N}$, existe un algoritmo α que proporciona una respuesta a la pregunta ¿es n un número primo?

Ciertamente, sabemos que la aritmética elemental contiene procedimientos o algoritmos que nos permiten decidir si un número natural dado es primo o compuesto (mirar ejercicio 2.36). Por ejemplo:

1. Hallar todos los $n_i \in \mathbb{N}$ tales que $n_i < n$ y $n_i \neq 1$.
2. Dividir a n por cada n_i y escribir el residuo respectivo r_i.

3. Si todos los $r_i \neq 0$, entonces $n \in P$, de lo contrario, $n \notin P$.

Por supuesto que existen algoritmos en la aritmética mucho más eficientes que el anterior. El conjunto P de los números primos es recursivo como lo probamos en el ejemplo 2.34.

Ejemplo 2.37. El conjunto de las fórmulas de la lógica de enunciados es un conjunto decidible. Efectivamente, existe un algoritmo que nos permite decidir si una palabra $\alpha \in L(\Sigma)$ es o no es una fórmula (ejercicio 2.18).

Como puede observarse, la noción de conjunto recursivo se instituye como una tentativa de formalización de la noción intuitiva de conjunto decidible. Esto es, la definición de conjunto recursivo corresponde con la noción intuitiva de conjunto decidible.

2.10. Conjuntos recursivamente enumerables

Intuitivamente pensamos un conjunto enumerable como aquel conjunto cuyos elementos pueden ser escritos o listados como una sucesión infinita (con eventuales repeticiones). Tal noción la podemos formalizar valiéndonos de la noción de aplicación o función sobreyectiva.

Definición 2.38 (Conjunto enumerable)**.** Decimos que un conjunto A es enumerable, si y sólo si, A es vacío o existe una aplicación sobreyectiva de $\mathbb{N}$ en el conjunto A. Esto es,

$$A \text{ es enumerable} \stackrel{\text{def}}{=} (A = \emptyset) \vee (\exists f)(f : \mathbb{N} \longmapsto A).$$

Si A es enumerable, decimos que la función f es una enumeración de A, es decir, f escribe los elementos de A como la sucesión: $f(0), f(1), \ldots, f(n), \ldots$.

Ejemplo 2.39. $\mathbb{N}$ es enumerable. La enumeración es: $\mathsf{id}(x) = x$.

Ejemplo 2.40. El conjunto de los números pares es enumerable. La enumeración es la función sobreyectiva $f(n) = 2n$.

Ejemplo 2.41. Todo conjunto finito es enumerable. Sea $A = \{a_1, a_2, \ldots, a_n\}$ y, sea $f : \mathbb{N} \longmapsto A$, tal que:

$$f(x) = \begin{cases} a_x, & \text{si } x < n; \\ a_n, & \text{si } x \geq n. \end{cases}$$

La función f es una función sobreyectiva de $\mathbb{N}$ en A. Luego A es un conjunto enumerable.

Una pregunta de bastante importancia, la cual nos ocupará en el resto de esta sección, es la siguiente ¿es posible hallar para toda enumeración f de un conjunto A un algoritmo o procedimiento efectivo que nos permita computar $f(n)$, para todo $n \in \mathbb{N}$?

Definición 2.42 (Conjunto recursivamente enumerable)**.** Sea el conjunto A un subconjunto de $\mathbb{N}$. Decimos que A es un conjunto recursivamente enumerable (r.e.) si A es el rango de una función recursiva, o si A es vacío.

La anterior definición implica que si $A \neq \emptyset$ es recursivamente enumerable, entonces existe una enumeración recursiva de A, es decir, existe una función recursiva tal que: $f(0), f(1), \ldots, f(n), \ldots$ es una lista exhaustiva de los elementos de A (con eventuales repeticiones). Esto es, $f(\mathbb{N}) = A$. Expresado de otra manera:

$$A \text{ es r.e.} \stackrel{\text{def}}{=} (A = \emptyset) \vee \Big((\exists f)(f : \mathbb{N} \longmapsto A) \wedge f \text{ es recursiva}\Big).$$

Por ello decimos también que la función f genera el conjunto A.

Ejemplo 2.43. Todo conjunto finito es r.e. Si $A = \{a_1, a_2, \ldots, a_n\}$, entonces, sea la función $f : \mathbb{N} \longmapsto A$, tal que:

$$f(x) = \begin{cases} a_x, & \text{si } x < n; \\ a_n, & \text{si } x \geq n. \end{cases}$$

La función f es una función sobreyectiva y recursiva de $\mathbb{N}$ en A (ejercicio 2.21).

Unas preguntas que podemos formularnos en este contexto son las siguientes: ¿son las nociones de conjunto recursivo y conjunto recursivamente enumerable diferentes?; ¿existe algún conjunto que sea recursivamente enumerable y no sea recursivo, o viceversa?

Teorema 2.44. *Todo conjunto recursivo (subconjunto de $\mathbb{N}$) es recursivamente enumerable.*

Demostración.

1. Supongamos que $A \neq \emptyset$ es un conjunto recursivo (si $A = \emptyset$, A es r.e. por definición).
2. Existe una función recursiva f tal que:

$$f(x) = \begin{cases} 0, & \text{si } x \in A; \\ 1, & \text{si } x \notin A. \end{cases}$$

3. Sean $g : \mathbb{N} \to A$ y $x_0 \in A$ (fijo) tales que:

$$g(x) = \begin{cases} x, & \text{si } x \in A; \\ x_0, & \text{si } x \notin A. \end{cases}$$

4. Los predicados $x \in A$ y $x \notin A$, son recursivos (puesto que f es una función recursiva).
5. Luego, la función g es recursiva (puesto que está definida mediante condiciones recursivas). □

El recíproco del teorema anterior no es válido, es decir, existen conjuntos r.e. que no son recursivos. Pero si adicionamos una condición al complemento del conjunto r.e., éste sí es recursivo.

Teorema 2.45. *Un conjunto A es recursivo, si y sólo si el conjunto A y su complemento son recursivamente enumerables.*

Demostración. (primera parte)

1. Supongamos que A es recursivo.
2. Luego, A es r.e. por el teorema 2.44.
3. Denotemos por A^c el complemento de A.
4. $f(x) = 1 \dot{-} C_A(x)$ es recursiva.
5. A^c es recursivo.
6. Luego, A^c es r.e. por el teorema 2.44. □

Demostración. (segunda parte)

1. Supongamos que A y A^c son r.e.
2. Existen enumeraciones recursivas de A y A^c.
3. Sean éstas respectivamente $f : \mathbb{N} \longmapsto A$ y $g : \mathbb{N} \longmapsto A^c$.
4. Sea $h : \mathbb{N} \to A$ tal que:

$$h(x) = \begin{cases} f\left(\left[\frac{x}{2}\right]\right), & \text{si } x \text{ es par;} \\ g\left(\left[\frac{x}{2}\right]\right), & \text{si } x \text{ es impar.} \end{cases}$$

5. La función h escribe la sucesión $f(0), g(0), f(1), g(1), \ldots$.
6. La función h es recursiva.
7. Definamos la función $k(y, z) = |h(y) - z|$, la cual es regular para la variable z.
8. La función $m(y) = \mu_z(k(y, z) = 0)$ es recursiva.
9. La función p, característica de los números pares es recursiva.
10. Definamos la función C_A, característica del conjunto A, como la función $C_A(x) = p(m(x))$.

11. C_A es una función recursiva.

12. Luego, el conjunto A es recursivo. □

Es conocido que la noción de conjunto recursivamente enumerable es una formalización de la noción intuitiva de conjunto efectivamente enumerable. Intuitivamente hablando, decimos que un conjunto A es efectivamente enumerable, si y sólo si, existe un enumeración f calculable de A. Los valores $f(0), f(1), \ldots$ son determinados mediante un algoritmo o procedimiento efectivo. En otras palabras se trata de considerar la noción intuitiva de función calculable mediante un algoritmo. Como se afirma con frecuencia, el sí una cierta descripción matemática particular de la noción de algoritmo corresponde exactamente a la idea intuitiva, no es algo que se pueda demostrar. No obstante, hay buenos argumentos para suponer que las descripciones matemáticas que se han hecho del concepto de algoritmo, son lo bastante generales como para incluir todos los algoritmos intuitivos. Dos ejemplos de ello nos lo proporcionan la clase de las funciones recursivas y la clase de las funciones Turing-computables. Por otro lado, es bastante razonable afirmar que una función parcial es calculable mediante un algoritmo, si existe uno que determine explícitamente el valor de la función cuando ésta está definida.

Una función $f(x_1, x_2, \ldots, x_n)$ es efectivamente calculable si existe un procedimiento mecánico o algoritmo que permita determinar el valor $f(a_1, a_2, \ldots, a_n)$ para cualquier n-tupla de elementos de $\mathbb{N}$.

Ejemplo 2.46. El conjunto de los números enteros es efectivamente enumerable, puesto que existe una enumeración efectiva de $\mathbb{Z}$, $g : \mathbb{N} \longmapsto \mathbb{Z}$ dada por $g(n) = (-1)^n([\frac{n+1}{2}])$. Esta fórmula es un algoritmo de enumeración de $\mathbb{Z}$.

Lema 2.47. *Todo entero positivo $n > 1$ puede ser factorizado de forma única como $n = P_1^{n_1} \times P_2^{n_2} \times \cdots \times P_m^{n_m}$ donde $P_1 < P_2 < \ldots < P_m$, son los m primeros primos y $n_m \neq 0$. Esta forma de expresar el número n se llama la forma normal de Cantor.*

Demostración. Ejercicio 2.22. □

Teorema 2.48. *El conjunto $\mathbb{N}^f$ de las sucesiones finitas de números naturales es un conjunto efectivamente enumerable.*

Demostración. Para probar el teorema, definamos la función f mediante el siguiente algoritmo.

1. Sea $n \in \mathbb{N}$ un elemento cualquiera.

2. Expresar m como $n + 2$.

3. Expresar m en la forma normal de Cantor (lema 2.47), es decir, $m = P_1^{\alpha_1} \times P_2^{\alpha_2} \times \cdots \times P_k^{\alpha_k}$.

4. Escribir la sucesión: $< \alpha_1, \alpha_2, \ldots, \alpha_k - 1 >$.

5. Sea $f(n) =< \alpha_1, \alpha_2, \ldots, \alpha_k - 1 >$.

6. Para probar que f es una enumeración efectiva exhaustiva, es decir, sobreyectiva, invierta el proceso de escribir la sucesión $< \alpha_1\alpha_2 \ldots \alpha_k - 1 >$. Así, dado $a_1, a_2, \ldots, a_s$, $n = P_1^{a_1} \times P_2^{a_2} \times \cdots \times P_s^{a_s+1} - 2$. □

2.11. Computabilidad y recursividad

Esta sección establece la coexistencia entre el conjunto de las de funciones Turing-computables y el de las funciones recursivas. La coexistencia es presentada por dos teoremas; el primero de ellos indica que las funciones recursivas son funciones Turing-computables; el segundo indica que las funciones Turing-computables son funciones recursivas.

Inicialmente presentaremos seis lemas que serán utilizados en la demostración de que una función recursiva es una función Turing-computable.

Lema 2.49. *La función cero,* $\mathsf{z}(x) = 0$*, es una función Turing-computable.*

Demostración.

1. Sea MT la máquina de Turing presentada en el ejemplo 1.19 que está definida por:

   i_1: q_1 | □ R q_1
   i_2: q_1 □ □ R *stop*

2. Sea α_1 la descripción instantánea inicial definida por $\alpha_1 \equiv q_1 \overrightarrow{x}$. Entonces la computación de la máquina MT está dada por (en donde $|^n$ representa n palitos):

$$\begin{aligned}
\alpha_1 \equiv q_1 \overrightarrow{x} &= q_1|^{x+1} \\
&\to \square q_1|^x \\
&\to \square\square q_1|^{x-1} \\
&\vdots \\
&\to \square\square \ldots \square q_1| \\
&\to \square\square \ldots \square\square q_1.
\end{aligned}$$

3. Por lo tanto, la función $f^{(1)}_{\mathsf{MT}}(x)$ asociada con la máquina de Turing MT está definida por:

$$\begin{aligned}
f^{(1)}_{\mathsf{MT}}(x) &=< \mathsf{Res}_{\mathsf{MT}}(\alpha_1) > \\
&=< \square\square \ldots \square\square q_1 > \\
&= 0 \\
&= \mathsf{z}(x).
\end{aligned}$$

□

Lema 2.50. *La función sucesor,* $\mathsf{s}(x) = x + 1$*, es una función Turing-computable.*

Demostración.

1. Sea MT la máquina de Turing, presentada en el ejemplo 1.20, que está definida por: i_1: $q_1 \square \square N\ q_1$.

2. Sea α_1 la descripción instantánea inicial definida por $\alpha_1 \equiv q_1 \overrightarrow{x}$. Entonces la computación de la máquina MT está dada por (en donde $|^n$ representa n palitos):

$$\begin{aligned} \alpha_1 \equiv q_1 \overrightarrow{x} &= q_1|^{x+1} \\ &\to q_1|^{x+1}. \end{aligned}$$

3. Por lo tanto, la función $f^{(1)}_{\mathsf{MT}}(x)$ asociada con la máquina de Turing MT está definida por:

$$\begin{aligned} f^{(1)}_{\mathsf{MT}}(x) &=< \mathsf{Res}_{\mathsf{MT}}(\alpha_1) > \\ &=< q_1|^{x+1} > \\ &= x + 1 \\ &= \mathsf{s}(x). \end{aligned}$$

□

Lema 2.51. *Las funciones k-ésima proyección,* $\mathsf{I}^n_k(x_1, x_2, \ldots, x_n) = x_k$*, son funciones Turing-computables.*

Demostración. Vamos a construir una máquina de Turing genérica para una función

$$\mathsf{I}^n_k(x_1, x_2, \ldots, x_n) = x_k.$$

1. Sea MT una máquina de Turing definida por:
   $q_1|\square N q_{2n+1}$; estas instrucciones borran el argumento x_1
   $q_1 \square\square R q_2$
   $q_{2n+1}\square\square R q_1$

   $q_2|\square N q_{2n+2}$; estas instrucciones borran el argumento x_2
   $q_2 \square\square R q_3$
   $q_{2n+2}\square\square R q_2$
   $\vdots$
   $q_{k-1}|\square N q_{2n+(k-1)}$; estas instrucciones borran el argumento x_{k-1}
   $q_{k-1}\square\square R q_k$
   $q_{2n+(k-1)}\square\square R q_{k-1}$
   $q_k|\square N q_k$, esta instrucción elimina un palito del argumento x_k

$q_k\square\square Rq_{k+1}$, esta instrucción envía a borrar los argumetos $x_{j>k}$
$q_{k+1}|\square Nq_{2n+(k+1)}$; estas instrucciones borran el argumento x_{k+1}
$q_{k+1}\square\square Rq_{k+2}$
$q_{2n+(k+1)}\square\square Rq_{k+1}$
$\vdots$
$q_n|\square Nq_{2n+n}$; estas instrucciones borran el argumento x_n
$q_n\square\square Rstop$
$q_{2n+n}\square\square Rq_n$

2. De la construcción anterior se concluye que la función $f^{(n)}_{\mathsf{MT}}(x_1, x_2, \ldots, x_n)$ asociada con la máquina de Turing MT es igual a $\mathsf{I}^n_k(x_1, x_2, \ldots, x_n) = x_k$. □

Lema 2.52. *El esquema de composición o sustitución preserva la propiedad de Turing-computabilidad. Es decir, si $g : \mathbb{N}^m \to \mathbb{N}$ y cada $h_i : \mathbb{N}^n \to \mathbb{N}$ son funciones Turing-computables (totales, parciales), entonces*

$$f(x_1, x_2, \ldots, x_n) = g(h_1(x_1, x_2, \ldots, x_n), \ldots, h_m(x_1, x_2, \ldots, x_n)),$$

es una función Turing-computable (total, parcial).

Demostración. La idea es construir una máquina MT_f que envíe a computar cada una de las funciones $h_i(x_1, x_2, \ldots, x_n), 1 \leq i \leq m$ y con los valores obtenidos, envíe a computar la función $g(h_1(x_1, x_2, \ldots, x_n), \ldots, h_m(x_1, x_2, \ldots, x_n))$.

1. Supongamos que la máquina MT_g computa la función $g : \mathbb{N}^m \to \mathbb{N}$ y que las máquinas MT_{h_i} computan las funciones $h_i : \mathbb{N}^n \to \mathbb{N}$.

2. Inicialmente la máquina MT_f ejecuta la máquina MT_{h_1} con la descripción inicial

   $$\alpha_1 \equiv q_1\overrightarrow{x_1, \ldots, x_n};$$

   y obtenemos a la salida $\overrightarrow{h_1(x_1, x_2, \ldots, x_n)}$.

3. Después la máquina MT_f ejecuta la máquina MT_{h_2} con la descripción inicial

   $$\alpha_1 \equiv q_1\overrightarrow{x_1, \ldots, x_n};$$

   y adicionando algunas instrucciones, obtenemos la salida

   $$\overrightarrow{h_1(x_1, x_2, \ldots, x_n)}\square\overrightarrow{h_2(x_1, x_2, \ldots, x_n)}.$$

4. La máquina MT_f contínua con la ejecución de las máquinas $\mathsf{MT}_{h_3}, \ldots, \mathsf{MT}_{h_m}$ con la descripción inicial

   $$\alpha_1 \equiv q_1\overrightarrow{x_1, \ldots, x_n};$$

   y con la adición de las instrucciones necesarias, obtenemos finalmente la salida

   $$\overrightarrow{h_1(x_1, x_2, \ldots, x_n)}\square\overrightarrow{h_2(x_1, x_2, \ldots, x_n)}\square \ldots \square\overrightarrow{h_m(x_1, x_2, \ldots, x_n)}.$$

5. Entonces la máquina MT_f ejecuta la máquina MT_g con la descripción inicial

$$\alpha_1 \equiv q_1\overrightarrow{h_1(x_1, x_2, \ldots, x_n)}\square\overrightarrow{h_1(x_1, x_2, \ldots, x_n)}\square \ldots \square\overrightarrow{h_m(x_1, x_2, \ldots, x_n)};$$

y obtenemos a la salida el valor de la función $f(x_1, x_2, \ldots, x_n)$. □

Lema 2.53. *El esquema de recurrencia primitiva preserva la propiedad de Turing-computabilidad. Es decir, si $g : \mathbb{N}^n \to \mathbb{N}$ y $h : \mathbb{N}^{n+2} \to \mathbb{N}$ son funciones Turing-computables (totales, parciales), entonces*

$$f(x_1, x_2, \ldots, x_n, 0) = g(x_1, x_2, \ldots, x_n),$$
$$f(x_1, x_2, \ldots, x_n, k+1) = h(x_1, x_2, \ldots, x_n, k, f(x_1, x_2, \ldots, x_n, k)),$$

es una función Turing-computable (total, parcial).

Demostración. La idea es construir una máquina MT_f que compute el valor de

$$f(x_1, x_2, \ldots, x_n, 0)$$

a partir de $g(x_1, x_2, \ldots, x_n)$, y que compute cada uno de los valores

$$f(x_1, x_2, \ldots, x_n, 1), \ldots, f(x_1, x_2, \ldots, x_n, k+1)$$

a partir de los valores de

$$h(x_1, x_2, \ldots, x_n, 0, f(x_1, x_2, \ldots, x_n,)), \ldots, h(x_1, x_2, \ldots, x_n, k, f(x_1, x_2, \ldots, x_n, k))$$

respectivamente. Es decir, inicialmente se computa el valor de $f(x_1, x_2, \ldots, x_n, 0)$, después el valor de $f(x_1, x_2, \ldots, x_n, 1)$ y así sucesivamente hasta computar el valor de

$$f(x_1, x_2, \ldots, x_n, k+1).$$

1. Supongamos que la máquina MT_g computa la función $g(x_1, x_2, \ldots, x_n)$ y la máquina MT_h computa la función $h(x_1, x_2, \ldots, x_n, k, z)$.

2. Inicialmente la máquina MT_f calcula el valor de $f(x_1, x_2, \ldots, x_n, 0)$, es decir, ejecuta la máquina MT_g con la descripción inicial

$$\alpha_1 \equiv q_1\overrightarrow{x_1, \ldots, x_n}\square\overrightarrow{0},$$

de donde obtenemos que $\overrightarrow{f(x_1, x_2, \ldots, x_n, 0)} = \overrightarrow{g(x_1, \ldots, x_n)}$.

3. Con el valor de $f(x_1, x_2, \ldots, x_n, 0)$ la máquina MT_f calcula el valor de $f(x_1, x_2, \ldots, x_n, 1)$, es decir, ejecuta la máquina MT_h con la descripción inicial

$$\alpha_1 \equiv q_1^h\overrightarrow{x_1, \ldots, x_n}\square\overrightarrow{0}\square\overrightarrow{f(x_1, x_2, \ldots, x_n, 0)}$$

de donde obtenemos que:

$$\overrightarrow{f(x_1, x_2, \ldots, x_n, 1)} = \overrightarrow{h(x_1, x_2, \ldots, x_n, 0, f(x_1, x_2, \ldots, x_n, 0))}.$$

4. La máquina MT_f contínua con el proceso anterior hasta calcular el valor de

$$f(x_1, x_2, \ldots, x_n, k+1)$$

a partir de la máquina MT_h con la descripción inicial dada por

$$\alpha_1 \equiv q_1 \overrightarrow{x_1, \ldots, x_n} \Box \overrightarrow{k} \Box \overrightarrow{f(x_1, x_2, \ldots, x_n, k)};$$

de donde finalmente obtenemos que:

$$\overrightarrow{f(x_1, x_2, \ldots, x_n, k+1)} = \overrightarrow{h(x_1, x_2, \ldots, x_n, k, f(x_1, x_2, \ldots, x_n, k))}. \qquad \Box$$

Lema 2.54. *El esquema de minimalización preserva la propiedad de Turing-computabilidad. Es decir, si $g(\vec{x}, y)$ es una función Turing-computable (total y regular, total), entonces*

$$f(\vec{x}) = \mu_y(g(\vec{x}, y) = 0),$$

es una función Turing-computable (total, parcial).

Demostración. La idea es construir una máquina MT_f que compute incrementalmente el valor de y hasta encontrar (si es el caso) que $g(\vec{x}, y) = 0$, entonces la máquina MT_f retorna este valor de y.

1. Supongamos que la máquina MT_g computa la función $g(\vec{x}, y)$.
2. Inicialmente la máquina MT_f ejecuta la máquina MT_g con la descripción inicial:

$$\alpha_1 \equiv q_1 \overrightarrow{x_1, \ldots, x_n} \Box \overrightarrow{0},$$

y obtenemos a la salida $\overrightarrow{g(x_1, \ldots, x_n, 0)}$.

3. Después la máquina MT_f ejecuta la máquina MT_g con la descripción inicial

$$\alpha_1 \equiv q_1 \overrightarrow{x_1, \ldots, x_n} \Box \overrightarrow{1},$$

y obtenemos a la salida $\overrightarrow{g(x_1, \ldots, x_n, 1)}$.

4. La máquina MT_f contínua ejecutando sucesivamente la máquina MT_g hasta encontrar la salida $\overrightarrow{g(x_1, \ldots, x_n, y)} = \overrightarrow{0}$ y retorna el valor y. Si tal salida no existe, entonces la máquina MT_f no se detiene. $\Box$

Ahora podemos presentar el primer teorema relacionado con la coexistencia entre el conjunto de las funciones recursivas y el conjunto de las funciones Turing-computables.

Teorema 2.55. *Las funciones recursivas (totales, parciales) son funciones Turing-computables (totales, parciales).*

Demostración. La demostración está sustentada en los elementos involucrados en la definición de una función recursiva, es decir, las funciones de base y los esquemas de composición, recurrencia primitiva y minimalización.

Los lemas 2.49, 2.50 y 2.51 muestran que la función cero $\mathsf{z}(x) = 0$, la función sucesor $\mathsf{s}(x) = x + 1$ y las funciones k-ésima proyección $\mathsf{I}_k^n(x_1, x_2, \ldots, x_n) = x_k$ son funciones Turing-computables. Por otra parte, los lemas 2.52, 2.53 y 2.54 muestran que la propiedad de Turing-computabilidad es preservada en la aplicación de los esquemas de composición, recurrencia primitiva y minimalización. □

A continuación presentamos el segundo teorema relacionado con la coexistencia entre el conjunto de las funciones recursivas y el conjunto de las funciones Turing-computables.

Teorema 2.56. *Las funciones Turing-computables (totales, parciales) son funciones recursivas (totales, parciales).*

Demostración.

1. La demostración se realizará para una función Turing-computable (total, parcial) $f(x_1, x_2)$. La restricción a dos argumentos no es importante, dado que es posible hacer las generalizaciones necesarias para funciones de n argumentos.

2. Inicialmente la cinta contiene la dupla (x_1, x_2) codificada tal como lo indica la definición 1.14, es decir,

$$\begin{aligned}(\overrightarrow{x_1, x_2}) &= \overrightarrow{x_1}\square\overrightarrow{x_2} \\ &= \underbrace{|||\ldots|}_{x_1+1\ veces} \square \underbrace{|||\ldots|}_{x_2+1\ veces}.\end{aligned}$$

   Al final la cinta contiene el valor de $f(x_1, x_2)$ codificado por:

$$\overrightarrow{f(x_1, x_2)} = \underbrace{|||\ldots|}_{f(x_1,x_2)+1\ veces}.$$

   En necesario observar que hemos mofidicado la forma de salida de la máquina. Es decir, en lugar de que la máquina al final contenga $f(x_1, x_2)$ palitos sobre la cinta, la máquina contendrá $f(x_1, x_2) + 1$ palitos sobre la cinta. Esto debido al uso de la función *lo* descrita en el numeral 7 de esta demostración.

3. Los contenidos de la cinta y de la celda visitada en cualquier momento se representarán en notación binaria de la siguiente forma: Si se considera que $\square \equiv 0$, los contenidos de las celdas a la izquierda de la celda visitada pueden ser pensados como un número binario; éste será denotado por *inum*. El contenido de la celda visitada y las celdas a su derecha, pueden ser pensados como un número binario escrito inversamente; este número será denotado por *dnum*.

4. Veamos qué sucede en los números *inum* y *dnum* de acuerdo con los posibles movimientos de la máquina (de nuevo se va a considerar $\square \equiv 0$):

   a) La máquina no se mueve:
      1) La máquina cambia el símbolo 1 por 0:
         El número *dnum* se decrementa en uno y el número *inum* no cambia.
      2) La máquina cambia el símbolo 0 por 1:
         El número *dnum* se incrementa en uno y el número *inum* no cambia.

   b) La máquina se mueve a la izquierda:
      1) Si *inum* es impar: $inum = \frac{inum-1}{2}$; $dnum = 2dnum + 1$.
      2) Si *inum* es par: $inum = \frac{inum}{2}$; $dnum = 2dnum$.

   c) La máquina se mueve a la derecha:
      1) Si *dnum* es impar: $inum = 2inum + 1$; $dnum = \frac{dnum-1}{2}$.
      2) Si *dnum* es par: $inum = 2inum$; $dnum = \frac{dnum}{2}$.

5. Veamos los valores de *inum* y *dnum* al comienzo y al final de la computación. Al comienzo la máquina está en el primer 1 correspondiente a $\overrightarrow{x_1}$, luego

$$\begin{aligned} inum &= 0, \\ dnum &= (2^{x_2+1} \dot{-} 1)2^{x_1+2} + (2^{x_1+1} \dot{-} 1). \end{aligned}$$

   Al final la máquina está en el primer | correspondiente a $\overrightarrow{f(x_1, x_2)}$, luego

$$\begin{aligned} inum &= 0 \\ dnum &= 2^{f(x_1,x_2)+1} - 1. \end{aligned}$$

6. La función $k(x_1, x_2) = (2^{x_2+1} \dot{-} 1)2^{x_1+2} + (2^{x_1+1} \dot{-} 1)$ es una función primitiva recursiva.

7. La función

$$d(x, y) = \begin{cases} 0, & \text{si } 2^y \leq x; \\ 1, & \text{si } 2^y > x; \end{cases}$$

   es primitiva recursiva.

   La función

$$Mx_w(f(\vec{x}_n, y)) = \begin{cases} \text{mayor } y \text{ entre 0 y } w \text{ inclusive, tal que } f(\vec{x}_n, y) = 0; \\ 0, \text{ si tal } y \text{ no existe;} \end{cases}$$

   es primitiva recursiva si la función $f(\vec{x}_n, y)$ lo es.

   Entonces la función $\mathsf{lo}(x) = Mx_y(d(x, y))$ es primitiva recursiva.

Como $\mathsf{lo}(2^{z+1}-1) = z$, entonces cuando la máquina se detiene en el cómputo de $f(x_1, x_2)$ tenemos que:

$$\begin{aligned} f(x_1, x_2) &= \mathsf{lo}(dnum) \\ &= \mathsf{lo}(2^{f(x_1,x_2)+1} - 1). \end{aligned}$$

8. Codificamos la dinámica de la máquina mediante dos funciones a y q, de la siguiente forma:

   a) Cada estado q_i es codificado por el número i.

   b) El símbolo $\square$ es codificado por el número 0 y el símbolo 1 por el número 1.

   c) Si la máquina de Turing ejecuta una instrucción de la forma $q_i\square\square N q_k$ entonces $a(i,0) = 0$ y $q(i,0) = k$.

   d) Si la máquina de Turing ejecuta una instrucción de la forma $q_i\square 1 N q_k$ entonces $a(i,0) = 1$ y $q(i,0) = k$.

   e) Si la máquina de Turing ejecuta una instrucción de la forma $q_i\square\square L q_k$ entonces $a(i,0) = 2$ y $q(i,0) = k$.

   f) Si la máquina de Turing ejecuta una instrucción de la forma $q_i\square\square R q_k$ entonces $a(i,0) = 3$ y $q(i,0) = k$.

   g) Si la máquina de Turing ejecuta una instrucción de la forma $q_i 1\square N q_k$ entonces $a(i,1) = 0$ y $q(i,1) = k$.

   h) Si la máquina de Turing ejecuta una instrucción de la forma $q_i 11 N q_k$ entonces $a(i,1) = 1$ y $q(i,1) = k$.

   i) Si la máquina de Turing ejecuta una instrucción de la forma $q_i 11 L q_k$ entonces $a(i,1) = 2$ y $q(i,1) = k$.

   j) Si la máquina de Turing ejecuta una instrucción de la forma $q_i 11 R q_k$ entonces $a(i,1) = 3$ y $q(i,1) = k$.

   k) En otro caso, $a(i,x) = q(i,x) = 0$.

   De acuerdo con lo anterior, las funciones $a(x,y)$ y $q(x,y)$ son funciones primitivas recursivas definidas por casos.

9. Definimos ahora algunas funciones primitivas recursivas:

   a) $\mathsf{tpl}(x,y,z) = 2^x 3^y 5^z$

   b) $\mathsf{lft}(w)$ = al mayor $x \leq w$ tal que 2^x divide a w.

   c) $\mathsf{crt}(w)$ = al mayor $x \leq w$ tal que 3^x divide a w.

   d) $\mathsf{rft}(w)$ = al mayor $x \leq w$ tal que 5^x divide a w.

e) $e(x) = \begin{cases} 0, & \text{si } x \text{ es par;} \\ 1, & \text{si } x \text{ es impar.} \end{cases}$

La función tpl codifica una tripleta de números (x, y, z) en un único número $\mathsf{tpl}(x, y, z)$, y las funciones lft, crt y rft obtienen de este número codificado los correspondientes valores de x, y y z.

Si la celda visitada contiene un 0, entonces $e(dnum) = 0$ y si contiene un 1 entonces $e(dnum) = 1$.

10. Vamos a construir una función primitiva recursiva $g(x_1, x_2, t)$ que indique el comportamiento de la máquina en el paso de ejecución t (que no es el paso en el cual la máquina se detiene). Esta función está definida por:

$$\begin{aligned} g(x_1, x_2, t) = \mathsf{tpl}(&\textit{inum} \text{ en } t, \\ &\text{estado de la máquina en } t, \\ &\textit{dnum} \text{ en } t). \end{aligned}$$

La definición de $g(x_1, x_2, t)$ se hará usando el esquema de recurrencia primitva. Para $g(x_1, x_2, 0)$, como la máquina comienza a ejecutar desde el estado q_1 y está parada en el primer 1 de $\overrightarrow{x_1}$ tenemos que:

$$g(x_1, x_2, 0) = \mathsf{tpl}(0, 1, k(x_1, x_2)).$$

Para $g(x_1, x_2, t + 1)$ es necesario considerar varios casos de acuerdo con las posibles situaciones e instrucciones que pueda realizar la máquina. Las convenciones utilizadas son las siguientes:

$$\begin{aligned} l &= \mathsf{lft}(g(x_1, x_2, t)), \\ c &= \mathsf{crt}(g(x_1, x_2, t)), \\ r &= \mathsf{rft}(g(x_1, x_2, t)), \\ q &= q(c, e(r)); \end{aligned}$$

entonces la función $g(x_1, x_2, t + 1)$ está definida por:

$$g(x_1, x_2, t+1) = \begin{cases} \mathsf{tpl}(l, q, r), & \text{si } a(c, e(r)) = 0 \text{ y } e(r) = 0; \\ \mathsf{tpl}(l, q, r \dot{-} 1), & \text{si } a(c, e(r)) = 0 \text{ y } e(r) = 1; \\ \mathsf{tpl}(l, q, r + 1), & \text{si } a(c, e(r)) = 1 \text{ y } e(r) = 0; \\ \mathsf{tpl}(l, q, r), & \text{si } a(c, e(r)) = 1 \text{ y } e(r) = 1; \\ \mathsf{tpl}(l/2, q, 2r), & \text{si } a(c, e(r)) = 2 \text{ y } e(l) = 0; \\ \mathsf{tpl}((l \dot{-} 1)/2, q, 2r + 1), & \text{si } a(c, e(r)) = 2 \text{ y } e(l) = 1; \\ \mathsf{tpl}(2l, q, r/2), & \text{si } a(c, e(r)) = 3 \text{ y } e(l) = 0; \\ \mathsf{tpl}(2l + 1, q, (r \dot{-} 1)/2), & \text{si } a(c, e(r)) = 3 \text{ y } e(l) = 1; \\ 0, & \text{de otro modo.} \end{cases}$$

Como la función $g(x_1, x_2, t)$ está definida a partir del esquema de recurrencia primitiva sobre funciones recursivas primitivas, entonces g es una función primitiva recursiva.

11. Para definir $f(x_1, x_2)$ como una función recursiva es necesario realizar algunos construcciones adicionales.

    a) Si la máquina se detiene en el estado t, entonces $\mathsf{crt}(g(x_1, x_2, y)) \neq 0$ para todo $y \leq t$ y $\mathsf{crt}(g(x_1, x_2, y)) = 0$ para todo $y > t$.

    b) Entonces la máquina para de computar a $f(x_1, x_2)$ si y sólo si t es el menor y tal que $\mathsf{crt}(g(x_1, x_2, y + 1)) = 0$.

    c) La función $h(x_1, x_2, y) = \mathsf{crt}(g(x_1, x_2, y + 1))$ es primitiva recursiva.

    d) Todas las funciones definidas hasta el momento son primitivas recursivas. La siguiente función ofrece el carácter de recursivad a las funciones Turing-computables.

    e) La función $p(x_1, x_2) = \mu_y(h(x_1, x_2, y) = 0)$ es recursiva.

    f) Sí la función $f(x_1, x_2)$ es Turing-computable total, la función $p(x_1, x_2)$ es una función recursiva total y si, la función $f(x_1, x_2)$ es Turing-computable parcial, la función $p(x_1, x_2)$ es una función recursiva parcial.

12. Finalmente definimos $f(x_1, x_2) = \mathsf{lo}(\mathsf{rft}(g(x_1, x_2, p(x_1, x_2))))$, como f está formada por composición de funciones recursivas (totales, parciales), entonces f es una función recursiva (total, parcial). □

Ejemplo 2.57. Como ejemplo del teorema 2.56 demostraremos que la función Turing-computable $f(x, y) = 0$ es una función primitiva recursiva.

1. Definición de la máquina de Turing que calcula $f(x, y) = 0$.

   Vamos a definir una máquina de Turing MT tal que la función asociada con la máquina de Turing sea la función $f(x, y) = 0$, es decir, $f^{(2)}_{\mathsf{MT}}(x, y) = f(x, y) = 0$. Como caso particular vamos a realizar el análisis para la instancia de la función $f(2, 1) = 0$. Se debe tener en cuenta que la máquina de Turing MT que calcula la función $f(2, 1) = 0$ debe estar implementada de manera que concuerde con la dinámica expuesta en el paso (8) de la demostración, es decir, debe poseer instrucciones simples. Además la máquina debe comenzar en el estado q_1 para que concuerde con la función primitiva recursiva g definida en el paso (10) de la demostración y debe finalizar sólo con un palito en la cinta, dado que el número de palitos al finalizar debe ser $f(2, 1) + 1$. De acuerdo con lo anterior, la máquina de Turing MT que calcula la función $f(x_1, x_2) = 0$ está definida por: $I = \{i_1, i_2, i_3, i_4, i_5, i_6\}$ donde:

i_1: q_1 1 $\square$ N q_2
i_2: q_2 $\square$ $\square$ R q_1
i_3: q_1 $\square$ $\square$ R q_3
i_4: q_3 1 $\square$ N q_4
i_5: q_4 $\square$ $\square$ R q_3
i_6: q_3 $\square$ 1 N q_5

2. Seguimiento del *inum* y del *dmun*

   La descripción instantánea inicial es $q_1\overrightarrow{x_1}\square\overrightarrow{x_2}$ que para el caso de $f(2,1)$ es q_1 1 1 1 $\square$ 1 1. Entonces $inum = 0$ y $dnum = 110111_2 = 55$, donde 110111_2 significa que el número está en binario. La máquina comienza entonces la ejecución de las instrucciones. La simulación de la ejecución, la descripción instantánea que se obtiene, el valor de *inum* y de *dnum* se presentan en la tabla 2.2. De acuerdo con la tabla 2.2 los valores finales de *inum* y *dnum* son $inum = 0$, $dnum = 2^{f(2,1)+1} - 1 = 1$. Luego $f(2,1) = \mathsf{lo}(dnum_{final}) = \mathsf{lo}(1) = 0$.

Instrucción	Descripción instantánea	*inum*	*dnum*
i_1	q_2 $\square$ 1 1 $\square$ 1 1	0	$110110_2 = 54$
i_2	$\square$ q_1 1 1 $\square$ 1 1	0	$11011_2 = 27$
i_1	$\square$ q_2 $\square$ 1 $\square$ 1 1	0	$11010_2 = 26$
i_2	$\square$ $\square$ q_1 1 $\square$ 1 1	0	$1101_2 = 13$
i_1	$\square$ $\square$ q_2 $\square$ $\square$ 1 1	0	$1100_2 = 12$
i_2	$\square$ $\square$ q_1 $\square$ 1 1	0	$110_2 = 6$
i_3	$\square$ $\square$ $\square$ $\square$ q_3 1 1	0	$11_2 = 3$
i_4	$\square$ $\square$ $\square$ $\square$ q_4 $\square$ 1	0	$10_2 = 2$
i_5	$\square$ $\square$ $\square$ $\square$ $\square$ q_3 1	0	$1_2 = 1$
i_4	$\square$ $\square$ $\square$ $\square$ $\square$ q_4 $\square$	0	$0_2 = 0$
i_5	$\square$ $\square$ $\square$ $\square$ $\square$ $\square$ q_3 $\square$	0	$0_2 = 0$
i_6	$\square$ $\square$ $\square$ $\square$ $\square$ $\square$ q_5 1	0	$1_2 = 1$

Tabla 2.2: Simulación máquina de Turing que calcula la función $f(2,1) = 0$.

3. Dinámica de la máquina de Turing MT

   Con base en el paso (8) de la demostración, codificamos la dinámica de la máquina MT

de la siguiente manera:

$$\begin{array}{ll}
i_1\colon q_1\ 1\ \square\ N\ q_2 & a(1,1)=0\ y\ q(1,1)=2 \quad \text{por } (g)\\
i_2\colon q_2\ \square\ \square\ R\ q_1 & a(2,0)=3\ y\ q(2,0)=1 \quad \text{por } (f)\\
i_3\colon q_1\ \square\ \square\ R\ q_3 & a(1,0)=3\ y\ q(1,0)=3 \quad \text{por } (f)\\
i_4\colon q_3\ 1\ \square\ N\ q_4 & a(3,1)=0\ y\ q(3,1)=4 \quad \text{por } (g)\\
i_5\colon q_4\ \square\ \square\ R\ q_3 & a(4,0)=3\ y\ q(4,0)=3 \quad \text{por } (f)\\
i_6\colon q_3\ \square\ 1\ N\ q_5 & a(3,0)=1\ y\ q(3,0)=5 \quad \text{por } (d)
\end{array}$$

4. Construcción de la función primitiva recursiva $f(2,1)=0$

   La construcción de la dinámica de la máquina MT del paso anterior se realizó para implementar la función primitiva recursiva:

   $$g(2,1,t)=\mathsf{tpl}(\textit{inun en } t, \textit{estado de la máquina en } t, \textit{dnum en } t),$$

   donde t es una variable que se incrementa cada vez que se ejecuta una instrucción de la máquina MT y $\mathsf{tpl}(x,y,z)$ es la función primitiva recursiva definida por $\mathsf{tpl}(x,y,z)=2^x3^y5^z$. La función primitiva recursiva g está definida por casos en el paso (10) de la demostración. De acuerdo con el paso (11) de la demostración, se define la función $p(2,1)$ de la siguiente manera:

   $$\begin{aligned} p(2,1) &= \mu_y(h(2,1,y)=0)\\ &= \mu_y(\mathsf{crt}(g(2,1,y+1))=0), \end{aligned}$$

   donde la función crt devuelve el número y de $\mathsf{tpl}(x,y,z)$. La función $p(2,1)$ calcula el menor $y+1$ tal que $\mathsf{crt}(g(2,1,y+1))=0$. Finalmente definimos la función $f(2,1)$ como $f(2,1)=\mathsf{lo}(\mathsf{rft}(g(2,1,p(2,1))))$, donde la función lo y la función rft son funciones primitivas recursivas definidas en los pasos (7) y (9) de la demostración.

5. Cálculo de $p(2,1)$

   Para calcular el valor de $p(2,1)$ y de allí el valor de $f(2,1)$ procedemos de la siguiente manera: suponemos $p(2,1)$ conocido y operamos de una manera recursiva, ya que la función g depende del estado de la configuración t anterior, hasta un $t=0$. Sea $p(2,1)=n$, entonces para calcular $f(2,1)=\mathsf{lo}(\mathsf{rft}(g(2,1,n)))$ necesitamos $g(2,1,n)$; para calcular $g(2,1,n)$ necesitamos $g(2,1,n-1)$; para calcular $g(2,1,n-1)$ necesitamos $g(2,1,n-2)$ y así sucesivamente hasta calcular $g(2,1,0)=\mathsf{tpl}(0,1,k(2,1))=\mathsf{tpl}(0,1,55)$ definido en el paso (10) de la demostración. Procedemos entonces a realizar estos cálculos.

   a) Para $t=0$:
   $g(2,1,0)=\mathsf{tpl}(0,1,55)$

b) Para $t = 1$:

$$\begin{aligned} c &= \mathsf{crt}(g(2,1,0)) = \mathsf{crt}(\mathsf{tpl}(0,1,55)) = 1 \\ r &= \mathsf{rft}(g(2,1,0)) = \mathsf{rft}(\mathsf{tpl}(0,1,55)) = 55 \\ l &= \mathsf{lft}(g(2,1,0)) = \mathsf{lft}(\mathsf{tpl}(0,1,55)) = 0 \\ e(r) &= 1 \text{ (Paridad)} \\ a(c,e(r)) &= a(1,1) = 0 \\ q(c,e(r)) &= q(1,1) = 2 \end{aligned}$$

Entonces $g(2,1,1) = \mathsf{tpl}(0,2,55 \dot{-} 1) = \mathsf{tpl}(0,2,54)$

c) Los cálculos necesarios para $t = 2$ hasta $t = 13$ se presentan en la tabla 2.3.

t	c	r	l	$e(r)$	$a(c,e(r))$	$q(c,e(r))$	$g(2,1,t)$
2	2	54	0	0	3	1	$\mathsf{tpl}(0,1,27)$
3	1	27	0	1	0	2	$\mathsf{tpl}(0,2,26)$
4	2	26	0	0	3	1	$\mathsf{tpl}(0,1,13)$
5	1	13	0	1	0	2	$\mathsf{tpl}(0,2,12)$
6	2	12	0	0	3	1	$\mathsf{tpl}(0,1,6)$
7	1	6	0	0	3	3	$\mathsf{tpl}(0,3,3)$
8	3	3	0	1	0	4	$\mathsf{tpl}(0,4,2)$
9	4	2	0	0	3	3	$\mathsf{tpl}(0,3,1)$
10	3	1	0	1	0	4	$\mathsf{tpl}(0,4,0)$
11	4	0	0	0	3	3	$\mathsf{tpl}(0,3,0)$
12	3	0	0	0	1	5	$\mathsf{tpl}(0,5,1)$
13	5	1	0	1	0	0	$\mathsf{tpl}(0,0,0)$

Tabla 2.3: Cálculo de $p(2,1)$.

d) Por primera vez en el seguimiento $\mathsf{crt}(g(2,1,13)) = \mathsf{crt}(\mathsf{tpl}(0,0,0)) = 0$. Esto implica que $n = 12$, es decir, $p(2,1) = 12$.

6. Cálculo de $f(2,1)$

Dado que $p(2,1) = 12$, entonces

$$\begin{aligned} f(2,1) &= \mathsf{lo}(\mathsf{rft}(g(2,1,12))) \\ &= \mathsf{lo}(\mathsf{rft}(\mathsf{tpl}(0,5,1))) \\ &= \mathsf{lo}(1) \\ &= 0. \end{aligned}$$

Sugerimos comparar las tablas 2.2 y 2.3 para observar que la computación realizada es la misma.

7. Hemos demostrado entonces que la función Turing-computable $f(2,1) = 0$ se puede definir como una función primitiva recursiva.

2.12. Tesis de Church-Turing

La tesis de Church-Turing, según la cual una función es efectivamente calculable si y sólo si es una función recursiva, ha conocido bastantes discusiones, tanto por filósofos como por matemáticos.

El hecho crucial de que las diversas caracterizaciones de las funciones calculables por un algoritmo hayan conducido a la clase de las funciones recursivas (Turing-computables) y el hecho de que aún no se haya producido una función efectivamente calculable que no sea recursiva, constituyen argumentos de peso o evidencia en favor de la conjetura conocida como tesis de Church-Turing (extendida), tesis que afirma que la clase de las funciones parciales calculables es idéntica a la clase de las funciones recursivas parciales.

De hecho, aceptar como válido este enunciado (puesto que no puede ser probado) implica establecer de una vez por todas que la noción intuitiva de algoritmo corresponde a las descripciones matemáticas que han sido dadas (máquinas de Turing, recursividad, etc.). En otras palabras, aceptar la tesis equivale a afirmar categóricamente que: toda función parcial calculable mediante un algoritmo es una función recursiva parcial.

La potencia de la tesis extendida de Church-Turing radica en el hecho de que se puedan usar estrategias o técnicas matemáticas para probar la existencia o no de algoritmos para un cierta clase particular de problemas. Y, recíprocamente, sabido de la existencia de un procedimiento mecánico o algoritmo particular, poder afirmar que el conjunto o función correspondiente es recursivo.

Ejemplo 2.58. Como un conjunto es recursivamente enumerable si existe una función recursiva f tal $f(0), f(1), \ldots$ es una lista exhaustiva del conjunto, entonces la tesis de Church-Turing conduce a la idea de que recursivamente enumerable es equivalente a efectivamente enumerable.

Ejemplo 2.59.

1. Es sabido que: el conjunto A de los números de Gödel de las fórmulas de la teoría de números N que son verdaderas, no es recursivo.

2. Aplicando la tesis de Church-Turing podemos entonces afirmar: no existe ningún algoritmo que sirva para responder la pregunta: ¿es la fórmula α verdadera en N?

3. Luego, no existe una función de decisión f para el conjunto A. Así, A no es un conjunto decidible (en sentido intuitivo)

Un sistema formal es recursivamente indecidible si el conjunto de los números de Gödel de los teoremas del sistema no es recursivo. Usando la tesis de Church-Turing podemos observar que un sistema formal es indecidible si y sólo si no existe un algoritmo que pueda responder la cuestión: ¿es la fórmula α un teorema del sistema?

Teorema 2.60. *Existe un subconjunto de* $\mathbb{N}$ *que es recursivamente enumerable pero no es recursivo.*

Demostración. Presentaremos en este contexto una justificación incompleta, dejando el resto al cuidado del lector.

1. Sea N el sistema formal de la teoría de números.
2. El conjunto A de los axiomas del sistema formal N es recursivo (a partir de los números de Gödel de los mismos).
3. Puede diseñarse un procedimiento efectivo para listar el conjunto de teoremas de N a partir del conjunto de axiomas A.
4. Por la tesis de Church-Turing, el conjunto de los teoremas de N es recursivamente enumerable.
5. N es recursivamente indecidible.
6. Luego, el conjunto de los teoremas de N no es recursivo. □

2.13. Ejercicios

Ejercicio 2.1. Demuestre que las funciones test de desigualdad denotada por ϵ y test de igualdad denotada por δ son funciones primitivas recursivas.

$$\epsilon(x,y) = \begin{cases} 0, & \text{si } x = y; \\ 1, & \text{si } x \neq y. \end{cases} \qquad \delta(x,y) = \begin{cases} 1, & \text{si } x = y; \\ 0, & \text{si } x \neq y. \end{cases}$$

Ejercicio 2.2. Demuestre que las siguientes funciones son primitivas recursivas:

1. $\mathsf{rm}(x,y)$ = residuo de la división de y por x.
2. $\mathsf{qt}(x,y)$ = cociente de la división de y por x.
3. $\lceil \sqrt{x} \rceil$ = mayor entero $\leq \sqrt{x}$.

Ejercicio 2.3. Demuestre que la intersección y unión de dos conjuntos primitivos recursivos y el complemento de un conjunto primitivo recursivo, es un conjunto primitivo recursivo.

Ejercicio 2.4. Demuestre que los siguientes predicados son primitivos recursivos:

1. $\leq$.
2. $>$.
3. $\geq$.
4. $|$ divide
5. $P(x)$: x es par.
6. $I(x)$: x es impar.
7. $Pr(x)$: x es primo.

Ejercicio 2.5. Demuestre que las siguientes funciones son primitivas recursivas:

1. $\mathsf{máx}(x_1, x_2, \ldots, x_n)$.
2. $\mathsf{mín}(x_1, x_2, \ldots, x_n)$.

Ejercicio 2.6. Demuestre que la sucesión de Fibonacci definida por $f(0) = 1$, $f(1) = 2$, $f(k+2) = f(k) + f(k+1)$ para $k \geq 0$, es primitiva recursiva.

Ejercicio 2.7. Demuestre que las siguientes funciones son primitivas recursivas:

1. $f(0) = 2$, $f(1) = 4$, $f(k+2) = 3f(k+1) \dot{-}(2f(k) + 1)$.
2. $f(0) = 5$, $f(n+1) = f(n) \cdot f(n \dot{-} 2) + f(n \dot{-} 5)$.
3. $f(0) = 5$, $f(n+1) = (n+1)^{f(n)} + n \cdot f(n \dot{-} 3)$.

Ejercicio 2.8. El siguiente programa calcula c empleando a como entrada. Expresar c como función de a

```
b := 3 + a;
c := 2 + a;
c := c * b;
```

Ejercicio 2.9. Expresar el valor final de la variable *suma* como una función primitiva recursiva.

for $i := 1$ **to** n **do** $suma := suma + i * i$

Ejercicio 2.10. Expresar la función $if(x, y, z)$ como una función primitiva recursiva.

$$if(x, y, z) = \begin{cases} y, & \text{si } x > 0, \\ z, & \text{si } x = 0. \end{cases}$$

Ejercicio 2.11. Demuestre la segunda parte del teorema 2.14.

Ejercicio 2.12. Las siguientes funciones no son regulares. Justifíquelo.

1. $f(x, y) = x + y$.
2. $g(x, y) = (x \dot{-} y) + y$.

Ejercicio 2.13. Verifique que la función $f(x, y, z) = (x \dot{-} z) + (y \dot{-} z)$ es regular para la variable z.

Ejercicio 2.14. Implemente en algún lenguaje de programación la función de Ackermann y observe el crecimiento de la función para algunos valores (x, y) contra el crecimiento del número de llamadas necesarias a la función para calcular dichos valores.

Ejercicio 2.15. La función de Ackermann es un ejemplo de una función recursiva que no es primitiva recursiva. La siguiente demostración afirma que la función de Ackermann es una función primitiva recursiva.

Demostración. Cada "brazo" de la función de Ackermann puede ser expresado de la siguientes forma:

1. $\mathsf{ack}(0, y) = \mathsf{s}(y)$, donde $\mathsf{s}(y)$ es la función sucesor.
2. $\mathsf{ack}(x + 1, 0) = \mathsf{ack}(\mathsf{pred}(x + 1), \mathsf{s}(\mathsf{z}(0)))$ donde pred es la función predecesor y z es la función cero.
3. $\mathsf{ack}(x + 1, y + 1) = \mathsf{ack}(\mathsf{pred}(x + 1), \mathsf{ack}(x + 1, \mathsf{pred}(y + 1)))$.

Como cada "brazo" de la función de Ackermann puede ser expresado por una función primitiva recursiva, o por composición de funciones primitivas recursivas, entonces la función de Ackermann es primitiva recursiva. □

Explicar por qué la demostración anterior no es válida.

Ejercicio 2.16. Presente un ejemplo de una función estrictamente recursiva parcial (que no sea total).

Ejercicio 2.17. Suponga que h es una función parcial recursiva la cual no es total y g es una función recursiva. Sea $f = h \circ g$, ¿puede f ser una función total?

Ejercicio 2.18. Para el ejemplo 2.36 hallar un algoritmo más eficiente que decida si un número $n \in \mathbb{N}$ es o no es un número primo.

Ejercicio 2.19. Para el conjunto del ejemplo 2.37, ¿cúal es el procedimiento de decisión?

Ejercicio 2.20. El conjunto de las tautologías de la lógica de enunciados es un conjunto decidible. ¿Cúal es el procedimiento de decisión?

Ejercicio 2.21. Demuestre que la función del ejemplo 2.43 es sobreyectiva y recursiva.

Ejercicio 2.22. Demuestre el lema 2.47.

2.14. Notas bibliográficas

Los textos de Boolos, Burges y Jeffrey [2007], Caicedo [1990], Davis [1982], Hermes [1969], Kleene [1974], Mendelson [2015], Minsky [1967], Sieg [1997], Soare [1996] y Yasuhara [1971], entre otros, presentan una contextualización a las funciones recursivas. Las funciones y relaciones numérico-teóricas son presentadas en el texto de Mendelson [2015]. En relación con la función de Ackermann, [Hermes 1969] presenta una demostración de que no es una función primitiva recursiva; por otra parte, algunos de los datos del número de llamadas de la función de Ackermann, fueron tomados de [Dotzel 1991]. Las nociones de conjunto enumerable y conjunto efectivamente enumerable son presentas en el texto de Caicedo [1990], así como los ejemplos 2.34 y 2.46 y el algoritmo presentado en el teorema 2.48. Para la demostración de los lemas 2.50, 2.51 y 2.54 seguimos a [Davis 1982] y para el lema 2.52 seguimos a [Caicedo 1990]. Para la demostración del teorema 2.56 seguimos la presentación realizada por Boolos y Jeffrey [1989]. El texto de Kleene [1974] y en especial el texto de Davis [1982], ofrecen una demostración mucho más elaborada del mismo. En cuanto a la tesis de Church-Turing, [Kleene 1974; Sieg 1997; Soare 1996] ofrecen algunos elementos, tanto históricos como técnicos.

Capítulo 3

Lenguajes y gramáticas

3.1. Alfabetos y lenguajes

Un alfabeto será para nosotros, en este contexto, cualquier conjunto de objetos que llamaremos símbolos indivisibles. Un ejemplo de ello es el alfabeto romano $\{a, \ldots, z, A, \ldots, Z\}$. Un alfabeto básico en la computación es el conjunto $\{0, 1\}$.

3.1.1. Definiciones preliminares

Definición 3.1 (Alfabeto)**.** Un alfabeto es un conjunto enumerable (finito o infinito) de símbolos indivisibles.

Ejemplo 3.2. Ejemplos de alfabetos:

$$\begin{aligned}\Sigma_1 &= \{0, 1\},\\ \Sigma_2 &= \{/\},\\ \Sigma_3 &= \{(, \neg\} \cup \{\, p_i \mid i \in \mathbb{N}\,\}.\end{aligned}$$

Definición 3.3 (Palabra)**.** Una palabra es una sucesión finita de símbolos de un alfabeto Σ. Como palabra de Σ se incluye la palabra vacía denotada por ε.

Ejemplo 3.4. Para $\Sigma_1 = \{0, 1\}$ son palabras $\alpha \equiv 00000$ y $\beta \equiv 0101010101$. Para $\Sigma_3 = \{\vee, \neg\} \cup \{\, p_i \mid i \in \mathbb{N}\,\}$ son palabras $\alpha_1 \equiv \neg(p_1 \vee p_2)$, $\alpha_2 \equiv \neg\neg\neg$ y ε.

Definición 3.5 (Lenguaje universal)**.** El conjunto Σ^* de todas las palabras que se pueden construir sobre un alfabeto Σ se denomina lenguaje universal para Σ, es decir:

$$\Sigma^* = \{\, a_1 a_2 \ldots a_n \mid n \in \mathbb{N}\,\} \cup \{\varepsilon\}, \quad a_i \in \Sigma.$$

La definición que presentamos a continuación (lenguaje sobre un alfabeto) es válida, pero poco útil, puesto que lo que nos interesa en este contexto es el poder dar una representación

finita o descripción finita de un lenguaje infinito. No obstante, para estos mismos propósitos es importante que presentemos tal definición.

Definición 3.6 (Lenguaje)**.** Sea Σ un alfabeto. Un lenguaje L sobre Σ denotado por $L(\Sigma)$ es un subconjunto de Σ^*, es decir, $L(\Sigma)$ es un lenguaje sobre Σ, si y sólo si, $L(\Sigma) \subseteq \Sigma^*$.

Ejemplo 3.7. Si Σ es un alfabeto, entonces, Σ^* y $\emptyset$ son lenguajes sobre Σ.

Ejemplo 3.8. Dado $\Sigma = \{a, b, \ldots, z\}$, los siguientes conjuntos son lenguajes finitos y por ende tienen una descripción finita.

$$L_1(\Sigma) = \{casa, hola, perro\},$$
$$L_2(\Sigma) = \{aaa, dfg, asd, jkl\}.$$

Si el lenguaje $L(\Sigma)$ es infinito no podemos, bajo estas consideraciones, describir todas sus palabras, lo cual nos obligará más tarde a buscar otros métodos para su representación finita (si tal cosa es posible).

Ejemplo 3.9. Los siguientes lenguajes sobre $\Sigma = \{0, 1\}$ son infinitos:

$$L_1(\Sigma) = \{0, 01, 011, 0111, 01111, \ldots\},$$
$$L_2(\Sigma) = \{\, \alpha \in \Sigma^* \mid l(\alpha) = 3k, k \in \mathbb{N} \,\},$$
$$L_3(\Sigma) = \{\, \alpha \in \Sigma^* \mid \alpha \equiv 0\beta, \beta \in \Sigma^* \,\}.$$

3.1.2. Operaciones sobre palabras

Sobre el conjunto de palabras que pueden construirse sobre un alfabeto dado, podemos definir una función matemática que asigne a cada palabra el número de símbolos de que consta, número que llamaremos longitud de dicha palabra.

Definición 3.10 (Longitud de una palabra)**.** Dada un alfabeto Σ, existe una función $l : \Sigma^* \to \mathbb{N}$ tal que:

$$l(\varepsilon) = 0,$$
$$l(\alpha a) = l(\alpha) + 1; \quad \alpha \in \Sigma^*, a \in \Sigma.$$

Ejemplo 3.11. Sea $\Sigma = \{a, b\}$ entonces

$$l(\varepsilon) = 0,$$
$$l(\varepsilon a) = l(\varepsilon) + 1 = 1, \quad a \in \Sigma,$$
$$l(ab) = l(a) + 1 = 1 + 1 = 2,$$
$$l(abc) = l(ab) + 1 = 2 + 1 = 3.$$

Definición 3.12 (Conjunto de palabras de longitud n)**.** Llamaremos Σ^n al conjunto de todas las palabras de longitud n construibles sobre un alfabeto Σ, luego:

$$\begin{aligned}
\Sigma^0 &= \{\varepsilon\},\\
\Sigma^1 &= \{\, \alpha \mid l(\alpha) = 1 \,\},\\
&\vdots\\
\Sigma^n &= \{\, \alpha \mid l(\alpha) = n \,\}.
\end{aligned}$$

El conjunto Σ^* es el conjunto infinito: $\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$, es decir:

$$\Sigma^* = \bigcup_{n\in\mathbb{N}} \Sigma^n.$$

Dos palabras sobre un mismo alfabeto pueden ser combinadas para formar una tercera mediante la operación de concatenación. La concatenación $\alpha \cdot \beta$, de las palabras α y β, es la palabra obtenida escribiendo β inmediatamente después de α.

Definición 3.13 (Concatenación de palabras)**.** La concatenación $\cdot$ es una ley de composición interna sobre Σ^*, es decir, es una aplicación $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$ definida por:

$$\cdot(a_1a_2\ldots a_n, b_1b_2\ldots b_m) = a_1a_2\ldots a_nb_1b_2\ldots b_m.$$

En otros términos, sean $\alpha \equiv a_1a_2\ldots a_n$, y $\beta \equiv b_1b_2\ldots b_m$, palabras de Σ^*, entonces:

$$\alpha \cdot \beta = \alpha\beta \equiv a_1a_2\ldots a_nb_1b_2\ldots b_m.$$

Ejemplo 3.14. Sea $\Sigma = \{a, b, c, d\}$, entonces:

$$\begin{aligned}
aaa \cdot bbb &= aaabbb,\\
abcd \cdot dcba &= abcddcba.
\end{aligned}$$

Definición 3.15 (Potencia de una palabra)**.** Sea α una palabra sobre un alfabeto Σ, entonces:

$$\begin{aligned}
\alpha^0 &= \varepsilon,\\
\alpha^1 &= \alpha,\\
\alpha^2 &= \alpha\alpha,\\
\alpha^3 &= \alpha^2\alpha,\\
&\vdots\\
\alpha^n &= \alpha^{n-1}\alpha \quad \text{para } n > 0.
\end{aligned}$$

Ejemplo 3.16. Si $\Sigma = \{1, 2, 3\}$ entonces:

$$\alpha \equiv 12 \text{ y } \alpha^2 \equiv 1212,$$
$$\beta \equiv 23 \text{ y } \beta^3 = \beta^2\beta \equiv (23)^2(23) = 232323.$$

Teorema 3.17. *Si $\alpha, \beta \in \Sigma^*$, entonces $l(\alpha \cdot \beta) = l(\alpha) + l(\beta)$.*

Demostración. Ejercicio 3.7. □

Teorema 3.18. *Si $\alpha \in \Sigma^*$, entonces $l(\alpha^n) = nl(\alpha)$.*

Demostración. La demostración se hará por inducción.

1. Si $n = 0$ entonces
   $l(\alpha^0) = l(\varepsilon) = 0 = 0 \cdot 0 = 0 \cdot l(\varepsilon)$.
2. Suponemos que si $n = k$, entonces, $l(\alpha^k) = kl(\alpha)$.
3. Para $n = k + 1$:

$$\begin{aligned} l(\alpha^{k+1}) &= l(\alpha^k \cdot \alpha) \\ &= l(\alpha^k) + l(\alpha) \\ &= kl(\alpha) + l(\alpha) \\ &= (k+1)l(\alpha). \end{aligned}$$

□

Definición 3.19 (Reflexión de una palabra)**.** Sea α una palabra sobre un alfabeto Σ, y $\alpha \equiv a_1a_2 \ldots a_{n-1}a_n$. Definimos la reflexión de α, denotada por α^{-1}, a la palabra:

$$\alpha^{-1} \equiv a_n a_{n-1} \ldots a_2 a_1.$$

Ejemplo 3.20. Sea $\Sigma = \{a, b, c, o\}$ y una palabra $\alpha \equiv acob$ entonces $\alpha^{-1} \equiv boca$.

3.1.3. Relación entre los conceptos de monoide y lenguaje

Pasemos ahora a estudiar la estructura algebraica del lenguaje universal. Inicialmente presentamos las definiciones de semigrupo y monoide.

Definición 3.21 (Semigrupo)**.** Sea $\mathsf{L} = \{*\}$ un lenguaje de primer orden, compuesto por un símbolo de función de aridez dos $(*)$. Un semigrupo $\mathsf{S} = (A, *)$ es un modelo de L que satisface el siguiente axioma:

$$\mathsf{S} \models \forall x \forall y \forall z((x * y) * z = x * (y * z)) \quad \text{(asociativa)} \tag{S1}$$

Definición 3.22 (Monoide)**.** Sea $\mathsf{L} = \{*, e\}$ un lenguaje de primer orden, compuesto por un símbolo de función de aridez dos $(*)$ y por un símbolo de constante (e). Un monoide $\mathsf{M} = (A, *, e)$ es un modelo de L que satisface los siguientes axiomas:

$$\mathsf{M} \models \forall x \forall y \forall z((x * y) * z = x * (y * z)) \quad \text{(asociativa)} \tag{M1}$$

$$\mathsf{M} \models \forall x(x * e = e * x = x) \quad \text{(existencia elemento neutro)} \tag{M2}$$

Ejemplo 3.23. $\langle \mathbb{N}, * \rangle$ y $\langle \mathbb{Z}, + \rangle$ son ejemplos de semigrupos. $\langle \mathbb{N}, *, 1 \rangle$ y $\langle \mathbb{Z}, +, 0 \rangle$ son ejemplos de monoides.

Presentamos entonces el teorema que prescribe la estructrua algebraica de un lenguaje universal.

Teorema 3.24. *Si Σ es un alfabeto y si $\langle \Sigma^*, \cdot, \varepsilon \rangle$ es una estructura donde; Σ^* es el lenguaje universal para Σ, $\cdot$ es la concatenación de palabras, y ε es la palabra vacía, entonces $\langle \Sigma^*, \cdot, \varepsilon \rangle$ es un monoide, llamado el monoide libre generado por Σ.*

Demostración.

1. $\cdot$ es una operación cerrada bajo Σ^*, es decir:
   $\langle \Sigma^*, \cdot, \varepsilon \rangle \models \forall x \forall y (x \cdot y \in \Sigma^*)$.
2. $\langle \Sigma^*, \cdot, \varepsilon \rangle \models \forall \alpha \forall \beta \forall \gamma ((\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma))$ (axioma de asociatividad).
3. $\langle \Sigma^*, \cdot, \varepsilon \rangle \models \forall \alpha (\alpha \cdot \varepsilon = \varepsilon \cdot \alpha = \alpha)$ (axioma existencia de elemento neutro). □

El teorema anterior afirma que $\langle \Sigma^*, \cdot, \varepsilon \rangle$ es un monoide. ¿Será esto cierto si reemplazamos Σ^* por L?, es decir, ¿cualquier lenguaje L sobre un alfabeto Σ, con la operación de concatenación y la palabra vacía, forman un monoide? Los siguientes ejemplos responderán a la pregunta.

Ejemplo 3.25. Sea $L = \{ \alpha \in \Sigma^* \mid l(\alpha) = 3k, k \in \mathbb{N} \}$. Entonces $\langle L, \cdot, \varepsilon \rangle$ es un monoide. Veamos que $\cdot$ es cerrada bajo L. Sean $\alpha, \beta \in L$, entonces:

$$\begin{aligned} l(\alpha \cdot \beta) &= l(\alpha) + l(\beta) \\ &= 3k_1 + 3k_2; \quad k_1, k_2 \in \mathbb{N} \\ &= 3(k_1 + k_2); \quad k_1, k_2 \in \mathbb{N} \\ &= 3p, p \in \mathbb{N}, \quad \text{donde} \quad p = k_1 + k_2. \end{aligned}$$

Luego, $\cdot$ es cerrada bajo L. Además L hereda la asociativa de Σ^* y e opera como elemento neutro (ε, donde $l(\varepsilon) = 0 = 3(0)$ luego $\varepsilon \in L$). Podemos entonces afirmar que $\langle L, \cdot, \varepsilon \rangle$ es un monoide.

Ejemplo 3.26. Sea $L = \{\, \alpha \in \Sigma^* \mid l(\alpha) = 2k+1, k \in \mathbb{N} \,\}$. Entonces $\langle L, \cdot, \varepsilon \rangle$ no es un monoide. Veamos que $\cdot$ no es cerrada bajo L. Sean $\alpha, \beta \in L$ tales que, $l(\alpha) = 5$ y $l(\beta) = 7$, entonces:

$$\begin{aligned} l(\alpha \cdot \beta) &= l(\alpha) + l(\beta) \\ &= 5 + 7 = 12; \end{aligned}$$

pero, 12 no se puede expresar como $2k+1$ con $k \in \mathbb{N}$. Luego $\cdot$ no es cerrada bajo L.

3.1.4. Operaciones entre lenguajes

Sean L y L' dos lenguajes sobre un alfabeto Σ. Definimos las siguientes operaciones entre ellos:

1. Reflexión: $L^{-1} = \{\, \alpha^{-1} \mid \alpha \in L \,\}$.
2. Unión: $L \cup L' = \{\, \alpha \mid \alpha \in L \text{ o } \alpha \in L' \,\}$.
3. Intersección: $L \cap L' = \{\, \alpha \mid \alpha \in L \text{ y } \alpha \in L' \,\}$.
4. Complemento: $\overline{L} = \{\, \alpha \in \Sigma^* \mid \alpha \notin L \,\}$
5. Concatenación: $L \cdot L' = \{\, \alpha \cdot \beta \mid \alpha \in L \text{ y } \beta \in L' \,\}$.
6. Clausura de Kleene:

   La clausura de Kleene de un lenguaje L, denotada por L^*, es el conjunto de todas las palabras finitas construibles con los elementos de L (incluyendo la palabra vacía), esto es:

   $$L^* = \{\, \alpha \in \Sigma^* \mid \alpha = \alpha_1 \cdot \alpha_2 \cdot \ldots \cdot \alpha_n \text{ y } n \geq 0 \text{ y } \alpha_i \in L \,\}.$$

   Podemos crear la sucesión:

   $$\begin{aligned} L^0 &= \{\varepsilon\}, \\ L^{k+1} &= L^k \cdot L, \end{aligned}$$

   luego,

   $$L^* = \bigcup_{n \in \mathbb{N}} L^n.$$

   Observemos que si $L = \Sigma$, entonces, $L^* = \Sigma^*$.
7. Clausura positiva:

   La clausura positiva de un lenguaje L, denotada por L^+, es el conjunto de todas las palabras finitas construibles con los elementos de L, sin incluir la palabra vacía, es decir:

   $$L^+ = \bigcup_{n \in \mathbb{Z}^+} L^n, \quad \text{donde } \mathbb{Z}^+ = \{1, 2, 3, \ldots\}.$$

Ejemplo 3.27. Sea $L = \{otla, cola, avu, aa\}$ entonces $L^{-1} = \{alto, aloc, uva, aa\}$.

Ejemplo 3.28. Sea $L(\Sigma) = \{\alpha\}$, donde $\alpha \in \Sigma^*$ entonces $L(\Sigma) \cdot \Sigma^* = \{\, \alpha \cdot \beta \mid \beta \in \Sigma^* \,\}$. Este lenguaje es el conjunto de todas las palabras que tienen prefijo α.

Ejemplo 3.29. Si $\Sigma = \{0, 1\}$ y $L(\Sigma) = \{01, 1, 100\}$ entonces:

$$\begin{aligned}
L^0(\Sigma) &= \{\varepsilon\},\\
L^1(\Sigma) &= L(\Sigma) = \{01, 1, 100\},\\
L^2(\Sigma) &= \{0101, 011, 01100, 101, 11, 1100, 10001, 1001, 100100\$,\\
L^3(\Sigma) &= L^2(\Sigma) \cdot L(\Sigma),\\
L^*(\Sigma) &= \bigcup_{n\in\mathbb{N}} L^n(\Sigma).
\end{aligned}$$

Ejemplo 3.30. Sean $L = \{A, \ldots, Z, a, \ldots, z\}$ y $L' = \{0, \ldots, 9\}$, entonces:

$$\begin{aligned}
L \cup L' &= \{\, \alpha \mid \alpha \text{ es una letra o es un dígito} \,\},\\
LL' &= \{\, \alpha\beta \mid \alpha \text{ es una letra y } \beta \text{ es un dígito} \,\},\\
L^* &= \{\, \alpha \mid \alpha \text{ es una secuencia de letras (incluyendo la secuencia vacía)} \,\},\\
L'^+ &= \{\, \alpha \mid \alpha \text{ es una secuencia de dígitos (no incluye la secuencia vacía)} \,\},\\
L(L \cup L')^* &= \{\, \alpha \mid \alpha \text{ es una secuencia de letras o dígitos comenzando por una letra} \,\}.
\end{aligned}$$

3.2. Sistemas formales y sistemas combinatorios

Definición 3.31 (Sistema formal)**.** Un sistema formal es una estructura matemática

$$\mathsf{SF} = \langle \Sigma, \Gamma, \Psi, \Delta \rangle,$$

donde:

(i) Σ es un alfabeto.

(ii) Γ es un conjunto de fórmulas (obtenido a partir de unas reglas de formación de fórmulas).

(iii) Ψ es un conjunto de reglas de transformación de fórmulas.

(iv) Δ es un conjunto de axiomas.

Ejemplo 3.32. Definamos un sistema formal para la lógica de enunciados por

$$\mathsf{LE} = \langle \Sigma, \Gamma, \Psi, \Delta \rangle,$$

donde:

(i) $\Sigma = \{\neg, \vee, \wedge, \rightarrow, \leftrightarrow\} \cup \{\, p_i \mid i \in \mathbb{N}\,\} \cup \{(,)\}$.

(ii) Γ está determinado por las siguientes reglas de formación de fórmulas:

R1. $\forall i \in \mathbb{N}\ (p_i \in \Gamma)$.

R2. Si $\alpha \in \Gamma$ entonces $\neg(\alpha) \in \Gamma$.

R3. Si $\alpha, \beta \in \Gamma$ entonces $(\alpha) \vee (\beta) \in \Gamma$.

(iii) Ψ está formado por una única regla llamada regla de separación:

$$(\alpha \wedge (\alpha \rightarrow \beta)) \rightarrow \beta.$$

(iv) Δ es un conjunto de axiomas formado por:

A1. $(\alpha \vee \alpha) \rightarrow \alpha$.

A2. $\alpha \rightarrow (\alpha \vee \beta)$.

A3. $(\alpha \vee \beta) \rightarrow (\beta \vee \alpha)$.

A4. $(\alpha \vee \beta) \rightarrow ((\gamma \vee \alpha) \rightarrow (\gamma \vee \beta))$.

Definición 3.33 (Sistema combinatorio)**.** Un sistema combinatorio es una estructura matemática

$$\mathsf{SC} = \langle \Sigma_p, \Sigma_a, \Gamma, \Psi, \Delta \rangle,$$

donde:

(i) Σ_p es un alfabeto principal.

(ii) Σ_a es un alfabeto auxiliar.

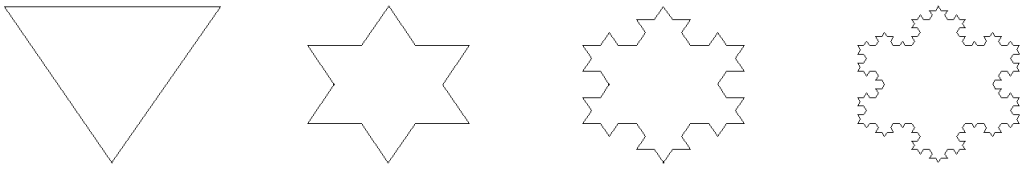
(iii) Γ es un conjunto de fórmulas (obtenido a partir de una reglas de formación de fórmulas).

(iv) Ψ es un conjunto de reglas de transformación de fórmulas.

(v) Δ es un conjunto unitario de axiomas.

Ejemplo 3.34. Si examinamos las figuras 3.1(a), 3.1(b), 3.1(c) y 3.1(d), podemos descubrir que cada figura se construye usando como base la figura anterior. Observemos el paso de la figura 3.1(a) a la figura 3.1(b); este paso es una transformación de cada uno de los segmentos de recta, que es la figura 3.1(a), en un conjunto de segmentos de recta. Si observamos la figura 3.1(c), veremos que ésta se obtiene de la figura 3.1(b) efectuando la misma transformación a cada uno de los segmentos de recta que la componen. La figura 3.1(d) es producto de hacer la misma transformación a la figura 3.1(c) y así, sucesivamente.

Éste es un ejemplo de lo que se conoce como un fractal (tipo L). Vamos a definir el fractal anterior conocido como curva de Koch por medio de un sistema combinatorio $\mathsf{SC} = \langle \Sigma_p, \Sigma_a, \Gamma, \Psi, \Delta \rangle$, donde:



(a) (b) (c) (d)

Figura 3.1: Curva de Koch.

(i) $\Sigma_p = \{L, +, -\}$. La interpretación que damos a los símbolos de Σ_p es la siguiente: L: segmento de recta, $+$: giro de 45° en el sentido de las manellicas del reloj y $-$: giro de 45° en sentido inverso al de las manecillas del reloj.

(ii) $\Sigma_a = \{\rightarrow\}$.

(iii) Γ es el conjunto de fractales obtenidos.

(iv) Ψ está compuesto por una única regla: $L \rightarrow L-L++L-L$.

(v) $\Delta = \{L\}$.

3.3. Gramáticas formales o de frase estructurada

3.3.1. Problema de la representación

Con respecto al tratamiento de los lenguajes (artificiales o naturales), un objetivo que se plantea es el de formalizarlos. Por lo general los lenguajes "interesantes" son infinitos; entonces surge el problema: ¿cómo representar un lenguaje infinito? Hay dos posibles soluciones, a saber:

1. Representación analítica: máquinas abstractas.

2. Representación generadora: sistema de generación finito con un conjunto finito de reglas (gramática generativa).

3.3.2. Gramáticas

Definición 3.35 (Gramática)**.** Una gramática está definida por una estructura matemática

$$\mathsf{G} = \langle \mathbf{N}, \mathbf{T}, \mathbf{P}, \mathbf{I} \rangle,$$

donde:

(i) $\mathbf{N}$ es un conjunto finito de símbolos llamados *no terminales*.

(ii) **T** es un conjunto finito de símbolos llamados *terminales*.

(iii) **P** es un conjunto de reglas de producción.

(iv) **I** es el axioma de inicio o símbolo inicial.

Los elementos de una gramática presentan las siguientes características:

G1. Los conjuntos **N** y **T** son disjuntos ($\mathbf{N} \cap \mathbf{T} = \emptyset$).

G2. $\mathbf{I} \notin (\mathbf{N} \cup \mathbf{T})$.

G3. Las producciones tienen la forma:

a) $\alpha A \beta \to \gamma \delta \eta$ donde $\alpha, \beta, \gamma, \delta, \eta \in (N \cup T)^*$; $A = \mathbf{I} \ \vee \ A \in \mathbf{N}$.

b) La palabra $\alpha A \beta$ recibe el nombre de "lado izquierdo" de la producción;.

c) La palabra $\gamma \delta \eta$ recibe el nombre de "lado derecho" de la producción.

d) El símbolo "$\to$" se lee: $\alpha A \beta$ deriva en $\gamma \delta \eta$.

Con respecto a la notación, se utilizan letras mayúsculas para denotar los símbolos no terminales y letras minúsculas para denotar los símbolos terminales.

Definición 3.36 (Lenguaje generado por una gramática)**.** El lenguaje generado por una gramática G, denotado por $L(\mathsf{G})$, es el conjunto de secuencias de símbolos terminales que se pueden derivar a partir de **I**, es decir, $L(\mathsf{G}) = \{ \alpha \in T^* \mid \mathbf{I} \to^* \alpha \}$ donde, $\to^*$ representa cero o más producciones, es decir, α se puede obtener por sucesivas derivaciones a partir de I. Para efectos de facilitar la notación (cuando no haya lugar a confusión), vamos a denotar $L(\mathsf{G})$ por L.

Ejemplo 3.37. Sea $\mathsf{G} = \langle \mathbf{N}, \mathbf{T}, \mathbf{P}, \mathbf{I} \rangle$ donde:

$\mathbf{I} = \mathbf{I}$,
$\mathbf{N} = \{$ORACIÓN, SUJETO, PREDICADO, ARTÍCULO, ADVERBIO, SUSTANTIVO, VERBO$\}$,
$\mathbf{T} = \{$la, el, mujer, hombre, lee, escribe, adecuadamente, inadecuadamente$\}$.

El conjunto **P** está formado por las siguientes reglas de producción:

$$\begin{aligned}
P_1 &: \mathbf{I} && \to \text{ORACIÓN}\\
P_2 &: \text{ORACIÓN} && \to \text{SUJETO PREDICADO}\\
P_3 &: \text{SUJETO} && \to \text{ARTÍCULO SUSTANTIVO}\\
P_4 &: \text{ARTÍCULO} && \to \text{el}\\
P_5 &: \text{ARTÍCULO} && \to \text{la}\\
P_6 &: \text{SUSTANTIVO} && \to \text{hombre}\\
P_7 &: \text{SUSTANTIVO} && \to \text{mujer}\\
P_8 &: \text{PREDICADO} && \to \text{VERBO ADVERBIO}\\
P_9 &: \text{VERBO} && \to \text{lee}\\
P_{10} &: \text{VERBO} && \to \text{escribe}\\
P_{11} &: \text{ADVERBIO} && \to \text{adecuadamente}\\
P_{12} &: \text{ADVERBIO} && \to \text{inadecuadamente}
\end{aligned}$$

Observemos que cada una de las producciones satisface el formato definido para las mismas.
Las palabras (en realidad son oraciones) que componen a L son:
$\alpha_1 \equiv$ la mujer lee adecuadamente
$\alpha_2 \equiv$ la mujer lee inadecuadamente
$\alpha_3 \equiv$ la mujer escribe adecuadamente
$\alpha_4 \equiv$ la mujer escribe inadecuadamente
$\alpha_5 \equiv$ el hombre lee adecuadamente
$\alpha_6 \equiv$ el hombre lee inadecuadamente
$\alpha_7 \equiv$ el hombre escribe adecuadamente
$\alpha_8 \equiv$ el hombre escribe inadecuadamente

Observemos que por ejemplo:
$\beta \equiv$ "el hombre lee", no es una palabra de L. Para que lo fuera, sería necesario que ADVERBIO se pudiera derivar en vacío.

Ejemplo 3.38. Sea $\mathsf{G} = \langle \{A, B\}, \{0, 1\}, \mathbf{P}, \mathbf{I} \rangle$, donde **P** está formado por las siguientes reglas de producción:

$$\begin{aligned}
\mathbf{I} &\to 1B\\
\mathbf{I} &\to 1\\
B &\to 0A\\
A &\to 1B\\
A &\to 1
\end{aligned}$$

De acuerdo con las reglas de producción, tenemos que:

$$\mathbf{I} \to 1B \to 10A \to 101B \to 1010A \to \ldots \to 10\ldots101;$$

es decir, $L = (10)^*1$.

Ejemplo 3.39. Sea $\mathsf{G} = \langle \{A\}, \{a, b\}, \{\mathbf{I} \to A, A \to aAb, A \to ab\}, \mathbf{I} \rangle$. Con base en las reglas de producción, tenemos:

$$\mathbf{I} \to A \to aAb \to aaAbb \to aaaAbbb \to \ldots \to a_1 a_2 \ldots a_n a b b_1 b_2 \ldots b_n;$$

es decir, $L = \{\, a^n b^n \mid n \geq 1 \,\}$.

Ejemplo 3.40. Sea G una gramática definida por las siguientes producciones:

$$\begin{aligned} \mathbf{I} &\xrightarrow{1} A \\ A &\xrightarrow{2} aABC \\ A &\xrightarrow{3} abC \\ CB &\xrightarrow{4} BC \\ bB &\xrightarrow{5} bb \\ bC &\xrightarrow{6} bc \\ cC &\xrightarrow{7} cc \end{aligned}$$

El lenguaje asociado con esta gramática es, $L = \{\, a^n b^n c^n \mid n \geq 1 \,\}$. Construyamos inicialmente la derivación para la palabra $\alpha \equiv aabbcc$. En este caso indicamos la producción utilizada en cada paso de la derivación.

$$\mathbf{I} \xrightarrow{1} A \xrightarrow{2} aABC \xrightarrow{3} aabCBC \xrightarrow{4} aabBCC \xrightarrow{5} aabbCC \xrightarrow{6} aabbcC \xrightarrow{7} aabbcc.$$

Ejemplo 3.41. Sea G una gramática definida por las siguientes producciones:

$$\begin{aligned} \mathbf{I} &\xrightarrow{1} ACaB \\ Ca &\xrightarrow{2} aaC \\ CB &\xrightarrow{3} DB \\ CB &\xrightarrow{4} E \\ AD &\xrightarrow{5} AC \\ aD &\xrightarrow{6} Da \\ aE &\xrightarrow{7} Ea \\ AE &\xrightarrow{8} \varepsilon \end{aligned}$$

El lenguaje asociado con esta gramática es $L = \{a^{2^k}\}$. Para $a^{2^2} = a^4$ se obtiene la siguiente derivación:
$\mathbf{I} \xrightarrow{1} ACaB \xrightarrow{2} AaaCB \xrightarrow{3} AaaDB \xrightarrow{6} AaDaB \xrightarrow{6} ADaaB \xrightarrow{5} ACaaB \xrightarrow{2} AaaCaB \xrightarrow{2} AaaaaCB \xrightarrow{4} AaaaaE \xrightarrow{7} AaaaEa \xrightarrow{7} AaaEaa \xrightarrow{7} AaEaaa \xrightarrow{7} AEaaaa \xrightarrow{8} aaaa$.

3.3.3. Taxonomía de las gramáticas

Sean $\mathsf{G} = \langle \mathbf{N}, \mathbf{T}, \mathbf{P}, \mathbf{I} \rangle$ una gramática, $\alpha, \beta, \gamma \in (\mathbf{N} \cup \mathbf{T})^*$, $A \in \mathbf{N} \cup \{\mathbf{I}\}$, $B \in \mathbf{N}$, $a \in \mathbf{T}$, y ε es la palabra vacía.

La taxonomía de las gramáticas está sustentada sobre el patrón de producciones que pueden tener, tal como está indicado en la tabla 3.1.

Tipo	Nombre	Producciones	Comentarios
0	Gramáticas no restringidas	$\alpha A \beta \to \alpha \gamma \beta$.	Admiten producciones que implican decrecimiento. La única restricción, es que no permite producciones de la forma $\varepsilon \to \gamma$.
1	Gramáticas sensibles al contexto	$\alpha A \beta \to \alpha \gamma \beta$; $\gamma \neq \varepsilon$.	No tiene producciones compresoras. Admite $\mathbf{I} \to \varepsilon$ (elegancia en el lenguaje que genera).
2	Gramáticas independientes del contexto	$A \to \gamma$; $\gamma \neq \varepsilon$.	El contexto es obligatoriamente vacío. De este tipo son la mayoría de los lenguajes de programación. Admite $\mathbf{I} \to \varepsilon$.
3	Gramáticas regulares, gramáticas de Kleene o k-gramáticas	Lineales o recursivas a la derecha: $A \to aB$ y $A \to a$. Lineales o recursivas a la izquierda: $A \to Ba$ y $A \to a$.	Pueden contener a lo sumo un símbolo no terminal en la parte derecha de la producción. Admite $\mathbf{I} \to \varepsilon$.

Tabla 3.1: Taxonomía de las gramáticas.

Para cada tipo de gramática existe un tipo de lenguaje y para cada tipo de lenguaje existe un tipo de máquina (abstracta) que reconoce las palabras de ese lenguaje. La tabla 3.2 esquematiza esta situación.

Finalmente diremos que $\mathsf{G}_3 \subseteq \mathsf{G}_2 \subseteq \mathsf{G}_1 \subseteq \mathsf{G}_0$, luego, $L(\mathsf{G}_3) \subseteq L(\mathsf{G}_2) \subseteq L(\mathsf{G}_1) \subseteq L(\mathsf{G}_0)$.

Ejemplo 3.42. La gramática del ejemplo 3.38 es una gramática tipo 3; la del ejemplo 3.39 es una tipo 2; la del ejemplo 3.40 es una tipo 1 y la del ejemplo 3.41 es una tipo 0.

Tipo gramática	**Tipo lenguaje**	**Reconocedor**
Gramáticas no restringidas (tipo 0) G_0.	Lenguajes no restringidos $L(\mathsf{G}_0)$.	Máquina de Turing.
Gramáticas sensibles al contexto (tipo 1) G_1.	Lenguajes sensibles al contexto $L(\mathsf{G}_1)$.	Autómatas linealmente independientes.
Gramáticas independientes del contexto (tipo 2) G_2.	Lenguajes independientes del contexto $L(\mathsf{G}_2)$.	Autómatas de pila (*stack*).
Gramáticas regulares (tipo 3) G_3.	Lenguajes regulares $L(\mathsf{G}_3)$.	Autómatas de estado finito.

Tabla 3.2: Gramáticas, lenguajes y reconocedores.

3.3.4. Notación alternativa para las gramáticas

Definición 3.43 (Notación BNF)**.** Un tipo de notación muy utilizada en informática para esfecificar gramáticas es la notación BNF (Backus - Nour Form):

BNF1. Los símbolos no terminales se encierran entre <>.

BNF2. Las producciones tienen la forma $\alpha ::= \beta$.

BNF3. Si existen varias derivaciones de un mismo "símbolo", éstas se representan por $\alpha ::= \beta_1 \mid \cdots \mid \beta_n$.

Ejemplo 3.44.

$$\begin{aligned} \mathbf{I} ::= &< \text{lista} > \\ < \text{lista} > ::= &< \text{lista} > + < \text{dígito} > \\ &\mid < \text{lista} > - < \text{dígito} > \\ &\mid < \text{dígito} > \\ < \text{dígito} > ::= &\ 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

Algunas palabras son: $\alpha_1 \equiv 3 - 3 + 2$, $\alpha_2 \equiv 1 + 2 - 3$, etc.

Ejemplo 3.45.

$$
\begin{aligned}
\mathbf{I} &::= < \text{entero} > \\
< \text{entero} > &::= < \text{entero-con-signo} > \\
&\mid < \text{entero-sin-signo} > \\
< \text{entero-con-signo} > &::= + < \text{entero-sin-signo} > \\
&\mid - < \text{entero-sin-signo} > \\
< \text{entero-sin-signo} > &::= < \text{dígito} > \\
&\mid < \text{dígito} >< \text{entero-sin-signo} > \\
< \text{dígito} > &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Algunas palabras son: -344, 567, +9784, 8, 0000, etc.

3.3.5. Algunos aspectos sobre las gramáticas

Definición 3.46 (Árbol de análisis sintáctico)**.** El objetivo del árbol de análisis sintáctico es ilustrar la derivación de una cadena del lenguaje a partir del símbolo inicial de la gramática. Este árbol también es llamado árbol de *parsing* o árbol de derivación. La construcción del árbol de análisis sintáctico está dirigida por las siguientes reglas:

1. La raíz es el símbolo inicial.
2. Las hojas son símbolos terminales.
3. Los nodos interiores son símbolos no terminales.
4. Si α es un nodo interior y $\beta_1, \beta_2, \ldots, \beta_n$ son sus hijos de izquierda a derecha, entonces $\alpha \to \beta_1\beta_2 \ldots \beta_n$ es una producción.

Ejemplo 3.47. Sea G una gramática definida por las siguientes producciones:

$$
\begin{aligned}
&\mathbf{I} \to E \\
&E \to E + E \mid \ E - E \mid \ D \\
&D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

La figura 3.2 representa el árbol de análisis sintáctico para la palabra $\alpha \equiv 1 + 2 + 3$.

La definición que realizamos del árbol de análisis sintáctico es válida únicamente para gramáticas tipo 2 o gramáticas tipo 3, es decir, gramáticas para las cuales el contexto del lado izquierdo de cualquier producción es vacío.

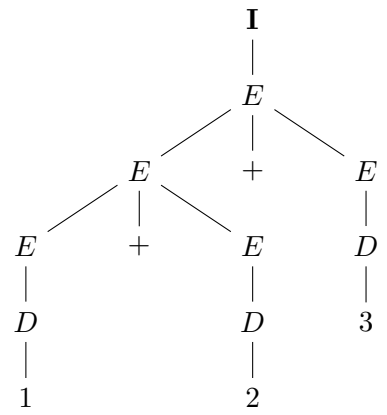


Figura 3.2: Árbol de análisis sintáctico para $\alpha \equiv 1 + 2 + 3$.

Definición 3.48 (Derivación: izquierda y derecha)**.** La derivación por izquierda consiste en sustituir en cada paso de la derivación de una cadena el símbolo no terminal más a la izquierda perteneciente a ella. Similarmente, la derivación por derecha consiste en sustituir en cada paso de la derivación de una cadena el símbolo no terminal más a la derecha perteneciente a ella.

Ejemplo 3.49. Sea G una gramática definida por las siguientes producciones:

$$\begin{aligned}
&\mathbf{I} \to E \\
&E \to E + E \\
&\quad \mid (E) \\
&\quad \mid -E \\
&\quad \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}$$

Sea $\alpha \equiv -(3 + 1)$.

Derivación por la izquierda:

$$\begin{aligned}
\mathbf{I} &\to E \\
&\to -E \\
&\to -(E) \\
&\to -(E + E) \\
&\to -(3 + E) \text{ se sustituyó el símbolo no terminal más a la izquierda} \\
&\to -(3 + 1)
\end{aligned}$$

Derivación por la derecha:

$$
\begin{aligned}
\mathbf{I} &\to E \\
&\to -E \\
&\to -(E) \\
&\to -(E+E) \\
&\to -(E+1) \text{ se sustituyó el símbolo no terminal más a la derecha} \\
&\to -(3+1)
\end{aligned}
$$

La figura 3.3 representa los árboles de derivación para la derivación por la izquierda y para la derivación por la derecha que hemos realizado. Es decir, sin importar qué derivación realizamos, obtenemos para la palabra $\alpha \equiv -(3+1)$ el mismo árbol de análisis sintáctico.

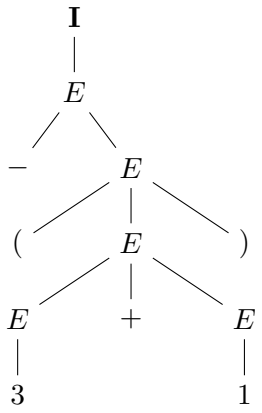


Figura 3.3: Árbol de derivación por la derecha y por la izquierda para $\alpha \equiv -(3+1)$.

Definición 3.50 (Ambigüedad)**.** Sea $\mathsf{G} = \langle \mathbf{N}, \mathbf{T}, \mathbf{P}, \mathbf{I} \rangle$ una gramática. Se dice que G es ambigua si (las siguientes condiciones son equivalentes):

1. Existe $\alpha \in L(\mathsf{G})$ tal que $\mathbf{I} \overset{*}{\to}_1 \alpha$ y $\mathbf{I} \overset{*}{\to}_2 \alpha$, son dos derivaciones diferentes para α. Es decir existen $\beta_1, \beta_2, \ldots, \beta_n$ y $\gamma_1, \gamma_2, \ldots, \gamma_m$ tales que:
   $I \to \alpha_1 \wedge \alpha_1 \to \beta_2 \wedge \cdots \wedge \beta_{n-1} \to \beta_n \wedge \beta_n \to \alpha$ y
   $I \to \gamma_1 \wedge \gamma_1 \to \gamma_2 \wedge \cdots \wedge \gamma_{m-1} \to \gamma_m \wedge \gamma_m \to \alpha$.

2. Existe más de una derivación por el mismo sentido para una palabra $\alpha \in L(\mathsf{G})$.

3. Una palabra $\alpha \in L(\mathsf{G})$ tiene más de un árbol de análisis sintáctico.

Ejemplo 3.51. Sea G una gramática definida por las siguientes producciones:

$$\begin{aligned} \mathbf{I} &\to E \\ E &\to E + E \mid \ E - E \\ &\mid \ 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

La figura 3.4 representa dos árboles de análisis sintáctico para la palabra $\alpha \equiv 1 - 2 + 3$. Luego la gramática G es ambigua.

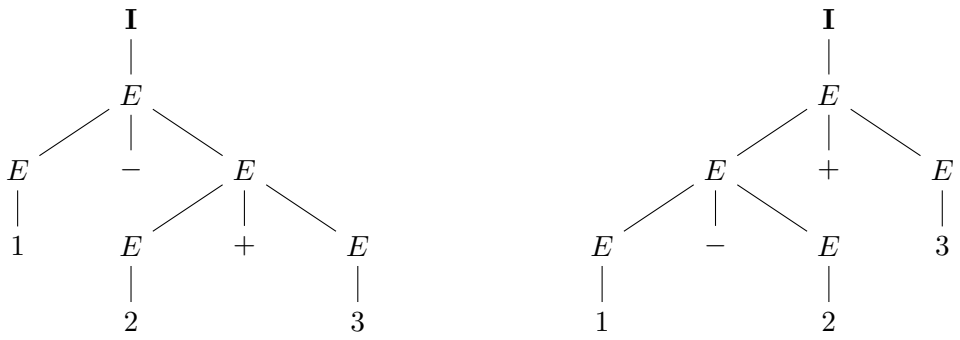


Figura 3.4: Árboles de análisis sintáctico para $\alpha \equiv 1 - 2 + 3$.

Ejemplo 3.52. Sea G la gramática para el **if-else** definida por:

$$\begin{aligned} \mathbf{I} &\to \text{PROP} \\ \text{PROP} &\to \text{if EXP then PROP} \\ \text{PROP} &\to \text{if EXP then PROP else PROP} \\ \text{PROP} &\to \text{OTRA} \\ \text{PROP} &\to p_1 \mid p_2 \\ \text{EXP} &\to e_1 \mid e_2 \\ \text{OTRA} &\to \text{PROP} \end{aligned}$$

Sea $\alpha \equiv$ if e_1 then if e_2 then p_1 else p_2. Las figuras 3.5 y 3.6 representan dos árboles de derivación para α.

De acuerdo con lo anterior, la gramática G para el **if-else** es ambigua. El problema consiste en el emparejamiento del **else**. La solución es formalizar en la gramática la siguiente regla: emparejar cada **else** con el **if** más cercano (está es la regla usual que implementan los lenguajes de programación).

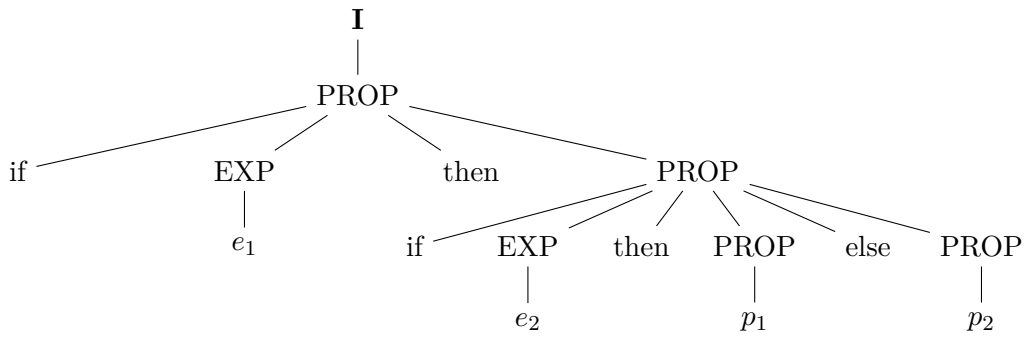


Figura 3.5: Primer árbol de análisis sintáctico con base en una gramática ambigua para $\alpha \equiv$ if e_1 then if e_2 then p_1 else p_2.

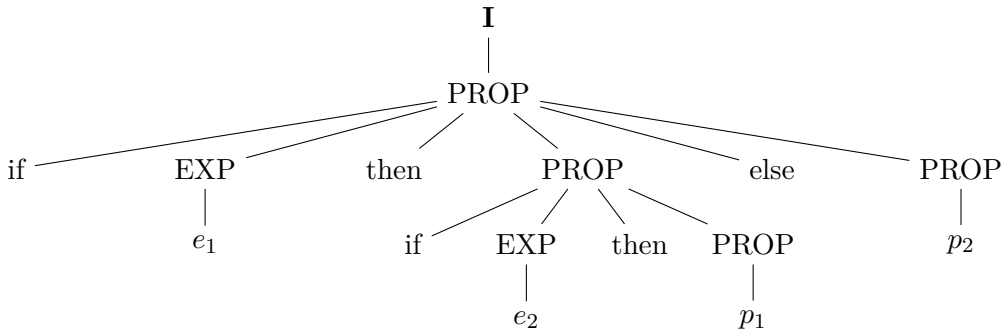


Figura 3.6: Segundo árbol de análisis sintáctico con base en una gramática ambigua para $\alpha \equiv$ if e_1 then if e_2 then p_1 else p_2.

Entonces, la gramática G para el **if-else** no ambigua está definida por:

$$
\begin{aligned}
\mathbf{I} &\to \text{PROP} \\
\text{PROP} &\to \text{IF} \\
\text{PROP} &\to \text{IFE} \\
\text{IFE} &\to \text{if EXP then IFE else IFE} \\
\text{IFE} &\to \text{OTRA} \\
\text{IF} &\to \text{if EXP then PROP} \\
\text{IF} &\to \text{if EXP then IFE elseIFE} \\
\text{EXP} &\to e_1 \mid e_2 \\
\text{OTRA} &\to p_1 \mid p_2
\end{aligned}
$$

La figura 3.7 representa el árbol de *parsing* para la palabra:

$\alpha \equiv$ if e_1 then if e_2 then p_1 else p_2.

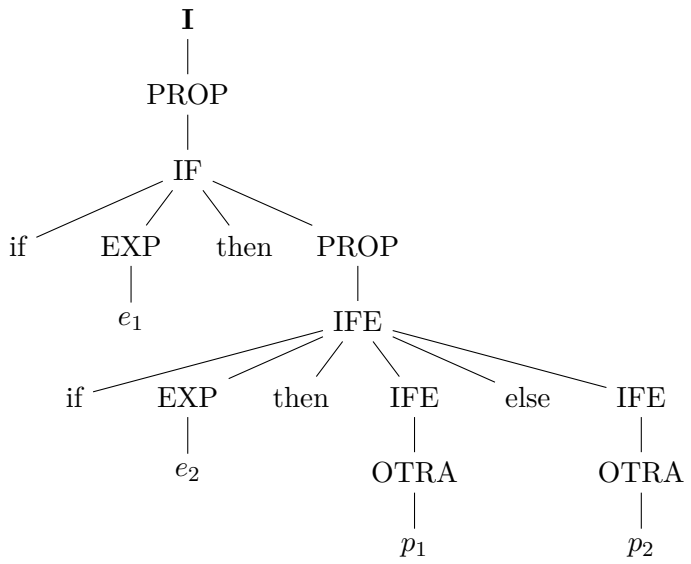


Figura 3.7: Árbol de análisis sintáctico con base en una gramática no ambigua para $\alpha \equiv$ if e_1 then if e_2 then p_1 else p_2.

Definición 3.53 (Recursividad)**.** La recursividad de un gramática es el sentido en el cual crece el árbol de análisis sintáctico.

Ejemplo 3.54. Sea $L = \{\, \alpha\beta \mid \alpha \equiv 1$ y β es una secuencia de cero o más $01 \,\}$ un lenguaje. Para L vamos a construir una gramática recursiva por la derecha y una gramática recursiva por la izquierda:

1. Gramática recursiva por la derecha:

$$\begin{aligned} \mathbf{I} &\to 1B \mid 1 \\ A &\to 1B \mid 1 \\ B &\to 0A \end{aligned}$$

Analizando estas producciones, podemos realizar la siguiente simplificación:

$$\begin{aligned} \mathbf{I} &\to A \\ A &\to 1B \mid 1 \\ B &\to 0A \end{aligned}$$

Y analizando de nuevo las producciones obtenidas, podemos realizar la siguiente simplificación:

$$\begin{aligned} \mathbf{I} &\to A \\ A &\to 10A \mid 1 \end{aligned}$$

Con lo cual podemos observar que la gramática crece por la derecha. En este caso $L = (10)^*1$. Para la palabra $\alpha \equiv 10101$, el árbol de análisis sintáctico, el cual refleja la recursividad por la derecha, está representado por la figura 3.8.

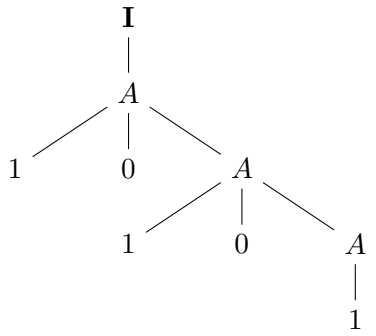


Figura 3.8: Recursividad por la derecha para $\alpha \equiv 10101$.

2. Gramática recursiva por la izquierda:

$$\begin{aligned} \mathbf{I} &\to B1 \mid 1 \\ A &\to B1 \mid 1 \\ B &\to A0 \end{aligned}$$

Analizando estas producciones podemos realizar la siguiente simplificación:

$$\begin{aligned} \mathbf{I} &\to A \\ A &\to B1 \mid 1 \\ B &\to A0 \end{aligned}$$

Y analizando de nuevo las producciones obtenidas, podemos realizar la siguiente simplificación:

$$\begin{aligned} I &\to A \\ A &\to A01 \mid 1 \end{aligned}$$

Así podemos observar que la gramática crece por la izquierda. En este caso, $L = 1(01)^*$. Para la palabra $\alpha \equiv 10101$, el árbol de análisis sintáctico, que refleja la recursividad por la izquierda, está representado por la figura 3.9.

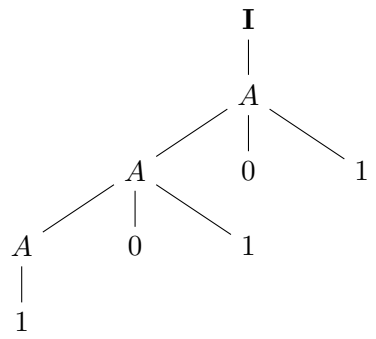


Figura 3.9: Recursividad por la izquierda para $\alpha \equiv 10101$.

3.3.6. Algunos teoremas sobre gramáticas

Teorema 3.55. *Una gramática es sensible al contexto (tipo 1), si y sólo si para las producciones de la forma $\alpha \to \beta$, se tiene que $l(\alpha) \leq l(\beta)$.*

Demostración. Si una gramática G es de tipo 1, las producciones son de la forma: $\alpha A\beta \to \alpha\gamma\beta$ $(\gamma \neq \varepsilon)$; donde $\alpha, \beta, \gamma \in (\mathbf{N} \cup \mathbf{T})^*$, $A \in \mathbf{N} \cup \mathbf{I}$, y ε es la palabra vacía, entonces:

$$l(\alpha A\beta) = l(\alpha) + l(A) + l(\beta);$$

como $l(A) = 1$ y $l(\gamma) \geq 1$, porque $\gamma \neq \varepsilon$,

$$\begin{aligned} l(\alpha A\beta) &\leq l(\alpha) + l(\gamma) + l(\beta) \\ &\leq l(\alpha\gamma\beta). \end{aligned}$$

□

Teorema 3.56. *Si G es una gramática lineal derecha, entonces existe una gramática lineal derecha G', tal que G' es equivalente a G y G' no contiene producciones de la forma $A \to b\mathbf{I}$, donde $\mathbf{I}$ es el símbolo inicial, ni contiene producciones de la forma $A \to \varepsilon$, donde ε es la palabra vacía.*

Demostración.

1. Eliminación de las producciones $A \to \varepsilon$

   Sea G con las siguientes producciones:

$$\begin{aligned} \mathbf{I} &\to bB \\ B &\to aB \\ B &\to \varepsilon \end{aligned}$$

Entonces G' queda así:

$$\begin{aligned}\mathbf{I} &\to bB\\ \mathbf{I} &\to b\\ B &\to aB\\ B &\to a\end{aligned}$$

2. Eliminación de las producciones $A \to b\mathbf{I}$

   Sea G con las siguientes producciones:

$$\begin{aligned}\mathbf{I} &\to bA\\ A &\to a\mathbf{I}\\ A &\to a\end{aligned}$$

   Entonces G' queda así:

$$\begin{aligned}\mathbf{I} &\to bA\\ B &\to bA\\ A &\to aB\\ A &\to a\end{aligned}$$

□

Teorema 3.57. *Para toda gramática lineal derecha (tipo 3) existe una gramática lineal izquierda (tipo 3) equivalente.*

Demostración. El teorema se demuestra con base en la construcción de grafos asociados con las gramáticas. La gramática lineal derecha no debe contener producciones de la forma $A \to b\mathbf{I}$, ni producciones de la forma $A \to \varepsilon$; el cumplimiento de estas restricciones se puede garantizar por el resultado del teorema 3.56.

Inicialmente indicamos la construcción de un digrafo DG para una gramática lineal derecha G:

1. Nodos: Símbolos no terminales, más $\mathbf{I}$ y ε.
2. Arcos: Existe un arco del nodo v_1 al nodo v_2, etiquetado con c, si y sólo si, $v_1 \to cv_2$ es una producción de G.

En este caso la producción de las palabras es de izquierda a derecha. Para el digrafo DG, cambiamos I por ε e invertimos los arcos. Así obtenemos un nuevo digrafo DG'. El digrafo DG' corresponde al digrafo de una gramática lineal izquierda G' equivalente a la gramática lineal derecha G. La lectura del digrafo DG' se realiza bajo las siguientes convenciones:

1. Nodos: Símbolos no terminales, más $\mathbf{I}$ y ε.

2. Arcos: Existe un arco del nodo v_1 al nodo v_2, etiquetado con c, si y sólo si, $v_1 \to v_2c$ es una producción de G'.

En este caso la producción de las palabras es de derecha a izquierda. Entonces, a partir del digrafo DG' se puede obtener la gramática lineal izquierda G' equivalente a la gramática lineal derecha G. □

Ejemplo 3.58. Sea G una gramática lineal derecha:

$$\begin{aligned} \mathbf{I} &\to 1B \mid 1 \\ A &\to 1B \mid 1 \\ B &\to 0A, \end{aligned}$$

donde $L = (10)^*1$.

La figura 3.10 representa el digrafo DG para G. Entonces cambiamos **I** por ε e invertimos los arcos, con lo cual obtenemos un nuevo digrafo DG', representado por la figura 3.11.

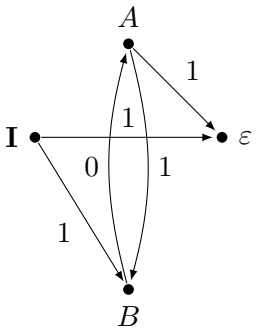


Figura 3.10: Digrafo para la gramática lineal derecha G.

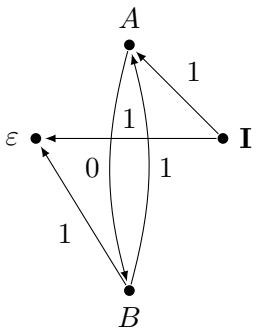


Figura 3.11: Digrafo para la gramática lineal izquierda G'.

De acuerdo con la figura 3.11, tenemos que la gramática lineal izquierda G' está formada por las siguientes producciones:

$$\begin{aligned} \mathbf{I} &\to A1 \mid 1 \\ A &\to B0 \\ B &\to 1 \mid A1, \end{aligned}$$

donde $L = 1(01)^*$.

Entonces G es un gramática lineal derecha y G' es la gramática lineal izquierda equivalente a G, obtenida por el método expuesto en el teorema 3.57.

3.4. Expresiones regulares

Las expresiones regulares introducidas por Stephen Kleene para denotar conjuntos regulares (lenguajes regulares) y son de uso frecuente en la descripción de la sintáxis de los lenguajes de programación.

Definición 3.59 (Expresión regular)**.** Presentamos una definición recursiva para las expresiones regulares. Sea Σ un alfabeto, entonces:

1. ε (palabra vacía) es un expresión regular de Σ.
2. Si $r \in \Sigma$, entonces r es un expresión regular de Σ.
3. Si r y s son expresiones regulares que denotan los lenguajes $L(r)$ y $L(s)$ respectivamente, entonces:
   - *a)* $r \mid s$ es un expresión regular, que denota la unión de $L(r)$ y $L(s)$.
   - *b)* $(r)(s)$ es un expresión regular, que denota la concatenación de $L(r)$ y $L(s)$.
   - *c)* $(r)^*$ es un expresión regular, que denota la clausura de Kleene de $L(r)$.

La precedencia de los operadores de mayor a menor es $^*, (), |$. Además todos los operadores son asociativos por la izquierda, por ejemplo $(a) \mid ((b*))(c) \equiv a \mid b^*c$.

Ejemplo 3.60. Sea $\Sigma = \{a, b\}$, entonces:

1. Si $r \equiv a \mid b$ entonces $L(r) = \{a, b\}$.
2. Si $r \equiv (a \mid b)(a \mid b)$ entonces $L(r) = \{aa, ab, ba, bb\}$.
3. Si $r \equiv a*$ entonces $L(r) = \{\varepsilon, a, aa, aaa, \dots\}$.
4. Si $r \equiv (a|b)*$ entonces $L(r) = \{$todas las cadenas de a y b incluyendo $\varepsilon\}$.
5. Si $r \equiv a|a*b$ entonces $L(r) = \{a, b, ab, aab, aaab, \dots\}$.

Definición 3.61 (Definición regular)**.** Una definición regular tiene el formato:

$$\text{nombre} \to \text{expresión regular}$$

Ejemplo 3.62. La definición regular para un identificador viene dada por:

$$\begin{aligned} \text{identificador} &\to \text{letra(letra } \mid \text{ dígito)}^* \\ \text{letra} &\to A \mid \cdots \mid Z \mid a \mid \cdots \mid z \\ \text{dígito} &\to 0 \mid \cdots \mid 9 \end{aligned}$$

Ejemplo 3.63. La definición regular para un número real sin signo viene dada por:

$$\begin{aligned} \text{dígito} &\to 0 \mid \cdots \mid 9 \\ \text{dígitos} &\to \text{dígito dígito}^* \\ \text{decimal} &\to .\text{dígitos} \mid \varepsilon \\ \text{exponente} &\to (E(+ \mid - \mid \varepsilon) \text{ dígitos}) \mid \varepsilon \\ \text{número} &\to \text{dígitos decimal exponente} \end{aligned}$$

Abreviación en la notación:

1. Uno o más casos: $+$.
2. Cero o más casos: *.
3. Cero o un caso: $?$.
4. Clase de caracteres: $[abc] \equiv (a \mid b \mid c$ y $[a - z] \equiv (a \mid \ldots \mid z)$.

Ejemplo 3.64. La definición regular para un número real sin signo, usando abreviaciones en la notación, viene dada por:

$$\begin{aligned} \text{dígito} &\to [0 - 9] \\ \text{dígitos} &\to \text{dígito}+ \\ \text{decimal} &\to .\text{dígitos}? \\ \text{exponente} &\to (E(+ \mid -)? \text{ dígitos})? \\ \text{número} &\to \text{dígitos decimal exponente} \end{aligned}$$

3.5. Ejercicios

Ejercicio 3.1. Sea $\Sigma = \{a, b, c, d, e\}$:

1. ¿Cuál es el cardinal de Σ^2 y de Σ^3?

2. ¿Cuántas palabras de Σ^* tienen una longitud de al menos cinco?

Ejercicio 3.2. Para $\Sigma = \{w, x, y, z\}$, determine el número de palabras de Σ^* de longitud cinco tal que:

1. Que comiencen por w.
2. Con precisamente dos w.
3. Sin w.
4. Con un número par de w.

Ejercicio 3.3. Si $\alpha \in \Sigma^*$ y $l(\alpha^4) = 36$ (longitud), ¿cuánto es $l(\alpha)$?

Ejercicio 3.4. Sea $\Sigma = \{\beta, x, y, z\}$, donde β denota un espacio en blanco, de modo que $x\beta \neq x$, $\beta\beta \neq \beta$ y $x\beta y \neq xy$, pero $x\varepsilon y = xy$. Calcule lo siguiente:

1. $l(\varepsilon)$.
2. $l(\varepsilon\varepsilon)$.
3. $l(\beta)$.
4. $l(\beta\beta)$.
5. $l(\beta^3)$.
6. $l(x\beta\beta y)$.
7. $l(\beta\varepsilon)$.
8. $l(\varepsilon^{10})$.

Ejercicio 3.5. Para el alfabeto $\Sigma = \{0, 1\}$, sean $A, B, C \in \Sigma^*$ los siguientes lenguajes:

$$
\begin{aligned}
A &= \{0, 1, 00, 11, 000, 111, 0000, 1111\}, \\
B &= \{\, \alpha \in \Sigma^* \mid 2 \leq l(\alpha) \,\}, \\
C &= \{\, \alpha \in \Sigma^* \mid 2 \geq l(\alpha) \,\}.
\end{aligned}
$$

Determine los siguientes lenguajes de Σ^*:

1. $A \cup B$.
2. $A - B$.
3. $A \triangle B$.

4. $A \cap B$.
5. $B \cap C$.
6. $B \cup C$.
7. $(A \cap C)^*$.
8. $A^* \cap C^*$.
9. $A^* \cap B^*$.

Ejercicio 3.6. Sean $A = \{10, 11\}$, $B = \{00, 1\}$ lenguajes para el alfabeto $\Sigma = \{0, 1\}$. Determine los siguientes lenguajes:

1. AB.
2. BA.
3. A^2.
4. B^2.

Ejercicio 3.7. Demostrar el teorema 3.17.

Ejercicio 3.8. Considere las siguientes gramáticas:

1. Gramática G_1:

$$\begin{aligned} \mathbf{I} &\to \varepsilon \\ \mathbf{I} &\to S \\ S &\to SS \\ S &\to c \end{aligned}$$

2. Gramática G_2:

$$\begin{aligned} \mathbf{I} &\to \varepsilon \\ \mathbf{I} &\to S \\ S &\to cSd \\ S &\to cd \end{aligned}$$

3. Gramática G_3:

$$\begin{aligned}
\mathbf{I} &\to \varepsilon \\
\mathbf{I} &\to S \\
S &\to Sd \\
S &\to cS \\
S &\to c \\
S &\to d
\end{aligned}$$

4. Gramática G_4:

$$\begin{aligned}
\mathbf{I} &\to cS \\
S &\to d \\
S &\to cS \\
S &\to Td \\
T &\to Td \\
T &\to d
\end{aligned}$$

5. Gramática G_5:

$$\begin{aligned}
\mathbf{I} &\to \varepsilon \\
\mathbf{I} &\to S \\
S &\to ScS \\
S &\to c
\end{aligned}$$

a. Describir $L(\mathsf{G}_i)$ para $i = 1, 2, 3, 4, 5$.

b. Indicar cualquier inclusión $L(\mathsf{G}_i)$.

c. Para cada lenguaje $L(\mathsf{G}_i)$, dar una derivación de una palabra de longitud 4.

Ejercicio 3.9. Construya una gramática que genere cada uno de los siguientes lenguajes:

1. $\{\, 0^m 1^n \mid m \geq n \geq 0 \,\}$.
2. $\{\, 0^m 1^n \mid$ m impar y n par o n impar y m par $\,\}$.
3. $\Big\{\, 0^k 1^m 0^n \mid n = k + m \,\Big\}$.
4. $\{\, wcw \mid w \in \{0,1\}^* \,\}$.

Ejercicio 3.10. ¿Es posible construir el árbol de análisis sintáctico para la gramática G que genera el lenguaje $L(\mathsf{G}) = \{\, a^n b^n c^n \mid n \geq 1 \,\}$ presentada en el ejemplo 3.40? Por qué?

Ejercicio 3.11. Demuestre que las siguientes gramáticas son ambiguas:

1. Gramática G_1:

$$\begin{aligned} \mathbf{I} &\to A \\ A &\to A0A \\ A &\to 1 \end{aligned}$$

2. Gramática G_2:

$$\begin{aligned} \mathbf{I} &\to A \\ A &\to B0 \\ A &\to A0 \\ B &\to B0 \\ A &\to 1 \\ B &\to 1 \end{aligned}$$

3. Gramática G_3:

$$\begin{aligned} \mathbf{I} &\to S \\ S &\to bA \mid aB \\ A &\to a \mid aS \mid bAA \\ B &\to b \mid bS \mid aBB \end{aligned}$$

Ejercicio 3.12. Para la gramática G_3 del ejercicio 3.11, construir una gramática no ambigua.

Ejercicio 3.13. Obtener una derivación por la izquierda para la palabra $\alpha \equiv abaca$, que sea distinta a la derivación por la derecha siguiente:

$$\mathbf{I} \to S \to ScS \to Sca \to SbSca \to Sbaca \to abaca;$$

para la gramática

$$\begin{aligned} \mathbf{I} &\to S \\ S &\to SbS \mid ScS \mid a \end{aligned}$$

Ejercicio 3.14. ¿Es posible obtener una gramática regular que sea ambigua? Si se puede, dar un ejemplo. Si no, justifique su respuesta.

Ejercicio 3.15. Una producción regular por la izquierda es una producción de la forma $A \to Bw$ donde A y B son no terminales y w es terminal. Una producción regular por la derecha es una producción de la forma $A \to wB$. Por tanto, las gramáticas regulares por la izquierda y las gramáticas regulares por la derecha contienen solamente producciones regulares por la izquierda y producciones regulares por la derecha, respectivamente. Demuestre que una gramática regular no puede contener ambos tipos de producciones.

Ejercicio 3.16. Determine el tipo de las siguientes gramáticas. Presente todas las caracterizaciones que sean posibles.

1. Gramática G_1:

$$\begin{aligned} \mathbf{I} &\to B \\ B &\to bB \\ B &\to b \\ B &\to aA \\ A &\to aB \\ A &\to bA \\ A &\to a \end{aligned}$$

2. Gramática G_2:

$$\begin{aligned} \mathbf{I} &\to AB \\ AB &\to BA \\ A &\to aA \\ B &\to Bb \\ A &\to a \\ B &\to b \end{aligned}$$

3. Gramática G_3:

$$\begin{aligned} \mathbf{I} &\to A \\ I &\to AAB \\ Aa &\to ABa \\ B &\to BA \\ A &\to aa \\ Bb &\to ABb \\ AB &\to ABB \\ B &\to b \end{aligned}$$

4. Gramática G_4:

$$\begin{aligned} \mathbf{I} &\to BAB \\ I &\to ABA \\ A &\to AB \\ B &\to BA \\ A &\to aA \\ A &\to ab \\ B &\to b \end{aligned}$$

5. Gramática G_5:

$$\begin{aligned} \mathbf{I} &\to C \\ C &\to AAC \\ AA &\to B \\ B &\to bB \\ A &\to a \end{aligned}$$

Ejercicio 3.17. Construya la gramática que genere las palabras no nulas que tengan la propiedad dada:

1. Palabras definidas sobre $\{a, b\}$ que empiecen por a.

2. Palabras definidas sobre $\{a, b\}$ que terminen en ba.

3. Palabras definidas sobre $\{a, b\}$ que contengan ba.

4. Números con punto flotante (como $0{,}294, 89{,}0, 67{,}284$).

5. Números exponenciales (que incluyen a los números con punto flotante y a otros tales como $6{,}9E3, 8E12, 9, 6E-4, 9E-10$).

Ejercicio 3.18. Una palabra α es un palíndromo si es su propio reverso, esto es $\alpha = \alpha^{-1}$. Un lenguaje L es palíndromo si cada una de sus palabras es un palíndromo. Sean las gramáticas:

1. Gramática G_1:

$$\begin{aligned}
\mathbf{I} &\to S \\
S &\to aSa \\
S &\to aSb \\
S &\to bSb \\
S &\to bSa \\
S &\to aa \\
S &\to bb
\end{aligned}$$

2. Gramática G_2:

$$\begin{aligned}
\mathbf{I} &\to S \\
S &\to aS \\
S &\to Sa \\
S &\to bS \\
S &\to Sb \\
S &\to a \\
S &\to b
\end{aligned}$$

a. Describir informalmente $L(\mathsf{G}_1)$ y $L(\mathsf{G}_2)$.

b. ¿Es alguno de ellos un lenguaje palíndromo?

Ejercicio 3.19. Obtener una gramática regular para los siguientes lenguajes:

1. $a^*b \mid a$.
2. $a^*b \mid b^*a$.
3. $(a^*b \mid b^*a)^*$.

Ejercicio 3.20. La gramática dada por

$$\begin{aligned}
\mathbf{I} &\to S \\
S &\to bA \mid aB \mid \varepsilon \\
A &\to abaS \\
B &\to babS
\end{aligned}$$

genera un lenguaje regular. Obtener una expresión regular para este lenguaje.

Ejercicio 3.21. Sean r, s y t expresiones regulares sobre el mismo alfabeto Σ. Demuestre:

1. $r \mid s = s \mid r$.
2. $r \mid \emptyset = r = \emptyset \mid r$.
3. $r \mid r = r$.
4. $r\emptyset = \emptyset r = r$.
5. $(rs)t = r(st)$.
6. $r^* = r^{**} = r^* r^*$.

3.6. Notas bibliográficas

La definición y operaciones sobre las palabras y/o lenguajes son presentadas por [Aho, Sethi y Ullman 1990; Crespo 1983; P. Gómez y C. Gómez 1992; Grimaldi 1997; Kelley 1995]. Las definiciones de semigrupo y monoide fueron tomadas de [López López y Gómez Marín 1993]. La relación entre monoide y lenguaje es presentada por [Kolman y Busby 1984]. Los sistemas formales son presentados por [P. Gómez y C. Gómez 1992; Ladrière 1969]. El ejemplo 3.34 de un sistema combinatorio fue tomado de [P. Gómez y C. Gómez 1992]. Varios textos presentan los elementos relacionados con las gramáticas (definición, derivaciones por la izquierda y por derecha, recursividad, ambigüedad); algunos de ellos son Aho, Sethi y Ullman [1990], Crespo [1983], Johnsonbaugh [1988] y Kelley [1995]. La clasificación de las gramáticas es presentada por [Crespo 1983]. La notación BNF es presentada por [Aho, Sethi y Ullman 1990; Johnsonbaugh 1988]. Las expresiones regulares son presentadas por [Aho, Sethi y Ullman 1990; Crespo 1983; Kelley 1995].

Capítulo 4

Autómatas de estado finito

Siguiendo el contexto de la teoría general de sistemas representamos un sistema S por la figura 4.1.

Figura 4.1: Representación de un sistema (1).

1. Elementos del sistema

   E1. $x(t)$: Vector de entradas al sistema (en un tiempo discreto), es decir,

   $$x(t) = \begin{pmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_n(t) \end{pmatrix}.$$

   E2. $s(t)$: Reacciones exógenas del sistema (en un tiempo discreto), es decir,

   $$s(t) = \begin{pmatrix} s_1(t) \\ s_2(t) \\ \vdots \\ s_m(t) \end{pmatrix}.$$

   E3. Definimos Σ como el espacio de entradas al sistema, es decir,

   $$\Sigma = \{\, x_i(t); i = 1, \ldots, n \,\}.$$

E4. Definimos Γ como el espacio de configuraciones internas (reacciones endógenas frente a $x(t)$).

E5. Definimos Δ como el espacio de reacciones exógenas del sistema, es decir,

$$\Delta = \{\, s_i(t); i = 1, \ldots, m \,\}.$$

2. Comportamiento del sistema

C1. La función endógena δ representa el comportamiento interno del sistema y está definida por:

$$\delta : \Gamma \times \Sigma \to \Gamma.$$

C2. La función exógena λ representa la salida del sistema y está definida por:

$$\lambda : \Gamma \times \Sigma \to \Delta.$$

Entonces nuestro sistema S queda representado por la figura 4.2.

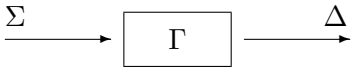


Figura 4.2: Representación de un sistema (2).

4.1. Máquinas de estado finito

Definición 4.1 (Máquina de estado finito)**.** Una máquina de estado finito está definida por la estructura matemática

$$\mathsf{MFE} = \langle \Sigma, \Delta, \Gamma, \delta, \lambda, k_0 \rangle,$$

donde:

(i) Σ es un alfabeto de entrada (finito y diferente de vacío).

(ii) Δ es un alfabeto de salida (finito y diferente de vacío).

(iii) Γ es un conjunto de estados (finito y diferente de vacío).

(iv) δ es una función de estado siguiente, definida por: $\delta : \Gamma \times \Sigma \to \Gamma$.

(v) λ es una función de salida, definida por: $\lambda : \Gamma \times \Sigma \to \Delta$.

(vi) k_0 es el estado inicial ($k_0 \in \Gamma$).

Para concretizar las diferentes representaciones de una máquina de estado finito, utilizaremos un ejemplo clásico de una máquina de estado finito, el cual corresponde a un sumador binario.

Ejemplo 4.2. Construyamos una máquina de estado finito que modele el comportamiento de un sumador binario con dos entradas, esquematizado por la figura 4.3. Sea $\mathsf{MFE} = \langle \Sigma, \Delta, \Gamma, \delta, \lambda, k_0 \rangle$, donde: $\Sigma = \{(00), (01)(10), (11)\}$, $\Delta = \{0, 1\}$, $\Gamma = \{N, A\}$ (N: No acarreo; A: Acarreo) y $k_0 = N$.

Las funciones δ y λ dependen de la representación que se escoja para la máquina de estado finito. Estas representaciones serán descritas a continuación.

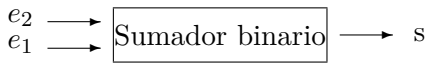


Figura 4.3: Sumador binario.

Definición 4.3 (Diagrama de transición)**.** Una máquina de estado finito se puede representar por medio de una digrafo, llamado digrafo de transición, siguiendo las siguientes convenciones:

1. Nodos: $k_i \in \Gamma$.
2. Arcos: Existe una arco del nodo q_i al nodo q_j, etiquetado con e/s, si y sólo si, $\delta(q_i, e) = q_j$ y $\lambda(q_i, e) = s$.
3. Se coloca un arco no etiquetado para indicar el estado inicial k_0.

Ejemplo 4.4. La figura 4.4 representa el diagrama de transición para el sumador binario.

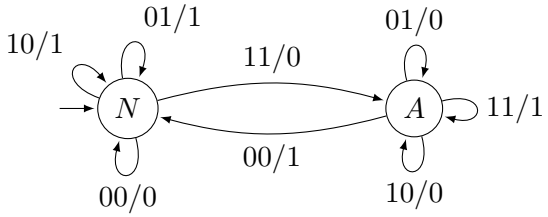


Figura 4.4: Diagrama de transición para un sumador binario (1).

Observación 4.5. Si existen varios arcos del nodo q_i al nodo q_j, para simplificar el diagrama, éstos se pueden representar mediante uno sólo arco con varias etiquetas (tal como lo indica la figura 4.5).

Definición 4.6 (Tabla de transición)**.** Una máquina de estado finito se puede representar por medio de una tabla T de transición, siguiendo las siguientes convenciones:

1. Filas: Estados.

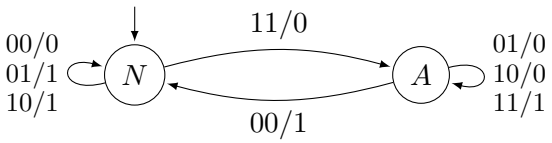


Figura 4.5: Diagrama de transición para un sumador binario (2).

2. Columnas: Símbolos del alfabeto de entrada Σ.
3. $T_{i,j} = (q', s')$, si y sólo si, $(\delta(q_i, e_j) = q' \wedge \lambda(q_i, e_j) = s')$.
4. Se denota el estado inicial k_o, subrayando éste en la fila correspondiente a los estados.

Ejemplo 4.7. La tabla 4.1 representa la tabla de transición para el sumador binario.

Γ/Σ	00	01	10	11
$\underline{N}$	$N, 0$	$N, 1$	$N, 1$	$A, 0$
A	$N, 1$	$A, 0$	$A, 0$	$A, 1$

Tabla 4.1: Tabla de transición para un sumador binario.

Definición 4.8 (Representación explícita)**.** Finalmente, una máquina de estado finito se puede representar listando explícitamente todos sus componentes. Esta forma de representación recibe el nombre de representación explícita.

Ejemplo 4.9. Para el sumador binario tenemos la siguiente representación explícita: $\Sigma = \{(00), (01)(10), (11)\}$, $\Delta = \{0, 1\}$, $\Gamma = \{N, A\}$ (N: No acarreo; A: Acarreo), $k_0 = N$ y

$$\begin{array}{llll} \delta(N, 00) = N, & \delta(N, 01) = N, & \delta(N, 10) = N, & \delta(N, 11) = A, \\ \delta(A, 00) = N, & \delta(A, 01) = A, & \delta(A, 10) = A, & \delta(A, 11) = A, \\ \lambda(N, 00) = 0, & \lambda(N, 01) = 1, & \lambda(N, 10) = 1, & \lambda(N, 11) = 0, \\ \lambda(A, 00) = 1, & \lambda(A, 01) = 0, & \lambda(A, 10) = 0, & \lambda(A, 11) = 1. \end{array}$$

Definición 4.10 (Máquina de Mealy)**.** Una máquina de Mealy está definida por la estructura matemática

$$\mathsf{MME} = \langle \Sigma, \Delta, \Gamma, \delta, \lambda \rangle,$$

donde:

(i) Σ es un alfabeto de entrada (finito y diferente de vacío).

(ii) Δ es un alfabeto de salida (finito y diferente de vacío).

(iii) Γ es un conjunto de estados (finito y diferente de vacío).

(iv) δ es un función de estado siguiente, definida por: $\delta : \Gamma \times \Sigma \to \Gamma$.

(v) λ es una función de salida, definida por: $\lambda : \Gamma \times \Sigma \to \Delta$.

De acuerdo con la definición anterior, una máquina de Mealy es una máquina de estado finito. En ésta no hemos incluido el estado inicial k_0, para simplificar el desarrollo formal.

¿Analizamos qué ocurre si desearamos adicionar la palabra vacía (representada por ε) a nuestro alfabeto de entrada? La idea es utilizar la palabra vacía como el elemento neutro de un monoide, lo cual será desarrollado en la próxima sección.

Veamos primero qué ocurre con la función de estado siguiente δ. Es claro que es necesario ampliar el dominio de la función a: $\delta : \Gamma \times \Sigma \cup \{\varepsilon\} \to \Gamma$. Además, necesitamos utilizar la convención de que $\delta(k, \varepsilon) = k$, para todo $k \in \Gamma$.

Veamos ahora qué sucede con la función de salida λ. Es claro que es necesario ampliar el dominio de la función a: $\lambda : \Gamma \times \Sigma \cup \{\varepsilon\} \to \Delta$.

¿Pero qué sucede con $\lambda(k, \varepsilon)$? Puede ocurrir que por lo menos para algún $k \in \Gamma$ suceda que $\lambda(k, \varepsilon) = e_1 \wedge \lambda(k, \varepsilon) = e_2 \wedge \cdots \wedge \lambda(k, \varepsilon) = e_n$. Esto es factible debido a que pueden existir diferentes (finitas) salidas asociadas con la llegada al estado k. Lo anterior nos imposibilita definir la función λ para el nuevo alfabeto de entrada $\Sigma \cup \{\varepsilon\}$.

Sólo es posible definir la función $\lambda(k, \varepsilon)$ si la máquina de Mealy satisface el siguiente enunciado:

$$\begin{aligned} \forall k, k', k'', e', e'' \quad & (((k, k', k'' \in \Gamma) \wedge (e', e'' \in \Sigma) \wedge (k = \delta(k', e') = \delta(k'', e''))) \Rightarrow \\ & (\lambda(k', e') = \lambda(k'', e''))), \end{aligned}$$

es decir, la salida sólo depende del estado que se alcanza. La satisfacción de este enunciado genera un clase de máquina abstracta llamada máquina de Moore.

Definición 4.11 (Máquina de Moore)**.** Una máquina de Moore está definida por la estructura matemática

$$\mathsf{MMO} = \langle \Sigma, \Delta, \Gamma, \delta, \lambda \rangle,$$

donde:

(i) Σ es un alfabeto de entrada (finito y diferente de vacío).

(ii) Δ es un alfabeto de salida (finito y diferente de vacío).

(iii) Γ es un conjunto de estados (finito y diferente de vacío).

(iv) δ es una función de estado siguiente, definida por: $\delta : \Gamma \times \Sigma \to \Gamma$.

(v) λ es una función de salida, definida por: $\lambda : \Gamma \times \Sigma \to \Delta$.

Además, una máquina de Moore debe satisfacer el siguiente enunciado:

$$\begin{aligned} \forall k, k', k'', e', e'' \quad & (((k, k', k'' \in \Gamma) \wedge (e', e'' \in \Sigma) \wedge (k = \delta(k', e') = \delta(k'', e''))) \Rightarrow \\ & (\lambda(k', e') = \lambda(k'', e''))). \end{aligned}$$

Es importante, entonces, que tengamos en cuenta que en una máquina de Mealy las salidas están asociadas con las transiciones; en cambio, en una máquina de Moore las salidas están asociadas con los estados, esto es, todas las transiciones que están asociadas con un mismo estado tienen la misma salida. Las máquinas de Moore, desde la perspectiva de las máquinas de estado, serán los autómatas de estado finito.

El teorema que presentamos a continuación establece la equivalencia entre las máquinas de Mealy y las máquinas de Moore.

Teorema 4.12.

1. *Toda máquina de Moore es equivalente a una máquina de Mealy.*
2. *Toda máquina de Mealy es equivalente a una máquina de Moore.*

Demostración.

1. Toda máquina de Moore es equivalente a una máquina de Mealy.

   De acuerdo con su definición, una máquina de Moore es una máquina de Mealy que satisface el enunciado especificado en su definición. Es decir, una máquina de Moore es un caso particular de una máquina de Mealy.

2. Toda máquina de Mealy es equivalente a una máquina de Moore.

   Sea $\mathsf{MME} = \langle \Sigma, \Delta, \Gamma, \delta, \lambda \rangle$ una máquina de Mealy. A partir de MME construimos una máquina de Moore $\mathsf{MMO} = \langle \Sigma', \Delta', \Gamma', \delta', \lambda' \rangle$, definida por:

   a) $\Sigma' = \Sigma$.

   b) $\Delta' = \Delta$.

   c) Γ' se obtiene dividiendo cada $k \in \Gamma$ en tantos estados k^s como salidas s se puedan asociar con k es decir:

   $$k^s \in \Gamma' \iff \exists k' \exists e \exists s \quad ((k \in \Gamma) \wedge (e \in \Sigma) \wedge (s \in \Delta) \wedge (\delta(k', e) = k) \wedge (\lambda(k', e) = s)).$$

   d) $\delta'(k^s, e) = \delta(k, e)^{\lambda(k,e)}$.

   e) $\lambda'(k^s, e) = \lambda(k, e)$. □

Ejemplo 4.13. Construir una máquina de Moore equivalente a la máquina de Mealy correspondiente al sumador binario.

La máquina de Mealy, para el sumador binario, está representada por la figura 4.6.

De acuerdo con el teorema 4.12, vamos a construir una máquina de Moore, $\mathsf{MMO} = \langle \Sigma', \Delta', \Gamma', \delta', \lambda' \rangle$, donde: $\Sigma' = \{(00), (01), (10), (11)\}$, $\Delta' = \{0, 1\}$, Γ': El estado N debe ser divido en dos estados N^0 y N^1; y el estado A también debe ser dividido en dos estados A^0 y A^1, con lo que $\Gamma' = \{N^0, N^1, A^0, A^1\}$.

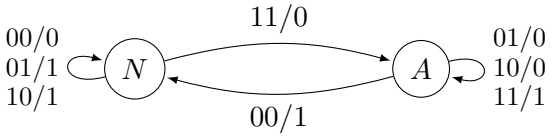


Figura 4.6: Máquina de Mealy para un sumador binario.

La función δ' está definida por:

$$\begin{aligned}
\delta'(N^0,00) &= \delta'(N^1,00) = \delta(N,00)^{\lambda(N,00)} = N^0,\\
\delta'(N^0,01) &= \delta'(N^1,01) = \delta(N,01)^{\lambda(N,01)} = N^1,\\
\delta'(N^0,10) &= \delta'(N^1,10) = \delta(N,10)^{\lambda(N,10)} = N^1,\\
\delta'(N^0,11) &= \delta'(N^1,11) = \delta(N,11)^{\lambda(N,11)} = A^0,\\
\delta'(A^0,00) &= \delta'(A^1,00) = \delta(A,00)^{\lambda(A,00)} = N^1,\\
\delta'(A^0,01) &= \delta'(A^1,01) = \delta(A,01)^{\lambda(A,01)} = A^0,\\
\delta'(A^0,10) &= \delta'(A^1,10) = \delta(A,10)^{\lambda(A,10)} = A^0,\\
\delta'(A^0,11) &= \delta'(A^1,11) = \delta(A,11)^{\lambda(A,11)} = A^1,
\end{aligned}$$

y la función λ' está definida por:

$$\begin{aligned}
\lambda'(N^0,00) &= \lambda'(N^1,00) = \lambda(N,00) = 0,\\
\lambda'(N^0,01) &= \lambda'(N^1,01) = \lambda(N,01) = 1,\\
\lambda'(N^0,10) &= \lambda'(N^1,10) = \lambda(N,10) = 1,\\
\lambda'(N^0,11) &= \lambda'(N^1,11) = \lambda(N,11) = 0,\\
\lambda'(A^0,00) &= \lambda'(A^1,00) = \lambda(A,00) = 1,\\
\lambda'(A^0,01) &= \lambda'(A^1,01) = \lambda(A,01) = 0,\\
\lambda'(A^0,10) &= \lambda'(A^1,10) = \lambda(A,10) = 0,\\
\lambda'(A^0,11) &= \lambda'(A^1,11) = \lambda(A,11) = 1.
\end{aligned}$$

La figura 4.7 representa la máquina de Moore para el sumador binario. Allí, cada estado (nodo) se marca con su nombre y con la salida asociada con él, por medio de una etiqueta de la forma $\text{NombreEstado}^{\text{SalidaAsociada}}$.

4.2. Autómatas de estado finito

En esta sección y en las próximas nos interesaremos en las relaciones existentes entre autómatas, gramáticas y lenguajes. Desde este punto de vista, señalemos de una vez que los

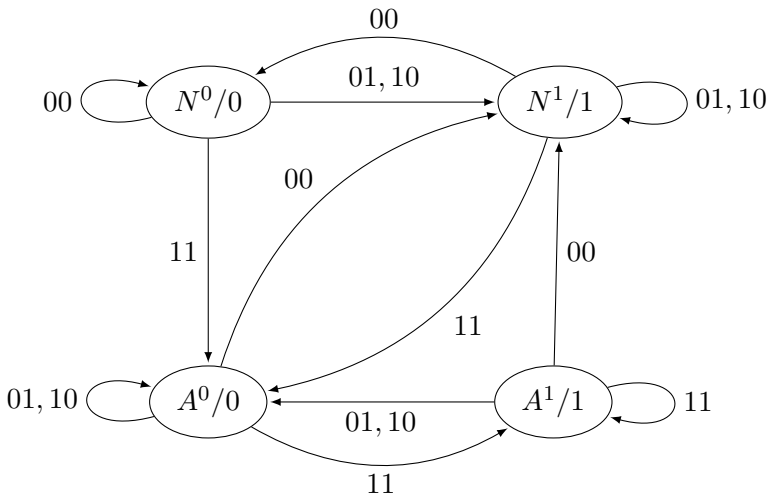


Figura 4.7: Máquina de Moore para un sumador binario.

autómatas desempeñan los siguientes papeles: como aceptadores-reconocedores (figura 4.8) o como generadores-traductores (figura 4.9).

Figura 4.8: Autómata como generador.

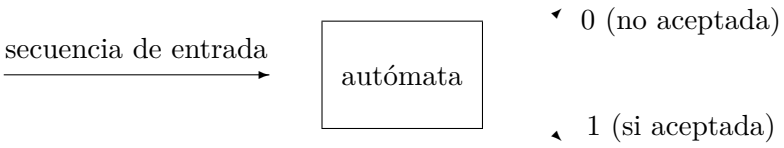


Figura 4.9: Autómata como reconocedor.

En este contexto nos interesa esencialmente el problema del reconocimiento. Este problema está divido en los siguientes problemas:

P-I Problema de síntesis: dado un lenguaje $L(\Sigma)$; ¿qué autómata lo reconoce?

P-II Problema de análisis: dado un autómata A; ¿qué lenguaje lo reconoce?

Nuestra presentación estará entonces dirigida a ofrecer algunos elementos para la solución

al problema del reconocimiento (tanto el problema de síntesis, como el problema de análisis) para el caso de los autómatas de estado finito y los lenguajes regulares (tipo 3).

Recordemos que en una máquina de Moore los estados están univocamente asociados con las salida. Si operamos sobre una máquina de Moore con únicamente dos salidas, podemos pensar sólo en dos tipos de estados: de aceptación y de no aceptación. La noción de un autómata de estado finito nos permite concretizar esta posibilidad.

Definición 4.14 (Autómata de estado finito)**.** Un autómata de estado finito está definido por la estructura matemática

$$\mathsf{AF} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle,$$

donde:

(i) Σ es un alfabeto de entrada (finito y diferente de vacío).

(ii) Γ es un conjunto de estados (finito y diferente de vacío).

(iii) Δ es el conjunto de estados de aceptación ($\Delta \subset \Gamma$).

(iv) δ es un función de estado siguiente, definida por: $\delta : \Gamma \times \Sigma \to \Gamma$.

(v) k_0 es el estado inicial ($k_0 \in \Gamma$).

De manera similar a las máquinas de estado finito, los autómatas de estado finito tienen tres formas de representación.

Definición 4.15 (Diagrama de transición)**.** Un autómata de estado finito se puede representar por medio de una digrafo, llamado digrafo de transición, de acuerdo con las siguientes convenciones:

1. Nodos simples: $k_i \in \Gamma - \Delta$, o estados de no aceptación.
2. Nodos dobles: $k_i \in \Delta$, o estados de aceptación.
3. Arcos: Existe una arco del nodo k_i al nodo k_j, etiquetado con e, si y sólo si, $\delta(k_i, e) = k_j$.
4. Se coloca un arco no etiquetado para indicar el estado inicial k_0.

Ejemplo 4.16. El diagrama de transición de la figura 4.10 representa un autómata de estado finito.

Definición 4.17 (Tabla de transición)**.** Un autómata de estado finito se puede representar por medio de una tabla T de transición, de acuerdo con las siguientes convenciones:

1. Filas: Estados.
2. Para los estados de aceptación se antepone un asterico en la fila correspondiente.

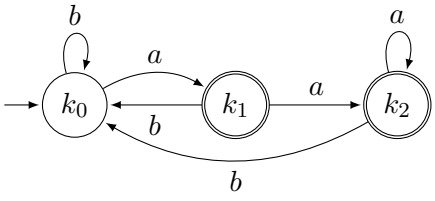


Figura 4.10: Diagrama de transición para un autómata de estado finito.

Γ/Σ	a	b
$\underline{k_0}$	k_1	k_0
$*k_1$	k_2	k_0
$*k_2$	k_2	k_0

Tabla 4.2: Tabla de transición para un autómata de estado finito.

3. Columnas: símbolos del alfabeto de entrada Σ.
4. $T_{i,j} = k'$ sii $\delta(k_i, e_j) = k'$.
5. Se denota el estado inicial k_0, subrayando éste en la fila correspondiente a los estados.

Ejemplo 4.18. El autómata de estado finito representado por la figura 4.10 tiene su tabla de transición representada por la tabla 4.2.

Definición 4.19 (Representación explícita)**.** Similarmene a las máquinas de estado finito, un autómata de estado finito se puede representar listando explícitamente todos sus componentes. Esta forma de representación recibe el nombre de representación explícita del autómata.

Ejemplo 4.20. El autómata de estado finito representado por la figura 4.10, puede representarse explícitamente como sigue, $\mathsf{AF} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle$ donde: $\Sigma = \{a, b\}$, $\Gamma = \{k_0, k_1, k_2\}$, $\Delta = \{k_1, k_2\}$, k_0: Estado inicial y

$$\begin{aligned} \delta(k_0, a) &= k_1, & \delta(k_0, b) &= k_0, \\ \delta(k_1, a) &= k_2, & \delta(k_1, b) &= k_0, \\ \delta(k_2, a) &= k_2, & \delta(k_2, b) &= k_0. \end{aligned}$$

4.3. Reconocedor finito

Un reconocedor finito de un lenguaje L, es un autómata de estado finito que sólo acepta las palabras de dicho lenguaje. Esto es, inicializando el autómata en un cierto estado e introduciendo una palabra de entrada perteneciente a L, finaliza en un estado de aceptación; y al introducir una palabra no perteneciente a L finaliza en un estado de no aceptación.

Definición 4.21 (Reconocedor finito)**.** Un reconocedor finito es un autómata de estado finito $\mathsf{RF} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle$.

Para pensar en el reconocimiento de palabras por parte del reconocedor finito, es necesario expandir la función δ para permitir el procesamiento de las palabras.

Definición 4.22 (Expansión de la función δ)**.** Sea $\mathsf{RF} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle$ un reconocedor finito. Definimos una nueva función δ^* denominada la expansión de la función δ, como sigue:

Sea $\alpha \equiv a_1 a_2 \dots a_n \in \Sigma^*$, entonces, la función $\delta^* : \Gamma \times \Sigma^* \to \Gamma$ está definida por: $\delta(k, \alpha) = \delta(\dots(\delta(\delta(k, a_1), a_2), \dots), a_n)$.

Definición 4.23 (Palabra aceptada por un reconocedor finito)**.** Sea $\mathsf{RF} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle$ un reconocedor finito y sea $\alpha \equiv a_1 a_2 \dots a_n \in \Sigma^*$. Decimos que RF reconoce la palabra α, si y sólo si, existe una secuencia de estados $k_0, k_1, \dots, k_n$ tales que:

1. k_0 es el estado inicial.
2. $\delta(k_{i-1}, a_i) = k_i$, para $0 < i \leq n$.
3. $k_n \in \Delta$.

Es decir, RF reconoce la palabra α si y sólo si $\delta^*(k_0, \alpha) \in \Delta$.

Definición 4.24 (Lenguaje aceptado por un reconocedor finito)**.** Sea RF un reconocedor finito $\mathsf{RF} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle$. El lenguaje aceptado por RF, denotado por, $L(\mathsf{RF})$ está definido por: $L(\mathsf{RF}) = \{ \alpha \in \Sigma^* \mid \delta^*(k_0, \alpha) \in \Delta \}$.

Ejemplo 4.25. El reconocedor finito RF representado por la figura 4.11 reconoce el lenguaje $L(\mathsf{RF}) = \{ab, aab, abb, aaa, \dots, b, abbb, \dots b, \dots\} = \{ a^n b^m; n, m \geq 1 \}$.

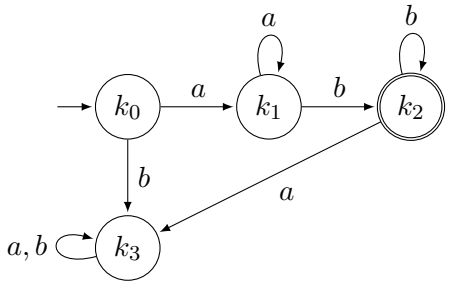


Figura 4.11: Reconocedor finito para $L = \{ a^n b^m; n, m \geq 1 \}$.

Ejemplo 4.26. El reconocedor finito representado por la figura 4.12 reconoce el lenguaje $L(\mathsf{RF}) = \{ 1(01)^n; n \geq 0 \}$.

Para $\alpha \equiv 1011$, tenemos que:

$$\begin{aligned}\delta^*(k_0, \alpha) &= \delta(\delta(\delta(\delta(k_0, 1), 0), 1), 1) \\ &= \delta(\delta(\delta(k_1, 0), 1), 1) \\ &= \delta(\delta(k_2, 1), 1) \\ &= \delta(k_1, 1) \\ &= k_3 \notin \Delta,\end{aligned}$$

entonces $\alpha \notin L(\mathsf{RF})$.

Para $\beta \equiv 101$, tenemos que:

$$\begin{aligned}\delta^*(k_0, \alpha) &= \delta(\delta(\delta(k_0, 1), 0), 1) \\ &= \delta(\delta(k_1, 0), 1) \\ &= \delta(k_2, 1)) \\ &= k_1 \in \Delta,\end{aligned}$$

entonces $\beta \in L(\mathsf{RF})$.

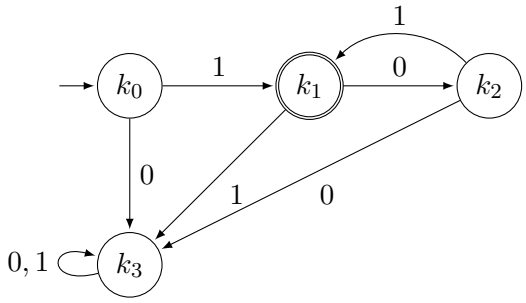


Figura 4.12: Reconocedor finito para $L = \{\, 1(01)^n; n \geq 0 \,\}$.

Hagamos una pausa en nuestro trabajo y formulemos la siguiente pregunta. Dado un lenguaje cualquiera, digamos $L \subseteq \Sigma^*$, ¿siempre es posible encontrar o construir un reconocedor finito para L?; ¿qué significa que tal pregunta hallase una respuesta afirmativa?

Dejamos al lector la segunda pregunta. La respuesta a la primera pregunta (en consonancia con la segunda) no puede ser otra que negativa. Para lograr una satisfacción que corrobore nuestra negación, sabemos que es suficiente hallar un contraejemplo. Esto es, hallar un lenguaje L para el cual se pueda probar la inexistencia de un reconocedor finito RF, tal que RF sea un reconocedor para L.

Ejemplo 4.27. Sea $\Sigma = \{0, 1\}$ un alfabeto y sea $L(\Sigma) = \left\{ 1^{n^2} \mid n \geq 1 \right\}$ un lenguaje. Para probar que tal lenguaje no puede ser reconocido por un reconocedor finito, razonemos por reducción al absurdo.

1. Supongamos que existe un reconocedor finito RF para L, esto es, $L(\mathsf{RF}) = L$ y $\overline{\overline{\Gamma}} = p$ (p es el cardinal del conjunto de estados).

2. Sea $k \in \mathbb{N}$. Podemos garantizar que $\delta^*(q_0, 1^{k^2}) \in \Delta$.

3. Sean, $\alpha_0 = 1^0, \alpha_1 = 1^1, \ldots \alpha_p = 1^p$; $p + 1$ palabras de Σ^*, donde $\overline{\overline{\Gamma}} = p$

4. La sucesión de estados $\delta^*(q_0, \alpha_0), \delta^*(q_0, \alpha_1), \ldots, \delta^*(q_0, \alpha_p)$ debe necesariamente tener alguna repetición (por 4).

5. Para algún i, j; tales que $0 < i, j < p$ y $i \neq j$ se tiene que $\delta^*(q_0, 1^i) = \delta^*(q_0, 1^j)$. Supongamos que $j > i$, entonces $0 < j - i < p$.

6. $\delta^*(q_0, 1^{k^2}) \in \Delta \Longrightarrow \delta^*(q_0, 1^{k^2+(j-i)}) \in \Delta$. Ya que:

$$\begin{aligned}
\delta^*(q_0, 1^{k^2}) &= \delta^*(q_0, 1^i 1^{k^2-i}) \\
&= \delta^*(\delta^*(q_0, 1^i), 1^{k^2-i}) \\
&= \delta^*(\delta^*(q_0, 1^j), 1^{k^2-i}) \\
&= \delta^*(q_0, 1^j 1^{k^2-i}) \\
&= \delta^*(q_0, 1^{k^2+(j-i)}).
\end{aligned}$$

7. Sabemos que, para p existe k tal que $(k+1)^2 - k^2 > p$.

8. Luego, $(k+1)^2 - k^2 > j - i$ (por 6 y 7).

9. Luego, $k^2 < k^2 + j - i < (k+1)^2$. Entonces $k^2 + j - i$ no es cuadrado perfecto.

10. $1^{k^2+(j-i)} \notin L(\Sigma)$ y $\delta^*(q_0, 1^{k^2+(j-i)}) \in \Delta$.

11. Luego, $L(\mathsf{RF}) \neq L(\Sigma)$. Contradicción con (1).

12. Luego, no existe un reconocedor finito para $L(\Sigma)$.

4.4. Algunas clases de autómatas

4.4.1. Autómatas de estado finito deterministas

Presentamos en esta sección dos distinciones entre los autómatas de estado finito.

Definición 4.28 (Autómata de estado finito determinista)**.** Sea $\mathsf{AF} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle$ un autómata de estado finito. El autómata AF es un autómata de estado finito determinista (AFD) si la función de estado siguiente ($\delta : \Gamma \times \Sigma \to \Gamma$) determina un y sólo un estado siguiente.

Ejemplo 4.29. Observemos la figura 4.13. En este caso $\delta(k_0, a) = k_0$ y $\delta(k_0, a) = k_1$, es decir, el comportamiento del estado k_0 para la entrada a, es no determinista. Además, de acuerdo con nuestra definición formal de un autómata de estado finito, δ debe ser una función y por ende no puede tener este comportamiento. Adicionalmente tenemos que $\delta(k_1, a)$ y $\delta(k_2, b)$ no están definidas.

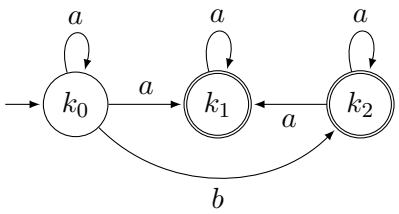


Figura 4.13: Autómata de estado finito no determinista.

Para formalizar el tipo de situaciones presentadas por el ejemplo anterior, es necesario modificar la definición de la función δ, de manera que se obtenga una máquina abstracta, la cual llamaremos autómata de estado finito no determinista

4.4.2. Autómatas de estado finito no deterministas

Definición 4.30 (Autómata de estado finito no determinista)**.** Un autómata de estado finito no determinista (AFND) está definido por la estructura matemática

$$\mathsf{AFND} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle,$$

donde:

(i) Σ es un alfabeto de entrada (finito y diferente de vacío).

(ii) Γ es un conjunto de estados (finito y diferente de vacío).

(iii) Δ es el conjunto de estados de aceptación ($\Delta \subset \Gamma$).

(iv) k_0 es el estado inicial ($k_0 \in \Gamma$).

Es decir; Σ, Γ, Δ y k_0 tienen el mismo significado que en un AFD. Esto es, la diferencia entre un AFD y un AFND está determinida por la función δ.
δ: Función de estado siguiente, definida por: $\delta : \Gamma \times \Sigma \to P(\Gamma)$ (donde $P(\Gamma)$ denota las partes de Γ).

Ejemplo 4.31. De acuerdo con la definición anterior, la figura 4.13 corresponde a un autóma finito no determinista. Tal autómata está dado por $\mathsf{AFND} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle$, donde: $\Sigma = \{a, b\}$, $\Gamma = \{k_0, k_1, k_2\}$, $\Delta = \{k_1, k_2\}$, k_0: estado inicial y

$$\begin{aligned}
\delta(k_0, a) &= \{k_0, k_1\},\\
\delta(k_0, b) &= \{k_2\},\\
\delta(k_1, a) &= \{k_1\},\\
\delta(k_1, b) &= \emptyset,\\
\delta(k_2, a) &= \{k_1, k_2\},\\
\delta(k_2, b) &= \emptyset.
\end{aligned}$$

El siguiente teorema establece la equivalencia entre los autómatas finitos deterministas y no deterministas.

Teorema 4.32. *Para todo AFND existe un AFD equivalente.*

Demostración. Sea $\mathsf{AFND} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle$ un autómata de estado finito no determinista. A partir de AFND es posible construir un autómata de estado finito determinista $\mathsf{AFD} = (\Sigma', \Gamma', \Delta', \delta', k_0')$, definido por:

1. $\Sigma' = \Sigma$.

2. $k_0' = \{k_0\}$.

3. $\Gamma' = P(\Gamma)$. Esto significa que hay tantos estados como subconjuntos tenga el conjunto Γ, es decir, $\overline{\overline{\Gamma'}} = 2^n$, donde $n = \overline{\overline{\Gamma}}$. Observe que Γ' contínua siendo un conjunto finito.

4. δ' es una función definida por:

   $\delta : \Gamma' \times \Sigma \to \Gamma'$, es decir, $\delta' : P(\Gamma) \times \Sigma \to P(\Gamma)$, donde:

$$\delta'(X, e) = \begin{cases} \emptyset, & \text{si } X = \emptyset; \\ \bigcup_{k \in X} \{\delta(k, e)\}, & \text{si } X \neq \emptyset. \end{cases}$$

5. $\Delta' = \{\, X \in P(\Gamma) \mid X \text{ contenga un estado de aceptación perteneciente a } \Delta \,\}$. □

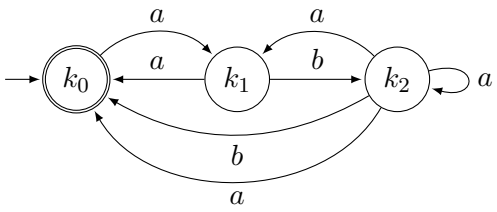


Figura 4.14: Ejemplo AFND.

Ejemplo 4.33. Construir para el AFND representado por la figura 4.14 un AFD equivalente.

De la figura 4.14 obtenemos los siguientes elementos del AFND: $\Sigma = \{a, b\}$; $\Gamma = \{k_0, k_1, k_2\}$; $\Delta = \{k_0\}$; el estado inicial es k_0 y finalmente la función δ se "lee" de la figura. De acuerdo con el teorema 4.32 construimos un $\mathsf{AFD} = (\Sigma', \Gamma', \Delta', \delta', k_0')$ como sigue: $\Sigma' = \Sigma$, $k_0' = k_0$, $\Gamma' = P(\Gamma) = \{\emptyset, \{k_0\}, \{k_1\}, \{k_2\}, \{k_0, k_1\}, \{k_0, k_2\}, \{k_1, k_2\}, \{k_0, k_1, k_2\}\}$. La función δ' está definida por:

$$\begin{aligned}\delta'(\{k_0, k_1, k_2\}, a) &= \delta(k_0, a) \cup \delta(k_1, a) \cup \delta(k_2, a)\\ &= \{k_1\} \cup \{k_0\} \cup \{k_0, k_1, k_2\}\\ &= \{k_0, k_1, k_2\},\end{aligned}$$

por lo tanto:

$$\begin{aligned}\delta'(\{k_0, k_1, k_2\}, b) &= \delta(k_0, b) \cup \delta(k_1, b) \cup \delta(k_2, b) = \emptyset \cup \{k_2\} \cup \{k_0\} = \{k_0, k_2\},\\ \delta'(\{k_0, k_1\}, a) &= \delta(k_0, a) \cup \delta(k_1, a) = \{k_1\} \cup \{k_0\} = \{k_0, k_1\},\\ \delta'(\{k_0, k_1\}, b) &= \delta(k_0, b) \cup \delta(k_1, b) = \emptyset \cup \{k_2\} = \{k_2\},\\ \delta'(\{k_0, k_2\}, a) &= \delta(k_0, a) \cup \delta(k_2, a) = \{k_1\} \cup \{k_0, k_1, k_2\} = \{k_0, k_1, k_2\},\\ \delta'(\{k_0, k_2\}, b) &= \delta(k_0, b) \cup \delta(k_2, b) = \emptyset \cup \{k_0\} = \{k_0\},\\ \delta'(\{k_1, k_2\}, a) &= \delta(k_1, a) \cup \delta(k_2, a) = \{k_0\} \cup \{k_0, k_1, k_2\} = \{k_0, k_1, k_2\},\\ \delta'(\{k_1, k_2\}, b) &= \delta(k_1, b) \cup \delta(k_2, b) = \{k_2\} \cup \{k_0\} = \{k_0, k_2\},\\ \delta'(\{k_0\}, a) &= \delta(k_0, a) = \{k_1\},\\ \delta'(\{k_0\}, b) &= \delta(k_0, b) = \emptyset,\\ \delta'(\{k_1\}, a) &= \delta(k_1, a) = \{k_0\},\\ \delta'(\{k_1\}, b) &= \delta(k_1, b) = \{k_2\},\\ \delta'(\{k_2\}, a) &= \delta(k_2, a) = \{k_0, k_1, k_2\},\\ \delta'(\{k_2\}, b) &= \delta(k_2, b) = \{k_0\},\\ \delta'(\emptyset, a) &= \emptyset,\\ \delta'(\emptyset, b) &= \emptyset.\end{aligned}$$

Además, como el conjunto de estados de aceptación es $\Delta = \{k_0\}$, entonces: $\Delta' = \{\{k_0\}, \{k_0, k_1\}, \{k_0, k_2\}, \{k_0, k_1, k_2\}\}$.

La figura 4.15 representa el AFD que hemos construido. Observe el lector que los estados $\{k_0, k_1\}$ y $\{k_1, k_2\}$ nunca se alcanzan, por lo cual se pueden eliminar del diagrama, obteniendo así la figura 4.16. Además, observe el lector que el estado $\emptyset$ es un estado absorbente, es decir, una vez se llega a él no es posible salir de ahí.

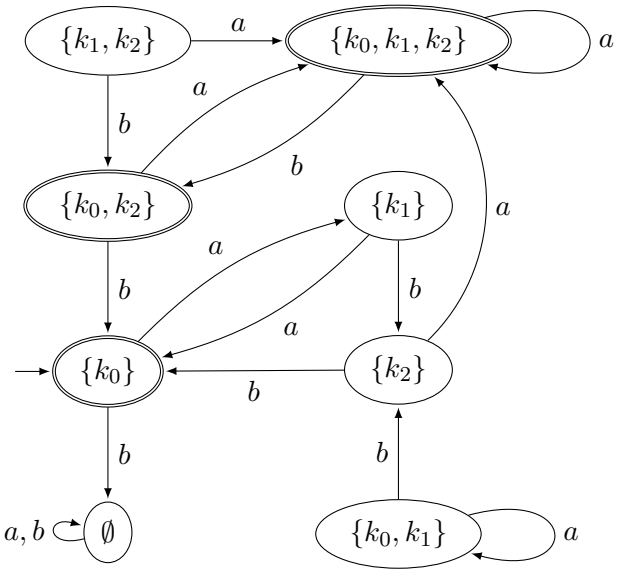


Figura 4.15: Construcción de un AFD a partir de un AFND (1).

4.5. Álgebra y autómatas

4.5.1. Monoides asociados con un autómata

Sea $\mathsf{AF} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle$ un autómata de estado finito. Podemos probar que el autómata AF tiene asociados dos monoides. Veámoslo.

1. Monoide generado por Σ: $\langle \Sigma^*, \cdot, \varepsilon \rangle$ donde:
   Σ^*: Lenguaje universal para el alfabeto Σ.
   $\cdot$: Concatenación de palabras.
   ε: Palabra vacía.

   Demostración. Es necesario demostrar que $\langle \Sigma^*, \cdot, \varepsilon \rangle$ satisface las propiedades de un monoide, es decir, es necesario demostrar que: la operación $\cdot$ es cerrada en Σ^*, es asociativa y por último probar la existencia de un elemento neutro.

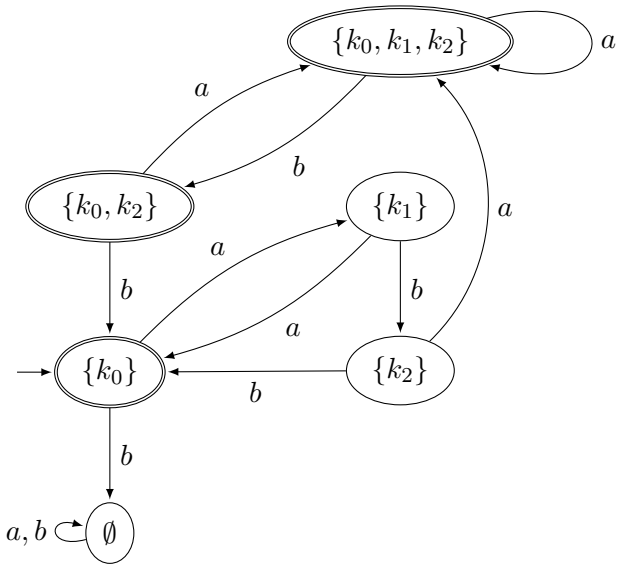


Figura 4.16: Construcción de un AFD a partir de un AFND (2).

a) La concatenación de palabras es una operación cerrada
Sean $\alpha, \beta \in \Sigma^*$ entonces $\alpha \cdot \beta \in \Sigma^*$.

b) Asociatividad
$\forall\alpha\forall\beta\forall\gamma \in \Sigma^*((\alpha \cdot (\beta \cdot \gamma)) = ((\alpha \cdot \beta) \cdot \gamma) = \alpha\beta\gamma)$.

c) Existencia elemento neutro
$\forall\alpha \in \Sigma^*(\alpha \cdot \varepsilon = \varepsilon \cdot \alpha = \alpha)$. □

2. Monoide de transformaciones de Γ: $\langle \Gamma^\Gamma, \circ, f_\varepsilon \rangle$ donde:

Γ^Γ: Conjunto de funciones de Γ en Γ, definidas para cada uno de los símbolos pertenecientes a Σ, es decir, $\Gamma^\Gamma = \{ f_a : \Gamma \to \Gamma \mid a \in \Sigma \}$.
$\circ$: Composición de funciones.
f_ε: Función de la palabra vacía. $\forall k \in \Gamma(f_\varepsilon(k) = k)$.

Demostración. Es necesario demostrar que $\langle \Gamma^\Gamma, \circ, f_\varepsilon \rangle$ satisfece las propiedades de un monoide, es decir, es necesario demostrar que la operación $\circ$, es cerrada en Γ^Γ, es asociativa y que existe un elemento neutro.

a) La composición de funciones es una operación cerrada
Sean $f_a, f_b \in \Gamma^\Gamma$ entonces $f_a \circ f_b \in \Gamma^\Gamma$.

b) Asociatividad
$\forall k \in \Gamma, \forall a, b, c \in S((f_a \circ (f_b \circ f_c)(k)) = ((f_a \circ f_b) \circ f_c)(k))$.

c) Existencia de elemento neutro
$\forall a \in \Sigma(f_a \circ f_\varepsilon = f_\varepsilon \circ f_a = f_a)$. □

Una vez definidos los monoides para el autómata AF, es necesario extender las funciones f_a, de manera que operen sobre cadenas de palabras y no únicamente sobre símbolos del alfabeto Σ. La función f_a está definida por: $f_a : \Gamma \to \Gamma \mid a \in \Sigma$. Ahora necesitamos una función f_α definida por: $f_\alpha : \Gamma \to \Gamma \mid \alpha \in \Sigma^*$. Sea $\alpha \equiv a_1 a_2 \dots a_n$, donde $a_i \in \Sigma$, entonces:

$$f_\alpha(k) = (f_{a_n} \circ f_{a_{n-1}} \circ \cdots \circ f_{a_1})(k) = f_{a_n}(f_{a_{n-1}}(\dots(f_{a_1}(k))\dots)).$$

Ejemplo 4.34. Para el autómata representado por la figura 4.17 tenemos que:
$\Gamma^\Gamma = \{ f_a : \Gamma \to \Gamma \mid a \in \Sigma \}$.
$f_a : \Gamma \to \Gamma$, donde $f_a(k_1) = k_1$ y $f_a(k_2) = k_1$.
$f_b : \Gamma \to \Gamma$, donde $f_b(k_1) = k_2$ y $f_b(k_2) = k_2$.
Luego, $\Gamma^\Gamma = \{f_a, f_b, f_\varepsilon\}$.
Sea $\alpha \equiv abb$, entonces:

$$\begin{aligned} f_\alpha(k_1) &= (f_b \circ f_b \circ f_a)(k_1) \\ &= f_b(f_b(f_a(k_1))) \\ &= f_b(f_b(k_1)) \\ &= f_b(k_2) \\ &= k_2; \end{aligned}$$

$$\begin{aligned} f_\alpha(k_2) &= (f_b \circ f_b \circ f_a)(k_2) \\ &= f_b(f_b(f_a(k_2))) \\ &= f_b(f_b(k_1)) \\ &= f_b(k_2) \\ &= k_2 \end{aligned}$$

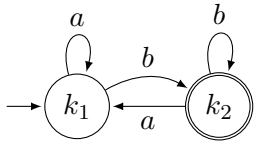


Figura 4.17: Expansión de la función $f_a, a \in \Sigma$ en $f_\alpha, \alpha \in \Sigma^*$.

El propósito que buscamos es poder expresar el comportamiento del autómata AF por medio de una función $t : \Sigma^* \to \Gamma^\Gamma$, donde $t(\alpha) = f_\alpha$. El objetivo es asignar a cada cadena de entrada de Σ^* una función de Γ en Γ.

4.5.2. Comportamiento entrada-estados de un autómata

Definición 4.35 (Homomorfismo monoides)**.** Sean $\langle A, *, e\rangle$ y $\langle A', *', e'\rangle$ dos monoides y sea $f : A \to A'$ una función. Se dice que f es un homomorfismo entre $\langle A, *, e\rangle$ y $\langle A', *', e'\rangle$ si f preserva el símbolo de función y f preserva el símbolo de constante, es decir, $\forall a \forall b \in A(f(a * b) = (f(a) *' f(b)))$, además $f(e) = e'$.

El siguiente teorema presenta un homomorfismo entre los monoides asociados con un autómata.

Teorema 4.36. *Si* $\mathsf{AF} = \langle \Sigma, \Gamma, \Delta, \delta, k_0\rangle$ *es un autómata de estado finito,* $\langle \Sigma^*, \cdot, \varepsilon\rangle$ *es el monoide generado por* Σ *y* $\langle \Gamma^\Gamma, \circ, f_\varepsilon\rangle$ *es el monoide de transformaciones de* Γ*, entonces la función* $t : \Sigma^* \to \Gamma^\Gamma$*, tal que* $t(\alpha) = f_\alpha$ *es un homomorfismo entre los monoides* $\langle \Sigma^*, \cdot, \varepsilon\rangle$ *y* $\langle \Gamma^\Gamma, \circ, f_\varepsilon\rangle$*.*

La figura 4.18 representa el homomorfismo entre los dos monoides.

$$\begin{array}{ccc} \langle \Sigma^*, \cdot, \epsilon\rangle & \xrightarrow{\ t\ } & \langle \Gamma^\Gamma, \circ, f_\epsilon\rangle \\ \cdot \downarrow & & \downarrow \circ \\ \langle \Sigma^*, \cdot, \epsilon\rangle & \xrightarrow[\ t\ ]{} & \langle \Gamma^\Gamma, \circ, f_\epsilon\rangle \end{array}$$

Figura 4.18: Homomorfismo entre los monoides $\langle \Sigma^*, \cdot, \epsilon\rangle$ y $\langle \Gamma^\Gamma, \circ, f_\epsilon\rangle$.

Demostración. Ejercicio 4.19. □

Definición 4.37 (Comportamiento entrada-estados)**.** Sea un autómata de estado finito $\mathsf{AF} = \langle \Sigma, \Gamma, \Delta, \delta, k_0\rangle$, sea $\langle \Sigma^*, \cdot, \varepsilon\rangle$ el monoide generado por Σ, sea $\langle \Gamma^\Gamma, \circ, f_\varepsilon\rangle$ el monoide de transformaciones de Γ y sea $t : \Sigma^* \to \Gamma^\Gamma$ (donde $t(\alpha) = f_\alpha$), el homomorfismo entre los monoides $\langle \Sigma^*, \cdot, \varepsilon\rangle$ y $\langle \Gamma^\Gamma, \circ, f_\varepsilon\rangle$. La función t es denominada el comportamiento de entrada-estados del autómata AF.

4.5.3. Relación de equirrespuesta de un autómata

Definición 4.38 (Relación de equirrespuesta)**.** Sea un autómata de estado finito $\mathsf{AF} = \langle \Sigma, \Gamma, \Delta, \delta, k_0\rangle$ y sea $t : \Sigma^* \to \Gamma^\Gamma$ el comportamiento de entrada-estados del autómata AF.

La relación de equirrespuesta de AF, representada por $\approx_A$, está definida por:

$$\begin{aligned}\alpha \approx_A \beta &\stackrel{\text{def}}{=} t(\alpha) = t(\beta)\\ &\stackrel{\text{def}}{=} f_\alpha = f_\beta\\ &\stackrel{\text{def}}{=} \forall k \in \Gamma(\delta^*(k,\alpha) = \delta^*(k,\beta) \text{ para } \alpha,\beta \in \Sigma^*.\end{aligned}$$

Para efectos de simplificación en la notación, '$\delta^*(k,\alpha)$' se representará, de ahora en adelante, por '$\delta(k,\alpha)$'.

Teorema 4.39. *La relación de equirrespuesta $\approx_A$ es una relación de equivalencia.*

Demostración. Es necesario demostrar que $\approx_A$ es una relación reflexiva, simétrica y transitiva.

1. Reflexiva
$$\forall k \in \Gamma(\delta(k,\alpha) = \delta(k,\alpha)) \Rightarrow \alpha \approx_A \alpha.$$

2. Simétrica
$$\begin{aligned}\alpha \approx_A \beta &\Rightarrow \forall k \in \Gamma(\delta(k,\alpha) = \delta(k,\beta))\\ &\Rightarrow \forall k \in \Gamma(\delta(k,\beta) = \delta(k,\alpha))\\ &\Rightarrow \beta \approx_A \alpha.\end{aligned}$$

3. Transitiva
$$\begin{aligned}\alpha \approx_A \beta \wedge \beta \approx_A \gamma &\Rightarrow \forall k \in \Gamma(\delta(k,\alpha) = \delta(k,\beta)) \wedge \forall k \in \Gamma(\delta(k,\beta) = \delta(k,\gamma))\\ &\Rightarrow \forall k \in \Gamma(\delta(k,\alpha) = \delta(k,\gamma))\\ &\Rightarrow \alpha \approx_A \gamma.\end{aligned}$$
□

Del teorema anterior podemos garantizar que la relación de equirrespuesta $\approx_A$ induce una partición sobre el conjunto Σ^* (en clases de equivalencia) definida por: $\Sigma^*/_{\approx_A} = \{\, [\alpha] \mid \alpha \in \Sigma^* \,\}$ donde, $[a] = \{\, \beta \in \Sigma^* \mid \alpha \approx_A \beta \,\}$.

Definición 4.40 (Partición de índice finito)**.** Una partición P tiene índice finito si el cardinal de P, denotado por $\overline{\overline{P}}$, es finito.

Teorema 4.41. *La partición $\Sigma^*/_{\approx_A}$ tiene índice finito.*

Demostración. Para el homomorfismo de comportamiento de entrada-estados $t : \Sigma^* \to \Gamma^\Gamma$, donde $t(\alpha) = f_\alpha$; se tiene que $\overline{\overline{\Gamma^\Gamma}} \leq n^n$, donde n es el número de estados del autómata de estado finito; luego $\overline{\overline{\Gamma^\Gamma}}$ tiene cardinal finito. Si el homomorfismo $t : \Sigma^* \to \Gamma^\Gamma$ es sobreyectivo, entonces $\overline{\overline{\Sigma^*/_{\approx_A}}} = \overline{\overline{\Gamma^\Gamma}}$; de lo contrario $\overline{\overline{\Sigma^*/_{\approx_A}}} \leq \overline{\overline{\Gamma^\Gamma}}$. En cualquier caso, $\Sigma^*/_{\approx_A}$ tiene cardinal finito, luego es de índice finito. □

4.5.4. Relaciones de congruencia

Definición 4.42 (Relación de congruencia derecha)**.** Sea $\langle A, *, e\rangle$ un monoide y R una relación de equivalencia definida en A. R es una relación de congruencia derecha en $\langle A, *, e\rangle$, si y sólo si, $\forall a \forall b \forall c \in A \quad ((aRb) \Rightarrow ((a * c)R(b * c)))$.

Definición 4.43 (Relación de congruencia izquierda)**.** Sea $\langle A, *, e\rangle$ un monoide y R una relación de equivalencia definida en A. R es una relación de congruencia izquierda en $\langle A, *, e\rangle$, si y sólo si, $\forall a \forall b \forall c \in A \quad ((aRb) \Rightarrow ((c * a)R(c * b)))$.

Definición 4.44 (Relación de congruencia)**.** Sea $\langle A, *, e\rangle$ un monoide y R una relación de equivalencia definida en A. R es una relación de congruencia en $\langle A, *, e\rangle$, si y sólo si, $\forall a \forall b \forall c \in A \quad ((aRb) \Rightarrow ((a * c)R(b * c) \wedge (c * a)R(c * b)))$.

Teorema 4.45. *La relación de equirrespuesta $\approx_A$ definida en Σ^* (Σ^* es el dominio del monoide $\langle \Sigma^*, \cdot, \varepsilon\rangle$), es una relación de congruencia.*

Demostración. Es necesario demostrar que $\approx_A$ es una relación de congruencia derecha e izquierda sobre $\langle \Sigma^*, \cdot, \varepsilon\rangle$.

1. $\approx_A$ es una relación de congruencia derecha sobre $\langle \Sigma^*, \cdot, \varepsilon\rangle$

$$\begin{aligned} \alpha \approx_A \beta &\Leftrightarrow t(\alpha) = t(\beta) \\ &\Leftrightarrow t(\alpha\eta) = t(\alpha) \circ t(\eta) = t(\beta) \circ t(\eta) = t(\beta\eta) \\ &\Leftrightarrow \alpha\eta \approx_A \beta\eta, \quad \text{para toda} \quad \eta \in \Sigma^*. \end{aligned}$$

2. $\approx_A$ es una relación de izquierda sobre $\langle \Sigma^*, \cdot, \varepsilon\rangle$

   Se demuestra de forma similar a la anterior.

Luego, $\approx_A$ es una relación de congruencia sobre $\langle \Sigma^*, \cdot, \varepsilon\rangle$. □

4.5.5. Relación equirrespuesta de un reconocedor finito

Definición 4.46 (Relación equirrespuesta de un reconocedor finito)**.** Sea un reconocedor finito (autómata de estado finito) $\mathsf{RF} = \langle \Sigma, \Gamma, \Delta, \delta, k_0\rangle$. La relación de equirrespuesta de RF, representada por $\approx_R$, está definida por:

$$\alpha \approx_R \beta \stackrel{\text{def}}{=} \delta(k_0, \alpha) = \delta(k_0, \beta) \text{ para todo } \alpha, \beta \in \Sigma^*.$$

Teorema 4.47. *La relación de equirrespuesta $\approx_R$ es una relación de equivalencia.*

Demostración. La demostración se deja como ejercicio. □

Del teorema anterior podemos garantizar que la relación de equirrespuesta $\approx_R$ induce una partición sobre el conjunto Σ^* en clases de equivalencia definida por: $\Sigma^*/_{\approx_R} = \{\, [\alpha] \mid \alpha \in \Sigma^* \,\}$, donde, $[a] = \{\, \beta \in \Sigma^* \mid \alpha \approx_R \beta \,\}$.

Teorema 4.48. *La relación de equirrespuesta $\approx_A$ de un autómata de estado finito implica la relación de equirrespuesta $\approx_R$ de un reconocedor finito.*

Demostración.

$$\begin{aligned}\alpha \approx_A \beta &\Rightarrow \forall k \in \Gamma(\delta(k,\alpha) = \delta(k,\beta)) \\ &\Rightarrow \delta(k_0,\alpha) = \delta(k_0,\beta) \\ &\Rightarrow \alpha \approx_R \beta.\end{aligned}$$

□

Teorema 4.49. *La relación de equirrespuesta $\approx_R$ de un reconocedor finito no implica la relación de equirrespuesta $\approx_A$ de un autómata de estado finito.*

Demostración.

$$\begin{aligned}\alpha \approx_R \beta &\Rightarrow \delta(k_0,\alpha) = \delta(k_0,\beta) \\ &\not\Rightarrow \forall k \in \Gamma(\delta(k,\alpha) = \delta(k,\beta)) \\ &\not\Rightarrow \alpha \approx_A \beta.\end{aligned}$$

□

Definición 4.50 (Refinamiento de una partición)**.** Se dice que una partición P refina una partición Q, si y sólo si, $\overline{\overline{P}} \geq \overline{\overline{Q}}$.

Teorema 4.51. *La partición $\Sigma^*/_{\approx_A}$ refina la partición $\Sigma^*/_{\approx_R}$.*

Demostración.

1. Como $\approx_A \Rightarrow \approx_R$ (por el teorema 4.48, entonces se observa la posible igualdad entre $\overline{\overline{\Sigma^*/_{\approx_A}}}$ y $\overline{\overline{\Sigma^*/_{\approx_R}}}$.
2. Como $\approx_R \not\Rightarrow \approx_A$ (por el teorema 4.49, entonces se observa la posibilidad de que $\overline{\overline{\Sigma^*/_{\approx_A}}} > \overline{\overline{\Sigma^*/_{\approx_R}}}$.
3. Entonces $\overline{\overline{\Sigma^*/_{\approx_A}}} \geq \overline{\overline{\Sigma^*/_{\approx_R}}}$, luego $\Sigma^*/_{\approx_A}$ refina a $\Sigma^*/_{\approx_R}$. □

Teorema 4.52. *La partición $\Sigma^*/_{\approx_R}$ tiene índice finito.*

Demostración. Como $\Sigma^*/_{\approx_A}$ tiene índice finito (teorema 4.41) y $\Sigma^*/_{\approx_A}$ refina a $\Sigma^*/_{\approx_R}$ (teorema 4.51), entonces $\Sigma^*/_{\approx_A}$ tiene índice finito. □

Teorema 4.53. *La relación de equirrespuesta $\approx_R$ definida en Σ^* (Σ^* que es el dominio del monoide $\langle \Sigma^*, \cdot, \varepsilon \rangle$), es una relación de congruencia derecha, es decir, $\forall \alpha, \beta, \eta \in \Sigma^*$ $((\alpha \approx_R \beta) \Rightarrow ((\alpha\eta) \approx_R (\beta\eta)))$.*

Demostración.

$$\begin{aligned}\alpha \approx_A \beta &\Rightarrow (\delta(k_0,\alpha) = \delta(k_0,\beta)) \\ &\Rightarrow (\delta(k_0,\alpha\eta) = \delta(\delta(k_0,\alpha),\eta) = \delta(\delta(k_0,\beta),\eta) = \delta(k_0,\beta\eta)) \\ &\Rightarrow \alpha\eta \approx_A \beta\eta \quad \text{para toda} \quad \eta \in \Sigma^*.\end{aligned}$$

□

4.6. Álgebra y lenguajes

4.6.1. Relación de congruencia derecha inducida por un lenguaje

Definición 4.54 (Relación inducida por un lenguaje)**.** Sea Σ un alfabeto y $L \subseteq \Sigma^*$ un lenguaje. La relación inducida por L, representada por $\approx_L$, está definida de Σ^* en Σ^* por: sean $\alpha, \beta, \delta \in \Sigma^*$, entonces:

$$\alpha \approx_L \beta \stackrel{\text{def}}{=} (\alpha\delta \in L \leftrightarrow \beta\delta \in L).$$

Teorema 4.55. *La relación inducida por L, $(\approx_L)$ es una relación de equivalencia.*

Demostración.

1. Reflexiva
$$(\alpha\delta \in L \leftrightarrow \alpha\delta \in L) \Rightarrow \alpha \approx_L \alpha.$$

2. Simétrica
$$\begin{aligned} \alpha \approx_L \beta &\Rightarrow (\alpha\delta \in L \leftrightarrow \beta\delta \in L) \\ &\Rightarrow \beta \approx_L \alpha. \end{aligned}$$

3. Transitiva
$$\begin{aligned} \alpha \approx_L \beta \wedge \beta \approx_L \gamma &\Rightarrow (\alpha\delta \in L \leftrightarrow \beta\delta \in L) \wedge (\beta\delta \in L \leftrightarrow \gamma\delta \in L) \\ &\Rightarrow (\alpha\delta \in L \leftrightarrow \gamma\delta \in L) \\ &\Rightarrow \alpha \approx_L \gamma. \end{aligned}$$

□

El teorema anterior nos garantiza que la relación $\approx_L$ induce una partición sobre el conjunto Σ^* (en clases de equivalencia) definida por: $\Sigma^*/_{\approx_L} = \{\, [\alpha] \mid \alpha \in \Sigma^* \,\}$; donde, $[a] = \{\, \beta \in \Sigma^* \mid \alpha \approx_L \beta \,\}$.

Teorema 4.56. *La relación inducida por L, $(\approx_L)$ es una relación de congruencia derecha, es decir: $\forall \alpha, \beta, \delta \in \Sigma^* \quad ((\alpha \approx_L \beta) \Rightarrow (\alpha\delta \approx_L \beta\delta))$.*

Demostración. Sea $\alpha, \beta, \delta \in \Sigma^*$; $\delta \equiv \eta\theta$

$$\begin{aligned} \alpha \approx_L \beta &\Rightarrow (\alpha\delta \in L \leftrightarrow \beta\delta \in L) \\ &\Rightarrow (\alpha\eta\theta \in L \leftrightarrow \beta\eta\theta \in L) \\ &\Rightarrow \alpha\eta \approx_L \beta\eta. \end{aligned}$$

□

4.6.2. Condición para que un lenguaje sea aceptado por un reconocedor finito

Teorema 4.57. *Sea Σ un alfabeto y $L \subseteq \Sigma^*$ un lenguaje. L es un lenguaje aceptado por un reconocedor finito, si y sólo si la relación de congruencia derecha inducida por L tiene índice finito.*

Demostración. (primera parte)

Si L es un lenguaje aceptado por un reconocedor finito (hipótesis), entonces la relación de congruencia derecha inducida por L tiene índice finito (tesis).

Hipótesis: Sean $\mathsf{RF} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle$ un reconocedor finito, $L \subseteq \Sigma^*$ un lenguaje, $L(\mathsf{RF})$ es el conjunto de cadenas aceptadas por RF y $L = L(\mathsf{RF})$.

Tesis: Sea $\alpha, \beta \in \Sigma^*$; $\alpha \approx_R \beta \Rightarrow \alpha \approx_L \beta$.

Entonces,

$$\begin{aligned}
\alpha \approx_R \beta &\Leftrightarrow \delta(k_o, \alpha) = \delta(k_o, \beta) \quad \text{(definición de } \approx_R) \\
&\Rightarrow \delta(k_o, \alpha\eta) = \delta(k_o, \beta\eta) \\
&\Rightarrow \delta(k_o, \alpha\eta) \in \Delta \leftrightarrow \delta(k_o, \beta\eta) \in \Delta \\
&\Rightarrow \alpha\eta \in L(\mathsf{RF}) \leftrightarrow \beta\eta \in L(\mathsf{RF}) \\
&\Rightarrow \alpha\eta \in L \leftrightarrow \beta\eta \in L \\
&\Rightarrow \alpha \approx_L \beta.
\end{aligned}$$

Luego, $\Sigma^*/_{\approx_R}$ refina a $\Sigma^*/_{\approx_L}$ y, como $\Sigma^*/_{\approx_R}$ tiene índice finito (teorema 4.52), entonces $\Sigma^*/_{\approx_L}$ tiene índice finito. □

Demostración. (segunda parte)

Si la relación de congruencia derecha inducida por L tiene índice finito (hipótesis), entonces L es un lenguaje aceptado por un reconocedor finito (tesis).

Idea: Construir un reconocedor finito que acepta L.

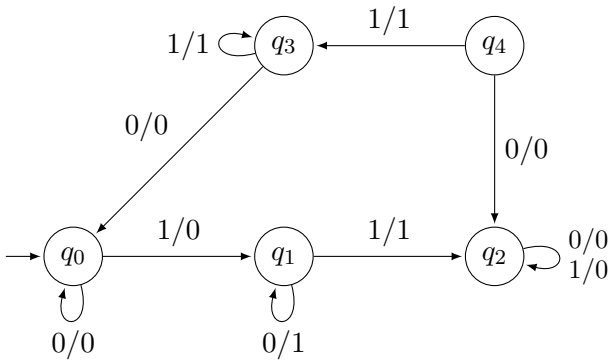
Sea $\mathsf{RF} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle$ un reconocedor finito donde:
Σ: Alfabeto del lenguaje L.
Γ: $\Sigma^*/_{\approx_L}$ (finito porque $\Sigma^*/_{\approx_L}$ tiene índice finito por hipótesis). Los elementos de Γ son clases de equivalencia $[a] = \{ \beta \in \Sigma^* \mid \alpha \approx_L \beta \}$.
$\delta : \Gamma \times \Sigma \to \Gamma$, donde $\delta([\alpha], a) = [\alpha a]$ para toda $a \in \Sigma$.
k_o: $[\varepsilon]$ (ε es la palabra vacía).
Δ: $\{ [\beta] \mid \beta \in L \}$.

Veamos que $L = L(\mathsf{RF})$.

$$\begin{aligned}\alpha \in L(\mathsf{RF}) &\Leftrightarrow \delta(k_o, \alpha) \in \Delta \\ &\Leftrightarrow \delta([\varepsilon], \alpha) \in \Delta \\ &\Leftrightarrow [\varepsilon\alpha] \in \Delta \\ &\Leftrightarrow [\alpha] \in \Delta \\ &\Leftrightarrow \alpha \in L \quad \text{(por definición de } \Delta\text{).}\end{aligned}$$ □

4.7. Ejercicios

Ejercicio 4.1. Para la máquina de estado finito representada por la figura:



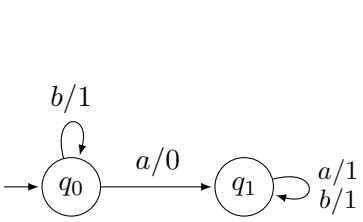
1. Determine la palabra de salida para la entrada 110111, comenzando en q_0. ¿Cuál es el último estado de transición?
2. Determine la palabra de salida para la entrada 110111, comenzando en q_1. ¿Qué sucede cuando q_2 y q_3 son los estados iniciales?
3. Encuentre la tabla de transición para esta máquina.
4. ¿Desde qué estado se debe comenzar para que la palabra de entrada 10010 produzca la salida 10000?

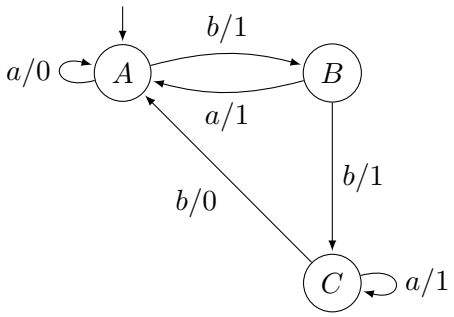
Ejercicio 4.2. Dibuje el diagrama de transición correspondiente a las máquinas de estado finito, indicadas por las siguientes tablas de transición:

Γ/Σ	a	b
$\underline{q_0}$	$q_1, 1$	$q_1, 1$
q_1	$q_0, 0$	$q_1, 1$

Γ/Σ	a	b	c
$\underline{q_0}$	$q_1, 1$	$q_0, 1$	$q_2, 2$
q_1	$q_0, 2$	$q_2, 0$	$q_2, 0$
q_2	$q_3, 1$	$q_3, 0$	$q_0, 1$
q_3	$q_1, 2$	$q_1, 0$	$q_0, 2$

Ejercicio 4.3. Para cada una de las máquinas de estado finito indicadas por el diagrama de transición, construya su representación explícita.





Ejercicio 4.4. Tal como está representado el sumador binario por el diagrama de la figura 4.5, es necesario sumar $01 + 01$ si se desea realizar la suma de $1 + 1$. 'Analice por qué.

Ejercicio 4.5. Un modelo simplificado del comportamiento de un estudiante puede ser descrito por una máquina de estado finito. Sea:
$\Sigma = \{\text{tarea, fiesta, examen}\}$.
$\Delta = \{\text{cantar, maldecir, dormir}\}$.
$\Gamma = \{\text{feliz, enojado, deprimido}\}$.
$k_0 = \text{feliz}$.
Construya una máquina de estado finito que modele un posible comportamiento del estudiante.

Ejercicio 4.6. Diseñe una máquina de estado finito que modele el comportamiento de una máquina expendedora de Coca-Cola y Malta. La máquina acepta monedas de 5, 10 y 25 pesos y dispone de dos botones; C para Coca-Cola y M para Malta. Cada producto cuesta 20 pesos y se espera por supuesto que la máquina entrege el producto solicitado y la devuelta si ésta existe.

Ejercicio 4.7. Un procedimiento sencillo y muy utilizado para detectar errores en una transmisión digital consiste en enviar un bit de paridad. Este bit puede ser tal, que haga par el número total de "unos" enviados, o haga par el número total de "ceros" enviados (en este caso se habla de paridad par), o también puede ser que este bit haga impar el número total de "unos" o el número total de "ceros" enviados.

Para el caso de paridad par de "unos", el generador de paridad actúa de la siguiente forma (supongamos la longitud del mensaje de 3 bits): si el mensaje a enviar tiene un número impar de "unos" (por ejemplo "100"), el generador de paridad adiciona un "uno" ("1100") y envía el mensaje; si por el contrario el mensaje que se enviará tiene un número par de "unos" (por ejemplo "101") el generador de paridad adiciona un "cero" ("0101") y envía el mensaje.

El detector de paridad (para el caso de paridad par de "unos") trabaja de la siguiente forma: si el mensaje recibido tiene un número par de "unos", la transmisión del mensaje NO tuvo errores; pero si el mensaje recibido tiene un número impar de "unos", entonces la transmisión del mensaje SÍ tuvo errores y debe ser retransmitido.

Construya una máquina de estado finito que se comporte como un detector de paridad par de "unos". Represente la máquina por medio de su diagrama de transiciones.

Ejercicio 4.8. Obtener la tabla de transiciones y el diagrama de transiciones de la máquina de Moore equivalente a la máquina de Mealy, descrita por la siguiente tabla:

Γ/Σ	e_1	e_2
q_0	q_3, s_1	q_2, s_2
q_1	q_4, s_2	q_3, s_2
q_3	q_4, s_1	q_2, s_2
q_4	q_2, s_2	q_4, s_1

Ejercicio 4.9. Definir las máquinas de Mealy y de Moore para un restador binario.

Ejercicio 4.10. Construya un autómata de estado finito para solucionar el problema expuesto en la siguiente carta:

Querido amigo:

Al poco tiempo de comprar esta vieja mansión tuve la desagradable sorpresa de comprobar que está hechizada con dos sonidos de ultratumba que la hacen prácticamente inhabitable: un canto picaresco y una risa sardónica. Aún conservo, sin embargo, cierta esperanza, pues la experiencia me ha demostrado que su comportamiento obedece a ciertas leyes, oscuras pero infalibles, y que puede modificarse tocando el órgano y quemando incienso.

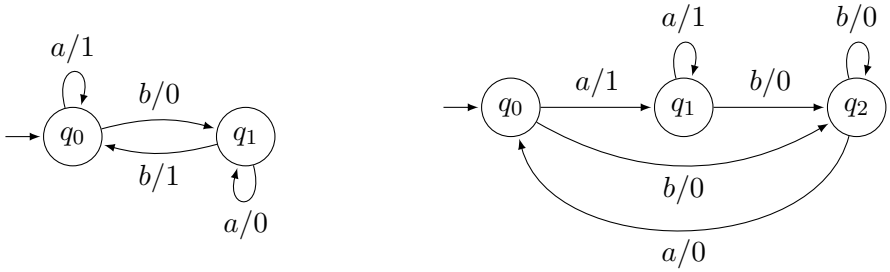
En cada minuto, cada sonido está presente o ausente. Lo que cada uno de ellos hará en el minuto siguiente depende de que lo pasa en el minuto actual, de la siguiente manera:

El canto conservará el mismo estado (presente o ausente), salvo si durante el minuto actual no se oye la risa y toco el órgano, en cuyo caso el canto toma el estado opuesto.

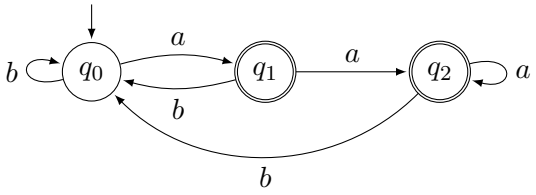
En cuanto a la risa, si no quemo incienso, se oirá o no según que el canto esté presente o ausente (de modo que la risa imita al canto con un minuto de retardo). Ahora bien, si quemo incienso la risa hará justamente lo contrario de lo que hacía el canto.

En el momento en que le escribo estoy oyendo a la vez la risa y el canto. Le quedaré muy agradecido si me dice qué manipulaciones de órgano e incienso debo seguir para restablecer definitivamente la calma.

Ejercicio 4.11. Demuestre que cada una de las máquinas de estado finito siguientes es un autómata de estado finito y trace de nuevo el diagrama de transición como uno de un autómata de estado finito.



Ejercicio 4.12. ¿Son las palabras *abaa* y *abbaa* aceptadas por el reconocedor finito indicado por la figura?



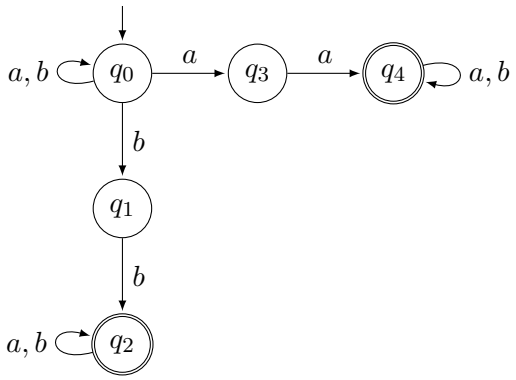
Ejercicio 4.13. Diseñe un reconocedor finito que acepte únicamente las palabras no nulas sobre $\{a, b\}$ que satisfagan las siguientes condiciones:

1. No contengan letras a.
2. Tienen un número par de letras aes.
3. Exactamente una letra b.
4. Exactamente dos letras a.
5. $\{\, \alpha \mid$ toda a de α está entre dos $bes \,\}$.
6. $\{\, \alpha \mid \alpha$ contiene la subcadena $abab \,\}$.
7. $\{\, \alpha \mid \alpha$ no contiene ninguna de las subcadenas aa o $bb \,\}$.
8. $\{\, \alpha \mid \alpha$ tiene un número impar de aes y un número par de $bes \,\}$.
9. $\{\, \alpha \mid \alpha$ tiene ab y ba como subcadenas $\,\}$.

Ejercicio 4.14. Dos reconocedores (autómatas) finitos RF_1 y RF_2 se dicen equivalentes si y sólo si aceptan el mismo lenguaje, es decir, $L(\mathsf{RF}_1) = L(\mathsf{RF}_2)$.

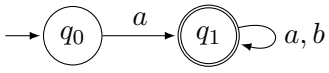
Sea RF el conjunto de todos los reconocedor finitos sobre un alfabeto Σ. Sea $R \subseteq \mathrm{RF} \times \mathrm{RF}$ la relación definida por: la pareja $(\mathsf{RF}_1, \mathsf{RF}_2)$ está en R si y sólo si RF_1 es equivalente a RF_2. Demuestre que la relación R es una relación de equivalencia en RF (y por tanto, que la definición de equivalencia de reconocedores finitos es consistente con el uso matemático habitual de los términos).

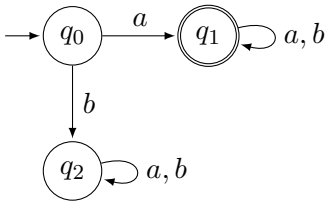
Ejercicio 4.15. Para el autómata de estado finito representado por la figura:



1. ¿Qué tipo de autómata es?
2. ¿Qué lenguaje reconoce?

Ejercicio 4.16. Para los autómatas de estado finito representados por la figuras:





1. ¿Cuál es un AFD?
2. ¿Cuál es un AFND?
3. Pruebe que reconocen el lenguaje dado por $a(a \mid b)^*$.
4. Justifique por qué son equivalentes.

Ejercicio 4.17. Diseñe un AFND que acepte únicamente los conjuntos de palabras dados por:

1. $\{a\}$.
2. $\{b\}$.
3. $\{a, b\}$.
4. $(a \mid b)^* \mid (aba)^+$.

5. Palabras de la forma *bowwow*, *bowwowwow*, *bowwowwowwow*, ...
6. Palabras de la forma *ohmy*, *ohmyohmy*, *ohmyohmyohmy*, ...
7. La unión de los dos lenguajes anteriores.

Ejercicio 4.18. Obtener los monoides del sumador y del restador binario y compararlos.

Ejercicio 4.19. Demuestre el teorema 4.36.

4.8. Notas bibliográficas

La presentación de un sistema desde la teoría de sistemas fue tomada de [Crespo 1983]. Las máquinas de estado finito son presentadas por [Crespo 1983; Fernández Fernández y Sáez Vacas 1987; Johnsonbaugh 1988; Kolman y Busby 1984; Minsky 1967], entre otros. Las máquinas de Moore y las máquinas de Mealy son presentadas por [Crespo 1983; Fernández Fernández y Sáez Vacas 1987]. Los autómatas de estado finito, tanto los deterministas como los no determinnistas, son presentados por [Crespo 1983; Fernández Fernández y Sáez Vacas 1987; Johnsonbaugh 1988; Kelley 1995]; el término de reconocedor finito es introducido por [Fernández Fernández y Sáez Vacas 1987]. Las secciones que establecen la relación entre álgebra, autómatas y lenguajes fueron adaptadas de [Fernández Fernández y Sáez Vacas 1987].

Capítulo 5

Autómatas de pila

En lo concerniente a nuestros desarrollos realizados en el contexto de los autómatas de estado finito es posible que hayamos notado, intuitivamente, que éstos tienen una memoria limitada, esto es, sólo tienen capacidad para una "memoria" finita. El problema se puede observar claramente en el contexto del reconocimiento de algunos lenguajes donde se requiere de un autómata que almace o guarde una gran cantidad de información. Tómese por ejemplo, el caso de lenguajes de la forma $\{ a^m c^m \mid m \geq 0 \}$ (independiente del contexto); para casos como éste, el autómata debería realizar diversos procesos como: debe verificar no sólo que toda 'a' preceda a toda 'c', sino que, además, tiene que contar el número de símbolos 'a' y de símbolos 'c'. Tendríamos entonces que limitar el número de símbolos 'a' que el autómata debe contar.

Necesitamos entonces, contar con autómatas que estén dotados de un dispositivo que les permita un almacenamiento no limitado de información y, además, con capacidad para guardar y comparar información. Este sería, por ejemplo, el caso para un autómata reconocedor del siguiente lenguaje independiente del contexto: $\{ c\alpha c\alpha^3 \mid \alpha \in \Sigma^*, \text{ y } \Sigma = \{a, b, c\} \}$.

Buscaremos, pues, construir formalmente un tipo de autómata que tenga las buenas propiedades antes mencionadas, de forma tal que lleguemos a solucionar el problema, al menos para cierto tipo de lenguajes; pues, como sabemos, existen lenguajes que no son reconocibles por ningún tipo de autómata finito de las clases que hemos venido considerando. El autómata que se requiere es conocido y se ha designado como autómáta de pila o autómata de *stack*.

En general, podemos decir que el comportamiento de un autómata de pila es muy similar a los autómatas de estado finito. La diferencia entre un autómata de pila y un autómata de estado finito estriba en que para el autómata de pila, además del estado actual y del símbolo de entrada, se debe considerar el símbolo de cierto alfabeto, que está (en el momento o tiempo considerado) en la cima de una pila, también llamada *stack*. La pila es, justamente, el nuevo tipo de dispositivo requerido para poder diseñar reconocedores para el tipo de lenguajes que hemos mencionado en esta introducción. Otro aspecto a resaltar en el

comportamiento o funcionamiento de un autómata de pila consiste en el siguiente hecho: un autómata de pila, además de requerir cambiar el estado actual cuando se tiene determinada configuración, también requiere cambiar la información que se halla en ese momento dado en la cima de la pila.

5.1. Autómata de pila no determinista

Desde un enfoque formal diremos que un autómata de pila no determinista es un estructura formal constituida como una héptupla, mediante la interrelación de un conjunto de estados, un alfabeto de entradas, un alfabeto de pila, una relación de transición, un conjunto de estados de aceptación y dos símbolos distinguidos (estado inicial y símbolo inicial de la pila).

Definición 5.1 (Autómata de pila no determinista)**.** Un autómata de pila no determinista (APND) (o autómata de *stack* no determinsta) está definido por la estructura matemática

$$\mathsf{APND} = \langle Q, \Sigma \cup \{\varepsilon\}, \Gamma, \delta, q_0, A, z_0 \rangle,$$

donde:

(i) Q es un conjunto de estados (finito y diferente de vacío).

(ii) Σ es un alfabeto de entrada (finito y diferente de vacío).

(iii) Γ es un alfabeto auxiliar o alfabeto de la pila (finito y diferente de vacío).

(iv) δ es un relación de transición.

(v) q_0 es el estado inicial del autómata ($q_0 \in Q$).

(vi) A es el conjunto de estados de aceptación ($A \subset Q$).

(vii) z_0 es el símbolo inicial de la pila ($z_0 \in \Gamma$).

Es importante que anotemos que en realidad δ sólo es una función para un autóma de pila determinista. Para el caso no determinista δ es una relación. De acuerdo con las precisiones hechas sobre el comportamiento de los autómatas no deterministas, diremos que δ es una relación tal que a cada terna $(q, s, z) \in Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$, asocia una o más parejas de la forma (q', β) donde $q' \in Q$ y $\beta \in \Gamma^*$.

Por consiguiente, para definir la regla de transición (δ), debemos considerar el estado actual q, el símbolo de entrada s en el momento considerado y la información que se halla en ese momento en la cima de la pila z; es decir, determinar la terna (q, s, z). Luego tenemos que considerar cuál debe ser la reacción del autómata (q') y cuál la información (β) que habrá de ser ubicada en la cima de la pila; es decir, determinar la pareja (q', β), donde la información a empilar β se ubica en el lugar del símbolo que se hallaba antes en la cima de la pila. Observemos igualmente que la naturaleza de la relación δ, es decir $\delta(q, s, z)$, obliga

a que se tenga siempre un símbolo en la cima de la pila, esto es, el autómata no podría efectuar ninguna transición si la pila está vacía.

Precisemos ahora algunas características fundamentales de un APND.

1. Para que un APND pueda efectuar algún movimiento es necesario que exista algún símbolo $z \in \Gamma$ en la cima de la pila. Ello se desprende de la definición de δ, ya que $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^*$.

2. Para cumplir la condición anterior es necesario precisar un símbolo inicial para la pila. $z_0 \in \Gamma$.

3. Dado que ε es la palabra vacía y que se puede tener que $\delta(q, \varepsilon, z) = \{(q', zz)\}$, entonces es posible, por ejemplo, que un APND cambie de estado y apile un símbolo $z \in \Gamma$ sin que ocurra ninguna entrada.

4. Como δ no necesariamente es una función, entonces puede ocurrir que existan ternas (q, a, z) tales que:

   a) $\delta(q, a, z)$ no existe y el APND se detiene.

   b) $\delta(q, a, z) \in Q \times \Gamma^*$ y $\overline{\overline{\delta(q, a, z)}} > 1$, es decir, $\delta(q, a, z)$ tiene más de una imágen, por lo tanto, la transición la realizará el APND de un modo no determinista.

5. Si $q_j \in A$, esto es, q_j es un estado de aceptación y $\delta(q, a, z) = \{(q_j, \beta)\}$, entonces el autómata se detiene, es decir, un APND se detiene toda vez que realice una transición a un estado de aceptación.

Ejemplo 5.2. Sea $\mathsf{APND} = \langle Q, \Sigma \cup \{\varepsilon\}, \Gamma, \delta, q_0, A, z_0 \rangle$ un autómata de pila no determinista, donde $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{c, d\}$, $\Gamma = \{A, B\}$, q_0 es el estado inicial, $A = \{q_2\}$, $z_0 = B$ y la relación de transición δ la definimos mediante la tabla de transiciones indicada por la tabla 5.1.

δ	(c, A)	(d, A)	(c, B)	(d, B)	(ε, A)	(ε, B)
q_0	(q_1, BA)	(q_2, AA)		(q_1, BB)	(q_2, A)	(q_2, B)
q_1	$\{(q_0, \varepsilon), (q_2, BA)\}$		(q_0, ε)	(q_1, BB)	(q_0, AA)	(q_0, B)
q_2				(q_1, BB)		(q_2, ε)

Tabla 5.1: Tabla de transición δ para el ejemplo 5.2.

A partir de la tabla de transición 5.1, podemos inferir los siguientes aspectos:

1. La relación δ depende del estado actual (las filas), del símbolo de entrada y del símbolo actual en la cima de la pila (etiquetas de columnas).

2. En el cruce de filas y columnas se especifica un estado siguiente y una acción de la pila para los símbolos actuales de la entrada y de la cima de la pila. Así, estando en el estado q_1, con entrada c y con B como símbolo actual en la cima de la pila, el autómata hace transición al estado q_0 y desempila a B.

3. Igualmente, puede verse de la tabla que no necesariamente hay transición del autómata para todas las posibles ternas de $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$. De allí que si el autómata pasa a un estado q_i para el cual no se programó su transición, entonces el autómata se detiene. Por ejemplo, cuando el autómata pasa al estado q_2 con A en la cima de la pila, observamos que no hay transición programada.

4. $\delta(q_1, c, B) = \{(q_0, \varepsilon)\}$, indica que el autómata estando en el estado q_1, con c como entrada y B en la cima de la pila, desempila la B y pasa al estado q_0.

5. $\delta(q_1, c, A) = \{(q_0, \varepsilon), (q_2, BA)\}$, indica que hay dos posibles respuestas, y que una de ellas será seleccionada de modo no determinista: estando en q_1 y con (c, A), el autómata, o bien desempila a A y pasa al estado q_0, o bien empila a B y pasa el estado q_2 (estado de aceptación).

6. Observemos que cuando el autómata está en el estado q_1 con (d, B), éste permanece en ese estado empilando signos B; y cuando está en q_1, con (c, B) pasa a q_0 y desempila a B.

7. Sea la palabra $dcdc \in \Sigma^*$; entonces, dado que $\delta(q_0, d, B) = \{(q_1, BB)\}$ tenemos que

$$\begin{aligned}\delta(q_0, dcdc, B) &= \delta(q_1, cdc, BB)\\ &= \delta(q_0, dc, B)\\ &= \delta(q_1, c, BB)\\ &= \delta(q_0, \varepsilon, B)\\ &= \delta(q_2, \varepsilon, B)\\ &= (q_2, \varepsilon).\end{aligned}$$

   Luego, el autómata se detiene en el estado de aceptación q_2, con la pila vacía. No es posible ningún movimiento.

   Eso significa que el autómata para computar sobre una palabra $\alpha \in \Sigma^*$ parte de la configuración inicial $\delta(q_0, s_1, B)$, donde q_0 es el estado inicial, B es el símbolo inicial de la pila y s_1 es el símbolo más a la izquierda de la palabra α.

5.2. Autómatas de pila y reconocedores

De modo similar a los autómatas que hemos estudiado en capítulos anteriores, un APND genera cierto lenguaje y puede servir de reconocedor de algunos lenguajes.

Definición 5.3 (Lenguaje generado por un APND)**.** Sea un autómata de pila no determinista $\mathsf{APND} = \langle Q, \Sigma \cup \{\varepsilon\}, \Gamma, \delta, q_0, A, z_0 \rangle$. El lenguaje generado por APND, denotado por $L(\mathsf{APND})$, está definido por el conjunto:

$$L(\mathsf{APND}) = \{\, \alpha \in \Sigma^* \mid (q_0, \alpha, z_0) \vdash_\delta (q, \varepsilon, \theta), \text{ donde } q \in A \text{ y } \theta \in \Gamma^* \,\}.$$

La expresión $(q_0, \alpha, z_0) \vdash_\delta (q, \varepsilon, \theta)$ significa que mediante la relación δ, partiendo de la configuración inicial (q_0, α, z_0), el autómata realiza sucesivas computaciones hasta que α se agote y termina en la configuración final (q, ε, θ); donde, q es un estado de aceptación y $\theta \in \Gamma^*$ puede o no ser vacía, esto es, el autómata puede o no terminar con la pila vacía. Si la pila está vacía el autómata necesariamente se detiene.

Definición 5.4 (APND como reconocedor de lenguajes)**.** Sea L un lenguaje. Decimos que un autómata de pila no determinista APND es un reconocedor de L, si y sólo si, $L \subseteq L(\mathsf{APND})$.

Observación 5.5. Con el símbolo $\vdash_1$, indicamos la configuración inmediata.

Ejemplo 5.6. En el ejemplo 5.2, el autómata:

1. Reconoce la palabra d :

$$\begin{aligned}(q_0, d, B) &\vdash_1 (q_1, \varepsilon, BB)\\ &\vdash_1 (q_0, \varepsilon, BB)\\ &\vdash_1 (q_2, \varepsilon, BB)\\ &\vdash_1 (q_2, \varepsilon, B)\\ &\vdash_1 (q_2, \varepsilon, \varepsilon),\end{aligned}$$

   donde q_2 es un estado de aceptación.

2. No reconoce la palabra c, es decir, $(q_0, c, B) \vdash_1$?

3. Reconoce la palabra dc, esto es, $(q_0, dc, B) \vdash_\delta (q_2, \varepsilon, \varepsilon)$:

$$\begin{aligned}(q_0, dc, B) &\vdash_1 (q_1, c, BB)\\ &\vdash_1 (q_0, \varepsilon, B)\\ &\vdash_1 (q_2, \varepsilon, B)\\ &\vdash_1 (q_2, \varepsilon, \varepsilon).\end{aligned}$$

4. De (2) y (3) se observa que la palabra dcc no es reconocida, pero la palabra dcd sí lo es.

5. El autómata reconoce el lenguaje $L_0 = \{\, d^n \mid n \geq 0 \,\}$.

a) $(q_0, d, B) \vdash (q_2, \varepsilon, \varepsilon)$. Reconoce a d^1 por (1).

b) Reconoce a d^2:

$$\begin{aligned}(q_0, dd, B) &\vdash_1 (q_1, d, BB)\\ &\vdash_1 (q_1, \varepsilon, BBB)\\ &\vdash_1 (q_0, \varepsilon, BBB)\\ &\vdash_1 (q_2, \varepsilon, BBB)\\ &\vdash_1 (q_2, \varepsilon, BB)\\ &\vdash_1 (q_2, \varepsilon, B)\\ &\vdash_1 (q_2, \varepsilon, \varepsilon).\end{aligned}$$

c) De (5) se puede ver que reconoce las palabras $d^3, d^4, \ldots$

d) Luego $L_0 \subseteq L(\mathsf{APND})$.

6. El autómata reconoce a: $d, dc, dcd, dcdc, dcdcd, dcdcdc, dcdcdcd, \ldots$

7. $L(\mathsf{APND}) = \{\, (dc)^* d^n \mid n \geq 0 \,\}$.

Ejemplo 5.7. Sea L_1 un lenguaje sobre $\Sigma = \{a, b\}$, tal que L_1 contiene igual cantidad de símbolos a y de símbolos b.

Nos interesa construir un APND que acepte o reconozca a L_1. Para tal efecto debemos poder contar las ocurrencias de los símbolos a y b en una palabra dada de Σ^*. Aquí podemos usar la idea de pila, simplemente empilando cuando leamos a y desempilando cuando leamos b, o viceversa.

Sea APND un autómata de pila no determinista definido por la tabla de transiciones indicada por la tabla 5.2.

δ	(a, A)	(a, B)	(a, C)	(b, A)	(b, B)	(b, C)
q_0	(q_0, AA)	(q_0, ε)	(q_0, AC)	(q_0, ε)	(q_0, BB)	(q_0, BC)
q_1						

δ	(ε, A)	(ε, B)	(ε, C)
q_0			(q_1, C)
q_1			

Tabla 5.2: Tabla de transición δ para el ejemplo 5.7.

Además, $Q = \{q_0, q_1\}$, $\Sigma = \{a, b\}$, $\Gamma = \{A, B, C\}$, $A = \{q_1\}$ y $z_0 = C$.

Para computar sobre la palabra $\alpha \equiv bababaab$, el autómata APND realiza el siguiente proceso:

$$\begin{aligned}(q_0, bababaab, C) &\vdash_1 (q_o, ababaab, BC)\\ &\vdash_1 (q_0, babaab, C), \quad \text{desempila}\\ &\vdash_1 (q_0, abaab, BC), \quad \text{empila}\\ &\vdash_1 (q_0, baab, C), \quad \text{desempila}\\ &\vdash_1 (q_0, aab, BC), \quad \text{empila}\\ &\vdash_1 (q_0, ab, C), \quad \text{desempila}\\ &\vdash_1 (q_0, b, AC), \quad \text{empila}\\ &\vdash_1 (q_0, \varepsilon, C), \quad \text{desempila}\\ &\vdash_1 (q_1, \varepsilon, C), \quad \text{estado de aceptación},\end{aligned}$$

debido a que no hay programada transición para la configuración (q_1, ε, C), el autómata se detiene y $\alpha \in L(\mathsf{APND})$. Luego, $L(\mathsf{APND}) = L_1$, por ende, APND es un reconocedor de L_1. ¿Qué ocurriría en el proceso cuando la palabra no es reconocida por el autómata?

5.3. Lenguajes independientes del contexto y autómatas de pila

En esta sección nos ocuparemos de la tarea de establecer algunas conexiones entre los APND y los lenguajes independientes del contexto. Recordemos que un lenguaje independiente del contexto se corresponde con el lenguaje generado por una grámatica independiente del contexto, grámaticas cuyas reglas de producción son de la forma $A \to \gamma$; $\gamma \neq \varepsilon$, donde $A \in \mathbf{N}$ (no terminal) y $\gamma \in (\mathbf{N} \cup \mathbf{T})^*$.

La primera conexión que nos interesa establecer está dada por el siguiente teorema, el cual establece que para todo lenguaje independiente del contexto siempre es posible construir un reconocedor que sea APND.

Teorema 5.8. *Si L es un lenguaje independiente del contexto, entonces existe al menos un autómata de pila no determinista* APND *tal que,* $L(\mathsf{APND}) = L$.

Demostración. La demostración de este teorema es esencialmente constructiva. Supongamos que tenemos una grámatica $\mathsf{G} = \langle \mathbf{N}, \mathbf{T}, \mathbf{P}, \mathbf{I} \rangle$ independiente del contexto. Asumimos que $L = L(\mathsf{G})$, para alguna grámatica G independiente del contexto. Se trata entonces de construir un autómata de pila no determinista APND tal que $L(\mathsf{APND}) = L(\mathsf{G})$; esto es, el APND nos debe permitir validar el hecho de que todas las palabras de $L(\mathsf{G})$ son reconocidas por el autómata. Sin perder generalidad, asumiremos que las palabras generadas por el autómata son derivaciones por la izquierda.

Pasemos ahora a construir dicho autómata. Supongamos que sólo necesitamos tres estados. Sea entonces $\mathsf{APND} = \langle Q, \Sigma \cup \{\varepsilon\}, \Gamma, \delta, q_0, A, z_0 \rangle$, donde:
$Q = \{q_0, q_1, q_2\}$,
$\Sigma = \mathbf{T}$,
$\Gamma = \mathbf{N} \cup \mathbf{T} \cup \{\mathbf{I}\}$,
$A = \{q_2\}$.

La relación de transición interna δ, estará determinada por el siguiente tipo de reglas de transición:

R1. Introducimos el símbolo inicial $\mathbf{I}$ en la pila, esto es $\delta(q_0, \varepsilon, z_0) = \{(q_1, \mathbf{I}z_0)\}$.

R2. $\delta(q_1, \varepsilon, A) = \{\, (q_1, \alpha) \mid A \to \alpha \,\}$, para todo $A \in \mathbf{N}$ (símbolos no terminales). Esto es, si el símbolo que está en la cima de la pila es un símbolo no terminal, entonces empilamos α (lado derecho de la producción $A \to \alpha$), conservando el mismo estado.

R3. $\delta(q_1, x, x) = \{(q_1, \varepsilon)\}$ para todo $x \in \Sigma = \mathbf{T}$ (símbolos terminales). Esto es, si el símbolo de entrada (x) y el símbolo de la cima de la pila son idénticos, entonces lo desempilamos, conservando el mismo estado.

R4. $\delta(q_1, \varepsilon, z_0) = \{(q_2, z_0)\}$. Esto es, en la cima de la pila queda el símbolo inicial de la pila, el autómata en el estado de aceptación q_2, y entonces, el autómata de detiene y acepta la palabra de $L(\mathsf{G})$.

Ahora, es posible probar que si $\alpha \equiv x_1 \dots x_n$ es aceptada por este APND, entonces α es derivable de la gramática G, esto es, $\alpha \in L(\mathsf{G})$.

Si en el APND antes construido tenemos que $(q_1, x, A\beta) \vdash_1 (q_1, x, \theta\beta)$, entonces podemos tener en G que, $A \to \theta$; de allí que tengamos:

$$\begin{aligned}(q_1, x_1 \dots x_n, \mathbf{I}z_0) &\vdash_1 (q_1, x_1 \dots x_n, x_1\theta z_0)\\ &\vdash_1 (q_1, x_2 \dots x_n, \theta z_0)\\ &\vdots\\ &\vdash_1 (q_1, x_n, x_n z_0)\\ &\vdash_1 (q_1, \varepsilon, z_0)\\ &\vdash_1 (q_2, \varepsilon, z_0).\end{aligned}$$

Luego, aplicando las reglas de G a lo anterior, tenemos que:

$$\mathbf{I} \to x_1\theta \wedge x_1\theta \to x_1x_2\theta \wedge \cdots \wedge x_1 \dots x_{n-1}\theta \to x_1 \dots x_n,$$

luego $\mathbf{I} \to^* x_1 \dots x_n$, de allí que $\alpha \equiv x_1 \dots x_n$ que fue aceptada por el APND es también derivable de las reglas de producción de G.

Recíprocamente, si $\mathbf{I} \to^* x_1 \dots x_n$, entonces se tiene en G:

$$\mathbf{I} \to x_1\theta \wedge x_1\theta \to x_1x_2\theta \wedge \dots \wedge x_1 \dots x_{n-1}\theta \to x_1 \dots x_n,$$

de allí, se sigue que del APND aquí construido:

$$\begin{aligned}(q_1, x_1 \dots x_n, \mathbf{I}z_0) &\vdash_1 (q_1, x_1 \dots x_n, x_1\theta z_0)\\ &\vdash_1 (q_1, x_2 \dots x_n, \theta z_0)\\ &\vdots\\ &\vdash_1 (q_1, x_n, x_n z_0)\\ &\vdash_1 (q_1, \varepsilon, z_0)\\ &\vdash_1 (q_2, \varepsilon, z_0);\end{aligned}$$

luego, el autómata de pila no determinista reconoce α, esto es, $\alpha \in L(\mathsf{APND})$. □

Ejemplo 5.9. El siguiente ejemplo nos ilustra el proceso de construcción de un APND para la gramática $\mathsf{G} = \langle \mathbf{N}, \mathbf{T}, \mathbf{P}, \mathbf{I}\rangle$, donde, $\mathbf{N} = \emptyset$, $\mathbf{T} = \{a, b, c\}$, $\mathbf{P} = \{\mathbf{I} \to a\mathbf{I}a, \mathbf{I} \to b\mathbf{I}b, \mathbf{I} \to c\}$. El lenguaje independiente del contexto a reconocer es $L(\mathsf{G}) = \{\, \alpha c\alpha^{-n} \mid \alpha \in \{a, b\}^* \,\}$ (α^{-n} es la palabra inversa de α).

Construyamos el APND que corresponde a $L(\mathsf{G})$ aplicando las reglas de construcción señaladas en la demostración del teorema 5.8, así:

$$\begin{aligned}\delta(q_0, \varepsilon, z_0) &= \{(q_1, \mathbf{I}z_0)\}, && \text{(R-R1.)}\\ \delta(q_1, \varepsilon, \mathbf{I}) &= \{(q_1, a\mathbf{I}a), (q_1, b\mathbf{I}b), (q_1, c)\}, && \text{(R-R2.)}\\ \delta(q_1, a, a) &= \delta(q_1, b, b) = \delta(q_1, c, c) = \{(q_1, \varepsilon)\}, && \text{(R-R3.)}\\ \delta(q_1, \varepsilon, z_0) &= \{(q_2, z_0)\}.\end{aligned}$$

Mostremos que, por ejemplo, la palabra $abcba \in L(\mathsf{G})$ es reconocida por el autómata, así:

$$
\begin{aligned}
(q_0, abcba, z_0) &\vdash_1 (q_1, abcba, \mathbf{I}z_0) \\
&\vdash_1 (q_1, abcba, a\mathbf{I}az_0) \\
&\vdash_1 (q_1, bcba, \mathbf{I}az_0), && \text{desempila } a \\
&\vdash_1 (q_1, bcba, b\mathbf{I}baz_0) \\
&\vdash_1 (q_1, cba, \mathbf{I}baz_0), && \text{desempila } b \\
&\vdash_1 (q_1, cba, cbaz_0) \\
&\vdash_1 (q_1, ba, baz_0), && \text{desempila } c \\
&\vdash_1 (q_1, a, az_0), && \text{desempila } b \\
&\vdash_1 (q_1, \varepsilon, z_0), && \text{desempila } a \\
&\vdash_1 (q_2, \varepsilon, z_0),
\end{aligned}
$$

luego, se reconoce *abcba*.

La segunda conexión que queremos esbozar hace relación con la pregunta de si todo lenguaje que sea reconocido por APND es un lenguaje independiente del contexto. Antes de presentar el teorema que establece esta conexión, introducimos primero una definición y un teorema cuya demostración dejamos como ejercicio.

Definición 5.10 (Lenguaje reconocido con una pila vacía)**.** Sea APND un autómata de pila no determinista. El siguiente conjunto define el lenguaje reconocido por la pila vacía del autómata:

$$P(\mathsf{APND}) = \{\, \alpha \in \Sigma^* \mid (q_0, \alpha, z_0) \vdash_\delta (q, \varepsilon, \varepsilon) \,\},$$

esto es, $P(\mathsf{APND})$ es el conjunto de entradas α que generan una sucesión de computaciones sucesivas que se inician en el estado inicial y culminan en un estado cualquiera q con la pila vacía.

Teorema 5.11. *Si* A *es un autómata de pila no determinista tal que*

$$P(\mathsf{A}) \neq L(\mathsf{A}),$$

entonces existe un autómata de pila no determinista A' *tal que*

$$L(\mathsf{A}) = P(\mathsf{A}').$$

Demostración. Ejercicio 5.12. □

Lo que afirma el teorema anterior es que podemos hallar un autómata de pila no determinista A' tal que el lenguaje reconocido por la pila vacía de A', coincida con el lenguaje reconocido por un autómata de pila no determinista A. Este teorema tiene sentido pues estos dos lenguajes no siempren son iguales para un autómata dado, como lo indica el siguiente ejemplo.

Ejemplo 5.12. Considérese el caso de un autómata de pila no determinista A cuya relación de transición sea: $\delta(q_0, c, z_0) = (q_0, cz_0)$, $\delta(q_0, a, z_0) = (q_1, \varepsilon)$, $\delta(q_0, c, c) = (q_2, \varepsilon)$. Aquí, $P(\mathsf{A}) = \{a\}$ y $L(\mathsf{A}) = \{cc\}$; donde el conjunto de estados de aceptación es $A = \{q_2\}$.

El hecho básico del teorema 5.11 consiste en construir un autómata de pila no determinista A' de modo tal que el conjunto de estados de aceptación sea un conjunto unitario, estado al cual A' alcanzaría sólo con la pila vacía.

Presentamos ahora el teorema que establece la segunda conexión entre los APND y los lenguajes independientes al contexto, mencionada anteriormente.

Teorema 5.13. *Si L es un lenguaje reconocido por un APND, entonces L es un lenguaje independiente del contexto.*

Demostración. La demostración de este teorema es esencialmente constructiva. Se trata de construir, a partir de una APND dado, una gramática independiente del contexto que le sea correspondiente, y de modo tal que una derivación en esta gramática simule los movimientos que el autómata haría para reconocer la palabra derivada.

Sea $\mathsf{A} = \langle Q, \Sigma \cup \{\varepsilon\}, \Gamma, \delta, q_0, A, z_0 \rangle$, un autómata de pila no determinista. Para hallar una gramática independiente del contexto cuyas reglas interpreten o simulen el comportamiento de A, se realiza el siguiente proceso constructivo:

1. Hallamos un APND A' equivalente al autómata dado, tal que A' sólo contenga un estado de aceptación q_f y llegue a él con la pila vacía (teorema 5.11).

2. Elegimos como símbolo inicial de la gramática $\mathbf{I} = [q_0 z_0 q_f]$. La intepretación se realiza en el paso siguiente.

3. Los elementos de $\mathbf{N}$ o símbolos no terminales se generan como expresiones de la forma $[qAq']$, donde $A \in \Gamma$ y $q, q' \in Q$. Interpretamos que $[qAq'] \to \alpha$, simula la acción del autómata, consistente en (estando en el estado actual q) desempilar a A y moverse al estado q' mientras consume la entrada α . Veamos, entonces, cómo se generan las reglas de producción para la gramática correspondiente al APND dado.

   a) Si $(q_j, \varepsilon) \in \delta(q_i, x, A)$ entonces se tiene la regla $[q_i A q_j] \to x$.

   b) Si $(q_j, BC) \in \delta(q_i, x, A)$ (la entrada x desempila a A, pero el autómata, en su proceso, debe desempilar a B y a C, para poder llegar a q_f con la pila vacía), entonces, se incluyen todas las producciones de la forma $[q_i A q_m] \to x[q_i B q_n]\ [q_n C q_m]$, donde q_m y q_n son cualquier estado del autómata. □

Ejemplo 5.14. Consideremos el autómata $\mathsf{APND} = \langle Q, \Sigma \cup \{\varepsilon\}, \Gamma, \delta, q_0, A, z_0 \rangle$ tal que $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, $\Gamma = \{A, z_0\}$, $A = \{q_2\}$ y la relación de transición δ definida

por:

$$\delta(q_0, a, z_0) = \{(q_0, Az_0)\}, \tag{5.1a}$$
$$\delta(q_0, a, A) = \{(q_0, AA)\}, \tag{5.1b}$$
$$\delta(q_0, b, A) = \{(q_1, \varepsilon)\}, \tag{5.1c}$$
$$\delta(q_1, b, A) = \{(q_1, \varepsilon)\}, \tag{5.1d}$$
$$\delta(q_1, \varepsilon, A) = \{(q_1, \varepsilon)\}, \tag{5.1e}$$
$$\delta(q_1, \varepsilon, z_0) = \{(q_2, \varepsilon)\}. \tag{5.1f}$$

Este autómata satisface la condición de que sólo hace transición al estado de aceptación q_2 cuando la pila esté vacía. Además, en la demostración del teorema 5.11 se verifica que las transiciones del autómata equivalente son de la forma $\delta(q_i, x, A) = \{c_1, c_2, \ldots, c_k\}$, donde las c_i son de la forma $c_i = (q, \varepsilon)$ ó $c_i = (q, BC)$ (sólo se empila o desempila con un único símbolo de Γ).

1. Símbolo inicial de la gramática: $\mathbf{I} = [q_0 z_0 q_2]$.

2. Reglas de producción de la gramática correspondientes a APND:

   a) Las reglas correspondientes a las transiciones (5.1c), (5.1d), (5.1e) y (5.1f) son del tipo (3*a*) (teorema 5.13). Luego, tenemos de ellas:
   Para (5.1c): $[q_1 A q_1] \to b$,
   para (5.1d): $[q_1 A q_1] \to b$,
   para (5.1e): $[q_1 A q_1] \to \varepsilon$,
   para (5.1f): $[q_1 z_0 q_2] \to \varepsilon$.

   b) Las reglas correspondientes a las transiciones (5.1a) y (5.1b) son del tipo (3*b*) (teorema 5.13). Luego, tenemos de ellas:
   Para (5.1a):

$$\begin{aligned}
[q_0 z_0 q_0] &\to a\ [q_0 A q_0]\ [q_0 z_0 q_0] \\
&\to a\ [q_0 A q_1]\ [q_1 z_0 q_0] \\
&\to a\ [q_0 A q_2]\ [q_2 z_0 q_0], \\
[q_0 z_0 q_1] &\to a\ [q_0 A q_0]\ [q_0 z_0 q_1] \\
&\to a\ [q_0 A q_1]\ [q_1 z_0 q_1] \\
&\to a\ [q_0 A q_2]\ [q_2 z_0 q_1], \\
[q_0 z_0 q_2] &\to a\ [q_0 A q_0]\ [q_0 z_0 q_2] \\
&\to a\ [q_0 A q_1]\ [q_1 z_0 q_2] \\
&\to a\ [q_0 A q_2]\ [q_2 z_0 q_2].
\end{aligned}$$

Para (5.1b):

$$\begin{aligned}
[q_0Aq_0] &\to a\ [q_0Aq_0]\ [q_0Aq_0]\\
&\to a\ [q_0Aq_1]\ [q_1Aq_0]\\
&\to a\ [q_0Aq_2]\ [q_2Aq_0],\\
[q_0Aq_1] &\to a\ [q_0Aq_0]\ [q_0Aq_1]\\
&\to a\ [q_0Aq_1]\ [q_1Aq_1]\\
&\to a\ [q_0Aq_2]\ [q_2Aq_1],\\
[q_0Aq_2] &\to a\ [q_0Aq_0]\ [q_0Aq_2]\\
&\to a\ [q_0Aq_1]\ [q_1Aq_2]\\
&\to a\ [q_0Aq_2]\ [q_2Aq_2].
\end{aligned}$$

Ejemplo 5.15. Indiquemos el modo como la derivación en la gramática simula el comportamiento del reconocedor.

Por ejemplo, la cadena *aaabbb* es reconocida por el autómata anterior. Veamos la secuencia de configuraciones correspondientes.

$$\begin{aligned}
(q_0, aaabbb, z_0) &\vdash_1 (q_0, aabbb, Az_0), && \text{de regla (5.1a)}\\
&\vdash_1 (q_0, abbb, AAz_0), && \text{de regla (5.1b)}\\
&\vdash_1 (q_0, bbb, AAAz_0), && \text{de regla (5.1b)}\\
&\vdash_1 (q_0, bb, AAz_0), && \text{de regla (5.1c)}\\
&\vdash_1 (q_0, b, Az_0), && \text{de regla (5.1c)}\\
&\vdash_1 (q_0, \varepsilon, z_0), && \text{de regla (5.1d)}\\
&\vdash_1 (q_0, \varepsilon, \varepsilon), && \text{de regla (5.1f).}
\end{aligned}$$

Luego, termina en el estado de aceptación q_2 con la pila vacía.

Ahora, la derivación en la gramática que contruimos en el ejemplo 5.14 para la palabra *aaabbb*, viene dada por:

$$\begin{aligned}
[q_0z_0q_0] &\to a\ [q_0Aq_0]\ [q_0z_0q_0]\\
&\to aa\ [q_0Aq_1]\ [q_1Aq_1]\ [q_1z_0q_2]\\
&\to aaa\ [q_0Aq_1]\ [q_1Aq_1]\ [q_1Aq_1]\ [q_1z_0q_2]\\
&\to aaab\ [q_1Aq_1]\ [q_1Aq_1]\ [q_1z_0q_2]\\
&\to aaabb\ [q_1Aq_1]\ [q_1z_0q_2]\\
&\to aaabbb\ [q_1z_0q_2]\\
&\to aaabbb\varepsilon\\
&\to aaabbb
\end{aligned}$$

Para cerrar este capítulo sobre autómatas de pila, consideramos importante realizar las siguientes observaciones:

1. Los autómatas de pila nos sirven entonces para probar, no obstante cierta complejidad, si un determinado lenguaje es o no independiente del contexto. Incluso se puede afirmar que, en la mayoría de los casos, es más sencillo construir el autómata que diseñar la gramática independiente del contexto.

2. Los autómatas de pila pueden ser usados como métodos más expeditos para probar o verificar algunas propiedades específicas de los lenguajes independientes del contexto (como lo ejemplifica la próxima sección de ejercicios).

5.4. Ejercicios

Ejercicio 5.1. Dada la siguiente tabla de transición para un autómata de pila no determinista que designamos por APND donde $Q = \{q_0, q_1, q_2, q_3\}$, $A = \{q_3\}$ y $z_0 = A$:

δ	(a, A)	(b, A)	(ε, A)	(a, B)	(b, B)	(ε, B)
q_0	$\{(q_1, BA), (q_3, \varepsilon)\}$		(q_3, ε)			
q_1				(q_1, BB)	(q_2, ε)	
q_2					(q_2, ε)	
q_3						

1. Analizar el comportamiento del autómata APND, cuando de halla en los estados q_0, q_1 y q_2.

2. Analizar el comportamiento del autómata APND toda vez que entra b, con B en la cima de la pila.

3. ¿Qué significa el hecho de que $\delta(q_0, a, A) = \{(q_1, BA), (q_3, \varepsilon)\}$?

4. Verificar si la palabra *abbabb* pertenece a $L(\mathsf{APND})$.

5. Determinar $L(\mathsf{APND})$.

Ejercicio 5.2. Sea $\mathsf{APND} = \langle Q, \Sigma \cup \{\varepsilon\}, \Gamma, \delta, q_0, A, z_0 \rangle$ un autómata de pila no determinista, donde $Q = \{q_0, q_1\}$, $\Sigma = \{a, b\}$, $\Gamma = \{A, B, Z\}$, $A = \{q_1\}$, $z_0 = Z$ y la relación de transición δ definida por:

$$\begin{aligned}
\delta(q_0, \varepsilon, Z) &= \{(q_1, Z)\},\\
\delta(q_0, a, Z) &= \{(q_0, AZ)\},\\
\delta(q_0, b, Z) &= \{(q_0, BZ)\},\\
\delta(q_0, a, A) &= \{(q_0, AA)\},\\
\delta(q_0, b, A) &= \{(q_0, \varepsilon)\},\\
\delta(q_0, a, B) &= \{(q_0, \varepsilon)\},\\
\delta(q_0, b, B) &= \{(q_0, BB)\}.
\end{aligned}$$

1. Elaborar una tabla de transición para APND.
2. Verificar que la palabra $babaabab\varepsilon \in L(\mathsf{APND})$.
3. Hallar $L(\mathsf{APND})$.

Ejercicio 5.3. Para la gramática presentada en el ejemplo 5.9, verificar si la palabra $aabcaab \notin L(\mathsf{G})$ es reconocida por el APND presentado en ese ejemplo (sugerencia: en un momento se obtiene la configuración $(q_1, aab, baaz_0)$. ¿Existe regla de transición para seguir?)

Ejercicio 5.4. Sea $\#(\alpha, a)$ el número de símbolos a en la palabra α. Diseñar un APND tal que reconozca el lenguaje L donde $L = \{\, \beta \in \{a, b\}^* \mid \#(\beta, a) = \#(\beta, b) \,\}$.

Ejercicio 5.5. Diseñar un APND que sea un reconocedor para $L = \{\, c^n b^n \mid n \geq 0 \,\}$.

Ejercicio 5.6. Diseñar un APND que sea un reconocedor para $L = \{\, a^m b^{2m} \mid m \geq 0 \,\}$.

Ejercicio 5.7. Dado el autómata APND, definido por la siguiente tabla donde $A = \{q_2\}$:

δ	(a, z_0)	(a, b)	(b, a)	(b, b)
q_0	$\{(q_2, a), (q_1, \varepsilon)\}$			
q_1		(q_2, ε)	(q_1, b)	(q_1, b)
q_2				

1. Hallar el lenguaje generado por el autómata.
2. ¿Qué lenguaje será reconocido por el autómata?
3. ¿Es la palabra a^3b^4 reconocida por este autómata?
4. Si cambiamos $A = \{q_2\}$ por $A' = \{q_1, q_2\}$, ¿cuál será el lenguaje generado?

Ejercicio 5.8. Dada la gramática G definida por las siguientes producciones:

$$\begin{aligned} \mathbf{I} &\to aAA, \\ A &\to b\mathbf{I}, \\ A &\to a\mathbf{I}, \\ A &\to a. \end{aligned}$$

1. Hallar $L(\mathsf{G})$.
2. Diseñar un autómata de pila no determinista APND tal que $L(\mathsf{APND}) = L(\mathsf{G})$.
3. Mostrar que $a^3b^3 \in L(\mathsf{APND})$.
4. Producir a^3b^3 mediante G.

Ejercicio 5.9. Hallar una gramática independiente del contexto que sea correspondiente al siguiente APND donde $A = \{q_3\}$, $\Gamma = \{A, B, z_0\}$:

δ	(a, z_0)	(ε, z_0)	(a, A)	(b, z_0)	(b, A)
q_0	(q_0, Az_0)		(q_3, ε)		(q_1, ε)
q_1		(q_2, ε)			
q_2					
q_3		(q_0, Az_0)			

Ejercicio 5.10. Diseñar un APND que sea un reconocedor del lenguaje generado por la gramática indicada por las siguientes reglas:

$$\begin{aligned} \mathbf{I} &\to aABB, \\ \mathbf{I} &\to aAA, \\ A &\to aBB, \\ A &\to \varepsilon, \\ B &\to bBB, \\ B &\to a. \end{aligned}$$

Ejercicio 5.11. Diseñar una gramática que sea correspondiente con el lenguaje generado por el siguiente APND donde $Q = \{q_0, q_1\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, z_0\}$, $A = \{q_1\}$:

$$\begin{aligned} \delta(q_0, a, z_0) &= \{(q_0, az_0)\}, \\ \delta(q_0, b, a) &= \{(q_0, aa)\}, \\ \delta(q_0, a, a) &= \{(q_1, \varepsilon)\}. \end{aligned}$$

¿Qué tipo de gramática se obtuvo?, ¿por qué?

Ejercicio 5.12. Demostrar el teorema 5.11.

Ejercicio 5.13. Dada las siguientes transiciones que definen una cierta relación de transición:

$$\begin{aligned}\delta(q_0, a, z_0) &= \{(q_0, az_0)\},\\ \delta(q_0, a, A) &= \{(q_0, A)\},\\ \delta(q_0, b, A) &= \{(q_1, \varepsilon)\},\\ \delta(q_1, \varepsilon, z_0) &= \{(q_3, \varepsilon)\}.\end{aligned}$$

1. Determine el mínimo Q y el mínimo Σ.
2. ¿Cúal es el conjunto A?, ¿por qué?
3. Hallar el lenguaje generado por este autómata.
4. ¿El autómata cumple las condiciones del teorema 5.13?
5. Investigar qué debe hacerse para hallar un autómata equivalente que las cumpla.
6. Hallar una gramática independiente del contexto que genere el mismo lenguaje que este último autómata.

Ejercicio 5.14. Probar que si L_0 es un lenguaje independiente del contexto y L_1 es regular, entonces $L_0 \cap L_1$ es un lenguaje independiente del contexto (sugerencia: considere un APND que sea un reconocedor de L_0 y un autómata de estado finito que sea un reconocedor de L_1. Mediante una adecuada combinación de ellos, construya un APND que sea un reconocedor de $L_0 \cap L_1$).

5.5. Notas bibliográficas

Los autómatas de pila son presentados en Hopcroft y Ullman [1997] y Kelley [1995]. El ejemplo 5.9 y el ejercicio 5.14 fueron tomados de [Kelley 1995]. La demostración del teorema 5.13 es un reordenamiento de la demostración presentada por [Kelley 1995].

Capítulo 6

Complejidad algorítmica

Una vez establecido qué problemas, funciones o lenguajes son computables, la teoría de la complejidad algorítmica clasifica estos objetos de acuerdo con la cantidad de recursos necesarios para computarlos. En el contexto de la complejidad algorítmica es necesario precisar los siguientes elementos: (i) Objetos a computar, (ii) Modelo(s) de computación empleado(s), (iii) Modos de computación, (iv) Recursos a acotar y (v) Funciones de complejidad.

Iniciamos nuestro trabajo en modo de computación determinista y en un modelo de computación denominado máquinas de Turing k-cintas.

6.1. Máquinas de Turing k-cintas

La idea intuitiva de una máquina de Turing k-cintas es la de una máquina de Turing que opera simultáneamente con k cintas de trabajo, en donde existe una cabeza de lectura-escritura para cada cinta.

Definición 6.1 (Máquina de Turing k-cintas)**.** Definimos formalmente una máquina de Turing determinista k-cintas, para $k > 1$, mediante la estructura matemática

$$\mathsf{MT}_k = \langle Q, \Sigma, M, I \rangle,$$

donde:

(i) Q, Σ y M son los mismos conjuntos que para una máquina de Turing.

(ii) I es una función definida de $Q \times \Sigma^k$ en $\Sigma^k \times M^k \times Q \cup \{q_Y, q_N\}$.

Al definir I como un conjunto *finito* de instrucciones $I = \{i_0, i_1, i_2, \ldots, i_p\}$, cada i_j es una $(3k+2)$-tupla de la forma: $q_m \ s_{m_i} \ \ldots \ s_{m_k} \ s_{n_1} \ \ldots \ s_{n_k} \ m_1 \ \ldots \ m_k \ q_n$, donde $q_m \in Q$, $q_n \in Q \cup \{q_Y, q_N\}$, $s_{m_i}, s_{n_i} \in \Sigma$ y $m_i \in M$; además los estados $\{q_Y, q_N\} \notin Q$ (estos estados serán usados para la definición de lenguajes recursivos y lenguajes recursivamente enumerables presentadas a continuación).

6.2. Lenguajes recursivos y lenguajes recursivamente enumerables

Los objetos para los cuales se van a definir las cotas en los recursos son los lenguajes formales.

Definición 6.2 (Lenguaje recursivamente enumerable)**.** Sea Σ una alfabeto de una máquina de Turing k-cinta MT_k y sea $L \subseteq (\Sigma - \{\square\})^*$ un lenguaje. Se dice que la máquina MT_k acepta el lenguaje L si para todo $\alpha \in (\Sigma - \{\square\})^*$ se cumple:

1. Si $\alpha \in L$, entonces, la máquina MT_k se detiene en el estado q_Y.
2. Si $\alpha \notin L$, entonces, la máquina no se detiene en q_Y. Esto permite tres posibilidades:
   a) MT_k se detiene en el estado q_N.
   b) MT_k se detiene en un estado diferente a q_Y y a q_N.
   c) MT_k no se detiene.

Así, si existe una MT_k que acepte el lenguaje L, entonces se dice que el lenguaje L es recursivamente enumerable.

Una clase particular de lenguajes recursivamente enumerables son los lenguajes recursivos en los que la máquina de Turing siempre se detiene.

Definición 6.3 (Lenguaje recursivo)**.** Sea Σ una alfabeto de una máquina de Turing k-cinta MT_k y sea $L \subseteq (\Sigma - \{\square\})^*$ un lenguaje. Se dice que la máquina MT_k decide el lenguaje L si para todo $\alpha \in (\Sigma - \{\square\})^*$ se cumple:

1. Si $\alpha \in L$ entonces la máquina MT_k se detiene en el estado q_Y.
2. Si $\alpha \notin L$ entonces la máquina MT_k se detiene en el estado q_N.

Así, si existe MT_k que decida el lenguaje L, entonces, se dice que el lenguaje L es recursivo.

Enunciamos sin demostración el siguiente teorema, debido a que es una instancia particular del teorema 2.44.

Teorema 6.4. *Todo lenguaje recursivo es una lenguaje recursivamente enumerable.*

Ejemplo 6.5. Sea L un lenguaje palíndromo, es decir, para toda $\alpha \in L$, $\alpha \equiv \alpha^{-1}$. Se presenta el comportamiento de la máquina de Turing 2-cintas MT_2 que acepta a L, dejando los detalles de construcción de la máquina al lector (ejercicio 6.1).

Al inicio la máquina tiene la palabra

$$\beta \equiv a_1 a_2 \dots a_{m-1} a_m a_m a_{m-1} \dots a_2 a_1$$

cinta_1	$\begin{matrix}a_1\\\uparrow\end{matrix}$	a_2	$\ldots$	a_{m-1}	a_m	a_{m-1}	$\ldots$	a_1	a_2

cinta_2	$\uparrow$		$\ldots$				$\ldots$		

Tabla 6.1: Aceptación para L palíndromo por MT_2 (1).

cinta_1	a_1	a_2	$\ldots$	a_{m-1}	a_m	a_{m-1}	$\ldots$	a_2	$\begin{matrix}a_1\\\uparrow\end{matrix}$

cinta_2	a_1	a_2	$\ldots$	a_{m-1}	a_m	a_{m-1}	$\ldots$	a_2	$\begin{matrix}a_1\\\uparrow\end{matrix}$

Tabla 6.2: Aceptación para L palíndromo por MT_2 (2).

en la cinta_1, como lo indica la tabla 6.1.

A continuación la máquina copia la palabra β en la cinta_2, como lo indica la tabla 6.2.

Después la máquina mueve la cabeza_1 a la posición inicial como lo indica la tabla 6.3.

Después, la máquina compara la palabra

$$a_1a_2\ldots a_{m-1}a_ma_ma_{m-1}\ldots a_2a_1$$

de la cinta_1, con la palabra

$$a_1a_2\ldots a_{m-1}a_ma_ma_{m-1}\ldots a_2a_1$$

de la cinta_2. Esta comparación se realiza moviendo la cabeza_1 a la derecha y la cabeza_2 a la izquierda a medida que se van leyendo los símbolos, como lo indica la tabla 6.4.

Finalmente, si la comparación es exitosa, la máquina pasa al estado q_Y y se detiene; de lo contrario, la máquina pasa al estado q_N y se detiene.

6.3. Complejidad temporal determinista

El primer recurso de computación que vamos a analizar es el tiempo.

Definición 6.6 (Complejidad temporal para MT_k)**.** Sea MT_k una máquina de Turing k-cintas. El tiempo requerido para computar una entrada $\alpha \in (\Sigma - \{\square\})^*$ es el número de pasos necesarios para que la máquina MT_k se detenga, cuando al inicio en la cinta_1 está la palabra α.

Sea $T : \mathbb{N} \to \mathbb{N}$ una función. Se dice que la máquina MT_k opera con límite temporal $T(n)$, si para toda $\alpha \in (\Sigma - \{\square\})^*$ el tiempo requerido para procesarla está dado por $T(l(\alpha))$, donde $l(\alpha)$ es la longitud de la palabra α.

Como es de esperarse que cualquier máquina de Turing k-cintas requiera al menos $n+1$ pasos para procesar una palabra de longitud n, entonces, en realidad el límite temporal $T(n)$ significa: el máximo entre $n+1$ y $T(n)$. Es decir, nuestra convención será

$$T(n) \stackrel{\text{def}}{=} \mathsf{máx}(n+1, \lceil T(n) \rceil).$$

Ejemplo 6.7. Sea una máquina de Turing k-cintas con límite temporal $T(n) = n\log_2 n$. De acuerdo con la convención $T(n) \stackrel{\text{def}}{=} \mathsf{máx}(n+1, \lceil T(n) \rceil)$ tenemos por ejemplo que $T(1) = \mathsf{máx}(2, 0) = 2$, $T(2) = \mathsf{máx}(3, 2) = 3$ y $T(3) = \mathsf{máx}(4, 4\log_2 3) = 4\log_2 3$. Cuando seleccionar $n+1$ ó $n\log_2 n$, está sujeto a los comportamientos de ambas funciones, tal como lo indica la figura 6.1.

Definición 6.8 (Clase de complejidad temporal)**.** Sea L un lenguaje aceptado por una

cinta_1	a_1 ↑	a_2	$\ldots$	a_{m-1}	a_m	a_{m-1}	$\ldots$	a_2	a_1
cinta_2	a_1	a_2	$\ldots$	a_{m-1}	a_m	a_{m-1}	$\ldots$	a_2	a_1 ↑

Tabla 6.3: Aceptación para L palíndromo por MT_2 (3).

cinta_1	a_1	a_2 ↑	$\ldots$	a_{m-1}	a_m	a_{m-1}	$\ldots$	a_2	a_1
cinta_2	a_1	a_2	$\ldots$	a_{m-1}	a_m	a_{m-1}	$\ldots$	a_2 ↑	a_1
				⋮					
cinta_1	a_1	a_2	$\ldots$	a_{m-1} ↑	a_m	a_{m-1}	$\ldots$	a_2	a_1
cinta_2	a_1	a_2	$\ldots$	a_{m-1}	a_m	a_{m-1} ↑	$\ldots$	a_2	a_1

Tabla 6.4: Decisión para L palíndromo por MT_2 (4).

Ejemplo 6.9. Para el lenguaje presentado en el ejemplo 6.5, calculemos $T(n)$. Inicialmente supongamos que $l(\beta) = n$. El copiar la palabra en la cinta_2 requiere $n + 1$ operaciones. El llevar la cabeza_1 hasta el final de la cinta_1 requiere $n + 1$ operaciones. Finalmente, el comparar las dos cintas requiere $n + 1$ operaciones. Por lo tanto $T(n) = 3n + 3$, es decir, si L es un lenguaje palíndromo, entonces, $L \in \mathsf{TIEMPOD}(3n + 3)$.

6.4. Notación asintótica

En el contexto de la complejidad algorítmica no estamos preocupados por las constantes multiplicativas o aditivas de las funciones de complejidad $f(n)$, por lo tanto, para representar éstas usualmente se utiliza la notación asintótica. Para efectos de completitud se presentaran las tres notaciones asintóticas más usuales; sin embargo, sólo se trabajará con una de ellas.

La notación O denota una cota superior asintótica. Sea $g(n)$ una función, con $O(g(n))$

$n \log_2 n$

$n + 1$

Figura 6.1: $T(n) = \text{máx}(n + 1, \lceil T(n) \rceil)$.

se denota el conjunto de funciones asintóticamente acotadas superiormente por $g(n)$.

Definición 6.10 (Notación 0)**.** $O(g(n)) = \{f(n) \mid$ existen constantes positivas c y n_0 tal que $0 \leq f(n) \leq cg(n)$, para todo $n \geq n_0\}$.

La convención $f(n) = O(g(n))$, indica que $f(n) \in O(g(n))$. La figura 6.2 muestra el significado de la notación O.

La notación Ω denota una cota inferior asintótica. Sea $g(n)$ una función, con $\Omega(g(n))$ se denota el conjunto de funciones asintóticamente acotadas inferiormente por $g(n)$.

Definición 6.11 (Notación Ω)**.** $\Omega(g(n)) = \{f(n) \mid$ existen constantes positivas c y n_0 tal que $0 \leq cg(n) \leq f(n)$, para todo $n \geq n_0\}$.

La convención $f(n) = \Omega(g(n))$, indica que $f(n) \in \Omega(g(n))$. La figura 6.3 muestra el significado de la notación Ω.

La notación Θ denota una cota asintótica. Sea $g(n)$ una función, con $\Theta(g(n))$ se denota el conjunto de funciones asintóticamente acotadas por $g(n)$.

Definición 6.12 (Notación Θ)**.** $\Theta(g(n)) = \{f(n) \mid$ existen constantes positivas c_1, c_2 y n_0 tal que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, para todo $n \geq n_0\}$.

La convención $f(n) = \Theta(g(n))$, indica que $f(n) \in \Theta(g(n))$. La figura 6.4 muestra el significado de la notación Θ.

A continuación, sin demostración enunciamos los siguientes teoremas relacionados con la notación O.

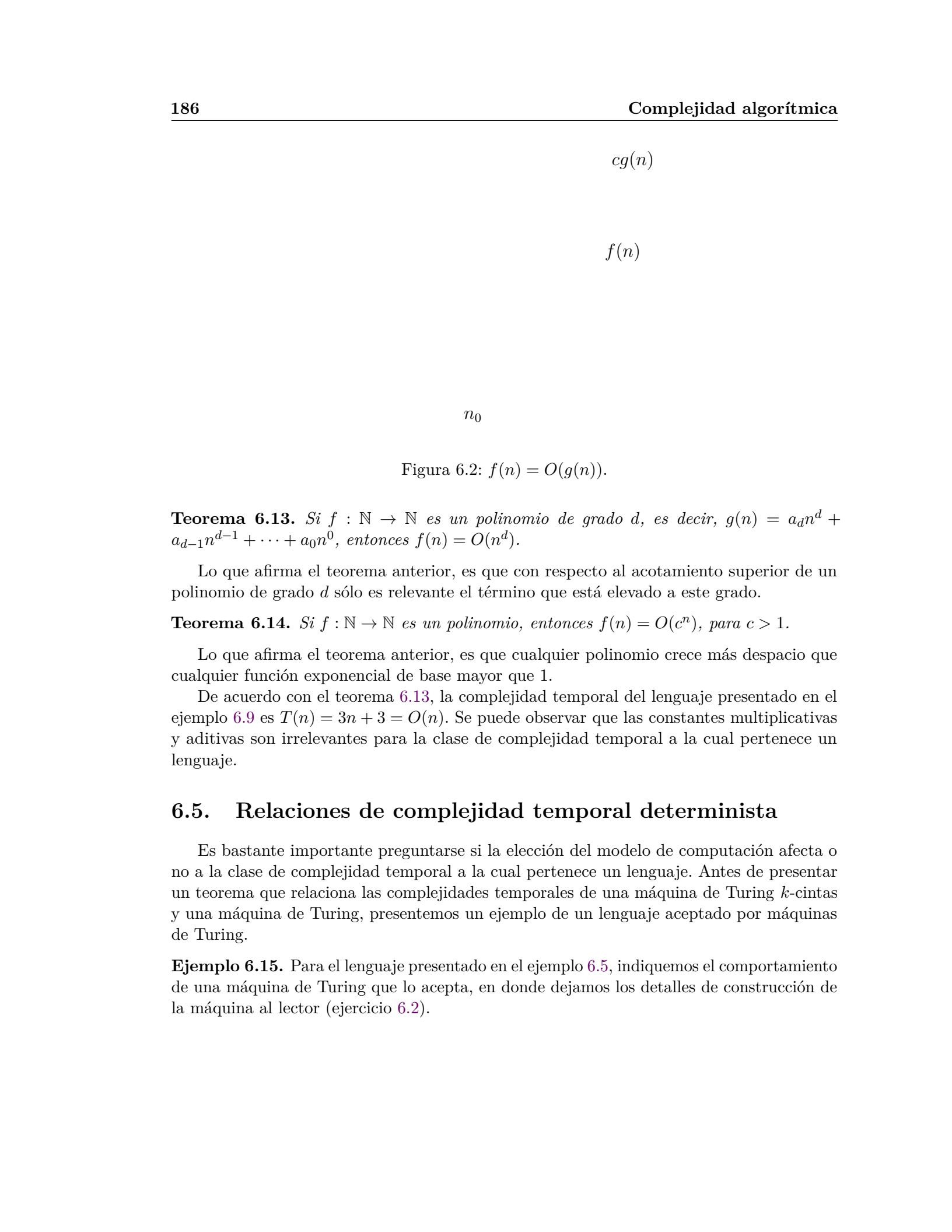


Figura 6.2: $f(n) = O(g(n))$.

Teorema 6.13. *Si $f : \mathbb{N} \to \mathbb{N}$ es un polinomio de grado d, es decir, $g(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_0 n^0$, entonces $f(n) = O(n^d)$.*

Lo que afirma el teorema anterior, es que con respecto al acotamiento superior de un polinomio de grado d sólo es relevante el término que está elevado a este grado.

Teorema 6.14. *Si $f : \mathbb{N} \to \mathbb{N}$ es un polinomio, entonces $f(n) = O(c^n)$, para $c > 1$.*

Lo que afirma el teorema anterior, es que cualquier polinomio crece más despacio que cualquier función exponencial de base mayor que 1.

De acuerdo con el teorema 6.13, la complejidad temporal del lenguaje presentado en el ejemplo 6.9 es $T(n) = 3n + 3 = O(n)$. Se puede observar que las constantes multiplicativas y aditivas son irrelevantes para la clase de complejidad temporal a la cual pertenece un lenguaje.

6.5. Relaciones de complejidad temporal determinista

Es bastante importante preguntarse si la elección del modelo de computación afecta o no a la clase de complejidad temporal a la cual pertenece un lenguaje. Antes de presentar un teorema que relaciona las complejidades temporales de una máquina de Turing k-cintas y una máquina de Turing, presentemos un ejemplo de un lenguaje aceptado por máquinas de Turing.

Ejemplo 6.15. Para el lenguaje presentado en el ejemplo 6.5, indiquemos el comportamiento de una máquina de Turing que lo acepta, en donde dejamos los detalles de construcción de la máquina al lector (ejercicio 6.2).

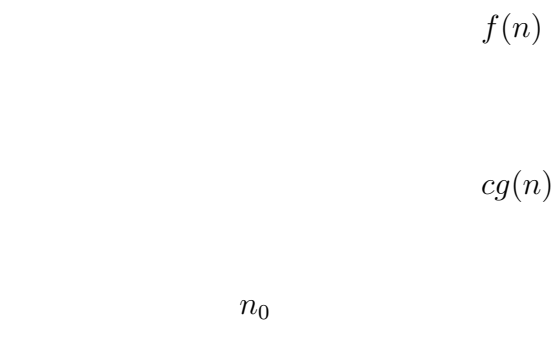


Figura 6.3: $f(n) = \Omega(g(n))$.

Al inicio la máquina tiene la palabra $\beta \equiv a_1 a_2 \dots a_{m-1} a_m a_m a_{m-1} \dots a_2 a_1$ en la cinta. A continuación la máquina compara el primer y último símbolo; los elimina, y después compara el segundo símbolo con el penúltimo, y así sucesivamente, como lo indica la tabla 6.5. Finalmente, si las comparaciones fueron exitosas, la máquina pasa al estado q_Y y se detiene; de lo contrario, la máquina pasa al estado q_N y se detiene.

Sea $l(\beta) = n$ la longitud de la palabra β. Para realizar la comporación del primer y último símbolo, la máquina necesita $2n + 1$ pasos. Después la máquina tiene una palabra de longitud $n - 2$ símbolos y la comparación del primer y último símbolo requiere $2(n - 2) + 1$ pasos. Después la máquina tiene una palabra de longitud $n - 4$ y la comparación del primer y último símbolo requiere $2(n - 4)$ pasos. Y así sucesivamente. Entonces la función de complejidad temporal viene dada por

$$\begin{aligned} T(n) &= \sum_{i=0}^{\frac{n}{2}} 2(n - 2i) + 1 \\ &= \frac{(n+1)(n+2)}{2} \\ &= 0(n^2). \end{aligned}$$

Es decir, si un lenguaje palíndromo L se acepta con una máquina de Turing, entonces, $L \in \mathsf{TIEMPO}(0(n^2))$. Pero si el lenguaje L se acepta con una máquina de Turing 2-cintas, entonces, $L \in \mathsf{TIEMPO}(0(n))$. Esta situación se generaliza con el siguiente teorema.

Teorema 6.16. *Si L es un lenguaje aceptado por una máquina de Turing k-cintas MT_k que opera con límite temporal $T(n)$, entonces existe una máquina de Turing MT con límite*

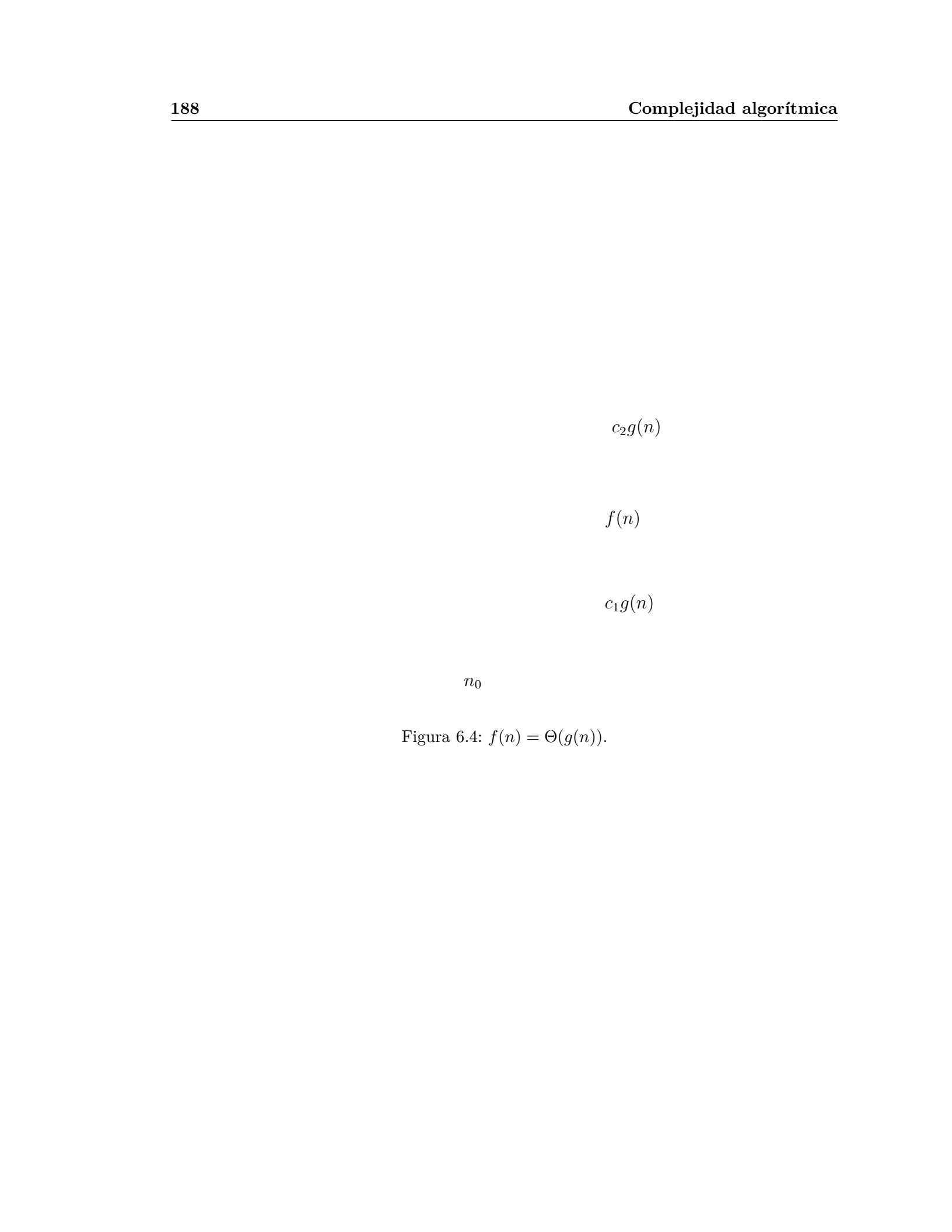


Figura 6.4: $f(n) = \Theta(g(n))$.

a_1 ↑	a_2	$\cdots$	a_{m-1}	a_m	a_{m-1}	$\cdots$	a_2	a_1
a_1	a_2	$\cdots$	a_{m-1}	a_m	a_{m-1}	$\cdots$	a_2	a_1 ↑
	a_2 ↑	$\cdots$	a_{m-1}	a_m	a_{m-1}	$\cdots$	a_2	
	a_2	$\cdots$	a_{m-1}	a_m	a_{m-1}	$\cdots$	a_2 ↑	
				$\vdots$				
		$\cdots$		a_m ↑		$\cdots$		

Tabla 6.5: Decisión para L palíndromo por una MT.

temporal $O(T(n)^2)$ *que acepta a* L.

El teorema anterior expresa que nuestro modelo inicial de computabilidad (la máquina de Turing), también es un buen modelo de complejidad algorítmica desde el punto de vista del recurso tiempo, pues una máquina de Turing k-cintas sólo nos proporciona una ganancia de orden polinomial con respecto a una máquina de Turing.

Ahora continuemos nuestro trabajo analizando el recurso espacial. Es decir, vamos a presentar algunos aspectos de complejidades algorítmicas espaciales. Inicialmente presentaremos el modelo de computación denominado máquina de Turing $(k, 1)$-cintas.

6.6. Máquinas de Turing $(k, 1)$-cintas

La idea intuitiva detrás de una máquina de Turing $(k, 1)$-cintas es la de una máquina de Turing (k+1)-cintas, la cual tiene una cinta de sólo lectura y k cintas de trabajo; es decir, contiene k cintas de lectura y escritura; de ahí la nomenclatura: máquina de Turing $(k, 1)$-cintas.

La idea de fijar una cinta de lectura, consiste en considerar que esta cinta no afectará el cálculo de la complejidad espacial de la máquina, pues, para una entrada de tamaño n, no se tendrá en cuenta el espacio ocupado por la entrada sino únicamente el espacio requerido para procesarla.

6.7. Complejidad espacial determinista

Definición 6.17 (Complejidad espacial para $\mathsf{MT}_{(k,1)}$)**.** Sea $\mathsf{MT}_{(k,1)}$ una máquina de Turing $(k,1)$-cintas. El espacio requerido para computar una entrada $\alpha \in (\Sigma - \{\square\})^*$ es el máximo de celdas utilizadas por cualquiera de la k cintas de trabajo para que la máquina $\mathsf{MT}_{(k,1)}$ se detenga, cuando al inicio en la cinta de lectura está la palabra α.

Sea $S : \mathbb{N} \to \mathbb{N}$ una función. Se dice que la máquina $\mathsf{MT}_{(k,1)}$ opera con límite espacial $S(n)$, si para toda $\alpha \in (\Sigma - \{\square\})^*$ el espacio requerido para procesarla está dado por $S(l(\alpha))$, donde $l(\alpha)$ es la longitud de la palabra α.

Como es de esperarse que cualquier máquina de Turing $(k,1)$-cintas requiere al menos una celda de alguna de las k cintas de trabajo para procesar una palabra de longitud n, entonces, en realidad el límite espacial $S(n)$ significa el máximo entre 1 y $S(n)$. Es decir, nuestra convención será

$$S(n) \stackrel{\text{def}}{=} \mathsf{máx}(1, \lceil S(n) \rceil).$$

Ejemplo 6.18. Sea una máquina de Turing $(k,1)$-cintas con límite espacial $S(n) = \mathsf{log}_2\, n$. De acuerdo a la convención $S(n) \stackrel{\text{def}}{=} \mathsf{máx}(1, \lceil S(n) \rceil)$, tenemos por ejemplo que: $S(1) = \mathsf{máx}(1, 0) = 1$, $S(2) = \mathsf{máx}(1, 1) = 1$ y $S(3) = \mathsf{máx}(1, \mathsf{log}_2\, 3) = \mathsf{log}_2\, 3$. Cuando, seleccionar 1 o $\mathsf{log}_2\, n$, está sujeto a los comportamientos de ambas funciones, tal como lo indica la figura 6.5.

$\mathsf{log}_2\, n$

1

Figura 6.5: $S(n) = \mathsf{máx}(1, \lceil S(n) \rceil)$.

Definición 6.19 (Clase de complejidad espacial para un lenguaje)**.** Sea L un lenguaje aceptado por una máquina de Turing $(k,1)$-cintas con límite espacial $S(n)$. Decimos entonces

que L pertenece a la clase de complejidad temporal denominada $\mathsf{ESPACIOD}(S(n))$. La clase $\mathsf{ESPACIOD}(S(n))$ es el conjunto formado por todos los lenguajes aceptados por máquinas de Turing $(k,1)$-cintas con límite espacial $S(n)$.

Ejemplo 6.20. Para el lenguaje presentado en el ejemplo 6.5, calculemos $S(n)$. Inicialmente supongamos que n es la longitud de la palabra inicial y que vamos a operar sobre una máquina de Turing $\mathsf{MT}_{(2,1)}$, es decir, sobre una máquina de Turing que contiene una cinta de lectura y dos cintas de trabajo.

Indiquemos el comportamiento de la máquina de Turing $(2,1)$-cintas $\mathsf{MT}_{(2,1)}$ que acepta a L, donde dejamos los detalles de construcción de la máquina al lector (ejercicio 6.3).

La máquina va a operar con dos contadores i, j escritos en binario en las cintas de trabajo. La cinta_i representará el contador i y la cinta_j representará el contador j. La máquina realiza dos bucles; un bucle externo desde $i = 1$ hasta $i \leq n$, y un bucle interno desde $j = 1$ hasta $j \leq i$. En cada iteración del bucle interno la máquina compara los valores i y j. Si $i < j$, incrementa el valor de j en uno. Si $i = j$, la máquina compara el i-ésimo símbolo de izquierda a derecha con el j-ésimo símbolo de derecha a izquierda de la palabra de entrada. Si la comparación es exitosa, entonces la máquina regresa al bucle externo; de lo contrario la máquina pasa al estado q_N y se detiene. Si la máquina finaliza el bucle externo, la máquina pasa al estado q_Y y se detiene.

Esta máquina requiere como máximo escribir el número n en notación binaria en las cintas de trabajo (cinta_i, cinta_j), por lo tanto, su límite espacial $S(n)$ está dado por $S(n) = \log_2 n = O(\log_2 n)$.

6.8. Relaciones de complejidad espacial entre los modelos de computación determinista

De nuevo es importante preguntarse si la elección del modelo de computación afecta o no la clase de complejidad espacial a la cual pertenece un lenguaje dado.

Teorema 6.21. *Si L es un lenguaje aceptado por una máquina de Turing k-cintas MT_k que opera con límite espacial $S(n)$, entonces existe una máquina de Turing $(k,1)$-cintas $\mathsf{MT}_{(k,1)}$ con límite espacial $O(S(n))$ que acepta a L.*

Demostración. La máquina $\mathsf{MT}_{(k,1)}$ copia la palabra de entrada en la primera cinta de trabajo y entonces puede operar como la máquina MT_k. □

Teorema 6.22. *Si L es un lenguaje aceptado por una máquina de Turing $(k,1)$-cintas $\mathsf{MT}_{(k,1)}$ que opera con límite espacial $S(n)$, entonces existe una máquina de Turing $(1,1)$-cintas $\mathsf{MT}_{(1,1)}$ con límite espacial $O(S(n))$ que acepta a L.*

Demostración. (indicación)

Sea Σ el alfabeto de la máquina $\mathsf{MT}_{(k,1)}$. Es posible considerar los símbolos de las k cintas de trabajo de la máquina $\mathsf{MT}_{(k,1)}$ como k-tuplas de Σ^k. Estas k-tuplas pueden ser codificadas por una alfabeto Σ' de 2^k símbolos. Entonces se contruye una máquina $\mathsf{MT}_{(1,1)}$ que opere con el alfabeto Σ' de forma equivalente a como opera la máquina $\mathsf{MT}_{(k,1)}$ con las k-tuplas. □

Teorema 6.23. *Si L es un lenguaje aceptado por una máquina de Turing $(1,1)$-cintas $\mathsf{MT}_{(1,1)}$ que opera con límite espacial $S(n)$, entonces existe una máquina de Turing MT con límite espacial $O(S(n))$ que acepta a L.*

Demostración. (indicación)

Para obtener la máquina MT se sigue un esquema de codificación similar al empleado en la demostración del teorema 6.22. □

De nuevo lo que indican los teoremas 6.21, 6.22 y 6.23 es que nuestro modelo inicial de computabilidad (la máquina de Turing), también es un buen modelo de complejidad algorítmica, desde el punto de vista del recurso espacial, pues una máquina de Turing $(k,1)$-cintas sólo proporciona una ganancia de orden constante con respecto a una máquina de Turing.

Mencionábamos al comienzo de este capítulo que uno de los aspectos importantes que debemos considerar en la teoría de complejidad es el modo de computación. Pero hasta el momento sólo hemos estudiado el modo de computación determinista. Abordamos ahora el estudio del modo de computación no determinista.

6.9. Máquina de Turing no determinista

Desde una perspectiva intuitiva podemos decir que una máquina de Turing no determinista es un máquina de Turing que puede seleccionar de un conjunto finito de posibilidades una acción o instrucción para ejecutar.

Definición 6.24 (Máquina de Turing no determinista)**.** Definimos una máquina de Turing no determinista por la estructura matemática

$$\mathsf{MTN} = \langle Q, \Sigma, M, I \rangle,$$

donde:

(i) Q, Σ y M son los mismos conjuntos que para una máquina de Turing.

(ii) I es una relación definida de $Q \times \Sigma$ en $\Sigma \times M \times Q \cup \{q_Y, q_N\}$, es decir, $I \subseteq Q \times \Sigma \times \Sigma \times M \times Q \cup \{q_Y, q_N\}$.

La diferencia esencial entre las definiciones formales de una máquina de Turing y una máquina de Turing no determinista radica en el símbolo I. En la primera, I es una función parcial, esto es, $I : Q \times \Sigma \to \Sigma \times M \times Q \cup \{q_Y, q_N\}$, mientras que en la segunda, I es una relación, esto es, $I \subseteq Q \times \Sigma \times \Sigma \times M \times Q \cup \{q_Y, q_N\}$.

Esta diferencia hace que las computaciones de una máquina de Turing no determinista presenten la forma de un árbol de computación, como se indica en la figura 6.6, mientras que las computaciones de una máquina de Turing presentan la forma de una línea de computación, como se indica en la figura 6.7.

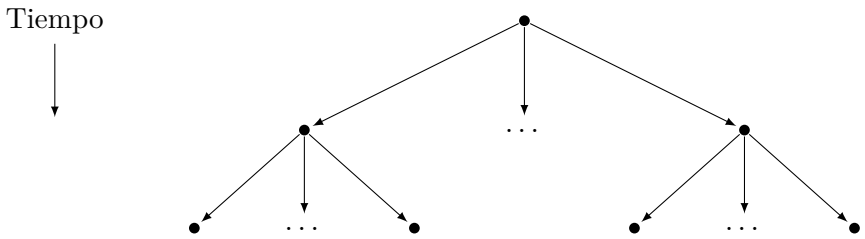


Figura 6.6: Árbol de computación: máquinas de Turing no deterministas.

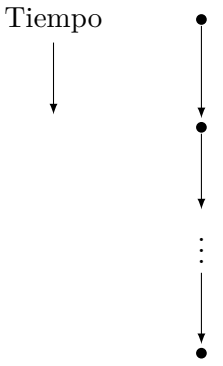


Figura 6.7: Línea de computación: máquinas de Turing.

6.10. Complejidad temporal y espacial no determinista

Definición 6.25 (Aceptación de lenguajes por MTN)**.** Sea Σ una alfabeto de una máquina de Turing no determinista MTN y sea $L \subseteq (\Sigma - \{\square\})^*$ un lenguaje. Se dice que la máquina MTN acepta el lenguaje L si para todo $\alpha \in (\Sigma - \{\square\})^*$ se cumple que:

1. Si $\alpha \in L$, entonces, existe al menos una posible selección de acciones de la máquina MTN tal que ésta se detiene en el estado q_Y.

2. Si $\alpha \notin L$, entonces, para cualquier posible acción de la máquina MTN, ésta no se detiene en el estado q_N.

Definición 6.26 (Complejidad temporal y espacial para MTN)**.** Sea una máquina de Turing no determinista MTN. El tiempo requerido para computar una entrada $\alpha \in (\Sigma - \{\square\})^*$ es el máximo de número de pasos necesarios para que la máquina MTN se detenga, bajo cualquier posible selección de acciones de la máquina, cuando al inicio en la cinta está la palabra α. Es decir, el tiempo requerido para computar α es la longitud del camino más largo en el árbol de computación de la máquina.

En otras palabras, el espacio requerido para computar una entrada $\alpha \in (\Sigma - \{\square\})^*$ es el máximo de celdas utilizadas por cualquiera de las posibles acciones de la máquina, de modo tal que ésta se detenga cuando al inicio en la cinta está la palabra α.

Sea $T : \mathbb{N} \to \mathbb{N}$ una función. Se dice que la máquina MTN opera con límite temporal $T(n)$ si para toda $\alpha \in (\Sigma - \{\square\})^*$ el tiempo requerido para procesarla está dado por $T(l(\alpha))$, donde $l(\alpha)$ es la longitud de la palabra α.

Sea $S : \mathbb{N} \to \mathbb{N}$ una función. Se dice que la máquina MTN opera con límite espacial $S(n)$ si para toda $\alpha \in (\Sigma - \{\square\})^*$ el espacio requerido para procesarla está dado por $S(l(\alpha))$, donde $l(\alpha)$ es la longitud de la palabra α.

Definición 6.27 (Clases de complejidad no deterministas)**.** Si L es un lenguaje aceptado por una máquina de Turing no determinista con límite temporal $T(n)$, entonces decimos que L pertenece a la clase de complejidad temporal denominada TIEMPON$(T(n))$. La clase TIEMPON$(T(n))$ es el conjunto formado por todos los lenguajes aceptados por máquinas de Turing no deterministas con límite temporal $T(n)$.

Si L es un lenguaje aceptado por una máquina de Turing no determinista con límite espacial $S(n)$, entonces se dice que L es un miembro de la clase de complejidad espacial ESPACION$(S(n))$. La clase ESPACION$(S(n))$ es el conjunto formado por todos los lenguajes aceptados por máquinas de Turing no deterministas con límite espacial $S(n)$.

6.11. Relaciones entre clases de complejidad

Existen algunas relaciones de inclusión entre las clases de complejidad temporal y espacial en los modos de computación determinista y no determinista que presentamos a continuación. Inicialmente presentamos las relaciones entre las clases de complejidad espacial determinista y no determinista.

Teorema 6.28. *Si* $L \in$ ESPACIOD$(S(n))$*, entonces* $L \in$ ESPACION$(S(n))$*. Es decir,* ESPACIOD$(S(n)) \subseteq$ ESPACION$(S(n))$*.*

Demostración. Toda máquina de Turing determinista es una máquina de Turing no determinista especial en donde esta última sólo tiene una posible acción en su conjunto de posibles acciones. Luego, si una máquina de Turing determinista tiene límite espacial $S(n)$,

la misma máquina considerada como una máquina de Turing no determinista tiene límite espacial $S(n)$. □

Para poder presentar el próximo teorema necesitamos dos definiciones. Estas definiciones están pensadas para elimininar ciertas funciones "patológicas" presentes en el conjunto de posibles funciones de complejidad espacial.

Definición 6.29 (Función espacialmente construible)**.** Sea $S : \mathbb{N} \to \mathbb{N}$ una función. Se dice que la función $S(n)$ es espacialmente construible si existe una máquina de Turing MT con límite espacial $S(n)$, tal que, para todo n existe una entrada α con $l(\alpha) = n$, la máquina MT utiliza $S(n)$ celdas.

Ejemplo 6.30. La función $\log_2 n$ es espacialmente construible. Sea $\mathsf{MT}_{(1,1)}$ una máquina de Turing con una cinta de lectura y una cinta de trabajo, y sea $\alpha \equiv a_1 \dots a_n$. La máquina por cada símbolo a_i escribe en la cinta de trabajo correspondiente el dígito i en binario. La máquina $\mathsf{MT}_{(1,1)}$ tiene límite espacial $\log_2 n$, donde $n = l(\alpha)$. Entonces, por el teorema 6.23 existe una máquina de Turing MT con límite espacial $\log_2 n$ equivalente a la máquina $\mathsf{MT}_{(1,1)}$.

Teorema 6.31. *Si $f(n)$ y $g(n)$ son funciones espacialmente construibles, entonces las funciones $f(n) + g(n)$, $f(n)g(n)$ y $2^{f(n)}$ son espacialmente construibles.*

Demostración. Ejercicio 6.4. □

Definición 6.32 (Construcción de manera completa)**.** Sea $S : \mathbb{N} \to \mathbb{N}$ una función. Se dice que la función $S(n)$ es espacialmente construible de manera completa si existe una máquina de Turing MT con límite espacial $S(n)$, tal que, para todo n y para toda entrada α con $l(\alpha) = n$, la máquina MT utiliza $S(n)$ celdas.

Teorema 6.33. *Si $f(n)$ es una función espacialmente construible y $f(n) \geq n$, entonces la función $f(n)$ es espacialmente construible de manera completa.*

Demostración. Ejercicio 6.5. □

Teorema 6.34 (Teorema de Savitch)**.** *Si $L \in \mathsf{ESPACION}(S(n))$, la función $S(n)$ es espacialmente construible de manera completa y $S(n) \leq \log_2 n$, entonces*

$$L \in \mathsf{ESPACIOD}((S(n))^2).$$

Demostración. Ejercicio 6.7. □

Presentemos ahora las relaciones entre las clases de complejidad temporal determinista y no determinista.

Teorema 6.35. *Si $L \in \mathsf{TIEMPOD}(T(n))$, entonces $L \in \mathsf{TIEMPON}(T(n))$. Es decir, $\mathsf{TIEMPOD}(T(n)) \subseteq \mathsf{TIEMPON}(T(n))$.*

Demostración. De la demostración del teorema 6.28, sabemos que si una máquina de Turing determinista tiene límite temporal $T(n)$, entonces la misma máquina considerada como una máquina de Turing no determinista tiene límite espacial $T(n)$. □

Teorema 6.36. *Si* $L \in \mathsf{TIEMPON}(f(n))$*, entonces existe una constante* d *tal que* $L \in \mathsf{TIEMPOD}(d^{f(n)})$*. Es decir,* $\mathsf{TIEMPON}(f(n)) \subseteq \mathsf{TIEMPOD}(d^{f(n)})$*.*

Demostración. Si se conoce el número de estados, el número de símbolos del alfabeto y la complejidad temporal de una máquina de Turing no determinista, es posible conocer el número máximo de descripciones instantáneas de la máquina, y por ende, es posible conocer el número máximo de celdas necesarias para decidir un lenguaje.

Sea MTN una máquina de Turing no determinista con límite temporal $T(n)$. En $T(n)$ movimientos la máquina puede inspeccionar como máximo $T(n)+1$ celdas. Si la máquina tiene $\overline{\overline{\Sigma}}$ símbolos, entonces existen como máximo $\overline{\overline{\Sigma}}^{T(n)+1}$ cadenas de longitud $T(n)+1$ de símbolos de Σ. La cabeza de lectura-escritura puede estar en cualesquiera de las $T(n)+1$ celdas, y para cada cadena el estado actual puede ser uno de los $\overline{\overline{Q}}$ estados. Por lo tanto, el número máximo de descripciones instantáneas para MTN es $\overline{\overline{Q}}(T(n)+1)\overline{\overline{\Sigma}}^{T(n)+1}$. Luego, existe c tal que $c^{T(n)} \geq \overline{\overline{Q}}(T(n)+1)\overline{\overline{\Sigma}}^{T(n)+1}$; es decir, el número de descripciones instantáneas de MTN está limitado por $c^{T(n)}$.

Una máquina de Turing MT puede simular una MTN que acepta o no un lenguaje de la siguiente manera. A partir de la descripción instantánea inicial, se genera sobre la cintas las descripciones instantáneas que se pueden alcanzar en un paso de computación de MTN; a partir de éstas, genera las que se pueden alcanzar en el siguiente paso de computación de MTN, y así sucesivamente hasta $T(n)$ pasos de computación de MTN. Sea $T(n)+2$ la longitud de una descripción instantánea, pues una máquina en $T(n)$ pasos puede escribir $T(n)+1$ símbolos y añadimos el símbolo del estado actual. Para generar una descripción instantánea son necesarios $3(T(n)+2)$ pasos; esto incluye escribir sus símbolos y buscar el estado q_Y; si lo encuentra termina la simulación; de lo contrario, genera la siguiente descripción instantánea. De cada descripción instantánea hay como máximo $c^{T(n)}$ descripciones instantáneas siguientes y como existen $T(n)$ pasos de computación, la cantidad total de movimientos es $3(T(n)+2)T(n)c^{T(n)}$. Por lo tanto, existe una constante d tal que $d^{T(n)} \geq 3(T(n)+2)T(n)c^{T(n)}$. □

Los teoremas que presentamos a continuación presentan las relaciones entre las clases de complejidad temporal y espacial deterministas.

Teorema 6.37. *Si* $L \in \mathsf{TIEMPOD}(f(n))$*, entonces* $L \in \mathsf{ESPACIOD}(O(f(n)))$*. Es decir,* $\mathsf{TIEMPOD}(f(n)) \subseteq \mathsf{ESPACIOD}(O(f(n)))$*.*

Demostración. Si una máquina de Turing realiza a lo sumo $f(n)$ pasos de computación, entonces ésta puede utilizar a lo sumo $f(n)+1$ celdas de su cinta. □

Teorema 6.38. *Si* $L \in \mathsf{ESPACIOD}(f(n))$ *y* $f(n) \leq \log_2 n$*, entonces existe una constante* c *tal que* $L \in \mathsf{TIEMPOD}(c^{f(n)})$*. Es decir, la clase* $\mathsf{ESPACIOD}(f(n))$ *está contenida en la clase* $\mathsf{TIEMPOD}(c^{f(n)})$*.*

Demostración. Si se conoce el número de estados, el número de símbolos del alfabeto y la complejidad espacial de una máquina de Turing, es posible conocer el número máximo de descripciones instantáneas de la máquina, y de allí conocer el número máximo posible de pasos que la máquina puede realizar para decidir un lenguaje.

Sea MT una máquina de Turing con límite espacial $f(n)$. Esta máquina puede escribir $\overline{\overline{\Sigma}}$ símbolos en un espacio de $f(n)$ de celdas de $\overline{\overline{\Sigma}}^{f(n)}$ formas distintas; además, para cada una de estas formas, la máquina puede estar en $\overline{\overline{Q}}$ estados diferentes y la cabeza de lectura-escritura puede estar en $f(n)$ posiciones diferentes. Por lo tanto el número máximo de descripciones instantáneas para MT está dado por $\overline{\overline{Q}} f(n) \overline{\overline{\Sigma}}^{f(n)}$.

Si, $f(n) \leq \log_2 n$, entonces existe c tal que $c^{f(n)} \geq \overline{\overline{Q}} f(n) \overline{\overline{\Sigma}}^{f(n)}$ para toda $n > 1$. Por lo tanto el número máximo de descripciones instantáneas tiene límite superior $c^{f(n)}$; de donde se concluye que una máquina de Turing que sea capaz de ejecutar todas estas posibles descripciones instantáneas debe tener como límite temporal $\mathsf{TIEMPOD}(c^{f(n)})$. □

La tabla 6.6 presenta las relaciones entre las clases de complejidad indicadas por los teoremas 6.28, 6.34, 6.35, 6.36, 6.37 y 6.38. Las restantes relaciones entre clases de complejidad pueden ser obtenidas a partir de esta tabla.

Relación	
Espacial	$\mathsf{ESPACIOD}(S(n)) \subseteq \mathsf{ESPACION}(S(n))$
	$\mathsf{ESPACION}(S(n)) \subseteq \mathsf{ESPACIOD}((S(n))^2)$ si $S(n)$ es una función espacialmente construible de manera completa y $S(n) \geq \log_2 n$
Temporal	$\mathsf{TIEMPOD}(T(n)) \subseteq \mathsf{TIEMPON}(T(n))$
	$\mathsf{TIEMPON}(f(n)) \subseteq \mathsf{TIEMPOD}(d^{f(n)})$
Determinista	$\mathsf{TIEMPOD}(f(n)) \subseteq \mathsf{ESPACIOD}(O(f(n)))$
	$\mathsf{ESPACIOD}(f(n)) \subseteq \mathsf{TIEMPOD}(c^{f(n)})$ si $f(n) \leq \log_2 n$

Tabla 6.6: Relaciones entre las clases de complejidad.

6.12. Problemas intratables

La clasificación de los problemas en tratables e intratables, puede realizarse desde dos perspectivas, a saber: temporal o espacial. En esta sección sólo consideraremos la primera de

ellas. Aunque todos los problemas computables sean, en teoría, tratables, desde un punto de vista práctico las cosas son diferentes. Así, en la práctica los problemas con límite temporal polinómico son considerados tratables y los que tienen límite temporal exponencial son considerados intratables.

Pasemos a realizar ciertas consideraciones importantes sobre el tema que nos ocupa, lo cual nos llevará a generar un conjunto de definiciones y teoremas importantes en el contexto de la complejidad algorítmica.

Existe en este contexto una importante clase de lenguajes que se pueden aceptar en modo determinista en tiempo polinómico, denominada la clase P.

Definición 6.39 (Clase P)**.**

$$\mathrm{P} = \bigcup_{i \geq 1} \mathsf{TIEMPOD}(n^i).$$

Otra clase importante de lenguajes en este contexto, son los que se pueden aceptar en modo no determinista en tiempo polinómico, denominada la clase NP.

Definición 6.40 (Clase NP)**.**

$$\mathrm{NP} = \bigcup_{i \geq 1} \mathsf{TIEMPON}(n^i).$$

Desde el punto de vista de los problemas de decisión, es decir, aquellos problemas en los cuales se espera una respuesta "SI" o una respuesta "NO", la diferencia entre entre la clase P y la clase NP puede ser vista como la diferencia entre encontrar la respuesta correcta (clase P) y probar que una posible respuesta es la correcta (clase NP). Intuitivamente es más fácil probar que una respuesta es la correcta que encontrar la respuesta correcta.

Aunque las clases P y NP son clases de lenguajes, hemos presentado una diferencia intuitiva entre ellas en términos de problemas de decisión. Cualquier problema de decisión puede ser visto como el problema de aceptar un cierto lenguaje construido bajo una codificación del problema de decisión en cuestión. Por esta razón hablaremos sin distinción de problemas de aceptación de lenguajes o de problemas de decisión.

Un problema importante que pertenece a la clase NP es el problema de la *satisfactibilidad*. Este problema jugará un papel muy importante en una clase de problemas que llamaremos NP-completos.

Ejemplo 6.41 (Problema de la satisfactibilidad)**.** Sea $X = \{x_1, \ldots, x_n\}$ un conjunto de variables booleanas y sea $t : X \to \{0, 1\}$ una asignación de valores de verdad para X; donde, $t(x) = 0$ significa que x es "falsa" y $t(x) = 1$ significa que x es "verdadera". Si $x \in X$ entonces x y $\overline{x}$ son literales sobre X. El literal $\overline{x}$ representa el opuesto de x, es decir $\overline{x}$ es "verdadero" bajo t si y sólo si x es falso bajo t. Una cláusula sobre X es una disyunción de literales sobre X. Una cláusula se dice satisfactible si existe una asignación de valores de verdad t para sus variables, tal que la cláusula sea "verdadera". Un conjunto de cláusulas

sobre X se dice satisfactible si existe una asignación de valores de verdad t que satisface simultáneamente todas las cláusulas del conjunto.

Por ejemplo, el conjunto formado por las tres cláusulas $C = \{x_1, x_2 \vee \overline{x}_3, x_1 \vee x_3\}$ es satisfactible, pues la asignación de valores de verdad $t(x_1) = t(x_2) = 1$ y $t(x_3) = 0$, hace verdaderas cada una de las cláusulas del conjunto.

Por otra parte, el conjunto de cláusulas $C = \{x_1, \overline{x_1} \vee \overline{x}_2, \overline{x_1} \vee x_2\}$ no es satisfactible, pues no existe ninguna asignación de valores de verdad t que haga las tres cláusulas verdaderas simultáneamente.

El problema de la satisfactibilidad (representado como SAT) consiste en decidir si un conjunto de cláusulas arbitrario es o no es satisfactible. El problema SAT $\in$ NP. Para expresar el problema de decisión SAT como el problema de aceptar un lenguaje L_{SAT}, realizaremos la siguiente codificación. Cada literal de la forma x_i se codifica como $\&i_b$ donde i_b es la representación en binario del número i. Cada literal de la forma $\overline{x}_i$ se codifica como $\neg\&i_b$ donde i_b es la representación en binario del número i. Entonces el alfabeto para L_{SAT} está definido por $\Sigma = \{\vee, \neg, \&, 0, 1\} \cup \{,\}$ y el lenguaje L_{SAT}, está definido por

$$L_{\text{SAT}} = \{\, \alpha \in \Sigma^* \mid \alpha \text{ representa un conjunto satisfactible de cláusulas} \,\}.$$

Dada $\alpha \in \Sigma^*$ necesitamos saber si, $\alpha \in L_{\text{SAT}}$, lo cual consiste en el problema de aceptar un lenguaje. Entonces afirmamos que $L_{\text{SAT}} \in$ NP.

Dado un conjunto de cláusulas codificado y una asignación de valores de verdad para sus variables, es posible evaluar su valor de verdad en tiempo polinómico. Entonces una máquina de Turing no determinista que seleccione una posible asignación de valores de verdad para las variables de un conjunto de cláusulas codificado, puede decidir en tiempo polinómico si este conjunto de cláusulas es satisfactible, tal como lo indica la figura 6.8.

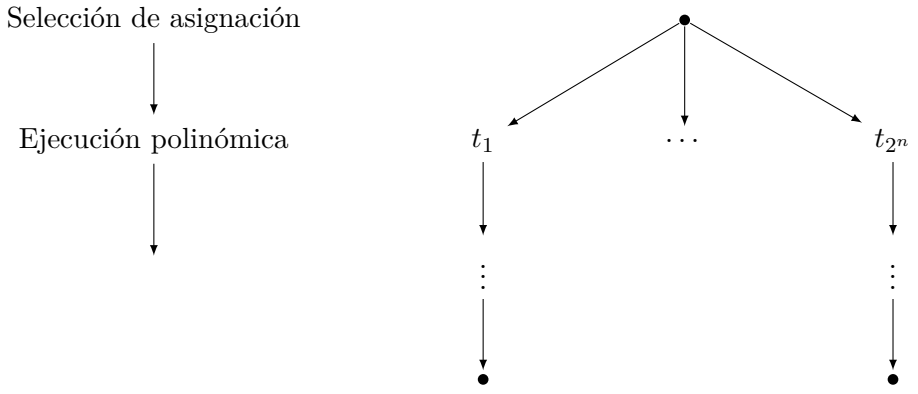


Figura 6.8: Árbol de computación para una MTN para el problema SAT.

Por el teorema 6.35, tenemos que $\mathsf{TIEMPOD}(n^k) \subseteq \mathsf{TIEMPON}(n^k)$, luego P $\subseteq$ NP. Uno de los problemas más importantes en el contexto de la complejidad algorítmica es decidir

si la inclusión P $\subseteq$ NP, es o no, una inclusión propia, lo cual es equivalente a decidir si P = NP o P $\neq$ NP.

P = NP equivale a decir que todo problema resuelto por una máquina de Turing no determinista con límite temporal polinómico puede ser resuelto por una máquina de Turing determinista con límite temporal polinómico. A partir del teorema 6.36 podemos afirmar que todo problema resuelto por una máquina de Turing no determinista con límite temporal polinómico, puede ser resuelto por una máquina de Turing determinista con límite temporal exponencial; sin embargo, este teorema no elimina la posibilidad de encontrar la máquina de Turing determinista con límite temporal polinómico.

Por otra lado, P $\neq$ NP equivale a decir que existe por lo menos un problema resuelto por una máquina de Turing no determinista con límite temporal polinómico, que no puede ser resuelto por una máquina de Turing determinista con límite temporal polinómico. Sin embargo, hasta el momento no se conoce ningún problema con estas características.

Un concepto importante para determinar si un lenguaje pertenece a la clase P o a la clase NP es el concepto de reducibilidad en tiempo polinómico.

Definición 6.42 (Función computable en tiempo polinómico)**.** Una función f se dice computable en tiempo polinómico si existe una máquina de Turing con límite temporal polinómico que la calcula.

Definición 6.43 (Reducción en tiempo polinómico)**.** Un lenguaje L_1 es reducible en tiempo polinómico a un lenguaje L_2, denotado por $L_1 <_p L_2$, si existe una función computable en tiempo polinómico f tal que, $f(\alpha) \in L_2$ si y sólo si $\alpha \in L_1$.

Teorema 6.44. *Si $L_2 \in$ P y $L_1 <_p L_2$, entonces $L_1 \in$ P.*

Demostración. Sea $L_2 \in$ P decidible en tiempo polinómico $p_2(n)$ y sea $L_1 <_p L_2$ en una reducción de tiempo polinomial f con límite temporal $p_1(n)$. Para decidir si $\alpha \in L_1$ se reduce α por medio de f y se decide entonces si $f(\alpha) \in L_2$. Entonces decidir si $\alpha \in L_1$ tiene un límite temporal $p_2(p_1(n))$ el cual es un límite temporal polinómico; por lo tanto $L_1 \in$ P. □

Teorema 6.45. *Si $L_2 \in$ NP y $L_1 <_p L_2$, entonces $L_1 \in$ NP.*

Demostración. Ejercicio 6.8. □

Además de permitirnos reconocer lenguajes de las clases P o NP, el concepto de reducción en tiempo polinómico nos permite hablar de unas clases importantes de lenguajes, denominados lenguajes C-completos y lenguajes C-difíciles.

Definición 6.46 (Lenguaje C-difícil)**.** Sea C una clase de lenguajes. Un lenguaje L se dice C-difícil si para todo $L' \in$ C, $L' <_p L$.

Definición 6.47 (Lenguaje C-completo)**.** Sea C una clase de lenguajes. Sea L un lenguaje C-difícil. Se dice que L es un lenguaje C-completo si $L \in$ C.

La clase de los problemas NP-completos es la clase de problemas NP que son equivalentes por medio de una reducción en tiempo polinómico. Por lo tanto, si se encuentra un solución determinista con límite temporal polinómico para cualquiera de los problemas en la clase NP-completos, entonces se ha encontrado una solución determinista con límite temporal polinómico para todos los problemas NP-completos.

El primer problema que se estableció como NP-completo fue el problema SAT.

Teorema 6.48 (Teorema de Cook)**.** $L_{\text{SAT}} \in$ NP-*completos.*

Demostración. (indicación)

Como SAT es el primer problema NP-completo, es necesario demostrar que cualquier problema NP es reducible en tiempo polinómico a SAT, es decir, es necesario demostrar que si $L \in$ NP entonces $L <_p L_{\text{SAT}}$. El conjunto de problemas en NP es bastante diverso y tienen como característica común el que para cada uno de ellos existe una máquina de Turing no determinista con límite temporal polinómico que lo resuelve. Por lo tanto, para demostrar que $L <_p L_{\text{SAT}}$, para cualquier $L \in$ NP es necesario construir una función f de tiempo polinómico tal que si $\alpha \in L$, entonces $f(\alpha) \in L_{\text{SAT}}$, y si $\alpha \notin L$, entonces $f(\alpha) \notin L_{\text{SAT}}$. Es decir, si $\alpha \in L$, entonces $f(\alpha)$ es una cláusula satisfactible y si $\alpha \notin L$, entonces $f(\alpha)$ es una cláusala no satisfactible. La construcción de f se realiza con base en la máquina de Turing no determinista que decide a L, por medio de una codificación de todas sus descripciones instantáneas posibles en un tiempo polinómico. □

Una vez establecido un primer problema NP-completo, para lograr demostrar que un nuevo problema es NP-completo, basta con demostrar que el nuevo problema es NP y que existe una reducción en tiempo polinómico de un problema NP-completo al nuevo problema. Esta propiedad la formalizamos mediante los siguientes teoremas.

Teorema 6.49. *Si* $L_1 \in$ NP-*completo y* $L_1 <_p L_2$*, entonces* $L_2 \in$ NP-*difícil.*

Demostración. Lo que necesitamos demostrar es que si $L_1 \in$ NP-completo y $L_1 <_p L_2$, entonces, para cualquier $L \in$ NP se tiene que $L <_p L_2$.

Sea $L \in$ NP, sea $L_1 \in$ NP-completo y $L_1 <_p L_2$. Como L_1 es NP-completo, existe una reducción de tiempo polinómico f_1 de L a L_1 tal que $\alpha \in L$ si y sólo si $f(\alpha) \in L_1$. Sea $p_1(n)$ el límite temporal polinómico de la reducción f_1. Como $L_1 <_p L_2$ existe una reducción de tiempo polinómico f_2 de L_1 a L_2 tal que $\alpha \in L_1$ si y sólo si $f(\alpha) \in L_2$. Sea $p_2(n)$ el límite temporal polinómico de la reducción f_2. La función $f_2(f_1(\alpha))$ es computable en tiempo polinómico $p_2(p_1(n))$ y tiene el comportamiento de una reducción en tiempo polinómico de L a L_2, es decir, $\alpha \in L$ si y sólo si $f_2(f_1(\alpha)) \in L_2$. Entonces para cualquier $L \in$ NP se tiene que $L <_p L_2$, de donde concluimos que $L_2 \in$ NP-difícil. □

Teorema 6.50. *Si* $L_1 \in$ NP-*completo,* $L_2 \in$ NP *y* $L_1 <_p L_2$*, entonces* $L_2 \in$ NP-*completo.*

Demostración. Ejercicio 6.9. □

Teorema 6.51. *Una variante del problema* SAT *es el problema* 3SAT*. En este caso cada cláusula debe contener tres literales. El problema* 3SAT $\in$ NP*-completos.*

Demostración. Ejercicio 6.10. □

Podemos hallar problemas NP-completos en diversos campos como la lógica, la teoría de grafos, el diseño de redes, la teoría de autómatas y lenguajes, entre otros. A continuación presentaremos un problema NP-completo de la teoría de grafos. Para lograr su presentación requerimos de algunas definiciones preliminares.

Definición 6.52 (Digrafo)**.** Un digrafo DG, es un modelo de un lenguaje $\mathsf{L} = \{P^2\}$, donde P^2 es un símbolo de predicado de aridez dos. En otros términos, un digrafo es una estructura matemática $\mathsf{DG} = (V, R)$, donde:

(i) V es un conjunto no vacío cuyos elementos llamamos vértices.

(ii) R es una relación binaria definida sobre V $(R \subset V \times V)$.

Cada digrafo $\mathsf{DG} = (V, R)$ puede ser representado por medio de un diagrama, en donde cada vértice $v \in V$ se representa por medio de un círculo etiquetado con el símbolo v, y cada $(v_i, v_j) \in R$ se representa por medio de un arco trazado del vértice v_i al vértice v_j.

Ejemplo 6.53. $\mathsf{DG}_2 = (\{0, 2, 5, 7, 8\}, \geq)$, representado por la figura 6.9 es un digrafo finito.

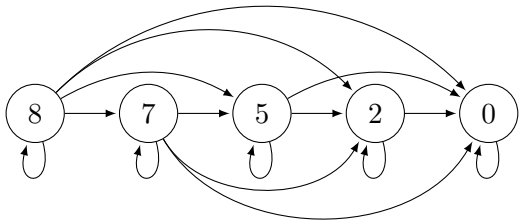


Figura 6.9: Ejemplo digrafo finito.

Aunque un grafo $\mathsf{G} = (V, R)$ también es un modelo de un lenguaje $\mathsf{L} = \{P\}$, donde P es un símbolo de predicado de aridez dos, la diferencia entre un digrafo y un grafo consiste en que este último debe satisfacer un axioma no exigido al primero.

Definición 6.54 (Grafo no dirigido)**.** Un grafo no dirigido o simplemente un grafo $\mathsf{G} = (V, R)$ es un modelo de un lenguaje $\mathsf{L} = \{P^2\}$, tal que G satisface el axioma de simetría para la relación R, es decir,

$$\mathsf{G} \models \forall x \forall y((x, y) \in R \Longrightarrow (y, x) \in R).$$

Cada grafo $\mathsf{G} = (V, R)$ puede ser representado por medio de un diagrama en donde cada vértice $v \in V$ se representa por medio de un circulo etiquetado con el símbolo v y cada par de elementos $(v_i, v_j), (v_j, v_i) \in R$ se representan por medio de una arista del vértice v_i al vértice v_j.

Ejemplo 6.55. El siguiente objeto matemático es un grafo finito.
$\mathsf{G}_2 = (\{A, B, C, D\}, R)$, donde,
$R = \{(A, A), (A, B), (B, A), (C, D), (D, C), (A, C), (C, A), (B, D), (D, B)\}$.
El grafo G_2 es representado por la figura 6.10.

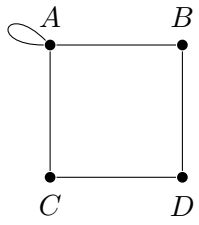


Figura 6.10: Ejemplo grafo no dirigido finito.

Nuestro siguiente problema NP-completo justamente es el problema del isomorfismo de grafos. Sabemos que la relación de isomorfismo es un relación entre estructuras cuya función esencial es reconocer y clasificar las que son estructuralmente idénticas. Igualmente sabemos que toda propiedad o fórmula que satisfaga una estructura debe satisfacer la otra. Tales propiedades las denominamos invariantes.

Definición 6.56 (Isomorfismo de grafos)**.** Sean $\mathsf{G} = (V, R)$ y $\mathsf{G}' = (V', R')$ dos grafos (digrafos). Decimos que G es isomorfo a G', si y sólo si existe una función biyectiva $\varphi : V \to V'$, tal que:

$$(v_1, v_2) \in R \Rightarrow (\varphi(v_1), \varphi(v_2)) \in R'; \quad \text{para todo } v_1, v_2 \in V.$$

La función $\varphi : V \to V'$ se denomina isomorfismo de grafos. Escribimos $\varphi : \mathsf{G} \simeq \mathsf{G}'$ para indicar que G es isomorfo a G'.

Ejemplo 6.57. Los grafos $\mathsf{G}_1 = (V_1, R_1)$ y $\mathsf{G}_2 = (V_2, R_2)$, representados por las figuras 6.11 y 6.12, son isomorfos.

$\mathsf{G}_1 \simeq \mathsf{G}_2$, ya que, $\overline{\overline{V_1}} = \overline{\overline{V_2}} = 4$; además podemos definir la función $\varphi : V_1 \to V_2$ tal que $\varphi(1) = a$, $\varphi(2) = b$, $\varphi(3) = c$ y $\varphi(4) = d$. Entonces, $(1, 2) \in R_1$ y $(\varphi(1), \varphi(2)) \in R_2$; $(1, 4) \in R_1$ y $(\varphi(1), \varphi(4)) \in R_2$; igualmente para las parejas restantes.

Un aspecto importante en el trabajo con grafos, es el concerniente a la determinación de grafos que no son isomorfos. Aunque existen algoritmos que pueden determinar en buena

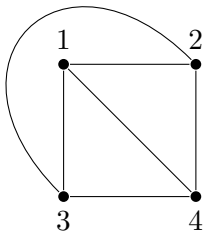


Figura 6.11: G_1 isomorfo al grafo de la figura 6.12.

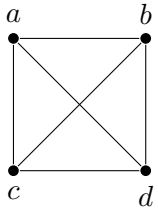


Figura 6.12: G_2 isomorfo al grafo de la figura 6.11.

medida si dos pares de grafos son isomorfos, una forma de determinar que no lo son consiste en buscar una propiedad que no se preserve.

Ejemplo 6.58. Los grafos G_1 y G_2, representados por las figuras 6.13 y 6.14 respectivamente, no son isomorfos puesto que G_1 tiene siete lados y G_2 tiene ocho lados.

Ejemplo 6.59. Los grafos G_a y G_b, representados por las figuras 6.15 y 6.16 respectivamente, no son isomorfos.

El vértice b en G_a es adyacente a dos vértices, luego le podríamos asociar el vértice 6 de G_b. Igualmente el vértice q de G_a se podría corresponder con el vértice 1 de G_b. En G_a no existen más vértices de grado dos, y en G_b existe todavía el vértice 3. Luego no podríamos construir una biyección que preserve las adyacencias.

Teorema 6.60. *Sean* $\mathsf{G} = (V, R)$ *y* $\mathsf{G}' = (V', R')$ *dos grafos (digrafos). Determinar si existe o no un isomorfismo entre los grafos* G *y* G' *es un problema* NP*-completo.*

6.13. Ejercicios

Ejercicio 6.1. Construir una máquina de Turing 2-cintas tal como es indicado en el ejemplo 6.5.

Ejercicio 6.2. Construir una máquina de Turing tal como es indicado en el ejemplo 6.15.

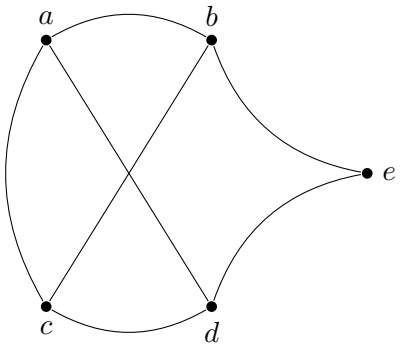


Figura 6.13: G_1 no isomorfo al grafo de la figura 6.14.

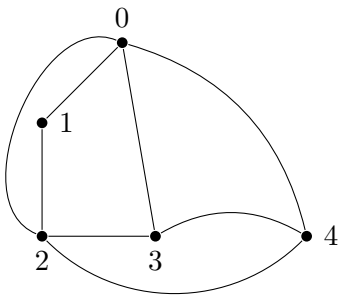


Figura 6.14: G_2 no isomorfo al grafo de la figura 6.13.

Ejercicio 6.3. Construir una máquina de Turing tal como es indicado en el ejemplo ejemplo 6.20.

Ejercicio 6.4. Demostrar el teorema 6.31.

Ejercicio 6.5. Demostrar el teorema 6.33.

Ejercicio 6.6. Demostrar que las funciones $n^2, 2^n, n!$ son funciones espacialmente construibles de manera completa

Ejercicio 6.7. Demostrar el teorema 6.34.

Ejercicio 6.8. Demostrar el teorema 6.45.

Ejercicio 6.9. Demostrar el teorema 6.50.

Ejercicio 6.10. Presentar una indicación para demostrar el teorema 6.51.

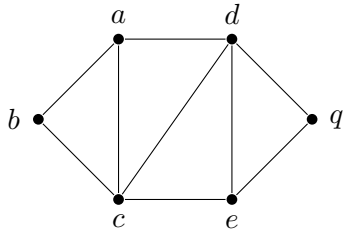


Figura 6.15: G_a no isomorfo al grafo de la figura 6.16.

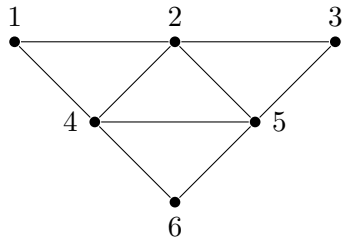


Figura 6.16: G_b no isomorfo al grafo de la figura 6.15.

6.14. Notas bibliográficas

Las definiciones de máquinas de Turing k-cintas y máquinas de Turing $(k, 1)$-cintas así como las definiciones de clase de complejidad temporal y espacial (determinista y no determinista) para un lenguaje fueron tomadas de [Hopcroft y Ullman 1997; Kelley 1995; Papadimitriou 1995]. Las definiciones de lenguaje recursivo y lenguaje recursivamente enumerable fueron tomadas de [Papadimitriou 1995]. Los ejemplos 6.5, 6.9 y 6.20 son presentados por [Hopcroft y Ullman 1997; Papadimitriou 1995]. Las convenciones $T(n) \stackrel{\text{def}}{=} \mathsf{máx}(n+1, \lceil T(n) \rceil)$ y $S(n) \stackrel{\text{def}}{=} \mathsf{máx}(1, \lceil S(n) \rceil)$ así como los ejemplos 6.9 y 6.18 son presentados por [Hopcroft y Ullman 1997]. La notación asintótica es presentada por [Cormen, Leiserson, Rivest y Stein 2022]. Los teoremas 6.13 y 6.14 y el ejemplo 6.15 fueron tomados de [Papadimitriou 1995]. El teorema 6.16 es presentado por [Hopcroft y Ullman 1997; Kelley 1995; Papadimitriou 1995]. El teorema 6.21 es presentado por [Hopcroft y Ullman 1997; Papadimitriou 1995]. El teorema 6.22 es presentado por [Kelley 1995]. Una versión un poco diferente del teorema 6.23 la ofrecen Hopcroft y Ullman [1997] y Kelley [1995]. Las definiciones de máquinas de Turing no deterministas y las clases de complejidad espaciales y temporales no deterministas se tomaron de [Hopcroft y Ullman 1997; Papadimitriou 1995]. El teorema 6.28 es presentado por [Kelley 1995]. Las nociones de función espacialmente construible y función espacialmente construible de manera completa son dadas por [Hopcroft y Ullman 1997]. El teorema 6.31 se tomó de [Hopcroft y Ullman 1997; Papadimitriou 1995]. El teorema 6.33 aparece en [Hopcroft y Ullman 1997]. El teorema 6.34 lo ofrecen [Hopcroft

y Ullman 1997; Kelley 1995; Papadimitriou 1995]. El teorema 6.35 es presentado por [Kelley 1995; Papadimitriou 1995]. Los teoremas 6.36, 6.37 y 6.38 aparecen en [Hopcroft y Ullman 1997; Kelley 1995]. Los elementos de la sección 6.12 son dados por [Garey y Johnson 1979; Hopcroft y Ullman 1997; Kelley 1995; Papadimitriou 1995]. El problema $P \stackrel{?}{=} NP$ fue propuesto por Smale [1998] (medallla *Field* en matemáticas) como uno de los problemas matemáticos más importantes para el próximo siglo; en la misma dirección, el *Clay Mathematics Institute* ofreció en el año 2.000 un premio bastante cuantioso en efectivo por la solución de siete problemas matemáticos, entre ellos el problema $P \stackrel{?}{=} NP$ [Clay Mathematics Institute 2000]. Algunos textos en teoría de grafos son Bogart [1996], Crespo [1983], Grassman y Tremblay [1996], Johnsonbaugh [1988] y Kolman y Busby [1984]. Los isomorfismos entre grafos son presentados por [Bogart 1996; Deo 1974; Grimaldi 1997]. El teorema 6.60 lo ofrece [Garey y Johnson 1979]. El libro de Garey y Johnson [1979] ofrece más de 300 problemas NP-completos. El último problema NP-completo de que los autores tienen conocimiento, es la solución general al juego "busca minas" del sistema operativo *Windows* [Stewart 2000].

Apéndice A

Función de Ackermann

El valor de $\mathsf{ack}(2,3)$, donde ack es la función de Ackermann definida en la sección 2.6 está dado por:

$$
\begin{aligned}
\mathsf{ack}(2,3) &= \mathsf{ack}(1,\mathsf{ack}(2,2)) \\
&= \mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(2,1))) \\
&= \mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(2,0)))) \\
&= \mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(1,1)))) \\
&= \mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(1,0))))) \\
&= \mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(0,1))))) \\
&= \mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(0,2)))) \\
&= \mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(1,3))) \\
&= \mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(1,2)))) \\
&= \mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(1,1))))) \\
&= \mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(1,0)))))) \\
&= \mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,1)))))) \\
&= \mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,2))))) \\
&= \mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(0,3)))) \\
&= \mathsf{ack}(1,\mathsf{ack}(1,\mathsf{ack}(0,4))) \\
&= \mathsf{ack}(1,\mathsf{ack}(1,5)) \\
&= \mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(1,4))) \\
&= \mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(1,3)))) \\
&= \mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(1,2))))) \\
&= \mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(1,1)))))) \\
&= \mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(1,0))))))) \\
&= \mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,1))))))) \\
&= \mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,2)))))) \\
&= \mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,3))))) \\
&= \mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,4)))) \\
&= \mathsf{ack}(1,\mathsf{ack}(0,\mathsf{ack}(0,5))) \\
&= \mathsf{ack}(1,\mathsf{ack}(0,6)) \\
&= \mathsf{ack}(1,7) \\
&= \mathsf{ack}(0,\mathsf{ack}(1,6)) \\
&= \mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(1,5))) \\
&= \mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(1,4)))) \\
&= \mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(1,3))))) \\
&= \mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(1,2)))))) \\
&= \mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(1,1))))))) \\
&= \mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(1,0)))))))) \\
&= \mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,1)))))))) \\
&= \mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,2))))))) \\
&= \mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,3)))))) \\
&= \mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,4))))) \\
&= \mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,5)))) \\
&= \mathsf{ack}(0,\mathsf{ack}(0,\mathsf{ack}(0,6))) \\
&= \mathsf{ack}(0,\mathsf{ack}(0,7)) \\
&= \mathsf{ack}(0,8) \\
&= 9
\end{aligned}
$$

Apéndice B

Ackermann

La idea es demostrar que la clase de la funciones recursivas contiene propiamente a la clase de las funciones primitivas recursivas.

Función de Ackermann:

$$\mathsf{ack}(0, y) = y', \tag{B.1}$$
$$\mathsf{ack}(x', 0) = \mathsf{ack}(x, 1), \tag{B.2}$$
$$\mathsf{ack}(x', y') = \mathsf{ack}(x, \mathsf{ack}(x', y)), \tag{B.3}$$

donde $x' = x + 1$.

Inicialmente se presentaran algunos teoremas auxiliares, antes de demostrar que la función de Ackermann no es primitiva recursiva.

Teorema B.1. $y < \mathsf{ack}(x, y)$.

Demostración. La demostración se hará por doble inducción sobre las variables x y y.

1. Para $x = 0$, es necesario demostrar que $y < \mathsf{ack}(0, y)$. Esto es verdadero, puesto que $\mathsf{ack}(0, y) = y'$ (por B.1).
2. Primera hipótesis inductiva: $y < \mathsf{ack}(x, y)$.
3. Para x', es necesario demostrar que $y < \mathsf{ack}(x', y)$.
   a) Para $y = 0$, es necesario demostrar que $0 < \mathsf{ack}(x', 0)$.
      1) $0 < 1$ y $1 < \mathsf{ack}(x, 1)$ (por primera hipótesis inductiva).
      2) $\mathsf{ack}(x, 1) = \mathsf{ack}(x', 0)$ (por B.2).
      3) Por lo tanto, $0 < \mathsf{ack}(x', 0)$.
   b) Segunda hipótesis inductiva: $y < \mathsf{ack}(x', y)$.

c) Para y', es necesario demostrar que $y' < \mathsf{ack}(x', y')$.

1) $\mathsf{ack}(x', y) < \mathsf{ack}(x, \mathsf{ack}(x', y))$ (por primera hipótesis inductiva, haciendo $y = \mathsf{ack}(x', y)$).
2) $\mathsf{ack}(x', y) < \mathsf{ack}(x', y')$, puesto que $\mathsf{ack}(x, \mathsf{ack}(x', y)) = \mathsf{ack}(x', y')$ (por B.3).
3) Si $y < y'$ y $y < \mathsf{ack}(x', y)$ (segunda hipótesis inductiva) entonces $y' \leq \mathsf{ack}(x', y)$.
4) Por lo tanto, $y' \leq \mathsf{ack}(x', y')$, puesto que $\mathsf{ack}(x', y) < \mathsf{ack}(x', y')$. □

Teorema B.2. $\mathsf{ack}(x, y) < \mathsf{ack}(x, y')$ *(Monotonía en el segundo argumento).*

Demostración. La demostración se hará por inducción sobre la variable x.

1. Para $x = 0$, es necesario demostrar que $\mathsf{ack}(0, y) < \mathsf{ack}(0, y')$. Esto es verdadero, puesto que $\mathsf{ack}(0, y) = y'$ y $\mathsf{ack}(0, y') = y''$ (por B.1).

2. Hipótesis inductiva: $\mathsf{ack}(x, y) < \mathsf{ack}(x, y')$.

3. Para x', es necesario demostrar que $\mathsf{ack}(x', y) < \mathsf{ack}(x', y')$.

   a) $\mathsf{ack}(x', y) < \mathsf{ack}(x, \mathsf{ack}(x'y))$ (por teorema B.1, haciendo $y = \mathsf{ack}(x', y)$).
   b) $\mathsf{ack}(x, \mathsf{ack}(x', y)) = \mathsf{ack}(x', y')$ (por B.3).
   c) Por lo tanto, $\mathsf{ack}(x', y) < \mathsf{ack}(x', y')$. □

Teorema B.3. $\mathsf{ack}(x, y') \leq \mathsf{ack}(x', y)$.

Demostración. La demostración se hará por inducción sobre la variable y.

1. Para $y = 0$, es necesario demostrar que $\mathsf{ack}(x, 1) \leq \mathsf{ack}(x', 0)$. Esto es verdadero, puesto que $\mathsf{ack}(x', 0) = \mathsf{ack}(x, 1)$ (por B.2).

2. Hipótesis inductiva: $\mathsf{ack}(x, y') \leq \mathsf{ack}(x', y)$.

3. Para y', es necesario demostrar que $\mathsf{ack}(x, y'') \leq \mathsf{ack}(x', y')$.

   a) $y' < y''$ y $y' < \mathsf{ack}(x, y')$ (por teorema B.1), entonces $y'' \leq \mathsf{ack}(x, y')$.
   b) $\mathsf{ack}(x, y') \leq \mathsf{ack}(x', y)$ (por hipóteis inductiva), entonces $y'' \leq \mathsf{ack}(x', y)$.
   c) $\mathsf{ack}(x, y'') \leq \mathsf{ack}(x, \mathsf{ack}(x', y)$ puesto que $y'' \leq \mathsf{ack}(x', y)$ (por teorema B.2) y $\mathsf{ack}(x, \mathsf{ack}(x', y) = \mathsf{ack}(x', y')$ (por B.3).
   d) Por lo tanto, $\mathsf{ack}(x, y'') \leq \mathsf{ack}(x', y')$. □

Teorema B.4. $\mathsf{ack}(x, y) < \mathsf{ack}(x', y)$ *(Monotonía en el primer argumento).*

Demostración.

1. $\mathsf{ack}(x, y) < \mathsf{ack}(x, y')$ (por teorema B.2).
2. $\mathsf{ack}(x, y') \leq \mathsf{ack}(x', y)$ (por teorema B.3).
3. Por lo tanto, $\mathsf{ack}(x, y) < \mathsf{ack}(x', y)$. □

Teorema B.5. $\mathsf{ack}(1, y) = y + 2$.

Demostración. La demostración se hará por inducción sobre la variable y.

1. Para $y = 0$, es necesario demostrar que $\mathsf{ack}(1, 0) = 2$. Esto es verdadero, puesto que $\mathsf{ack}(1, 0) = \mathsf{ack}(0, 1) = 2$ (por B.2).
2. Hipótesis inductiva: $\mathsf{ack}(1, y) = y + 2$.
3. Para y', es necesario demostrar que $\mathsf{ack}(1, y') = y' + 2$.

$$\begin{aligned} \mathsf{ack}(1, y') &= \mathsf{ack}(0, \mathsf{ack}(1, y)) && \text{(por B.3)} \\ &= \mathsf{ack}(0, y + 2) && \text{(por hipótesis inductiva)} \\ &= y + 3 && \text{(por B.1)} \\ &= y' + 2 \end{aligned}$$

□

Teorema B.6. $\mathsf{ack}(2, y) = 2y + 3$.

Demostración. Similar a la demostración del teorema anterior. □

Teorema B.7. *Para constantes arbitrarias $c_1, c_2, \ldots, c_n$, existe una constante c tal que*

$$\sum_{i=1}^{n} \mathsf{ack}(c_i, x) \leq \mathsf{ack}(c, x).$$

Demostración. Sólo es necesario realizar la demostración para $n = 2$, puesto que para $n > 2$ la generalización es inmediata. Sea $d = \mathsf{máx}(c_1, c_2)$ y sea $c = d + 4$ entonces:

$$\begin{aligned} \mathsf{ack}(c_1, x) + \mathsf{ack}(c_2, x) &\leq \mathsf{ack}(d, x) + \mathsf{ack}(d, x) && \text{(por teorema B.4)} \\ &< 2\mathsf{ack}(d, x) + 3 \\ &= \mathsf{ack}(2, \mathsf{ack}(d, x)) && \text{(por teorema B.6)} \\ &< \mathsf{ack}(d + 2, \mathsf{ack}(d, x)) && \text{(por teorema B.4)} \\ &< \mathsf{ack}(d + 2, \mathsf{ack}(d + 3, x)) && \text{(por teoremas B.4 y B.2)} \\ &= \mathsf{ack}(d + 3, x') && \text{(por B.3)} \\ &\leq \mathsf{ack}(d + 4, x) && \text{(por teorema B.3)} \\ &= \mathsf{ack}(c, x). \end{aligned}$$

□

El siguiente teorema afirma que la función de Ackermann crece más rápido que cualquier función primitiva recursiva.

Teorema B.8. *Sea $g(x_1, \ldots, x_n)$ una FPR, entonces existe c tal que $g(x_1, \ldots, x_n) < \mathsf{ack}(c, x_1 + \cdots + x_n)$.*

Demostración. Es necesario demostrar que las funciones iniciales satisfacen el teorema y que los esquemas de composición de funciones y recurrencia primitiva, preservan la propiedad enunciada por el teorema.

1. La función cero satisface el teorema, puesto que $\mathsf{z}(x) = 0 < \mathsf{ack}(0, x) = x'$.

2. La función sucesor satisface el teorema, puesto que $\mathsf{s}(x) = \mathsf{ack}(0, x) < \mathsf{ack}(1, x)$ (por teorema B.4).

3. Cada una de las funciones k-ésima proyección satisface el teorema, puesto que

$$\mathsf{I}_k^n(x_1, \ldots, x_n) = x_i < \mathsf{ack}(0, x_1 + \cdots + x_n) = (x_1 + \cdots + x_n)'.$$

4. Para el esquema de composición, es necesario demostrar que si

$$\begin{aligned} g(y_1, \ldots, y_m) &< \mathsf{ack}(c, y_1 + \cdots + y_m), \\ h_1(x_1, \ldots, x_n) &< \mathsf{ack}(c_1, x_1 + \cdots + x_n), \\ \vdots \qquad &< \qquad \vdots \\ h_m(x_1, \ldots, x_n) &< \mathsf{ack}(c_m, x_1 + \cdots + x_n), \end{aligned}$$

   entonces,

$$\begin{aligned} f(x_1, \ldots, x_n) &= g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n)) \\ &< \mathsf{ack}(d, x_1 + \cdots + x_n). \end{aligned}$$

   Demostración.

$$\begin{aligned} & f(x_1, \ldots, x_n) \\ & = g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n)) \\ & < \mathsf{ack}(c, h_1(x_1, \ldots, x_n) + \cdots + h_m(x_1, \ldots, x_n)) && \text{(por hipótesis)} \\ & < \mathsf{ack}(c, \mathsf{ack}(c_1, x_1 + \cdots + x_n) + \cdots + \mathsf{ack}(c_n, x_1 + \cdots + x_n)) && \text{(por teorema B.2)} \\ & \leq \mathsf{ack}(c, \mathsf{ack}(c^*, x_1 + \cdots + x_n)) && \text{(por teorema B.7)} \\ & \leq \mathsf{ack}(c + c^*, \mathsf{ack}(c + c^* + 1, x_1 + \cdots + x_n)) && \text{(por teoremas B.2 y B.4)} \\ & = \mathsf{ack}(c + c^* + 1, x_1 + \cdots + x_n + 1) && \text{(por B.3)} \\ & \leq \mathsf{ack}(c + c^* + 2, x_1 + \cdots + x_n) && \text{(por teorema B.3)} \end{aligned}$$

5. Para el esquema de recurrencia primitiva, es necesario demostrar que si

$$g(x_1,\ldots,x_n) < \mathsf{ack}(c_1, x_1+\cdots+x_n),$$
$$h(x_1,\ldots,x_n,y,z) < \mathsf{ack}(c_2, x_1+\cdots+x_n+y+z),$$

entonces, la función $f(x_1,\ldots,x_n,y)$ definida por:

$$f(x_1,x_2,\ldots,x_n,0) = g(x_1,x_2,\ldots,x_n),$$
$$f(x_1,x_2,\ldots,x_n,k+1) = h(x_1,x_2,\ldots,x_n,k,f(x_1,x_2,\ldots,x_n,k)),$$

satisface el teorema, es decir, $f(x_1,\ldots,x_n,y) < \mathsf{ack}(c, x_1+\cdots+x_n+y)$.

Demostración. En lugar de demostrar que $f(x_1,\ldots,x_n,y) < \mathsf{ack}(c, x_1+\cdots+x_n+y)$, vamos a demostrar que

$$f(x_1,\ldots,x_n,y)+x_1+\cdots+x_n+y < \mathsf{ack}(c, x_1+\cdots+x_n+y),$$

a) Inicialmente se demuestra que $g(x_1,\ldots,x_n)+x_1+\cdots+x_n+ < \mathsf{ack}(c_1, x_1+\cdots+x_n)$.

$$\begin{aligned}
&g(x_1,\ldots,x_n)+x_1+\cdots+x_n+y) = \\
&g(x_1,\ldots,x_n)+\mathsf{I}_1^n(x_1,\ldots,x_n)+\cdots+\mathsf{I}_n^n(x_1,\ldots,x_n) \\
&< \mathsf{ack}(c_1, x_1+\cdots+x_n)+\mathsf{I}_1^n(x_1,\ldots,x_n)+\cdots+\mathsf{I}_n^n(x_1,\ldots,x_n) \\
&< \mathsf{ack}(c_1, x_1+\cdots+x_n)+\mathsf{ack}(0, x_1+\cdots+x_n)+\cdots+\mathsf{ack}(0, x_1+\cdots+x_n) \\
&< \mathsf{ack}(c_1^*, x_1+\cdots+x_n) \qquad \text{(por teorema B.7)}
\end{aligned}$$

b) De forma similar a la anterior se demuestra que $h(x_1,\ldots,x_n,y,z)+x_1+\cdots+x_n+y+z < \mathsf{ack}(c_2^*, x_1+\cdots+x_n+y+z)$.

c) Ahora de demuestra que $f(x_1,\ldots,x_n,y)+x_1+\cdots+x_n+y < \mathsf{ack}(c, x_1+\cdots+x_n+y)$ por inducción sobre la variable y.
Sea $c = \mathsf{máx}(c_1^*, c_2^*)+1$, entonces:

1) Para $y=0$, es necesario demostrar que $f(x_1,\ldots,x_n,0)+x_1+\cdots+x_n+ < \mathsf{ack}(c, x_1+\cdots+x_n)$.

$$\begin{aligned}
f(x_1,\ldots,x_n,0)+x_1+\cdots+x_n &= g(x_1,\ldots,x_n)+x_1+\cdots+x_n \\
&< \mathsf{ack}(c_1^*, x_1+\cdots+x_n) \\
&< \mathsf{ack}(c, x_1+\cdots+x_n)
\end{aligned}$$

2) Hipótesis inductiva: $f(x_1, \ldots, x_n, y) + x_1 + \cdots + x_n + y < \mathsf{ack}(c, x_1 + \cdots + x_n + y)$.
3) Para y', es necesario demostrar que $f(x_1, \ldots, x_n, y') + x_1 + \cdots + x_n + y' < \mathsf{ack}(c, x_1 + \cdots + x_n + y')$.

$$
\begin{aligned}
& f(x_1, \ldots, x_n, y') + x_1 + \cdots + x_n + y' = \\
& h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)) + x_1 + \cdots + x_n + y' \\
& < \mathsf{ack}(c_2^*, x_1 + \cdots + x_n + y + f(x_1, \ldots, x_n, y)) + x_1 + \cdots + x_n + y' \\
& \leq \mathsf{ack}(c_2^*, x_1 + \cdots + x_n + y + f(x_1, \ldots, x_n, y)) + 1 \\
& < \mathsf{ack}(c_2^*, \mathsf{ack}(c, x_1 + \cdots + x_n + y)) + 1 \quad \text{(por hip. indu. y teorema B.3)} \\
& \leq \mathsf{ack}(c - 1, \mathsf{ack}(c, x_1 + \cdots + x_n + y)) + 1 \quad \text{(por definición de } c\text{)} \\
& = \mathsf{ack}(c, x_1 + \cdots + x_n + y')) + 1 \quad \text{(por B.3)} \\
& < \mathsf{ack}(c, x_1 + \cdots + x_n + y')) \quad \text{(porque el signo } < \text{ aparece dos veces)}
\end{aligned}
$$

□

Finalmente demostramos que la función de Ackerman no es una función primitiva recursiva.

Teorema B.9. *La función de Ackermann no es FPR.*

Demostración.

1. Sea la función de Ackermann $\mathsf{ack}(x, y)$ una FPR.
2. La función $f(x) = \mathsf{ack}(x, x)$ es una FPR.
3. Existe c tal que $f(x) < \mathsf{ack}(c, x)$ (por teorema B.8).
4. Para $x = c$, tenemos que $f(c) < \mathsf{ack}(c, c) = f(c)$. Contradicción. □

Bibliografía

Alfred V. Aho, Ravi Sethi y Jeffrey D. Ullman (1990). *Compiladores: Principios, Técnicas y Herramientas*. Addison-Wesley Iberoamericana (vid. pág. 130).

Kenneth P. Bogart (1996). *Matemáticas Discretas*. Editorial Limusa S.A. de C.V. (vid. pág. 207).

George S. Boolos, John P. Burges y Richard C. Jeffrey (2007). *Computability and Logic*. 5.ª ed. Cambridge University Press (vid. pág. 95).

George S. Boolos y Richard C. Jeffrey (1989). *Computability and Logic*. 3.ª ed. Cambridge University Press (vid. pág. 95).

Xavier Caicedo (1990). *Elementos de Lógica y Calculabilidad*. 2.ª ed. Una empresa docente. Universidad de los Andes (vid. pág. 95).

Clay Mathematics Institute (2000). *Millennium Prize Problems*. URL: www.claymath.org/prize_problems/ (visitado 10-07-2000) (vid. pág. 207).

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein (2022). *Introduction to Algorithms*. 4.ª ed. MIT Press (vid. pág. 206).

Decoroso Crespo (1983). *Informática Teórica. Primera Parte*. Departamento de Publicaciones de la Facultad de Informática. Universidad Politécnica de Madrid (vid. págs. 130, 161, 207).

Martin Davis (1982). *Computability and Unsolvability*. Dover Publications (vid. págs. 55, 56, 95).

Narsing Deo (1974). *Graph Theory with Application to Engineering and Computer Science*. Prentice-Hall (vid. pág. 207).

Günter Dotzel (1991). «Letter to the Editor». En: *The ModulaTor* 11. URL: http://www.modulaware.com/mdlt17.htm (visitado 04-02-2014) (vid. pág. 95).

Gregorio Fernández Fernández y Fernando Sáez Vacas (1987). *Fundamentos de Informática*. Alianza Editorial (vid. pág. 161).

Michael R. Garey y David S. Johnson (1979). *Computers and Intractability. A Guide to the Theory of NP-completeness*. W. H. Freeman y Company (vid. pág. 207).

Pedro Gómez y Cristina Gómez (1992). *Sistemas Formales, Informalmente*. Segunda versión. Una empresa docente. Universidad de los Andes (vid. pág. 130).

Karl Grassman y Jean-Paul Tremblay (1996). *Matemáticas Discretas y Lógica. Una Perspectiva desde la Ciencia de la Computación*. Trad. por Rafael García-Bermejo,

María Luisa Díez Platas y Vivian de los Ángeles Fernández Vásquez. Prentice Hall (vid. pág. 207).

Ralph P. Grimaldi (1997). *Matemáticas Discreta y Combinatoria*. 3.ª ed. Addison-Wesley Iberoamericana (vid. págs. 130, 207).

Hans Hermes (1969). *Enumerability · Decidability · Computability*. Second revised edition. Translated G. T. Hermann and O. Plassmann. Springer-Verlag (vid. pág. 95).

John Hopcroft (mayo de 1984). «Turing Machines». En: *Scientific American* 250, págs. 86-98. DOI: 10.1038/scientificamerican0584-86 (vid. pág. 55).

John Hopcroft y Jefferey D. Ullman (1997). *Introducción a la Teoría de Automátas, Lenguajes y Computación*. Compañía Editorial Continental, S.A. (CECSA) (vid. págs. 179, 206, 207).

Richard Johnsonbaugh (1988). *Matemáticas Discretas*. Grupo Editorial Iberoamérica, S.A. (vid. págs. 130, 161, 207).

Dean Kelley (1995). *Teoría de Autómatas y Lenguajes Formales*. Prentice-Hall (UK) Ltd. (vid. págs. 130, 161, 179, 206, 207).

Stephen C. Kleene (1974). *Introducción a la Metamatemática*. Trad. por Manuel Garrido. Vol. 42. Estructura y Función. Editorial Tecnos (vid. págs. 55, 95).

Bernard Kolman y Robert Busby (1984). *Discrete Mathematical Structures for Computer Sciencs*. Prentice-Hall (vid. págs. 130, 161, 207).

Jean Ladrière (1969). *Limitaciones Internas de los Formalismos*. Editorial Tecnos (vid. pág. 130).

León López López y Rául Gómez Marín (1993). *Matemáticas Básicas para la Informática*. Vol. I. Universidad EAFIT (vid. pág. 130).

Heiner Marxen y Jürgen Buntrock (1990). «Attacking the Busy Beaver 5». En: *Bulletin of the EATCS* 000.40, págs. 247-251. URL: http://www.drb.insel.de/~heiner/BB/bb-mabu90.ps (visitado 04-02-2014) (vid. pág. 56).

Elliott Mendelson (2015). *Introduction to Mathematical Logic*. 6.ª ed. CRC Press (vid. págs. 55, 95).

Marvin L. Minsky (1967). *Computation: Finite and Infinite Machines*. Prentice-Hall (vid. págs. 55, 95, 161).

Christos Papadimitriou (1995). *Computational Complexity*. Reprinted with corrections. Addison-Wesley (vid. págs. 206, 207).

Roger Penrose (1991). *La Nueva Mente del Emperador*. Colección: Libro de mano No. 38. Grijalbo Mondadori (vid. págs. 55, 56).

——— (1996). *Las Sombras de la Mente*. Colección: Drakontos. Crítica (vid. pág. 55).

Jefferey Shallit (1998). *The Busy Beaver Problem*. URL: http://grail.cba.csuohio.edu/~somos/beaver.ps (visitado 02-06-2000) (vid. pág. 56).

Andrés Sicard (1997). «Máquina Universal de Turing: Algunas Indicaciones para su Construcción». En: *Revista Universidad EAFIT* 33.108, págs. 61-106 (vid. pág. 56).

Andrés Sicard Ramírez (1996). «Máquinas de Turing». En: *Revista Universidad EAFIT* 103, págs. 29-45 (vid. págs. 55, 56).

Wilfred Sieg (1997). «Step by Recursive Step: Church's Analysis of Effective Calculability». En: *The Bulletin of Symbolic Logic* 3.2, págs. 154-180. DOI: 10.2307/421012 (vid. págs. 55, 95).

Steve Smale (1998). «Mathematical Problems for the Next Century». En: *The Mathematical Intelligencer* 20.2, págs. 7-13 (vid. pág. 207).

Robert I. Soare (1996). «Computability and Recursion». En: *The Bulletin of Symbolic Logic* 2.3, págs. 284-321. DOI: 10.2307/420992 (vid. págs. 55, 95).

Ian Stewart (2000). *Million-Dollar Minesweeper*. URL: http://www.claymath.org/prize_problems/million-dollar-minesweeper.htm (visitado 07-12-2000) (vid. pág. 207).

Alan M. Turing (1936-1937). «On Computable Numbers, with an Application to the Entscheidungsproblem». En: *Proceeding of the London Mathematical Society* s2-42, págs. 230-265. DOI: 10.1112/plms/s2-42.1.230 (vid. págs. 55, 56).

Ann Yasuhara (1971). *Recursive Function Theory and Logic*. Academic Press (vid. pág. 95).

Índice alfabético