Teorías de Autómatas
y Lenguajes Formales

Colección manuales uex - 55
(E.E.E.S.)

Elena
Jurado Málaga

55

TEORÍA DE AUTÓMATAS
Y LENGUAJES FORMALES

MANUALES UEX

55

(E.E.E.S.)
Espacio
Europeo
Educación
Superior

ELENA JURADO MÁLAGA

TEORÍA DE AUTÓMATAS Y LENGUAJES FORMALES

UNIVERSIDAD DE EXTREMADURA

2008

La publicación del presente manual fue subvencionada por el Vicerrectorado de Calidad y Formación Continua de la Universidad de Extremadura en la Convocatoria de Acciones para la Mejora de la Calidad Docente del curso 2007/08 dentro de la modalidad B.2: "Diseño y desarrollo de materiales docentes adaptados a la metodología derivada del E.E.E.S." Esta convocatoria de acciones forma parte del Plan de Adaptación de la UEX al Espacio Europeo de Educación Superior.

UNIVERSIDAD DE EXTREMADURA

UNIÓN EUROPEA
Fondo Social Europeo

JUNTA DE EXTREMADURA

Edita

Universidad de Extremadura. Servicio de Publicaciones
C./ Caldereros, 2 - Planta 2ª - 10071 Cáceres (España)
Telf. 927 257 041 - Fax 927 257 046
publicac@unex.es
www.unex.es/publicaciones

ISSN 1135-870-X
ISBN 978-84-691-6345-0
Depósito Legal M-45.211-2008

Edición electrónica: Pedro Cid, S.A.
Teléf.: 914 786 125

A Juan

Prólogo

El estudio de la teoría de autómatas y de los lenguajes formales se puede ubicar en el campo científico de la Informática Teórica, un campo clásico y multidisciplinar dentro de los estudios universitarios de Informática. Es un campo clásico debido no sólo a su antigüedad (anterior a la construcción de los primeros ordenadores) sino, sobre todo, a que sus contenidos principales no dependen de los rápidos avances tecnológicos que han hecho que otras ramas de la Informática deban adaptarse a los nuevos tiempos a un ritmo vertiginoso. Es multidisciplinar porque en sus cimientos encontramos campos tan aparentemente dispares como la lingüística, las matemáticas o la electrónica.

El hecho de que esta materia no haya sufrido grandes cambios en las últimas décadas no le resta un ápice de interés. El estudio de las máquinas secuenciales que abarca la teoría de autómatas, por una parte, sienta las bases de la algoritmia y permite modelar y diseñar soluciones para un gran número de problemas. Por otra parte, permite abordar cuestiones de gran interés para un informático como qué tipo de problemas pueden ser resueltos por un computador o, caso de existir una solución computable para un problema, cómo podemos medir la calidad (en términos de eficacia) de dicha solución. Es decir, la teoría de autómatas es la puerta que nos permite la entrada hacia campos tan interesantes como la computabilidad y la complejidad algorítmica. Además, una de las principales aportaciones del estudio de los lenguajes formales, sobre todo desde un punto de vista práctico, es su contribución al diseño de lenguajes de programación y a la construcción de sus correspondientes traductores. En este sentido la asignatura ayudará a conocer con mayor profundidad la estructura de los lenguajes de programación y el funcionamiento de los compiladores.

Este manual es el resultado del trabajo que durante varios años he realizado impartiendo la asignatura de Teoría de Autómatas y Lenguajes Formales en las titulaciones de Informática de la Universidad de Extremadura. Dicha asignatura es troncal para los alumnos de Ingeniería Informática y de Ingeniería Técnica en Informática de Sistemas pero también puede ser cursada como materia optativa por los alumnos de Ingeniería Técnica en Informática de Gestión. Se imparte en el tercer curso de estas titulaciones.

La asignatura se ha diseñado teniendo en cuenta las diferentes circunstancias de los alumnos de las tres titulaciones, así como el tiempo disponible a lo largo del

curso para impartirla. Por tanto, y teniendo en cuenta sobre todo esto último, los contenidos han sido seleccionados de forma realista con la intención de que puedan ser abarcados por completo a lo largo del curso académico. Por este motivo, la alta carga formal que suele acompañar a los libros de esta materia ha sido aligerada, procurando no incluir, siempre que ha sido posible, demostraciones complejas y otras cuestiones formales que harían el texto inabordable en una única asignatura. Como contrapartida se han intentado presentar los conceptos e ideas básicos de una manera intuitiva y clara.

Otro de los aspectos que se ha tenido en cuenta al diseñar los contenidos del manual ha sido destacar una de las aplicaciones prácticas más interesantes de esta materia: el diseño de lenguajes de programación y de compiladores. Esto ha motivado que, al clásico estudio de la jerarquía para las gramáticas formales de Chomsky, de los Autómatas Finitos, de los Autómatas de Pila y de las Máquinas de Turing, se hayan añadido temas como el de las gramáticas atribuidas, los reconocedores LR o un anexo sobre la generación automática de analizadores léxicos y sintácticos. Si bien estás últimas cuestiones pueden considerarse al margen de la asignatura, tienen un claro interés desde el punto de vista práctico.

Considerando la próxima implantación de los nuevos planes de estudio dentro del espacio europeo de educación superior, se ha incluido un apartado en el que se describe el plan docente de la asignatura, indicando la metodología docente, el plan de trabajo del estudiante, así como las competencias específicas de la materia y de la titulación. En este sentido, y teniendo en cuenta que el trabajo no presencial del estudiante tiene cada vez más relevancia en el nuevo marco de las enseñanzas universitarias, consideramos que este manual puede convertirse en una herramienta de gran utilidad para los estudiantes, como complemento a los apuntes y como material de ayuda en la preparación previa de los temas.

Quisiera terminar este prólogo agradeciendo la colaboración de los estudiantes que han cursado esta asignatura durante los últimos años. Sus comentarios y sugerencias han resultado imprescindibles para mejorar los contenidos y el diseño de las primeras versiones de este manual.

Elena Jurado Málaga

Plan Docente

I. Descripción y contextualización

Identificación y características de la materia

Denominación: Teoría de Autómatas y Lenguajes Formales
Curso y Titulaciones:
3º Ingeniería Informática,
3º Ingeniería Técnica en Informática de Sistemas,
3º Ingeniería Técnica en Informática de Gestión
Área: Lenguajes y Sistemas Informáticos
Departamento: Ingeniería de Sistemas Informáticos y Telemáticos
Tipo:
Troncal en Ingeniería Informática,
Troncal en Ingeniería Técnica en Informática de Sistemas,
Optativa en Ingeniería Técnica en Informática de Gestión
Coeficientes:
Practicidad: 3 (Medio)
Agrupamiento: 3 (Medio)
Duración: Anual, 8.18 créditos ECTS (204.5 horas)
Distribución ECTS:
Grupo Grande: 62 horas (30.31 %)
Seminario-Laboratorio: 20 horas (9.77 %)
Tutoría ECTS: 10 horas (4.88 %)
No presencial: 112.5 horas (55.01 %)
Descriptores (según BOE): Teoría de Autómatas y Lenguajes Formales

Perfiles (y subperfiles) profesionales de la titulación

I. Perfil Profesional de Desarrollo Software

II. Perfil Profesional de Sistemas

III. Perfil Profesional de Gestión y Explotación de Tecnologías de la Información

Competencias Específicas de la Titulación (y perfiles relacionados)

1. Aprender de manera autónoma nuevos conocimientos y técnicas adecuados para la concepción, el desarrollo o la explotación de sistemas informáticos.(Todos)

2. Comunicar de forma efectiva, tanto por escrito como oral, conocimientos, procedimientos, resultados e ideas relacionadas con las TIC y, concretamente de la Informática, conociendo su impacto socioeconómico.(Todos)

3. Comprender la responsabilidad social, ética y profesional, y civil en su caso, de la actividad del Ingeniero en Informática y su papel en el ámbito de las TIC y de la Sociedad de la Información y del Conocimiento.(Todos)

4. Concebir y llevar a cabo proyectos informáticos utilizando los principios y metodologías propios de la ingeniería.(I,II)

5. Diseñar, desarrollar, evaluar y asegurar la accesibilidad, ergonomía, usabilidad y seguridad de los sistemas, aplicaciones y servicios informáticos, así como de la información que proporcionan, conforme a la legislación y normativa vigentes.(I)

6. Definir, evaluar y seleccionar plataformas hardware y software para el desarrollo y la ejecución de aplicaciones y servicios informáticos de diversa complejidad.(I,II)

7. Disponer de los fundamentos matemáticos, físicos, económicos y sociológicos necesarios para interpretar, seleccionar, valorar, y crear nuevos conceptos, teorías, usos y desarrollos tecnológicos relacionados con la informática, y su aplicación.(Todos)

8. Concebir, desarrollar y mantener sistemas y aplicaciones software empleando diversos métodos de ingeniería del software y lenguajes de programación adecuados al tipo de aplicación a desarrollar manteniendo los niveles de calidad exigidos.(I)

9. Concebir y desarrollar sistemas o arquitecturas informáticas centralizadas o distribuidas integrando hardware, software y redes.(II)

10. Proponer, analizar, validar, interpretar, instalar y mantener soluciones informáticas en situaciones reales en diversas áreas de aplicación dentro de una organización.(Todos)

11. Concebir, desplegar, organizar y gestionar sistemas y servicios informáticos en contextos empresariales o institucionales para mejorar sus procesos de negocio, responsabilizándose y liderando su puesta en marcha y mejora continua, así como valorar su impacto económico y social.(III)

II. Objetivos. Competencias Específicas de la Materia (y relación con CET)

1. Ser capaz de realizar operaciones básicas con cadenas y con lenguajes.(7)
2. Conocer los diferentes tipos de gramáticas y de lenguajes que forman la jerarquía de Chomsky y su utilidad en el diseño de los lenguajes de programación y sus traductores.(7)
3. Saber cómo reconocer de que tipo es una determinada gramática o lenguaje.(7)
4. Ser capaz de construir autómatas para resolver diferentes tipos de problemas y para reconocer diferentes lenguajes.(7,11)
5. Conocer el funcionamiento de diferentes tipos de autómatas y entender el tipo de problemas que cada uno puede resolver.(7,11)
6. Conocer y ser capaz de utilizar métodos que permitan construir, dado un lenguaje regular: gramáticas que los generan, expresiones regulares que los representan y autómatas finitos que los reconocen.(7)
7. Conocer el esquema funcional de un traductor.(4)
8. Conocer los diferentes aspectos léxicos, sintácticos y semánticos que hay que tener en cuenta a la hora de diseñar lenguajes formales y traductores para esos lenguajes.(4,8)
9. Ser capaz de construir un traductor.(1,2,4,8,10)
10. Ser capaz de diseñar una gramática para un lenguaje de programación sencillo.(4,7)
11. Conocer metalenguajes que permitan describir lenguajes regulares y lenguajes independientes del contexto.(1,8)
12. Saber utilizar herramientas que permitan generar automáticamente analizadores léxicos y sintácticos.(1,8)
13. Ser capaz de calcular la complejidad espacial y temporal de máquinas de Turing sencillas.(5,6,10,11)
14. Entender el concepto de recursividad y de calculabilidad.(5,6,7,10,11)
15. Ser capaz de demostrar que algunas funciones son recursivas primitivas o μ-recursivas.(5,6,7)
16. Entender el concepto y conocer ejemplos de problemas de la clase P, NP o NP-completo.(5,6,7)

III. Contenidos

Los contenidos de la asignatura son los que se describen a lo largo de este documento (ver *Índice General*).

Interrelaciones

1. Las asignaturas de 1º curso, **Elementos de Programación** y **Laboratorio de Programación I**, y de 2º curso, **Estructuras de Datos y Algoritmos** y **Laboratorio de Programación II**, son requisitos para esta asignatura ya que proporcionan los conocimientos sobre programación necesarios para abordar la tarea de la construcción de un compilador. También permiten que el alumno conozca con antelación conceptos básicos para la asignatura como el de lenguaje de programación y el de compilador.

2. El concepto de complejidad algorítmica se ha tratado en asignaturas como **Elementos de Programación** y **Estructuras de Datos y Algoritmos**. En nuestra asignatura éste se relaciona con el concepto de Maquina de Turing.

3. La asignatura de Teoría de Autómatas y Lenguajes Formales debe sentar las bases para que los alumnos de Ingeniería Informática puedan abordar con garantías la asignatura de 5º curso **Procesadores de Lenguajes**

IV. Metodología docente y plan de trabajo del estudiante

En esta sección se describe la metodología utilizada para cada uno de los temas que aparecen en este manual. A cada una de las actividades programadas se le asigna un determinado tipo y el número de horas estimadas de dedicación del estudiante. A continuación se indican las siglas utilizadas para representar los diferentes tipos de actividades:

GG	Grupo grande	**S**	Seminario-Laboratorio
Tut	Tutoría ECTS	**NO**	No Presencial.

Tema 1. Preliminares (Objetivos: 1,7)

Lectura y estudio (previo y/o posterior)(NP): 1h.

Explicación, discusión y ejemplificación en clase(GG): 3h.

Realización de ejercicios propuestos (NP): 3h.

Tema 2. Lenguajes y Gramáticas Formales (Objetivos: 1,2)

Lectura y estudio (previo y/o posterior)(NP): 2h.

Explicación, discusión y ejemplificación en clase(GG): 6h.

Explicación de cuestiones y ejercicios relacionados con la teoría(GG): 2h.

Realización de ejercicios propuestos (NP): 10h.

Tema 3. Expresiones y gramáticas regulares (Objetivos: 1,3,8,10,12,11)

Lectura y estudio (previo y/o posterior)(NP): 2h.

Explicación, discusión y ejemplificación en clase(GG): 1h.

Explicación de cuestiones y ejercicios relacionados con la teoría(GG): 2h.

Prácticas (S): 2h.

Revisión de las actividades prácticas (Tut): 2h.

Realización de ejercicios propuestos (NP): 4h.

Tema 4. Autómatas Finitos (Objetivos: 3,4,5,8,6,12,11)

Lectura y estudio (previo y/o posterior)(NP): 10h.

Explicación, discusión y ejemplificación en clase(GG): 10h.

Explicación de cuestiones y ejercicios relacionados con la teoría(GG): 7h.

Prácticas (S): 6h.

Revisión de las actividades prácticas (Tut): 2h.

Realización de ejercicios propuestos (NP): 20h.

Evaluación del primer bloque temático

Tema 5. Gramáticas Independientes del Contexto (GIC) y Autómatas de Pila (Objetivos: 3,4,5,8,9,10,12,11)

Lectura y estudio (previo y/o posterior)(NP): 10h.

Explicación, discusión y ejemplificación en clase(GG): 6h.

Explicación de cuestiones y ejercicios relacionados con la teoría(GG): 5h.

Prácticas (S): 10h.

Revisión de las actividades prácticas (Tut): 3h.

Realización de ejercicios propuestos (NP): 12h.

Tema 6. Gramáticas Atribuidas (Objetivos: 7,8,10,12,11)

Lectura y estudio (previo y/o posterior)(NP): 3h.

Explicación, discusión y ejemplificación en clase(GG): 2h.

Explicación de cuestiones y ejercicios relacionados con la teoría(GG): 2h.

Prácticas (S): 2h.

Revisión de las actividades prácticas (Tut): 3h.

Realización de ejercicios propuestos (NP): 5h.

Tema 7. Máquinas de Turing(MT) (Objetivos: 4,5)

Lectura y estudio (previo y/o posterior)(NP): 4h.

Explicación, discusión y ejemplificación en clase(GG): 5h.

Explicación de cuestiones y ejercicios relacionados con la teoría(GG): 4h.

Realización de ejercicios propuestos (NP): 14h.

Tema 8. Gramáticas de tipo 0 y 1 (Objetivos: 3)

Lectura y estudio (previo y/o posterior)(NP): 2h.

Explicación, discusión y ejemplificación en clase(GG): 2h.

Realización de ejercicios propuestos (NP): 2h.

Tema 9. Computabilidad y Máquinas de Turing (Objetivos: 13,14,15)

Lectura y estudio (previo y/o posterior)(NP): 2h.

Explicación, discusión y ejemplificación en clase(GG): 2h.

Explicación de cuestiones y ejercicios relacionados con la teoría(GG): 1h.

Realización de ejercicios propuestos (NP): 4h.

Tema 10. Introducción a la Complejidad Computacional (Objetivos: 16)

Lectura y estudio (previo y/o posterior)(NP): 2h.

Explicación, discusión y ejemplificación en clase(GG): 2h.

Realización de ejercicios propuestos (NP): 2h.

V. Evaluación

Criterios de Evaluación

1. Aplicar los conceptos y métodos estudiados para la resolución de problemas relacionados con el diseño de autómatas y gramáticas.

   En este aspecto, las principales habilidades a tener en cuenta son:

   a) Definir formalmente un lenguaje.

b) Ser capaz de determinar el tipo al que pertenece un lenguaje.

c) Ser capaz de convertir un Autómata Finito No Determinista en Autómata Finito Determinista.

d) Diseñar una gramática a partir del autómata que reconoce al lenguaje que ésta genera.

e) Construir el autómata que reconoce a un determinado lenguaje.

f) Diseñar una máquina de Turing que resuelva un problema dado o que reconozca o genere un lenguaje determinado.

g) Ser capaz de probar que una función es recursiva primitiva o μ-recursiva.

2. Diseñar un lenguaje formal y construir, utilizando las herramientas adecuadas, un traductor para dicho lenguaje.

Actividades e instrumentos de evaluación

Examen parcial Prueba de desarrollo escrito con 1 pregunta dirigida a la comprensión de conceptos y 4 o 5 a la aplicación los métodos para resolver problemas relacionados con el diseño de lenguajes y autómatas (Temas 1-4). (20 %)

Examen final Prueba de desarrollo escrito con 1 pregunta dirigida a la comprensión de conceptos y 4 o 5 a la aplicación los métodos para resolver problemas relacionados con el diseño de lenguajes y autómatas. (55 %)

Seminarios y Tutorías ECTS Revisión y análisis del trabajo no presencial del alumno así como del trabajo desarrollado en los Seminarios.

1. Revisión de ejercicios prácticos a realizar por el alumno durante los Seminarios. (5 %)
2. Revisión del lenguaje y del traductor construido por el alumno, haciendo especial hincapié en un correcto diseño de ambos. Prueba en la que el alumno debe realizar una sencilla ampliación de su trabajo práctico con el objetivo de valorar el control que tiene sobre su trabajo. (20 %)

Índice General

1. Preliminares **5**
- 1.1. Antecedentes históricos y conceptos básicos . . . 5
- 1.2. Desarrollo de la asignatura . . . 8
- 1.3. Conceptos básicos sobre compiladores . . . 10
  - 1.3.1. Componentes de un compilador . . . 11

2. Lenguajes y Gramáticas Formales **15**
- 2.1. Definiciones básicas . . . 15
- 2.2. Operaciones con palabras . . . 16
- 2.3. Lenguajes y operaciones con lenguajes . . . 17
- 2.4. Concepto de gramática formal . . . 19
  - 2.4.1. Definiciones previas . . . 19
  - 2.4.2. Ejemplo con un lenguaje natural(castellano) . . . 20
  - 2.4.3. Ejemplo en un lenguaje artificial . . . 21
  - 2.4.4. Definición de gramática formal . . . 22
- 2.5. Clasificación de las gr. formales . . . 24
  - 2.5.1. Gramáticas de tipo 0 . . . 24
  - 2.5.2. Gramáticas de tipo 1 . . . 25
  - 2.5.3. Gramáticas de tipo 2 . . . 25
  - 2.5.4. Gramáticas de tipo 3 . . . 26
- 2.6. Gramáticas equivalentes . . . 26
  - 2.6.1. Simplificación de gramáticas . . . 26
- 2.7. Problemas y cuestiones . . . 29

3. Expresiones y gramáticas regulares **33**
- 3.1. Definición de expresión regular . . . 33
- 3.2. Álgebra de las expresiones regulares . . . 34
- 3.3. Definición de gramática regular . . . 35
- 3.4. Ejemplos . . . 36
- 3.5. Problemas . . . 38

4. Autómatas Finitos **39**
4.1. Introducción . . . 39
4.2. Definición de Autómata Finito Determinista . . . 40
4.3. Representación de Autómatas . . . 40
4.4. Los AFD como reconocedores de lenguajes . . . 43
4.5. Minimización de un AFD . . . 43
4.6. Autómatas Finitos No Deterministas(AFND) . . . 47
4.7. Lenguaje aceptado por un AFND . . . 50
4.8. Simulación de un AFD y AFND . . . 51
4.9. Paso de un AFND a AFD . . . 52
4.10. Relación entre AF, gr. y exp. reg. . . . 55
4.10.1. Construcción de la expresión regular reconocida por un AF . . . 55
4.10.2. Construcción del AF que reconoce una expresión regular . . . 58
4.10.3. Relación entre A.F. y gramáticas regulares . . . 62
4.11. Límites para los leng. regulares . . . 65
4.11.1. El lema del bombeo(*pumping lemma*) . . . 65
4.11.2. El teorema de Myhill-Nerode . . . 66
4.12. Problemas . . . 70

5. G.I.C y Autómatas de Pila **73**
5.1. Definición de G.I.C. . . . 74
5.2. Autómatas de Pila . . . 74
5.3. Árboles de derivación . . . 77
5.3.1. Ambigüedad. . . . 78
5.4. Reconocimiento descendente . . . 80
5.4.1. Simplificación de las GIC . . . 81
5.4.2. Reconocedores LL(1) . . . 83
5.5. Reconocimiento ascendente . . . 87
5.5.1. Construcción de la Tabla de Acciones . . . 89
5.6. Propiedades de los L.I.C. . . . 93
5.6.1. El lema del bombeo para LIC(*pumping lemma*) . . . 94
5.7. Problemas . . . 95

6. Gramáticas Atribuidas **97**
6.1. Concepto de Semántica y de Gramática Atribuida . . . 97
6.2. Atributos heredados y sintetizados . . . 99
6.3. Gramáticas S-atribuidas y L-Atribuidas . . . 100
6.3.1. Gramáticas S-atribuidas . . . 100
6.3.2. Gramáticas L-atribuidas . . . 101
6.4. Problemas . . . 104

7. Máquinas de Turing **105**
7.1. Introducción. Antecedentes históricos . . . 105
7.2. Definición y ejemplos de M.T.'s . . . 107
7.3. Restricciones a la M.T. . . . 110
7.3.1. M.T. con alfabeto binario . . . 111
7.3.2. M.T. con la cinta limitada en un sentido . . . 112
7.3.3. M.T. con restricciones en cuanto a las operaciones que realiza simultáneamente . . . 112
7.4. Modificaciones de la M.T. . . . 113
7.4.1. Almacenamiento de información en el control finito . . . 113
7.4.2. Pistas múltiples . . . 114
7.4.3. Símbolos de chequeo . . . 114
7.4.4. Máquinas multicinta . . . 115
7.4.5. M.T. no determinista . . . 116
7.5. Técnicas para la construcción de M.T. . . . 116
7.6. La M.T. Universal . . . 118
7.7. La M.T. como generadora de lenguajes . . . 119
7.8. La tesis de Church-Turing . . . 120
7.9. Problemas . . . 120

8. Gramáticas de tipo 0 y 1 **123**
8.1. Gramáticas de tipo 0 . . . 123
8.2. Lenguajes de tipo 0 . . . 124
8.3. El problema de la parada . . . 126
8.4. Lenguajes y gramáticas de tipo 1 . . . 126

9. Computabilidad y Máquinas de Turing **129**
9.1. Funciones calculables . . . 129
9.2. Funciones recursivas . . . 131
9.2.1. Funciones recursivas primitivas . . . 132
9.2.2. Funciones μ-recursivas . . . 134
9.3. Problemas . . . 134

10. Introducción a la Complejidad Computacional **137**
10.1. Complejidad y Máquinas de Turing . . . 137
10.2. Medidas de complejidad algorítmica . . . 138
10.3. Problemas P, NP y NP-completos . . . 141

A. Generación automática de analizadores **143**
A.1. Generador de analizadores léxicos . . . 143
A.1.1. Cómo utilizar PCLEX . . . 144
A.1.2. Estructura de un programa **Lex** . . . 144

A.1.3. Cómo representar una expresión regular 146
A.1.4. Variables y procedimientos predefinidos 147
A.1.5. Condiciones de comienzo 148
A.1.6. Acciones 148
A.2. Generador de analizadores sintácticos 149
A.2.1. Cómo utilizar PCYACC 150
A.2.2. Estructura de un programa para YACC 151
A.2.3. Gramáticas atribuidas 152
A.2.4. Prioridad y asociatividad de operadores 152

Tema 1

Preliminares

Contenido

1.1. Antecedentes históricos y conceptos básicos 5
1.2. Desarrollo de la asignatura 8
1.3. Conceptos básicos sobre compiladores 10

En este primer tema de la asignatura pretendemos sentar las bases de la misma y explicar cuál va a ser su estructura. En la sección 1.1 explicaremos cuales son los temas centrales sobre los que va a girar la asignatura, así como los antecedentes históricos sobre los que se han desarrollado estos temas, mencionando los personajes que más han influido en su nacimiento. La estructura de la asignatura es el contenido fundamental de la sección 1.2 en la que también se detallan algunas de las aplicaciones más interesantes de los conceptos teóricos que estudiaremos. Teniendo en cuenta que probablemente la aplicación práctica más interesante de esta asignatura es el diseño de lenguajes de computación y la construcción de sus correspondientes traductores, en la sección 1.3 se sentarán las bases que permitirán entender la estructura de un compilador y las tareas que debe llevar a cabo.

1.1. Antecedentes históricos y conceptos básicos

La mayor parte del conocimiento científico es el resultado de muchos años de investigación, con frecuencia sobre temas que aparentemente no tienen una relación directa. Como veremos, esto sucede también con un campo como la Informática Teórica (ámbito en el que se enmarca la asignatura de Teoría de Autómatas y Lenguajes Formales). Esta materia se ha desarrollado gracias a la confluencia de campos muy diferentes, como son: las matemáticas, la teoría de máquinas, la lingüística, etc. Podemos considerarla, por tanto, como una materia multidisciplinar. En este apartado pretendemos explicar de manera intuitiva los conceptos básicos que constituyen las

bases de la asignatura así como los campos científicos que han influido fundamentalmente en el desarrollo de esta materia y que nos ayudarán a entender sus aplicaciones más importantes.

Veamos, en primer lugar, la relación entre conceptos que trataremos a lo largo de todo el temario: **lenguaje**, **gramática** y **autómata**. Toda comunicación conlleva la utilización de un lenguaje, que podemos definir como un conjunto de palabras (también llamadas *cadenas*) formadas por símbolos de un alfabeto. Las gramáticas permitirán definir la estructura de los lenguajes, es decir, proporcionarán las formas válidas en las que se pueden combinar los símbolos del alfabeto para construir cadenas correctas. Una máquina abstracta o autómata es un dispositivo teórico que recibe como entrada una cadena de símbolos y los procesa, cambiando de estado, de manera que genera una determinada salida. Los autómatas pueden servir, entre otras cosas, para determinar si una palabra pertenece o no a un determinado lenguaje. Por lo tanto, las gramáticas nos permitirán definir lenguajes y los autómatas podrán reconocer las palabras de dichos lenguajes. A pesar de la conexión que existe entre estos conceptos, los trabajos iniciales sobre autómatas y lenguajes tienen, como veremos a continuación, un origen diferente.

Para encontrar los principios de la Informática Teórica debemos remontarnos a los años 30, década en la que el mundo de las matemáticas se hallaba ocupado, sobre todo, en temas como la lógica y la definición de sistemas axiomáticos.

El método axiomático requiere una colección de enunciados básicos, llamados **axiomas**, que describen las propiedades fundamentales del sistema que se estudia. A partir de estos axiomas, se derivan enunciados adicionales, llamados teoremas, aplicando secuencias finitas de **reglas de inferencia**.

Una ventaja del método axiomático es que ofrece un modelo de razonamiento deductivo en el cual todas las suposiciones están aisladas en los axiomas iniciales y las reglas de inferencia. Cualquier enunciado que se derive posteriormente será una consecuencia de estas suposiciones.

A principios del siglo XX, muchos matemáticos creían que era posible encontrar un único sistema axiomático en el que podrían basarse todas las matemáticas. Su meta era encontrar un conjunto de axiomas y reglas de inferencia correctos de manera que las matemáticas pudieran reducirse a un sistema computacional con el cual pudiera deducirse la veracidad o falsedad de cualquier enunciado matemático. Uno de los principales defensores de esta idea era el conocido matemático alemán **Hilbert**.

Sin embargo, en 1931, el austriaco **Kurt Gödel** publicó el "*Teorema de la incompletitud*", en el que demostraba que era imposible la completa axiomatización de las matemáticas. Este teorema incrementó el debate por el poder de los métodos axiomáticos y los procesos computacionales.

En 1937, el matemático inglés **Alan Turing**, en su artículo "*Sobre los números computables*", presentó la conocida **Máquina de Turing**(M.T.), una entidad matemá

abstracta con la que se formalizó el concepto de algoritmo[1], además demostró que muchos problemas perfectamente definidos no pueden ser resueltos mediante una M.T., es decir, no son computables y de esta manera ratificó la teoría de Gödel. Esta máquina sería la precursora, desde un punto de vista teórico, de los computadores que se construyeron durante la siguiente década. La Máquina de Turing tiene el poder computacional más alto conocido hasta el momento, es decir, es capaz de resolver cualquier problema que tenga una solución computacional.

Podemos considerar todo esto como el primer eslabón dentro del campo de la Informática Teórica. El segundo eslabón se ubicaría en un campo muy cercano a la Electrónica en el que el matemático **Shannon** estableció las bases para la aplicación de la lógica matemática al diseño de los circuitos combinatorios y secuenciales. Las ideas de Shannon derivarían en la formalización de una teoría de máquinas secuenciales y autómatas, cuyo principal objetivo era representar de manera formal el comportamiento de un determinado dispositivo electrónico o mecánico. Los autómatas son, en un sentido amplio, sistemas que aceptan señales del medio que les rodea, cambian de estado como consecuencia de estas señales y trasmiten otras señales al medio. En este sentido, un electrodoméstico común, una central telefónica, un ordenador, e incluso ciertas facetas del comportamiento de los seres vivos pueden modelarse mediante autómatas. A finales de los años 50 se comenzó a estudiar la utilidad de los autómatas en relación con los lenguajes de programación y su proceso de traducción.

La Teoría de Autómatas estudia diferentes niveles de autómatas entre los que podemos destacar, de una parte, los Autómatas Finitos por constituir el grupo más sencillo de autómatas y, de otra, las Máquinas de Turing que, por el contrario, son los autómatas más complejos y con mayor poder computacional. Estos dos tipos de autómatas representan los dos extremos de la jerarquía, otros niveles intermedios los encontramos en los Autómatas de Pila y en los Autómatas Linealmente Acotados. Básicamente, la diferencia principal entre estos autómatas estriba en la utilización o no de memoria auxiliar y en la forma de acceso a dicha memoria.

Para llegar al tercer eslabón de la Informática Teórica hay que saltar al campo de la lingüística. En la década de los años 50, el lingüista y pensador **Noam Chomsky**, en un intento de formalizar los lenguajes naturales, estableció las bases de la lingüística matemática o formal y con ello proporcionó una poderosa herramienta que facilitó la definición de los primeros lenguajes de programación, que empezaban a surgir en esa época.

[1]Un algoritmo puede considerarse como un método genérico que, en un número finito de pasos o computaciones, permite resolver un determinado problema. Recordemos que la palabra algoritmo debe su nombre a un matemático persa Abu Ja'far Mohamed ibn Musa al-Jowâizmî, autor de un tratado de aritmética publicado en el año 825.

1.2. Desarrollo de la asignatura

Chomsky clasificó las gramáticas formales (y los lenguajes que éstas generan) de acuerdo a una jerarquía de cuatro niveles representada en la tabla 1.1. Sorprendentemente, es posible establecer una relación biunívoca entre los diferentes niveles de la jerarquía de Chomsky y cuatro niveles de una jerarquía definida entre los diferentes tipos de autómatas. A cada nivel de gramática se le puede asociar de forma natural un conjunto de lenguajes que serán los que esas gramáticas generan, pero además, se le puede asociar una clase de autómatas formada por aquellos que podrán reconocer a dichos lenguajes.

La siguiente figura describe la relación que hay entre los diferentes niveles de gramáticas de la jerarquía de Chomsky, los lenguajes que generan y las máquinas que reconocen estos lenguajes.

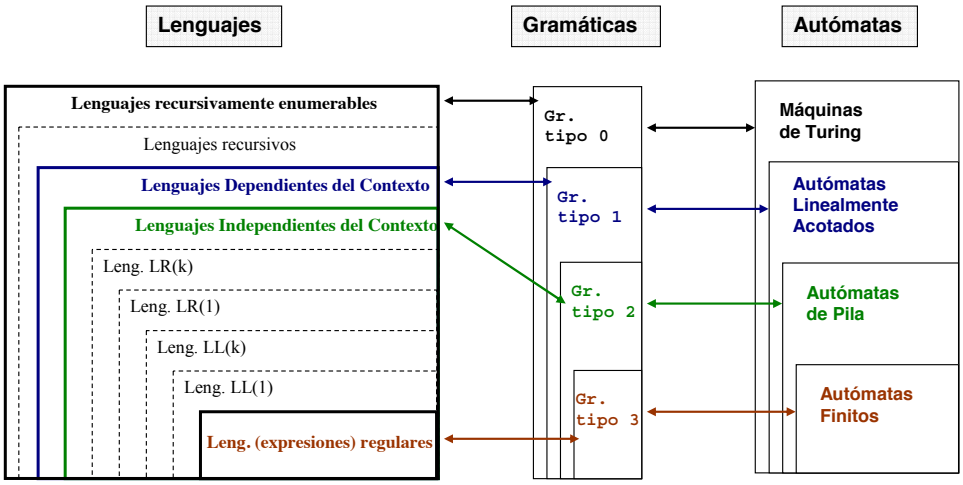


Figura 1.1: Jerarquía de gramáticas, lenguajes y autómatas

Cada nivel de lenguaje se corresponde con un tipo de autómata. Por ejemplo, dado un lenguaje de tipo 3 siempre será posible encontrar un Autómata Finito que reconozca dicho lenguaje, es decir, que permita determinar si una determinada palabra pertenece o no al lenguaje. Si el lenguaje es de tipo 2 será necesario utilizar un autómata más complejo, concretamente un Autómata de Pila. La figura 1.1, además de sintetizar la Jerarquía de Chomsky, presenta un sencillo esquema de los temas a abordar durante el curso. Comenzaremos con el estudio de los lenguajes más sencillos

(los de tipo 3) y de los autómatas asociados a ellos, para acabar con el estudio de los lenguajes de tipo 0 y de las M.T. Los nombres de los lenguajes de la jerarquía de Chomsky aparecen destacados en esta figura, pero también aparecen otros conjuntos de lenguajes que representan niveles intermedios de esta jerarquía y que por su interés serán también estudiados en la asignatura. Concretamente, los lenguajes LL(1) y LR(1), incluidos en el conjunto de los lenguajes independientes del contexto se estudiarán en el tema 5 y los lenguajes recursivos serán tratados en el tema 8.

Como muestra la figura 1.1, en los tres conceptos estudiados (gramáticas, lenguajes y autómatas) cada nivel contiene al anterior. Por ejemplo, cualquier lenguaje de tipo 3 es a su vez un lenguaje de tipo2 (sin embargo, comprobaremos que lo contrario no es cierto), es decir ($L3 \subsetneq L2 \subsetneq L1 \subsetneq L0$). De la misma forma, un Autómata Finito puede considerarse como un caso particular de Autómata de Pila y éste como un caso particular de Máquina de Turing.

A continuación comentaremos cuáles son las principales aplicaciones de los temas que se estudiarán en esta asignatura.

- El estudio de las gramáticas formales será una herramienta muy útil para el diseño de los lenguajes de programación. El estudio de determinados autómatas (concretamente los autómatas finitos y los de pila) permitirá construir de manera sistemática algunos de los componentes básicos de los compiladores.

- Los autómatas finitos se pueden aplicar con éxito en el procesamiento del lenguaje natural, por ejemplo, en la categorización gramatical de las palabras en una oración o en la extracción de información a partir de grandes volúmenes de texto. También pueden ser utilizados para manipular ficheros de texto que almacenan una información estructurada, por ejemplo, con objeto de modificar dicha estructura.

- Los autómatas, en general, tienen una gran aplicación en el mundo industrial, ya que permiten modelar el comportamiento de cualquier dispositivo electromecánico: una cadena de montaje, un robot, un electrodoméstico, etc. También pueden ser utilizados para el reconocimiento de patrones y para el diseño de redes neuronales. Los Autómatas Finitos ayudan a diseñar software que compruebe la corrección de cualquier sistema que tenga un número finito de estados: protocolos de comunicación, protocolos para el intercambio seguro de información, etc. También se utilizan en el diseño y la verificación del comportamiento de circuitos digitales.

- Los lenguajes regulares se pueden utilizar para especificar argumentos en determinados comandos de un sistema operativo o de un sistema de búsqueda de información.

- Los autómatas son también esenciales para el estudio de los límites de la computación. En este terreno, existen dos cuestiones importantes que nos podemos plantear y que se estudiarán en los últimos temas de la asignatura:

  1. ¿Qué puede hacer un computador? Utilizaremos el concepto de *computabilidad* para aplicarlo a los problemas que puede resolver un computador.
  2. ¿Qué puede hacer un computador eficientemente? Diremos que un problema es *tratable*, si un computador puede resolverlo en un tiempo que crezca *lentamente* al aumentar el tamaño de los datos de entrada.

1.3. Conceptos básicos sobre compiladores

Salvo los dos últimos temas que se centrarán en la teoría de la computación, el planteamiento del resto de los temas estará, en general, enfocado al diseño de lenguajes de programación y a la construcción de compiladores, teniendo en cuenta que es ésta una de las aplicaciones más interesantes de los conceptos teóricos tratados en esta asignatura. Por este motivo, en esta sección se introducen conceptos muy elementales sobre el proceso de traducción de los lenguajes de programación.

Un traductor es un programa que recibe como entrada un texto escrito en un lenguaje, llamado **fuente**, y genera como salida otro texto equivalente pero escrito en un lenguaje diferente denominado **objeto**.

En el caso de que el lenguaje fuente sea un lenguaje de programación de alto nivel y el objeto sea un lenguaje de bajo nivel (ensamblador o código máquina), a dicho traductor se le denomina **compilador**. Análogamente, un **ensamblador** es un traductor cuyo lenguaje fuente es el lenguaje ensamblador.

A diferencia de los programas mencionados anteriormente, un **intérprete** es un traductor que no genera un programa en código objeto, sino que toma una sentencia del programa fuente en un lenguaje de alto nivel, la traduce y la ejecuta directamente.

En los primeros lenguajes, y debido a la escasez de memoria de los ordenadores de la época, se impuso la utilización de intérpretes frente a la de compiladores, pues el programa fuente y el intérprete juntos requerían menos memoria que la que era necesaria para el proceso de compilación. Por ello, los primero ordenadores personales tenían instalado habitualmente un intérprete para el lenguaje BASIC. Sin embargo, con el tiempo se impusieron los compiladores debido, sobre todo, a la información que ofrecían sobre los errores cometidos por el programador, y a una mayor velocidad de ejecución del código resultante. A modo de resumen, los siguientes párrafos indican las ventajas que pueden tener un método de traducción frente al otro.

Ventajas del compilador frente al intérprete

- El programa se compila una sola vez, pero se puede ejecutar muchas.

- La ejecución del programa objeto es mucho más rápida que la interpretación del fuente.

- El compilador tiene una visión completa del programa, por lo que puede dar una información más detallada de los errores cometidos por el programador.

Ventajas del intérprete

- El intérprete necesita menos memoria que el compilador.

- Permite una mayor interactividad con el código en tiempo de desarrollo.

Un compilador no suele funcionar de manera aislada sino que se apoya en otros programas para conseguir su objetivo. Algunos de estos programas de apoyo se describen a continuación. El **preprocesador** se ocupa de incluir ficheros, expandir macros, eliminar comentarios, etc. El **enlazador** (*linker*) construye el fichero ejecutable añadiendo al fichero objeto las cabeceras necesarias y las funciones de librería utilizadas por el programa fuente. El **depurador** permite seguir paso a paso la ejecución del programa. Finalmente, muchos compiladores generan un programa en lenguaje ensamblador que debe después convertirse en un ejecutable mediante la utilización de un **ensamblador**.

1.3.1. Componentes de un compilador

Un compilador es un programa complejo en el que no es fácil distinguir claramente unas partes de otras. Sin embargo, se ha conseguido establecer una división lógica del compilador en fases, lo que permite formalizar y estudiar por separado cada una de ellas. En la práctica, estas fases no siempre se ejecutan secuencialmente sino que lo hacen simultáneamente, pudiendo ser unas fases tratadas como subrutinas de otras.

Análisis léxico El analizador léxico, también conocido como *scanner*, lee los caracteres del programa fuente, uno a uno, desde el fichero de entrada y va formando grupos de caracteres con alguna relación entre sí (*tokens*). Cada token es tratado como una única entidad, constituyendo la entrada de la siguiente fase del compilador.

Existen diferentes tipos de tokens y a cada uno se le puede asociar un *tipo* y, en algunos casos, un *valor*. Los tokens se pueden agrupar en dos categorías:

Cadenas específicas, como las palabras reservadas (*if, while,* ...), signos de puntuación (., ,, =, ...), operadores aritméticos (+,*, ...) y lógicos (*AND, OR,* ...), etc. Habitualmente, las cadenas específicas no tienen asociado ningún valor, sólo su tipo.

Cadenas no específicas, como los identificadores o las constantes numéricas o de texto. Las cadenas no específicas siempre tienen tipo y valor. Por ejemplo, si **dato** es el nombre de una variable, el tipo del token será *identificador* y su valor será *dato*.

Frecuentemente el analizador léxico funciona como una subrutina del analizador sintáctico. Para el diseño del analizadores léxicos se utilizan los Autómatas Finitos.

Análisis sintáctico El analizador sintáctico, también llamado *parser*, recibe como entrada los tokens que genera el analizador léxico y comprueba si estos tokens van llegando en el orden correcto. Siempre que no se hayan producido errores, la salida teórica de esta fase del compilador será un árbol sintáctico. Si el programa es incorrecto se generarán los mensajes de error correspondientes. Para el diseño de los analizadores sintácticos se utilizan los Autómatas de Pila.

Análisis semántico El analizador semántico trata de determinar si el significado de las diferentes instrucciones del programa es válido. Para conseguirlo tendrá que calcular y analizar información asociada a las sentencias del programa, por ejemplo, deberá determinar el tipo de los resultados intermedios de la expresiones, comprobar que los argumentos de un operador pertenecen al conjunto de los operandos posibles, comprobar que los operandos son compatibles entre si, etc.

La salida teórica de esta fase será un árbol semántico. Éste es una ampliación de un árbol sintáctico en el que cada rama del árbol ha adquirido, además, el significado que debe tener el fragmento de programa que representa. Esta fase del análisis es más difícil de formalizar que las dos anteriores y se utilizarán para ello las gramáticas atribuidas.

Generación de código intermedio Cuando una empresa se dedica a la generación de compiladores, normalmente trabaja con muchos lenguajes fuentes (**m**) y con muchos lenguajes objetos (**n**) diferentes. Para evitar el tener que construir **m*n** compiladores, se utiliza un lenguaje intermedio. De esta forma sólo hay que construir **m** programas que traduzcan cada lenguaje fuente al código intermedio (**front ends**) y **n** programas que traduzcan del lenguaje intermedio a cada lenguaje objeto (**back ends**). La utilización del lenguaje intermedio permite construir en menos tiempo compiladores para nuevos lenguajes y para nuevas máquinas. Desgraciadamente, no existe consenso para utilizar un único lenguaje intermedio.

Optimización de código La mayoría de los compiladores suelen tener una fase de optimización de código intermedio (independiente de los lenguajes fuente y objeto), y una fase de optimización de código objeto (no aplicable a otras máquinas).

Estas fases se añaden al compilador para conseguir que el programa objeto sea más rápido y necesite menos memoria para ejecutarse.

Veamos en los siguientes párrafos algunos ejemplos de optimización:

- Eliminar expresiones comunes. Por ejemplo:

A := B+C+D		Aux := B+C
E := (B+C)*F	se convierte en	A := Aux + D
		E := Aux * F

- Optimizar los bucles. Se trata de sacar de los bucles aquellas expresiones que sean invariantes.

```
REPEAT
    B := 1
    A := A-B
UNTIL A = 0   La asignación B := 1 se puede sacar del bucle
```

Generación de código objeto En esta fase, el código intermedio optimizado es traducido a una secuencia de instrucciones en ensamblador o en código máquina. Por ejemplo, la sentencia A := B+C se convertiría en una colección de instrucciones que podrían representarse así:

```
LOAD B
ADD C
STORE A
```

Tabla de símbolos El compilador necesita gestionar la información de los *elementos* que se va encontrando en el programa fuente: variables, tipos, funciones, clases, etc. Esta información se almacena en una estructura de datos interna conocida como **tabla de símbolos**.

Para que la compilación sea eficiente la tabla debe ser diseñada cuidadosamente de manera que contenga toda la información que el compilador necesita. Además, hay que prestar especial atención a la velocidad de acceso a la información con objeto de no ralentizar el proceso.

Control de errores Informar adecuadamente al programador de los errores que hay en su programa es una de las misiones más importantes y complejas de un compilador. Es una tarea difícil porque a veces unos errores ocultan a otros, o porque un error desencadena una avalancha de errores asociados. El control de errores se lleva a cabo, sobre todo, en las etapas de análisis sintáctico y semántico.

Tema 2

Lenguajes y Gramáticas Formales

Contenido

2.1. Definiciones básicas **15**
2.2. Operaciones con palabras **16**
2.3. Lenguajes y operaciones con lenguajes **17**
2.4. Concepto de gramática formal **19**
2.5. Clasificación de las gr. formales **24**
2.6. Gramáticas equivalentes **26**
2.7. Problemas y cuestiones **29**

En este tema se abordan los conceptos de gramática y lenguaje formal. El tema comienza con la definición de una serie de conceptos básicos, seguidamente, se estudian las diferentes operaciones que se pueden llevar a cabo con palabras y con lenguajes, y las propiedades que estas operaciones tienen. Probablemente el punto central de este tema es la introducción del concepto de gramática formal, que se hará en primer lugar de forma intuitiva, hasta llegar a una definición formal. El tema acaba con la presentación de una taxonomía de las gramáticas formales realizada por Noam Chomsky y con el estudio de algunos métodos para simplificar ciertas gramáticas.

2.1. Definiciones básicas

A continuación se incluyen las definiciones de conceptos elementales que se utilizarán a lo largo de toda la asignatura.

Alfabeto: conjunto no vacío y finito de símbolos. A estos símbolos también se les suele llamar *letras* del alfabeto. Se denota con la letra griega Σ. Ejemplos:

$\Sigma_1 = \{a,b,c,...,z\}$ $\Sigma_2 = \{0,1\}$

Palabra: secuencia finita de símbolos de un alfabeto. Lo correcto es hablar de "*palabras definidas sobre un alfabeto*". Habitualmente utilizaremos en nuestros ejemplos las últimas letras minúsculas de nuestro alfabeto (x, y, z) para denotar a las palabras. Ejemplos:

x = *casa* es una palabra definida sobre el alfabeto Σ_1

y = *010100* es una palabra definida sobre el alfabeto Σ_2

Palabra vacía: es una palabra que no tiene ningún símbolo y se representa como λ.

Longitud de una palabra: es el número de símbolos que componen la palabra. Se representa utilizando dos barras verticales(||). Ejemplos:

$|x| = 4$ $|y| = 6$ $|\lambda| = 0$

Lenguaje Universal definido sobre un alfabeto es el conjunto de todas las palabras que se pueden construir con las letras de dicho alfabeto. Se denota por $\omega(\Sigma)$. El lenguaje universal de cualquier alfabeto es infinito, y siempre pertenece a él la palabra vacía.

Ejemplo: si $\Sigma = \{a\}$ entonces $\omega(\Sigma) = \{\lambda, a, aa, aaa, ...\}$

Lenguaje L definido sobre un alfabeto Σ, es un conjunto cualquiera de palabras definidas sobre dicho alfabeto, por lo tanto, $L \subset \omega(\Sigma)$.

2.2. Operaciones con palabras

En este apartado se presenta una colección de operaciones que se pueden realizar con palabras y las propiedades que cumplen estas operaciones.

1. Concatenación Sean **x** e **y** dos palabras, se concatenan para formar otra palabra que se denota **xy** y que está formada por todas las letras de **x** seguidas por las letras de **y**.

Ejemplo: x= casa y=blanca ⇒xy = casablanca

Propiedades:

- Operación cerrada. Si x e y están definidas sobre el mismo alfabeto, xy también lo estará.
- Asociativa. $(xy)z = x(yz)$
- Elemento neutro (λ). $x\lambda = \lambda x = x$

- $|xy| = |x| + |y|$
- No es una operación conmutativa

2. Potencia i-esima de una palabra (x^i) Consiste en concatenar una palabra **x** consigo misma, **i** veces. $x^i = x \ldots_i x$

Ejemplo: $x = la \Rightarrow x^4 = lalalala$

Propiedades:

- $x^{i+j} = x^i x^j$
- $|x^i| = i|x|$
- $x^0 = \lambda$

3. Reflexión (o inversa) de una palabra (x^{-1}) Es otra palabra definida sobre el mismo alfabeto y formada por los mismos símbolos que x, dispuestos en orden inverso.

Ejemplo: $x = casa \Rightarrow x^{-1} = asac$

Propiedad: $|x| = |x^{-1}|$

2.3. Lenguajes y operaciones con lenguajes

Como hemos visto anteriormente un lenguaje definido sobre un alfabeto no es más que un subconjunto del lenguaje universal de ese alfabeto. Por ejemplo, si $\Sigma = \{0,1\}$, podemos definir diferentes lenguajes sobre ese alfabeto:

$L_1 = \{x/|x| = 4\}$ $\quad L_2 = \{0^n 1^n / n > 0\}$

$L_3 = \{x/x$ no contenga un número par de 0's $\}$

Sobre cualquier alfabeto se pueden definir lenguajes *especiales* como el lenguaje vacío, que se representa como $L_\emptyset = \emptyset$ y que no tiene ninguna palabra. También existe el lenguaje que contiene solamente la palabra vacía $L_\lambda = \{\lambda\}$

Se presentan, a continuación, diferentes operaciones que se pueden definir sobre los lenguajes. Casi todas están basadas en las operaciones sobre palabras que se explicaron en la sección anterior.

1. Unión de lenguajes La unión de dos lenguajes L_1 *y* L_2 definidos sobre el mismo alfabeto Σ es otro lenguaje, también definido sobre ese alfabeto, que contiene todas las palabras de L_1 y todas las de L_2.

$$L = L_1 \cup L_2 = \{x/x \in L_1 \vee x \in L_2\}$$

Propiedades:

- Operación cerrada. El lenguaje resultante está definido sobre el mismo alfabeto que L_1 y L_2.
- Asociativa. $(L_1 \cup L_2) \cup L_3 = L_1 \cup (L_2 \cup L_3)$
- Conmutativa. $L_1 \cup L_2 = L_2 \cup L_1$
- Elemento Neutro **(Ø)**. $L \cup \emptyset = \emptyset \cup L = L$
- Idempotencia. $L \cup L = L$

2. Concatenación Sean dos lenguajes L_1 *y* L_2 definidos sobre el mismo alfabeto Σ, la concatenación de ambos lenguajes estará formada por todas las palabras obtenidas al concatenar una palabra cualquiera de L_1 con otra de L_2.

$$L = L_1 L_2 = \{xy / x \in L_1 \ \wedge \ y \in L_2\}$$

Propiedades:

- Operación cerrada. El lenguaje resultante está definido sobre el mismo alfabeto que L_1 y L_2.
- Asociativa. $(L_1\, L_2)\, L_3 = L_1\, (L_2\, L_3)$
- Elemento Neutro $(L_\lambda = \{\lambda\})$. $L_\lambda L \ = \ L L_\lambda = L$
- No es conmutativa.

3. Potencia i-esima Es el resultado de concatenar un lenguaje consigo mismo un número **i** de veces. $L^i = L \ldots_i L$

Propiedades:

- $L^{i+j} = L^i L^j$
- $L^0 = L_\lambda = \{\lambda\}$

4. Clausura (o cierre de Kleene) La clausura de un lenguaje (L^*) es el resultado de unir todas las potencias de dicho lenguaje, es decir,

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

5. Clausura positiva La clausura positiva de un lenguaje (L^+) es la unión de todas las potencias de ese lenguaje, exceptuando la potencia cero.

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

Propiedades:

- $L^* = L^+ \cup \{\lambda\}$
- $L^+ = L^*L = LL^*$
- $\omega(\Sigma) = \Sigma^*$

  En este caso el alfabeto es considerado como un lenguaje, concretamente el formado por todas las cadenas de longitud 1.
- $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$

6. Reflexión La reflexión de un lenguaje (L^{-1}) está formada por las inversas de todas las palabras de ese lenguaje. $L^{-1} = \{x^{-1}/x \in L\}$

2.4. Concepto de gramática formal

Si nos referimos a los lenguajes naturales el concepto de gramática es muy antiguo. Los primeros trabajos aparecen en la India durante los comienzos del primer milenio antes de Cristo, alcanzándose el máximo apogeo con Panini (siglos VII y VI a. C.). Al mismo tiempo en Grecia se desarrollaba una corriente de investigación gramatical, cuyo máximo representante sería Pitágoras. Sin embargo, el concepto de gramática desde un punto de vista formal tiene su origen en los trabajos de Chomsky a mediados del siglo XX.

2.4.1. Definiciones previas

Comenzaremos este apartado con una serie de definiciones cuyo interés práctico quedará de manifiesto con los ejemplos que aparecen a continuación.

Definición 2.1 (Producción)
Sea Σ un alfabeto, llamamos producción (o regla) definida sobre ese alfabeto a un par ordenado de palabras (x, y) donde $x, y \in \Sigma^*$. Se dice que x es la parte izquierda *de la producción y que y es la* parte derecha. *A las producciones también se las llama* reglas de derivación. *Se representa $x ::= y$.*

Definición 2.2 (Producción compresora)
Se dice que una producción es compresora si la longitud de su parte derecha es menor que la de la parte izquierda.

Definición 2.3 (Derivación directa)
Sea Σ un alfabeto, P un conjunto de producciones definidas sobre ese alfabeto

$$P = \left\{ \begin{array}{c} x_1 ::= y_1 \\ x_2 ::= y_2 \\ \cdots \\ x_n ::= y_n \end{array} \right\}$$

y $v, w \in \Sigma^*$.

Decimos que v produce directamente a w, o que w deriva directamente de v, si $\exists z, u \in \Sigma^*$ y una producción $x_i ::= y_i$ tal que $v = zx_iu$ y $w = zy_iu$

La notación utilizada para representar una derivación directa es $v \rightarrow w$

Definición 2.4 (Derivación)
Sea Σ un alfabeto, P un conjunto de producciones definidas sobre ese alfabeto y $v, w \in \Sigma^*$.

Decimos que v produce a w, o que w deriva de v, si $\exists w_0, w_1, \ldots, w_m \in \Sigma^*$ tales que

$$\begin{array}{rcl} v = w_0 & \rightarrow & w_1 \\ w_1 & \rightarrow & w_2 \\ & \cdots & \\ w_{m-1} & \rightarrow & w_m = w \end{array}$$

La notación utilizada en este caso es $v \stackrel{*}{\rightarrow} w$

2.4.2. Ejemplo con un lenguaje natural(castellano)

Estamos familiarizados con el concepto tradicional de gramática que, de forma intuitiva, podríamos considerar como un conjunto de reglas que nos indican qué es correcto y qué no lo es en un lenguaje natural. Con el fin de acercarnos a una definición más formal comenzaremos con un ejemplo en lengua castellana.

La gramática debe describir la estructura de las frases y de las palabras de un lenguaje. Veamos una serie de reglas muy sencillas que nos permitirían comprobar que la frase "*el perro corre deprisa*" es correcta.

Reglas gramaticales:

1. $<$ sentencia $>::=<$ sujeto $><$ predicado $>$
2. $<$ sujeto $>::=<$ articulo $><$ nombre $>$
3. $<$ predicado $>::=<$ verbo $><$ complemento $>$
4. $<$ predicado $>::=<$ verbo $>$
5. $<$ articulo $>$ ::= *el*
6. $<$ articulo $>$ ::= *la*
7. $<$ nombre $>$ ::= *perro*
8. $<$ nombre $>$ ::= *gata*
9. $<$ verbo $>$ ::= *corre*
10. $<$ verbo $>$ ::= *come*
11. $<$ complemento $>$ ::= *deprisa*
12. $<$ complemento $>$ ::= *mucho*

Estas reglas pueden ser consideradas como un conjunto de producciones. Si utilizamos algunas de estas producciones para llevar a cabo derivaciones a partir del item $<$ sentencia $>$ podemos llegar a obtener frases como: "*el perro corre deprisa*", "*la gata come mucho*" o "*la gata corre*". Sin embargo, nunca podríamos llegar a construir la frase "*mucho deprisa perro*".

Veamos, paso a paso, como se podría generar la frase "la gata corre" a partir del símbolo $<$ *sentencia* $>$. En cada fase del proceso hemos destacado en negrita el símbolo que se transforma.

Aplicando la pr. 1 $<$ sentencia $>\longrightarrow <$ **sujeto** $><$ predicado $>$
Aplicando la pr. 2 $<$ sentencia $>\xrightarrow{*} <$ articulo $><$ nombre $><$ **predicado** $>$
Aplicando la pr. 4 $<$ sentencia $>\xrightarrow{*} <$ **articulo** $><$ nombre $><$ verbo $>$
Aplicando la pr. 6 $<$ sentencia $>\xrightarrow{*}$ la$<$ **nombre** $><$ verbo $>$
Aplicando la pr. 8 $<$ sentencia $>\xrightarrow{*}$ la gata$<$ **verbo** $>$
Aplicando la pr. 9 $<$ sentencia $>\xrightarrow{*}$ la gata corre

Sin embargo, la forma más habitual de representar este mismo proceso de generación de una cadena de símbolos es mediante un árbol de derivaciones (o árbol *parser*) como el que se muestra en la figura 2.1.

2.4.3. Ejemplo en un lenguaje artificial

Aplicaremos el mismo método para definir un fragmento de un lenguaje de programación. Pretendemos describir cómo son las instrucciones que permiten asignar el valor de una expresión a una variable en un lenguaje como C.

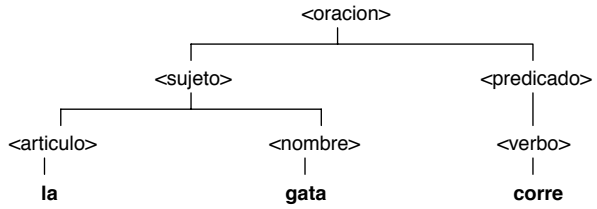


Figura 2.1: Árbol de derivación

1. $< asignacion > ::= < variable > \ '=' < expresion >$
2. $< expresion > ::= < numero >$
3. $< expresion > ::= < variable >$
4. $< expresion > ::= < expresion > \ '+' \ < expresion >$
5. $< expresion > ::= < expresion > \ '*' \ < expresion >$

Si consideramos que A, B y C pueden ser considerados como $< variable >$ y que 2 y 4 pueden ser considerados como $< numero >$, es fácil comprobar que a partir del símbolo $< asignacion >$ y utilizando diferentes producciones podemos llegar a construir instrucciones como:

$A = B + C$

$B = B * 2$

$C = C + 4$

Sin embargo, no podríamos construir sentencias como $A = + 2 * / + 4$ o $4 = A$

Es decir, en los ejemplos anteriores podemos ver que hay construcciones gramaticalmente correctas y otras que no lo son.

2.4.4. Definición de gramática formal

Analizando los ejemplos anteriores podemos observar como el objetivo es llegar a tener una secuencia correcta de símbolos (en el primer ejemplo, estos símbolos son: el, la, perro, gata, etc. y en el segundo, los símbolos son: A, B, *,+, 2, etc.) partiendo de un determinado símbolo, que llamaremos inicial, ($< oracion >$ en el primer caso o $< asignacion >$ en el segundo), y utilizando algunas de las producciones definidas. A partir de estas ideas intuitivas, formalizaremos la definición de gramática.

Definición 2.5 (Gramática Formal)
Se llama gramática formal definida sobre un alfabeto Σ *a una tupla de la forma* $G = \{\Sigma_T, \Sigma_N, S, P\}$ *donde:*

•$\mathbf{\Sigma_T}$ *es el alfabeto de símbolos terminales*

•$\mathbf{\Sigma_N}$ *es el alfabeto de símbolos no terminales (aparecen en los ejemplos encerrados entre <>)*

•**S** *es el símbolo inicial de la gramática*

•**P** *es un conjunto de producciones gramaticales*

Hay que tener en cuenta que:

- $S \in \Sigma_N$
- $\Sigma_T \cap \Sigma_N = \emptyset$
- $\Sigma = \Sigma_T \cup \Sigma_N$

Ejemplo 2.1 $\Sigma_T = \{+, -, 0, 1, 2, \ldots, 9\}$
$\Sigma_N = \{< Signo >, < Digitos >, < Numero >, < Caracter >\}$
$S =< Numero >$

$$P = \left\{ \begin{array}{l} < Numero > ::= < Signo >< Digito > \\ < Signo > ::= + \\ < Signo > ::= - \\ < Digito > ::= < Caracter >< Digito > \\ < Digito > ::= < Caracter > \\ < Caracter > ::= 0 \\ < Caracter > ::= 1 \\ \qquad\qquad \ldots \\ < Caracter > ::= 9 \end{array} \right\}$$

Con esta gramática, y a partir del símbolo $< Numero >$, podemos generar cualquier número natural, siempre que vaya precedido por un signo. Por ejemplo: -57, +5, -4999.

Hasta este momento hemos distinguido los símbolos no terminales de los terminales encerrando a los primeros entre <>. Sin embargo, en los ejemplos que veremos a partir de ahora y por simplicidad, utilizaremos las letras mayúsculas para representar a los símbolos no terminales y las minúsculas para los terminales. Además, utilizaremos la notación BNF (Backus Normal Form). Con esta notación se utilizan los símbolos **::=** para separar la parte izquierda de una producción de la derecha y además, se emplea el símbolo | para indicar que la parte izquierda de una producción coincide con la de la anterior, así no se repite la parte izquierda de determinadas producciones. Por tanto, en el ejemplo anterior la descripción de las producciones quedaría así:

$$P = \left\{ \begin{array}{l} < Numero > ::= < Signo >< Digito > \\ < Signo > ::= + \\ \qquad\qquad | - \\ < Digito > ::= < Caracter >< Digito > \\ \qquad\qquad | < Caracter > \\ < Caracter > ::= 0 \\ \qquad\qquad | 1 \\ \qquad\qquad | \cdots \\ \qquad\qquad | 9 \end{array} \right\}$$

Definición 2.6 (Lenguaje generado por una gramática)
Sea una gramática definida como $G = \{\Sigma_T, \Sigma_N, S, P\}$ *llamamos lenguaje generado por dicha gramática a* $L = \{x \in \Sigma_T^* / S \xrightarrow{*} x\}$

Por lo tanto, las palabras del lenguaje estarán formadas por cadenas de símbolos terminales generadas a partir del símbolo inicial de la gramática, utilizando las producciones que la definen.

Definición 2.7 (Recursividad)
Una gramática es recursiva si tiene alguna derivación recursiva, es decir, si $A \xrightarrow{*} xAy$ *donde* $A \in \Sigma_N,\ x, y \in \Sigma^*$. *Si* $x = \lambda$ *se dice que la gramática es recursiva por la izquierda, y si* $y = \lambda$ *se dice que es recursiva por la derecha.*

Es evidente que si una gramática tiene producciones recursivas, es decir producciones con la forma $A ::= xAy$, entonces es recursiva.

Teorema 2.1
Un lenguaje es infinito si y sólo si existe una gramática recursiva que lo genera.

2.5. Clasificación de las gr. formales

Noam Chomsky clasificó las gramáticas en cuatro grupos (G_0, G_1, G_2, G_3), donde cada uno contiene al siguiente. La diferencia entre cada grupo se centra en la forma de las producciones. La misma clasificación puede ser aplicada a los lenguajes, es decir, los lenguajes de tipo 0 son los generados por las gramáticas de tipo 0 y así sucesivamente.

2.5.1. Gramáticas de tipo 0

También se las llama gramáticas sin restricciones o gramáticas **recursivamente enumerables**.

Las producciones de las gramáticas de tipo 0 tienen la forma:

$$xAy ::= v \text{ donde } A \in \Sigma_N,\ x, y, v \in \Sigma^*$$

Sin embargo, es posible demostrar que cualquier lenguaje de tipo 0 puede ser también generado por una gramática que pertenece a un grupo algo más restringido, las *gramáticas con estructura de frase*. Podemos decir, por tanto, que las gr. de tipo 0 y las gr. con estructura de frase tienen el mismo poder generativo.

Las producciones de las gramáticas con estructura de frase tienen la forma:

$$xAy ::= xvy \text{ donde } A \in \Sigma_N,\ x, y, v \in \Sigma^*$$

2.5.2. Gramáticas de tipo 1

A este tipo de gramáticas también se las llama gramáticas **dependientes del contexto**. Las producciones deben tener la siguiente forma:

$$xAy ::= xvy \text{ donde } A \in \Sigma_N,\ v \in \Sigma^+\ x, y \in \Sigma^*$$

Es obvio que todas las gramáticas de tipo 1 son también gramáticas con estructura de frase, pero en este caso hay una restricción añadida y es que la longitud de la parte derecha de las producciones es siempre mayor o igual que la de la parte izquierda, es decir, no hay producciones compresoras.

El nombre de gramáticas dependientes del contexto se debe a que las producciones se pueden interpretar como que "A se convierte en v, siempre que se encuentre en un determinado contexto, es decir, precedida por x y seguida por y". Por lo tanto, es necesario conocer el contexto en el que se encuentra A para poder aplicar la producción.

2.5.3. Gramáticas de tipo 2

Son también llamadas gramáticas **independientes del contexto**. Sus producciones son aún más restrictivas. En este caso, la parte izquierda de la producción está formada por un único símbolo no terminal. Por lo tanto, las producciones son de la forma:

$$A ::= v \ \text{ donde } \ A \in \Sigma_N \quad v \in \Sigma^*$$

En este tipo de gramáticas, la conversión de A en v se realiza independientemente del contexto en el que se encuentre A, de ahí su nombre. Son especialmente adecuadas para representar los aspectos sintácticos de cualquier lenguaje de programación.

2.5.4. Gramáticas de tipo 3

Es el grupo más restringido de gramáticas y también son llamadas **regulares**. En este caso también se le imponen restricciones a la parte derecha de las producciones, que tendrán como máximo dos símbolos. Hay dos tipos de gramáticas regulares y sus producciones pueden ser de la siguiente forma:

1. Para las gramáticas **lineales por la derecha (GLD)**

   a) $A ::= a$

   b) $A ::= aV$ donde $A, V \in \Sigma_N,\ a \in \Sigma_T$

   c) $S ::= \lambda$ y S es el símbolo inicial de la gramática.

2. Para las gramáticas **lineales por la izquierda (GLI)**

   a) $A ::= a$

   b) $A ::= Va$ donde $A, V \in \Sigma_N,\ a \in \Sigma_T$

   c) $S ::= \lambda$ y S es el símbolo inicial de la gramática.

Cualquier lenguaje de tipo 3 puede ser generado tanto por una gramática lineal por la derecha como por una lineal por la izquierda. Es decir, estos dos grupos de gramáticas tienen el mismo poder generativo.

2.6. Gramáticas equivalentes

A continuación veremos como, en ocasiones, es recomendable simplificar ciertas gramáticas, eliminando símbolos o producciones no deseadas. En estos casos, el objetivo será llegar a definir una gramática equivalente a la primera pero que no tenga esos elementos *indeseables*. En este apartado trabajaremos exclusivamente con gramáticas independientes del contexto.

Definición 2.8 (Gramáticas equivalentes)
Dos gramáticas son equivalentes cuando generan el mismo lenguaje. Es evidente que, para que esto suceda, deben estar definidas sobre el mismo Σ_T.

2.6.1. Simplificación de gramáticas

Comenzaremos definiendo los elementos *indeseables* de los que hablábamos anteriormente.

Definición 2.9 (Reglas innecesarias)
Son produciones de la forma $A ::= A$. Evidentemente no aportan información a la gramática.

Definición 2.10 (Símbolos inaccesibles)
Son símbolos no terminales que no aparecen en ninguna cadena de símbolos que pueda derivarse a partir del símbolo inicial de la gramática.

Definición 2.11 (Símbolos no generativos)
Son símbolos no terminales a partir de los cuales no puede derivarse ninguna cadena de símbolos terminales.

Veamos a continuación métodos para localizar los símbolos inaccesibles y los no generativos.

Método para localizar los símbolos inaccesibles de una gramática En realidad, el método trata de identificar los símbolos accesibles, así el resto serán símbolos inaccesibles. Para conseguirlo se puede diseñar un algoritmo que construya de forma incremental este conjunto de símbolos accesibles.

Inicialmente cualquier símbolo que aparezca en la parte derecha de una producción que tiene a S en la parte izquierda, es un símbolo accesible. A partir de aquí, si un símbolo está en la parte derecha de una producción que tiene a un símbolo accesible en la parte izquierda es también accesible.

Veamos un algoritmo para llevar a cabo esta tarea:

Algoritmo 2.1 Búsqueda Símbolos inaccesibles

Output: SimInacc $\subset \Sigma_N$
Begin
SimAccesibles = $\{V \in \Sigma_N / \exists\ S ::= xVy,\ x, y \in \Sigma^*\}$
Auxiliar = $\emptyset$
while Auxiliar $\neq$ SimAccesibles **do**
Auxiliar = SimAccesibles
SimAccesibles = SimAccesibles $\cup$ $\{V \in \Sigma_N / \exists\ A ::= xVy,\ x, y \in \Sigma^*\ A \in$ *SimAccesibles*$\}$
end while
SimInacc = $\Sigma_N \setminus$ SimAccesibles.
End

Método para localizar los símbolos no generativos El método es análogo al anterior, es decir, tiene como objetivo localizar, en primer lugar, los símbolos generativos. Inicialmente son símbolos generativos aquellos que aparecen en la parte

izquierda de una producción que tiene sólo símbolos terminales o la cadena nula en la parte derecha. Veamos el algoritmo:

Algoritmo 2.2 Búsqueda Símbolos No Generativos

Output: SimNoGen $\subset \Sigma_N$

Begin

SimGenerativos = $\{V \in \Sigma_N / \exists\, V ::= \alpha,\ \alpha \in \Sigma_T^*\}$

Auxiliar = $\emptyset$

while Auxiliar $\neq$ SimGenerativos **do**

Auxiliar = SimGenerativos

SimGenerativos = SimGenerativos $\cup$ $\{V \in \Sigma_N / \exists\ V ::= \alpha,\ \alpha \in (\Sigma_T \cup SimGenerativos)^*\}$

end while

SimNoGen = $\Sigma_N \setminus$ SimGenerativos

End

Tanto las reglas innecesarias como los símbolos no generativos o los inaccesibles pueden eliminarse de cualquier gramática, ya que no aportan información relevante a la misma.

Definición 2.12 (Gramática Limpia)
Decimos que una gramática es limpia si no tiene reglas innecesarias, ni símbolos no generativos, ni símbolos inaccesibles.

Definición 2.13 (Reglas de redenominación)
Son reglas en las que hay un único símbolo no terminal tanto en la parte izquierda de la producción como en la derecha. Es decir, tienen la forma:

$A ::= B, \quad A, B \in \Sigma_N$

Para eliminar las reglas de redenominación de una gramática es necesario sustituirlas por otras producciones que sean equivalentes.

Por ejemplo, si tenemos las producciones

A::=B

B::=x

B::=y donde x,y $\in \Sigma^*$

y deseamos eliminar A::=B, las producciones anteriores deben ser sustituidas por las siguientes producciones:

A::=x

A::=y

B::=x

B::=y

Definición 2.14 (Reglas no generativas)
Son aquellas en las que sólo aparece λ en la parte derecha de la producción.

Para eliminar estas reglas también es necesario añadir otras a la gramática.

Si queremos eliminar la producción $A ::= \lambda$ es necesario localizar las producciones que tiene a A en la parte derecha (por ejemplo: $B ::= xAy$) y añadir para cada producción de este tipo otra equivalente en la que no aparece A (en este caso, $B ::= xy$).

La producción $S ::= \lambda$ no puede eliminarse de ninguna gramática ya que es imprescindible si se pretende que el lenguaje generado contenga la palabra nula.

Definición 2.15 (Gramática bien formada)
Decimos que una gramática está bien formada si no tiene reglas de redenominación ni reglas no generativas.

2.7. Problemas y cuestiones

2.1 Dado el alfabeto $\Sigma = \{a, b\}$, ¿cuántas palabras hay en el lenguaje $A^n B$ y cómo son dichas palabras?, considerando los siguientes casos:

1. $A = \{a\} \qquad B = \{b\}$
2. $A = \{a\} \qquad B = \{b, \lambda\}$
3. $A = \{a, \lambda\} \qquad B = \{b, \lambda\}$
4. $A = \{a, \lambda\} \qquad B = \{b\}$

2.2 Dado el alfabeto $\Sigma = \{a, b\}$, y el lenguaje definido sobre él, $L = \{aa, bb\}$ ¿cómo son las palabras del lenguaje L^4?

2.3 ¿En qué situación se cumple que $L^* = L^+$?

2.4 Dadas las siguientes gramáticas, indicar de qué tipo son y cómo es el lenguaje que generan:

1. $\Sigma_T = \{a, b, c, 0, 1\} \quad \Sigma_N = \{S\}$

$$P = \left\{ \begin{array}{l} S ::= a \\ \quad\quad |b \\ \quad\quad |c \\ \quad\quad |Sa \\ \quad\quad |Sb \\ \quad\quad |Sc \\ \quad\quad |S0 \\ \quad\quad |S1 \end{array} \right\}$$

2. $\Sigma_T = \{a, b\} \quad \Sigma_N = \{S, A\}$

$$P = \left\{ \begin{array}{c} S ::= A \\ \quad |\lambda \\ A ::= aAb \\ \quad |ab \end{array} \right\}$$

3. $\Sigma_T = \{a, b\} \quad \Sigma_N = \{S, A\}$

$$P = \left\{ \begin{array}{c} S ::= A \\ \quad |\lambda \\ A ::= aA \\ \quad |Ab \\ \quad |a \\ \quad |b \end{array} \right\}$$

4. $\Sigma_T = \{a\} \quad \Sigma_N = \{S, A\}$

$$P = \left\{ \begin{array}{c} S ::= A \\ \quad |\lambda \\ A ::= AaA \\ \quad |a \end{array} \right\}$$

2.5 Dados los siguientes lenguajes, diseñar una gramática que los genere

1. $L_1 = \{ab^n a / \ n = 0, 1, \ldots\}$
2. $L_2 = \{a^m b^n / \ m \geq n \geq 0\}$
3. $L_3 = \{a^k b^m a^n / \ n = k + m\}$
4. $L_4 = \{waw^{-1} / \ w$ es una cadena binaria definida en el alfabeto $\{0, 1\}\}$

2.6 Dadas las siguientes gramáticas, obtener gramáticas equivalentes a ellas que sean limpias y bien formadas

1. $\Sigma_T = \{a, b\} \quad \Sigma_N = \{S, A, B, C, D, E\}$

$$P = \left\{ \begin{array}{c} S ::= Aa \\ \quad |Ca \\ \quad |a \\ B ::= Aa \\ \quad |Ca \\ \quad |a \\ C ::= Bb \\ D ::= Ca \\ E ::= Cb \end{array} \right\}$$

2. $\Sigma_T = \{x, y, z\} \quad \Sigma_N = \{S, P, Q\}$

$$P = \left\{ \begin{array}{c} S ::= zPzQz \\ P ::= xPx \\ \quad |Q \\ Q ::= yPy \\ \quad |\lambda \end{array} \right\}$$

Tema 3

Expresiones y gramáticas regulares

Contenido

3.1. Definición de expresión regular 33
3.2. Álgebra de las expresiones regulares 34
3.3. Definición de gramática regular 35
3.4. Ejemplos 36
3.5. Problemas 38

En este tema se van a estudiar las gramáticas regulares, también llamadas de tipo 3 atendiendo a la clasificación de Chomsky. También se aborda el estudio de las expresiones regulares que permiten definir de forma precisa los lenguajes generados por estas gramáticas (lenguajes regulares).

3.1. Definición de expresión regular

Una expresión regular es una notación normalizada para representar lenguajes regulares, es decir, lenguajes generados por gramáticas de tipo 3. Como veremos, las expresiones regulares permiten describir con exactitud y sencillez cualquier lenguaje regular. Para definir una expresión regular(e.r.) se pueden utilizar todos los símbolos del alfabeto Σ y, además, λ y $\emptyset$. Los operadores que también se pueden utilizar son:

- $+$ representa la unión
- **.** representa la concatenación (este símbolo no se suele escribir)
- * representa el cierre de Kleene
- ( ) modifican las prioridades de los demás operadores

Una expresión regular se puede definir de acuerdo a los siguientes criterios:

1. $\emptyset$ es una e.r. que representa al lenguaje vacío (no tiene palabras) $L_\emptyset = \emptyset$
2. λ es una e.r. que representa al lenguaje $L_\lambda = \{\lambda\}$
3. $a \in \Sigma$ es una e.r. que representa al lenguaje $L_a = \{a\}$
4. Si α y β son e.r. entonces $\alpha + \beta$ también lo es y representa al lenguaje $L_{\alpha+\beta} = L_\alpha \cup L_\beta$
5. Si α y β son e.r. entonces $\alpha\beta$ también lo es y representa al lenguaje $L_{\alpha\beta} = L_\alpha L_\beta$
6. Si α es una e.r. entonces α^* también lo es y representa al lenguaje $L_{\alpha^*} = \bigcup_{i=0}^{\infty} L_\alpha^i$ que también se puede representar $\alpha^* = \Sigma_{i=0}^{\infty}\alpha^i$

Sólo son e.r. aquellas que puedan ser definidas utilizando los 6 puntos vistos anteriormente.

La prioridad de las operaciones, que puede modificarse utilizando paréntesis, es de mayor a menor: $*$. $+$

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

1. $01 + 001$ es una e.r. que representa al lenguaje $L = \{01, 001\}$
2. 0^*10^* es una e.r. que representa a cualquier cadena binaria en la que hay un solo 1, $L = \{0^n10^m / n, m \geq 0\}$

   Sea $\Sigma = \{a, b, c\}$
3. $a(a + b + c)^*$ representa a cualquier cadena que empiece por a
4. $(a + b + c)^*$ representa al lenguaje universal definido sobre el alfabeto

3.2. Álgebra de las expresiones regulares

Propiedades de la unión (+)

1. Asociativa $(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma)$
2. Conmutativa $\alpha + \beta = \beta + \alpha$
3. Elemento neutro ($\emptyset$) $\alpha + \emptyset = \alpha$
4. Idempotencia $\alpha + \alpha = \alpha$

Propiedades de la concatenación (.)

1. Asociativa $(\alpha\beta)\gamma = \alpha(\beta\gamma)$
2. Distributiva respecto de la unión $\alpha(\beta + \gamma) = \alpha\beta + \alpha\gamma$
3. Elemento neutro (λ) $\alpha\lambda = \lambda\alpha = \alpha$
4. $\alpha\emptyset = \emptyset$
5. No es conmutativa

Propiedades del cierre de Kleene (*)

1. $\lambda^* = \lambda$
2. $\emptyset^* = \lambda$
3. $\alpha^*\alpha^* = \alpha^*$
4. $\alpha^*\alpha = \alpha\alpha^*$
5. $(\alpha^*)^* = \alpha^*$
6. $\alpha^* = \lambda + \alpha\alpha^*$
7. $\alpha^* = \lambda + \alpha + \alpha^2 + \ldots + \alpha^n + \alpha^{n+1}\alpha^*$

   Si tenemos una función $f : E_\Sigma^n \longrightarrow E_\Sigma$ (por ejemplo $f(\alpha, \beta) = \alpha^*\beta$), donde E_Σ representa al conjunto de las expresiones regulares definidas sobre Σ, entonces:
8. $f(\alpha, \beta, \gamma, \ldots) + (\alpha + \beta + \gamma + \ldots)^* = (\alpha + \beta + \gamma + \ldots)^*$
9. $(f(\alpha^*, \beta^*, \gamma^*, \ldots))^* = (\alpha + \beta + \gamma + \ldots)^*$

3.3. Definición de gramática regular

Como vimos en el tema anterior existen dos tipos de gramáticas de tipo 3: las gramáticas lineales por la derecha y las lineales por la izquierda.

Las producciones de las gramáticas lineales por la derecha pueden ser de la forma:

1. $A ::= a$
2. $A ::= aV$ donde $A, V \in \Sigma_N,\ a \in \Sigma_T$
3. $S ::= \lambda$ y S es el símbolo inicial de la gramática.

Las producciones de las gramáticas lineales por la izquierda pueden ser de la forma:

1. $A ::= a$
2. $A ::= Va$ donde $A, V \in \Sigma_N,\ a \in \Sigma_T$
3. $S ::= \lambda$ y S es el símbolo inicial de la gramática.

Estos dos tipos de gramáticas tienen el mismo poder generativo, es decir, dada una gramática lineal por la izquierda siempre existe una gramática lineal por la derecha que es equivalente a ella y viceversa. Además, dado un lenguaje regular siempre existen, al menos, una gramática lineal por la izquierda y una gramática lineal por la derecha que lo generan.

Todo lenguaje de tipo 3 puede representarse mediante una e.r. y una e.r. siempre representa a un lenguaje de tipo 3.

3.4. Ejemplos

Supongamos que $\Sigma_1 = \{a, b, \ldots, z\}$

1. El lenguaje $L_1 = \{\lambda, a, aa, aaa, \ldots\}$ puede representarse con la e.r. a^* y puede ser generado por la gr. lineal por la izquierda

$$P = \{S ::= \lambda | Sa\}$$

y por la gr. lineal por la derecha

$$P = \{S ::= \lambda | aS\}$$

2. El lenguaje de todas las palabras que empiezan por a puede representarse con la e.r. $a(a + b + \ldots + z)^*$ y puede ser generado por la gr. lineal por la izquierda

$$P = \left\{ \begin{array}{l} S ::= a \\ \quad\ \ |Sa \\ \quad\ \ |Sb \\ \quad\ \ \ldots \\ \quad\ \ |Sz \end{array} \right\}$$

y por la gr. lineal por la derecha

$$P = \left\{ \begin{array}{l} S ::= aA \\ A ::= aA \\ \quad\ \ |bA \\ \quad\ \ \ldots \\ \quad\ \ |zA \\ \quad\ \ |\lambda \end{array} \right\}$$

3. El lenguaje de las palabras que empiezan por a, terminan por c y además de estas dos letras sólo pueden tener b´s (tantas como se quieran) puede representarse con la e.r. ab^*c y puede ser generado por la gr. lineal por la izquierda

$$P = \left\{ \begin{array}{c} S ::= Ac \\ A ::= Ab \\ \quad\quad |a \end{array} \right\}$$

y por la gr. lineal por la derecha

$$P = \left\{ \begin{array}{c} S ::= aA \\ A ::= bA \\ \quad\quad |c \end{array} \right\}$$

Los lenguajes regulares son especialmente adecuados para representar las características léxicas de un lenguaje de programación, como veremos en los siguientes ejemplos en los que consideraremos un lenguaje de programación similar a C.

4. Las cadenas que pueden ser consideradas como un *identificador* del lenguaje (nombres inventados por el programador para definir variables, funciones, etc.) están formadas por letras (mayúsculas y minúsculas), dígitos y el guión bajo (_), pero no pueden comenzar por un dígito. Este lenguaje definido sobre el alfabeto $\Sigma_2 = \{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9, \_\}$ se puede representar por la e.r.

$$(a + \ldots + z + A + \ldots + Z + \_)(a + \ldots + z + A + \ldots + Z + \_ + 0 + \ldots + 9)^*$$

Esta e.r. representa palabras como $Suma$ o $Total1$ pero no representaría a $1Ab$

Si trabajamos con el alfabeto $\Sigma_3 = \{0, \ldots, 9, .,_+ , -, e, E\}$ (el símbolo de la suma $_+$ ha sido representado en un tamaño menor del habitual para distinguirlo con claridad del operador *unión* $(+)$)

5. La e.r. $\alpha = (0 + \ldots + 9)(0 + \ldots + 9)^*$ permite representar a cualquier número natural (por ejemplo: 4, 27 o 256).

6. La e.r. $\beta = \alpha.(0 + \ldots + 9)^*$ representa números reales no negativos con una notación clásica (por ejemplo: 55.7, 854.95 o 5.).

7. La e.r. $\gamma = (\beta(e + E)\alpha) + (\beta(e + E)_+\alpha) + (\beta(e + E) - \alpha)$ representa números reales no negativos con una notación científica (por ejemplo: 5.5e+10).

3.5. Problemas

3.1 Describir los lenguajes representados por las siguientes expresiones regulares definidas sobre el alfabeto $\Sigma = \{a, b, c\}$

1. $(a + b)^*c$
2. $(aa^+)(bb^*)$
3. $(aa^+) + (bb^*)$
4. $a^*b^*c^*$

3.2 Representar, mediante una expresión regular, los siguientes lenguajes

1. Considerando que $\Sigma = \{a\}$,
   a) el lenguaje formado por cadenas de a's de longitud par
   b) el lenguaje formado por cadenas de a's de longitud impar
2. Considerando que $\Sigma = \{a, b\}$, el lenguaje formado por cadenas de a's y b´s, de longitud impar, en las que se van alternando los dos símbolos, es decir, nunca aparece el mismo símbolo dos veces seguidas. Por ejemplo: *abababa* o *bab*

3.3 Dadas las siguientes expresiones regulares escribir, para cada una de ellas, una palabra que pertenezca al lenguaje que la expresión representa y otra que no pertenezca a dicho lenguaje

1. $(1^* + 0^*)(1^* + 0^*)(1^* + 0^*)$
2. $(1 + 0)^*10(1 + 0)^*$
3. $1^*(0 + 10^*)1^*$
4. $10^* + 01^*$

3.4 Simplificar las siguientes expresiones regulares

1. $(a + b + ab + ba)^*$
2. $(a + \lambda)^*$
3. $a(a^*a + a^*) + a^*$
4. $(a + b)^*ba(a + b)^* + a^*b^*$

3.5 Dadas dos expresiones regulares
$\alpha = 0^* + 1^* \qquad \beta = 01^* + 10^* + 1^*0 + (0^*1)^*$
encontrar

1. una palabra que pertenezca a α pero no a β
2. una palabra que pertenezca a β pero no a α
3. una palabra que pertenezca a α y a β
4. una palabra que no pertenezca a α ni a β

Tema 4

Autómatas Finitos

Contenido

4.1. Introducción **39**
4.2. Definición de Autómata Finito Determinista **40**
4.3. Representación de Autómatas **40**
4.4. Los AFD como reconocedores de lenguajes **43**
4.5. Minimización de un AFD **43**
4.6. Autómatas Finitos No Deterministas(AFND) **47**
4.7. Lenguaje aceptado por un AFND **50**
4.8. Simulación de un AFD y AFND **51**
4.9. Paso de un AFND a AFD **52**
4.10. Relación entre AF, gr. y exp. reg. **55**
4.11. Límites para los leng. regulares **65**
4.12. Problemas **70**

4.1. Introducción

Aunque no se puede considerar como una definición correcta de autómata, está muy extendida una idea que confunde el concepto de autómata con el de robot. Por lo tanto, se considera erróneamente que un autómata es una máquina que imita funciones típicas de los seres vivos, sobre todo relacionadas con el movimiento, pudiendo incluso ejecutar ciertas órdenes. En realidad el concepto de autómata es mucho más genérico, ya que podemos considerarlo como un dispositivo que procesa cadenas de símbolos que recibe como entrada, cambiando de estado y produciendo una salida que, en algunos casos, puede estar formada por otra cadena de símbolos.

La teoría de autómatas se ocupa de clasificar y estudiar de modo sistemático diferentes tipos de máquinas abstractas que llevan a cabo un procesamiento secuencial de la información. Dentro del conjunto de las máquinas abstractas que estudiaremos en esta asignatura, los Autómatas Finitos constituyen el grupo de máquinas más sencillas y que, por tanto, tienen un menor poder funcional.

El estudio de los autómatas finitos se utiliza para modelar el comportamiento de dispositivos mecánicos y también de sistemas naturales. Concretamente, permite estudiar procesos que dependen de una *historia*, es decir, sistemas cuyo comportamiento actual depende del pasado. También se aplican en el procesamiento del lenguaje natural, pero en el contexto de esta asignatura su principal aplicación será el reconocimiento de lenguajes regulares (de tipo 3).

4.2. Definición de Autómata Finito Determinista

Los Autómatas Finitos son máquinas teóricas que van cambiando de estado dependiendo de la entrada que reciban. La salida de estos Autómatas está limitada a dos valores: *aceptado* y *no aceptado*, que pueden indicar si la cadena que se ha recibido como entrada es o no válida. Generalmente utilizaremos los Autómatas Finitos para reconocer lenguajes regulares, es decir, una palabra se considerará válida sólo si pertenece a un determinado lenguaje.

Formalmente, un Autómata Finito Determinista (AFD) se define como una tupla

$$AFD = (\Sigma, Q, f, q_0, F), \text{ donde}$$

Σ es el alfabeto de entrada

Q es el conjunto finito y no vacío de los estados del Autómata

f es la función de transición que indica en qué situaciones el Autómata pasa de un estado a otro, se define $f : Q \times \Sigma \longrightarrow Q$

$q_0 \in Q$ es el estado inicial

$F \subset Q$ es el conjunto de estados finales de aceptación ($F \neq \emptyset$)

4.3. Representación de Autómatas

Existen dos formas de representar un AFD, mediante **tablas de transición** o mediante **diagramas de transición**. Introduciremos estas dos representaciones con un ejemplo.

Sea el siguiente AFD: $\Sigma = \{a, b\} \quad Q = \{p, q, r\} \quad q_0 = p \quad F = \{q\}$

donde f se define de la siguiente forma:

$f(p, a) = q$ $\qquad$ $f(p, b) = r$

$f(q, a) = q$ $\qquad$ $f(q, b) = r$

$f(r, a) = r$ $\qquad$ $f(r, b) = r$

Tabla de transición El AFD se representaría mediante la siguiente tabla que representa los valores de la función de transición.

		a	b
$\rightarrow$	p	q	r
$*$	q	q	r
	r	r	r

La flecha indica que p es el estado inicial, y el asterisco indica que q es un estado final de aceptación (en general, pueden aparecer muchos asteriscos aunque sólo puede aparecer una flecha ya que sólo hay un estado inicial).

Diagrama de transición La figura 4.1 representa de forma gráfica las transiciones del autómata. Los estados finales de aceptación se identifican por estar encerrados en un doble circulo. El estado inicial se destaca con una flecha *arrugada*.

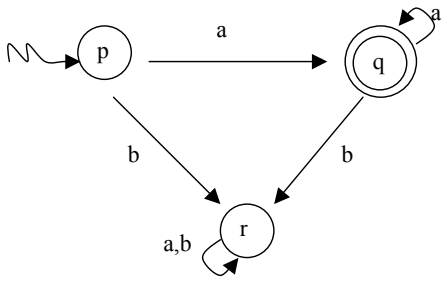


Figura 4.1: Ejemplo de AFD

Al analizar el autómata del ejemplo es evidente que sólo considera como cadenas aceptadas aquellas que están formadas solamente por **a**'s. Cualquier cadena que contenga una **b** hará que el autómata acabe en el estado $\boxed{r}$, que es un estado *muerto*. Diremos que un estado está *muerto* si no es un estado final de aceptación y no parte de él ninguna transición hacia otro estado. Es evidente que si durante el análisis de una cadena se llega a un estado *muerto*, como ya no es posible *salir* de dicho estado, la cadena no será aceptada por el autómata.

Autómatas Incompletos A menudo nos encontramos con autómatas para los que no están definidas todas las transiciones. Las situaciones que no están definidas deben ser consideradas como *situaciones de error*, es decir, si una cadena hace llegar al autómata hasta una situación no definida, consideraremos que la cadena no ha sido reconocida por dicho autómata.

Si deseamos completar un autómata (no es imprescindible) bastará con añadir un estado *muerto* que reciba todas las transiciones que le faltan al autómata incompleto.

En la figura 4.2 podemos ver un ejemplo de esta situación. El autómata de la izquierda está incompleto, pero podemos completarlo trasformándolo en el de la derecha, al que hemos añadido el estado **r**, que es un estado *muerto*.

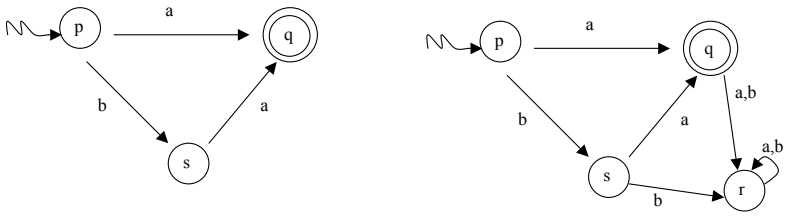


Figura 4.2: Ejemplo de AF incompleto y completo

Estados accesibles y autómatas conexos

Definición 4.1 (Autómata Conexo)
Un autómata es conexo si todos sus estados son accesibles desde el estado inicial.

Definición 4.2 (Parte conexa de un autómata)
Si un autómata no es conexo, se llama parte conexa del autómata al conjunto de estados accesibles desde el estado inicial.

Ejemplo 4.1 El autómata representado en la figura 4.3 no es conexo y su parte conexa es la formada por los estados **p** y **q** y por las transiciones que hay entre ellos.

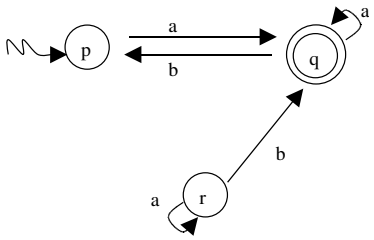


Figura 4.3: AF no conexo

4.4. Los AFD como reconocedores de lenguajes

Como se ha visto en secciones anteriores, la función de transición f ha sido definida de manera que depende de un único símbolo del alfabeto de entrada. A continuación, se ampliará esta definición de forma que dicha función pueda *actuar* sobre cadenas de símbolos, es decir, la función indicará a qué estado pasa el autómata ante la llegada de una cadena de símbolos (y no solamente de un único símbolo). Por tanto, consideraremos que $f : Q \times \Sigma^* \longrightarrow Q$. Para conseguir esta ampliación, f se redefine de forma recursiva:

- $f(q, \lambda) = q \qquad \forall q \in Q$
- $f(q, ax) = f(f(q, a), x) \qquad \forall a \in \Sigma,\ x \in \Sigma^*,\ q \in Q$

Ejemplo 4.2 Considerando el autómata que aparece en la figura 4.1 la función de transición extendida devolvería los siguientes valores:

$f(p, a) = q$ $\qquad$ $f(p, aa) = q$
$f(p, ab) = r$ $\qquad$ $f(p, aabbb) = r$
$f(p, baba) = r$ $\qquad$ $f(r, abb) = r$

Con esta nueva definición de la función de transición es posible definir formalmente cuál es el lenguaje aceptado por un AFD.

Definición 4.3 (Leng. aceptado por un AFD)
El lenguaje que acepta un AFD es el conjunto de palabras definidas sobre Σ *que hacen que el autómata llegue a un estado final de aceptación*

$$L = \{x \in \Sigma^* / \ f(q_0, x) \in F\}$$

4.5. Minimización de un AFD

En ocasiones nos encontramos con autómatas que tienen algunos estados equivalentes, en estos casos esos estados se pueden agrupar de manera que se consigue un autómata, equivalente al primero, pero con un menor número de estados. Se dice que el autómata ha sido minimizado. Un concepto diferente al de minimizar un autómata es el de *simplificar* un autómata que consiste en eliminar estados muertos o inaccesibles.

A continuación se presenta un algoritmo para minimizar un AFD. El objetivo principal de este algoritmo consiste en agrupar estados equivalentes. Consideraremos que dos estados son equivalentes cuando las transiciones que parten de ellos, para cada uno de los símbolos del alfabeto, llevan al mismo estado o a estados que también son

equivalentes entre sí. Todos los estados que sean equivalentes entre sí se fundirán en un único estado en el autómata resultante. Para conseguir este objetivo se construirá una partición de Q, que se irá refinando paulatinamente, de manera que finalmente cada elemento de la partición agrupará estados equivalentes. Inicialmente, se construye una partición de Q formada por dos únicos elementos: los estados de aceptación y los que no lo son. Dicha partición se irá refinando todo lo posible, separando en diferentes elementos a los estados que no son equivalentes. Recordemos que una partición de Q consiste en dividir Q en varios subconjuntos $\{G_i\}_{1\leq i\leq n}$ de tal forma que:

$$G_i \cap G_j = \emptyset \ \forall i \neq j \quad y \quad \bigcup_{1\leq i\leq n} G_i = Q$$

Algoritmo 4.1 Minimización de un AFD

Input: $AFD\ A = (\Sigma, Q, f, q_0, F)$
Output: $AFD\ A' = (\Sigma, Q', f', q_0', F')$
Begin
Partición = $\{G_1, G_2\}$ donde $G_1 = F$ y $G_2 = Q \setminus F$
Auxiliar = $\emptyset$
while Auxiliar $\neq$ Partición **do**
Auxiliar = Partición
$\forall G_i \in$ Partición y $\forall a \in \Sigma$ separar en diferentes grupos a los estados s y $t \in G_i$ siempre que $f(s,a) \in G_j, f(t,a) \in G_k$ siendo $j \neq k$
end while
Cada elemento G_i de Partición se convierte en un estado de Q', las transiciones serán las mismas que define f
End

Ejemplo 4.3 Se minimizará el Autómata representado en la figura 4.4.

Inicialmente, Partición = $\{G_1, G_2\}$ $G_1 = \{E\}$ $G_2 = \{A, B, C, D\}$

Evidentemente no es posible refinar el grupo G_1. Pero hay que comprobar cómo se comportan los estados de G_2 con los símbolos a y b.

G_2		símbolo a	G_2		símbolo b
A	$\rightarrow$	$B \in G_2$	A	$\rightarrow$	$C \in G_2$
B	$\rightarrow$	$B \in G_2$	B	$\rightarrow$	$D \in G_2$
C	$\rightarrow$	$B \in G_2$	C	$\rightarrow$	$C \in G_2$
$\boldsymbol{D}$	$\rightarrow$	$\boldsymbol{B \in G_2}$	$\boldsymbol{D}$	$\rightarrow$	$\boldsymbol{E \in G_1}$

Analizando el comportamiento de los cuatro estados con el símbolo b, es evidente que D no es equivalente a los otros tres estados. Por tanto, es necesario dividir G_2 en dos nuevos elementos.

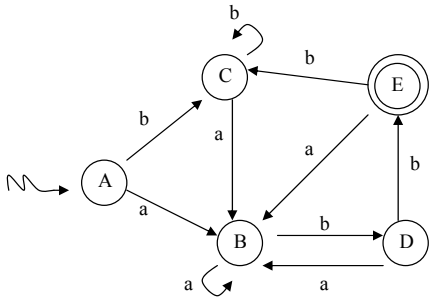


Figura 4.4: Autómata a minimizar

Partición $= \{G_1, G_2', G_3\}$ $G_1 = \{E\}$ $G_2' = \{A, B, C\}$ $G_3 = \{D\}$
Veamos como se comporta G_2' con los símbolos a y b.

G_2'		símbolo a	G_2'		símbolo b
A	$\rightarrow$	$B \in G_2'$	A	$\rightarrow$	$C \in G_2'$
$\boldsymbol{B}$	$\boldsymbol{\rightarrow}$	$\boldsymbol{B \in G_2'}$	$\boldsymbol{B}$	$\boldsymbol{\rightarrow}$	$\boldsymbol{D \in G_3}$
C	$\rightarrow$	$B \in G_2'$	C	$\rightarrow$	$C \in G_2'$

De nuevo es evidente que el estado B no es equivalente a los otros dos. Es necesario dividir G_2'. Ahora, Partición $= \{G_1, G_2'', G_3, G_4\}$ donde
$G_1 = \{E\}$ $G_2'' = \{A, C\}$ $G_3 = \{D\}$ $G_4 = \{B\}$
Volvemos a comprobar el comportamiento de G_2'', el único grupo que es posible refinar.

G_2''		símbolo a	G_2''		símbolo b
A	$\rightarrow$	$B \in G_4$	A	$\rightarrow$	$C \in G_2''$
C	$\rightarrow$	$B \in G_4$	C	$\rightarrow$	$C \in G_2''$

No es posible refinar más la partición. Es evidente que A y C son equivalentes y, por tanto, deben formar parte del mismo grupo. A continuación aparece el nuevo autómata (figura 4.5), equivalente al anterior, que tiene un estado menos, debido al agrupamiento entre A y C(representado por G_2'').

Definición 4.4 (AFD equivalentes)
Dos AFD son equivalentes si reconocen el mismo lenguaje. Es evidente que para que esto ocurra deben estar definidos sobre el mismo alfabeto.

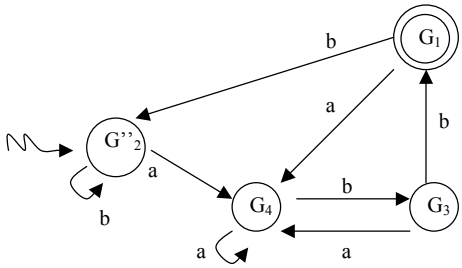


Figura 4.5: Autómata minimizado

A veces es fácil comprobar de manera intuitiva si dos AFD son equivalentes, pero esto no siempre ocurre. Un método para comprobar si dos AFD son equivalentes consiste en *unirlos* de manera que formen un único AFD que, por supuesto, no es conexo. Formalmente, la unión de los dos autómatas se llevaría a cabo así:

Sean $A_1 = \{\Sigma, Q_1, f_1, q_{01}, F_1\}$ $\quad A_2 = \{\Sigma, Q_2, f_2, q_{02}, F_2\}$

El autómata resultante tras la *unión* sería

$A = A_1 + A_2 = \{\Sigma, Q_1 \cup Q_2, f, q_0, F_1 \cup F_2\}$ donde

$$f(q,a) = \begin{cases} f_1(q,a) & si\ q \in Q_1 \\ f_2(q,a) & si\ q \in Q_2 \end{cases}$$

El nuevo estado inicial q_0 puede ser tanto q_{01} como q_{02}.

Una vez construido el autómata *unión*, éste se minimiza. Si al concluir el proceso de minimización, los dos estados iniciales q_{01} y q_{02} forman parte del mismo elemento de la partición, los autómatas originales son equivalentes y el AFD que hemos obtenido es el autómata mínimo equivalente a ambos.

Ejemplo 4.4 Dados los autómatas de la figura 4.6, aplicaremos el método anterior para decidir si son equivalentes o no.

Como hemos visto inicialmente, Partición $= \{G_1, G_2\}$

donde $\quad G_1 = \{q, r, w\} \quad G_2 = \{p, s, v, u\}$

Hay que comprobar cómo se comportan los elementos de G_1 y G_2 con los símbolos a y b.

G_1		símbolo a	G_1		símbolo b
q	$\rightarrow$	$r \in G_1$	q	$\rightarrow$	$p \in G_2$
r	$\rightarrow$	$q \in G_1$	r	$\rightarrow$	$p \in G_2$
w	$\rightarrow$	$w \in G_1$	w	$\rightarrow$	$v \in G_2$

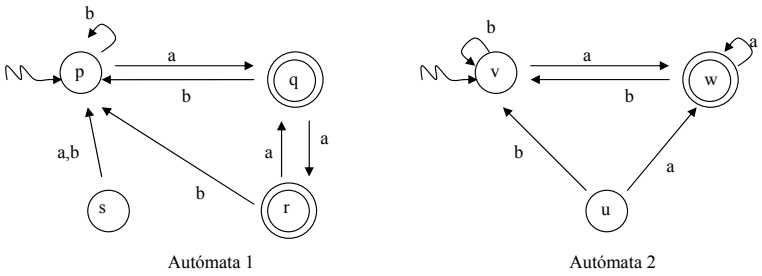


Figura 4.6: ¿Autómatas equivalentes?

G_2		símbolo a	G_2		símbolo b
p	$\rightarrow$	$q \in G_1$	p	$\rightarrow$	$p \in G_2$
$\boldsymbol{s}$	$\boldsymbol{\rightarrow}$	$\boldsymbol{p \in G_2}$	$\boldsymbol{s}$	$\boldsymbol{\rightarrow}$	$\boldsymbol{p \in G_2}$
v	$\rightarrow$	$w \in G_1$	v	$\rightarrow$	$v \in G_2$
u	$\rightarrow$	$w \in G_1$	u	$\rightarrow$	$v \in G_2$

En principio, parece que los estados del elemento G_1 son equivalentes, sin embargo, analizando el comportamiento de los estados del elemento G_2 con el símbolo a, es evidente que s no es equivalente a los otros tres estados. Por tanto, es necesario dividir G_2 en dos subgrupos.

Partición $= \{G_1, G_2', G_3\}$ $G_1 = \{q, r, w\}$ $G_2' = \{p, v, u\}$ $G_3 = \{s\}$

Veamos cómo se comporta G_2' con los símbolos a y b.

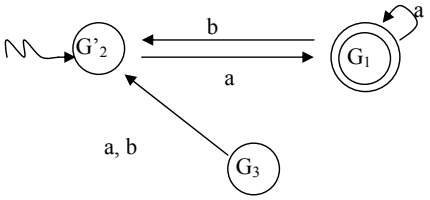
G_2'		símbolo a	G_2'		símbolo b
p	$\rightarrow$	$q \in G_1$	p	$\rightarrow$	$p \in G_2'$
v	$\rightarrow$	$w \in G_1$	v	$\rightarrow$	$v \in G_2'$
u	$\rightarrow$	$w \in G_1$	u	$\rightarrow$	$v \in G_2'$

Es evidente que todos los estados de G_2' son equivalentes. Por tanto, no es posible refinar más la partición. Podemos determinar que los Autómatas 1 y 2 son equivalentes ya que los estados iniciales p y v son equivalentes. Además, el autómata que se muestra en la figura 4.7 es equivalente a ambos y es mínimo.

Como el estado G_3 es inaccesible se puede eliminar. Así el autómata quedaría como muestra la figura 4.8.

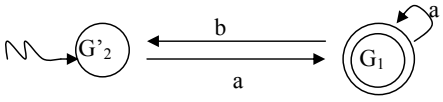
4.6. Autómatas Finitos No Deterministas(AFND)

En los autómatas deterministas sabemos exactamente cuál es la transición que debemos llevar a cabo ante una determinada situación. Sin embargo, en los no deterministas podemos encontrarnos con varias opciones e, incluso, con λ-transiciones



$G_1 = \{q, r, w\}$ $G'_2 = \{p, u, v\}$ $G_3 = \{s\}$

Figura 4.7: Autómata mínimo



$G_1 = \{q, r, w\}$ $G'_2 = \{p, u, v\}$

Figura 4.8: Autómata simplificado

que se realizan sin considerar el correspondiente símbolo de la cadena de entrada.

Para tener en cuenta estas consideraciones, los AFND se definen como una tupla: $AFND = (\Sigma, Q, f, q_0, F, T), \quad f : Q \times \Sigma \longrightarrow 2^Q$ donde

- 2^Q es el conjunto formado por los subconjuntos de Q, incluyendo a $\emptyset$
- T es una relación binaria definida sobre Q que indica las λ-transiciones del autómata (si $pTq \Rightarrow$ existe una λ-transición desde p hasta q)

El resto de los símbolos tiene el mismo significado que en la definición de AFD.

Ejemplo 4.5 Representación de un AFND utilizando un diagrama de transiciones: ver figura 4.9

Descripción del mismo autómata mediante una tabla de transiciones:

	a	b	λ
$\rightarrow *p$	$\{q\}$	$\emptyset$	$\emptyset$
q	$\{p, s\}$	$\{r, p\}$	$\{s\}$
r	$\emptyset$	$\{s, p\}$	$\{r, s\}$
$*s$	$\emptyset$	$\emptyset$	$\{r\}$

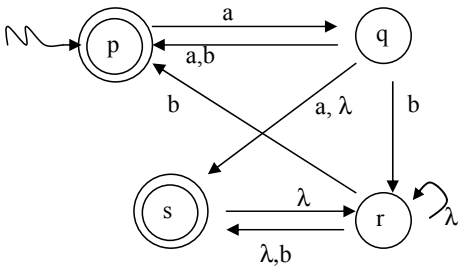


Figura 4.9: A.F. no determinista

Es evidente que un AFD no es más que un caso particular de AFND, es decir, AFD $\subset$ AFND. En realidad, un AFD es un AFND que cumple

$$T = Id \quad y \quad |f(q, a)| = 1 \quad \forall q \in Q,\ \forall a \in \Sigma$$

Es útil conocer el cierre transitivo de la relación T, que se denota T^*. Si pT^*q, entonces q es accesible desde p utilizando exclusivamente λ-transiciones.

Para calcular T^* podemos utilizar una matriz booleana (B_T) que permita representar a T y calcular después todas las potencias de dicha matriz (llega un momento en que las potencias se repiten y no es necesario calcular más). Así $B_{T^2} = (B_T)^2$ representa las parejas de estados que están conectadas por dos λ-transiciones, B_{T^3} las que están conectadas por tres, y así sucesivamente. Por lo tanto $Id + B_T + B_{T^2} + B_{T^3} + \ldots$ es una matriz booleana que representa la relación que deseamos calcular T^*. En nuestro ejemplo:

$$B_T = \begin{matrix} p \\ q \\ r \\ s \end{matrix} \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad (B_T)^2 = B_{T^2} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

$$B_{T^3} = B_{T^4} = \ldots = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \quad B_{T^*} = B_{Id} + B_T + B_{T^2} + B_{T^3} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Definición 4.5 (λ-clausura)
Sea $q \in Q$, se llama λ-clausura(q) al conjunto de estados de Q que son accesibles desde q mediante λ-transiciones. Por lo tanto,

$$\lambda - \text{clausura}(q) = \{p \in Q / qT^* p\}$$

Esta definición se puede ampliar a conjuntos de estados de manera natural.

Si $R \subset Q$, entonces $\lambda - \text{clausura}(R) = \bigcup_{q \in R} \lambda - \text{clausura}(q)$

Aunque para calcular la λ-clausura de un estado se pueden utilizar las matrices booleanas que acabamos de ver, a continuación se describe un algoritmo que también nos permite calcular la λ-clausura.

Algoritmo 4.2 Cálculo de la λ-clausura(q)

Output: Clausura $\subset Q$
Begin
Clausura = $\{q\}$
Auxiliar = $\emptyset$
while Auxiliar $\neq$ Clausura **do**
Auxiliar = Clausura
Clausura = Clausura $\cup$ $\{s \in Q / \exists$ λ-transición desde p hasta s, siendo $p \in$ *Auxiliar*$\}$
end while
λ-clausura(q) = Clausura
End

4.7. Lenguaje aceptado por un AFND

El lenguaje aceptado por un AFND es el conjunto de todas las cadenas de símbolos terminales que pueden hacer que el AFND llegue a un estado final de aceptación. Para llegar a una definición formal de este lenguaje ampliaremos la definición de la función de transición con objeto de que acepte cadenas de caracteres. Es decir, si la función de transición de un AFND de define así: $f : Q * \Sigma \to 2^Q$. Definiremos una función de transición *ampliada*, de la siguiente forma $f' : Q * \Sigma^* \to 2^Q$, donde

- $f'(q, \lambda) = \lambda - \text{clausura}(q)$
- $f'(q, ax) = \{p \in f'(r, x)/r \in f'(q, a)\} =$
  $= \{p \in Q/\ \exists r \in f'(q, a) \text{ y } p \in f'(r, x)\}$ siendo $a \in \Sigma,\ x \in \Sigma^*$

Una vez ampliada la definición de la función de transición, el lenguaje aceptado por el AFND es: $L(AFND) = \{x \in \Sigma^*/f'(q_0, x) \cap F \neq \emptyset\}$

4.8. Simulación de un AFD y AFND

En esta sección veremos sendos algoritmos que nos permitirán simular el comportamiento de un AFD y de un AFND. Por tanto, permitirán determinar si una cadena pertenece o no al lenguaje que reconoce el autómata.

Algoritmo 4.3 Simulación de un AFD

Input: $x \in \Sigma^*$
Begin
  c es el primer carácter de x
  $q = q_0$
  while c $\neq$ Fin **do**
    $q = f(q, c)$
    c es el siguiente carácter de x
  end while
  if $q \in F$ **then**
    la palabra x ha sido reconocida por el AFD
  else
    la palabra x no ha sido reconocida por el AFD
  end if
End

Para detallar el algoritmo de simulación de un AFND, supondremos que tenemos implementadas las siguientes funciones:

- $f(R, a) = \cup_{q \in R} f(q, a)$, siendo $R \subset Q$ y $a \in \Sigma$
- $\lambda - \text{clausura}(R)$, siendo $R \subset Q$

La diferencia fundamental entre ambos algoritmos está en el significado de q y S. En el algoritmo 4.3, q representa el estado que el autómata tiene en cada instante. Sin embargo, en el algoritmo 4.4, q es sustituido por S que representa al conjunto de los estados en los que *puede* estar el autómata.

Algoritmo 4.4 Simulación de un AFND

Input: $x \in \Sigma^*$
Begin
c es el primer carácter de x
$S = \lambda - \text{clausura}(q_0)$
while c $\neq$ Fin **do**
$S = \lambda - \text{clausura}(f(S, c))$
c es el siguiente carácter de x
end while
if $S \cap F \neq \emptyset$ **then**
la palabra x ha sido reconocida por el AFND
else
la palabra x no ha sido reconocida por el AFND
end if
End

4.9. Paso de un AFND a AFD

Los AFND y los AFD tienen el mismo poder computacional (esto no ocurre en otros niveles de la jerarquía de los autómatas), es decir, pueden resolver los mismos problemas. Por lo tanto, dado un AFND siempre es posible encontrar un AFD que sea equivalente a él. En esta sección estudiaremos un método para resolver este problema. En primer lugar explicaremos de manera genérica el paso de un autómata a otro, para después ilustrar con un ejemplo este mecanismo de trasformación.

Partimos de un AFND $= (\Sigma, Q, f, q_0, F, T)$ y queremos construir un AFD $= (\Sigma, Q', f', q_0', F')$ que sea equivalente, donde:

1. $Q' = 2^Q$
2. $q_0' = \lambda - \text{clausura}(q_0)$
3. $F' = \{C \subset Q / C \cap F \neq \emptyset\}$
4. $f'(C, a) = \{C' \subset Q / C' = \bigcup_{q \in C} \lambda - \text{clausura}(f(q, a))\}$, siendo $C \subset Q$

El autómata que se obtiene por este método no tiene porqué ser mínimo y podría llegar a tener, como máximo, $2^{|Q|}$ estados.

Ejemplo 4.6 Calcularemos el AFD equivalente al AFND que se muestra en la figura 4.10. Comenzamos calculando el estado inicial q_0 que es la λ-clausura del estado inicial del AFND. $q_0 = \lambda$-clausura(A) = {A,C}

A continuación hay que calcular la función de transición para el estado q_0

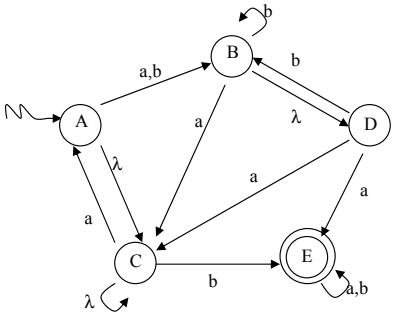


Figura 4.10: Ejemplo

$$
\begin{aligned}
f(q_0,a) &= \lambda\text{-clausura}(f(A,a)\cup f(C,a)) = \lambda\text{-clausura}(B,A) = \{A,B,C,D\} = q_1\\
f(q_0,b) &= \lambda\text{-clausura}(f(A,b)\cup f(C,b)) = \lambda\text{-clausura}(B,E) = \{B,D,E\} = q_2
\end{aligned}
$$

Es necesario seguir calculando la función de transición para los nuevos estados que van apareciendo.

$$
\begin{aligned}
f(q_1,a) &= \lambda\text{-clausura}(f(A,a)\cup f(B,a)\cup f(C,a)\cup f(D,a))\\
&= \lambda\text{-clausura}(B,C,A,E) = \{A,B,C,D,E\} = q_3\\
f(q_1,b) &= \lambda\text{-clausura}(f(A,b)\cup f(B,b)\cup f(C,b)\cup f(D,b))\\
&= \lambda\text{-clausura}(B,E) = \{B,D,E\} = q_2\\
f(q_2,a) &= \lambda\text{-clausura}(f(B,a)\cup f(D,a)\cup f(E,a))\\
&= \lambda\text{-clausura}(C,E) = \{C,E\} = q_4\\
f(q_2,b) &= \lambda\text{-clausura}(f(B,b)\cup f(D,b)\cup f(E,b))\\
&= \lambda\text{-clausura}(B,E) = \{B,D,E\} = q_2\\
f(q_3,a) &= \lambda\text{-clausura}(f(A,a)\cup f(B,a)\cup f(C,a)\cup f(D,a)\cup f(D,a))\\
&= \lambda\text{-clausura}(B,C,A,E) = \{A,B,C,D,E\} = q_3\\
f(q_3,b) &= \lambda\text{-clausura}(f(A,b)\cup f(B,b)\cup f(C,b)\cup f(D,b)\cup f(D,b))\\
&= \lambda\text{-clausura}(B,E) = \{B,D,E\} = q_2\\
f(q_4,a) &= \lambda\text{-clausura}(f(C,a)\cup f(E,a)) = \lambda\text{-clausura}(A,E) = \{A,C,E\} = q_5\\
f(q_4,b) &= \lambda\text{-clausura}(f(C,b)\cup f(E,b)) = \lambda\text{-clausura}(E) = \{E\} = q_6\\
f(q_5,a) &= \lambda\text{-clausura}(f(A,a)\cup f(C,a)\cup f(E,a))\\
&= \lambda\text{-clausura}(B,A,E) = \{A,B,C,D,E\} = q_3\\
f(q_5,b) &= \lambda\text{-clausura}(f(A,b)\cup f(C,b)\cup f(E,b))\\
&= \lambda\text{-clausura}(B,E) = \{B,D,E\} = q_2\\
f(q_6,a) &= \lambda\text{-clausura}(f(E,a)) = \lambda\text{-clausura}(E) = q_6\\
f(q_6,b) &= \lambda\text{-clausura}(f(E,b)) = \lambda\text{-clausura}(E) = q_6
\end{aligned}
$$

El AFD resultante aparece en la siguiente figura.

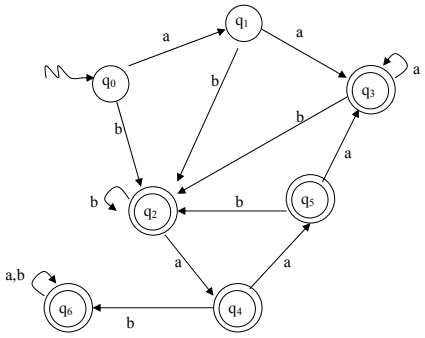


Figura 4.11: AFD equivalente al de la figura 4.10

Una vez obtenido el AFD, intentaremos minimizarlo.

Inicialmente, Partición $= \{G_1, G_2\}$ $G_1 = \{q_2, q_3, q_4, q_5, q_6\}$ $G_2 = \{q_0, q_1\}$

Analizaremos el comportamiento de todos los estados con a y b.

G_1		símbolo a	G_1		símbolo b
q_2	$\rightarrow$	$q_4 \in G_1$	q_2	$\rightarrow$	$q_2 \in G_1$
q_3	$\rightarrow$	$q_3 \in G_1$	q_3	$\rightarrow$	$q_2 \in G_1$
q_4	$\rightarrow$	$q_5 \in G_1$	q_4	$\rightarrow$	$q_6 \in G_1$
q_5	$\rightarrow$	$q_3 \in G_1$	q_5	$\rightarrow$	$q_2 \in G_1$
q_6	$\rightarrow$	$q_6 \in G_1$	q_6	$\rightarrow$	$q_6 \in G_1$

G_2		símbolo a	G_2		símbolo b
q_0	$\rightarrow$	$q_1 \in G_2$	q_0	$\rightarrow$	$q_2 \in G_1$
q_1	$\rightarrow$	$q_3 \in G_1$	q_1	$\rightarrow$	$q_2 \in G_1$

Todos los estados de G_1 son equivalentes, sin embargo, esto no ocurre con los de G_2, por lo que es necesario separarlos en dos elementos diferentes. La nueva y definitiva partición sería:

$\{G_1, G_2', G_3\}$ $G_1 = \{q_2, q_3, q_4, q_5, q_6\}$ $G_2' = \{q_0\}$ $G_3 = \{q_1\}$.

A continuación se representa el nuevo autómata (figura 4.12) que es determinista, mínimo y equivalente al AFND del que partimos.

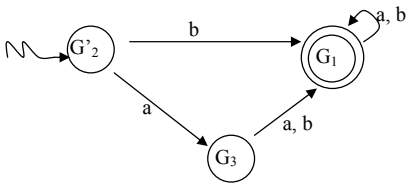


Figura 4.12: AFD mínimo equivalente al del figura 4.11

4.10. Relación entre Autómatas Finitos, gramática y expresiones regulares

Como sabemos las gramáticas regulares generan lenguajes regulares que pueden ser representados mediante expresiones regulares. A su vez, estos lenguajes pueden ser reconocidos por Autómatas Finitos.

En esta sección estudiaremos diferentes métodos que nos permitirán construir unos elementos a partir de otros.

4.10.1. Construcción de la expresión regular reconocida por un AF

Estudiaremos dos métodos diferentes.

- El método del sistema de ecuaciones
- El método de las funciones recursivas

Previamente se define y demuestra la *Regla de Inferencia* que se aplicará en el primer método como un mecanismo para *despejar incógnitas*.

Teorema 4.1 (Regla de Inferencia)
Sean R, S, T tres expresiones regulares de manera que $\lambda \notin S$.
Se cumple que $R = SR + T \Leftrightarrow R = S^*T$

Demostración:

- Suponemos que $R = S^*T$ y queremos demostrar que $R = SR + T$

  $R = S^*T = (\lambda + S^+)T = (\lambda + SS^*)T = T + SS^*T = T + S(S^*T) = T + SR$

- Suponemos que $R = SR + T$ y queremos demostrar que $R = S^*T$, para ello comenzaremos demostrando que $S^*T \subset R$

  Si $\alpha \in S$ y $\beta \in T$ (como $R = SR + T$ entonces $T \subset R$ y por tanto $\beta \in R$) $\Rightarrow \alpha\beta \in SR \subset R$

  Análogamente y aplicando n veces el mismo razonamiento,

  si $\alpha_1, \ldots, \alpha_n \in S$ y $\beta \in T \Rightarrow \alpha_1 \ldots \alpha_n\beta \in R \Rightarrow S^*T \subset R$

  Supongamos que $S^*T \neq R$, entonces $R = S^*T + C$

  (consideramos que $C \cap S^*T = \emptyset$)

  $$\begin{aligned} R &= SR + T = S(S^*T + C) + T = SS^*T + SC + T = (SS^* + \lambda)T + SC \\ &= S^*T + C = S^*T + SC \end{aligned}$$

  Eso significa que cualquier palabra de C debe pertenecer a SC (ya que no puede pertenecer a S^*T) y esto es absurdo ya que $\lambda \notin S$ por lo tanto $C = \emptyset$ y se cumple que $S^*T = R$ □

El método del sistema de ecuaciones

Este método se basa en la definición de una serie de e.r. que inicialmente serán las incógnitas de un sistema de ecuaciones.

Partimos de un $AFD = (\Sigma, Q, f, q_0, F)$ y para cada uno de los estados $q_i \in Q$ definimos una e.r. X_i que representa a todas las cadenas que permiten llegar desde el estado i hasta algún estado final de aceptación.

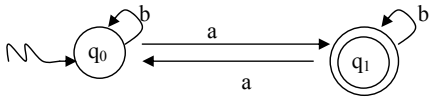
- $X_i = \emptyset$ si F es inaccesible desde q_i
- $a \in X_i, \quad \forall a \in \Sigma$ tal que $f(q_i, a) \in F$
- Todas las cadenas de la forma aX_j pertenecen a X_i, si $f(q_i, a) = q_j$

Teniendo en cuenta estas consideraciones, X_i se define de la siguiente forma:

$X_i = \Sigma_{j=1}^{n} a_{ij}X_j + a_{f1} + a_{f2} + \ldots + a_{fm}$ donde $f(q_i, a_{ij}) = q_j$ y $f(q_i, a_{fk}) \in F$. Además, hay que añadir λ si $q_i \in F$.

Si aplicamos esta definición para todos los estados del AF, conseguimos construir un sistema de n ecuaciones con n incógnitas, donde n es el número de estados del AF. El sistema de ecuaciones se resolverá sustituyendo unas ecuaciones en otras y aplicando la *regla de inferencia* para despejar dichas incógnitas. En realidad, no es necesario resolver el sistema completo ya que la única incógnita que nos interesa es X_0 (considerando que q_0 es el estado inicial) que es la e.r. buscada.

Ejemplo 4.7 Sea el siguiente AF



Definimos el sistema de ecuaciones:

$$\left\{ \begin{array}{l} X_0 = bX_0 + aX_1 + a \\ X_1 = aX_0 + bX_1 + b + \lambda \end{array} \right\}$$

Aplicamos la regla de inferencia en la 2 ecuación.

(considerando que $R = X_1,\ S = b$ y $T = aX_0 + b + \lambda$)

$X_1 = aX_0 + bX_1 + b + \lambda = b^*(aX_0 + b + \lambda)$

Sustituimos X_1 en la primera ecuación.

$$\begin{array}{rcl} X_0 & = & bX_0 + aX_1 + a \\ & = & bX_0 + a(b^*(aX_0 + b + \lambda)) + a \\ & = & bX_0 + ab^*aX_0 + ab^*b + ab^* + a \\ & = & (b + ab^*a)X_0 + ab^* \end{array}$$

Aplicamos de nuevo la *regla de inferencia*, esta vez en la primera ecuación.

(ahora, $R = X_0,\ S = b + ab^*a$ y $T = ab^*$)

$X_0 = (b + ab^*a)X_0 + ab^* = (\mathbf{b + ab^*a})^*\mathbf{ab^*}$

El método de las funciones recursivas

Para que este método pueda llevarse a cabo sin ambigüedades es necesario numerar los estados a partir del 1, es decir, $Q = \{q_1, \ldots, q_n\}$.

También en este caso definiremos una serie de e.r.(en este caso, de forma recursiva) que inicialmente serán incógnitas que es necesario calcular.

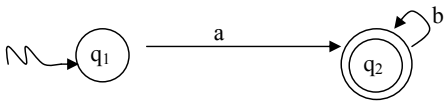
Cada e.r. R_{ij}^k representará a las cadenas que permiten llegar del estado q_i al estado q_j pasando exclusivamente por los estados $q_1, \ldots, q_k$. Definiremos R_{ij}^0 como el conjunto de cadenas (en este caso, símbolos) que nos llevarán directamente del estado q_i al estado q_j. Las e.r del tipo R_{ij}^0 se definen de forma directa, mientras que las e.r. $R_{ij}^k, k \geq 1$ se definen de forma recursiva.

$$R_{ij}^0 = \left\{ \begin{array}{ll} \{a \in \Sigma / f(q_i, a) = q_j\} & \text{si } i \neq j \\ \{a \in \Sigma / f(q_i, a) = q_j\} \cup \lambda & \text{si } i = j \end{array} \right.$$

$$R_{ij}^k = R_{ij}^{k-1} + R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1}, \quad k \geq 1$$

Una vez calculadas las e.r. R_{ij}^k, la e.r. buscada es: $R_{1f_1}^n + R_{1f_2}^n + \ldots$ donde q_1 es el estado inicial y $q_{f_i} \in F$

Ejemplo 4.8 Dado el siguiente AFD, calculamos las e.r. R_{ij}^k



	$k=0$	$k=1$	$k=2$
R_{11}	λ	λ	λ
R_{12}	a	a	ab^*
R_{21}	$\emptyset$	$\emptyset$	$\emptyset$
R_{22}	$b+\lambda$	$b+\lambda$	b^*

Las dos últimas columnas se han calculado mediante la fórmula recursiva vista anteriormente.

$R_{11}^1 = R_{11}^0 + R_{11}^0(R_{11}^0)^*R_{11}^0 = \lambda + \lambda\lambda^*\lambda = \lambda$
$R_{12}^1 = R_{12}^0 + R_{11}^0(R_{11}^0)^*R_{12}^0 = a + \lambda\lambda^*a = a$
$R_{21}^1 = R_{21}^0 + R_{21}^0(R_{11}^0)^*R_{11}^0 = \emptyset + \emptyset\lambda^*\lambda = \emptyset$
$R_{22}^1 = R_{22}^0 + R_{21}^0(R_{11}^0)^*R_{12}^0 = (b+\lambda) + \emptyset\lambda^*a = b + \lambda$
$R_{11}^2 = R_{11}^1 + R_{12}^1(R_{22}^1)^*R_{21}^1 = \lambda + a(b+\lambda)^*\emptyset = \lambda$
$R_{12}^2 = R_{12}^1 + R_{12}^1(R_{22}^1)^*R_{22}^1 = a + a(b+\lambda)^*(b+\lambda) = a + ab^* = ab^*$
$R_{21}^2 = R_{21}^1 + R_{22}^1(R_{22}^1)^*R_{21}^1 = \emptyset + (b+\lambda)(b+\lambda)^*\emptyset = \emptyset$
$R_{22}^2 = R_{22}^1 + R_{22}^1(R_{22}^1)^*R_{22}^1 = (b+\lambda) + (b+\lambda)(b+\lambda)^*(b+\lambda) = b^*$

Teniendo en cuenta que sólo hay un estado final de aceptación q_2, la e.r. que estamos buscando será $R_{12}^2 = \mathbf{ab^*}$

4.10.2. Construcción del AF que reconoce una expresión regular

Estudiaremos dos métodos que nos ayudarán a construir un autómata que reconoce el lenguaje que representa una e.r. dada. El primero construye un AFND mientras que el segundo permite construir un AFD.

Paso de expresión regular a AFND De la misma forma que las e.r. se definieron de forma recursiva, este método para construir el AFND, que está basado en la definición de las e.r., también puede considerarse recursivo. Para cada tipo de e.r. construiremos un AFND, de esta manera diferentes autómatas pueden ensamblarse para construir otro más complejo.

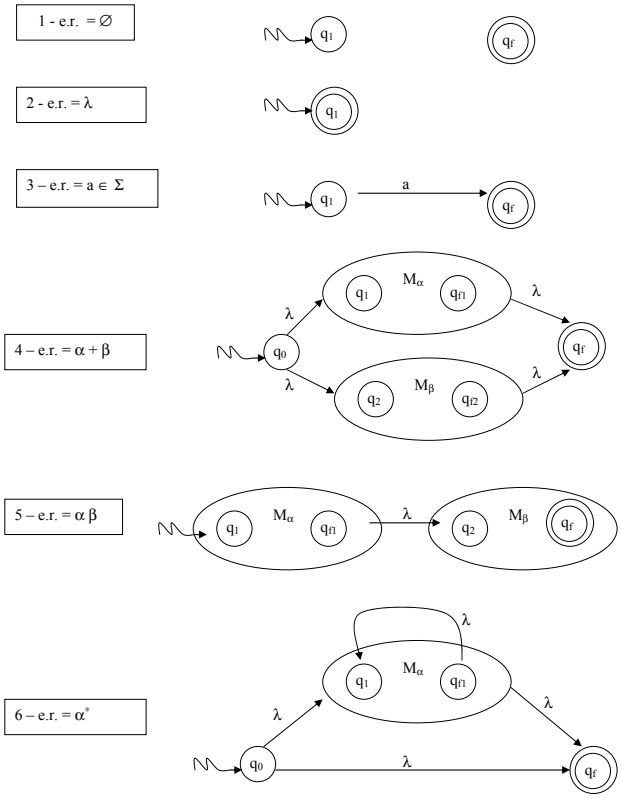


Figura 4.13: Paso de e.r. a AFND

En la figura anterior se detallan los esquemas asociados a cada una de las operaciones que podemos encontrar en una e.r., M_α y M_β representan a los autómatas que reconocen a las e.r. α y β respectivamente.

Paso de expresión regular a AFD Introduciremos este método mediante un ejemplo con la e.r. **(a + b)*abb**.

Para desarrollar este método es necesario etiquetar con un número a cada uno de los símbolos que componen la e.r. A estas etiquetas las llamaremos *posiciones*. Además, añadiremos el símbolo # a la derecha de la e.r. para indicar el final de las

palabras del lenguaje representado.

(	a	+	b	)	*	a	b	b	#
	1		2			3	4	5	6

Para cada una de las *posiciones* es necesario definir su conjunto *siguiente* que estará formado por las posiciones que pueden seguir a una dada en cualquier palabra que pertenezca al lenguaje representado por la e.r. Para calcular estos conjuntos es necesario analizar cada una de las operaciones que intervienen en la e.r. y estudiar cómo afectan a las diferentes posiciones. Para este ejemplo, los valores de estos conjuntos serían:

$$sig(1) = \{1,2,3\} \qquad sig(3) = \{4\} \qquad sig(5) = \{6\}$$
$$sig(2) = \{1,2,3\} \qquad sig(4) = \{5\} \qquad sig(6) = \emptyset$$

Cada estado de nuestro AFD será un conjunto de *posiciones*. Los estados finales de aceptación serán aquellos que contienen a la posición asociada al símbolo # (en el ejemplo, la posición 6). Se calculan simultáneamente estos estados y las transiciones correspondientes mediante el algoritmo 4.5. En este algoritmo hay que tener en cuenta que $simb(i)$ indica el símbolo del alfabeto asociado a la *posición i*. En nuestro caso:

$$simb(1) = \{a\} \qquad simb(3) = \{a\} \qquad simb(5) = \{b\}$$
$$simb(2) = \{b\} \qquad simb(4) = \{b\} \qquad simb(6) = \emptyset$$

Además, *pp* representa a las primeras posiciones de la e.r., es decir, las posiciones por las que puede comenzar cualquier palabra representada por la e.r. En este ejemplo, $pp = \{1,2,3\}$

Las siglas **EM** y **ENM** significan Estados Marcados y Estados No Marcados respectivamente, y representan a dos conjuntos de estados del autómata que se está construyendo. Estos conjuntos se utilizan para saber si un estado ha sido marcado o no. Marcar un estado significa procesarlo, es decir, calcular las transiciones que parten de dicho estado. Cuando un estado se procesa pasa del conjunto ENM al conjunto EM. La construcción del autómata terminará cuando no quede ningún estado sin marcar.

Veamos como se aplicaría el algoritmo 4.5 al ejemplo con el que estamos trabajando.

Comenzamos definiendo el estado inicial $q_0 = pp = \{1,2,3\}$.

Inicialmente, $EM = \emptyset$ y $ENM = \{q_0\}$

A continuación hay que calcular la función de transición para el estado q_0

$f(q_0, a) = sig(1) \cup sig(3) = \{1,2,3,4\} = q_1$
$f(q_0, b) = sig(2) = \{1,2,3\} = q_0$ Ahora: $EM = \{q_0\}$ y $ENM = \{q_1\}$
Es necesario seguir calculando la función de transición para los nuevos estados que van apareciendo.

Algoritmo 4.5 Construcción del AFD a partir de la e.r.

Input: $x \in \Sigma^*$
Begin
ENM = pp
EM = $\emptyset$
while ENM $\neq \emptyset$ **do**
Pasar T desde ENM hasta EM
for all $a \in \Sigma$ **do**
$R = \bigcup_{\forall i \in T} sig(i)$ tal que $simb(i) = a$
if $R \neq \emptyset$ and $R \notin (EM \cup ENM)$ **then**
añadir R a ENM
$f(T, a) = R$
end if
end for
end while
End

$f(q_1, a) = sig(1) \cup sig(3) = \{1, 2, 3, 4\} = q_1$
$f(q_1, b) = sig(2) \cup sig(4) = \{1, 2, 3, 5\} = q_2 \quad EM = \{q_0, q_1\}$ y $ENM = \{q_2\}$
$f(q_2, a) = sig(1) \cup sig(3) = \{1, 2, 3, 4\} = q_1$
$f(q_2, b) = sig(2) \cup sig(5) = \{1, 2, 3, 6\} = q_3 \quad EM = \{q_0, q_1, q_2\}, ENM = \{q_3\}$
$f(q_3, a) = sig(1) \cup sig(3) = \{1, 2, 3, 4\} = q_1$
$f(q_3, b) = sig(2) = \{1, 2, 3\} = q_0 \qquad EM = \{q_0, q_1, q_2, q_3\}$ y $ENM = \emptyset$

La figura 4.14 representa gráficamente al autómata construido. Como se puede observar, el único estado final de aceptación es q_3.

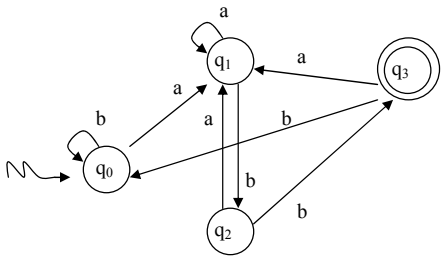


Figura 4.14: AFD correspondiente a la e.r. $(a + b)^*abb$

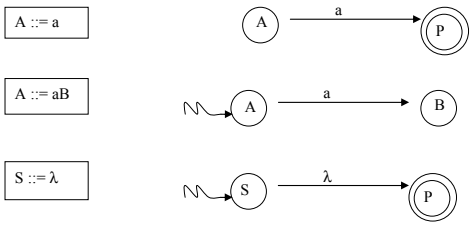
4.10.3. Relación entre A.F. y gramáticas regulares

En este apartado, se estudiarán métodos, similares entre si, para construir el AF que reconoce al lenguaje generado por una gramática regular (distinguiendo si es lineal por la izquierda o por la derecha). De forma análoga se estudiarán métodos para construir gramáticas que generen el lenguaje que reconoce un AF dado. Para poder definir una relación entre los AF's y las gr. regulares estableceremos, en primer lugar, las siguientes correspondencias:

- Cada estado del autómata se corresponderá con un símbolo no terminal de la gramática.
- Cada transición del autómata se corresponderá con una producción de la gramáti

Paso de GLD a AFND En este caso:

1. El estado inicial del autómata se corresponderá con el símbolo inicial de la gramática.
2. Definiremos un estado final de aceptación P que no se corresponde con ningún símbolo no terminal de la gramática.
3. A cada producción de la gramática le corresponde una transición en el autómata según el siguiente esquema:



Ejemplo 4.9 En la figura 4.15 se muestra la obtención de un autómata finito a partir de una gramática lineal por la derecha.

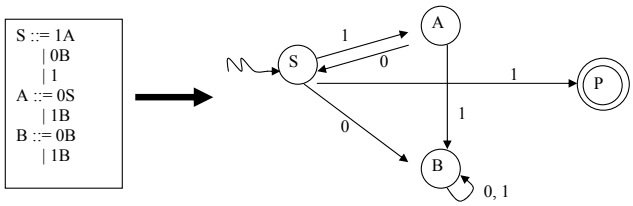


Figura 4.15: Ejemplo de paso de GLD a AFND

Paso de AFND a GLD En este caso debemos tener en cuenta las mismas relaciones que hemos visto en el caso anterior. Sin embargo, las transiciones que llevan a un estado final dan lugar a dos producciones diferentes como indica la siguiente figura.

Ejemplo 4.10 En la figura 4.16 se muestra la obtención de una gramática lineal por la derecha a partir de un autómata finito. La gramática ha sido posteriormente simplificada ya que el símbolo **P** no es un estado generativo.

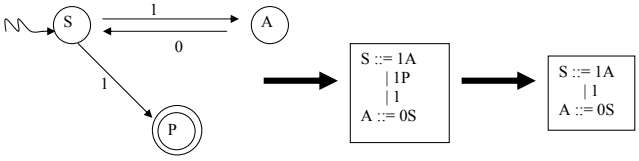


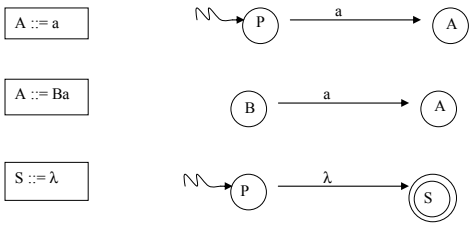
Figura 4.16: Ejemplo de paso de AFND a GLD

Paso de GLI a AFND Si trabajamos con GLI's, debemos tener en cuenta que:

1. El estado final del autómata se corresponderá con el símbolo inicial de la gramática.

2. Definiremos un estado inicial llamado P que no se corresponde con ningún símbolo no terminal de la gramática.

3. A cada producción de la gramática le corresponde una transición en el Autómata según el siguiente esquema:



Ejemplo 4.11 En la figura 4.17 se muestra el autómata obtenido a partir de una gramática lineal por la izquierda.

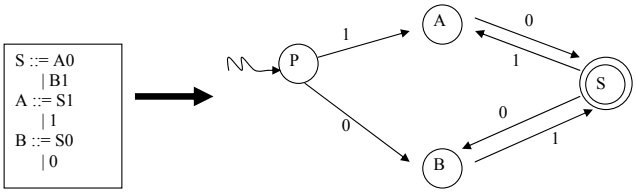
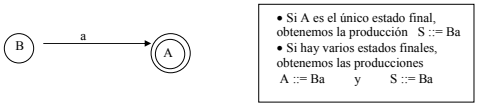


Figura 4.17: Ejemplo de paso de GLI a AFND

Paso de AFND a GLI En este caso debemos tener en cuenta las mismas relaciones que hemos visto en el caso anterior. Sin embargo, hay que tener en cuenta que puede haber varios estados finales, en ese caso las transiciones que llevan a un estado final dan lugar a dos producciones según se indica en el siguiente esquema.



Ejemplo 4.12 En la figura 4.18 se muestra la gramática lineal por la izquierda obtenida a partir de un autómata finito.

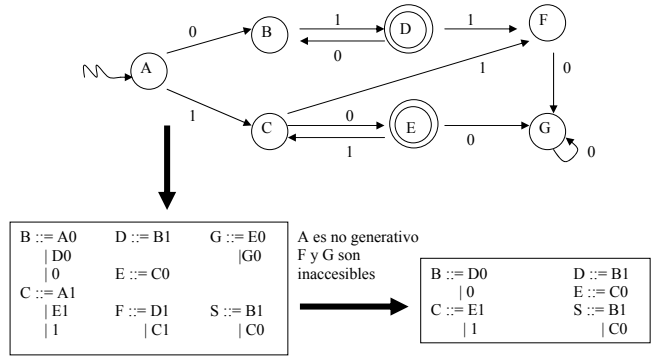


Figura 4.18: Ejemplo de paso de AFND a GLI

4.11. Límites para los leng. regulares

En esta sección estudiaremos dos resultados, el lema del *bombeo* y el teorema de Myhill-Nerode que nos permitirán establecer *límites* para determinar si un lenguaje es o no regular.

4.11.1. El lema del bombeo(*pumping lemma*)

El lema del bombeo enuncia una propiedad que deben cumplir todos los lenguajes regulares. El hecho de comprobar que un lenguaje no cumple dicha propiedad es suficiente para demostrar que no es regular. Sin embargo, en ningún caso este lema servirá para demostrar que un lenguaje es regular.

Lema 4.1 (El lema del bombeo para leng. regulares)
Sea L un lenguaje regular, entonces existe una constante asociada al lenguaje $n > 0$, de manera que $\forall z \in L$ tal que $|z| \geq n$, se cumple que z se puede descomponer en tres partes $z = uvw$ que verifican:

1. $|v| \geq 1$
2. $|uv| \leq n$
3. $\forall i \geq 0 \Rightarrow uv^i w \in L$

Ejemplo 4.13 Utilizaremos el lema del bombeo para demostrar que el lenguaje $L = \{a^k b^k / k \geq 0\}$ no es regular. La demostración se realizará por el método de *reducción al absurdo*. Es decir, supondremos que L es regular, si partiendo de esta hipótesis llegamos a una situación absurda habremos comprobado que nuestra suposición inicial era falsa.

Supongamos que L es regular y que $n \geq 0$ es la constante asociada a L que menciona el lema del bombeo. Evidentemente sea cual sea el valor de n siempre es posible encontrar una palabra en L cuya longitud sea mayor que n, por ejemplo, sea $z = a^n b^n$, en este caso $|z| = 2n > n$. Veamos diferentes formas de dividir z en tres partes:

1. $z = uvw = a \ldots |a \ldots a| \ldots ab \ldots b$
2. $z = uvw = a \ldots ab \ldots |b \ldots b| \ldots b$
3. $z = uvw = a \ldots |a \ldots ab \ldots b| \ldots b$

En el primer caso sólo se bombearían a's con los que conseguiríamos cadenas con más a's que b's, que no pertenecerían a L. En el segundo caso ocurriría lo contrario ya que sólo bombearíamos b's. En el tercer caso bombeamos a's y b's simultáneamente, por tanto sería posible obtener el mismo número de a's que de b's. Sin embargo, las cadenas obtenidas contendrían subcadenas del tipo *ababab* o *aabbaabbaabb*, de cualquier forma estas palabras nunca pertenecerían a L. No existe ninguna otra forma de dividir la cadena en tres partes; por tanto, hemos comprobado que el lema no se cumple y podemos asegurar que el lenguaje no es regular.

4.11.2. El teorema de Myhill-Nerode

Este teorema nos permitirá saber si un lenguaje es o no regular. Además y en el caso de que el lenguaje sea regular, la demostración del teorema nos muestra un método para construir el AFD mínimo que reconoce a dicho lenguaje. Antes de enunciar el teorema será necesario conocer algunas definiciones acerca de las relaciones binarias. Consideraremos que R es una relación de equivalencia definida sobre el conjunto X.

Definición 4.6 (Relación invariante por la derecha)
Se dice que R es invariante por la derecha respecto a una operación $\circ$ definida sobre X, si se cumple que:

$$si\ xRy \Rightarrow \forall z \in X,\ x \circ zRy \circ z$$

Definición 4.7 (Relación de índice finito)
Se dice que R es de índice finito si el cardinal de su conjunto cociente es finito, es decir, si el número de clases de equivalencia que define R es finito.

Teorema 4.2 (Teorema de Myhill-Nerode)
Dado un lenguaje L definido sobre un alfabeto Σ, las siguientes afirmaciones son equivalentes:

1. *$L \subset \Sigma^*$ es regular*

2. *L es la unión de algunas clases de equivalencia de una relación de equivalencia R_M definida sobre Σ^* que es de índice finito e invariante por la derecha respecto a la concatenación.*

3. *A partir de L se puede definir una relación binaria R_L sobre Σ^*, de la siguiente forma:*

$$xR_Ly \Longleftrightarrow \forall z \in \Sigma^* \quad xz \in L \Leftrightarrow yz \in L$$

Es decir, o ambas cadenas (xz e yz) pertenecen a L, o ninguna de las dos pertenece a L.

Se cumple que la relación R_L es una relación de equivalencia, de índice finito e invariante por la derecha respecto a la concatenación.

Demostración.
Para demostrar este teorema comprobaremos en primer lugar que $1 \Rightarrow 2$, después que $2 \Rightarrow 3$ y finalmente que $3 \Rightarrow 1$.

$1 \Longrightarrow 2$ Suponemos que L es regular, entonces existe un $AFD = (\Sigma, Q, f, q_0, F)$ que lo reconoce. A partir de este autómata podemos definir una relación binaria sobre Σ^* a la que llamaremos R_M:

$$\forall x, y \in \Sigma^* \qquad xR_My \Longleftrightarrow f(q_0, x) = f(q_0, y)$$

Es evidente que esta relación es de equivalencia y además tendremos tantas clases de equivalencia como estados tenga el autómata, por tanto, será una relación de índice finito.

Veamos que R_M es de invariante por la derecha respecto a la concatenación. Si $xR_My \Rightarrow f(q_0, x) = f(q_0, y) \Rightarrow f(q_0, xz) = f(q_0, yz)\ \forall z \in \Sigma^* \Rightarrow xzR_Myz$ c.q.d.

Si $x \in \Sigma^*$ entonces $[x]$ representa a la clase de equivalencia de x, es decir, $[x]$ es el conjunto de todas las cadenas que están relacionadas con x.

Utilizando esta notación y teniendo en cuenta la definición de R_M es evidente que $L = \{\cup[x]/f(q_0, x) \in F\}$. Es decir, L es la unión de varias clases de equivalencia de R_M, concretamente, tantas como estados finales tenga el autómata.

2 $\Longrightarrow$ 3 Suponemos que L es la unión de varias clases de equivalencia definidas por una relación a la que llamaremos R_M que es de índice finito e invariante por la derecha.

Por otra parte, se ha definido la relación R_L de la siguiente forma:

$$xR_Ly \Longleftrightarrow \forall z \in \Sigma^* \quad xz \in L \Leftrightarrow yz \in L$$

Es evidente que R_L es una relación de equivalencia. Veamos que es invariante por la derecha:

$$\begin{array}{lll} xR_Ly & \Longrightarrow \forall z \in \Sigma^* & xz \in L \Leftrightarrow yz \in L \\ & \Longrightarrow \forall z, z' \in \Sigma^* & xzz' \in L \Leftrightarrow yzz' \in L \\ & \Longrightarrow \forall z \in \Sigma^* & xzR_Lyz \end{array}$$

Comprobaremos a continuación que $xR_My \Rightarrow xR_Ly$. Si esto ocurre el número de clases de equivalencia que genera R_M será mayor o igual que el número de clases de R_L, lo que permitirá afirmar que R_L es de índice finito.

$$xR_My \Longrightarrow \forall z \in \Sigma^* \; xzR_Myz \Longrightarrow \text{como} [xz] = [yz] \begin{cases} \text{si } [xz] \subseteq L \Rightarrow xz \in L, yz \in L \\ \text{si } [xz] \nsubseteq L \Rightarrow xz \notin L, yz \notin L \end{cases}$$

$$\Longrightarrow xz \in L \Leftrightarrow yz \in L \Longrightarrow xzR_Lyz \quad \text{c.q.d.}$$

3 $\Longrightarrow$ 1 Suponemos que R_L es de índice finito e invariante por la derecha. Para demostrar que L es un lenguaje regular, vamos a construir un $AFD = (\Sigma, Q, f, q_0, F)$ que lo reconozca.

$$\begin{array}{ll} Q = \Sigma^*/R_L & f([x], a) = [xa] \\ q_0 = [\lambda] & F = \{[x]/x \in L\} \end{array}$$

- Q es un conjunto finito porque R_L es de índice finito.
- f está bien definido porque si $[x] = [y] \Rightarrow [xa] = [ya] \forall a \in \Sigma$ ya que R_L es invariante por la derecha respecto a la concatenación $\Rightarrow f([x], a) = f([y], a)$
- Este autómata finito que se ha definido reconoce a L ya que
  $x \in L \Longleftrightarrow [x] \in F \Longleftrightarrow f(q_0, x) = f([\lambda], x) = [x] \in F$

Como hemos demostrado que L es reconocido por un Autómata Finito, podemos asegurar que L es un lenguaje regular. □

Ejemplo 4.14 Utilizando el teorema de Myhill-Nerode construiremos el AFD mínimo que reconozca el lenguaje **0*10***.

$q_0 = [\lambda] \Longrightarrow f(q_0, 0) = [\lambda 0] = [0] \qquad f(q_0, 1) = [\lambda 1] = [1]$

$[0] = [\lambda]? \Longleftrightarrow \forall z \quad 0z \in L \Leftrightarrow z \in L?$ **Cierto** $\Longrightarrow f(q_0, 0) = [0] = q_0$

$[1] = [\lambda]? \Longleftrightarrow \forall z \quad 1z \in L \Leftrightarrow z \in L?$ **Falso** $\Longrightarrow f(q_0, 1) = [1] = q_1$

$f(q_1, 0) = [10] \qquad f(q_1, 1) = [11]$

$[10] = [\lambda]? \Longleftrightarrow \forall z \quad 10z \in L \Leftrightarrow z \in L?$ **Falso**

$[10] = [1]? \Longleftrightarrow \forall z \quad 10z \in L \Leftrightarrow 1z \in L?$ **Cierto** $\Longrightarrow f(q_1, 0) = [10] = q_1$

$[11] = [\lambda]? \Longleftrightarrow \forall z \quad 11z \in L \Leftrightarrow z \in L?$ **Falso**

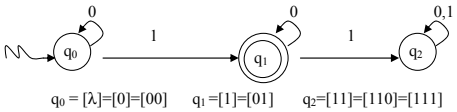
$[11] = [1]? \Longleftrightarrow \forall z \quad 11z \in L \Leftrightarrow 1z \in L?$ **Falso** $\Longrightarrow f(q_1, 1) = [11] = q_2$

$f(q_2, 0) = [110] \qquad f(q_2, 1) = [111]$

$[110] = [11]? \Longleftrightarrow \forall z \quad 110z \in L \Leftrightarrow 11z \in L?$ **Cierto** $\Longrightarrow f(q_2, 0) = q_2$

$[111] = [11]? \Longleftrightarrow \forall z \quad 111z \in L \Leftrightarrow 11z \in L?$ **Cierto** $\Longrightarrow f(q_2, 1) = q_2$

La siguiente figura muestra el autómata que se acaba de construir.



Es importante destacar que el estado $q_0 = [\lambda] = [0]$ representa a todas las cadenas binarias que no tienen ningún 1, el estado $q_1 = [1] = [10]$ representa a las cadenas binarias que tienen un solo 1 y, por tanto, pertenecen al lenguaje (por ese motivo es el único estado final de aceptación), y el estado $q_2 = [11]$ representa a las cadenas que tienen más de un 1. Es evidente que q_0, q_1 y q_2 o, lo que es lo mismo, las clases de equivalencia $[\lambda], [1]$ y $[11]$ constituyen una partición del lenguaje universal $\mathbf{(0+1)^*}$.

Ejemplo 4.15 Utilizando el teorema de Myhill-Nerode es posible demostrar que el lenguaje $L = \{a^n b^n / \quad n \geq 0\}$ no es regular.

Comprobaremos que existe un número infinito de clases de equivalencia para la relación R_L, concretamente $[a] \neq [a^2] \neq [a^3] \neq \ldots$

Bastará demostrar que $[a^i] \neq [a^j]$ cuando $i \neq j$, o lo que es lo mismo, que $a^i \not R_L\, a^j$ cuando $i \neq j$.

Sea $z = b^j$ entonces $a^j z \in L$ y sin embargo $a^i z \notin L \Longrightarrow a^i \not R_L\, a^j$

4.12. Problemas

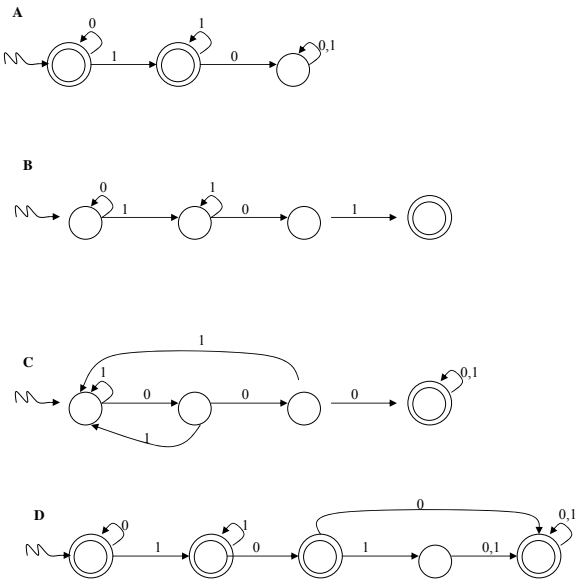
4.1 Minimizar el siguiente autómata

		a	b
$\rightarrow$	q_0	q_1	q_2
	q_1	q_3	q_2
	q_2	q_4	q_1
$*$	q_3	q_4	q_3
$*$	q_4	q_3	q_4

4.2 Construir un AFD mínimo a partir del siguiente AFND:

		0	1
$\rightarrow$	p	$\{q, s\}$	q
$*$	q	$-$	q
$*$	s	$-$	p

4.3 ¿Qué lenguajes reconocen los siguientes autómatas?



4.4 A partir de los siguiente Autómatas Finitos:

		a	b	λ
$\rightarrow$	A	$\{B, D\}$	$-$	$-$
	B	C	$-$	A
$*$	C	$-$	$-$	$-$
$*$	D	$-$	A	$-$

		0	1
$\rightarrow$	A	B	C
	B	$-$	D
	C	E	$-$
$*$	D	B	$-$
$*$	E	$-$	C

construir una GLD y una GLI, limpias y bien formadas, para generar los lenguajes que reconocen dichos autómatas. Calcular las expresiones regulares que describen dichos lenguajes utilizando el método del sistema de ecuaciones.

4.5 Utilizando el método de los conjuntos siguientes calcular el AFD que reconoce a cada uno de los siguientes lenguajes:

1. $ab^*c + a^*c^*$
2. $b^*(a + bc)$
3. $a(bc)^* + ab(cb)^*cd$

4.6 Utilizando el teorema de Myhill-Nerode construir un AFD mínimo que reconozca los lenguajes $L_1 = (ab)^*$ y $L_2 = ab^*$

Tema 5

Gramáticas Independientes del Contexto y Autómatas de Pila

Contenido

5.1. Definición de G.I.C. **74**
5.2. Autómatas de Pila **74**
5.3. Árboles de derivación **77**
5.4. Reconocimiento descendente **80**
5.5. Reconocimiento ascendente **87**
5.6. Propiedades de los L.I.C. **93**
5.7. Problemas **95**

En este tema se estudiarán las Gramáticas Independientes del Contexto (GIC), los lenguajes que éstas definen, llamados lenguajes independientes del contexto (LIC), y los autómatas que reconocen a estos últimos, los Autómatas de Pila (AP). Considerando la jerarquía de Chomsky, también se les llama respectivamente gramáticas y lenguajes de tipo 2. Estas gramáticas, igual que ocurre con las regulares, tienen una gran importancia práctica en la definición de lenguajes de programación, ya que nos permiten formalizar el concepto de sintaxis, de la misma forma que los Autómatas de Pila nos permitirán modelar el funcionamiento del analizador sintáctico, una de las partes fundamentales de un compilador. Análogamente, los lenguajes regulares y los Autómatas Finitos permiten representar los aspectos léxicos y el análisis léxico, respectivamente, de los lenguajes de programación.

5.1. Definición de G.I.C.

En las Gramáticas Independientes del Contexto las producciones son menos restrictivas que en las gramáticas regulares. En este caso, la parte izquierda de la producción también está formada por un único símbolo no terminal, pero no hay restricciones respecto a la parte derecha de la producción. Por lo tanto, las producciones son de la forma:

$$A ::= v \text{ donde } A \in \Sigma_N, \quad v \in \Sigma^*$$

En este tipo de gramáticas, la conversión de A en v se realiza independientemente del contexto en el que se encuentre A, de ahí su nombre.

5.2. Autómatas de Pila

De la misma forma que cualquier lenguaje regular puede ser reconocido por un Autómata Finito, cualquier lenguaje independiente del contexto puede ser reconocido por un Autómata de Pila. Sin embargo, en este caso la equivalencia es menos satisfactoria ya que los Autómatas de Pila no son dispositivos deterministas y el conjunto de los Autómatas de Pila Deterministas sólo permite reconocer a un subconjunto de los lenguajes de tipo 2. Afortunadamente, este subconjunto suele ser suficiente para definir los aspectos más comunes de cualquier lenguaje de programación.

En esencia, un Autómata de Pila es un Autómata Finito al que se le ha incorporado memoria que se gestiona como una pila, con lo que se aumenta su poder funcional. El dispositivo será no determinista y tendrá un número finito de movimientos (o transiciones) a elegir en cada situación. Hay dos tipos de movimientos:

1. Dependiendo del estado actual del Autómata, del símbolo que hay en la cima de la pila y del que hay en la cadena de entrada, habrá que elegir entre un conjunto de posibles transiciones. Cada transición está formada por un posible cambio de estado y por una cadena (puede ser λ) que reemplazará al símbolo que ocupa la cima de la pila. Después de realizar un movimiento se avanza en el análisis de la cinta de entrada.

2. Se le llama λ-movimiento y es similar al anterior salvo que el símbolo de la cadena de entrada no se tiene en cuenta y, por tanto, el análisis de dicha cadena no avanza.

Definiremos formalmente un Autómata de Pila de la siguiente forma:

$$AP = \{Q, \Sigma, f, \Gamma, q_0, z_0, F\}$$

- Q es el conjunto finito de estados

- Σ es el alfabeto de la cinta de entrada
- Γ es el alfabeto de la pila
- $q_0 \in Q$ es el estado inicial
- $z_0 \in \Gamma$ es el símbolo inicial del la pila
- $F \subset Q$ es el conjunto de estados finales
- $f : Q \times (\Sigma \cup \lambda) \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma^*)$

  $f(q, a, z) = \{(p_1, \psi_1), (p_2, \psi_2), \ldots, (p_n, \psi_n)\}$

  Estas transiciones indican que si el Autómata de Pila se encuentra en el estado q, recibe como entrada el símbolo a y z es el símbolo que se encuentra en la cima de la pila, el Autómata puede pasar al estado p_1 y reemplazar en la pila el carácter z por la cadena ψ_1, o bien elegir cualquiera de las otras posibilidades.

Para describir formalmente la configuración de un Autómata de Pila en un instante dado, utilizamos la **Descripción Instantánea(DI)**. La DI estará definida por una tupla (q, w, γ) donde q es el estado actual del Autómata, w es la cadena de símbolos de entrada que aún queda por procesar, y γ es la cadena de los símbolos almacenados en la pila (el carácter más a la izquierda de γ será la cima de la pila).

Utilizaremos la notación $(q, aw, z\gamma) \longrightarrow (p, w, \beta\gamma)$ cuando $(p, \beta) \in f(q, a, z)$. La notación $(q, w, \gamma) \xrightarrow{*} (p, w', \beta)$ indica que se ha pasado de la primera situación a la segunda en un número indeterminado de transiciones.

Lenguaje aceptado por un Autómata de Pila El lenguaje aceptado por un Autómata de Pila se puede definir de dos formas diferentes y equivalentes:

1. De forma análoga a los Autómatas Finitos, es decir, el lenguaje aceptado es el conjunto de entradas que hacen que el Autómata llegue a un estado final. $L(M) = \{w \in \Sigma^* / (q_0, w, z_0) \xrightarrow{*} (p, \lambda, \gamma),\ p \in F\}$

2. El lenguaje está formado por el conjunto de entradas que vacían la pila. En este caso decimos que es un *Autómata de Pila Vacía*. Para esta definición el conjunto F es irrelevante y podemos considerar que $F = \emptyset$.

   $N(M) = \{w \in \Sigma^* / (q_0, w, z_0) \xrightarrow{*} (p, \lambda, \lambda)\}$.

Dado un lenguaje independiente del contexto siempre es posible encontrar un Autómata de Pila con estados finales y un Autómata de Pila Vacía que reconozcan a dicho lenguaje.

Ejemplo 5.1 Construiremos un Autómata de Pila con estados finales para el lenguaje $L = \{0^n 1^n / n \geq 0\}$, generado por la GIC $S ::= 0S1|\lambda$

La estrategia será la siguiente: mientras se procesan los primeros caracteres de la cadena (que deberán ser 0's), éstos se almacenan en la pila. Cuando se llega a la segunda mitad de la cadena y comienzan a llegar 1's, se pasa a otro estado cuya misión será eliminar un 0 de la pila por cada 1 que se procese. Cuando se termine de procesar la cadena, la pila deberá estar vacía (en realidad, sólo almacenará el símbolo inicial de la pila z_0).

El AP se define: $Q = \{q_0, q_1, q_2\}$ $\Sigma = \{0, 1\}$ $\Gamma = \{z_0, 0\}$ $F = \{q_0\}$

Y la función de transición se define de la siguiente forma:

$f(q_0, 0, z_0) = (q_1, 0z_0)$
$f(q_1, 0, 0) = (q_1, 00)$ en el estado q_1 se añaden caracteres a la pila
$f(q_1, 1, 0) = (q_2, \lambda)$
$f(q_2, 1, 0) = (q_2, \lambda)$ en el estado q_2 se eliminan caracteres de la pila
$f(q_2, \lambda, z_0) = (q_0, \lambda)$

En este caso el estado inicial también es estado final debido a que $\lambda \in L$. Cualquier situación que no haya sido definida indicará un error. Por ejemplo, si la cadena comienza con 1 el autómata detectará el error, nótese que la función de transición no está definida para la situación $(q_0, 1, z_0)$.

Ejemplo 5.2 Construiremos un Autómata de Pila Vacía para el lenguaje $L = \{w2w^{-1} / w \in (0+1)^*\}$ generado por la gramática

$$\begin{array}{rcl} S & ::= & 0S0 \\ & | & 1S1 \\ & | & 2 \end{array}$$

La estrategia será la siguiente: cuando se está procesando la primera mitad de la cadena (antes de recibir el 2), los caracteres se almacenan en la pila. Cuando comienza a llegar la segunda mitad de la cadena, cada carácter debe coincidir con el que está en la cima de la pila, si es así se borra la cima y se continua el proceso. Cuando se termine de procesar la cadena, la pila debe estar vacía.

El AP se define $Q = \{q_0, q_1\}$ $\Sigma = \{0, 1, 2\}$ $\Gamma = \{z_0, 0, 1\}$

La función de transición será:

$f(q_0, 0, z_0) = (q_0, 0z_0)$
$f(q_0, 1, z_0) = (q_0, 1z_0)$ esta función se puede simplificar
$f(q_0, 0, 0) = (q_0, 00)$ utilizando como comodín * que
$f(q_0, 1, 0) = (q_0, 10)$ representa a $\{z_0, 0, 1\}$
$f(q_0, 0, 1) = (q_0, 01)$
$f(q_0, 1, 1) = (q_0, 11)$
$f(q_0, 2, z_0) = (q_1, z_0)$ $f(q_0, 0, *) = (q_0, 0*)$
$f(q_0, 2, 0) = (q_1, 0)$ $f(q_0, 1, *) = (q_0, 1*)$
$f(q_0, 2, 1) = (q_1, 1)$ $f(q_0, 2, *) = (q_1, *)$
$f(q_1, 0, 0) = (q_1, \lambda)$ $f(q_1, 0, 0) = (q_1, \lambda)$
$f(q_1, 1, 1) = (q_1, \lambda)$ $f(q_1, 1, 1) = (q_1, \lambda)$
$f(q_1, \lambda, z_0) = (q_1, \lambda)$ $f(q_1, \lambda, z_0) = (q_1, \lambda)$

Ejemplo 5.3 Construiremos un Autómata de Pila Vacía para el lenguaje $L = \{ww^{-1}/w \in (0+1)^*\}$ generado por la GIC $S ::= 0S0|1S1|\lambda$.

En este caso no podremos construir un AP determinista ya que no es posible conocer cuál es el punto medio de la cadena, momento en el que sería necesario cambiar de estado. Cada vez que lleguen dos símbolos iguales seguidos cabe la posibilidad de que estemos en el centro de la cadena por lo que hay que considerar el hecho de que el autómata pueda cambiar de estado. La estrategia será similar a la del ejemplo anterior.

El AP se define $Q = \{q_0, q_1\}$ $\Sigma = \{0, 1\}$ $\Gamma = \{z_0, 0, 1\}$

La función de transición será:

$f(q_0, 0, z_0) = (q_0, 0z_0)$
$f(q_0, 1, z_0) = (q_0, 1z_0)$
$f(q_0, 0, 0) = \{(q_0, 00), (q_1, \lambda)\}$ hay dos posibilidades, considerando o no
$f(q_0, 0, 1) = (q_0, 01)$ el haber llegado al centro de la cadena
$f(q_0, 1, 0) = (q_0, 10)$
$f(q_0, 1, 1) = \{(q_0, 11), (q_1, \lambda)\}$ idem
$f(q_1, 0, 0) = (q_1, \lambda)$
$f(q_1, 1, 1) = (q_1, \lambda)$
$f(q_1, \lambda, z_0) = (q_1, \lambda)$ la pila queda vacía

5.3. Árboles de derivación

El árbol de derivación, al representar las producciones utilizadas para generar una palabra, está indicando además su estructura, lo que resulta determinante para entender su significado. Por esa razón, en los mecanismos para reconocer lenguajes independientes del contexto no es suficiente con indicar si una cadena determinada pertenece o no al lenguaje, también es muy importante que el reconocedor construya el árbol de derivación de dicha cadena.

Cada nodo interno del árbol será un símbolo no terminal de la gramática mientras que las hojas serán los símbolos terminales. Una producción como $A ::= X_1 \dots X_n$ se representará como un subárbol cuyo nodo padre es A siendo sus hijos los símbolos $X_1, \dots, X_n$.

Si en un paso de la construcción del árbol, se aplica una producción al símbolo no terminal que está situado más a la izquierda del árbol, se dice que es una derivación por la izquierda. La misma definición se aplica a derivación por la derecha.

5.3.1. Ambigüedad.

Una gramática es ambigua cuando es posible construir dos o más árboles de derivación diferentes para una misma palabra. El problema de la ambigüedad es muy complejo ya que no existe ningún algoritmo que permita reconocer si una gramática es o no ambigua y, en el caso de que lo sea, tampoco existe ningún algoritmo que permita eliminar dicha ambigüedad (ni siquiera es posible eliminarla en todos los casos). Los lenguajes independientes del contexto para los cuales todas las GIC que los generan son ambiguas, se dice que tienen una *ambigüedad inherente*.

Ejemplo 5.4 (Gramática ambigua) Un ejemplo clásico de gramática ambigua se presenta en la definición de las expresiones aritméticas que aparecen comúnmente en los lenguajes de programación. El siguiente ejemplo simplificado permite definir expresiones en las que intervienen las cuatro operaciones aritméticas básicas con operandos que pueden ser identificadores (id) o constantes (cte). Llamaremos a esta gramática **G_Exp_0**.

$\Sigma_T = \{id, cte, (,), +, -, *, /\}$

$\Sigma_N = \{< expre >, < op >\}$

$S =< expre >$

$$P = \left\{ \begin{array}{l} < expre > ::= < expre >< op >< expre > \\ \qquad\qquad | (< expre >) \\ \qquad\qquad | id \\ \qquad\qquad | cte \\ < op > ::= + \\ \qquad\qquad | - \\ \qquad\qquad | * \\ \qquad\qquad | / \end{array} \right\}$$

Es fácil demostrar que esta gramática es ambigua construyendo dos árboles diferentes para generar la misma expresión, concretamente **id + cte * id**

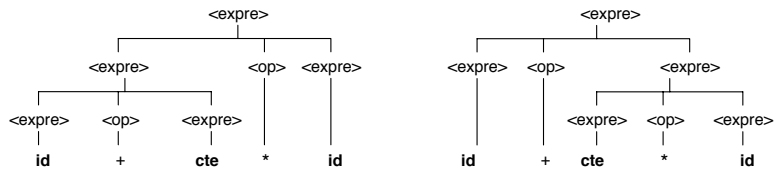


Figura 5.1: Ejemplo de ambigüedad

Analizando la figura 5.1 es fácil comprobar que la ambigüedad está provocada por la ausencia de una jerarquía entre los operadores. En el árbol de la izquierda la operación suma, entre los dos primeros operandos, se lleva a cabo antes que la multiplicación. Sin embargo, en el árbol de la derecha se comenzaría multiplicando los dos últimos operandos y al resultado de esta operación se le sumaría el valor del primer operando. Es evidente que a pesar de que la expresión es correcta, la utilización de cada árbol generaría en cada caso resultados diferentes.

Para resolver este caso de ambigüedad hay que imponer una jerarquía entre los operadores. Como suele ser habitual consideraremos que la multiplicación y la división tienen una prioridad más alta que la suma y la resta. Si aparecen varias operaciones con la misma prioridad se ejecutarán de izquierda a derecha, aunque en este caso el resultado de la expresión siempre sería el mismo. Para definir la jerarquía se van a introducir en la gramática nuevos símbolos no terminales:

$< termino >$ y $< op - adt >$ estarán asociados a los operadores aditivos suma y resta.

$< factor >$ y $< op - mult >$ estarán asociados a los operadores multiplicación y división.

Así llegamos a la siguiente gramática, equivalente a **G_Exp_0**:

$\Sigma_N = \{< expre >, < termino >, < factor >, < op - adt >, < op - mult >\}$

$S =< expre >$

$$P = \left\{ \begin{array}{l} < expre > ::= < expre > < op - adt > < termino > \\ \qquad\qquad | < termino > \\ < termino > ::= < termino > < op - mult > < factor > \\ \qquad\qquad | < factor > \\ < factor > ::= (< expre >) \\ \qquad\qquad | id \\ \qquad\qquad | cte \\ < op - adt > ::= + \\ \qquad\qquad | - \\ < op - mult > ::= * \\ \qquad\qquad | / \end{array} \right\}$$

Con esta nueva gramática, a la que llamaremos **G_Exp_1**, la expresión anterior tendría un único árbol de derivación que se representa en la figura 5.2.

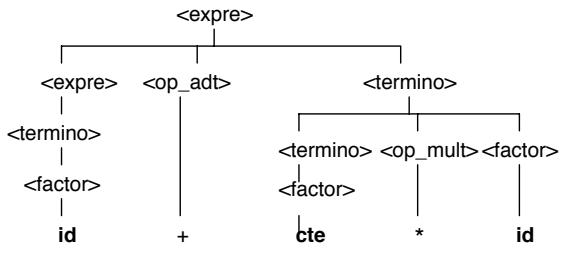


Figura 5.2: Nuevo árbol de derivación

Este árbol representa la estructura de la expresión **id + cte * id**, obligando a que la multiplicación se realice antes que la suma.

5.4. Reconocimiento descendente

En general, un reconocedor es un algoritmo que recibe como entrada una palabra $w \in \Sigma_T^+$, examina sus símbolos de izquierda a derecha e intenta construir un árbol de derivación para dicha palabra. Con el proceso de construcción del árbol se obtiene además la estructura de la palabra, las producciones gramaticales que han de aplicarse y el orden en el que deben utilizarse.

Un reconocedor descendente o analizador sintáctico descendente es un método de reconocimiento de palabras de un LIC que se caracteriza porque construye el árbol de derivación de cada palabra de manera descendente, es decir, desde la raíz hasta las hojas.

A continuación se describirá un reconocedor descendente llamado **LL(1)** (Left-Left(1)). De su nombre, la primera **L** indica que la cadena se analiza de izquierda a derecha, la segunda **L** indica que en cada paso se construye la derivación por la izquierda, y el **1** indica que sólo es necesario un carácter para que el reconocedor decida qué producción debe utilizar en la construcción del árbol.

Como paso previo a la descripción de los reconocedores LL(1) estudiaremos algunos *aspectos* de las GIC que es importante detectar y evitar para el correcto funcionamiento del método.

5.4.1. Simplificación de las GIC

Hay diferentes formas de restringir el formato de las producciones sin mermar por ello el poder generativo de una GIC. En determinadas situaciones nos interesará trasformar una gramática en otra equivalente de forma que las producciones cumplan ciertos requisitos que faciliten la construcción de un reconocedor para dicha gramática. Podemos encontrar en las GIC tres *defectos* que es conveniente eliminar: los prefijos comunes, la recursividad por la izquierda y la ambigüedad.

1. **Eliminación de los prefijos comunes.** Una gramática tiene prefijos comunes cuando hay dos o más producciones que, teniendo la misma parte izquierda, tienen algunos símbolos coincidentes en el comienzo de la parte derecha de la producción. La forma de eliminar los prefijos comunes es muy sencilla, se pretende *sacar factor común* de los símbolos que constituyen el prefijo común. A esta operación se la llama *factorizar por la izquierda*.

   En general, si nos encontramos con la siguiente situación:

   $A ::= \delta\alpha_1|\delta\alpha_2|\ldots|\delta\alpha_n|\beta_1|\ldots|\beta_m$ considerando que $n \geq 2$ y que $|\delta| > 0$

   Estas producciones se pueden sustituir por las siguientes, en las que ha sido necesario añadir un nuevo símbolo no terminal A'.

   $A ::= \delta A'|\beta_1|\ldots|\beta_m \qquad A' ::= \alpha_1|\alpha_2|\ldots|\alpha_n$

2. **Eliminación de la recursividad por la izquierda.** En una gramática es muy frecuente encontrar producciones recursivas. Éstas tienen la forma $X ::= \alpha X\beta$. Serán recursivas por la izquierda cuando su forma sea $X ::= X\beta$, y recursivas por la derecha si son de la forma $X ::= \alpha X$

   La recursividad por la izquierda resulta perjudicial a la hora de construir reconocedores LL(1), por lo que la eliminaremos sustituyéndola por recursividad por la derecha.

Considerando la siguiente situación: $A ::= A\alpha_1|A\alpha_2|\ldots|A\alpha_n|\beta_1|\ldots|\beta_m$

Estas producciones se pueden sustituir por las siguientes, en las que ha sido necesario añadir un nuevo símbolo no terminal A':

$$A ::= \beta_1 A'|\ldots|\beta_m A' \qquad A' ::= \alpha_1 A'|\alpha_2 A'|\ldots|\alpha_n A'|\lambda$$

Ejemplo 5.5 Utilizaremos la gramática de las expresiones aritméticas presentada en la página 78. Para representar de forma más compacta dicha gramática utilizaremos letras mayúsculas para indicar cuales son los símbolos no terminales de acuerdo al siguiente criterio:

$< expre >= E \qquad < factor >= F \qquad < termino >= T$

$< op-adt >= A \qquad < op-mult >= M$

De esta forma, representamos a continuación la gramática no ambigua original a la izquierda y la nueva versión, a la que llamaremos **G_Exp_2**, a la derecha. En esta última versión de la gramática se ha eliminado la recursividad por la izquierda. Como la recursividad por la izquierda aparece en las producciones que tienen a **E** y a **T** en la parte izquierda, será necesario añadir dos nuevos símbolos no terminales a los que llamaremos **E'** y **T'**.

	G_Exp_1		**G_Exp_2**
1.	E::=E A T	1.	E::=T E'
2.	|T	2.	E'::=A T E'
3.	T::=T M F	3.	|λ
4.	|F	4.	T::=F T'
5.	F::=(E)	5.	T' ::= M F T'
6.	|id	6.	|λ
7.	|cte	7.	F::=(E)
8.	A::=+	8.	|id
9.	|-	9.	|cte
10.	M::=*	10.	A::=+
11.	|/	11.	|-
		12.	M::=*
		13.	|/

3. **Ambigüedad.** No existe ningún algoritmo que nos permite eliminar la ambigüedad de forma sistemática. Sin embargo, y como vimos en el ejemplo de la página 78, en ocasiones es posible resolver este problema analizando cuales son sus causas.

5.4.2. Reconocedores LL(1)

En la construcción de los reconocedores LL(1) es muy importante el papel que juegan los símbolos directores de las producciones que, como su nombre indica, dirigirán el análisis de la cadena, es decir, indicarán cuál es la producción que ha de utilizarse en cada paso de la construcción del árbol. Para llegar a la definición de símbolos directores de una producción será necesario conocer otras definiciones previas, todas ellas relativas a una GIC. Los ejemplos que aparecen en esta sección están basados en la gramática **G_Exp_2**.

Definición 5.1 (Cadena o palabra anulable)
Una cadena $w \in \Sigma_N^*$ es anulable si, a partir de ella, y utilizando algunas producciones gramaticales se puede generar la palabra nula (λ).

Es evidente que no puede haber símbolos terminales en una cadena anulable.

Ejemplos de cadenas anulables en G_Exp_2: $E' \quad E'T' \quad T'E'$

Definición 5.2 (Producción anulable)
Una producción $X ::= \alpha$ es anulable si α es una cadena anulable.

Una producción anulable en modo alguno puede considerarse *eliminable*. Esto se debe a que aunque a partir de una producción anulable es posible llegar a λ, también es posible generar otras cadenas utilizando derivaciones diferentes. Las producciones no generativas son, obviamente, anulables.

Ejemplos de producciones anulables en G_Exp_2: Las producciones nº 3 y 6.

Definición 5.3 (Símbolos Iniciales)
Los símbolos iniciales de una cadena $w \in (\Sigma_N \cup \Sigma_T)^*$ son los símbolos terminales por los que pueden comenzar todas las palabras que podamos obtener a partir de ella.

$$INIC(w) = \{a \in \Sigma_T / \ w \xrightarrow{*} a\beta, \ \beta \in \Sigma^*\}$$

Método para calcular los Símbolos Iniciales Hay que tener en cuenta las siguientes consideraciones:

1. Si w comienza por un símbolo terminal es trivial: $w = a\beta, \ a \in \Sigma_T \Longrightarrow INIC(w) = \{a\}$
2. Si w comienza por un símbolo no terminal, hay que considerar la posibilidad de que el símbolo por el que comienza sea anulable y aplicar esta consideración reiteradamente. $w = X\beta, \ X \in \Sigma_N \Longrightarrow$

$$INIC(w) = \begin{cases} INIC(X) \cup INIC(\beta) & \text{si X es anulable} \\ INIC(X) & \text{si X no es anulable} \end{cases}$$

3. Si $X ::= \alpha_1|\alpha_2|\ldots|\alpha_n$ entonces $INIC(X) = INIC(\alpha_1) \cup \ldots \cup INIC(\alpha_n)$

Ejemplos de símbolos iniciales: $INIC(ATE') = \{+, -\}$
$INIC((E)) = \{(\}$ $INIC(FT') = \{(, id, cte\}$ $INIC(MFT') = \{*, /\}$

Definición 5.4 (Forma sentencial)
Es una cadena $w \in (\Sigma_N \cup \Sigma_T)^*$ que puede generarse a partir del símbolo inicial de una GIC utilizando un número indeterminado de producciones.

Definición 5.5 (Símbolos Seguidores)
Los símbolos seguidores de un símbolo no terminal X, son los símbolos terminales que pueden aparecer inmediatamente a la derecha de X en una forma sentencial cualquiera. $SEG(X) = \{a \in \Sigma_T / \ S \xrightarrow{*} \alpha X a \beta \quad \alpha, \beta \in \Sigma^*\}$

Ejemplos de símbolos seguidores:
$E \to TE' \to FT'E' \to (E)T'E' \Longrightarrow) \in SEG(E)$
$E \to TE' \to FT' \to idMFT' \to idMcteT' \Longrightarrow cte \in SEG(M)$

Método para calcular los Símbolos Seguidores de X. Este método se basa en el examen de las producciones de la gramática. El método se divide en dos fases y es necesario calcular simultáneamente los símbolos seguidores de todos los símbolos no terminales.

Fase 1. En la primera fase examinaremos aquellas producciones en las que X aparece en la parte derecha seguido por algún símbolo gramatical.

Situación A. Si hay una producción de la forma: $Y ::= \alpha X a \beta$ donde $a \in \Sigma_T \quad \alpha, \beta \in \Sigma^* \Longrightarrow a \in SEG(X)$

Situación B. Si hay una producción de la forma: $Y ::= \alpha X Z \beta$ donde $Z \in \Sigma_N \quad \alpha, \beta \in \Sigma^* \Longrightarrow INIC(Z) \subset SEG(X)$

Situación C. Si hay una producción de la forma: $Y ::= \alpha X \delta \beta$ donde $\delta \in \Sigma_N^+$ es anulable $\quad \alpha, \beta \in \Sigma^* \Longrightarrow INIC(\beta) \subset SEG(X)$

Una vez que han sido consideradas estas tres situaciones se obtiene una lista provisional de símbolos seguidores que es necesario ampliar con la segunda fase.

Fase 2. En esta segunda fase buscaremos producciones en las que X se encuentre al final de la parte derecha.

Situación D. Si hay una producción de la forma: $Y ::= \alpha X$ donde $\alpha \in \Sigma^* \Longrightarrow SEG(Y) \subset SEG(X)$

Situación E. Si hay una producción de la forma: $Y ::= \alpha X \delta$ donde $\delta \in \Sigma_N^+$ es anulable y $\alpha \in \Sigma^* \Longrightarrow SEG(Y) \subset SEG(X)$

Tras considerar las situaciones D y E con todas las producciones, se obtiene una colección de relaciones de inclusión entre los conjuntos de símbolos seguidores previamente calculados. Si se consideran ordenadamente todas estas inclusiones, tomando como punto de partida la lista provisional calculada en la fase 1, se consigue la lista definitiva.

Si consideramos que el símbolo $ aparece al final de cualquier cadena, hay que tener en cuenta siempre que $\$ \in SEG(S)$

Ejemplo 5.6 Los seguidores de los símbolos no terminales para la gramática **G_Exp_** son los siguientes:

	Fase 1	**Fase 2**
E	) $	
E'		) $
T	+ -	) $
T'		+ -) $
F	* /	+ -) $
A	(id cte	
M	(id cte	

Para llevar a cabo la segunda fase del método se han considerado las siguientes relaciones de inclusión:
$SEG(E) \subset SEG(E') \subset SEG(T) \subset SEG(T') \subset SEG(F)$

Definición 5.6 (Símbolos directores de una producción)
El cálculo de los símbolos directores de una producción ($X ::= \alpha$) es inmediato. Sabiendo calcular los Símbolos Iniciales, los Seguidores y sabiendo identificar las Cadenas anulables, basta con aplicar la siguiente fórmula:

$$DIR(X ::= \alpha) = \begin{cases} INIC(\alpha) & \text{si } \alpha \text{ no es anulable} \\ INIC(\alpha) \cup SEG(X) & \text{si } \alpha \text{ es anulable} \end{cases}$$

Cuando se está construyendo el árbol de derivación correspondiente a una palabra, se analizan de izquierda a derecha los caracteres de dicha palabra y se decide cuál es la producción que se va a utilizar. Para tomar esa decisión hay que considerar que, en cada momento, se debe llevar a cabo la derivación por la izquierda siempre y cuando el carácter que se está procesando en ese momento forme parte de los Símbolos Directores de la producción a utilizar.

Gramáticas LL(1)

Una gramática será LL(1) si es posible construir para ella un reconocedor LL(1) determinista. Para que esto ocurra es necesario que la consulta del siguiente símbolo

de la palabra que se está analizando permita determinar sin incertidumbre la producción que se debe utilizar para proseguir con el análisis. Por lo tanto, para que una gramática sea LL(1) es necesario que todas las producciones que tienen el mismo símbolo en la parte izquierda no tengan ningún Símbolo Director en común. Es decir, considerando que:

$X ::= \alpha_1|\alpha_2|\ldots|\alpha_n$ y que

$D_1 = DIR(X ::= \alpha_1)$
$D_2 = DIR(X ::= \alpha_2)$
$\ldots$
$D_n = DIR(X ::= \alpha_n)$

La gramática será LL(1) si $D_i \cap D_j = \emptyset,\ i \neq j \quad i, j \in \{1, \ldots, n\}$

Para que una GIC sea LL(1) es imprescindible que no sea ambigua, que no tenga prefijos comunes, ni recursividad por la izquierda.

Ejemplo 5.7 Los símbolos directores para las producciones de la gramática **G_Exp_1** son los siguientes:

$DIR_1 = DIR(E ::= TE') = \{(, cte, id\}$
$DIR_2 = DIR(E' ::= ATE') = \{+, -\}$
$DIR_3 = DIR(E' ::= \lambda) = SEG(E') = \{), \$\}$
$DIR_4 = DIR(T ::= FT') = \{(, cte, id\}$
$DIR_5 = DIR(T' ::= MFT') = \{*, /\}$
$DIR_6 = DIR(T' ::= \lambda) = SEG(T') = \{+, -,), \$\}$
$DIR_7 = DIR(F ::= (E)) = \{(\}$
$DIR_8 = DIR(F ::= id) = \{id\}$
$DIR_9 = DIR(F ::= cte) = \{cte\}$
$DIR_{10} = DIR(A ::= +) = \{+\}$
$DIR_{11} = DIR(A ::= -) = \{-\}$
$DIR_{12} = DIR(M ::= *) = \{*\}$
$DIR_{13} = DIR(M ::= /) = \{/\}$

Es fácil comprobar que es una gramática LL(1) ya que:

$DIR_2 \cap DIR_3 = \emptyset \qquad DIR_5 \cap DIR_6 = \emptyset \qquad DIR_7 \cap DIR_8 \cap DIR_9 = \emptyset$
$DIR_{10} \cap DIR_{11} = \emptyset \qquad DIR_{12} \cap DIR_{13} = \emptyset$

Ejemplo 5.8 Gramática que no es LL(1). La gramática que se describe a continuación permitirá representar la clásica estructura alternativa de cualquier lenguaje de programación. Es una gramática ambigua y por ese motivo no es LL(1) como comprobaremos a continuación.

$\Sigma_N = \{S, E, R\}$
$\Sigma_T = \{i, t, a, e, b\}$

Los símbolos gramaticales tienen el siguiente significado:

S = Sentencia E = Expresión R = Resto de la sentencia

i = if t = then a = accion e = else b = boolean

$$P = \left\{ \begin{array}{l} S ::= iEtSR \\ \quad\quad |a \\ R ::= eS \\ \quad\quad |\lambda \\ E ::= b \end{array} \right\}$$

Símbolos seguidores:

	Fase 1	**Fase 2**
S	e \$	
R		e \$
E	t	

Símbolos directores de las producciones:

$DIR_1 = DIR(S ::= iEtSR) = \{i\}$
$DIR_2 = DIR(S ::= a) = \{a\}$
$DIR_3 = DIR(R ::= eS) = \{e\}$
$DIR_4 = DIR(R ::= \lambda) = SEG(R) = \{e, \$\}$
$DIR_5 = DIR(E ::= b) = \{b\}$

Como $DIR_3 \cap DIR_4 = \{e\}$ podemos afirmar que la gramática no es LL(1)

5.5. Reconocimiento ascendente

El reconocimiento ascendente o análisis sintáctico ascendente se caracteriza por construir el árbol de derivación de manera ascendente, es decir, desde las hojas hasta la raíz.

A continuación se describirá un reconocedor ascendente llamado **LR(1)** (Left-Right(1)). En este nombre, **L** indica que la cadena se analiza de izquierda a derecha, **R** indica que en cada paso se construye la derivación por la derecha en orden inverso, y el **1** indica que sólo es necesario un carácter para que el reconocedor decida qué acción se debe realizar.

Una gramática es LR(1) si es posible construir para ella un reconocedor LR(1) determinista.

Algunas de las ventajas de los reconocedores LR(1):

- Son más potentes que los reconocedores LL(1). Es decir, el conjunto de los lenguajes LL(1) está contenido en el conjunto de los LR(1).
- Un analizador LR(1) detecta un error en una cadena tan pronto como sea posible.

- Prácticamente todas las gramáticas que definen los lenguajes de programación son LR(1).

La desventaja de los reconocedores LR(1) es que su construcción *a mano* es más compleja. Sin embargo, existen herramientas (YACC) que permiten la construcción automática de este tipo de reconocedores.

El funcionamiento de un reconocedor LR(1) depende de su Tabla de Acciones. En esta tabla encontraremos dos tipos de procesos:

1. **Desplazamientos**. Indican la transición de un estado a otro. Se representan como D_i, donde i identifica el estado al que se va a pasar.

2. **Reducciones**. Esta acción se lleva a cabo cuando en el árbol aparece la parte derecha de una producción y se añade la parte izquierda, subiendo un nivel en la construcción del árbol. Se representa como R_i, donde i permite identificar la producción utilizada en el proceso de reducción.

Los analizadores LR(1) también utilizan una pila en la que se van almacenando los caracteres que se van procesando así como los estados por los que el reconocedor ha pasado.

Veamos con un sencillo ejemplo como funcionaría un reconocedor LR(1), conociendo su Tabla de Acciones. Posteriormente estudiaremos cómo construir dicha tabla.

Ejemplo 5.9 Sea la gramática definida con los siguientes símbolos:
$\Sigma_T = \{a, (,)\}$ $\Sigma_N = \{S, A\}$ y las producciones:
1. $S ::= A$ 2. $A ::= a$ 3. $A ::= (a)$

Las acciones se definen en función del estado del autómata y del símbolo de la cadena de entrada que se procesa en cada momento. Hay que tener en cuenta que las situaciones no definidas se consideran situaciones de error. La Tabla de Acciones asociada a este ejemplo se muestra a continuación. En la tabla aparece el símbolo **$**, que indica el final de la cadena de entrada.

	$	**a**	(	)	**A**
q_0		D_2	D_3		D_1
q_1	**R_1**				
q_2	R_2				
q_3		D_4			
q_4				D_5	
q_5	R_3				

Utilizando esta tabla veamos como se procesaría la cadena **(a)**:

Entrada	Pila	Acción
$(a)\$$	q_0	Desplazar a q_3
$a)\$$	$q_0(q_3$	Desplazar a q_4
$)\$$	$q_0(q_3aq_4$	Desplazar a q_5
$\$$	$q_0(q_3aq_4)q_5$	Reducir por la 3ª producción $A ::= (a)$
$\$$	q_0A	Desplazar a q_1
$\$$	q_0Aq_1	Reducir por $S ::= A$ (el proceso termina)

Veamos cómo se procesaría una cadena incorrecta como **a)**

Entrada	Pila	Acción
$a)\$$	q_0	Desplazar a q_2
$)\$$	q_0aq_2	Error

Se produce un error ya que en el estado q_2 no hay ninguna acción asociada al símbolo).

5.5.1. Construcción de la Tabla de Acciones

La Tabla de Acciones se construye al mismo tiempo que se definen los estados. Para llevar este trabajo a cabo es necesario introducir el concepto de **LR-item** que indicará el progreso del análisis de la cadena.

Denominamos LR-item a una producción a la que se le coloca una marca (un punto) en algún lugar de la parte derecha. Esta marca indica qué parte de la cadena ha sido ya procesada y cuál es la que queda por analizar. Además, en los LR-items hay un conjunto de símbolos terminales (separados por una coma de la producción) a los que se llama *símbolos directores del LR-item*.

Ejemplo de un LR-item: $[\mathbf{A ::= x.By, w_1, w_2}]$ (suponemos que A ::= xBy es una producción de la gramática y $w_1, w_2 \in \Sigma_T$)

Los estados del analizador LR(1) serán conjuntos de LR-items y cada LR-item, según la posición en la que se encuentre el punto, indicará la acción que se debe llevar a cabo.

- Si detrás de la marca hay algún símbolo, la acción que se realizará será un **desplazamiento** al estado que contenga un LR-item similar pero con la marca desplazada en una posición a la derecha.

  Es decir, si [A ::= x.By, w] $\in p$ y [A ::= xB.y, w] $\in q$ entonces

  Acción(p,B)= Desplazamiento a q

- Si la marca está situada al final de la producción, la acción a realizar será una **reducción** por la producción que representa el LR-item, siempre y cuando el carácter analizado sea uno de sus símbolos directores.

Por ejemplo, si [A ::= xBy., w] $\in p$ entonces

Acción(p,w)= Reducir por la producción A ::= xBy

En cada instante, el analizador va a tener una configuración (descripción instantánea) determinada que viene dada por el contenido de la pila y por el fragmento de la cadena de entrada que aún no ha sido procesado. En la pila se almacenan los estados por los que ha ido pasando el analizador así como los símbolos (terminales y no terminales) que están ubicados en las zonas superiores del árbol de derivación.

Por ejemplo: $(q_0X_1q_1 \dots X_mq_m, a_ja_{j+1} \dots a_n\$) \quad q_i \in Q \quad X_i \in \Sigma \quad a_i \in \Sigma_T$

Veamos como varía la configuración del analizador en función de la acción que se realice:

Desplazamiento Si $Accion(q_m, a_j) = D_r$

La nueva configuración será: $(q_0X_1q_1 \dots X_mq_ma_jq_r, a_{j+1} \dots a_n\$)$

Se añade a la pila el carácter procesado y el estado actual.

Reducción Si $Accion(q_m, a_j) = R_i$ y la producción i es $A ::= \beta$ donde $|\beta| = r$ supondremos que los últimos r símbolos almacenados en la pila coinciden con β. Es decir, $\beta = X_{m-r+1} \dots X_m$.

En este caso, se sustituyen los últimos r símbolos de la pila (y los estados que les acompañan) por A.

La nueva configuración será: $(q_0X_1q_1 \dots X_{m-r}q_{m-r}Aq_p, a_ja_{j+1} \dots a_n)$ donde $Accion(q_{m-r}, A) = D_p$

En este caso no ha sufrido ninguna modificación el fragmento de cadena que aún queda por procesar. Aunque a_j ha sido tenido en cuenta para decidir la operación a realizar, no podemos considerar que haya sido procesado, es decir, no forma parte de la pila ni tampoco del árbol de derivación.

Método para la construcción de los LR-items Los estados se van creando en dos fases: en primer lugar se construye lo que podemos denominar el *núcleo* del estado, posteriormente, y siempre que sea necesario, se añaden otros LR-items hasta *cerrar* el estado.

Determinación del núcleo de q_0. Para cada una de las producciones que tienen al símbolo inicial en la parte izquierda (S::=x), añadir al estado q_0 el LR-item [S::= .x, $]

Cierre de un estado. Si [A::=x.By,w] $\in$ q y B $\in \Sigma_N$ hay que añadir al estado **q** LR-items construidos a partir de todas las producciones de la gramática que tienen a B en la parte izquierda.

Para la producción B::=z, hay que añadir el LR-item [B::= .z, u] donde u = INIC(yw).

Creación del núcleo de un nuevo estado. Si existe un LR-item de la forma [A::= se crea un nuevo estado con el LR-item [A::=xY.z,w]

Ejemplo 5.10 Veamos cómo se crea la Tabla de Símbolos para el ejemplo anterior:

1. Crear el núcleo del estado q_0

   [S::=.A, $] $\in q_0$

2. Cerrar q_0

   q_0 = {[S::=.A, $], [A::=.(a), $], [A::=.a, $]}

   Analizando estos tres LR-items y los que se construyen después, es evidente que:

   $Accion(q_0, A) = D_1 \quad Accion(q_0, () = D_3 \quad Accion(q_0, a) = D_2$

3. Crear nuevos estados (en este caso están ya cerrados)

   q_1 = {[S::=A., $]} $Accion(q_1, \$) = R_1$
   q_2 = {[A::=a., $]} $Accion(q_2, \$) = R_2$
   q_3 = {[A::=(.a), $]} $Accion(q_3, a) = D_4$
   q_4 = {[A::=(a.), $]} $Accion(q_4,)) = D_5$
   q_5 = {[A::=(a)., $]} $Accion(q_5, \$) = R_3$

Ejemplo 5.11 Cálculo del reconocedor LR(1) para la siguiente gramática:
$\Sigma_T = \{a, b\}$ $\Sigma_N = \{S, A, B\}$ y las producciones:

1. $S ::= A$	2. $A ::= BA$	4. $B ::= aB$
	3. $\lvert\lambda$	5. $\lvert b$

Esta gramática genera el lenguaje $(\mathbf{a}^*\mathbf{b})^*$
Definición de los estados:

q_0 = {[S::=.A, $],[A::=.BA, $],[A::=., $],[B::=.aB, a, b, $],[B::=.b, a, b,$]}
q_1 = {[S::=A., $]}
q_2 = {[A::=B.A,$], [A::=.BA,$], [A::=.,$], [B::=.aB,a,b,$], [B::=.b,a,b,$]}
q_3 = {[B::=a.B, a, b, $], [B::=.aB, a, b, $], [B::=.b, a, b,$]}
q_4 = {[B::=b., a, b,$]}
q_5 = {[A::=BA., $]}
q_6 = {[B::=aB., a, b, $]}

Analizando los LR-items de los estados se construye la siguiente tabla de acciones:

	$	a	b	A	B
q0	R_3	D_3	D_4	D_1	D_2
q1	**R1**				
q2	R_3	D_3	D_4	D_5	D_2
q3		D_3	D_4		D_6
q4	R_5	R_5	R_5		
q5	R_2				
q6	R_4	R_4	R_4		

Utilizando esta tabla, veamos como se procesaría la cadena **ab**:

Entrada	**Pila**	**Acción**
$ab\$$	q_0	Desplazar a q_3
$b\$$	q_0aq_3	Desplazar a q_4
$\$$	$q_0aq_3\underline{bq_4}$	Reducir por la 5ª producción $B ::= b$ (desp. q_6)
$\$$	$q_0\underline{aq_3Bq_6}$	Reducir por la 4ª producción $B ::= aB$ (desp. q_2)
$\$$	q_0Bq_2	Reducir por la 3ª producción $A ::= \lambda$ (desp. q_5)
$\$$	$q_0\underline{Bq_2Aq_5}$	Reducir por la 2ª producción $A ::= BA$ (desp. q_1)
$\$$	$q_0\underline{Aq_1}$	Reducir por la 1ª producción $S ::= A$ (finaliza el p

Veamos cómo se procesaría una cadena incorrecta como **a**

Entrada	**Pila**	**Acción**
$a\$$	q_0	Desplazar a q_3
$\$$	q_0aq_3	Error

Gramáticas LR(1)

Una gramática es LR(1) siempre que sea posible construir un reconocedor LR(1) que sea determinista. Para que esto pueda ocurrir deben cumplirse las siguientes condiciones:

1. La gramática debe ser *aumentada*, es decir, el símbolo inicial no debe aparecer nunca en la parte derecha de ninguna producción. Si no es así, basta con añadir una producción como **S' ::= S**, en la que S' se convierte en el nuevo símbolo inicial de la gramática.

2. No deben aparecer conflictos a la hora de construir la tabla de acciones. Hay tres tipos de conflicto:

   a) **Conflicto desplazamiento/desplazamiento.** Aparece cuando en un estado **q** hay dos items del siguiente tipo: $[A ::= \alpha.a\beta, u_1]$ y $[B ::= \gamma.a\delta, u_2]$. La acción asociada al estado **q** ante la llegada del símbolo **a** podría ser un desplazamiento al estado que contiene al LR-item $[A ::= \alpha a.\beta, u_1]$ o un desplazamiento al que contiene a $[B ::= \gamma a.\delta, u_2]$.

b) **Conflicto reducción/reducción.** Aparece cuando en un estado **q** hay dos items del siguiente tipo: $[A ::= \alpha., u]$ y $[B ::= \beta., u]$. La acción asociada al estado **q** ante la llegada del símbolo **u** podría ser una reducción utilizando la producción $A ::= \alpha$ o una reducción utilizando la producción $B ::= \beta$.

c) **Conflicto desplazamiento/reducción.** Aparece cuando en un estado **q** hay dos items del siguiente tipo: $[A ::= \alpha.a\beta, u_1]$ y $[B ::= \beta., a]$. La acción asociada al estado **q** ante la llegada del símbolo **a** podría ser un desplazamiento al estado que contiene al LR-item $[A ::= \alpha a.\beta, u_1]$ o una reducción utilizando la producción $B ::= \beta$.

Ejemplo 5.12 Gramática que no es LR(1).

$\Sigma_T = \{a, b, c\}$ $\Sigma_N = \{S, A, B\}$ y las producciones:

1. $S ::= cAB$	3. $A ::= Aa$	5. $B ::= Bb$
2. $\quad \mid c$	4. $\quad \mid a$	6. $\quad \mid b$

Esta gramática genera el lenguaje $\mathbf{c + ca^+b^+}$

Comenzamos la definición de los estados:

$q_0 = \{$[S::=.cAB, $], [S::=.c, $]$\}$
$q_1 = \{$[S::=c.AB,$],[A::=.Aa, b],[A::=.a, b]$\}$
$q_2 = \{$[S::=c., $]$\}$

No es necesario continuar ya que con esta información el conflicto es evidente: ¿qué acción hay que realizar si estando en el estado q_0 llega el símbolo **c**? Si atendemos al primer LR-item deberíamos desplazarnos a q_1 pero si atendemos al segundo el desplazamiento debería hacerse a q_2, por lo tanto es imposible construir un reconocedor LR(1) determinista.

5.6. Propiedades de los L.I.C.

Teorema 5.1
El conjunto de los LIC está cerrado para la unión, la concatenación y el cierre de Kleene.

Teorema 5.2
El conjunto de los LIC no está cerrado para la intersección ni para la complementación.

Ejemplo 5.13 Sean $L_1 = \{a^n b^n c^n,\ n \geq 0\}$, $L_2 = \{a^n b^n c^m,\ n, m \geq 0\}$ y $L_3 = \{a^n b^m c^m,\ n, m \geq 0\}$. L_2 y L_3 son independientes del contexto, sin embargo, $L_1 = L_2 \cap L_3$ no lo es (se puede comprobar utilizando el lema del bombeo).

Como $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$, si los LIC fueran cerrados para la complementación también lo serían para la intersección y hemos comprobado en el ejemplo anterior que esto no es cierto.

Teorema 5.3
Si L es un LIC y R es un Lenguaje regular, entonces $L \cap R$ es un LIC.

Ejemplo 5.14 Sea $L_1 = \{a^n b^m a^n b^m,\ n, m \geq 0\}$, $L_2 = \{ww,\ w \in (0+1)^*\}$ y $L_3 = a^+ b^+ a^+ b^+$. Tenemos que $L_1 = L_2 \cap L_3$. Como L_1 no es un LIC (se puede demostrar con el lema del bombeo) y L_3 es un lenguaje regular entonces, aplicando el teorema anterior, se deduce que L_2 tampoco es un LIC.

Definición 5.7 (Sustitución)
Sean Σ y Γ dos alfabetos, se define una sustitución como una función
$s : \Sigma \longrightarrow \mathcal{P}(\Gamma^*)$ *tal que* $\forall a \in \Sigma \quad s(a)$ *es un LIC.*
Esta función se puede extender a cadenas de caracteres y a lenguajes de forma natural.

Definición 5.8 (Homomorfismo)
Un homomorfismo es un caso particular de sustitución en el que $h : \Sigma \longrightarrow \Gamma^*$

Teorema 5.4
El conjunto de los LIC está cerrado para las sustituciones y (como caso particular) para los homomorfismos.

5.6.1. El lema del bombeo para LIC(*pumping lemma*)

El lema del bombeo enuncia una propiedad que deben cumplir todos los lenguajes independientes del contexto. El hecho de comprobar que un lenguaje no cumple dicha propiedad es suficiente para demostrar que no es independiente del contexto. Sin embargo, en ningún caso este lema servirá para demostrar que un lenguaje es LIC.

Lema 5.1 (El lema del bombeo para LIC)
Sea L un lenguaje independiente del contexto, entonces existe una constante asociada al lenguaje $n > 0$, de manera que $\forall z \in L$ tal que $|z| \geq n$, se cumple que z se puede descomponer en cinco partes $z = uvwxy$ que verifican:

1. $|vx| \geq 1$
2. $|vwx| \leq n$
3. $\forall i \geq 0$ *se cumple que* $uv^i wx^i y \in L$

5.7. Problemas

5.1 Construir un Autómata de Pila Vacía para los siguientes lenguajes definidos sobre el alfabeto $\Sigma = \{a, b, c, 0, 1\}$

1. $L_1 = \{a^{2n}b^n,\ n \geq 0\}$
2. $L_2 = \{awbw^{-1}c,\ w \in (0+1)^*\}$
3. $L_3 = \{ab^*c\}$
4. $L_4 = \{ab^ncd^n,\ n \geq 0\}$

5.2 Demuestra que cada una de las siguientes gramáticas de tipo 2 es ambigua y encuentra otra gramática equivalente que no lo sea

1. $S ::= A \qquad A ::= AA|a|b$
2. $S ::= A|B \qquad A ::= aAb|ab \qquad B ::= abB|\lambda$
3. $S ::= aB|Ab \qquad A ::= aA|\lambda \qquad B ::= bB|\lambda$

5.3 Para cada una de las siguientes gramáticas de tipo 2 hay que comprobar si son LL(1) y/o LR(1). Si lo son hay que construir el correspondiente reconocedor y si no lo son hay que explicar el motivo

1. $S ::= aA \qquad A ::= bA|\lambda$
2. $S ::= aA \qquad A ::= Ab|b$
3. $S' ::= S \qquad S ::= 0S0|A \qquad A ::= 1A|\lambda$
4. $S ::= A \qquad A ::= AB|\lambda \qquad B ::= aB|b$
5. $S ::= E \qquad E ::= E + T|T \qquad T ::= a|(E)$
6. $S ::= cAb \qquad A ::= aA|\lambda$

5.4 Utliza el lema del bombeo para demostrar que los siguientes lenguajes no son independientes del contexto

1. $L_1 = \{a^nb^nc^n,\ n \geq 0\}$
2. $L_2 = \{a^nb^nc^m,\ m \geq n \geq 0\}$
3. $L_3 = \{a^nb^mc^nd^m,\ n, m \geq 0\}$
4. $L_4 = \{a^nb^nc^m,\ n \neq m \quad n, m \geq 0\}$

5. $L_5 = \{a^n b^n c^m,\ 0 \leq n \leq m \leq 2n\}$

5.5 Demuestra que la gramática que aparece en el ejemplo 5.8 (página 86) es ambigua y no es LR(1)

5.6 Demuestra que una gramática LR(1) no puede ser ambigua

5.7 Demuestra que el lenguaje $L = \{a^n b^m c^m d^n,\ n, m \geq 0\}$ es independiente del contexto

5.8 Diseña una gramática que genere paréntesis anidados y demuestra que es LR(1)

5.9 Modifica la siguiente gramática de manera que sea LL(1) y demuestra que realmente lo es

$\Sigma_N = \{D, T, L\}$ $S = D$ $\Sigma_T = \{entero, real, id, ,\}$

$D ::= TL$ $T ::= entero|real$ $L ::= L, id|id$

Tema 6

Gramáticas Atribuidas

Contenido

6.1. Concepto de Semántica y de Gramática Atribuida . . . 97
6.2. Atributos heredados y sintetizados 99
6.3. Gramáticas S-atribuidas y L-Atribuidas 100
6.4. Problemas 104

Hasta ahora hemos visto como las gramáticas regulares (tipo 3) son adecuadas para representar las características morfológicas de los elementos básicos de un lenguaje de programación. Igualmente, las gramáticas independientes del contexto (tipo 2) lo son para representar las características sintácticas. Sin embargo, dichas gramáticas no son lo suficientemente potentes como para representar los aspectos semánticos de un lenguaje de programación. Veremos, a lo largo de este tema, cómo las gramáticas atribuidas pueden ser útiles para realizar esta labor.

6.1. Concepto de Semántica y de Gramática Atribuida

La semántica, considerada desde el punto de vista de los lenguajes de programación, se ocupa del significado que tienen las instrucciones de los programas escritos en un determinado lenguaje.

Para poder realizar un análisis semántico de un programa éste debe ser correcto tanto desde el punto de vista léxico como desde el sintáctico.

En la semántica de un lenguaje de programación pueden distinguirse dos aspectos: estático y dinámico.

- La **semántica estática** se ocupa de las condiciones que deben cumplir las construcciones de un programa para que su significado sea correcto.

Ejemplo 6.1 Una sentencia de asignación en C como $\mathbf{A} := \mathbf{5} + \mathbf{B}$ es correcta desde un punto de vista léxico y sintáctico, pero para que sea semánticamente correcta la variable **B** debe ser de un tipo al que se le pueda aplicar el operador + y, además, el valor resultante de evaluar la expresión $\mathbf{5} + \mathbf{B}$ debe ser de un tipo compatible con el de la variable sobre la que se efectúa la asignación (**A**).

- La **semántica dinámica** se ocupa del significado de una construcción con objeto de que pueda ser traducido a código directamente ejecutable por la máquina.

  Ejemplo: A partir de una sentencia de asignación **V** ← **Expresión** debe generarse código que permita:

  1. obtener la dirección de memoria de la variable **V**
  2. calcular el valor de la expresión asignada
  3. almacenar el valor calculado en el paso 2 en la dirección de memoria obtenida en el paso 1

La semántica de un lenguaje de programación, en sus dos vertientes estática y dinámica, es más difícil de especificar que los aspectos léxicos y sintácticos de un lenguaje. Se podría utilizar el lenguaje natural pero su falta de precisión provocaría probablemente ambigüedad. También resultaría difícil confirmar que se ha realizado una especificación completa de todas las características semánticas del lenguaje. Las gramáticas atribuidas son un buen mecanismo formal para dar solución a este problema.

Las gramáticas atribuidas pueden considerarse como una ampliación de las independientes del contexto que tiene por objeto dotar a la gramática de capacidad para especificar correctamente la semántica de un lenguaje de programación. Para conseguirlo, a los símbolos de la gramática se les asocian atributos que permiten definir ciertas características semánticas de los mismos, y a las producciones se les asocian funciones semánticas que permiten calcular, evaluar y controlar dichos atributos.

Definición 6.1 (Gramática atribuida)
Si una Gramática Independiente del Contexto se define como

$$G = \{\Sigma_T, \Sigma_N, S, P\}$$

la gramática atribuida se define como

$$GA = \{G, AS, FS\}, \text{ donde}$$

- *AS son los* ***atributos semánticos*** *asociados a los símbolos de la gramática.*
- *FS son las* ***funciones semánticas*** *asociadas a las producciones de la gramática.*

De la misma forma que G permite definir los aspectos sintácticos de un lenguaje, GA permite definir los aspectos semánticos.

Un mismo símbolo de la gramática puede tener asociados varios atributos que permitan representar diferentes aspectos de dicho símbolo. Por ejemplo, un símbolo llamado **X** que representara variables de un lenguaje podría tener dos atributos, uno llamado **tipo** que podría tener los valores: *real, entero, lógico,...* y otro llamado **valor** cuyo valor dependería del tipo mencionado anteriormente. Podríamos representarlo así:

X	**X.tipo**	**X.valor**
	entero	*534*

Las funciones semánticas definen las relaciones que se cumplen entre los atributos de los distintos símbolos de la gramática. Estas relaciones permiten calcular los valores de unos atributos en función de los valores de otros.

Cuando se desea describir las funciones semánticas asociadas a una producción, éstas deben aparecen encerradas entre llaves al final o en algún punto intermedio de la producción. Si un símbolo aparece varias veces en la misma producción (por ejemplo, en las producciones recursivas) se utilizan subíndices para distinguir unas instancias de otras. El subíndice 0 se reserva para hacer referencia al símbolo repetido cuando éste aparece en la parte izquierda de la producción y los subíndices 1, 2, ... se utilizan en orden para las diferentes apariciones del símbolo en la parte derecha de la producción. Por ejemplo:

$$expr ::= expr + expr \quad \{expr_0.valor = expr_1.valor + expr_2.valor\}$$

Esta producción indica que una expresión aritmética puede construirse como la suma de otras dos expresiones más simples y, en este caso, la acción semántica indica que el valor de la expresión resultante será la suma de las dos expresiones simples.

6.2. Atributos heredados y sintetizados

Una producción gramatical puede representarse gráficamente como un árbol. Por ejemplo, la producción $A ::= x\ y\ z$ se puede representar así:

$$\overbrace{x \quad y \quad z}^{A}$$

Empleando la terminología típica de las estructuras arbóreas podemos decir que A es el nodo *padre* de x, o que x es un nodo *hermano* de y. De acuerdo a esta terminología podemos distinguir dos tipos diferentes de atributos.

Los atributos pueden ser **sintetizados** y **heredados**. El valor de los atributos sintetizados se calcula en función de los atributos de los nodos hijos. El valor de los

atributos heredados se calcula en función de los atributos de los nodos hermanos y/o padre.

6.3. Gramáticas S-atribuidas y L-Atribuidas

En las gramáticas atribuidas el principal problema radica en la evaluación de los atributos. Gráficamente se puede pensar en el árbol de derivación que representa la estructura sintáctica de manera que los atributos están localizados en los diferentes nodos del árbol. Así, un recorrido apropiado del árbol puede permitir calcular todos lo atributos. Pero si no se imponen ciertas restricciones este proceso puede ser largo y costoso.

En función de que los atributos de una gramática sean heredados o sintetizados se pueden definir las gramáticas S o L atribuidas. Dependiendo del tipo de la gramática atribuida resultará recomendable utilizar un determinado mecanismo para construir y procesar el árbol sintáctico.

6.3.1. Gramáticas S-atribuidas

Las gramáticas **S-atribuidas** son aquellas cuyos símbolos sólo tienen atributos sintetizados.

Con las gramáticas S-atribuidas es muy eficaz utilizar un reconocedor ascendente (por ejemplo, un reconocedor del tipo LR) ya que al mismo tiempo que se construye el árbol sintáctico se pueden ir evaluando los atributos. El programa PCYACC genera un reconocedor sintáctico-semántico que realiza esta tarea.

Ejemplo 6.2 El siguiente ejemplo de gramática S-atribuida se utiliza para generar expresiones aritméticas. Por simplicidad, consideraremos que las únicas operaciones que se pueden realizar son la suma y la multiplicación. Los símbolos **expr** y **num**, que representan las expresiones aritméticas y los números que aparecen en ellas respectivamente, tienen ambos un atributo sintetizado llamado *valor* que permite calcular el valor de las expresiones aritméticas analizadas. Por simplicidad, el símbolo **num** será considerado como terminal y su valor habrá sido obtenido en el proceso de análisis léxico previo.

Las acciones semánticas de estas producciones permitirán obtener el valor de una expresión aritmética a partir de los valores de expresiones más básicas.

expr ::= **num**	$\{expr.valor = num.valor\}$
expr ::= **expr** + **expr**	$\{expr_0.valor = expr_1.valor + expr_2.valor\}$
expr ::= **expr** $*$ **expr**	$\{expr_0.valor = expr_1.valor * expr_2.valor\}$

La figura 6.1 muestra el árbol de derivación de la expresión **9 + 7 * 5**. Al mismo tiempo que el árbol se construye de manera ascendente se calculan los atributos

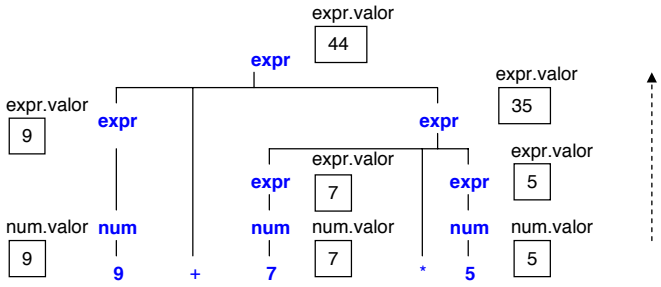


Figura 6.1: Gramática S-atribuida

sintetizados de todos los símbolos. Finalmente se obtiene el valor de la expresión completa.

6.3.2. Gramáticas L-atribuidas

Las gramáticas L-atribuidas tienen atributos sintetizados y heredados, pero a los atributos heredados se les imponen algunas restricciones con objeto de facilitar el diseño de un algoritmo que permita evaluar todos los atributos de la gramática. Los atributos heredados sólo pueden depender de los atributos heredados del nodo padre o de cualquier atributo de sus nodos hermanos, siempre que esos nodos hermanos aparezcan a su izquierda en la producción. Es decir, si tenemos una producción

$$A ::= x_1 x_2 \dots x_n$$

Los atributos heredados de x_i solo pueden calcularse en función de :

1. los atributos heredados de A
2. cualquier atributo (heredado o sintetizado) de $x_1 \dots x_{i-1}$

Ejemplo 6.3 Veamos con un ejemplo abstracto cómo especificar claramente un fragmento de gramática L-atribuida. Supongamos que tenemos la producción

$$A \quad ::= \quad x \quad y \quad z$$

- el símbolo A tiene un atributo sintetizado(a_1) y otro heredado (a_2)
- el símbolo x tiene sólo un atributo heredado (x_1)

- el símbolo y tiene un atributo sintetizado(y_1) y otro heredado (y_2)
- el símbolo z tiene un atributo sintetizado(z_1)

Si deseamos representar los atributos de cada símbolo dentro de la producción, lo haremos de la siguiente forma:

$$A.a_1 \uparrow a_2 \downarrow ::= x.x_1 \downarrow \quad y.y_1 \uparrow y_2 \downarrow \quad z.z_1 \uparrow$$

donde el símbolo $\downarrow$ indica que el atributo es heredado y el símbolo $\uparrow$ indica que es sintetizado.

Si además se quisieran representar las acciones semánticas, éstas deberían aparecer encerradas entre llaves y colocadas en el punto adecuado de la producción, siguiendo las siguientes reglas:

1. Una acción semántica no podrá hacer referencia a un atributo sintetizado de un símbolo que se encuentra a la derecha de dicha acción.

2. Para calcular un atributo heredado de un símbolo que está en la parte derecha de una producción, la acción debe estar situada inmediatamente antes de ese símbolo.

3. Los atributos sintetizados de un símbolo sólo se pueden calcular en una producción en la que dicho símbolo se encuentre en la parte izquierda y una vez que se hayan evaluado todos los atributos de los símbolos que aparecen en la parte derecha de la producción, por tanto, la acción correspondiente se ubicará al final de la producción.

Veamos un ejemplo de cómo se aplicarían estas reglas en el ejemplo anterior:

$$\mathbf{A} ::= \{x.x_1 \downarrow = f(A.a_2 \downarrow)\}\, \mathbf{x}\, \{y.y_2 \downarrow = g(x.x_1 \downarrow)\}\, \mathbf{y}\, \mathbf{z}\, \{A.a_1 \uparrow = h(y.y_2 \downarrow, z.z_1 \uparrow)\}$$

No podemos asociar a esta producción acciones semánticas que evalúen los atributos a_2 (por ser heredado se evaluará en una producción en la que el símbolo A esté ubicado en la parte derecha), y_1 y z_1 (por ser atributos sintetizados se deben evaluar en producciones en las que los símbolos y y z aparezcan en la parte izquierda).

Las tres reglas vistas anteriormente tienen sentido si analizamos el algoritmo que se utiliza para evaluar los atributos en una gramática L-atribuida al hacer un recorrido en preorden de cada uno de los nodos que forman el árbol sintáctico.

Algoritmo 6.1 Visitar (N)

Input: N es un nodo de un árbol sintáctico

```
Begin
  for all H hijo del nodo N (de izquierda a derecha) do
    Evaluar los atributos heredados del nodo H
    Visitar (H)
  end for
  Evaluar los atributos sintetizados del nodo N
End
```

Ejemplo 6.4 La siguiente gramática L-atribuida permite generar una declaración de variables siguiendo las pautas del lenguaje C. Los símbolos no terminales **D**, **T** y **L** representan los conceptos de *declaración, tipo* y *lista de identificadores*, respectivamente. En este ejemplo, los atributos que tienen los diferentes símbolos gramaticales almacenan información que representa el tipo de las variables que se van a definir. Además, el símbolo terminal **id** tiene un atributo sintetizado que permite conocer el nombre de la variable y que habrá sido calculado en el proceso de análisis léxico.

La semántica de estas instrucciones de declaración debe asociar a cada variable su tipo correspondiente y además debe almacenar, utilizando el procedimiento *añadir_tabla*, la información asociada a cada variable en la tabla de símbolos del compilador.

$$\mathbf{D} ::= \mathbf{T}\,\{L.tipoh = T.tipo\}\,\mathbf{L}$$

$$\mathbf{T} ::= \mathbf{INT} \qquad \{T.tipo = 0\}$$

$$\mathbf{T} ::= \mathbf{REAL} \qquad \{T.tipo = 1\}$$

$$\mathbf{L} ::= \{ID.tipoh = L.tipoh\} \qquad \mathbf{ID}$$
$$\{\text{añadir\_tabla}(ID.nombre,\ ID.tipoh)\}$$

$$\mathbf{L} ::= \{L_1.tipoh = L_0.tipoh\}\,\mathbf{L},\ \{ID.tipoh = L_0.tipoh\} \qquad \mathbf{ID}$$
$$\{\text{añadir\_tabla}(ID.nombre,\ ID.tipoh)\}$$

Los atributos de L y de ID son heredados, por eso los llamamos *tipoh*, el atributo de T es sintetizado y se calcula en aquellas producciones que tienen a T en la parte izquierda (la segunda y la tercera).

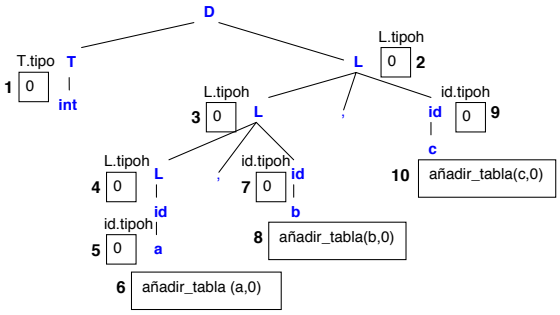


Figura 6.2: Gramática L-atribuida

La figura 6.2 muestra el árbol de derivación de una declaración de variables como **int a, b, c** , la numeración indica el orden en el que se realizarían las diferentes acciones semánticas de acuerdo al algoritmo 6.1

6.4. Problemas

6.1 Dada la siguiente gramática atribuida:

$\Sigma_T = \{id, cte, =, +\}$ $\Sigma_N = \{asig, expr\}$ $S = asig$

1. $asig ::= id = expr$ $\{id.valor = expr.valor\}$
2. $expr ::= id$ $\{expr.valor = id.valor\}$
3. $\quad |cte$ $\{expr.valor = cte.valor\}$
4. $\quad |expr + expr$ $\{expr_0.valor = expr_1.valor + expr_2.valor\}$

¿Es esta gramática S-atribuida o L-atribuida?

Tema 7

Máquinas de Turing

Contenido

7.1. Introducción. Antecedentes históricos 105
7.2. Definición y ejemplos de M.T.'s 107
7.3. Restricciones a la M.T. 110
7.4. Modificaciones de la M.T. 113
7.5. Técnicas para la construcción de M.T. 116
7.6. La M.T. Universal 118
7.7. La M.T. como generadora de lenguajes 119
7.8. La tesis de Church-Turing 120
7.9. Problemas 120

La máquina de Turing es un dispositivo teórico muy simple pero con una gran capacidad computacional, es decir, permite resolver problemas de una gran complejidad. Como veremos en los próximos temas, la máquina de Turing es una herramienta formal muy útil para estudiar la teoría de la computabilidad.

7.1. Introducción. Antecedentes históricos

En la década de los 30, el inglés **Allan Turing** diseñó el modelo matemático de una máquina teórica con un gran poder computacional, llamada Máquina de Turing **(M.T.)**. En los siguientes párrafos analizaremos los motivos científicos e históricos que llevaron a Turing a diseñar esta máquina.

A finales del siglo XIX, la recién nacida *Teoría de Conjuntos* había causado un gran impacto entre los matemáticos. Sin embargo, algunos pensadores como Bertrand Russell opinaban que dicha teoría no estaba bien formalizada ya que permitía enunciados tan paradójicos como éste:

> Si R es el conjunto de todos los conjuntos que no son miembros de sí mismo, ¿$R \in R$?

Para evitar este tipo de problemas, Russell y Whitehead desarrollaron un sistema matemático de axiomas y reglas de inferencia altamente formalizado, cuyo propósito era poder traducir a este esquema cualquier razonamiento matemático correcto. Las reglas estaban cuidadosamente seleccionadas para evitar planteamientos que pudieran llevar a conclusiones paradójicas.

Simultáneamente, el prestigioso matemático alemán David Hilbert comenzó la tarea de establecer un esquema mucho más completo y manejable. Hilbert pretendía demostrar que el sistema estaba libre de contradicciones. Nunca vaciló en proclamar su convicción de que algún día con este sistema se podría resolver cualquier problema matemático o demostrar que carece de solución. Aseguraba que existía un procedimiento por medio del cual era posible afirmar *a priori* si un problema podía o no ser resuelto. Llamó a este problema *Entscheidungsproblem* ("el problema de la decisión"). Sin embargo, un joven austriaco, llamado Kurt Gödel, publico en 1931 un artículo titulado "Sobre proposiciones formalmente indecidibles de los fundamentos de las matemáticas y sistemas relacionales". En él, Gödel probó el *Teorema de la incompletitud* en el que se afirmaban dos importantes cuestiones:

1. Si la teoría axiomática de conjuntos es consistente existen teoremas que no pueden ser probados ni refutados.

2. No existe ningún procedimiento constructivo que pruebe que la teoría axiomática de conjuntos es consistente.

En la demostración de este teorema Gödel también propuso un interesante método para enumerar objetos que originalmente no parecen ser enumerables. Utilizó para ello el teorema fundamental de la Aritmética o teorema de factorización única que afirma que todo entero positivo se puede representar de forma única como producto de factores primos. Por ejemplo, $6936 = 2^3 \cdot 3^1 \cdot 17^2$ y no hay ninguna otra factorización del número 6936 en números primos. Esta idea es muy interesante y puede ser utilizada para catalogar el conjunto de las M.T.'s como veremos en el siguiente tema. Veamos su aplicación para enumerar, por ejemplo, las expresiones aritméticas.

En primer lugar es necesario enumerar todos los elementos que pueden formar parte de una expresión aritmética (operaciones y dígitos):

+	-	*	÷	0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10	11	12	13	14

De esta manera, una expresión como **3*5-2** podría representarse mediante la tupla **(8-3-10-2-7)** que puede asociarse con el número $\mathbf{2^8 \cdot 3^3 \cdot 5^{10} \cdot 7^2 \cdot 11^7}$. Debido a la unicidad de la factorización en números primos podemos reconstruir la expresión aritmética a partir de su código (número de orden) asociado.

Para los científicos mencionados anteriormente era fundamental el concepto de *algoritmo* o *proceso efectivo*, considerándolo como un método para resolver un problema genérico en un número finito de pasos mediante operaciones conocidas y realizables. Desde este punto de vista, Turing diseñó su máquina como un dispositivo capaz de realizar un algoritmo. Apoyó las teorías de Gödel al demostrar que algunos problemas no pueden resolverse con una M.T.

La gran ventaja de la M.T. es que, a pesar de su simplicidad, tiene un gran poder computacional y no se ve limitada por las características tecnológicas de una computadora como la velocidad de procesamiento o la capacidad de la memoria. Por esta razón, se la considera como un símbolo invariante de la informática.

En esta misma época, Emil Post, así como Church y Kleene, realizaron estudios similares a los de Turing.

7.2. Definición y ejemplos de M.T.'s

La M.T. es un modelo matemático para representar a una máquina teórica. A pesar de su simplicidad la M.T. tiene el mismo poder computacional que una computadora de propósito general.

La M.T. es interesante, sobre todo, por el conjunto de lenguajes que permite reconocer y también generar (lenguajes recursivamente enumerables) y por el conjunto de funciones que puede computar (funciones calculables).

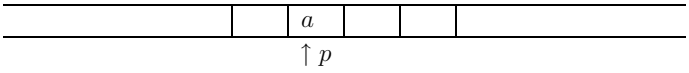
El modelo básico de M.T. está formado por un autómata finito, con un dispositivo de lectura/escritura que controla una cinta de longitud infinita. La cinta está dividida en celdas en las que se almacena un único símbolo o un espacio en blanco. Aunque la longitud de la cinta es infinita, en cada momento sólo un número finito de celdas contienen símbolos diferentes al espacio en blanco, que pertenecen a un alfabeto también finito. El dispositivo de lectura/escritura puede explorar(leer) el contenido de una celda y grabar(escribir) un nuevo símbolo sobre ella, seguidamente se desplaza en una posición hacia la izquierda o hacia la derecha. Por tanto, la M.T. dependiendo del símbolo leído y del estado actual del autómata, realiza las siguientes operaciones:

1. Cambia de estado.

2. Escribe un símbolo en la celda analizada.

3. Desplaza la cabeza de lectura/escritura a la izquierda o a la derecha en una posición.

Con el siguiente esquema se representa una M.T. que se encuentra en un estado llamado p y cuya cabeza de lectura/escritura señala a una celda que contiene al símbolo a



Definición 7.1 (Máquina de Turing)
Formalmente la M.T. se define como una tupla:

$$MT = \{Q, \Sigma, \Gamma, f, q_0, b, F\}$$

- Q *es el conjunto finito de estados*
- Σ *es el alfabeto de la cadena de entrada*
- Γ *es el alfabeto de la cinta*
- $q_0 \in Q$ *es el estado inicial*
- b *es el espacio en blanco* $b \in \Gamma$, *pero* $b \notin \Sigma$
- $F \subset Q$ *es el conjunto de estados finales*
- $f : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{I, D\}$

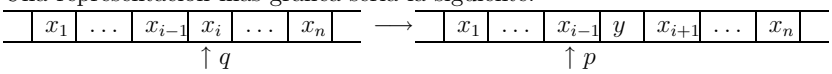
Inicialmente, en la cinta hay una colección finita de símbolos de Σ precedida y seguida por blancos (b), el estado de la M.T. es q_0 y la cabeza de lectura/escritura suele apuntar al primer símbolo distinto de b que hay en la cinta.

Definición 7.2 (Descripción Instantánea de una M.T.)
Llamaremos Descripción Instantánea*(D.I.) de la M.T. a* $\boxed{\alpha_1 \mathbf{q} \alpha_2}$ *donde q es el estado actual de la máquina y $\alpha_1\alpha_2 \in \Gamma^*$ es el contenido de la cinta. La cabeza de lectura/escritura analiza en ese momento el primer símbolo de α_2. Dentro de la cadena $\alpha_1\alpha_2$ podemos encontrar el carácter b, pero el primer carácter de α_1 es el carácter diferente de b más a la izquierda de la cinta y el último carácter de α_2 es el carácter diferente de b más a la derecha de la cinta.*

Teniendo en cuenta el concepto de D.I. los movimientos de la M.T. pueden definirse de la siguiente forma:

- Sea una M.T. cuya D.I. es $x_1 \dots x_{i-1}\mathbf{q}x_i \dots x_n$
- Si $f(q, x_i) = (p, y, I)$ entonces la nueva D.I. es $x_1 \dots x_{i-2}\mathbf{p}x_{i-1}yx_{i+1} \dots x_n$
- Podemos representar la transición de la forma:
  $x_1 \dots x_{i-1}\mathbf{q}x_i \dots x_n \longrightarrow x_1 \dots x_{i-2}\mathbf{p}x_{i-1}yx_{i+1} \dots x_n$

Una representación más gráfica sería la siguiente:



Se puede utilizar el símbolo $\xrightarrow{*}$para indicar que hemos pasado de una situación a otra en uno o más pasos.

Definición 7.3 (Lenguaje aceptado por una M.T.)
Está formado por la cadenas definidas sobre el alfabeto Σ que hacen que la M.T. llegue a un estado final, partiendo de una situación inicial en que la cadena está en la cinta de entrada, q_0 es el estado inicial y la cabeza de lectura/escritura apunta a la celda ocupada por el primer carácter de dicha cadena.

$$L(MT) = \{w \in \Sigma^* / \, q_0 w \xrightarrow{*} \alpha_1 p \alpha_2 \quad \text{donde } p \in F \; \alpha_1, \alpha_2 \in \Gamma^*\}$$

Ejemplo 7.1 (Complemento binario) Esta M.T. obtiene el complemento binario de un número binario almacenado en la cinta.

$Q = \{q_0, q_1\}$ $F = \{q_1\}$ $\Sigma = \{0, 1\}$ $\Gamma = \{0, 1, b\}$

La función de transición se define en la siguiente tabla:

	0	**1**	**b**
q_0	$q_0 1 D$	$q_0 0 D$	$q_1 b I$

Ejemplo 7.2 (Paridad) Esta M.T. calcula la paridad del número de 1's que hay en la cadena binaria de entrada. Al final del proceso, se añade a la derecha de la cadena un 0 para indicar que hay un número par de 1's y un 1 para indicar que el número de 1's es impar.

$Q = \{q_0, q_1, q_2\}$ $F = \{q_2\}$ $\Sigma = \{0, 1\}$ $\Gamma = \{0, 1, b\}$

La función de transición se define en la siguiente tabla:

	0	**1**	**b**
q_0	$q_0 0 D$	$q_1 1 D$	$q_2 0 I$
q_1	$q_1 0 D$	$q_0 1 D$	$q_2 1 I$

Ejemplo 7.3 (Duplicar) Esta M.T. recibe una cadena formada por 1's en la cinta de entrada y duplica su tamaño. El proceso comienza en el extremo derecho de la cadena. Para llevar a cabo este trabajo, cada vez que encuentra un 1 lo *marca* sustituyéndolo por un 0 y se desplaza hasta el final de la cadena donde añade otro 0 (que representa a un 1 *marcado*). Al final del proceso, en el estado q_2 se cambian todos los 0's por 1's para reconstruir la información.

$Q = \{q_0, q_1, q_2, q_3\}$ $F = \{q_3\}$ $\Sigma = \{1\}$ $\Gamma = \{0, 1, b\}$

La función de transición se define en la siguiente tabla:

	1	**0**	**b**
q_0	$q_1 0 D$	$q_0 0 I$	$q_2 b D$
q_1	$q_1 1 D$	$q_1 0 D$	$q_0 0 I$
q_2		$q_2 1 D$	$q_3 b I$

Ejemplo 7.4 (Imagen especular) Esta M.T. construye la imagen especular de un número binario. Es decir, si recibe la cadena de entrada **100** al acabar el proceso la información que hay en la cinta es **100001**. Esta máquina es similar a la anterior con la diferencia de que, en este caso, hay que duplicar dos tipos de símbolos (0 y 1) en lugar de uno. También en este caso los dígitos se procesarán de derecha a izquierda. Los 0's y 1's son reemplazados por A's y B's respectivamente cada vez que se procesan (duplican). Al final del proceso, en el estado q_3 se reconstruye la cadena cambiando las A's y B's por 0's y 1's.

$Q = \{q_0, q_1, q_2, q_3, q_4\}$ $F = \{q_4\}$ $\Sigma = \{0, 1\}$ $\Gamma = \{0, 1, A, B, b\}$

La función de transición se define en la siguiente tabla:

	0	**1**	**A**	**B**	**b**
q_0	$q_1 AD$	$q_2 BD$	$q_0 AI$	$q_0 BI$	$q_3 bD$
q_1			$q_1 AD$	$q_1 BD$	$q_0 AI$
q_2			$q_2 AD$	$q_2 BD$	$q_0 BI$
q_3			$q_3 0 D$	$q_3 1 D$	$q_4 bI$

Ejemplo 7.5 (Paréntesis anidados) Esta M.T. reconoce cadenas de paréntesis anidados. Las parejas de paréntesis que se van procesado se *marcan* convirtiéndolos en *. En este caso se dispone de dos estado finales f_1 y f_2 que indican respectivamente si la cadena inicial es o no correcta, además, a pesar de ser algo redundante se escribe en la cinta la letra 'S' o 'N' (**si** o **no**). Al final del proceso no se reconstruye la información inicial que, por tanto, se pierde. Esta máquina puede reconocer cadenas como éstas: ((())) o (()()) pero no reconocería ()))(.

$Q = \{q_0, q_1, q_2, q_3, f_1, f_2\}$ $F = \{f_1, f_2\}$ $\Sigma = \{(,)\}$ $\Gamma = \{(,), *, S, N, b\}$

La función de transición se define en la siguiente tabla:

	(	**)**	*****	**b**	
q_0	$q_0 (D$	$q_1 * I$	$q_0 * D$	$q_2 bI$	marca un) y pasa a q_1
q_1	$q_0 * D$		$q_1 * I$	$f_2 ND$	marca un (y pasa a q_0
q_2	$q_3 * I$		$q_2 * I$	$f_1 SD$	al final, comprueba que no quedan (
q_3	$q_3 * I$		$q_3 * I$	$f_2 ND$	se llega a q_3 si hay más (que)

7.3. Restricciones a la M.T.

La M.T. que se ha definido en la sección anterior es la más genérica posible. Sin embargo, veremos a continuación que se le pueden imponer restricciones a la definición sin que esto afecte a la potencia computacional de la máquina.

Las restricciones se aplicarán sucesivamente a:

1. el alfabeto,
2. la estructura de la cinta, y
3. la capacidad de la máquina para realizar diferentes operaciones (escribir, desplazarse o cambiar de estado) en una sola transición.

7.3.1. M.T. con alfabeto binario

Cualquier M.T. es equivalente (esto significa que realiza la misma tarea) a una M.T. con un alfabeto binario $\Gamma = \{0, 1, b\}$. Para conseguirlo, será necesario codificar en binario los caracteres del alfabeto original. Cada transición original se desglosará en varias transiciones de la máquina con alfabeto binario con el consiguiente incremento de estados *intermedios*, como se indica en el siguiente ejemplo:

Sea Z una M.T. definida sobre el alfabeto $\Sigma = \{x, y, z, w\}$. Supongamos que Z' es la M.T. definida sobre un alfabeto binario, equivalente a Z, que pretendemos construir. Codificamos los símbolos de Σ de la siguiente forma:

$x = 00 \qquad y = 10 \qquad z = 01 \qquad w = 11$

Supongamos que $f(p, x) = (q, y, D)$ es una de las transiciones de Z. En la máquina Z' esta transición se desglosaría en las siguientes:

$f'(p, 0) = (p_0, 0, D)$ reconoce el primer 0 de x
$f'(p_0, 0) = (p_x, 0, I)$ reconoce el segundo 0 de x
$f'(p_x, 0) = (p_{x0}, 1, D)$ cambia 00 por 10 (x por y)
$f'(p_{x0}, 0) = (q, 0, D)$ se desplaza a la derecha y pasa al estado q

Para conseguir en Z' transiciones equivalentes a la de Z ha sido necesario utilizar tres nuevos estados p_0, p_x, p_{x0}.

En general, si Z tiene un alfabeto de tamaño m será necesario buscar $n \in \mathbb{N}$ tal que $2^{n-1} < m \leq 2^n$ de manera que todos los símbolos del alfabeto de Z podrán ser codificados mediante una cadena binaria de longitud n. El proceso que deberá llevar a cabo Z' será el siguiente:

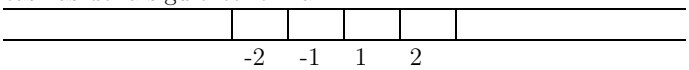
1. Analizar los n caracteres que representan a un símbolo del alfabeto original. En el peor de los casos serán necesarias n transiciones y $2^n + 2^{n-1} + \ldots + 2^0$ nuevos estados.

2. Una vez reconocido el símbolo, Z' deberá retroceder como máximo n posiciones con el fin de poder modificar la cadena de longitud n que acaba de analizar.

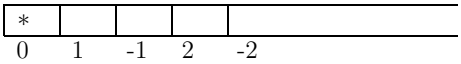
3. Finalmente la cabeza de lectura/escritura deberá desplazarse hasta llegar a la posición adecuada.

7.3.2. M.T. con la cinta limitada en un sentido

Dada una M.T. con una cinta infinita en ambos sentidos, siempre existe una máquina equivalente cuya cinta está limitada por un extremo pero es infinita por el otro. Sea Z una M.T. con una cinta infinita en ambos sentidos. Se pueden numerar las casillas de la siguiente forma:



Es posible construir una M.T. equivalente Z' cuya cinta tendrá la siguiente estructura:



En Z' existe una nueva casilla, numerada con el 0, que contiene un símbolo nuevo $(*)$. Además de añadir esta nueva casilla, se ha redistribuido la información de Z, de manera que cada casilla de Z' contiene la misma información que su correspondiente casilla de Z (la que tiene la misma numeración). Veamos con un ejemplo el funcionamiento de Z'.

Supongamos que $f(p,x) = (q,y,D)$ es una de las transiciones de Z y supongamos que x está almacenado en la casilla $n (n > 0)$, después de realizar el cambio de símbolo hay que desplazarse a la casilla $n+1$ (que ahora está situada dos posiciones a la derecha de n). En la máquina Z' está transición se desglosaría en las siguientes:

$f'(p,x) = (p_D, y, D)$
$f'(p_D, ?) = (q, ?, D)$? es un comodín que representa a cualquier símbolo

El comportamiento de Z' sería diferente si la casilla ocupada por x fuera la etiquetada con $-n$ ya que un desplazamiento a la derecha en Z supone un doble desplazamiento hacia la izquierda en Z'. La transición anterior se desglosaría en:

$f'(p,x) = (p'_D, y, I)$
$f'(p'_D, ?) = (q, ?, I)$

Si la posición ocupada por x fuera la -1, habría que añadir una transición al pasar por la posición 0. $f'(q,*) = (q,*,D)$

En general, la nueva máquina tiene un número de estados 6 veces mayor que Z ya que cada estado q de Z se multiplica por 3 (q, q_I, q_D) en la parte *positiva* de la cinta y otro tanto ocurre con la parte *negativa* de la cinta (q', q'_I, q'_D).

7.3.3. M.T. con restricciones en cuanto a las operaciones que realiza simultáneamente

En el modelo original de M.T., se realizan tres operaciones en cada transición:
1. Escritura de un símbolo
2. Cambio de estado
3. Movimiento de la cabeza de lectura/escritura

Veamos, con diferentes ejemplos, como, aumentando el número de estados, se puede restringir el número de operaciones que se realizarán simultáneamente.

Imposibilidad para escribir y cambiar de estado simultáneamente

La transición $f(p,x) = (q,y,D)$ se convierte en:

$f'(p,x) = (p_{xD}, x, P)$
$f'(p_{xD}, x) = (p_{xD}, y, P)$
$f'(p_{xD}, y) = (q, y, D)$

Imposibilidad para escribir y desplazarse simultáneamente

La transición $f(p,x) = (q,y,D)$ se convierte en:

$f'(p,x) = (p_D, y, P)$
$f'(p_D, y) = (q, y, D)$

De forma análoga, dada una M.T. es posible construir otra equivalente a ella que ejecute una sola operación en cada transición.

7.4. Modificaciones de la M.T.

Una de las razones de la gran aceptación de la M.T. como modelo general de computación es que el modelo estándar definido al comienzo del tema es equivalente a otras versiones de M.T. que, sin aumentar el poder computacional del dispositivo, permiten resolver con más facilidad determinados problemas.

7.4.1. Almacenamiento de información en el control finito

Es posible utilizar el estado de control (perteneciente a un conjunto finito) para almacenar una cantidad finita de información. Para ello cada estado se representa como un par ordenado donde el primer elemento representa realmente al estado y el segundo a la información que se pretende almacenar.

Veamos su utilidad con un ejemplo en el que se define una M.T. que reconoce el lenguaje $L = 01^* + 10^*$. Como el primer símbolo de la cadena no puede volver a aparecer, se almacena con el estado de control. Las transiciones no definidas son situaciones de error, es decir, la cadena no ha sido reconocida.

$Q = \{q_0, q_1\} * \{0, 1, b\}$ estado inicial $= [q_0, b]$ $F = \{[q_1, b]\}$
$\Sigma = \{0, 1\}$ $\Gamma = \{0, 1, b\}$

	0	1	b	
$[q_0, b]$	$[q_1, 0]0D$	$[q_1, 1]1D$		
$[q_1, 0]$		$[q_1, 0]1D$	$[q_1, b]0P$	este estado indica que la cadena comienza p
$[q_1, 1]$	$[q_1, 1]0D$		$[q_1, b]0P$	este estado indica que la cadena comienza p

Es evidente que no se aumenta la potencia computacional del modelo ya que simplemente se ha utilizado una notación diferente para designar a los estados.

7.4.2. Pistas múltiples

En este caso se considera que la cinta está dividida en un número k de pistas, de manera que los símbolos de la cinta se representan como k-tuplas. Tampoco en este caso se aumenta la potencia del modelo ya que sólo hay un cambio en la representación de los símbolos de la M.T. En este sentido, es un caso análogo al anterior. Esta variación de la M.T., concretamente con tres pistas, puede resultar muy útil para resolver operaciones aritméticas con dos datos. Cada dato se almacena en una pista y el resultado se almacena en la tercera (que inicialmente está vacía). Por ejemplo, una M.T. que sume números binarios comenzaría con la siguiente información en la cinta:

	b	1	1	0	b
	b	1	1	1	b
	b	b	b	b	b

y finalmente la cinta quedaría así:

	b	1	1	0	b
	b	1	1	1	b
b	1	1	0	1	b

Ejemplos de transiciones para esta máquina:

$$f(q_0, \begin{bmatrix} 0 \\ 1 \\ b \end{bmatrix}) = (q_0, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, I) \qquad f(q_0, \begin{bmatrix} 0 \\ 0 \\ b \end{bmatrix}) = (q_0, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, I)$$

7.4.3. Símbolos de chequeo

Como se ha visto en ejemplos anteriores, para resolver muchos problemas es importante *marcar* aquellos símbolos que se van procesando. Hasta ahora estos símbolos se *marcaban* sustituyéndolos por otros diferentes, sin embargo esta misma idea puede llevarse a cabo de una forma más intuitiva y sencilla utilizando en la cinta una segunda pista en la que sólo se almacenan espacios en blanco y otro símbolo (por ejemplo, $*$) que sólo aparece debajo del símbolo de la primera pista (la original) que ha sido procesado. Esto no es más que un caso particular de una M.T. con pistas múltiples.

Veamos en el siguiente ejemplo como se utilizarían estos *símbolos de chequeo* para resolver el problema planteado en el ejemplo 3 en el que la M.T. duplicaba el número de 1's que había en la cinta inicialmente.

Ejemplo 7.6 (Duplicar - segunda versión) $Q = \{q_0, q_1, q_2\}$ $F = \{q_2\}$ $\Sigma = \{1\}$ $\Gamma = \{0, 1, b\}$

El proceso comienza en el extremo derecho de la cinta y la función de transición se define en la siguiente tabla:

	$\begin{pmatrix} \mathbf{1} \\ \mathbf{b} \end{pmatrix}$	$\begin{pmatrix} \mathbf{1} \\ * \end{pmatrix}$	$\begin{pmatrix} \mathbf{b} \\ \mathbf{b} \end{pmatrix}$
$\mathbf{q_0}$	$q_1 \begin{pmatrix} 1 \\ * \end{pmatrix} D$	$q_0 \begin{pmatrix} 1 \\ * \end{pmatrix} I$	$q_2 \begin{pmatrix} b \\ b \end{pmatrix} D$
$\mathbf{q_1}$		$q_1 \begin{pmatrix} 1 \\ * \end{pmatrix} D$	$q_0 \begin{pmatrix} 1 \\ * \end{pmatrix} I$

7.4.4. Máquinas multicinta

En este caso la M.T. dispone de k cintas y k cabezas de *lectura/escritura*. Cada cabeza trabaja (lee y escribe) sobre su cinta y se mueve con total independencia de las demás. Para comprobar que este tipo de máquinas no aumenta el poder computacional de la M.T. estándar se puede construir una máquina equivalente a ella utilizando varias pistas y símbolos de chequeo. Supongamos que la máquina Z dispone de tres cintas y una situación concreta puede ser representada como se indica en el siguiente esquema:

		A_i			
		$\uparrow p$			
				B_j	
				$\uparrow q$	
			C_k		
			$\uparrow r$		

Es posible construir una nueva máquina *multipista* que sea equivalente a ella. Esta nueva máquina tendrá un número de pistas igual al doble de las cintas de la máquina original, en este caso seis. Las pistas pares almacenan un símbolo de chequeo que sirve para indicar qué celda de la pista inmediatamente superior se está procesando en ese momento. Las pistas impares tienen la misma función que las correspondientes cintas de la máquina original. Evidentemente, esta nueva M.T. tendrá más estados porque tiene que ir procesando secuencialmente cada pareja de pistas buscado el símbolo de chequeo correspondiente.

		A_i			
		$*$			
				B_j	
				$*$	
			C_k		
			$*$		

7.4.5. M.T. no determinista

Dado un estado y un símbolo analizado, la M.T. no determinista tiene un número finito de posibilidades entre las que elegir para realizar el siguiente movimiento. Cada elección consiste en un nuevo estado, un símbolo a grabar y una dirección de movimiento de la cabeza. Por ejemplo:

$$f(q,a) = \{(p_1,a,D),(p_2,b,I),(p_3,a,I),\ldots\}$$

Como ocurría con los Autómatas Finitos, el *No Determinismo* no añade más potencia a la M.T., ni siquiera la combinación del *No Determinismo* con cualquiera de las modificaciones previamente planteadas aumenta el poder computacional de la M.T.

7.5. Técnicas para la construcción de M.T.

De la misma forma que al programar, al diseñar M.T.'s también es posible utilizar técnicas de diseño modular que simplifican la tarea de su construcción. De hecho, una M.T. puede simular el comportamiento de cualquier tipo de subrutina, incluyendo procedimientos recursivos con cualquier mecanismo conocido de paso de parámetros. Para que una M.T. actúe como subrutina de otra, la idea general es que su estado final se convierta en un estado de retorno a la máquina que la ha llamado. La llamada se efectúa, por tanto, a través del estado inicial de la subrutina y el retorno a través de su estado final.

Veamos, en primer lugar, como *encadenar* dos M.T. M_1 y M_2 de manera que primero trabaje M_1 e inmediatamente después lo haga M_2 utilizando como entrada la salida de M_1. Para llevar a cabo este *encadenamiento* basta con realizar una transición desde el estado final de M_1 hasta el inicial de M_2. Es decir, si p_1 y q_1 son los estados inicial y final, respectivamente de M_1 y p_2 y q_2 son los estados inicial y final de M_2, hay que incluir la transición $f(q_1,x) = (p_2,x,P)\ \forall x \in \Gamma$. Por ejemplo, si deseamos construir una M.T.M que calcule la función $f() = 2x^2$, podríamos construir M_1 que calcule $g(x) = x^2$ y M_2 que calcule $h(x) = 2x$, así M sería el resultado de *encadenar* M_1 con M_2.

En ocasiones sólo es deseable que trabaje M_2 si el proceso de M_1 ha terminado bajo ciertas condiciones (simulación de una sentencia *if*). Por ejemplo $M_1 \xrightarrow{a} M_2$ indica que M_2 sólo trabajará en el caso de que al acabar M_1, la cabeza de lect./esc. esté situada sobre el símbolo a. De la misma forma, $M_1 \xrightarrow{a,b} M_2$ indica que M_2 sólo trabajará en el caso de que al acabar M_1 la cabeza de lec./esc. esté situada sobre una celda que contiene al símbolo a o al símbolo b. En este segundo caso, habría que añadir las siguientes transiciones:

$f(q_1,a) = (p_2,a,P)$
$f(q_1,b) = (p_2,b,P)$

$f(q_1, x) = (f, x, P)$ $\quad \forall x \in \Gamma \backslash \{a, b\}$ donde f es un nuevo estado final.

El esquema $M_1 \begin{matrix} a\nearrow & M_2 \\ b\searrow & M_3 \end{matrix}$ indica que si al acabar el proceso, M_1 está señalando a una celda que contiene el símbolo a, la máquina M_2 debe continuar el proceso, pero si la casilla contiene el símbolo b, será M_3 la máquina que continuará trabajando. En otro caso el proceso finaliza. Para materializar esta situación habría que añadir las transiciones:

$f(q_1, a) = (p_2, a, P)$
$f(q_1, b) = (p_3, b, P)$
$f(q_1, x) = (f, x, P)$ $\quad \forall x \in \Gamma \backslash \{a, b\}$ donde f es un nuevo estado final.

Ejemplo 7.7 (Multiplicar) Este ejemplo servirá para ilustrar la utilización de una M.T. como subrutina de otra. Se construirá una M.T. que multiplique números enteros, que se representan como cadenas de 1's. Por ejemplo, un 4 se representa como 1111. Se utiliza el 0 como separador entre los datos iniciales. La M.T. se diseñará de manera que si la entrada es 1^m01^n0 la salida deberá ser $1^{n\times m}$. Básicamente, la M.T. copiará al final de la cadena el segundo bloque de 1's (1^n) tantas veces como 1's haya en el primer bloque (1^m), los 1's de este primer bloque se van borrando (es una forma sencilla de *marcarlos*).

En primer lugar, se diseña una M.T., llamada `COPIAR` que copia al final de la cadena de entrada **n** 1's. Es decir, si la situación inicial de la M.T. se describe así $1^m0\mathbf{q_1}1^n01^i$, la situación final será $1^m0\mathbf{q_5}1^n01^{i+n}$. Los 1's (del segundo bloque) que se van copiando se *marcan* cambiándolos temporalmente por un 2, y además se procesan de izquierda a derecha.

La función de transición de `COPIAR` se define en la siguiente tabla, además q_1 y q_5 son respectivamente los estados inicial y final:

	0	**1**	**2**	**b**	
q_1	q_40I	q_22D			cambia un 1 por un 2 y pasa a q_2
q_2	q_20D	q_21D		q_31I	se desplaza al extr. derecho y añade un 1
q_3	q_30I	q_31I	q_12D		se desplaza hacia la izq. buscando un 2
q_4	q_50D		q_41I		cambia los 2's por 1's

A continuación, se diseña la M.T. `MULTIPLICAR` que utiliza la máquina `COPIAR` a modo de subrutina. La M.T. `MULTIPLICAR` va borrando los 1's del primer dato al mismo tiempo que, mediante la M.T. `COPIAR`, copia **n** 1's al final de la cadena. Cuando se han borrado todos los 1's del primer dato se borran también los del segundo así como los 0's que actúan de separadores de manera que finalmente en la cinta sólo queda el resultado de la operación ($1^{n\times m}$). En este caso los estados inicial y final son q_0 y q_{12}.

	0	1	b	
q_0		q_6bD		elimina un 1 del primer dato
q_5		q_71I		
q_6	q_10D	q_61D		llama a COPIAR
q_7	q_80I			
q_8		q_91I	$q_{10}bD$	
q_9		q_91I	q_0bD	retrocede al extr. izq. para volver a empezar
q_{10}	$q_{11}bD$			borra un 0
q_{11}	$q_{12}bD$	$q_{11}bD$		borra el segundo dato y un 0

7.6. La M.T. Universal

El concepto de M.T. es tan general y potente que es posible construir una M.T. que sea capaz de simular el comportamiento de cualquier otra M.T., a esta máquina se la llama Máquina de Turing Universal, y su forma de trabajar puede compararse con la de un ordenador que se comporta de una u otra forma dependiendo del programa que ejecuta en cada momento.

En primer lugar, veremos cómo se podría representar toda la información necesaria para simular el comportamiento de una M.T. llamada **M**.

Es necesario conocer la función de transición, el estado inicial, el contenido inicial de la cinta y la posición de la cabeza de lectura/escritura. Sin perdida de generalidad supondremos que **M** trabaja con un alfabeto binario y que necesitamos **m** bits para codificar cada uno de sus estados. El movimiento de la cabeza se puede codificar con un único bit (por ejemplo, 0=Derecha, 1=Izquierda). Así cada transición $f(p,a) = (q,b,D)$ puede codificarse con $2m+3$ símbolos binarios de la forma $\underbrace{xx\ldots x}_{m}\, a\, \underbrace{yy\ldots y}_{m}\, b0$, donde $xx\ldots x$ es una codificación binaria del estado p y $yy\ldots y$ representa al estado q.

La M.T. **M** puede representarse de la siguiente forma:

$$dd\ldots *\ldots d < q\ldots qd > xx\ldots xayy\ldots yb0 > xx\ldots xayy\ldots yb0 > \ldots$$

Los símbolos < y > permiten delimitar los diferentes bloques que constituyen esta cadena. El primer bloque($dd\ldots *\ldots d$) representa el contenido de la cinta de la M.T. **M**, y el asterisco se coloca inmediatamente a la izquierda del símbolo al que apunta la cabeza de lectura/escritura. Esta información va cambiando a lo largo del proceso representando en cada momento el contenido de la cinta de **M**. El segundo bloque representa el estado actual de **M** ($q\ldots q$) y el símbolo al que apunta la cabeza (d), que es el que está a la derecha del asterisco en el bloque anterior. Esta información tiene longitud $m+1$ y también cambia a lo largo del proceso. A partir de este punto,

el resto de los bloques (de longitud $2m+3$) representan a los diferentes valores de la función de transición.

Cada transición puede considerarse como un registro formado por dos partes: la primera ($xx \ldots xa$) es una *etiqueta* que identifica a la transición, la segunda ($yy \ldots yb0$) puede ser considerada como el *dato* de la transición. Por lo tanto, esta última parte de la cadena ($> xx \ldots xayy \ldots yb0 > \ldots$) puede ser vista como una colección de registros formados cada uno de ellos por una *etiqueta* y un *dato*. Con estas consideraciones, será necesario construir una M.T. `localizadora de información` cuya misión sea buscar en esta colección de registros aquél que tenga una etiqueta determinada. Es evidente que, conociendo el significado de los datos que hay en la cinta, el segundo bloque de información ($< q \ldots qd >$) constituye la etiqueta que es necesario buscar. Una vez localizado el registro (transición) adecuado será necesario copiar el dato ($yy \ldots yb$) en el segundo bloque de la cinta, y modificar adecuadamente el primer bloque. Básicamente, la M.T. Universal está constituida por una M.T. que localiza información en la cinta y otra que copia información de una parte a otra de la cinta.

7.7. La M.T. como generadora de lenguajes

En los ejemplos de M.T. que hemos visto hasta ahora, éstas básicamente se ocupaban de reconocer lenguajes o de calcular funciones. Sin embargo, hay una tercera forma de utilizar una M.T. y es como generadora de cadenas. Este tipo de M.T.'s disponen de varias cintas de forma que en una de ellas llamada *cinta de salida*, inicialmente vacía, sólo se llevan a cabo operaciones de escritura que van llenando la cinta con las palabras del lenguaje. Es necesario utilizar un símbolo, que no pertenece al alfabeto sobre el que está definido el lenguaje, y que actúa como separador entre una palabra y otra.

Por ejemplo, si se construyera una M.T. que generara el lenguaje formado por todas las cadenas binarias y se utilizara como separador el símbolo ♯, en un momento determinado la información que habría en la *cinta de salida* sería:

$\ldots$	b	0	♯	1	♯	0	0	♯	0	1	♯	$\ldots$

Es evidente que este tipo de M.T.'s sólo para en el caso de que el lenguaje a generar sea finito, por tanto, la máquina descrita en el ejemplo anterior no pararía nunca.

Para los lenguajes de tipo 0, que se estudian con más detalle en el próximo tema, siempre es posible construir una M.T. que los genere. Como veremos, dentro de este conjunto de lenguajes existe un importante subconjunto que es el de los lenguajes recursivos. En el caso de los lenguajes recursivos es posible construir una M.T. que genere las palabras de dicho lenguaje en orden creciente de tamaños, es decir, el tamaño de la palabra generada en la posición $n+1$ es mayor o igual que el de la

generada en la posición n.

7.8. La tesis de Church-Turing

Podemos decir que una M.T. es un modelo general de computación y esto es equivalente a decir que cualquier *procedimiento algorítmico* que sea ejecutable (por una persona o una máquina) puede ser desarrollado por una M.T.

"La noción de procedimiento algorítmico que actúa sobre una secuencia de símbolos es idéntica al concepto de un proceso que puede ser ejecutado por una M.T." Esta afirmación la formuló el lógico Alonzo Church a principios de la década de los 30 y suele denominarse **tesis de Church** o **tesis de Church-Turing**. No es una afirmación matemática exacta, ya que carecemos de una definición precisa para el término *procedimiento algorítmico* y, por lo tanto, no es algo que pueda comprobarse. Sin embargo esta tesis es aceptada de forma general, debido a diferentes motivos:

1. No se ha planteado ningún tipo de computo que pueda incluirse en la categoría de *procedimiento algorítmico* y que no pueda ejecutarse en una M.T.

2. Se han propuesto mejoras al diseño de la M.T. original y en todos los casos ha sido posible demostrar que el poder computacional de las M.T.'s no se veía modificado.

3. Se han propuesto otros modelos teóricos de cómputo, que siempre son equivalentes al de las M.T.'s

La tesis de Church-Turing no puede probarse de manera precisa debido justamente a la imprecisión del término *proceso algorítmico*. Sin embargo, una vez adoptada esta tesis, ya podemos darle un significado preciso al término: "un algoritmo es un procedimiento que puede ser ejecutado por una M.T." Esta definición nos proporciona un punto de partida para analizar problemas que pueden o no resolverse con una M.T. como veremos en los siguientes temas.

7.9. Problemas

7.1 Construir una M.T. que reciba como entrada dos cadenas de 1's separadas por el símbolo ♯ y que compruebe si tienen la misma longitud.

7.2 Construir una M.T. con tres pistas que reciba dos números binarios y que indique cuál de los dos es mayor. Consideraremos que los datos están almacenados en las dos primeras pistas y alineados a la derecha, es decir, los bits menos significativos de ambos están situados en la misma columna. En la tercera pista se escribirá una

G, una **P** o una **I** indicando respectivamente que el primer número es más grande, más pequeño o igual que el segundo.

7.3 Construir una M.T. con tres pistas que sume dos números binarios. Consideraremos que los datos están almacenados en las dos primeras pistas y alineados a la derecha. El resultado se escribirá en la tercera pista que inicialmente está vacía.

7.4 Construir una M.T. para reconocer cada uno de los siguientes lenguajes

- $L_1 = \{0^n 1^n, n \in \mathbb{N}\}$
- $L_2 = \{ww^{-1}, w \in (0+1)^*\}$
- $L_3 = \{wcw' / \ w, w' \in (a+b)^* \ w \neq w'\}$

Tema 8

Gramáticas de tipo 0 y 1

Contenido

8.1. Gramáticas de tipo 0 123
8.2. Lenguajes de tipo 0 124
8.3. El problema de la parada 126
8.4. Lenguajes y gramáticas de tipo 1 126

Este capítulo se dedica al estudio de las gramáticas menos restrictivas, las de tipo 0 y las de tipo 1. Los autómatas que reconocen estas gramáticas son las Máquinas de Turing (estudiadas en el capítulo anterior) y los Autómatas Linealmente Acotados respectivamente. La diferencia principal entre ambos autómatas está en el tamaño de la cinta que utilizan. Mientras que la Máquina de Turing dispone de una cinta teóricamente infinita, los Autómatas Linealmente Acotados utilizan una cinta finita, aunque su longitud pueda ser tan grande como sea necesario en cada caso.

8.1. Gramáticas de tipo 0

Dentro de la jerarquía de Chomsky, el grupo más amplio de gramáticas, llamadas gramáticas de tipo 0 y también gramáticas sin restricciones son aquellas cuyas producciones tienen la siguiente forma:

$$u ::= v \quad u \in \Sigma^+ \quad v \in \Sigma^*$$

además $u = xAy$ donde $A \in \Sigma_N \quad x, y \in \Sigma^*$

Es decir, en la parte izquierda de las producciones hay, al menos, un símbolo no terminal.

Este conjunto de gramáticas es equivalente al de gramáticas con *estructura de frase*, cuya definición es algo más restringida. Las producciones de las gramáticas con estructura de frase son de la forma:

$$xAy ::= xvy \qquad \text{donde} \qquad A \in \Sigma_N \quad x, y, v \in \Sigma^*$$

En estas gramáticas es posible que la parte derecha de la producción sea más corta que la izquierda (cuando $v = \lambda$), en este caso se dice que es una producción *compresora*. Una gramática *compresora* es la que tiene alguna regla compresora, en este caso las derivaciones pueden ser decrecientes.

El conjunto de los lenguajes generados por las gramáticas de tipo 0 coincide con el conjunto de los generados por las gramáticas con estructura de frase y se llaman **lenguajes recursivamente enumerables**. Estos lenguajes son reconocidos por Máquinas de Turing.

Ejemplo 8.1 La gramática de tipo 0 que se describe a continuación genera el lenguaje $L = \{a^{2^i}\}$ $\Sigma_N = \{S, A, B, C, D, E\}$ $\Sigma_T = \{a\}$

$$P = \left\{ \begin{array}{l} S ::= ACaB \\ Ca ::= aaC \\ CB ::= DB|E \\ aD ::= Da \\ AD ::= AC \\ aE ::= Ea \\ AE ::= \lambda \end{array} \right\}$$

Esta gramática no tiene estructura de frase (debido, por ejemplo, a la quinta y séptima producciones), sin embargo es posible encontrar una gramática con estructura de frase que sea equivalente a ella.

8.2. Lenguajes de tipo 0

Como ya sabemos, los lenguajes generados por las gramáticas de tipo 0 se llaman lenguajes recursivamente enumerables y son los reconocidos por las Máquinas de Turing. Pero, además, estos lenguajes también pueden ser generados por Máquinas de Turing.

El nombre de *recursivamente enumerables* es debido a que sus palabras pueden ser generadas ordenadamente por una Máquina de Turing. Dicho de otra forma, existe un algoritmo que permite generar sus palabras una tras otra. Evidentemente, si el lenguaje es infinito, la Máquina de Turing que lo genera no para nunca.

Este grupo de lenguajes incluye a algunos para los que no es posible saber si una palabra **no** pertenece al lenguaje. Si L es uno de estos lenguajes, cualquier Máquina de Turing que lo reconozca no parará cuando reciba como entrada algunas palabras que no pertenecen a L. En este caso, si $w \in L$ sabemos que la M.T. parará, pero si

la máquina no para, no podemos saber si el motivo es que la palabra no pertenece al lenguaje o es que todavía no ha terminado el proceso de reconocimiento.

Teniendo en cuenta estas cuestiones es conveniente considerar un subconjunto de los lenguajes recursivamente enumerables, a los que llamaremos **lenguajes recursivos**, que son los aceptados por Máquinas de Turing que paran siempre, sea cual sea la entrada que reciban. Por supuesto, la parada de la máquina debe ir precedida por la aceptación o no de la palabra. En el caso de los lenguajes recursivos siempre es posible diseñar una Máquina de Turing que genere las palabras del lenguaje según un orden creciente de tamaño.

A continuación veremos unos resultados que, entre otras cosas, demuestran que la unión es una operación cerrada para estos dos conjuntos de lenguajes.

Teorema 8.1

1. *L es recusivo $\iff$ L y $\overline{L}$ son recursivamente enumerables*
2. *L es recursivo $\iff$ $\overline{L}$ es recursivo*
3. *L_1 y L_2 son recursivos $\Longrightarrow$ $L_1 \cup L_2$ es recursivo*
4. *L_1 y L_2 son recursivamente enumerables $\Longrightarrow$ $L_1 \cup L_2$ es recursivamente enumerable*

Demostración.

1. Si L y $\overline{L}$ son recursivamente enumerables existen dos Máquinas de Turing M_1 y M_2 que reconocen respectivamente las palabras de L y $\overline{L}$. A partir de M_1 y M_2 es posible construir una nueva máquina M que se limitaría a anotar cuál de las dos máquinas M_1 o M_2 acepta a la cadena de entrada. Evidentemente M reconoce a L (y también a $\overline{L}$) y además para, sea cual sea la entrada, por lo que L (y también $\overline{L}$) es un lenguaje recursivo. Si L no fuera recursivo, no podríamos construir M, ya que tampoco podríamos construir M_2. M_1 resulta insuficiente para construir M, ya que, en el caso de que M_1 no pare no es posible saber si la máquina ha entrado en un bucle infinito o si necesita más tiempo para procesar la cadena.

   Si L es recursivo es evidente que también es recursivamente enumerable, además podemos afirmar que $\overline{L}$ también es recursivamente enumerable ya que la M.T. que reconoce a L también permitiría reconocer a $\overline{L}$.

2. Si L es recursivo la misma M.T. que lo reconoce permite reconocer a $\overline{L}$, basta con intercambiar el significado de las salidas.

3. Si L y $\overline{L}$ son recursivos existen dos Máquinas de Turing M_1 y M_2 que reconocen respectivamente las palabras de L y $\overline{L}$. A partir de M_1 y M_2 es posible construir

una nueva máquina M que permita reconocer a las palabras de $L_1 \cup L_2$. Además esta máquina pararía siempre.

4. Si L y $\overline{L}$ son recursivamente enumerables existen dos Máquinas de Turing M_1 y M_2 que reconocen respectivamente las palabras de L y $\overline{L}$. A partir de M_1 y M_2 es posible construir una nueva máquina M que permita reconocer a las palabras de $L_1 \cup L_2$. No es posible asegurar que M parará cuando reciba como entrada una palabra que no pertenezca a $L_1 \cup L_2$, ya que en ese caso es posible que no paren ni M_1 ni M_2. □

8.3. El problema de la parada

Dada una Máquina de Turing con un dato en la cinta de entrada, no existe ningún algoritmo que permita conocer *a priori* si la máquina se detendrá o no. Este problema es, por ese motivo, *indecidible*.

Este problema puede enunciarse de otra forma. Teniendo en cuenta que el conjunto de las Máquinas de Turing puede enumerarse, es posible asociar a cada máquina un número natural. ¿Es posible construir una Máquina de Turing H que tome como datos de entrada al par (n, x) donde n representa a la máquina p_n y x a un dato para esa máquina, y devuelva 1 o 0 dependiendo de si la máquina p_n para o no teniendo a x como dato? La respuesta a esta pregunta es negativa, no es posible construir dicha máquina.

Para demostrarlo utilizaremos la técnica de *reducción al absurdo*. Supongamos que E es un lenguaje recursivamente enumerable pero no recursivo (sabemos que existen). Sea $M = p_r$ la Máquina de Turing que reconoce a E, esta máquina no para cuando recibe como entrada algunas de las palabras que no pertenecen a E. Por hipótesis H debería comportarse de la siguiente forma:

$$H(r, \alpha) = \begin{cases} 1 & \text{si M para con el dato } \alpha \text{ (esto ocurre si } \alpha \in E) \\ 0 & \text{si M no para con el dato } \alpha \text{ (esto ocurre si } \alpha \notin E) \end{cases}$$

De esta manera H permitiría reconocer las palabras del lenguaje E y además pararía siempre, pero entonces E sería un lenguaje recursivo, lo cual es falso por hipótesis. Por lo que podemos concluir que es imposible construir H.

8.4. Lenguajes y gramáticas de tipo 1

Las gramáticas de tipo 1 de la jerarquía de Chomsky, también llamadas gramáticas dependientes (o sensibles) al contexto son aquellas cuyas producciones tienen la forma:

$$xAy ::= xvy \qquad \text{donde} \qquad A \in \Sigma_N \quad x, y \in \Sigma^* \quad v \in \Sigma^+$$

Es decir, en cada derivación, un símbolo no terminal A se sustituye por una cadena v (no nula), siempre que éste se encuentre en un determinado contexto. Se puede incluir la producción $S ::= \lambda$, necesaria cuando λ pertenece al lenguaje. Es evidente que el conjunto de las gramáticas de tipo 1 está incluido en el conjunto de las gramáticas con estructura de frase y que no admiten producciones compresoras.

Los lenguajes de tipo 1 pueden ser reconocidos por Autómatas Linealmente Acotados. Estos autómatas constituyen un caso particular de las Máquinas de Turing en el que la cinta está acotada por ambos lados. Para ello, se definen dos símbolos de la cinta especiales (por ejemplo: $<$ y $>$) que delimitan el principio y el final de la *zona útil* de la cinta que puede ser manipulada. La cabeza de lectura/escritura no podrá desplazarse más a la izquierda del símbolo que indica el comienzo de dicha *zona útil* ni más a la derecha del símbolo que indica el final. Por tanto, los Autómatas Linealmente Acotados disponen de una cinta finita pero con un tamaño que puede ser tan grande como sea necesario para el proceso.

Tema 9

Computabilidad y Máquinas de Turing

Contenido

9.1. Funciones calculables 129
9.2. Funciones recursivas 131
9.3. Problemas 134

En este tema abordaremos el concepto de computabilidad o calculabilidad que se puede aplicar a aquellos problemas que pueden ser resueltos mediante un algoritmo. A partir de la definición de máquina de Turing podremos plantear una definición formal para este concepto. También estudiaremos el concepto de recursividad y su relación con el de calculabilidad.

9.1. Funciones calculables

La Teoría de la Computabilidad parte de la idea inicial de hacer precisa la noción intuitiva de función calculable, es decir, una función que pueda ser calculada automáticamente mediante un algoritmo.

Dada una función $f : \mathbb{N}^r \longrightarrow \mathbb{N}$, a veces es posible construir una Máquina de Turing Z_f que se comporte como dicha función.

Es decir, si Z_f inicia su funcionamiento con la siguiente información en la cinta $\underbrace{11\ldots1}_{n_1}0\underbrace{11\ldots1}_{n_2}0\ldots0\underbrace{11\ldots1}_{n_r}$, donde cada ristra de unos representa a un número natural, la máquina debe detenerse dejando en la cinta un número α de unos, siendo $f(n_1, n_2, \ldots, n_r) = \alpha$.

Es posible que para una tupla concreta la máquina no se detenga, en ese caso la función no estaría definida para esos valores.

Definición 9.1 (Función parcialmente calculable)
*Se dice que una función $f : D \subset \mathbb{N}^r \longrightarrow \mathbb{N}$ es **parcialmente calculable** si existe una Máquina de Turing tal que para cada tupla $t \in D$, la máquina calcula $f(t)$*

Definición 9.2 (Función calculable)
*Se dice que una función es **calculable** si es parcialmente calculable y está definida sobre todo $\mathbb{N}^r$, es decir, $D = \mathbb{N}^r$*

Ejemplo 9.1 Funciones calculables y parcialmente calculables

1. $f_1(n_1, n_2) = n_1 + n_2$ es calculable
2. $f_2(n_1, n_2) = n_1 - n_2$ si $n_1 \geq n_2$ es parcialmente calculable
3. $f_3(n_1, n_2) = \begin{cases} n_1 - n_2 & \text{si } n_1 \geq n_2 \\ 0 & \text{si } n_1 < n_2 \end{cases}$ es calculable

Las definiciones anteriores pueden ampliarse de forma natural a funciones definidas sobre otros conjuntos, por ejemplo, definidas sobre cadenas de símbolos. De esta forma podemos establecer las siguientes relaciones entre lenguajes de tipo 0 y funciones calculables o parcialmente calculables.

Dado un lenguaje $L \subseteq \Sigma^*$, definimos su función característica de la siguiente forma:

$$\begin{array}{rccl} f_L: & \Sigma^* & \longrightarrow \mathbb{N} & \\ & w & \longrightarrow f_L(w) = \begin{cases} 1 & \text{si } w \in L \\ 0 & \text{si } w \notin L \end{cases} \end{array}$$

La función característica de un lenguaje determina si una palabra pertenece o no pertenece a dicho lenguaje. También es posible definir una función que sólo indique si una palabra pertenece a un lenguaje, dicha función no estaría definida en todo el dominio y se comportaría de la siguiente forma:

$$\begin{array}{rccl} f'_L: & \Sigma^* & \longrightarrow \mathbb{N} & \\ & w & \longrightarrow f'_L(w) = 1 & \text{si } w \in L \end{array}$$

Considerando estas definiciones, podemos afirmar que:

- Si L es un lenguaje recursivo, su función característica es calculable.
- Si L es un lenguaje recursivamente enumerable, entonces es el dominio de una función parcialmente calculable (f'_L).

Ejemplo 9.2 (Función no calculable) En este ejemplo no sólo presentaremos una función no calculable sino también un lenguaje que no es recursivo y otro que ni siquiera es recursivamente enumerable.

El conjunto de todas las Máquinas de Turing asociadas a funciones definidas de $\mathbb{N}$ en $\mathbb{N}$ es infinito pero numerable, basta tener en cuenta que una M.T. cualquiera puede describirse como una cadena finita de símbolos (tal y como se explicó en la definición de la Máquina de Turing Universal) así como el método que Gödel propuso para enumerar objetos que inicialmente no parecen ser enumerables (página 106). Por tanto, podemos describir dicho conjunto de M.T.'s de la siguiente forma:

$$MT = \{p_0, p_1, \ldots, p_n, \ldots\}$$

A partir de dicha lista de máquinas, podríamos definir la siguiente función:

$$\begin{array}{rcl} f: & \mathbb{N} & \longrightarrow \mathbb{N} \\ & n & \longrightarrow f(n) = \left\{ \begin{array}{ll} 0 & \text{si } p_n \text{ no para con el dato } n \\ k+1 & \text{si } p_n \text{ para con el dato } n \text{ y devuelve el valor } k \end{array} \right. \end{array}$$

Si suponemos que f es calculable, existe una Máquina de Turing que realiza el mismo trabajo que f. Consideremos que p_r sea esta máquina.

- Si $f(r) = 0 \Rightarrow p_r$ no para con la entrada r
- Si $f(r) \neq 0 \Rightarrow f(r) = k+1$ donde k es el resultado que devuelve la máquina p_r

En cualquiera de los dos casos, la situación que se produce es absurda ya que f y p_r no tienen el mismo comportamiento, por lo tanto, la suposición inicial es falsa y f no es una función calculable.

El conjunto $L_1 = \{n \in \mathbb{N} / \ p_n$ no para si recibe como dato a $n\}$ no es recursivamente enumerable y además, el conjunto $L_2 = \overline{L_1}$ es recursivamente enumerable pero no es recursivo. Veamos, a continuación, como justificar estas afirmaciones. En la sección 8.3 quedó demostrado que L_1 no es un lenguaje recursivamente enumerable, ya que es imposible construir una M.T. que lo reconozca. Por otra parte, $L_2 = \{n \in \mathbb{N} / \ p_n$ para si recibe como dato a $n\}$ es un lenguaje recursivamente enumerable porque la Máquina de Turing Universal permite saber si un número pertenece a L_2 y, por tanto, puede reconocer los elementos de L_2, sin embargo no es recursivo ya que su complementario $\overline{L_2} = L_1$ no es recursivamente enumerable (teorema 8.1).

9.2. Funciones recursivas

La recursión consiste en definir un concepto en términos de si mismo. Esta idea puede aplicarse a las definiciones de funciones, de conjuntos y, en particular, de lenguajes.

A continuación veremos una primera definición de función recursiva.

Definición 9.3 (Función recursiva)
Una función $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ se considera recursiva si tiene los siguientes elementos:

- *Definiciones básicas que establecen de manera axiomática el valor que toma la función para determinados valores del conjunto origen.*

  Por ejemplo, $fact(0) = 1$.

- *Una o más reglas recursivas que permiten calcular nuevos valores de la función a partir de otros valores conocidos.*

  Por ejemplo, $fact(n) = n \times fact(n-1), \quad n \geq 1$

- *Aplicar las definiciones básicas y las recursivas un número finito de veces debe ser suficiente para calcular cualquier valor de la función.*

A pesar de que esta definición es muy intuitiva no es lo suficientemente estricta como para definir los casos más complejos de recursividad. Por este motivo, se introducirán en las siguientes secciones, los conceptos de función recursiva primitiva y función μ-recursiva.

Diremos que una función recursiva es total si está definida para todos los valores del conjunto origen y parcial si lo está sólo para un subconjunto propio del conjunto origen.

Para definir un conjunto de manera recursiva se especifican ciertos objetos del conjunto y luego se describen uno o más métodos generales que permiten obtener nuevos elementos a partir de los existentes.

Por ejemplo, podemos definir recursivamente el lenguaje Σ^* así:

- $\lambda \in \Sigma^*$
- Para cada $w \in \Sigma^*$ y cada $a \in \Sigma$ tenemos que $aw \in \Sigma^*$

9.2.1. Funciones recursivas primitivas

Es posible definir las funciones recursivas utilizando sólo funciones básicas y ciertas reglas que permiten mezclar funciones para construir otras más complejas.

Las funciones consideradas básicas (Kleene) son las siguientes:

Función constante 0 Su valor es independiente del argumento.

$$C(x) = 0, \forall x \in \mathbb{N}$$

Función sucesor A cada número natural le hace corresponder su sucesor siguiendo el orden habitual, de menor a mayor. Por ejemplo, $S(5) = 6, \quad S(22) = 23$. Hay que hacer notar que la suma no se ha definido formalmente, por eso no puede ser utilizada en esta definición.

Funciones de proyección Reciben n argumentos y devuelven el argumento i-esimo, es decir la proyección sobre la dimensión i. $P_i^n(x_1, \ldots, x_n) = x_i$

A continuación veremos dos reglas que nos permitirán construir funciones más complejas a partir de las básicas.

Definición 9.4 (Regla de composición)
Dadas las siguientes funciones: $\alpha_1, \ldots, \alpha_m, \beta$ *definidas como*

$$\alpha_i : \mathbb{N}^n \longrightarrow \mathbb{N} \qquad \beta : \mathbb{N}^m \longrightarrow \mathbb{N}$$

Se define la composición de estas $m+1$ *funciones como* $\phi : \mathbb{N}^n \longrightarrow \mathbb{N}$ *tal que,*

$$\phi(x_1, \ldots, x_n) = \beta(\alpha_1(x_1, \ldots, x_n), \ldots, \alpha_m(x_1, \ldots, x_n))$$

Definición 9.5 (Regla de recursión primitiva)
Dadas dos funciones: α, β *definidas como* $\alpha : \mathbb{N}^n \longrightarrow \mathbb{N} \quad \beta : \mathbb{N}^{n+2} \longrightarrow \mathbb{N}$
Se puede definir a partir de ellas, una nueva función $\phi : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$ *tal que,*

$$\phi(0, x_1, \ldots, x_n) = \alpha(x_1, \ldots, x_n)$$

$$\phi(y+1, x_1, \ldots, x_n) = \beta(y, \phi(y, x_1, \ldots, x_n), x_1, \ldots, x_n)$$

En el caso particular de que $n = 0$, ϕ *se define así:*

$$\phi(0) = k$$

$$\phi(y+1) = \beta(y, \phi(y))$$

Definición 9.6
Decimos que una función es recursiva primitiva si se puede definir a partir de las funciones básicas mediante cero o más aplicaciones de las reglas de composición y de recursión primitiva.

Aunque el número de funciones recursivas primitivas es muy amplio, existen algunas funciones recursivas que no son primitivas. Un ejemplo es la función de Ackermann, que se define de la siguiente forma:

$$A(0, x) = x + 1$$
$$A(n+1, 0) = A(n, 1)$$
$$A(n+1, x+1) = A(n, A(n+1, x))$$

Por este motivo debemos incluir otra regla para la generación de funciones recursivas que nos permitirá ampliar su número, de esta manera llegamos a la definición de función μ-recursiva.

9.2.2. Funciones μ-recursivas

Para ampliar el número de funciones que podemos generar a partir de las funciones recursivas primitivas, añadiremos una nueva regla de composición: la minimización.

Definición 9.7 (Regla de minimización)
Dada la función: $\alpha : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$ *se define la función de minimización* $\phi : \mathbb{N}^n \longrightarrow \mathbb{N}$ *tal que,*

$$\phi(x_1, \ldots, x_n) = min\{y/\alpha(x_1, \ldots, x_n, y) = 0 \wedge \forall z < y, \phi(x_1, \ldots, x_n, y) \textit{está definida}\}$$

En otras palabras, ϕ hace corresponder a cada entrada $\overline{x}$ el menor entero que cumple que $\alpha(\overline{x}, y) = 0$.

Definición 9.8 (Función μ-recursiva)
Decimos que una función es μ-recursiva, o simplemente recursiva, si se puede definir a partir de las funciones básicas mediante cero o más aplicaciones sucesivas de las reglas de composición, recursión primitiva y minimización.

El concepto de recursividad está íntimamente unido al de computabilidad, ya que toda función computable es μ-recursiva. Además, el conjunto de las máquinas de Turing es equivalente al de las funciones μ-recursivas. Es decir un problema se puede resolver mediante máquinas de Turing si y sólo si se puede plantear utilizando funciones recursivas.

9.3. Problemas

9.1 Define recursivamente las siguientes funciones:

1. La suma de dos números enteros
2. La multiplicación de dos números enteros

9.2 Define recursivamente los siguientes lenguajes:

1. $L_1 = \{ww - 1/w \in (0+1)^*\}$
2. $L_2 = \{0^n 1^n / n \geq 0\}$
3. Dado el alfabeto $\Sigma = \{i, (,), +, -\}$, considera el lenguaje formado por expresiones aritméticas que pueden o no tener paréntesis. Por ejemplo:
   $(i + i)$, $\quad i + i$, $\quad (i + i) - (i + i)$, etc.

9.3 Demuestra que las siguientes funciones son recursivas primitivas:

1. La función constante K(x)= k
2. La función predecesor,

$$P(x) = \begin{cases} x-1 & x > 1 \\ 0 & x = 0 \end{cases}$$

3. La función suma
4. La función producto
5. La función sustracción propia,

$$sp(x,y) = \begin{cases} x-y & x > y \\ 0 & x \leq y \end{cases}$$

6. La función sustracción absoluta, $sa(x,y) = |x-y|$
7. Las funciones máximo y mínimo
8. La función signo,

$$sg(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \end{cases}$$

Tema 10

Introducción a la Complejidad Computacional

Contenido

10.1. Complejidad y Máquinas de Turing 137
10.2. Medidas de complejidad algorítmica 138
10.3. Problemas P, NP y NP-completos 141

De entre todos los problemas que pueden plantearse, el conjunto de aquellos que son computables, es decir, que pueden ser resueltos aplicando un algoritmo, es muy pequeño. Sin embargo, no todos los problemas computables son factibles en la realidad por requerir a veces demasiados recursos, ya sean de espacio de memoria o de tiempo. La teoría de la complejidad algorítmica es la encargada de definir los criterios básicos para saber si un problema computable es factible, dicho de otro modo, si existe un algoritmo eficiente para su resolución y en este caso cuál es su grado de eficiencia.

10.1. Complejidad y Máquinas de Turing

Hasta este momento hemos analizado si los problemas son solubles o insolubles, es decir, si pueden o no ser resueltos mediante una máquina de Turing (un algoritmo). Sin embargo, en la vida real los recursos de computo disponibles son limitados y por tanto podemos encontrarnos con problemas que son solubles, pero que podemos considerar *intratables* debido a la gran cantidad de tiempo y memoria que son necesarios para resolverlos. En esta sección introduciremos una terminología que nos permitirá analizar preguntas como *¿cuánto tiempo tardaremos en resolver un problema?*

A priori, establecer un criterio universal para poder saber cuán eficiente es un algoritmo no es sencillo porque un mismo problema puede resolverse sobre diferentes tipos de máquinas, con diferentes grados de eficiencia. La eficiencia de un algoritmo está en función del tiempo y de la cantidad de memoria que se necesite para ejecutar el programa, y esto a su vez depende del hardware utilizado.

Sin embargo, la Máquina de Turing permite que nos liberemos de la mayor parte de estas ligaduras materiales. Recordemos que al estudiar la Máquina de Turing no se impuso ningún límite de espacio ni de tiempo, es decir, la Máquina de Turing no padece limitaciones en cuanto a la longitud de la cinta utilizada ni en cuanto al número de movimientos que realiza (tiempo de ejecución). Por esta razón utilizaremos la Máquina de Turing para medir la complejidad de un algoritmo.

10.2. Medidas de complejidad algorítmica

La complejidad de una computación se mide por la cantidad de espacio y de tiempo que consume. Las computaciones eficientes tienen unas exigencias de recursos *pequeñas* (este calificativo debe ser considerado de una manera relativa). Los recursos que necesita una computación que acepta cadenas de algún lenguaje, suelen depender del tamaño (longitud) de la cadena de entrada.

En los siguientes párrafos consideraremos, en primer lugar, los recursos espaciales y después los temporales.

Si M es una máquina de Turing, denotaremos por L(M) al lenguaje que reconoce dicha máquina.

Definición 10.1 (Complejidad espacial)
La máquina M con k pistas tiene una complejidad espacial S(n), si para cualquier entrada de longitud n, consulta como máximo S(n) casillas. También podemos decir que M es una Máquina de Turing espacialmente acotada por S(n), o que L(M) es un lenguaje con complejidad espacial S(n).

Se cumple siempre que $S(n) \geq 1$

Teorema 10.1
Si una Máquina de Turing con k pistas y cota espacial S(n) acepta al lenguaje L, entonces existe una Máquina de Turing con una sola pista y cota espacial S(n) que también lo acepta.

Es decir, el número de pistas de trabajo utilizadas para aceptar un lenguaje no afecta a la complejidad espacial de L.

Definición 10.2 (Clases de complejidad espacial)
La familia de los lenguajes aceptados por Máquinas de Turing deterministas con complejidad espacial S(n) se llama ESPACIOD(S(n)). La familia de los lenguajes

aceptados por Máquinas de Turing no deterministas con esa misma complejidad espacial se llama ESPACION(S(n)). Estas clases se conocen como clases de complejidad espacial.

Teorema 10.2
Sean S_1, S_2 y S funciones de $\mathbb{N}$ en $\mathbb{N}$. Supongamos que $S_1(n) \leq S_2(n)$, para todo $n \in \mathbb{N}$, y que $c > 0$. Entonces se cumple:

1. *ESPACIOD($S_1(n)$) $\subseteq$ ESPACIOD($S_2(n)$)*
2. *ESPACION($S_1(n)$)$\subseteq$ ESPACION($S_2(n)$)*
3. *ESPACIOD(S(n)) $\subseteq$ ESPACION(S(n))*
4. *ESPACIOD(S(n)) = ESPACIOD(cS(n))*
5. *ESPACION(S(n)) $\subseteq$ ESPACIOD($S(n)^2$)*

Aunque el espacio es un recurso importante de cualquier Máquina de Turing, el tiempo de computación también lo es. Consideraremos que la complejidad temporal se mide por el número de movimientos que hace la máquina.

Definición 10.3 (Complejidad temporal)
Supongamos que M es una máquina de Turing que realiza como máximo T(n) movimie tos sobre una cadena de longitud n, entonces se dice que M tiene una complejidad temporal T(n) o que es una máquina con una cota temporal T(n). También se dice que L(M) es un lenguaje temporalmente acotado por T(n).

Siempre se cumple que T(n) $\geq$ n+1 (ya que n+1 es el número mínimo de movimientos necesarios para leer todos los símbolos que inicialmente hay en la cinta)

Definición 10.4 (Clases de complejidad temporal)
La familia de los lenguajes aceptados por Máquinas de Turing deterministas con complejidad temporal T(n), es TIEMPOD(T(n)). La familia de los lenguajes aceptados por Máquinas de Turing no deterministas con complejidad temporal T(n) es TIEMPON(T(n)). Estas clases se conocen como clases de complejidad temporal.

Teorema 10.3
Sean T_1, T_2 y T funciones de $\mathbb{N}$ en $\mathbb{N}$. Supongamos que $T_1(n) \leq T_2(n)$, para todo $n \in \mathbb{N}$. Entonces se cumple:

1. *TIEMPOD($T_1(n)$) $\subseteq$ TIEMPOD($T_2(n)$)*
2. *TIEMPON($T_1(n)$) $\subseteq$ TIEMPON($T_2(n)$)*
3. *TIEMPOD(T(n)) $\subseteq$ TIEMPON(T(n))*

Teorema 10.4
Si L ∈ TIEMPOD(f(n)) entonces, L ∈ ESPACIOD(f(n))

Demostración. Supongamos que M es la Máquina de Turing que reconoce a L y que realiza, como máximo, $f(n)$ movimientos sobre la cadena de longitud n. Es evidente que puede inspeccionar un máximo de $1 + f(n)$ celdas, por tanto, L ∈ ESPACIOD(f(n)+1) ⇒ L ∈ ESPACIOD(f(n)). □

Las funciones S(n) y T(n) que hemos utilizado no tienen porque ser una función exacta, sino sólo un indicador de la forma de variación del espacio o del tiempo ocupado, en función de la longitud de la entrada de datos. Habitualmente nos interesa poder comparar estas funciones con otras (habitualmente más simples) de manera que las tasas de crecimiento sean análogas.

Veamos a continuación la notación que se utiliza para comparar tasas de crecimiento.

Definición 10.5
Sean dos funciones $f, g : \mathbb{N} \to \mathbb{R}$. Se dice que:

- *$f = O(g)$ si existen dos constantes C y k tales que $\forall x \geq k, f(x) \leq Cg(x)$*
- *$f = o(g)$ si para cada constante C, existe k tal que $\forall x \geq k, f(x) \leq Cg(x)$*
- *$f = \theta(g)$ si $f = O(g)$ y $g = O(f)$*

La expresión $f = O(g)$ quiere decir que para valores suficientemente grandes de x, la función $f(x)$ es menor o igual a una función proporcional a g. La constante de proporcionalidad C puede ser muy grande de forma que el valor real de $f(x)$ puede ser mayor que $g(x)$, sin embargo la tasa de crecimiento de f no es mayor que la de g.

Si $f = \theta(g)$ las dos funciones tienen la misma tasa de crecimiento, es decir, los dos valores son aproximadamente equivalentes, con valores grandes de x.

Si $f = o(g)$ la tasa de crecimiento de f es menor que la de g, es decir, $f(x)$ será en última instancia menor que $g(x)$, pero no sabemos cuán grande tiene que ser x para que esto ocurra.

Por ejemplo, si pretendemos estudiar la eficiencia de un algoritmo ejecutado por una Máquina de Turing M, no existe una gran diferencia, a efectos prácticos, entre que su tiempo de ejecución T(n) sea $14n^2 + 25n - 4$ o que sea $3n^2$, ya que en ambos casos podemos decir que su tasa de crecimiento no es mayor que la de n^2, es decir, $T = O(n^2)$. Diremos que la máquina es de complejidad temporal n^2, o (n^2)-limitada en el tiempo. Las funciones anteriores tienden (cuando $n \to \infty$) a las asíntotas $14n^2$ y $3n^2$ respectivamente, que son funciones del tipo cn^2. Así pues estamos hablando de complejidad en términos asintóticos.

Cuando queremos hacer referencia al crecimiento de los recursos necesarios para la ejecución de un determinado algoritmo utilizaremos la notación O(f(n)), que representa su comportamiento asintótico. Así un tiempo de ejecución $14n^2 + 25n - 4$, se resume en que la complejidad del algoritmo es $O(n^2)$.

Los algoritmos cuya complejidad del tipo $O(n)$, $O(n^2)$, $O(n^3)$, en general $O(n^c)$, se llaman algoritmos polinómicos, o de complejidad polinómica. Los algoritmos que se comportan como 2^n (en general c^n) son algoritmos exponenciales, o de complejidad exponencial.

Otro aspecto diferente y muy importante dentro de la Teoría de la Complejidad es el que se refiere, no a la complejidad del algoritmo, sino a la complejidad del problema. Esto nos obliga a comparar todos los algoritmos posibles para resolver un determinado problema. Se conoce como *cota superior* a la complejidad de un problema a la complejidad del mejor algoritmo que se haya podido encontrar para resolverlo. Es posible probar, a veces, que no existe algoritmo que pueda resolver un determinado problema sin emplear como mínimo una cierta cantidad de recursos, a la que se llama *cota inferior*.

10.3. Problemas P, NP y NP-completos

Diremos que un algoritmo es eficiente si existe una Máquina de Turing determinista que lo ejecute con una complejidad temporal polinómica. A la clase de los algoritmos (o de los problemas que estos algoritmos resuelven) eficientes se la denomina clase P.

Existen algoritmos no deterministas que no siguen un flujo fijo, sino que actúan en función de una serie de decisiones tomadas en tiempo real. De entre los algoritmos no deterministas existe un amplio conjunto de ellos que pueden considerarse eficientes, pero es indemostrable que estén en P, debido precisamente a que no son deterministas. A esta clase de problemas se les llama NP.

Cualquier problema se puede plantear como un lenguaje formado por todas las soluciones posibles para ese problema. Un algoritmo que reconozca al lenguaje también servirá para resolver el problema. Por esta razón resulta equivalente considerar que P y NP son clases de problemas o clases de lenguajes.

Definición 10.6 (Clases P y NP)
La clase P de lenguajes esta compuesta por todos los lenguajes que acepta alguna Máquina de Turing determinista que tiene una cota temporal polinómica. La clase NP se compone por todos los lenguajes que acepta alguna Máquina de Turing no determinista con una cota temporal polinómica.

Un ejemplo de problema NP es el conocido problema de Hamilton: dado un conjunto de puntos, ¿puede encontrase un camino que pase una sola vez por cada uno de

los puntos?. Un algoritmo determinista para resolver el problema es muy costoso, sin embargo dada una posible solución, resulta muy sencillo comprobar que es válida.

En este terreno el problema principal que podemos plantearnos es el siguiente: ¿es el conjunto NP igual al conjunto P? Este es el llamado **problema P-NP**, y todavía no tiene solución. Es evidente que P $\subset$ NP, pero para poder demostrar que la igualdad se cumple habría que demostrar que todo problema NP es también P, es decir, habría que encontrar una forma de transformar una Máquina de Turing no determinista con cota temporal polinómica en una Máquina de Turing determinista con cota temporal también polinómica. Por otro lado, tampoco se ha podido demostrar la desigualdad entre los dos conjuntos, para ello habría que encontrar un lenguaje que pertenezca a NP y que no pertenezca a P.

Dentro de la clase NP hay un cierto número de problemas que pueden catalogarse entre los más *duros*, en el siguiente sentido: si se encontrase un algoritmo de tiempo polinómico para cualquiera de ellos habría un algoritmo de tiempo polinómico para todo problema en NP. Se dice que cualquier problema de esta categoría es **NP-completo**.

Definición 10.7
Se dice que un lenguaje L_1 es reducible en tiempo polinómico a un lenguaje L_2 si hay una función f computable en tiempo polinómico para la cual f(u) $\in L_2$ sii u $\in L_1$.

Se utiliza la notación $<_p$ para indicar que L_1 es reducible en tiempo polinómico a L_2. Observar que si $L_1 <_p L_2$ entonces determinar si w $\in L_1$ no es más difícil que determinar si f(w) $\in L_2$. Basándonos en la misma idea podemos decir que un problema es reducible en tiempo polinómico a otro.

Definición 10.8 (Problemas NP-completos)
Un problema $P \in NP$ es NP-completo si todos los demás problemas de la clase NP se pueden reducir a él en tiempo polinómico.

Esta clase de problemas es importante porque si pudiéramos encontrar una solución en tiempo polinómico en una máquina de Turing determinista para un solo problema NP-completo habríamos demostrado que P=NP.

Ejemplos de problemas NP-completos

SAT Dada una expresión lógica, por ejemplo: $p \wedge (q \vee \neg p)$, hay que determinar si es posible satisfacerla, es decir, si es posible encontrar valores para los predicados p y q que hagan que la expresión tenga un valor verdadero.

CV - cobertura de vértices Dado un grafo G con un conjunto de vértices V y un conjunto de arcos E, y dado un número k, se trata de averiguar si existe un subconjunto V' de V, con no más de k elementos, tal que para todo arco de E alguno de los nodos unidos por este arco pertenece a V'.

Apéndice A

Generadores automáticos de analizadores léxicos y sintácticos

Contenido

A.1. Generador de analizadores léxicos 143
A.2. Generador de analizadores sintácticos 149

Existen herramientas que generan analizadores léxicos a partir de la definición de una serie de expresiones regulares. Estos analizadores llevan a cabo el procesamiento secuencial de una colección de caracteres de manera que al encontrar una cadena que *encaja* con una de las expresiones definidas realiza las acciones que se hayan indicado previamente. Internamente, los analizadores léxicos se comportan como Autómatas Finitos que reconocen expresiones regulares. Los analizadores léxicos constituyen una parte fundamental de cualquier compilador pero también pueden ser utilizados con otros objetivos, por ejemplo, para modificar la estructura de un fichero de texto.

Análogamente, existen herramientas que generan analizadores sintácticos a partir de la definición de una gramática independiente del contexto. Concretamente, generan un reconocedor ascendente del tipo LR(1).

La utilización conjunta de ambas herramientas simplificará el proceso de diseño y construcción de traductores para un determinado lenguaje formal. Esta será una de sus principales aplicaciones en el ámbito de esta asignatura.

A.1. Generador de analizadores léxicos

Existen diferentes programas que, con un comportamiento similar, permiten generar automáticamente analizadores léxicos. Entre los más utilizados están PCLEX, LEX y FLEX, estos dos últimos son utilizados en entorno UNIX/LINUX. Todos ellos

traducen la descripción de un analizador léxico, realizada en un metalenguaje al que llamaremos **Lex**, a un programa escrito en C. Utilizando las opciones adecuadas es posible generar dicho programa escrito en otros lenguajes de alto nivel como C++, Java, Pascal, etc.

Lex es un lenguaje de alto nivel de propósito específico, diseñado para representar expresiones regulares. Además, el código puede extenderse con secciones escritas en C o en el lenguaje de programación correspondiente.

A.1.1. Cómo utilizar PCLEX

PCLEX se ejecuta en línea de comandos, de la siguiente forma:
`PCLEX [opciones]` *nombre_fichero*

nombre_fichero es el nombre del fichero que contiene la especificación del analizador léxico en **Lex**. Por claridad, se recomienda que el nombre de estos ficheros tenga extensión **.l**. Por defecto se generará un fichero con el mismo nombre y con extensión **.c**. Por ejemplo, si el fichero de entrada se llama *ejemplo.l*, el de salida se llamará *ejemplo.c*.

Se pueden utilizar las siguientes opciones a la hora de ejecutar PCLEX:

-c	el fichero de salida tiene el nombre *yylex.c*
-C<nom_fichero>	el fichero de salida tiene el nombre que indica <nom_fichero>
-h	muestra una pantalla de ayuda
-i	construye un analizador insensible a la diferencia mayúsculas/minúsculas
-n	suprime las directivas "#line" en el fichero de salida
-p<nom_fichero>	utiliza <nom_fichero> como fichero *esqueleto* para construir el fichero de salida, en lugar de utilizar el fichero por defecto
-s	suprime la regla por defecto, es decir, las entradas que no encajen con ninguna regla provocan la salida del programa con el mensaje : *pclex scanner jammed*

Los demás programas anteriormente mencionados se utilizan de forma similar aunque el significado de las opciones puede variar.

A.1.2. Estructura de un programa Lex

Un programa **Lex** se divide en tres partes, cada una de ellas se separa de la siguiente utilizando el delimitador %%. Es decir, el programa tendría el siguiente aspecto:

zona de definiciones
%%
zona de reglas
%%
procedimientos del programador

Zona de definiciones En esta parte del programa se pueden definir expresiones regulares que se utilizarán posteriormente. También se pueden incluir todas las definiciones en C que sean necesarias. Las definiciones en C, deberán ir encerradas entre los delimitadores %{ y %}. Por ejemplo:

```
%{
        #include "stdlib.h"
        int x,y;
%}
```

Las definiciones de expresiones regulares, tendrán el siguiente formato:

```
nombre          expresión regular
```

A partir de este momento cada vez que deseemos utilizar esa expresión podemos hacerlo escribiendo su nombre entre llaves. En el siguiente ejemplo, se define una expresión regular llamada *digito*, que representa a un dígito cualquiera y otra, llamada *entero*, que representa a una colección de dígitos, o sea a un número entero y positivo:

```
digito      [0 − 9]
entero      {digito}+
```

Zona de reglas Cada regla está formada por una expresión regular seguida por una serie de acciones (codificadas en C) que serán las que el analizador léxico ejecute cuando encuentre una cadena de caracteres que *encaje* con la expresión regular. Por ejemplo:

```
ab*     {printf("hola");}
```

Procedimientos del programador En esta parte del programa se incluye el programa C que el usuario haya diseñado. En el caso más sencillo, en el que sólo se desee generar un analizador léxico, el programa principal deberá incluir al menos una llamada al procedimiento *yylex*. Es decir:

```
void main() {
    yylex();
}
```

yylex es el procedimiento que actúa como analizador léxico y cuyo código C es generado a partir de las especificaciones que aparecen en la **zona de reglas**. El

programa principal puede ser mucho más complejo e incluir todos los procedimientos que se deseen.

A.1.3. Cómo representar una expresión regular

Para describir las expresiones regulares que pueden aparecer en la zona de definiciones o en la zona de reglas, hay que seguir las siguientes normas:

- La concatenación entre símbolos se representa escribiéndolos uno junto a otro sin utilizar ningún símbolo especial. Por ejemplo:

  ```
  int
  char
  while
  ```

  representarían palabras reservadas de C.

- El punto (.) es un comodín y representa a cualquier carácter (sólo a uno) del código ASCII salvo a la marca fin de línea (representada por \n). Por ejemplo:

  `a.` representa a la letra **a** seguida por cualquier otro carácter.

- Podemos utilizar los corchetes para representar la unión entre varios símbolos y el guión para representar un rango de valores:

`[a-z]`	representa a cualquier letra minúscula
`[Ff][Oo][Rr]`	representa a la palabra **for** escrita utilizando letras mayúsculas o minúsculas (sería la manera de representar palabras reservadas en un lenguaje como Pascal)

- Las repeticiones de símbolos, o de conjuntos de símbolos, se pueden representar utilizando diferentes operadores:

operador	**nº de repeticiones**	**ejemplo**	**cadenas válidas**
*	$0, 1, \ldots$	ab*	a, ab, abb, ...
+	$1, 2, \ldots$	ab+	ab, abb, ...
{n}	n	ab{3}	abbb
{n,m}	n, ...,m	ab{3,5}	abbb, abbbb, abbbbb
?	0,1	ab?	a, ab

- La barra vertical (|) representa la unión entre expresiones regulares. Por ejemplo:

  `ab|cd` representa a la cadena **ab** o a la cadena **cd**

Marcas de contexto Si utilizamos ^ al comienzo de una expresión, ésta sólo se tendrá en cuenta en el caso de que la cadena analizada esté al comienzo de una línea.

Si utilizamos $ al final de una expresión, ésta sólo se tendrá en cuenta en el caso de que la cadena analizada esté al final de una línea.

Si utilizamos / entre dos expresiones, sólo se tendrá en cuenta la primera de ellas en el caso de que aparezca seguida por la segunda. Ejemplos:

`^ab`	la cadena **ab** debe aparecer al comienzo de la línea
`ab$`	la cadena **ab** debe aparecer al final de la línea
`ab/cd`	la cadena **ab** debe aparecer seguida por la cadena **cd**

Prioridades Los operadores que hemos visto tienen diferentes prioridades. Aparecen listados a continuación, de mayor a menor prioridad:

()	paréntesis
[]	unión entre símbolos
* + ? { }	repeticiones
ee	concatenación
|	unión de expr. regulares
^$	indicadores de contexto

A.1.4. Variables y procedimientos predefinidos

yylex() es el procedimiento principal a partir de las expresiones regulares definidas, actúa como un analizador léxico.

yytext es una variable de tipo cadena de caracteres y almacena la cadena que acaba de ser analizada por el scanner.

yyleng es una variable entera que almacena la longitud de yytext.

yyin, yyout son los nombres de los ficheros de entrada y de salida del analizador léxico.

ECHO es una acción predefinida que escribe la cadena analizada en el fichero de salida, por tanto, es equivalente a la instrucción

```
{fprintf(yyout,"%s",yytext);}
```

REJECT hace que el scanner analice por segunda vez la misma cadena. En este segundo análisis, la regla que contiene a REJECT no será tenida en cuenta.

A.1.5. Condiciones de comienzo

Las condiciones de comienzo se pueden activar o desactivar dependiendo de la llegada de algún símbolo o cadena de símbolos. Existen dos tipos, las **exclusivas** y las **no exclusivas**.

Se definen en la primera zona del programa utilizando las palabras reservadas **%s**, **%** o **%Star** para las no exclusivas, y **%x** para las exclusivas. Después deberán escribirse los nombres de las condiciones que se vayan a utilizar. Por ejemplo:

```
%s nombre1 nombre2 ...
```

Para activarlas se utilizará la acción `BEGIN(nombre)` y para desactivarlas `BEGIN(0)`

Se utilizan colocando su nombre, encerrado entre los símbolos <>, delante de una regla. Por ejemplo:

```
<nombre>ab*          ECHO;
```

Esta regla sólo se tendrá en cuenta si la condición `nombre` está activa.

Las condiciones de comienzo exclusivas se caracterizan porque las reglas que no llevan su nombre delante sólo se tienen en cuenta si la condición está desactivada. En el caso de las condiciones no exclusivas, estas reglas se tienen en cuenta en cualquier caso.

El siguiente ejemplo permite eliminar los comentarios del fichero de entrada. Suponemos que existen dos tipos diferentes de comentarios, unos encerrados entre llaves y otros entre (* y *).

```
%x coment1 coment2
%%
"{"                  BEGIN(coment1)
<coment1>"}"         BEGIN(0)
"(*"                 BEGIN(coment2)
<coment2>"*)"        BEGIN(0)
<coment1,coment2>.   ;
```

A.1.6. Acciones

Cuando el analizador léxico encuentra una cadena de caracteres que *encaja* con alguna de las expresiones regulares definidas, ejecuta las acciones asociadas a esta expresión. Estás acciones pueden ser las predefinidas ECHO o REJECT, o cualquier instrucción escrita en C. A continuación se describen algunas situaciones especiales:

- Si los caracteres de la entrada pueden encajar con diferentes expresiones regulares, se elegirá aquella que valide la cadena más larga.

```
ab          acción1
ab+         acción2
```

Teniendo en cuenta el ejemplo anterior, si en el fichero de entrada está incluida la cadena **abb** se ejecutará la `acción2`, ya que la primera regla solo validaría 2 caracteres y la segunda validaría 3.

En el caso de que dos reglas diferentes validen cadenas de la misma longitud, se elegirá la que aparezca en primer lugar. Por ejemplo, si la cadena **hola** aparece en el fichero de entrada y se han definido las dos reglas siguientes:

```
hola        acción1
[a-z]+      acción2
```

Se ejecutará la `acción1` simplemente porque está escrita en primer lugar.

- La acción que se ejecuta por defecto cuando un carácter (o cadena de caracteres) no encaja con ninguna de las expresiones definidas, consiste en escribir dicho carácter en el fichero de salida (yyout); salvo que se haya utilizado la opción **-s** al ejecutar PCLEX.

- Si detrás de una expresión regular sólo escribimos el símbolo |, la acción asociada a esta expresión es la misma que la asociada a la siguiente. Los siguiente ejemplos son, por tanto, equivalentes:

```
hola        |                       hola        acción
[a-z]+      acción                  [a-z]+      acción
```

- Si detrás de una expresión regular sólo escribimos el símbolo **;** no hay ninguna acción asociada a esta expresión, ni siquiera escribir la cadena validada en el fichero de salida, podría entenderse como *no hacer nada*.

A.2. Generador de analizadores sintácticos

Existen programas como YACC (Yet Another Compiler-Compiler), PCYACC o BISON, este último para entornos LINUX, que permiten generar automáticamente analizadores sintácticos de tipo LR(1), a partir de la definición de una gramática independiente del contexto descrita con una notación similar a la BNF(Backus Normal Form).

Concretamente, se genera una función, llamada **yyparse**, que reconocerá programas escritos en el lenguaje definido por la gramática y detectará los errores si los hubiera.

La función **yyparse()**, llama repetidamente al analizador léxico **yylex()** que convierte cadenas de caracteres del fichero de entrada en símbolos terminales de la gramática (llamados **tokens**). Utilizando una terminología anglosajona, al analizador

léxico se le denomina **scanner** y al sintáctico se le denomina **parser**. La forma convencional por la que el scanner envía al parser información adicional sobre los tokens es a través de la variable **yylval**. Por defecto esta variable es de tipo *int* pero, como veremos, esto se puede cambiar.

A.2.1. Cómo utilizar PCYACC

PCYACC se ejecuta en línea de comandos, de la siguiente forma:

`PCYACC` [`opciones`] *nombre_fichero*

nombre_fichero es el nombre del fichero que contiene la especificación de la gramátic por claridad se recomienda que el nombre de estos ficheros tenga extensión **.y**. Por defecto se generará un fichero con el mismo nombre y con extensión **.c**. Por ejemplo, si el fichero de entrada se llama *ejemplo.y*, el de salida se llamará *ejemplo.c*.

Se pueden utilizar las siguientes opciones a la hora de ejecutar PCYACC:

-c	el fichero de salida tiene el nombre *yytab.c*
-C<nom_fichero>	el fichero de salida tiene el nombre que indica <nom_fichero>
-d	se genera un fichero cabecera llamado *yytab.h*
-D<nom_fichero>	se genera un fichero cabecera con el nombre que indica <nom_fichero>
-h	muestra una pantalla de ayuda
-n	suprime las directivas "#line" en el fichero de salida
-p<nom_fichero>	utiliza <nom_fichero> como fichero *esqueleto* para construir el fichero de salida, en lugar de utilizar el fichero por defecto (*yaccpar.c*)
-r	informa durante la ejecución
-s	genera vectores internos cuyos elementos son de tipo *short int*
-S	el programa se para después de realizar la fase de análisis sintáctico
-t	construye un árbol parser y lo almacena en el fichero *yy.ast*
-T<nom_fichero>	construye un árbol parser y lo almacena en el fichero <nom_fichero>
-v	genera un fichero llamado *yy.lrt* con información sobre el proceso y la tabla parser
-V<nom_fichero>	genera el fichero de la opción anterior pero con el nombre <nom_fichero>

Otros programas como YACC o BISON se utilizan de forma similar aunque el significado de las opciones pueda variar.

A.2.2. Estructura de un programa para YACC

Un programa para YACC tiene la misma estructura que un programa para LEX. Es decir, tiene tres partes, con el mismo significado que en el caso anterior.

1. En la primera parte, la **zona de definiciones**, se pueden incluir declaraciones en C, de la misma forma que se hacía con LEX.

   Además, es necesario realizar algunas definiciones que necesita conocer el parser, para ello se utilizan palabras reservadas (todas comienzan por %).

   - La definición del símbolo inicial de la gramática se realiza utilizando la palabra reservada **%start**. Por ejemplo:

     ```
     %start programa
     ```

   - La definición de los símbolos terminales de la gramática se realiza utilizando la palabra reservada **%token**. Por ejemplo:

     ```
     %token NUMERO IDENTIFICADOR
     ```

2. En la segunda parte, la **zona de las reglas**, se describe la G.I.C. siguiendo la siguiente notación:

   - El símbolo : se utiliza para separar la parte izquierda de una producción de la parte derecha.
   - Todas las reglas que tienen la misma parte izquierda se pueden separar con el símbolo |, sin necesidad de repetir la parte izquierda. Una colección de producciones con la misma parte izquierda debe acabar con ;. Por ejemplo, las siguientes definiciones son equivalentes:

     ```
     lista_var: lista_var var        lista_var: lista_var var
     lista_var: var                             |var
                ;                               ;
     ```

   - Los símbolos de la gramática que no hayan sido declarados como tokens, se considerarán símbolos no terminales, excepto los caracteres simples, encerrados entre comillas que también se consideran símbolos terminales. Por ejemplo: '+','*'.

3. En la tercera parte del programa, **procedimientos del programador** es necesario, como mínimo llamar al procedimiento **yyparse()**. También es necesario que el programador defina la rutina **yyerror**. Esta rutina será llamada por el analizador cada vez que encuentre un error sintáctico. Un ejemplo de definición de yyerror puede ser:

   ```
   void yyerror(char *s) {
           printf("%s"\n,s);
   }
   ```

A.2.3. Gramáticas atribuidas

En ocasiones es necesario trabajar con información adicional sobre determinados símbolos de la gramática. Diremos que estos símbolos tienen *atributos*. Esta información se almacena en una variable predefinida llamada *yylval* que es de tipo *YYSTYPE*. Como hemos comentado anteriormente, por defecto, ese tipo es *int*, pero en ocasiones puede resultar útil cambiar su definición.

Este cambio en la definición de *YYSTYPE* se realiza en la zona de definiciones y se puede llevar a cabo de diferentes formas. La más sencilla consiste en utilizar la palabra reservada **%union**, de la siguiente forma:

```
%union   {
         int num;
         char cadena[10];
}
```

- Suponiendo que YYSTYPE ha sido definido según el ejemplo anterior, es necesario especificar que símbolos de la gramática van a tener atributos y de que tipo van a ser. Esto se realiza con la palabra reservada **%type**. Por ejemplo:

  ```
  %type <num> NUMERO expresion
  %type <cadena> IDENTIFICADOR
  ```

- En el caso de los símbolos terminales, esto mismo se puede definir con %token. Por ejemplo:

  ```
  %token <num> NUMERO
  %token <cadena> IDENTIFICADOR
  ```

- En la parte derecha de una producción es posible insertar acciones escritas en C que serán ejecutadas cuando el analizador sintáctico llegue a ese punto del análisis. Estas acciones deberán ir encerradas entre llaves.

  Cuando queramos utilizar los atributos asociados a algunos símbolos de una producción, utilizaremos $$, $1, $2, ...

  $$ es el atributo del símbolo que aparece en la parte izquierda de la producción.

  $1 es el atributo del primer símbolo de la parte derecha, $2 el del segundo símbolo, etc. Por ejemplo:

  ```
  expr:expr'+'expr          {$$=$1+$2;}
  ```

A.2.4. Prioridad y asociatividad de operadores

YACC permite especificar la prioridad y la asociatividad de determinados símbolos de la gramática (utilizados normalmente como operadores).

Utilizando las palabras reservadas **%left, %right, %nonassoc**, podemos definir el tipo de asociatividad que tienen los símbolos gramaticales. Además, la prioridad de los símbolos queda implícita al especificar su asociatividad. Los símbolos que aparecen en la misma línea tienen la misma prioridad entre si, y la prioridad será más alta cuanto más tarde(en el texto) haya sido definida la asociatividad. Por ejemplo:

```
%left '+', '-'
%left '*', '/'
```

Estas definiciones indican que la suma y la resta tienen la misma prioridad, ambas tienen asociatividad por la izquierda y, además, tienen una prioridad menor que la multiplicación y la división.

Bibliografía

[1] Hopcroft J. E. y Motwani R. y Ullman J. D. *Teoría de Autómatas, Lenguajes y Computación*. Pearson-Addison Wesley, 2008.

[2] Alfonseca E. y Alfonseca M. y Moriyón R. *Teoría de Autómatas y Lenguajes Formales*. McGraw-Hill, 2007.

[3] Isasi P. y Martínez P. y Borrajo D. *Lenguajes, Gramáticas y Autómatas. Un enfoque práctico*. Addison Wesley, 2001.

[4] Alfonseca M. y Sancho y Orga. *Teoría de Lenguajes, Gramáticas y Autómatas*. Ed. Universidad, 1993.

[5] Martin J. *Lenguajes Formales y Teoría de la Computación*. McGraw-Hill, 2003.

[6] Brookshear. *Teoría de la Computación. Lenguajes Formales, Autómatas y Complejidad*. Addison Wesley, 1993.

[7] Kelley D. *Teoría Autómatas y Lenguajes Formales*. Prentice Hall, 1995.

[8] Apple A. *Modern Compiler Implementation in Java/C*. Cambridge University Press, 1998.

[9] Aho A.V. y Sethi R. y Ullman J.D. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[10] Alfonseca y de la Cruz y Ortega y Pulido. *Compiladores e Intérpretes: Teoría y Práctica*. Prentice-Hall, 2006.

Índice alfabético

Árbol de derivación, 21, 77

Alfabeto, 15
  de entrada, 40
  de símbolos no terminales, 22
  de símbolos terminales, 22
Analizador
  léxico, 11, 143
  semántico, 12
  sintáctico, 12, 149
Atributos semánticos, 98
  heredados, 99
  sintetizados, 99
Autómata, 6
  conexo, 42
  equivalente, 45
  incompleto, 41
  minimización, 43
  de pila, 74
  finito determinista, 40
  finito no determinista, 47
  linealmente acotado, 127

Bombeo
  lema para gr. de tipo 2, 94
  lema para gr. de tipo 3, 65

Chomsky, 7
  jerarquía de, 8, **24**
Church-Turing, tesis, 120
Cierre
  de Kleene, 18
  positivo, 19
Compilador, 10
Complejidad
  espacial, 138
  temporal, 139
Concatenación
  de un lenguaje, 18
  de una palabra, 16

Derivación, 20
  directa, 20
Descripción instantánea
  de un autómata de pila, 75
  de un reconocedor LR(1), 90
  de una máquina de Turing, 108
Desplazamiento, 88

Ensamblador, 10
Estado
  muerto, 41
  final, 40
  inicial, 40
Expresión regular, 33

Forma sentencial, 84
Función
  μ-recursiva, 134
  calculable, 130
  característica de un lenguaje, 130
  de transición, 40
  parcialmente calculable, 130
  recursiva, 134
  recursiva primitiva, 133

Gödel, 6, 106
Gramática, 6, **22**
  bien formada, 29

equivalente, 26
limpia, 28
ambigua, 78
atribuida, 98
L-atribuida, 101
S-atribuida, 100
compresora, 124
con estructura de frase, 25, 123
dependiente del contexto, 25, **126**
independiente del contexto, 25, **74**
lineal, 35
LL(1), 85
LR(1), **92**
recursiva, 24
recursivamente enumerable, 24, **123**
regular, 26, **35**

Hilbert, 6
Homomorfismo, 94

Intérprete, 10
Inversa de una palabra, 17

Lenguaje, 6, **16**
universal, 16
recursivamente enumerable, 124
recursivo, 125
Lex, 143
LR-item, 89

Máquina de Turing, 107
universal, 118
Myhill-Nerode, teorema, 66

Palabra, 16
vacía, 16
anulable, 83
Potencia
de un lenguaje, 18
de una palabra, 17
Problema
de clase NP, 141
de clase P, 141
NP-completo, 142
Producción, 19
compresora, 20
de redenominación, 28
innecesaria, 27
no generativa, 28
anulable, 83
con prefijos comunes, 81
con recursividad por la izquierda, 81

Reconocedor
ascendente, 87
descendente, 80
LL(1), 81
LR(1), 87
Reducción, 88
Reflexión
de un lenguaje, 19
de una palabra, 17

Símbolo
inaccesible, 27
no generativo, 27
director
de una producción, 85
del LR-item, 89
inicial
de una cadena, 83
de una gramática, 22
seguidor, 84
Semántica, 97
Shannon, 7
Sustitución, 94

Tabla
de acciones, 88, 89
de símbolos, 13
Turing, 6, 105

Unión de lenguajes, 17

Yacc, 149