

Departamento de Automática
Escuela Politécnica Superior
Universidad de Alcalá



Proyecto Fin de Carrera

gp1990c (GNU Pascal 1990 Compiler)

Séptemběr - MMXIV

Autor: Diego Antonio Lucena Pumar
Titulación: Ingeniería Técnica en Informática de Gestión

Agradecimientos

“If my theory of relativity is proven successful, Germany will claim me as a German and France will declare me a citizen of the world. Should my theory prove untrue, France will say that I am a German, and Germany will declare that I am a Jew.”¹

Albert Einstein

En primer lugar agradecer a mis padres que desde pequeño me inculcaran que la única manera de sentirnos realmente humanos sea por medio de la razón, que todo el trabajo y el esfuerzo siempre da sus frutos y que la victoria, no es más que una sucesión de derrotas.

A todos los que lucháis por un mundo mejor, justo, sincero y más humano, también os dedico mi trabajo.

Agradecer también a mi hermana, compañera siempre en buenos y malos momentos.

Igualmente a mi tutora Virginia, por su enorme comprensión y apoyo a lo largo de todas las fases de este proyecto.

Hacer también una mención a todos las personas que a lo largo de mi vida me han mostrado afecto, cariño y amistad verdadera. A todos lo que han confiado en mi y en mi trabajo, también esto es más vuestro que mio.

¹“Si mi Teoría de la Relatividad es cierta, los alemanes dirán que soy alemán y los franceses que soy ciudadano del mundo. Pero si no, los franceses dirán que soy alemán, y los alemanes que soy judío.”

Prefacio

El presente texto consta de tres apartados conceptuales y seis capítulos cuyo propósito es dar un sentido pedagógico a la exposición, partiendo de lo general para llegar a lo particular.

I. Introducción

Capítulo I: El primer capítulo pretende sintetizar la matemática necesaria y básica sobre la que se asientan los desarrollos de la Teoría de Compiladores y Analizadores Automáticos de Lenguajes. Por ello, se fundamenta en tres corrientes matemáticas:

- i. Teoría de Conjuntos.
- ii. Funciones.
- iii. Teoría de Grafos.

Capítulo II: Este capítulo tiene el objetivo de sintetizar todo el trabajo sobre el que gira `gp1990c`:

- i. Motivo del proyecto.
- ii. Síntesis, partes y desarrollo de las fases de Análisis Léxico (`gp1991a`) y Análisis Sintáctico (`gp1990sa`) de Pascal ISO 1990 (el conjunto Software).

Capítulo III: En este apartado se describe el Lenguaje Pascal y su evolución a lo largo de los años.

En el contexto de su nacimiento (principios de los años setenta del siglo XX) la Computación sufrió una intensa evolución desde una computación para grandes corporaciones y con altos costes de explotación, pasando por las primeras máquinas de Apple y PC de IBM, hasta lo que conocemos hoy día.

Por ello se describen los lenguajes sobre los que se basó Nicklaus Wirth para crear Pascal y como, su concepto de lenguaje con un repertorio discreto de instrucciones (frente a otros de la época como Fortran o Algol) además de su expresividad han sido fundamentales para la construcción de importantes lenguajes que hoy día y como es el caso de ADA, son líderes indiscutibles en la Computación a Tiempo Real.

Capítulo IV: Es un capítulo que precede al desarrollo del propio Analizador Léxico y que busca dar sentido y forma a la sucesión de compiladores de Pascal. Son destacables los hitos:

- i. Creación de PUG (**P**ascal **U**ser's **G**roup).

- ii. Pascal-P2 y P4.
- iii. UCSD Pascal: Concepto de p-systems e independencia en su ejecución.
- iv. Borland Pascal: Compilador asequible para estudiantes de programación en la era PC.
- v. PFC: Implementación GPL de un potente y multiplataforma IDE de Pascal.

II. gp19901a (Analizador Léxico)

Capítulo V: Dicho Capítulo tiene como objetivo sintetizar los aspectos matemáticos de los Analizadores Léxicos. Por ello, en el siguiente orden, se estudian:

- i. Lenguajes Formales.
- ii. Teoría de Autómatas.

De igual manera se incluye una introducción al uso de LEX además del análisis de los elementos léxicos que forman parte del compilador:

- i. Expresiones Regulares.
- ii. Conjunto de Tokens.

III. gp1990sa (Analizador Sintáctico)

Capítulo VI: El mismo se resumen los aspectos fundamentales de los Analizadores Sintácticos. Contiene los siguientes subapartados:

- i. Lenguajes Formales (LFs)
- ii. Jerarquía de Chomsky (JC)
- iii. Gramáticas Formales
- iv. Analizadores Sintácticos
- v. Yacc (Yet another compiler-compiler)

IV. Anexos

Anexo A: Vida y obra de Blaise Pascal.

Anexo B: Código fuente: `gp1990sa.y`

Anexo C: Gramáticas de familia Pascal (Pascal; Modula, Oberon).

Anexo D: Cronología de publicaciones de *Pascal Users group Newsletter*.

Anexo E: Introducción a UNIX y GNU.

Anexo F: Introducción a Linux.

Anexo G: Tabla ASCII $[0, (2^8 - 1)]$

Anexo H: Manifiesto GPL v.2

Índice general

I	Introducción	1
	Francisco M. Ortega Palomares. <i>Ideario</i>	3
1.	Formalismos	5
1.1.	Introducción a la Teoría de Conjuntos	5
1.1.1.	Definiciones	5
1.1.2.	Operaciones	6
1.2.	Relaciones	11
1.2.1.	Definiciones	11
1.2.2.	Producto Cartesiano	11
1.2.3.	Representaciones	12
1.2.4.	Tipos	13
1.3.	Funciones	15
1.3.1.	Propiedades	16
1.3.2.	Tipos	16
1.3.3.	Operaciones	17
1.4.	Álgebra de Boole	19
1.4.1.	Generalidades	19
1.4.2.	Lógica Binaria	20
1.4.3.	Funciones Booleanas	21
1.5.	Nociones sobre Grafos	21
1.5.1.	Definiciones	21
1.5.2.	Clasificación	23
1.5.3.	Tipos	25
1.5.4.	Circuitos y Ciclos	26
1.5.5.	Árboles	27
1.5.5.1.	Generalidades	27
1.5.5.2.	Árboles Generadores	27
1.5.5.2.1.	Algoritmo de Prim	27
1.5.5.2.2.	Algoritmo de Kruskal	29
1.5.5.3.	Árboles <i>m-arios</i>	30
	Notas	33
2.	Resumen: Proyecto gp1990c	35
2.1.	Objetivos	35
2.2.	Descripción general del proyecto	35

2.3.	Transformación de una Expresión Regular en Software	36
2.4.	Breve descripción de <code>gp19901a</code>	37
2.5.	Breve descripción de <code>gp1990sa</code>	38
2.6.	Métodos y Fases de desarrollo	38
2.7.	Entorno de desarrollo	41
	Notas	43
3.	El Lenguaje de Programación Pascal	45
3.1.	Introducción	45
3.2.	Influencias del Lenguaje Pascal	46
3.2.1.	Fortran (The IBM Mathematical Formula Translating System)	46
3.2.1.1.	Análisis de Fortran	48
3.2.2.	ALGOL (ALGO ^r ithmic Language)	50
3.2.2.1.	Definiciones	50
3.2.2.2.	Historia	50
3.2.2.3.	ALGOL 60	51
3.2.2.4.	ALGOL W	53
3.3.	El Lenguaje Pascal	53
3.3.1.	Pascal ISO 7185:1990	54
3.3.1.1.	Alfabeto	54
3.3.1.2.	Tipos de Datos	55
3.3.1.3.	Biblioteca	57
3.3.1.4.	Estructura de un programa	59
3.4.	Evoluciones del Lenguaje Pascal	60
3.4.1.	Modula/Modula-2	60
3.4.1.1.	Símbolos y Gramática	61
3.4.2.	Ada	62
3.4.3.	Oberon	64
3.4.3.1.	Símbolos y Gramática	64
	Notas	67
4.	Compiladores del Lenguaje Pascal	69
4.1.	Pascal User's Group (PUG)	69
4.1.1.	Historia	69
4.2.	Pascal-P (The Portable Pascal Compiler)	70
4.2.1.	Historia Pascal CDC 6000	70
4.2.2.	Historia Pascal-P	70
4.3.	UCSD Pascal	71
4.3.1.	Historia	71
4.4.	Pascaline	74
4.4.1.	IP Pascal	74
4.5.	Borland Pascal	74
4.5.1.	Historia	74
4.5.2.	Valores internos para datos numéricos simples	75
4.5.3.	Biblioteca estándar	75
4.6.	GNU Pascal Compiler (GPC)	77

4.6.1.	¿Qué es GPC?	77
4.6.2.	Estructura de GPC	77
4.7.	FreePascal	78
4.7.1.	¿Qué es FreePascal?	78
4.7.2.	Historia	78
4.7.2.1.	Versiones	78
4.7.3.	Estructura de FreePascal	79
Notas		81

II gp19901a (Analizador Léxico) 83

Gustavo Adolfo Bécquer. *XI* 85

5. Formalidades del Analizador Léxico 87

5.1.	Introducción	87
5.2.	Teoría de Lenguajes	89
5.2.1.	Definiciones	89
5.2.2.	Palabras	89
5.2.2.1.	Operaciones	89
5.2.3.	Lenguajes	91
5.2.3.1.	Operaciones	91
5.3.	Lenguajes Regulares	94
5.4.	Expresiones Regulares	95
5.5.	Autómatas	96
5.5.1.	¿Qué es un Autómata?	96
5.5.2.	Representación	97
5.5.3.	Autómata Finito Determinista	97
5.5.4.	Autómata Finito no Determinista	98
5.5.5.	Algoritmo: AFnD \Rightarrow AFD	99
5.5.6.	Algoritmo: Expresión Regular \Rightarrow AFnD	102
5.6.	El Lenguaje LEX	103
5.7.	Código fuente: gp19901a.l	104
5.7.1.	Expresiones Regulares	104
5.7.2.	Tokens	104
Notas		107

III gp1990sa (Analizador Sintáctico) 109

Antonio Machado. *Retrato* 111

6. Formalidades del Analizador Sintactico 113

6.1.	Introducción a los Lenguajes Formales (LFs)	113
6.1.1.	Definiciones	113
6.1.2.	Especificación de los LF's	114
6.1.3.	¿Qué diferencia a un Lenguaje Natural (Humano) de un LF?	114

6.2. Gramáticas Independientes de Contexto	115
6.2.1. Derivaciones	116
6.2.1.1. Representación mediante Árboles	116
6.3. Jerarquía de Chomsky (JC)	117
6.3.1. Niveles	117
6.4. Descripción de Gramáticas Formales	120
6.4.1. Backus-Naur Form	120
6.4.2. Wijngaarden Form	121
6.5. Analizadores Sintácticos	121
6.6. Análisis Sintáctico Descendente	122
6.6.1. Autómatas LL(1)	122
6.7. Análisis Sintáctico Ascendente	124
6.8. Yacc (Yet another compiler-compiler)	124
Notas	127

IV Anexos y Formalidades 129

A. Blaise Pascal 131

B. gp1990sa.y 133

B.1. Yacc	133
---------------------	-----

C. Gramáticas 147

C.1. Pascal ISO 1990:7185	147
C.2. Modula-2	152
C.3. Oberon	156

D. Pascal Users group Newsletter 161

E. GNU's Not UNIX! 163

E.1. UNIX	163
E.2. Proyecto GNU	166
E.2.1. Licencia GPL	166
Notas	169

F. Linux 173

F.1. Historia	174
Notas	177

G. Tabla ASCII $[0, (2^8 - 1)]$ 179

H. GNU General Public License (Version 2, June 1991) 181

Bibliografía 187

Índice alfabético 189

Índice de figuras

1.1. Relación de Unión.	7
1.2. Relación de Intersección.	8
1.3. Relación de Resta.	9
1.4. Relación de Disjunción.	10
1.5. Relación de Diferencia Simétrica.	10
1.6. Relación de Complemento.	11
1.7. Representaciones genéricas del Producto Cartesiano.	12
1.8. Representaciones del Producto Cartesiano; $O \times P$	13
1.9. Representación genérica de una Relación Binaria mediante una Matriz.	14
1.10. Representaciones para la Función: $f(x + 5)$	15
1.11. Relaciones entre los principales elementos de una Función.	16
1.12. Tipos de funciones basadas en la relación de Dominio y Recorrido.	17
1.13. Representaciones para la Función: $\frac{8x^2+3x+3}{2}$	18
1.14. Representaciones para la Función: $\frac{-8x^2+3x+7}{2}$	18
1.15. Representaciones para la Función: $\frac{13x^3+20x^2-3x-5}{2}$	18
1.16. Representaciones para la Función: $\frac{3x+5}{8x^2-2}$	19
1.17. Representaciones para la función: $6x^2 + 1$	19
1.18. Representaciones comunes del Operador Booleano NOT.	21
1.19. Representaciones comunes del Operador Booleano OR.	21
1.20. Representaciones comunes del Operador Booleano AND.	22
1.21. Representaciones comunes del Operador Booleano XOR.	22
1.22. Ejemplos de Grafos.	23
1.23. Ejemplo de Multigrafo y Grafo no simple y Grafo dirigido.	24
1.24. Ejemplo de Grafos Isomorfos.	25
1.25. Ejemplo de Grafos: Completo, Regular y Bipartito.	26
1.26. Grafo origen para Algoritmo de Prim.	28
1.27. Grafo origen para Algoritmo de Kruskal.	29
1.28. Ejemplo de Árboles <i>m-arios</i>	30
1.29. Ejemplo de Árbol con Raíz.	31
1.30. Grafo de Königsberg.	33
2.1. Síntesis y Partes de gp1990c	36
2.2. Transformación desde una Expresión Regular a su Implementación.	36
2.3. Diagrama de Gantt para desarrollo de gp1990c	40
3.1. Relaciones entre los primeros Lenguajes de Programación.	46
3.2. Evolución del Lenguaje Fortran.	47

3.3. Símbolos especiales de Fortran 2003.	49
3.4. Evolución del Lenguaje ALGOL.	51
3.5. Evolución del Lenguaje Ada.	63
4.1. Evolución de Portable Pascal.	71
4.2. Evolución de compiladores para Pascal.	73
4.3. Arquitectura de GPC.	78
4.4. Arquitectura de FPC.	80
5.1. Relación entre el Analizador Léxico y el Programa Fuente.	88
5.4. Representación de un Autómata Finito.	97
5.5. Representación de Autómatas.	97
5.6. Ejemplo Autómata Finito Determinista.	98
5.7. Ejemplos AFD.	99
5.8. Ejemplo Autómata Finito no Determinista.	100
5.9. Autómata Finito no Determinista $a \cdot a^*$	101
5.11. Autómata Finito Determinista a partir de AFnD $a \cdot a^*$	102
5.12. Autómata Finito Determinista Mínimo a partir de AFnD $a \cdot a^* \equiv a^+$	103
5.13. Ciclo de Thompson.	103
5.14. Conjunto de Tokens para gp19901a	105
6.1. Relación entre: Teoremas, LFs y Cadenas de Caracteres.	114
6.2. Ejemplo genérico de Árbol de Derivación.	116
6.3. Ejemplo Árbol de Derivación para obtener: <i>abba</i>	117
6.4. Relación entre el Analizador Léxico, Analizador Sintáctico y el Programa Fuente.	122
6.5. Relación entre el primitivas de Lex y Yacc.	125
E.1. Evolución de UNIX y sus distintas versiones en el tiempo.	165
E.2. Evolución de la Licencia GPL.	166
F.1. Evolución de Linux y distintas distribuciones en el tiempo.	175

Índice de tablas

1.1. Tabla de Pesos Crecientes para Figura 1.27	30
2.1. Comparativa para los distintos tipos de implementaciones para un AL.	37
3.1. Convención entre <i>Reference Language</i> y otras publicaciones de ALGOL.	51
4.1. Versiones de Pascal-P.	70
4.2. Relación entre la Biblioteca Estándar de Pascal y el Cálculo Matemático.	76
4.3. Comparativa entre compiladores de Pascal.	77
5.1. Ejemplo de relación entre: Lexema, Patrón y Token.	88
5.2. Relación de operadores entre Lenguajes Regulares y Expresiones Regulares.	95
5.3. Operadores comunes para Expresiones Regulares en UNIX.	96
5.4. $L(M)$ para Autómatas.	101
6.1. Relación entre: Nivel, Lenguaje y Autómata en la JC.	117
G.1. Tabla ASCII $[0, (2^7 - 1)]$	179
G.2. Tabla ASCII $[2^7, (2^8 - 1)]$	180

Conjuntos de Números y Operadores

\mathbb{N}	Conjunto de los números Naturales.
\mathbb{Z}	Conjunto de los números Enteros.
\mathbb{Q}	Conjunto de los números Racionales.
$\mathbb{R}, \mathbb{R}^+, \mathbb{R}^n$	Conjunto de los números Reales.
\emptyset	Conjunto Vacío.
\in	Operador de Pertenecia.
\notin	Operador de no Pertenecia.
\subseteq	Subconjunto Exclusivo.
\cup	Operador de Unión.
\cap	Operador de Intersección.
\times	Producto Cartersiano.
\wedge	Operador de Conjunción.
\vee	Operador de Disjunción.
\oplus	Operador OR Exclusivo.
\rightarrow	Operador de Implicación.
\leftrightarrow	Operador de Equivalencia.
\Rightarrow	Implicación.
\Leftarrow	Implicación Inversa.
\Leftrightarrow	Equivalencia Lógica.
\circ	Operador de Concatenación.
\forall	Cuantificador Universal.
\exists	Cuantificador Existencial.

Alfabeto Griego

A	α	alfa
B	β	beta
Γ	γ	gamma
Δ	δ	delta
E	ϵ	epsilón
Z	ζ	dseta
H	η	eta
Θ	θ	zeta
I	ι	iota
K	κ	cappa
Λ	λ	lambda
M	μ	my
N	ν	ny
Ξ	ξ	xi
O	o	omicrón
Π	π	pi
P	ρ	rho
Σ	σ	sigma
T	τ	tau
Υ	υ	ypsilón
Φ	ϕ	fi
X	χ	ji
Ψ	ψ	psi
Ω	ω	omega

Parte I

Introducción

Ideario

Me da vértigo el punto muerto
y la marcha atrás,
vivir en los atascos,
los frenos automáticos y el olor a gasoil.

Me angustia el cruce de miradas
la doble dirección de las palabras
y el obsceno guiñar de los semáforos.

Me da pena la vida, los cambios de sentido,
las señales de stop y los pasos perdidos.

Me agobian las medianas,
las frases que están hechas,
los que nunca saludan y los malos profetas.

Me fatigan los dioses bajados del Olimpo
a conquistar la Tierra
y los necios de espíritu.

Me entristecen quienes me venden clines
en los pasos de cebra,
los que enferman de cáncer
y los que sólo son simples marionetas.

Me aplasta la hermosura
de los cuerpos perfectos,
las sirenas que ululan en las noches de fiesta,
los códigos de barras,
el baile de etiquetas.

Me arruinan las prisas y las faltas de estilo,
el paso obligatorio, las tardes de domingo
y hasta la línea recta.

Me enervan los que no tienen dudas
y aquellos que se aferran
a sus ideales sobre los de cualquiera.

Me cansa tanto tráfico
y tanto sinsentido,
parado frente al mar mientras que el mundo gira.

Francisco M. Ortega Palomares

Capítulo 1

Formalismos

Resumen:

1.1. Introducción a la Teoría de Conjuntos	5
1.2. Relaciones	11
1.3. Funciones	15
1.4. Álgebra de Boole	19
1.5. Nociones sobre Grafos	21
Notas	33

1.1. Introducción a la Teoría de Conjuntos

1.1.1. Definiciones

Definición 1.1.1. Se conoce por Conjunto ¹ a una estructura finita de elementos que guardan una relación entre si.

Definición 1.1.2. Los elementos que componen un conjunto reciben también el nombre de objetos.

Corolario 1.1.3. *Existen dos métodos para describir un conjunto: conjuntos por extensión y conjuntos por compresión.*

- Conjunto por Extensión: Se dice que un conjunto está descrito por extensión cuando todos los elementos que lo componen se puede enumerar. Normalmente se denotan los elementos entre corchetes:

Ejemplo 1.1.4. El conjunto O , contiene los dígitos de 0 a 9:

$$O = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \quad (1.1)$$

- Conjunto por Compresión: Se dice que un conjunto está descrito por compresión cuando sus elementos se describen a través de una propiedad.

Ejemplo 1.1.5. El conjunto P , contiene los número pares de 0 a 9:

$$P = \{x \mid (x \% 2 = 0) \wedge (x \geq 0 \ \& \ x < 10)\} \quad (1.2)$$

Definición 1.1.6. Dos conjuntos son iguales si y sólo si contienen los mismos elementos (incluyendo los repetidos).

Ejemplo 1.1.7. Son iguales los siguientes conjuntos:

$$A = B \Rightarrow \{0, 1, 2, 3, 4, 5\} = \{0, 0, 4, 4, 4, 5, 3, 3, 2, 1, 1, 1\} \quad (1.3)$$

Definición 1.1.8. Se dice que un conjunto O es un subconjunto de P , si todos los elementos de O forman parte de P .

Ejemplo 1.1.9. Para los conjuntos: $O = \{1, 2, 3, 5, 7, 9\}$ y $P = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ decimos:

$$O \subseteq P \quad (1.4)$$

Definición 1.1.10. Se denomina **Conjunto Universal** al conjunto origen a partir del cual derivan otros conjuntos. Se denota como U .

Ejemplo 1.1.11. El conjunto universal U contiene a todos los números naturales:

$$U = \{1, 2, \dots, n, \dots\} \quad (1.5)$$

Luego P es subconjunto de U porque P se describe como el **conjunto de los números naturales primos**:

$$P = \{1, 2, 3, 5, \dots, n, \dots\} \subseteq U \quad (1.6)$$

Definición 1.1.12. Se denomina **Conjunto Vacío** al conjunto que no contiene ningún elemento. Se denota como: \emptyset .

Ejemplo 1.1.13. Se puede decir formalmente que el conjunto vacío:

$$\emptyset \equiv \{ \} \equiv \{\emptyset\} \quad (1.7)$$

1.1.2. Operaciones

I. Unión:

Definición 1.1.14. Dados los conjuntos O y P se tiene por **Unión** de ambos (denotado mediante el signo \cup) $O \cup P$, a otro conjunto que contiene los elementos de: O y P y ambos.

Ejemplo 1.1.15. Sea $O = \{v, o, c, a, l, e, s\}$ y $P = \{a, e, i, o, u\}$. Se tiene:

$$O \cup P = \{a, e, i, o, u, v, c, l, s\} \quad (1.8)$$

Propiedades:

i. Propiedad Conmutativa:

$$O \cup P \equiv P \cup O = \{a, e, i, o, u, v, c, l, s\} \quad (1.9)$$

ii. Propiedad Asociativa: Para $Q = \{a, b, c, d\}$

$$(O \cup P) \cup Q \equiv O \cup (P \cup Q) = \{a, e, i, o, u, v, c, l, s, b, d\} \quad (1.10)$$

iii. Propiedad de Absorción:

$$O \cup U = U \quad (1.11)$$

iv. Propiedad de Idempotencia:

$$O \cup O \equiv O = \{v, o, c, a, l, e, s\} \quad (1.12)$$

v. Propiedad de Neutralidad:

$$O \cup \emptyset \equiv O = \{v, o, c, a, l, e, s\} \quad (1.13)$$

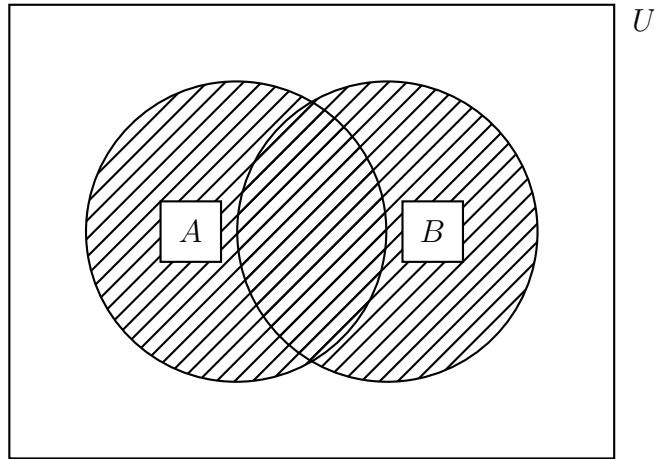


Figura 1.1: Relación de Unión.

II. Intersección:

Definición 1.1.16. Dados los conjuntos O y P se tiene por **Intersección** de ambos (denotado mediante el signo \cap) $O \cap P$, a otro conjunto que contiene los elementos comunes a O y P .

Ejemplo 1.1.17. Sea $O = \{v, o, c, a, l, e, s\}$ y $P = \{a, e, i, o, u\}$. Se tiene:

$$O \cap P = \{o, a, e\} \quad (1.14)$$

Propiedades:

i Propiedad Conmutativa:

$$O \cap P \equiv P \cap O = \{o, a, e\} \quad (1.15)$$

ii Propiedad Asociativa: $Q = \{a, b, c, d\}$

$$(O \cap P) \cap Q \equiv O \cap (P \cap Q) = \{a\} \quad (1.16)$$

iii Propiedad de Absorción:

$$\emptyset \cap O \equiv \emptyset = \emptyset \quad (1.17)$$

iv Propiedad de Idempotencia:

$$O \cap O \equiv O = \{v, o, c, a, l, e, s\} \quad (1.18)$$

v Propiedad de Neutralidad:

$$O \cap U \equiv O = \{v, o, c, a, l, e, s\} \quad (1.19)$$

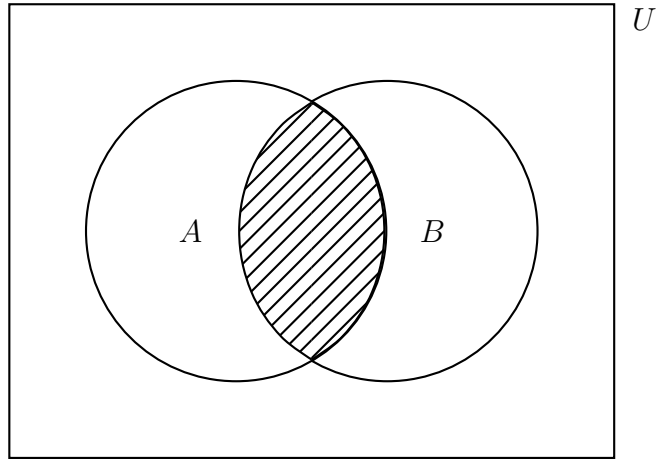


Figura 1.2: Relación de Intersección.

III. Leyes de De Morgan: Nos permiten establecer equivalencias entre los operadores antes vistos (Unión e Intersección)

Definición 1.1.18. Primera Ley:

$$\overline{O \cup P} = \bar{O} \cap \bar{P} \quad (1.20)$$

$$\overline{O \cup P} = \{v, c, l, s\} \equiv \bar{O} \cap \bar{P} = \{v, c, l, s\} \quad (1.21)$$

Definición 1.1.19. Segunda Ley:

$$\overline{O \cap P} = \bar{O} \cup \bar{P} \quad (1.22)$$

IV. Resta de Conjuntos:

Definición 1.1.20. Dados los conjuntos O y P se tiene por **Resta** de ambos (denotado mediante el signo $-$) $O - P$, a aquellos elementos de O que no estén en P

$$O \cap P = \{v, c, l, s\} \quad (1.23)$$

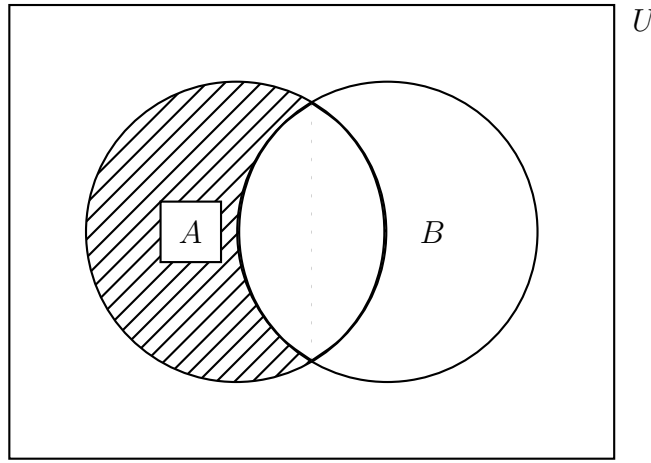


Figura 1.3: Relación de Resta.

V. Disjunción:

Definición 1.1.21. Dos conjuntos son **Disjuntos** cuando su intersección es vacía.

Ejemplo 1.1.22. Dados los conjuntos: $P = \{a, e, i, o, u\}$ y $Q = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$$P \cap Q = \emptyset \quad (1.24)$$

VI. Diferencia Simétrica:

Definición 1.1.23. Dados los conjuntos O y P se entiende por **Diferencia Simétrica**, denotado como \oplus , a todos los elementos que están en O y no en P u todos los elementos que están en P y no están en el conjunto O .

Formalidad 1.1.24. $O \oplus P = (O - P) \cup (P - O)$

Ejemplo 1.1.25. Sea $O = \{a, b, c, d, e, f, g, i\}$ y $P = \{a, e, i, o, u\}$ se tiene:

$$O - P = \{b, c, d, f, g\} \wedge P - O = \{o, u\} \Rightarrow O \oplus P = \{b, c, d, f, g, o, u\} \quad (1.25)$$

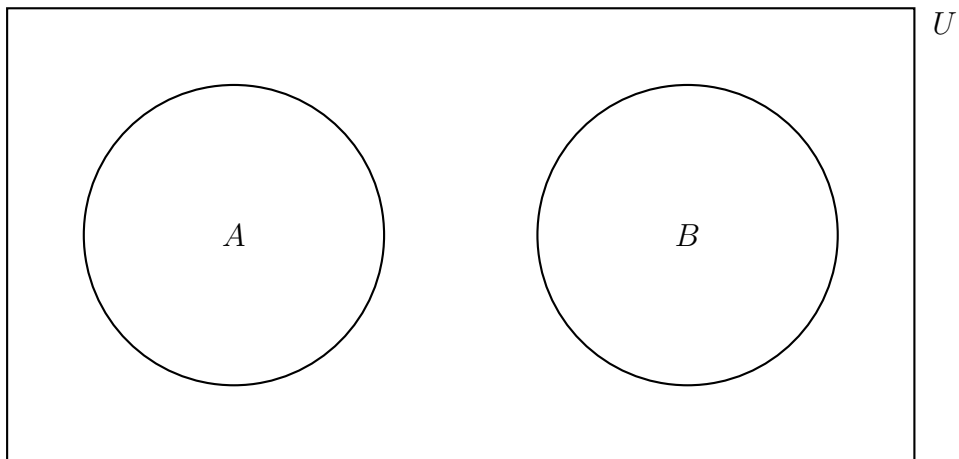


Figura 1.4: Relación de Disjunción.

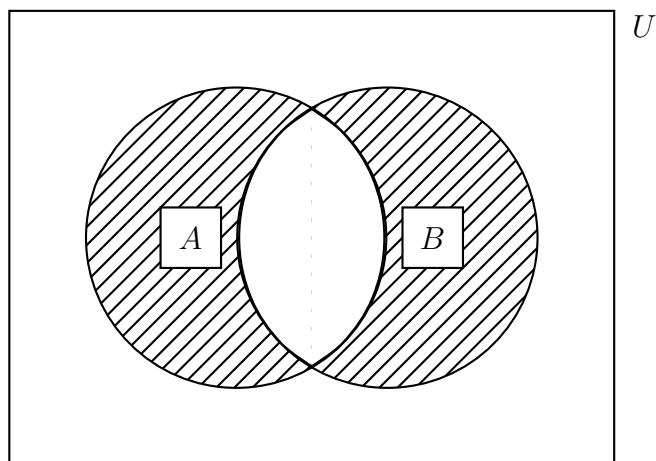


Figura 1.5: Relación de Diferencia Simétrica.

VII. Complemento:

Definición 1.1.26. Dado el conjunto P se tiene por **Complementario** (denotado mediante el signo \bar{P}), a aquellos elementos de U que no están en P .

Formalidad 1.1.27. $\bar{P} = U - P$

Ejemplo 1.1.28. En nuestro caso siendo U el alfabeto castellano y $P = \{a, e, i, o, u\}$ se tiene:

$$\bar{P} = \{b, c, d, \dots, x, y, z\} \quad (1.26)$$

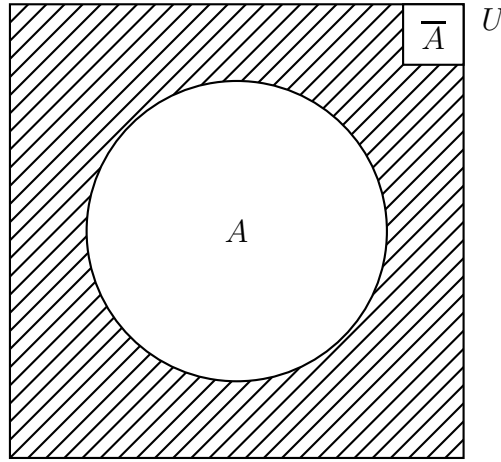


Figura 1.6: Relación de Complemento.

1.2. Relaciones

1.2.1. Definiciones

Definición 1.2.1. Denominamos **Par** a **todo conjunto finito de dos elementos**:

$$P = (a, b) \quad (1.27)$$

de modo que:

- i. a es la **primera coordenada** o primer elemento.
- ii. b de manera análoga, es la **segunda coordenada** o segundo elemento.

Definición 1.2.2. Dos pares: (a, b) y (c, d) **son iguales** si:

$$a \div c \wedge b \div d \quad (1.28)$$

Definición 1.2.3. Un Par es **idéntico** si:

$$a \div b \quad (1.29)$$

Definición 1.2.4. El Par **recíproco** a (a, b) es:

$$(b, a) \quad (1.30)$$

1.2.2. Producto Cartesiano

Definición 1.2.5. Formalmente diremos que el **Producto Cartesiano** para A, B es:

$$A \times B = \{(a, b) \mid (a \in A) \wedge (b \in B)\} \quad (1.31)$$

Ejemplo 1.2.6. Para los conjuntos: $O = \{1, 3, 6\}$ y $P = \{2, 4\}$ tenemos:

$$O \times P = \{(1, 2), (1, 4), (3, 2), (3, 4), (6, 2), (6, 4)\} \quad (1.32)$$

Propiedades:

i. **No Conmutativo:** Dados los conjuntos: $R = (a)$ y $S = (b)$ tenemos:

$$R \times S = \{(a, b) \mid (a \in R) \wedge (b \in S)\} \quad (1.33)$$

Por contra:

$$S \times R = \{(b, a) \mid (b \in S) \wedge (a \in R)\} \quad (1.34)$$

ii. **Asociativo:** Dados los conjuntos: $R = (a)$, $S = (b)$ y $T = (c)$ tenemos:

$$R \times S \times T = (R \times S) \times T = R \times (S \times T) \quad (1.35)$$

iii. **Distributivo:**

$$R \times (S \cap T) = (R \times S) \cap (R \times T) \quad (1.36)$$

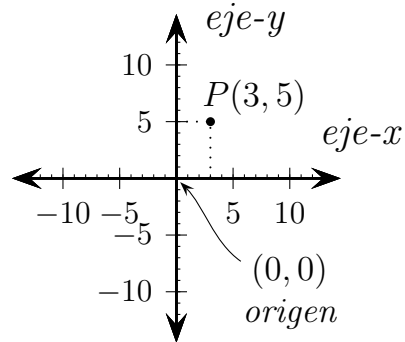
1.2.3. Representaciones

I. **Representación Mediante Tabla:** Para los conjuntos $O = (o_1, o_2, \dots, o_m)$ y $P = (p_1, p_2, \dots, p_n)$, cada elemento de O sería el índice de cada columna y, de manera análoga cada elemento de P constituiría el índice de una fila. **La intersección representa el Par resultado.**

II. **Representación Cartesiana:** Se representa mediante dos ejes. El eje horizontal corresponde al conjunto O y, el eje vertical corresponde al conjunto P . **La intersección de ambos (un Punto) es un Par producto.**

\emptyset	O_1	\dots	O_m
P_1	$O \times P_{11}$	\dots	$O \times P_{1m}$
P_2	$O \times P_{21}$	\dots	$O \times P_{2m}$
\vdots	\dots	\dots	\dots
P_n	$O \times P_{n1}$	\dots	$O \times P_{nm}$

(a) Representación mediante tabla.

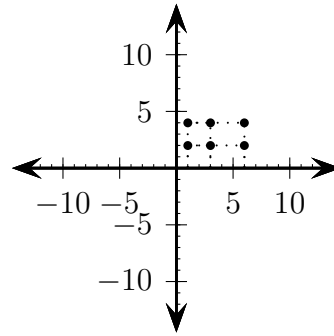


(b) Representación mediante Sistema Cartesiano.

Figura 1.7: Representaciones genéricas del Producto Cartesiano.

\emptyset	1	3	6
2	(1, 2)	(3, 2)	(6, 2)
4	(1, 4)	(3, 4)	(6, 4)

(a) Representación Mediante Tabla.



(b) Representación mediante Sistema Cartesiano.

Figura 1.8: Representaciones del Producto Cartesiano; $O \times P$.

Nota: Para el Ejemplo (1.2.6):

1.2.4. Tipos

I. Relaciones Binarias:

Definición 1.2.7. Para dos conjuntos dados A y B y la relación \mathfrak{R} decimos que: “La Relación Binaria de A hacia B es de la forma”:

$$\mathfrak{R} = \{(a, b) \mid ((a, b) \in A \times B) \wedge (a \mathfrak{R} b)\} \quad (1.37)$$

Ejemplo 1.2.8. Para el conjunto: $O = \{1, 2, 3\}$; $o_i \mathfrak{R} o_j \Leftrightarrow o_i \cdot o_j$ es número par. Por ello tenemos:

$$o_i \mathfrak{R} o_j = \{(1, 2), (2, 1), (2, 2), (3, 2)\} \quad (1.38)$$

Representaciones de las Relaciones Binarias:

Nota: Para el Ejemplo (1.2.8):

- i. Representación Cartesiana: Partiendo de la definición (1.2.3), **la intersección de los conjuntos es un Par de la Relación.**
- ii. Representación Sagital: Partiendo de la definición (1.2.3), **el punto de intersección es un Par de la Relación.**
- iii. Representación Matricial: Se trata de la transcripción directa de la **Representación Cartesiana** a Matriz donde, **cada a_{ij} representa un Par de la Relación.**

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

Figura 1.9: Representación genérica de una Relación Binaria mediante una Matriz.

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

II. Relación Inversa:

Definición 1.2.9. Definimos **Relación Inversa** (denotada como \mathfrak{R}^{-1}) a aquella relación entre pares que establece:

$$\mathfrak{R}^{-1} = \{(a, b) | (b, a) \in \mathfrak{R}\} \quad (1.39)$$

Ejemplo 1.2.10. Para el Ejemplo (1.2.8):

$$o_i \mathfrak{R}^{-1} o_j = \{(2, 1), (1, 2), (2, 2), (2, 3)\} \quad (1.40)$$

III. Relación Complementaria:

Definición 1.2.11. Definimos **Relación Complementaria** (denotada como $\overline{\mathfrak{R}}$) a aquella relación entre pares que establece:

$$\forall a \in A, b \in B; a \overline{\mathfrak{R}} b \Leftrightarrow a \mathfrak{R} b \notin (A \times B) \quad (1.41)$$

Ejemplo 1.2.12. Para el Ejemplo (1.2.8):

$$o_i \overline{\mathfrak{R}} o_j = \{(1, 1), (1, 3), (2, 3), (3, 1), (3, 3)\} \quad (1.42)$$

IV. Relaciones Transitivas:

Definición 1.2.13. Definimos **Relación Transitiva** a aquella que cumple:

$$\forall a \in A, b \in B, c \in C; a \mathfrak{R} b \wedge b \mathfrak{R} c \Rightarrow a \mathfrak{R} c \quad (1.43)$$

Ejemplo 1.2.14. Para el Ejemplo (1.2.8) y el conjunto $P = \{4, 5, 6\} \Rightarrow p_i \mathfrak{R} p_j$ es número par =

$$o_i \mathfrak{R} p_j = \quad (1.44)$$

V. Relación Compuesta:

Definición 1.2.15. Definimos **Relación Compuesta** a aquella relación (en nuestro caso, con tres conjuntos origen) que se establece $\forall a \in A, b \in B, c \in C; A \subseteq B \wedge a \mathfrak{R} b$ y $B \subseteq C \wedge b \mathfrak{S} c$:

$$\mathfrak{R} \circ \mathfrak{S} = \{(a, c) / \exists b \in B \Leftrightarrow a \mathfrak{R} b \wedge b \mathfrak{S} c\} \quad (1.45)$$

Ejemplo 1.2.16. Para el Ejemplo (1.2.8) y la relación en el conjunto P $p_i \mathfrak{S} p_j \Leftrightarrow o_i + o_j \% 3 = 0$

$$\mathfrak{R} \circ \mathfrak{S} = \{(1, 2), (2, 1)\} \quad (1.46)$$

1.3. Funciones

Definición 1.3.1. De manera somera podemos decir que una **Función**² es una regla que transforma un conjunto (**Conjunto Inicial** o *Dominio*) en otro nuevo conjunto (**Conjunto Imagen** o *Recorrido*). Si establecemos el Conjunto Origen como D_1 y el Conjunto Imagen como R_1 tenemos la relación:

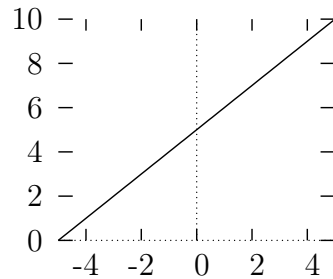
$$f : D_1 \longrightarrow R_1 \quad (1.47)$$

Ejemplo 1.3.2. Tenemos la función $f(x + 5)$ y el Conjunto Origen $U = \{0, 1, 2, 3, 4, 5\}$ por lo que:

$$f(O + 5) = \{5, 6, 7, 8, 9, 10\} \quad (1.48)$$

<i>Dominio</i>	<i>Recorrido</i>
0	5
1	6
2	7
3	8
4	9
5	10

(a) Representación mediante tabla.



(b) Representación gráfica.

Figura 1.10: Representaciones para la Función: $f(x + 5)$.

Definición 1.3.3. Formalmente **una función para una variable** (tomaremos x por convención) que pertenece al conjunto $\text{Dominio}(x)$ le corresponden uno o varios valores en y que a su vez pertenece al conjunto $\text{Recorrido}(x)$

$$y = f(x) \quad (1.49)$$

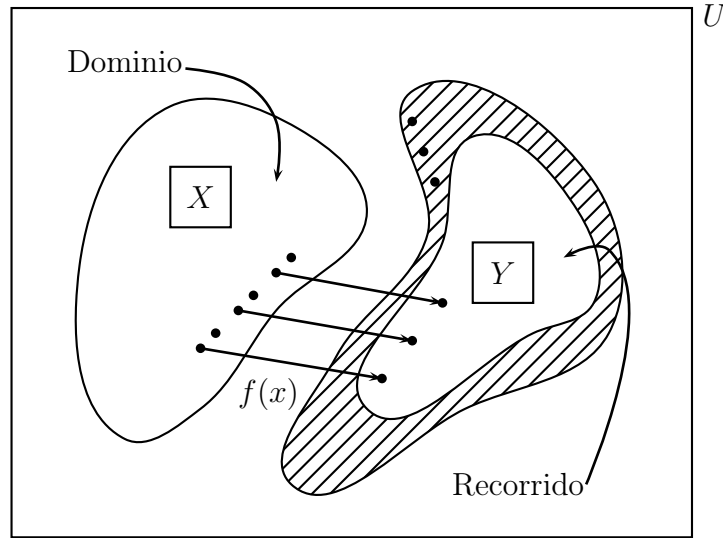


Figura 1.11: Relaciones entre los principales elementos de una Función.

1.3.1. Propiedades

Para la relación: $f : D_1 \longrightarrow R_1$ tenemos la siguientes propiedades:

- I. **Representación Gráfica:** el conjunto D_1 es un subconjunto del Producto Cartesiano $D_1 \times R_1$
- II. **Imagen:** Establecemos que la Imagen de X como X' por lo que:

$$X' \subset X : f(X') = \{f(x') \mid x' \in X'\} \quad (1.50)$$

- III. **Imagen Recíproca:** Es la función inversa del Conjunto Imagen es decir:

$$f^{-1} : y \in Y = \{x \in X \mid f(x) = y\} \quad (1.51)$$

- IV. **Restricción** de f sobre $U \subset X$:

$$f : U \longrightarrow Y \mid \{u_i \in U, u_i \in X, y \in Y\} \quad (1.52)$$

1.3.2. Tipos

Se conocen tres tipos de funciones dada por la relación entre los valores del Conjunto Inicial y los valores del Conjunto Imagen: $f : X \longrightarrow Y$

- I. **Funciones Exhaustivas o Suprayectiva:** Una **Función es Exhaustiva** si para cada elemento de X existe al menos un elemento en Y

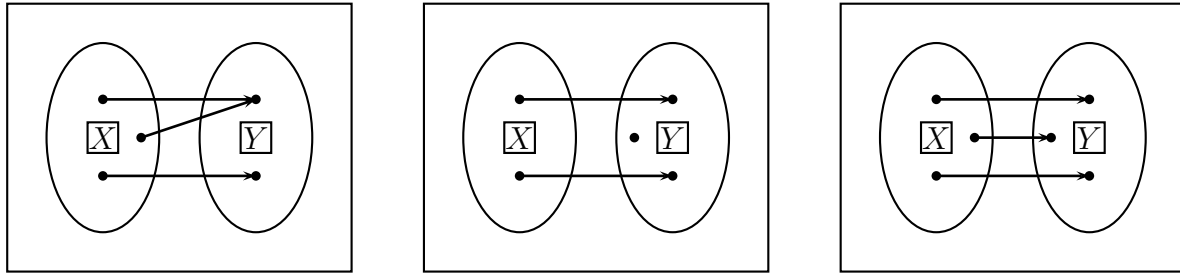
$$\forall y \in Y \exists x \in X \mid f(x) = y \quad (1.53)$$

II. **Funciones Inyectivas:** Una **Función es Inyectiva** si para cada elemento de Y existe como máximo un elemento en X

$$\forall x \in X \exists y \in Y \text{ } \not\! / \text{ } f(x) = y \quad (1.54)$$

III. **Funciones Biyectivas:** Una **Función es Biyectiva** si para cada elemento de X existe un único elemento de Y

$$\exists x \in X, \exists y \in Y \text{ } \not\! / \text{ } f(x) = y \quad (1.55)$$



(a) Función Suprayectiva α

(b) Función Inyectiva β

(c) Función Biyectiva γ

Figura 1.12: Tipos de funciones basadas en la relación de Dominio y Recorrido.

1.3.3. Operaciones

Nota: Usaremos las funciones genéricas: $F(x) = (f_1, f_2x, \dots, f_nx^{n-1})$ y $G(x) = (g_1, g_2x, \dots, g_nx^{n-1})$ con $\{n \in \mathbb{N}\}$. A modo de ejemplos tendremos las funciones: $U(x) = \frac{3x+5}{2}$ y $V(x) = 4x^2 - 1$.

I. **Suma:**

$$F(x) + G(x) = (f_1 + g_1, f_2x + g_2x, \dots, f_nx^{n-1} + g_nx^{n-1}) \text{ } \not\! / \text{ } \{n \in \mathbb{N}\} \quad (1.56)$$

Ejemplo 1.3.4.

$$U(x) + V(x) = \frac{8x^2 + 3x + 3}{2} \quad (1.57)$$

II. **Resta:**

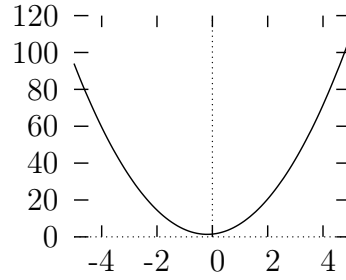
$$F(x) - G(x) = (f_1 - g_1, f_2x - g_2x, \dots, f_nx^{n-1} - g_nx^{n-1}) \text{ } \not\! / \text{ } \{n \in \mathbb{N}\} \quad (1.58)$$

Ejemplo 1.3.5.

$$U(x) - V(x) = \frac{-8x^2 + 3x + 7}{2} \quad (1.59)$$

<i>Dominio</i>	<i>Recorrido</i>
0	$\frac{3}{2}$
1	7
\vdots	\vdots
n	$\frac{8n^2+3n+3}{2}$

(a) Representación mediante tabla.

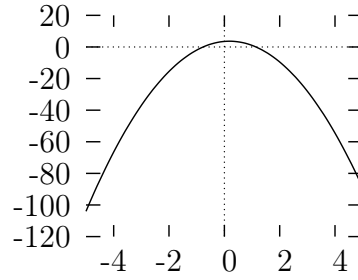


(b) Representación gráfica.

Figura 1.13: Representaciones para la Función: $\frac{8x^2+3x+3}{2}$.

<i>Dominio</i>	<i>Recorrido</i>
0	$\frac{3}{2}$
1	1
\vdots	\vdots
n	$\frac{-8n^2+3n+7}{2}$

(a) Representación mediante tabla.



(b) Representación gráfica.

Figura 1.14: Representaciones para la Función: $\frac{-8x^2+3x+7}{2}$.

III. Producto:

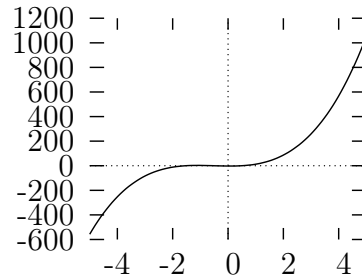
$$F(x) \cdot G(x) = \sum_{i=1}^{i=n} \prod_{j=1}^{j=n} f_i \cdot g_j \quad / \quad \{i, j \leq n\} \text{ e } \{i, j \in \mathbb{N}\} \quad (1.60)$$

Ejemplo 1.3.6.

$$U(x) \cdot V(x) = \frac{13x^3 + 20x^2 - 3x - 5}{2} \quad (1.61)$$

<i>Dominio</i>	<i>Recorrido</i>
0	$-\frac{5}{2}$
1	$\frac{25}{2}$
\vdots	\vdots
n	$\frac{13n^3+20n^2-3n-5}{2}$

(a) Representación mediante tabla.



(b) Representación gráfica.

Figura 1.15: Representaciones para la Función: $\frac{13x^3+20x^2-3x-5}{2}$.

IV. División:

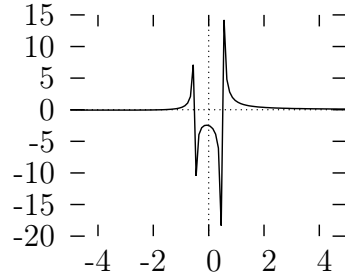
$$\frac{F(x)}{G(x)} = \frac{(f_1, f_2x, \dots, f_nx^{n-1})}{(g_1, g_2x, \dots, g_nx^{n-1})} = \frac{f_1}{g_1} + \frac{f_2x}{g_2x} + \dots, \frac{f_nx^{n-1}}{g_nx^{n-1}} \setminus \{n \in \mathbb{N}\} \quad (1.62)$$

Ejemplo 1.3.7.

$$\frac{U(x)}{V(x)} = \frac{3x+5}{8x^2-2} \quad (1.63)$$

<i>Dominio</i>	<i>Recorrido</i>
0	\emptyset
1	\emptyset
\vdots	\vdots
n	$\frac{3n+5}{8n^2-2}$

(a) Representación mediante tabla.



(b) Representación gráfica.

Figura 1.16: Representaciones para la Función: $\frac{3x+5}{8x^2-2}$.

V. Composición:

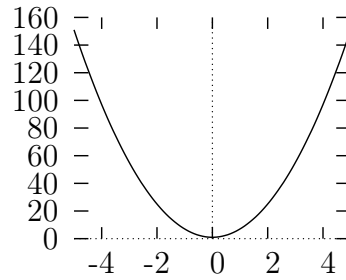
$$F(x) \circ G(x) = F(G(x)) = (f_1, f_2(g(x)), \dots, f_n(g(x))) \setminus \{n \in \mathbb{N}\} \quad (1.64)$$

Ejemplo 1.3.8.

$$U(x) \circ V(x) = U(V(x)) = \frac{3(4x^2-1)+}{2} = \frac{12x^2+2}{2} = 6x^2+1 \quad (1.65)$$

<i>Dominio</i>	<i>Recorrido</i>
0	1
1	7
\vdots	\vdots
n	$6n^2+1$

(a) Representación mediante tabla.



(b) Representación gráfica.

Figura 1.17: Representaciones para la función: $6x^2+1$.

1.4. Álgebra de Boole

1.4.1. Generalidades

Definición 1.4.1. Un **Álgebra de Boole** se define como una tupla de cuatro elementos (también denomina retícula booleana):

$$(\mathfrak{B}, \sim, \oplus, \odot) \quad (1.66)$$

Dónde:

- i. \mathfrak{B} : Se trata del **Conjunto de Variables Booleanas**.
- ii. \sim : Se trata de una **operación interna unitaria** ($\mathfrak{B} \rightarrow \mathfrak{B}$) que cumple:

$$a \rightarrow b = \sim a \text{ } / \text{ } \{a, b \in \mathfrak{B}\} \quad (1.67)$$

- iii. \oplus : Se trata de una **operación binaria interna** ($\mathfrak{B} \times \mathfrak{B} \rightarrow \mathfrak{B}$:) que cumple:

$$(a, b) \rightarrow c = a \oplus b \text{ } / \text{ } \{a, b, c \in \mathfrak{B}\} \quad (1.68)$$

- iv. \odot : Se trata de una **operación binaria interna** ($\mathfrak{B} \times \mathfrak{B} \rightarrow \mathfrak{B}$:) que cumple:

$$(a, b) \rightarrow c = a \odot b \text{ } / \text{ } \{a, b, c \in \mathfrak{B}\} \quad (1.69)$$

Siendo las condiciones necesarias:

- i. $a \oplus b = b$
- ii. $a \odot b = a$
- iii. $\sim a \oplus b = U$
- iv. $a \odot \sim b = \emptyset$

Definición 1.4.2. Se establece una **relación directa** entre el **Álgebra de Boole** y la **Lógica Binaria** de manera que:

$$(\mathfrak{B}, \sim, \oplus, \odot) \equiv (\{0, 1\}, \bar{\cdot}, +, \cdot) \quad (1.70)$$

1.4.2. Lógica Binaria

Definición 1.4.3. Decimos que x, y son **Variables Booleanas Binarias** si:

$$x, y \in (\{0, 1\}, \bar{\cdot}, +, \cdot) \quad (1.71)$$

por lo que cumplen:

- I. **Operación Complemento** también denominada Operación NOT (ver Figura 1.18):
- II. **Operación de Suma** también denominada Operación OR (ver Figura 1.19):
- III. **Operación de Producto** también denominada Operación AND (ver Figura 1.20):
- IV. **Operación de Suma Exclusiva** también denominada Operación XOR (ver Figura 1.21):

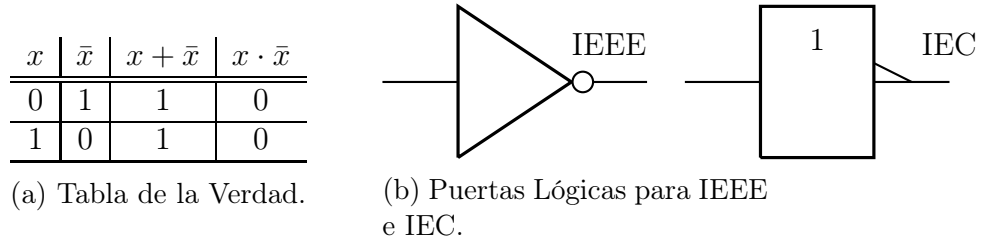


Figura 1.18: Representaciones comunes del Operador Booleano NOT.

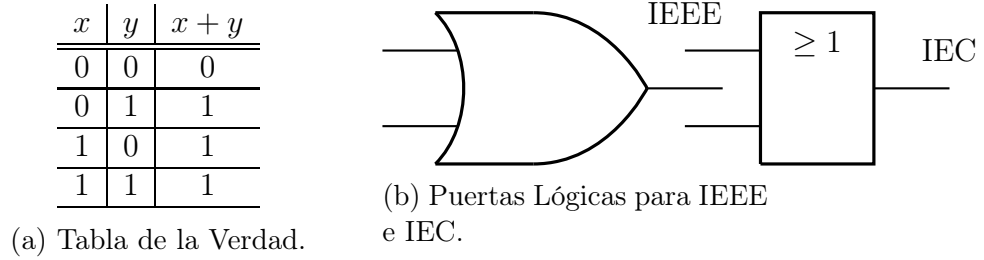


Figura 1.19: Representaciones comunes del Operador Booleano OR.

1.4.3. Funciones Booleanas

Definición 1.4.4. Decimos que O es una **Función Booleana**:

$$O = (u_1, u_2, \dots, u_n) \Rightarrow u_i \in (\mathfrak{B}, \sim, \oplus, \odot) \quad (1.72)$$

de igual manera decimos que O en una **Función Booleana Binaria** si:

$$O = (u_1, u_2, \dots, u_n) \Rightarrow u_i \in (\{0, 1\}, -, +, \cdot) \quad (1.73)$$

Para el Álgebra de Boole tenemos dos operaciones fundamentales:

I. **Operación de Suma** de Funciones Booleanas Binarias para O y P :

$$O + P = (u_1, u_2, \dots, u_n) + (v_1, v_2, \dots, v_n) = (u_1 + v_1, u_2 + v_2, \dots, u_n + v_n) \quad (1.74)$$

II. **Operación de Producto** de Funciones Booleanas Binarias sobre O y P :

$$U \cdot V = (u_1, u_2, \dots, u_n) \cdot (v_1, v_2, \dots, v_n) = (u_1 \cdot v_1, u_2 \cdot v_2, \dots, u_n \cdot v_n) \quad (1.75)$$

1.5. Nociones sobre Grafos

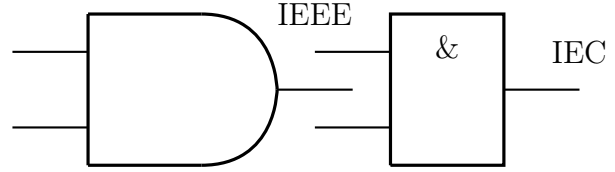
1.5.1. Definiciones

Definición 1.5.1. Un Grafo G esta compuesto por tres conjuntos finitos y necesariamente uno de ellos no vacío:

- i. Conjunto V : El Conjunto de sus **Vértices** (no puede ser vacío).

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

(a) Tabla de la Verdad.

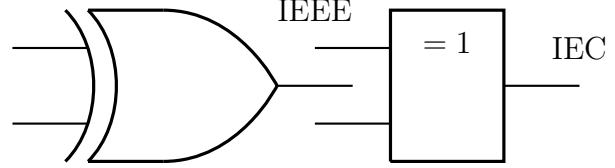


(b) Puertas Lógicas para IEEE e IEC.

Figura 1.20: Representaciones comunes del Operador Booleano AND.

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

(a) Tabla de la Verdad.



(b) Puertas Lógicas para IEEE e IEC.

Figura 1.21: Representaciones comunes del Operador Booleano XOR.

ii. Conjunto E : El Conjunto de sus **Aristas**.

$$V \times V \rightarrow E \quad (1.76)$$

iii. Conjunto p : El Conjunto de los **Pesos** o **Etiquetas** por aristas.

$$p : E \quad (1.77)$$

Por ello establecemos la siguiente notación para describir un Grafo:

$$G = (V, E, p) \quad (1.78)$$

Definición 1.5.2. Para una arista dada: $a_\lambda = (v_1, v_2)$ decimos que:

- i. v_1 es el **origen**.
- ii. v_2 es el **destino** o **final**.

Ejemplo 1.5.3. Dado el siguiente grafo G_3 (ver Figura 1.22):

- i. $V = \{a, b, c\}$
- ii. $E = \{\{a, b\}, \{b, c\}, \{c, a\}\}$

Definición 1.5.4. Decimos que una arista a_λ es **incidente** para dos vértices v_1, v_2 si une dichos vértices:

$$a_\lambda = (v_1, v_2) \text{ } \nearrow \text{ } v_1, v_2 \in V, \text{ } a_\lambda \in E \quad (1.79)$$

Definición 1.5.5. Un vértice v_1 es **adyacente** sobre otros vértices v_λ si dicho vértice forma parte de la relación:

$$a_\lambda = (v_1, v_\lambda) \text{ } / \text{ } v_\lambda \in V, \text{ } a_\lambda \in E \quad (1.80)$$

Definición 1.5.6. Una arista del tipo $a_1 = (v_1, v_1)$ se denomina **bucle** puesto que el vértice origen y destino son el mismo.

Definición 1.5.7. Si (a_1, a_2) inciden sobre el mismo vértice, se dice que son **aristas paralelas**.

Definición 1.5.8. Un vértice v_μ es un **vértice aislado** si para el conjunto E no existe ningún par o relación.

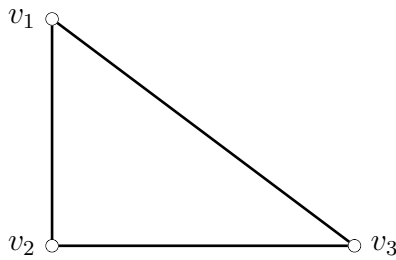
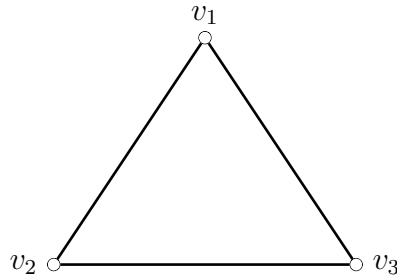
(a) Grafo G_1 .(b) Grafo G_2 .(c) Grafo G_3 .

Figura 1.22: Ejemplos de Grafos.

1.5.2. Clasificación

Definición 1.5.9. Un grafo $G = (V, E, p = \emptyset)$ que contiene más de un par de aristas para uno de sus vértices en un **Grafo Multigrafo**.

Definición 1.5.10. Un grafo $G = (V, E, p = \emptyset)$ que contiene al menos un bucle y ningún conjunto de aristas paralelas es lo que convencionalmente denominamos **Grafo**.

Definición 1.5.11. Para un grafo $G = (V, E, p)$, si $p = \emptyset$ y no existen bucles, se dice que es un **Grafo Simple**.

Corolario 1.5.12. Si $p = \emptyset$ y existen bucles, el grafo se denomina **Grafo no Simple**.

Definición 1.5.13. Para un grafo G de tipo simple, si $p \neq \emptyset$ entonces se denomina **Grafo Dirigido**.

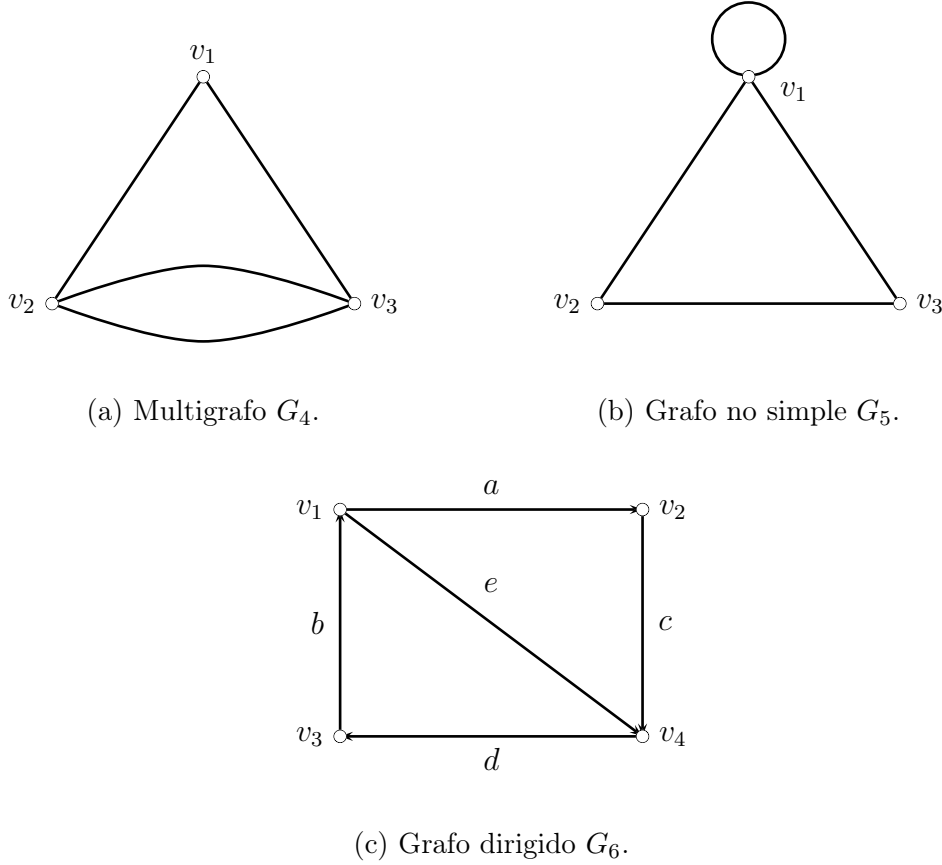


Figura 1.23: Ejemplo de Multigrafo y Grafo no simple y Grafo dirigido.

Definición 1.5.14. Dos grafos G_1 y G_2 son **Isomorfos** si existe una **Bijección** entre ellos α .

$$V_{G_1} = \{a, b, c\} \equiv \alpha V_{G_1} = V_{G_2} = \{\alpha(a) = d, \alpha(b) = e, \alpha(c) = f\} \quad (1.81)$$

Ejemplo 1.5.15. Para $G_9 \Rightarrow V(G_9) = \alpha \{v_1, v_2, v_3, v_4\} \in V(G_8) = \{\alpha(v_1) = v'_1, \alpha(v_2) = v'_2, \alpha(v_3) = v'_3, \alpha(v_4) = v'_4\}$

Definición 1.5.16. Denominamos grado de un vértice v_λ para un grafo $G = (V, E, p)$ al numero de aristas del grafo G en dicho vértice.

$$\delta(v) \text{ } / \text{ } v \in V = \sum e_i \text{ } / \text{ } \{e \in E, v \in e\} \quad (1.82)$$

Ejemplo 1.5.17. Para $G_4 \Rightarrow \delta(v_1) = 2; \delta(v_2) = 2; \delta(v_3) = 2;$

Teorema 1.5.18. La suma de los grados de los vértices de un **grafo no dirigido** es igual al doble del número de aristas.

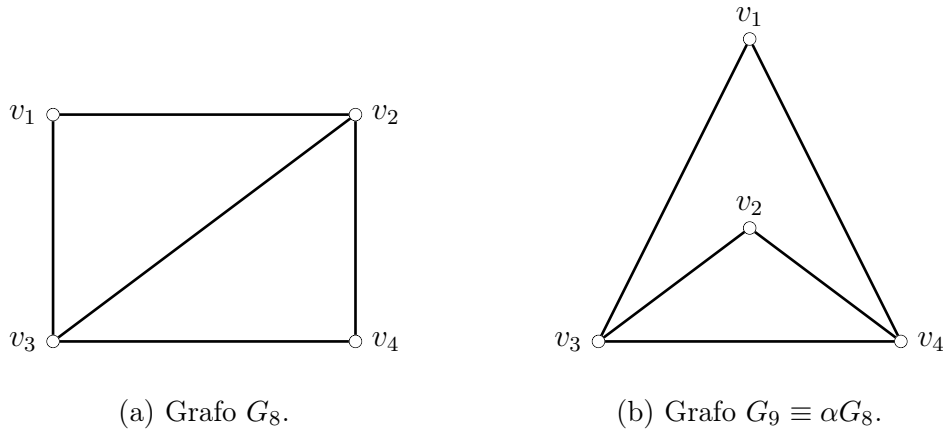


Figura 1.24: Ejemplo de Grafos Isomorfos.

$$\sum_{i=0}^{i=n} \delta(v_i) = 2 \cdot |E| \quad (1.83)$$

Ejemplo 1.5.19. Para $G_5 \Rightarrow \delta(v_1) = 3; \delta(v_2) = 2; \delta(v_3) = 2; \delta(v_4) = 3; \equiv 2 \cdot |E| = 2 \cdot 4 = 8$

Teorema 1.5.20. Para un **grafo dirigido** $G = (V, E, p)$ se cumple:

$$\sum_{i=0}^{i=n} \delta(v_i)^+ = \sum_{j=0}^{j=n} \delta(v_j)^- = |E| \quad (1.84)$$

Dónde:

- i. $\delta(v)^+$: Es el número de aristas que se dirigen a v .
- ii. $\delta(v)^-$: Es el número de aristas que parten de v .

Ejemplo 1.5.21. Para G_6 :

- i. $\delta^+ \Rightarrow \delta(v_1)^+ = 2; \delta(v_2)^+ = 1; \delta(v_3)^+ = 1; \delta(v_4)^+ = 1$
- ii. $\delta^+ \Rightarrow \delta(v_1)^- = 1; \delta(v_2)^- = 1; \delta(v_3)^- = 1; \delta(v_4)^- = 2$
- iii. $E = 5$

1.5.3. Tipos

Definición 1.5.22. Se denomina **Grafo Completo** a aquel grafo simple de n vértices que tiene una sola arista entre cada par de vértices. Se denotan como K_n .

Definición 1.5.23. Se denomina **Grafo Regular** a aquel que tiene en mismo grado en todos sus vértices.

Definición 1.5.24. Se dice que un grafo es **Bipartito** si su número de vértices se pueden dividir en dos conjuntos $G = G_1 \cup G_2$ disjuntos $G_1 \cap G_2 = \emptyset$.

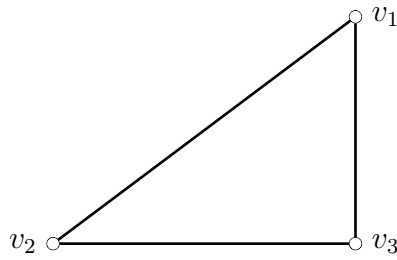
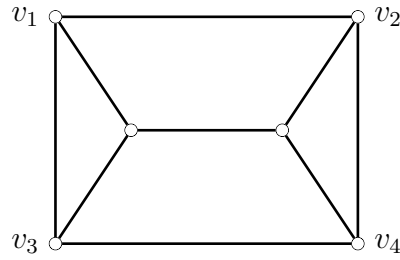
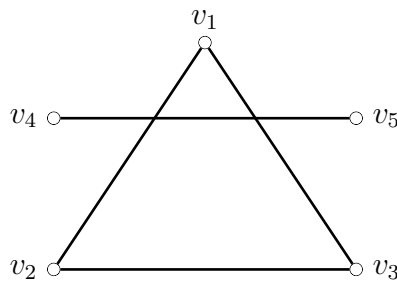
(a) Grafo Completo G_{10} .(b) Grafo Regular G_{11} .(c) Grafo Bipartito G_{12} .

Figura 1.25: Ejemplo de Grafos: Completo, Regular y Bipartito.

1.5.4. Circuitos y Ciclos

I. Recorrido y Circuito Eulero:

Definición 1.5.25. Un grafo $G = (V, E, p)$ (o multigrafo sin vértices asilados) contiene un camino simple (**Camino Simple de Euler o Recorrido Eulero**) que parte de v_0 hasta v_n y que pasa una sola vez por cada uno de los vértices.

Definición 1.5.26. Recibe el nombre de Circuito de Euler a todo camino que pase una sola vez por todos los lados de un grafo G .

Corolario 1.5.27. Si un grafo $G = (V, E)$ tiene un Circuito de Euler³, es un grafo Eulero.

Teorema 1.5.28. El Teorema de Euler dice que para un grafo $G = (V, E, p)$ o multigrafo (no digrafo) sin vértices aislados, G posee un Circuito de Euler si y sólo si G es conexo y cada vértice tiene grado par.

II. Recorrido y Ciclo Hamiltoniano:

Definición 1.5.29. Para un grafo $G = (V, E, p)$ con $|V| \geq 3$ sin vértices aislados. G tiene un **Camino Hamiltoniano** natural que recorre todos sus vértices.

Definición 1.5.30. Un grafo $G = (V, E, p)$ tiene un **Ciclo Hamiltoniano**⁴ si existe un ciclo para todos los vértices de V .

Corolario 1.5.31. *Si un grafo tiene un Ciclo Hamiltoniano se dice que es un grafo hamiltoniano.*

Teorema 1.5.32. *Si un grafo $G = (V, E, p)$ tiene $|V| \geq 3$ y $\delta(v_i) \geq 2$, entonces G es hamiltoniano.*

1.5.5. Árboles

1.5.5.1. Generalidades

Definición 1.5.33. Un Árbol se define como un grafo conectado sin ciclos.

Teorema 1.5.34. *Dado un grafo $T = (V, E)$ decimos que se trata de un árbol si:*

- i. T es un grafo acíclico.
- ii. T está tiene un número de vértices n y de arista que las interconectan $(n - 1)$.
- iii. Cada par de vértices está conectado únicamente por una arista.

Corolario 1.5.35. *Si para un árbol T eliminamos una arista de un par de vértices (u, v) el grafo resultante T' no tiene estructura de árbol.*

1.5.5.2. Árboles Generadores

Definición 1.5.36. Para una grafo simple $G = (V, E, p)$, existe un Árbol Generador T si y sólo si: $T(E) = G(E)$

Corolario 1.5.37. *Un Árbol Generador Mínimo de un Grafo Ponderado es un árbol en el que la suma de sus aristas es la mínima posible.*

1.5.5.2.1. Algoritmo de Prim

Programa 1.5.38. Pseudocódigo del Algoritmo de Prim:

```

1 FUNCTION Prim( $L[1..n], [1..n]$ ):
2    $T = \phi$ ;
3   FOR  $i \leftarrow 2$  TO  $n$  DO
4      $\text{maxProx}[i] \leftarrow 1$ 
5      $\text{absoluteMin}[i] \leftarrow L[i, 1]$ 
6   WHILE  $n - 1$  DO
7      $\text{min} \leftarrow \infty$ 
8     FOR  $j \leftarrow 2$  TO  $n$  DO
9       IF  $0 \leq \text{absoluteMin}[j]$  THEN  $\text{min} \leftarrow \text{absoluteMin}[j]$ 
10       $k \leftarrow j$ 
11     $T \leftarrow T \cup \text{maxProx}[k]$ 
12     $\text{absoluteMin}[k] \leftarrow -1$ 
13    FOR  $j \leftarrow 2$  TO  $n$  DO
14      IF  $L[j, k] < \text{absoluteMin}[j]$  THEN  $\text{absoluteMin}[k] \leftarrow L[j, k]$ 
15       $\text{maxProx}[j] \leftarrow k$ 
16  RETURN  $T$ 

```

Algoritmo 1.5.39. Para un grafo $G = (V, E, p)$

- i. Seleccionar un vértice v_0 aleatorio de: $G(E)$
- ii. Establecer para v_0 las aristas que lo conectan. Si existen dos aristas con idéntico peso, seleccionar cualquiera de ellas.
- iii. Seleccionar la nueva arista con peso mínimo.
- iv. Añadir el vértice y el lado que lo interconecta al conjunto T como resultado.
- v. Iterar pasos *ii*...*iv*.

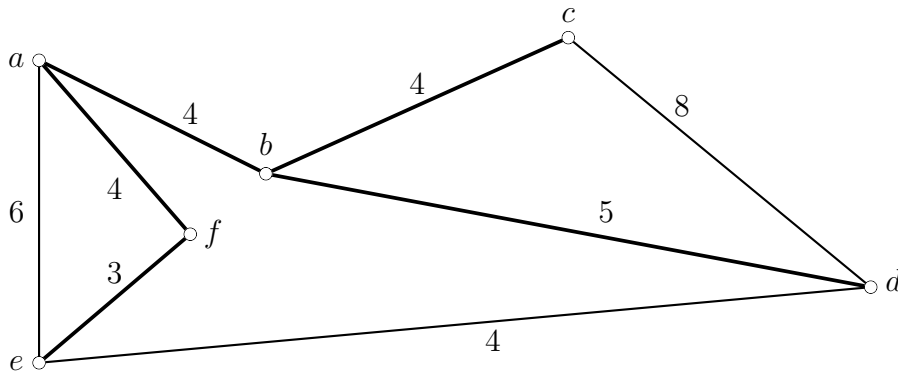


Figura 1.26: Grafo origen para Algoritmo de Prim.

Dónde:

- i. $V = \{a, b, c, d, e, f\}$
- ii. $E = \{ab, bc, cd, de, db, ef, af\}$

Ejemplo 1.5.40. Aplicar el Algoritmo de Prim al siguiente Grafo y encontrar su Árbol Recubridor Mínimo (Figura 1.26)

- i. Seleccionamos el vértice $\{d\}$ como origen.
- ii. Calculamos sobre los pesos de las aristas conexas: $\{dc = 8, de = 7, db = 5\}$
- iii. Tomamos como vértice de resultado b ; $T(E) = \{d, b\}$
- iv. Continuamos iterando para obtener el Árbol Recubrido Mínimo: $T(E) = \{d, b, c, a, f, e\}$

1.5.5.2.2. Algoritmo de Kruskal

Programa 1.5.41. Pseudocódigo del Algoritmo de Kruskal:

```

1 FUNCTION Krukal( $G(V, E)$ ):
2    $n \leftarrow \text{numNodes}$ ;
3    $T = \phi$ ;
4   DO
5      $e \leftarrow \{u, v\}$ 
6      $\text{nodeU} \leftarrow \text{find}(u)$ 
7      $\text{nodeV} \leftarrow \text{find}(v)$ 
8     IF ( $\text{nodeU} \neq \text{nodeV}$ ) THEN
9        $\text{union}(\text{nodeU}, \text{nodeV})$ 
10       $T \leftarrow T \cup \{e\}$ 
11  WHILE  $T = n - 1$ 
12  RETURN  $T$ 

```

Algoritmo 1.5.42. Para un grafo $G = (V, E, p)$

- i. Clasificar las aristas de: $G(E)$ en orden creciente.
- ii. Añadir a $T(E)$ cualquiera de los los lados de G con menor peso y que no formen ciclo con otros lados.
- iii. Iterar el paso *ii* desde: $i = 1$ hasta $G(E) - 1$.

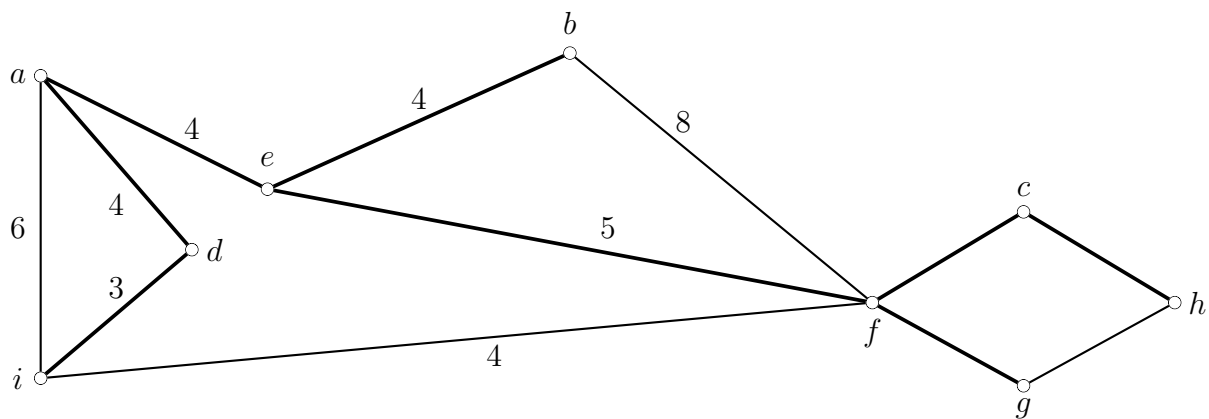


Figura 1.27: Grafo origen para Algoritmo de Kruskal.

Dónde:

- i. $V = \{a, b, c, d, e, f, \dots\}$
- ii. $E = \{ab, bc, cd, de, db, ef, af, \dots\}$

Ejemplo 1.5.43. Aplicar el Algoritmo de Kruskal al siguiente Grafo y encontrar su Árbol Recubridor Mínimo (Figura 1.27)

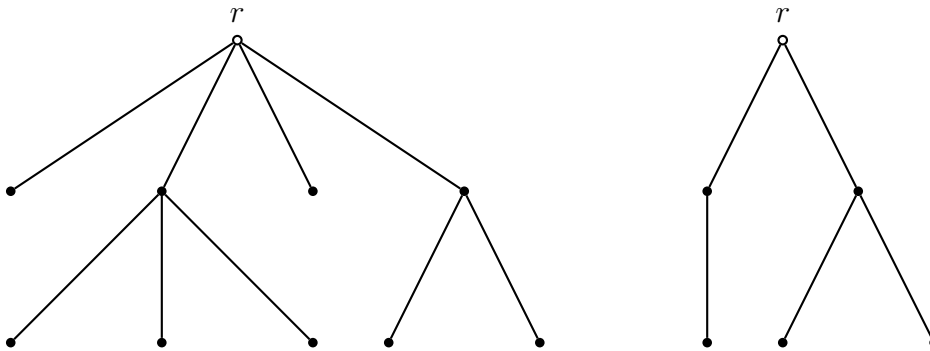
Lados	ai	ad	fg	ae	eb	fc	ch	ef	ai	gh	bf	if
Pesos	1	1	1	2	2	4	4	4	6	6	7	7
¿Añadir?	Si	Si	Si	Si	Si	Si	Si	Si	No	No	No	No

Tabla 1.1: Tabla de Pesos Crecientes para Figura 1.27

- i. Clasificamos las aristas de en orden creciente (Tabla 1.1)
- ii. Añadir a $T(E)$; $T(E) = \{a\}$; $T(E) = \{a, e\} \dots$
- iii. Obtenemos el Árbol Recubrido Mínimo: $T(E) = \{a, e, b, c, h, d, f, i\}$

1.5.5.3. Árboles m -arios

Definición 1.5.44. Definimos un **Árbol con Raíz** si uno de sus vértices se nombra de esta manera (vértice Raíz R).

(a) Ejemplo de Árbol Raíz m -ario.

(b) Ejemplo de Árbol Raíz Binario.

Figura 1.28: Ejemplo de Árboles m -arios.

Definición 1.5.45. Un **Árbol con Raíz es m -ario** (con $m \geq 2$) si de designamos al número máximo de hijos por cada nodo con m .

Definición 1.5.46. Un Árbol es m -ario completo si por cada vértice tiene m hijos o ninguno.

Recorridos: Existen tres tipos de algoritmos para recorrer Árboles m -arios:

Nota: Siendo R_1, R_2, \dots, R_n subárboles de R de Izquierda a Derecha.

- I. **Preorden** (Raíz, Izquierda, Derecha): Parte de la raíz r para recorrer los vértices de: R_1, R_2, \dots, R_n en Preorden.

Ejemplo 1.5.47. En el caso de la Figura 1.29: $\{a, b, c, f, g, h, d, e\}$

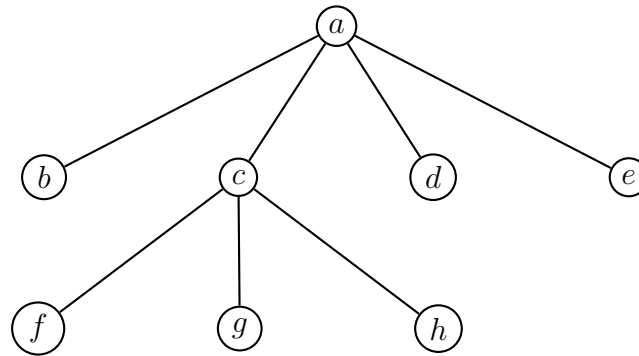


Figura 1.29: Ejemplo de Árbol con Raíz.

II. **Postorden** (Izquierda, Derecha, Raíz): Recorre los vértices: R_1, R_2, \dots, R_n en Postorden para terminar finalmente en r .

Ejemplo 1.5.48. En el caso de la Figura 1.29: $\{b, f, g, h, c, d, e, a\}$

III. **Inorden** (Izquierda, Raíz, Derecha): Si r contiene: R_1, R_2, \dots, R_n entonces recorre los nodos de izquierda a derecha R_i para volver a r y recorrer en Inorden R_{i+1} hasta finalizar en R_n .

Ejemplo 1.5.49. En el caso de la Figura 1.29: $\{b, a, f, c, g, h, d, e\}$

Definición 1.5.50. Los Árboles Binarios son de tipo *2-ario*, es decir: $m = 2$.

Notas

¹La Teoría de Conjuntos se trata de la síntesis de siglos de trabajo con el objetivo de llegar a una descripción formal de un grupo o elementos relacionados. La figura que finalmente dio forma a estos grupos de elementos es Georg Ferdinand Ludwig Philipp Cantor, nacido el 3 de Marzo de 1845 en San Petersburgo (Rusia) y fallecido el 6 de Enero de 1918 en Halle, Alemania.

²El concepto de Función como unidad estructural del Cálculo se debe al intenso trabajo de: **René Descartes**, **Isaac Newton** y **Gottfried Leibniz** siendo este último, el estableció términos como: función, variable, constante y parámetro. **Gottfried Leibniz** nacido el 1 de Julio de 1646 en el Electorado de Sajonia y fallecido el 14 de Noviembre de 1716 en Hannover, Electorado de Brunswick-Lüneburg, **fue un importante filósofo y matemático del siglo XVII padre del Cálculo Infinitesimal** (desde una perspectiva matemática junto a Isaac Newton (desde un principio físico). **Igualmente inventó el Sistema Binario** que actualmente es la lógica base de cualquier computadora digital.

³El origen de la **Teoría de Grafos** parte de la famosa publicación “**Los siete puentes de Königsberg**” donde su autor **Leonhard Euler**, nacido el 15 de Abril de 1707 en Basilea (Suiza) y fallecido el 18 de Septiembre de 1783 en San Petersburgo (Rusia), se preguntaba como en la propia ciudad de Königsberg (actual Kaliningrad) era posible cruzar los siete puentes una sola vez del río Pregel iniciando y finalizado el trayecto en el mismo punto. Para ello determinó un modelo:

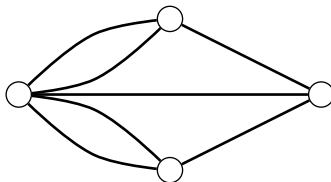


Figura 1.30: Grafo de Königsberg.

y postuló su famoso Teorema. *El Teorema de Euler dice que para un grafo $G = (V, E, p)$ o multigrafo (no digrafo) sin vértices aislados, G posee un Circuito de Euler si y sólo si G es conexo y cada vértice tiene grado par.*

⁴La figura de **Sir William Rowan Hamilton** nacido el 4 de Agosto de 1805 en Dublin (Irlanda) y fallecido en 1865 en Dublin, reformó el trabajo previo sobre la Teoría de Grafos llegando a la conclusión de que **en ciertas condiciones es posible recorrer un grafo con el mismo punto de origen y destino pasando por todas sus aristas una sola vez**. Tras este trabajo postuló su Teorema: *Si un grafo $G = (V, E, p)$ tiene $|V| \geq 3$ y $\delta(v_i) \geq 2$, entonces G es hamiltoniano*

Capítulo 2

Resumen: Proyecto gp1990c

Resumen:

2.1. Objetivos	35
2.2. Descripción general del proyecto	35
2.3. Transformación de una Expresión Regular en Software	36
2.4. Breve descripción de gp19901a	37
2.5. Breve descripción de gp1990sa	38
2.6. Métodos y Fases de desarrollo	38
2.7. Entorno de desarrollo	41
Notas	43

2.1. Objetivos

El Proyecto Fin de Carrera gp1990c gira en torno a dos conceptos:

- i. **Desarrollar el estándar ISO Pascal 7185:1990⁵ además de la construcción de un prototipo** para su parte léxica (basada en Flex) y su parte sintáctica (basada en Bison).
- ii. **Síntesis y Lenguaje Matemático propio de la Teoría de Lenguajes de Programación** así como su evolución e influencias históricas.

2.2. Descripción general del proyecto

El Software estará compuesto por dos elementos atómicos desde el punto de vista funcional pero que se interconectan para constituir, como hemos dicho el analizador.

Brevemente enumeraremos sus partes:

- i. **Analizador Léxico (gp19901a):** Es el elemento encargado de verificar que el conjunto de palabras de código fuente pertenecen al lenguaje. Genera un fichero `lex.yy.c`.

- ii. Analizador Sintáctico ⁶. (**gp1990sa**): Es el elemento encargado de comprobar que el orden de esas palabras corresponde a la propia sintaxis (Reglas) del lenguaje. Genera un fichero `y.tab.c`

Compilación: El proceso para crear un ejecutable a partir de código Flex/Bison⁷ sería:

```
$ bison -yd gp1990sa.y  
$ flex gp1990la.l  
$ gcc y.tab.c lex.yy.c -lfl -o programa
```

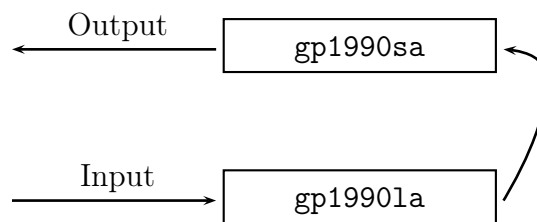


Figura 2.1: Síntesis y Partes de **gp1990c**.

2.3. Transformación de una Expresión Regular en Software

Dicho proceso comprende una serie de etapas:

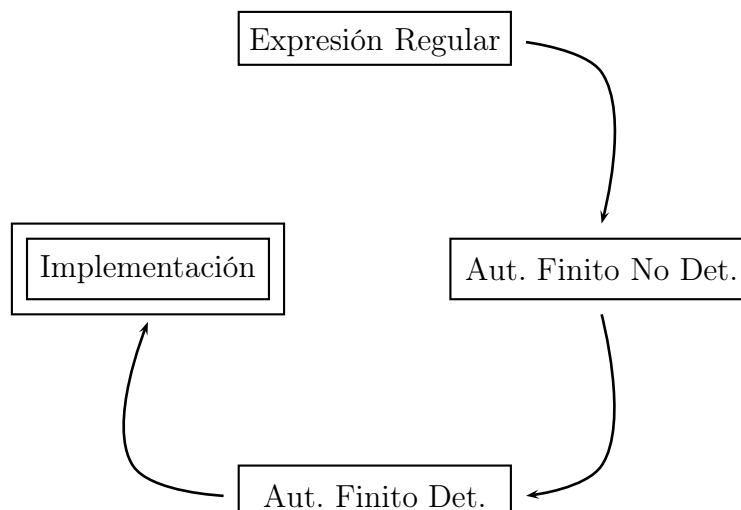


Figura 2.2: Transformación desde una Expresión Regular a su Implementación.

Definición 2.3.1. Expresión Regular: Se trata de una simplificación de una cadena de caracteres (Ver Apartado 5.4).

Definición 2.3.2. AFND (Autómata Finito no Determinista): Reconocedor de expresiones regulares con transiciones δ del tipo (Ver Apartado 5.5.4):

$$\delta(i, o) \rightarrow q; / e \in Q \wedge s \in \Sigma \cup \{\lambda\} \wedge q \subset Q \quad (2.1)$$

Definición 2.3.3. AFD (Autómata Finito Determinista): Reconocedor de expresiones regulares con transiciones δ del tipo (Ver Apartado 5.5.3):

$$\delta(i, o) \rightarrow q_i; / e \in Q \wedge s \in \Sigma \cup \{\lambda\} \wedge q_i \subset Q \quad (2.2)$$

Definición 2.3.4. Minimización de los estados: Dicho proceso se basa en el siguiente teorema:

Teorema 2.3.5. Para cualquier Autómata Finito, existe un Autómata Finito Mínimo equivalente (Ver Apartado 5.5.5).

Implementación: La implementación y desarrollo de un analizador depende en gran medida de tipo de lenguaje base. Existen la siguiente clasificación para el análisis de Gramáticas Libres de Contexto⁸

2.4. Breve descripción de gp1990la

Definición 2.4.1. gp1990la en un Analizador Léxico para ISO Pascal 7185:1990.

Implementación: Hay tres tipos de implementaciones para un Analizador Léxico:

- i. Implementación Software con Lenguaje de Alto Nivel: Se programa el analizador con un lenguaje que permita rutinas de bajo nivel, normalmente Lenguajes C y C++.
- ii. Implementación en Lenguaje Ensamblador: Código “a priori” nativo para una determinada arquitectura.
- iii. Lex: Se trata de un programa que se adapta a las necesidades de un alfabeto y es capaz de reconocer y ordenar tokens.

Implementación	Eficiencia	Velocidad	Portabilidad
Aplicación Lex	Regular	Regular	Óptima
Código C	Buena/Muy Buena	Buena/Muy Buena	Óptima
Código C++	Buena	Buena/Muy Buena	Buena/Muy Buena
Código Ensamblador	Muy Buena	Óptima	Muy Mala

Tabla 2.1: Comparativa para los distintos tipos de implementaciones para un AL.

Nota: Las formalidades que describen a un Analizador Léxico se tratan con más detalle en el Capítulo 5.

2.5. Breve descripción de gp1990sa

Definición 2.5.1. gp1990sa es un Analizador Sintáctico para ISO Pascal 7185:1990.

Definición 2.5.2. La finalidad de p1990sa o Analizador Sintáctico es la de certificar que las palabras del lenguaje se organizan de acuerdo a la estructura del lenguaje.

Implementación: Existen tres tipos de implementaciones para un Analizador Sintático:

- i. Analizador Sintáctico Descendente (Top-Down-Parser): Se basa en analizar una gramática a partir de cada símbolo no terminal (es un desarrollo de arriba hacia abajo). Son formalmente denominados Analizadores LL.
- ii. Analizador Sintáctico Ascendente (Bottom-Up-Parser): Analizan la gramática a partir de los símbolos terminales (por ello es un desarrollo de abajo hacia arriba). Son formalmente denominados Analizadores LR.
- iii. Yacc: A partir de una gramática no ambigua (aunque también acepta gramáticas ambiguas con resolución de problemas) genera un autómata tipo LALR (analizador sintáctico LR con lectura anticipada).

2.6. Métodos y Fases de desarrollo

El proyecto se construirá siguiendo el modelo clásico de **Ciclo de desarrollo en Cascada**⁹: Análisis, Diseño, Codificación y Pruebas.

- I. Análisis: Consistirá en un estudio sobre los fundamentos matemáticos de los compiladores (con especial énfasis en el Lenguaje de Programación Pascal) además de una contextualización y evolución de los compiladores.
- II. Diseño: Sobre la base teórica antes descrita, se hará un estudio teórico-práctico sobre las herramientas Lex/Yacc cara a la especificación de los prototipos: gp1990la y gp1990sa.
- III. Codificación: Para las herramientas Flex/Bison:
 - i. gp1990la.1: Fichero de especificación léxica. Será un modelo funcional que incluirá todo lo necesario para realizar programas sencillos.
 - ii. gp1990sa.y: Fichero de especificación de las reglas sintácticas.
 - iii. GNU Build System (Autoconf¹⁰): Ficheros makefile.am y configure.ac con el objetivo mejorar la compatibilidad de la herramientas con otras familias UNIX (principalmente GNU/Linux y BSD) además de ser una potente ayuda para futuras correcciones y mejoras

IV. Pruebas: Usando GNU Pascal Compiler¹¹(**gpc**) y Free Pascal¹²(**fpc**) se realizará una batería de pruebas sobre las partes léxica y sintáctica basadas en algoritmos clásicos:

- i. Algoritmos de Ordenación: Selección Directa, Inserción Directa, Intercambio Directo, Ordenación Rápida (*Quick Sort*) y Ordenación por Mezcla (*Merge Sort*),
- ii. Algoritmos de Búsqueda: Búsqueda Secuencial, Búsqueda Secuencial Ordenada y Búsqueda Binaria.

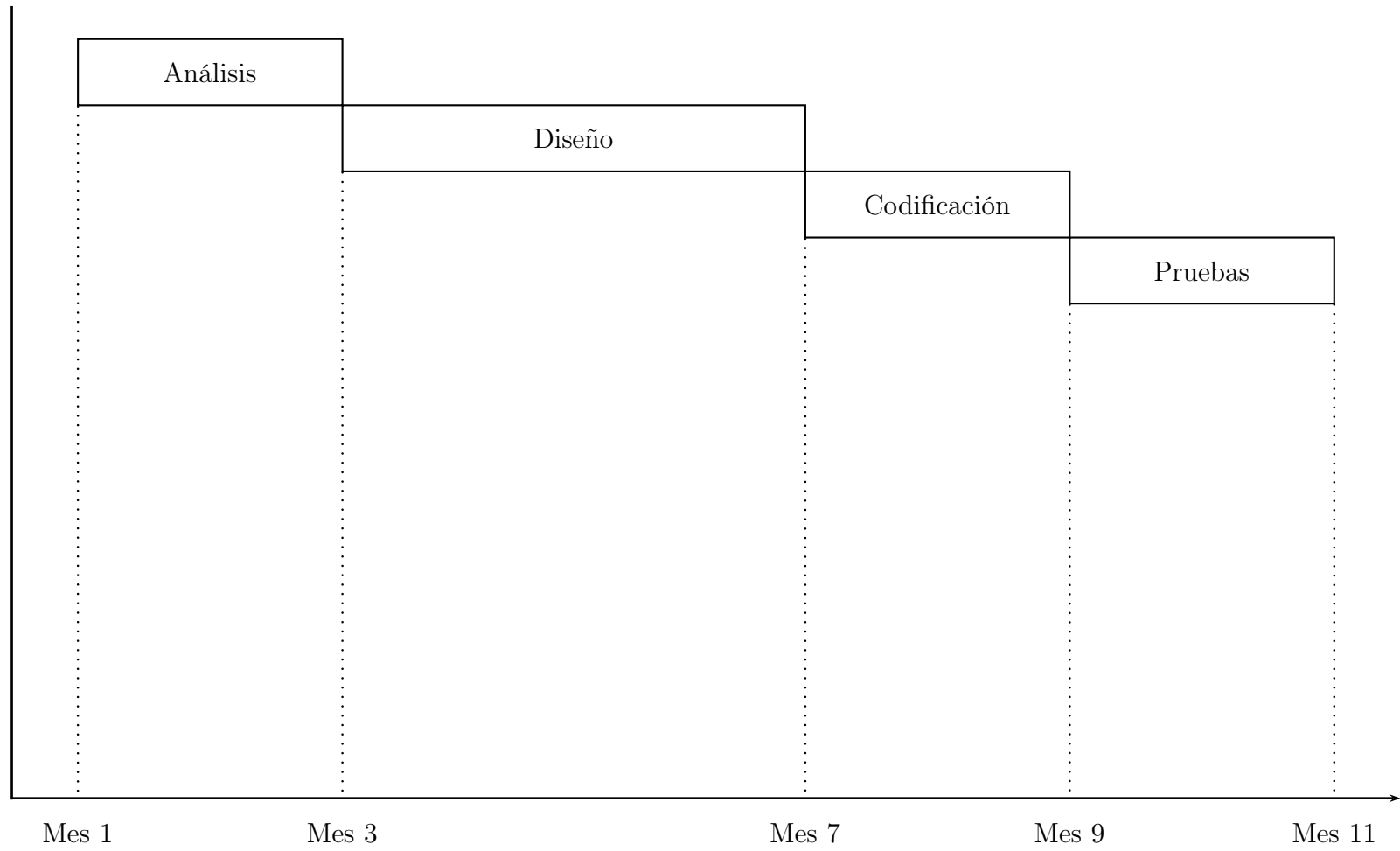


Figura 2.3: Diagrama de Gantt para desarrollo de gp1990c.

2.7. Entorno de desarrollo

- I. GNU/Linux: Sistema Operativo base (Gentoo GNU/Linux¹³ para el desarrollo del Software y la documentación. La elección de GNU/Linux se debe principalmente a la plena compatibilidad con las herramientas de desarrollo tanto del Software como de la documentación (escrita con $\text{\LaTeX} 2_{\epsilon}$). También es resaltable el hecho de que es compatible con otras familias UNIX como BSD.
- II. BSD¹⁴: Principalmente usaremos la versión FreeBSD (derivado de BSD-Lite 4.4) para mejorar la compatibilidad del Software. Se usará especialmente para configurar y ajustar las herramientas GNU Build System así como la pruebas de estabilidad y optimización del código fuente.
- III. GCC¹⁵: Metacompilador que nos servirá para generar programas ejecutables.
- IV. GNU Build System: Conjunto de herramientas:
 - i GNU Autoconf: Se trata de una herramienta de propósito general para generar ficheros ejecutables para distintas versiones de UNIX. Usa: `configure.ac` y `makefile.in` para generar `makefile` sobre el entorno.
 - ii GNU Automake: Genera el fichero `makefile.in` a partir de las especificaciones de `makefile.am` necesario para Autoconf.
 - iii GNU Libtool: Se trata de una herramienta que genera bibliotecas estáticas y dinámicas para las distintas versiones de UNIX.
- V. Flex¹⁶: Flex (Fast Lexical Analyzer Generator) se trata de un programa para el análisis léxico de Lenguajes Regulares (versión GNU de Lex). Internamente es un Autómata Finito Determinista (AFD).
- VI. Bison¹⁷: Se trata de un analizador sintáctico (versión GNU de Yacc) para Gramáticas Libres de Contexto (también es capaz de generar código para algunos tipos de Gramáticas Ambiguas). Se trata de una analizador que genera un autómata LALR para los Lenguajes C, C++ y Java (principalmente).
- VII. \TeX Live 2011¹⁸: Es la Metadistribución de \TeX común para sistemas GNU. Contiene todos los paquetes oficiales propuestos por \TeX Users Group. Se usarán además los entornos PStricks y MetaPost para la generación de gráficos vectoriales.

Notas

⁵<http://www.moorecad.com/standardpascal/>

⁶En inglés se denomina “parser”

⁷Compatible con Lex/Yacc.

⁸Gramática Chomskiana de Tipo 2: $P = \{(S \rightarrow \lambda) \vee (A \rightarrow p_2) \mid p_2 \in \Sigma^+; A \in N\}$

⁹Debido al contexto del proyecto se omite la fase de Mantenimiento.

¹⁰<http://www.gnu.org/software/autoconf/>

¹¹**GNU Pascal Compiler:**

- i. Desarrollador: GNU Pascal Development Team.
- ii. Última versión estable: 2.1
- iii. Tipo de sistema base: UNIX y clones.
- iv. Licencia: GPL.
- v. Página Web: <http://www.gnu-pascal.de/>

¹²**Free Pascal:**

- i. Desarrollador: Free Pascal Team.
- ii. Última versión estable: 2.6.0
- iii. Tipo de sistema base: Multiplataforma.
- iv. Licencia: GPL.
- v. Página Web: <http://www.freepascal.org/>

¹³**Gentoo GNU/Linux:**

- i. Desarrollador: Comunidad Gentoo GNU/Linux.
- ii. Última versión estable: 12.1
- iii. Tipo de sistema base: Monolítico.
- iv. Licencia: GPL y otras Licencias Libres.
- v. Página Web: <http://www.gentoo.org/>

¹⁴**FreeBSD (Free Berkeley Software Distribution):**

- i. Desarrollador: Comunidad FreeBSD.
- ii. Última versión estable: 9.1
- iii. Tipo de sistema base: Monolítico.
- iv. Licencia: Licencia BSD.
- v. Página Web: <http://www.freebsd.org/>

¹⁵**GCC (GNU Compiler Collection):**

- i. Desarrollador: Proyecto GNU.
- ii. Última versión estable: 4.8.1
- iii. Tipo de sistema base: UNIX y clones.
- iv. Licencia: Licencia GPLv3.
- v. Página Web: <http://gcc.gnu.org/>

¹⁶**Flex (Fast Lexical Analyzer Generator):**

- i. Desarrollador: Vern Paxson.

- ii. Última versión estable: 2.5.37 (3 de Agosto de 2012)
- iii. Tipo de sistema base: UNIX y clones.
- iv. Licencia: Licencia BSD.
- v. Página Web: <http://flex.sourceforge.net/>

¹⁷**Bison (GNU Bison):**

- i. Desarrollador: Proyecto GNU.
- ii. Última versión estable: 3.0 (26 de Julio de 2013)
- iii. Tipo de sistema base: UNIX y clones.
- iv. Licencia: Licencia GPL.
- v. Página Web: <http://www.gnu.org/software/bison/>

¹⁸**TeX Live 2011:**

- i. Desarrollador: TeX Users Group.
- ii. Última versión estable: 2013.
- iii. Tipo de sistema base: Familia UNIX, Familia GNU/Linux y Familia Win2k.
- iv. Licencia: LaTeX Project Public License (LPPL), GPLv2.
- v. Página Web: <http://www.tug.org/texlive/>

Capítulo 3

El Lenguaje de Programación Pascal

Resumen:

3.1. Introducción	45
3.2. Influencias del Lenguaje Pascal	46
3.3. El Lenguaje Pascal	53
3.4. Evoluciones del Lenguaje Pascal	60
Notas	67

3.1. Introducción

*A programming language called Pascal is described which was developed on the basis of Algol 60. Compared to Algol 60, its range of applicability is considerably increased due to a variety of data structuring facilities. In view of its intended usage both as convenient basis to teach programming and as an efficient tool to write large programs, emphasis was placed on keeping the number of fundamental concepts reasonably small, on a simple and systematic language structure, and on efficient implementability. A one-pass compiler has been constructed for the CDC 6000 computer family; it is expressed entirely in terms of Pascal itself.*¹⁹ [Wir71]

El Lenguaje de Programación Pascal fue creado por el profesor Niklaus Wirth²⁰ a finales de la década de los sesenta del siglo XX. En 1970 fue finalmente publicado, fijando dos objetivos en su diseño arquitectónico:

- Crear un **lenguaje claro y natural orientado a la enseñanza** de los fundamentos de la programación de computadores. Por ello se estructuran los módulos como funciones y procedimientos.
- Definir un lenguaje que **permita realizar programas lo más eficientes posibles**. El tipado de datos es explícito.

Pascal recibe su nombre en honor al matemático francés Blaise Pascal (ver Anexo A).

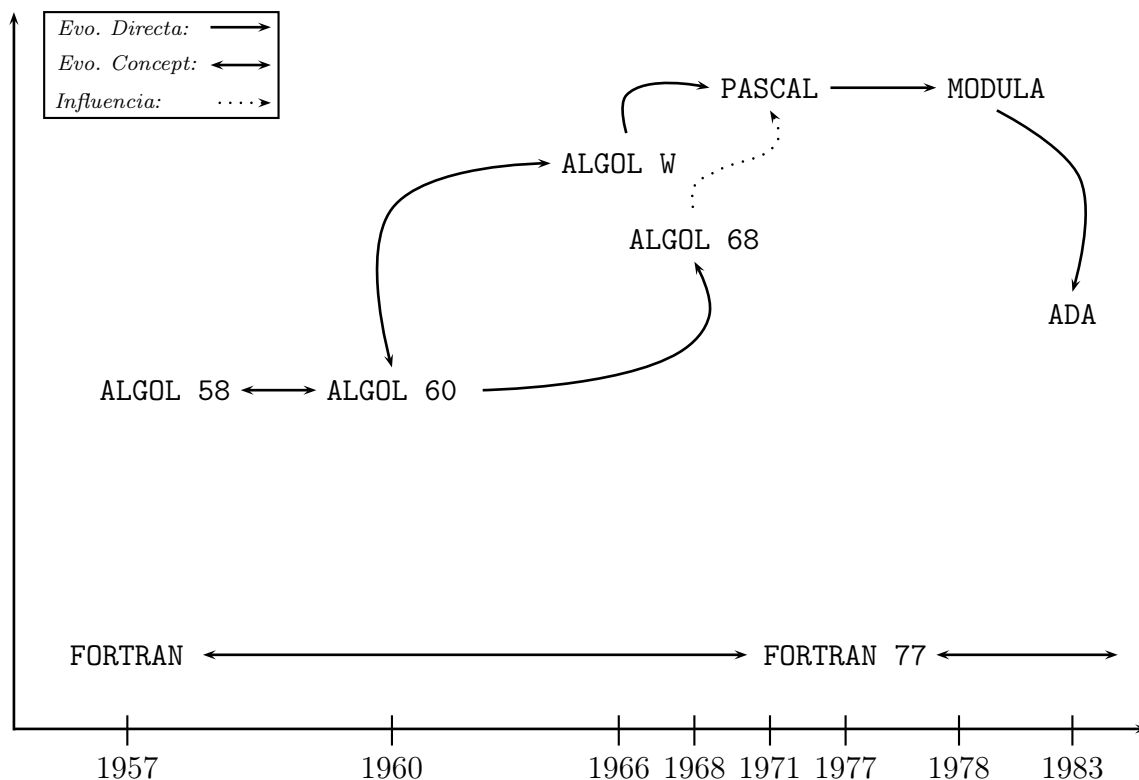


Figura 3.1: Relaciones entre los primeros Lenguajes de Programación.

3.2. Influencias del Lenguaje Pascal

3.2.1. Fortran (The IBM Mathematical Formula Translating System)

Fortran, inicialmente conocido como **FORTRAN** es el acrónimo de *The IBM Mathematical Formula Translating System*.

Fortan se trata del primer lenguaje de alto nivel. Es multipropósito y se basa en el paradigma de la programación estructurada.

Su origen tiene que ver con la necesidad de crear aplicaciones científicas de manera más sencilla y lógica para el entendimiento humano.

The FORTRAN language is intended to be capable of expressing any problem of numerical computation. In particular, it deals easily with problems containing large sets of formulae and many variables, and it permits any variable to have up to three independent subscripts. However, for problems in which machine words have a logical rather than a numerical meaning it is less satisfactory, and it may fail entirely to express some such problems. Nevertheless, many logical operations not directly expressable in the FORTRAN language can be obtained by making use of provisions for incorporating library routines. ²¹ [Sta66]

El primer proyecto de compilador de FORTRAN fue un Milestone que ocupaba 15KB aproximadamente. Era muy rudimentario y funcionaba con rutinas muy primitivas de los SSOO de la época, prácticamente era código ensamblador.

El compilador oficial de FORTRAN fue escrito entre 1954 y 1957 a cargo de John W. Backus y grandes programadores como: Sheldon F. Best, Harlan Herrick, Peter Sheridan, Roy Nutt, Robert Nelson, Irving Ziller, Richard Goldberg, Lois Haibt and David Sayre. La primera

ejecución del compilador se realizó sobre una máquina IBM 704.

Su primeros programas fueron para control energético de reactores nucleares. Demostraba ser mucho más rápido que otras soluciones tradicionales sobre Lenguaje Ensamblador.

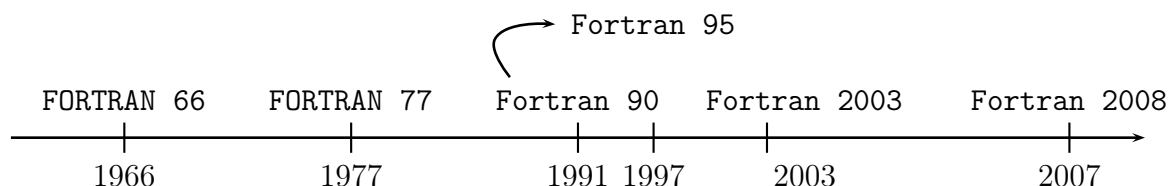


Figura 3.2: Evolución del Lenguaje Fortran.

El Lenguaje Fortran ha sido parte de seis estandarizaciones:

- I. FORTRAN o FORTRAN 66: La característica más destacada es la separación de las fases de compilación, además de la posibilidad de enlazar con rutinas de lenguaje ensamblador.
- II. FORTRAN 77: Entre sus características destacan:
 - i. Bucles `DO` con variable índice de incremento y decremento.
 - ii. Bloque de secuencias: `{IF...THEN...ELSE...ENDIF.}`
 - iii. Pruebas antes de compilación de bucles `{DO}`.
 - iv. Tipo de dato `CHARACTER`.
 - v. El símbolo apostrofe (') como delimitador de conjuntos de caracteres.
 - vi. Final de un programa sin necesidad de usar la palabra `{STOP}`.
- III. Fortran 90: Sus principales novedades son:
 - i. Nuevas estructuras de flujo: `{CASE & DO WHILE}`.
 - ii. Estructuras de datos tipo `RECORD`.
 - iii. Mejora en el manejo de `ARRAY` (nuevos operadores).
 - iv. Memoria dinámica.
 - v. Sobrecarga de operadores.
 - vi. Paso de argumentos por referencia.
 - vii. Control de precisión y rango.
 - viii. Módulos (paquetes de código).
- IV. Fortran 95:
 - i. Construcciones `{FORALL}`.
 - ii. Procedimientos `PURE` y `ELEMENTAL`.
 - iii. Mejoras en la inicialización de objetos.

- iv. Sentencia {DO} para tipos de datos: REAL y DOUBLE PRECISION.
- v. Sentencia {END IF} para terminar bloque.
- vi. Sentencia {PAUSE}.
- vii. Incorporación de ISO/IEC 1539-1:1997 que incluye dos tipos de módulos opcionales:
 - a. STRINGS dinámicos ISO/IEC 1539-2:2000.
 - b. Compilación condicional ISO/IEC 1539-3:1998.

V. Fortran 2003:

- i. Soporte de Programación Orientada a Objetos: Extensión de tipos, Polimorfismo y completo soporte para TADS (Tipos Abstractos de Datos) entre otras características.
- ii. Mejora en la manipulación de memoria: Valores por referencia, atributo VOLATILE, especificación explícita de constructores para ARRAY y sentencia {POINTER}.
- iii. Mejoras en Entrada/Salida: Transferencia asíncrona, acceso por flujo (STEAM), especificación de operaciones de transferencia, sentencias de control y de redondeo para conversiones y sentencia {FLUSH}.
- iv. Soporte para aritmética flotante de IEEE.
- v. Interoperabilidad con el Lenguaje de Programación C.
- vi. Internacionalización ISO 1064.
- vii. Mejora en la integración con SSOO anfitrión: Acceso a línea de comandos, variables de sistema, procesos y mensajes de error.

VI. Fortran 2008:

- i. Submódulos ISO/IEC TR 19767:2005.
- ii. Modelos de ARRAY para ejecución en paralelo.
- iii. Construcción {DO CONCURRENT}.
- iv. Atributo CONTIGUOUS.
- v. Construcciones de tipo BLOCK.
- vi. Componentes recursivos dinámicos.

3.2.1.1. Análisis de Fortran

Nota: Basado en: Fortran ISO 2003.

I. Alfabeto:

- i. 26 letras²²: 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k'
| 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x'
| 'y' | 'z'
- ii. 10 dígitos: '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
- iii. Carácter *Underscore*: '_'

Carácter	Nombre	Carácter	Nombre
	Blank	;	Semicolon
=	Equals	!	Exclamation point
+	Plus	"	Quotation mark or quote
-	Minus	%	Percent
*	Asterisk	&	Ampersand
/	Slash	~	Tilde
\	Backslash	<	Less than
(Left parenthesis	>	Greater than
)	Right parenthesis	?	Question mark
[Left square bracket	'	Apostrophe
]	Right square bracket	`	Grave accent
{	Left curly bracket	^	Circumflex accent
}	Right curly bracket		Vertical line
,	Comma	\$	Currency symbol
.	Decimal point or period	#	Number sign
:	Colon	@	Commercial at

Figura 3.3: Símbolos especiales de Fortran 2003.

iv. Símbolos especiales: ver Figura (3.3).

v. Otros símbolos: Dichos símbolos pueden ser representables pero solamente aparecen en: comentarios, caracteres de constantes, registros de entrada/salida y descripciones.

II. Gramática: Ver bibliografía capitular [ISO04].

Programa 3.2.1. helloProgrammer.f95

```

1 PROGRAM HELLOPROGRAMMER
2     WRITE (*,*) "Hello programmer!"
3 END PROGRAM

```

Notas sobre compilación: Para compilar el archivo fuente `helloProgrammer.f95` sobre GNU, usaremos el compilador GNU Fortran²³.

Las ordenes para compilarlo y ejecutarlo son las siguientes:

```

$ gfortran -o helloProgrammer helloProgrammer.f95
$ ./helloProgrammer
Hello programmer!

```

3.2.2. ALGOL (ALGO^rithmic Language)

3.2.2.1. Definiciones

Definición 3.2.2. ALGOL inicialmente recibió el nombre de IAL *Internationa Alogrithmic Language*.

Definición 3.2.3. ALGOL se trata de una familia de Lenguajes de Programación basados todos ellos en la primera versión del Lenguaje base ALGOL 58.

Definición 3.2.4. Diseñado entre 1957 y 1960 por un comité de científicos europeos y americanos que se basaban en dos ideas principales:

- i. Mejorar las deficiencias estructurales de FORTRAN (todavía sin estándar pero ampliamente usado).
- ii. Crear un lenguaje altamente expresivo que sea capaz de dar una respuesta común a todos los científicos.

Corolario 3.2.5. *Unos de sus notables avances fue el de limitar unidades de código (sentencias) en bloques {BEGIN...END.}*

3.2.2.2. Historia

The purpose of the algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well known arithmetic expression containing as constituents numbers, variables, and functions. From such expressions are compounded, by applying rules of arithmetic composition, self-contained units of the language—explicit formulae—called assignment statements. ²⁴ [ea60]

Como hemos dicho anteriormente, ALGOL tiene su primera especificación formal en el año 1958. Este documento base (ALGOL 58) fue oficialmente presentado en tres formatos:

- i. *Reference Language* (Lenguaje de Referencia): Es el documento donde se recoge íntegramente el trabajo del Comité (Enero de 1960). En el mismo, se define el lenguaje basándose en notación matemática. Así mismo, es la referencia básica de ALGOL.
- ii. *Publications Language* (Lenguaje de Publicaciones): Frente a *Reference Language* permite variaciones en la simbología del documento para que pueda ser publicado y distribuido internacionalmente.
- iii. *Hardware Representations* (Representaciones Hardware): Se trata de consideraciones sobre *Reference Language* con el objetivo de limitar la especificación al Hardware de la época.

ALGOL desde su comienzo tuvo un importante nicho entre científico europeos y americanos. De igual manera, ALGOL introduce en su especificación formal la notación Backus-Naur Form que ha sido utilizada desde entonces como método descriptivo de los Lenguajes de Programación. También es notable el hecho de que ALGOL es el primer lenguaje que combina el flujo imperativo con *Lambda-Calculus*.

La primera versión estandarizada de ALGOL es ALGOL 58 que finalmente fue mejorado y actualizado con la nueva versión ALGOL 60.

Símbolo	Descripción
Corchetes []	Determinar la existencia o no de espacio ' ' y otros caracteres.
Exponenciación ↑	Determinar la operación Exponente.
Paréntesis ()	Usado como: Paréntesis o Corchete.
Base 10: a_{10}	Determinar la notación de Base 10 en operaciones matemáticas.

Tabla 3.1: Convención entre *Reference Language* y otras publicaciones de ALGOL.

ALGOL 60 es uno de los estándares más usados y marco de referencia básica para la creación y especificación de otros lenguajes. Ha sido por ello, base de lenguajes tan importantes como: BCPC, B, Pascal, Simula o C.

El problema que intentó solucionar esta versión fue la de hacer de ALGOL un lenguaje con aspiraciones comerciales. Por ello, se trabajó intensamente en mejorar la Entrada/Salida y la relación con el entorno de ejecución (SSOO).

Partiendo de este estándar conceptual y muy avanzado surgieron dos nuevas propuestas:

- i. ALGOL 68: Sobre ALGOL 68 destacar que añade gran cantidad de utilidades que eran comúnmente utilizadas por programadores de la época, entre ellas: declaración de tipos, estructuras de unión y modelos de variables por referencia.

La especificación de esta nueva revisión adoptó la notación de Adriann van Wijgaarden, que usaba gramáticas libres de contexto para generar infinitos conjuntos de producción.

- ii. ALGOL W: ALGOL W fue un proyecto encargado al profesor Nicklaus Wirth que tras la publicación de ALGOL 68 y las enormes quejas que despertó, trataba de actualizar ALGOL 60 intentando conservar la esencia y cultura del lenguaje.

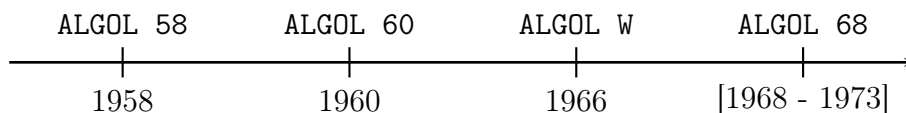


Figura 3.4: Evolución del Lenguaje ALGOL.

3.2.2.3. ALGOL 60

I. Alfabeto:

i. Letras:

```
<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l |
           m | n | o | p | q | r | s | t | u | v | w | x | y | z | A |
           B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
           Q | R | S | T | U | V | W | X | Y | Z
```

ii. 10 dígitos: `<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

iii. Valores lógicos: `<logical value> ::= true | false`

iv. Delimitadores:

```
<delimiter> ::= <operator> | <separator> | <bracket> |
               <declarator> | <specificator>
```

```
<operator> ::= <arithmetic operator> | <relational operator> |
               <logical operator> | <sequential operator>
```

```
<arithmetic operator> ::= + | - | TIMES | / | ÷ | POWER
```

```
<relational operator> ::= < | NOTGREATER | = | NOTLESS | > | NOTEQUAL
```

```
<logical operator> ::= EQUIVALENCE | IMPLICATION | OR | AND | ¬
```

```
<sequential operator> ::= goto | if | then |
                        else | for | do (2)
```

```
<separator> ::= , | . | 10 | : | ; | := | BLANK | step |
               until | while | comment
```

```
<bracket> ::= ( | ) | [ | ] | ' | ' | begin | end
```

```
<declarator> ::= own | Boolean | integer |
               real | array | switch |
               procedure
```

```
<specificator> ::= string | label |
                 value
```

II. Gramática: Ver bibliografía capitular .

Programa 3.2.6. helloProgrammer.a60

```
1 | (
2 |   printf((gl,"Hello Programmer!"))
3 | )
```

Notas sobre compilación: Para compilar el archivo fuente `helloProgrammer.a60` sobre GNU, usaremos traductor de ALGOL a Lenguaje C Marst ²⁵.

Las ordenes para compilarlo y ejecutarlo son las siguientes:

```
marst helloProgrammer.a60 -o helloProgrammer.a60
cc helloProgrammer.a60 -lalgol -lm -o ./helloProgrammer
./helloProgrammer
```

Hello Programmer!

3.2.2.4. ALGOL W

ALGOL W, inicialmente denominado ALGOL X, fue desarrollado por Nicklaus Wirth y C.A.R como sucesor directo de ALGOL 60 a propuesta en IFIP Working Group. La especificación del lenguaje se vio insuficiente y fue finalmente publicada como: “*A contribution to the development of ALGOL*”.

Dicha especificación incluía mejoras que en ningún momento querían romper la armonía original de ALGOL 60. Se trataba de una revisión conservadora. Entre estas mejoras destacan:

- i. Tipo de datos `STRING`.
- ii. Incorporación de Números Complejos.
- iii. Llamada por referencia de tipos de datos `RECORD`.
- iv. Añade la estructura `WHILE`.
- v. Reemplazo de la estructura `SWITCH` por `CASE`.

3.3. El Lenguaje Pascal

Pascal se sustenta sobre dos poderosos lenguajes del ámbito científico: FORTRAN y ALGOL, de los que anteriormente hemos hablado.

Lo que trataba de hacer Wirth era actualizar ALGOL 60 en un nuevo lenguaje de propósito mucho más general.

El primer prototipo de esta idea fue ALGOL W programado sobre una computadora IBM 360. Fue una versión bastante conservadora del lenguaje lo que dio fuerza a la idea de que el nuevo lenguaje que deseaba construir Wirth tenía que soportar un repertorio de mucho más amplio.

Wirth además **quería que dicho lenguaje tuviera fines educativos**, es decir, **que enseñase la cultura de “la buena programación”**. Para ello tomo como referencia a FORTRAN y al Lenguaje Ensamblador.

En 1968 cuando empezó a implementar estos hitos en el futuro lenguaje.

Había igualmente una alta competencia con las soluciones de compilación que ofrecía FORTRAN, por ello decidió que su compilador sería de una sola pasada²⁶ basada en el diseño “Top-Down”.

El compilador se completó finalmente a mediados de 1970.

Pascal después llegó a ser un lenguaje muy popular en círculos universitarios durante la década de los ochenta y noventa del siglo XX debido principalmente a la venta de compiladores muy económicos, y un IDE de propósito general que se basaba en el mismo, hablamos de Turbo Pascal.

3.3.1. Pascal ISO 7185:1990

En 1977 BSI²⁷ [ISO91] produjo el estándar del Lenguaje de Programación Pascal, publicado en 1979. Ese mismo año, el organismo BSI propuso que Pascal fuese parte del programa ISO. Fue aceptado con denominación ISO/TC97/SC5/WG4.

En los Estados Unidos de América, IEEE aprobó el 10978 del proyecto 770 (Pascal).

En Diciembre, 178 X3J9 convino el resultado de SPARCH²⁸ para la resolución de US TAG²⁹ para la ISO Pascal.

En Febrero de 1979, representantes de IEEE combinaron los proyectos X3 y IEEE 770 bajo el comité X3J9/IEEE-P770 Pascal Standards (JPC).

La resolución de JFC fue avalada en NSI/IEEE770X3 .97-1983 por ANSI bajo American National Standard Pascal Computer Programming Language.

Las especificaciones de BSI se hicieron públicas en 1982, internacionalmente conocido como Standard 7185.

3.3.1.1. Alfabeto

I. Unidades:

- i. letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' .
- ii. digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .

II. Símbolos:

- i. special-symbol = '+' | '-' | '*' | '/' | '=' | '<' | '>' | '[' | ']' | '.' | ',' | ':' | ';' | '^' | '(' | ')' | '<>' | '<=' | '>=' | ':=' | '..' | word-symbol .
- ii. word-symbol = 'and' | 'array' | 'begin' | 'case' | 'const' | 'div' | 'do' | 'downto' | 'else' | 'end' | 'file' | 'for' | 'function' | 'goto' | 'if' | 'in' | 'label' | 'mod' | 'nil' | 'no' | 'of' | 'or' | 'packed' | 'procedure' | 'program' | 'record' | 'repeat' | 'set' | 'then' | 'to' | 'type' | 'until' | 'var' | 'while' | 'with' .

III. Identificadores: identifier = letter letter | digit .

IV. Directivas: directive = letter letter | digit .

V. Números:

- i. signed-number = signed-integer | signed-real .
- ii. signed-real = [sign] unsigned-real .
- iii. signed-integer = [sign] unsigned-integer .
- iv. unsigned-number = unsigned-integer | unsigned-real .
- v. sign = '+' | '-' .

- vi. unsigned-real = digit-sequence '.' fractional-part ['e' scale-factor | digit-sequence 'e' scale-factor .
- vii. unsigned-integer = digit-sequence
- viii. fractional-part = digit-sequence .
- ix. scale-factor = [sign] digit-sequence .
- x. digit-sequence = digit digit

VI. Etiquetas: label = digit-sequence .

VII. Cadenas de caracteres:

- i. character-string = '' string-element string-element '' .
- ii. string-element = apostrophe-image | string-character .
- iii. apostrophe-image = ''' .
- iv. string-character = one-of-a-set-of-implementation-defined-characters .

VIII. Separadores: ('{' | '(') commentary ('*' | '}')

3.3.1.2. Tipos de Datos

I. Datos Simples: Se trata de los tipos de datos base del propio lenguaje.

- i. Entero (Integer): Es un tipo de dato ordinal y representa en Conjunto de los Números Enteros.

Ejemplo 3.3.1. {+5; -4, 7}

- ii. Reales (Real): Representa el Conjunto de los Números Reales. Siendo los propios Reales de naturaleza matemática infinita, la precisión del número vendrá dada por la el tamaño del bloque en memoria asignado al dato.

Ejemplo 3.3.2. {+5.5; -4.2; 7.5}

- iii. Booleano (Boolean): Se trata de un conjunto ordinal de datos con dos posibles valores.

Formalidad 3.3.3. {TRUE; FALSE} \equiv {0, 1}

- iv. Carácter (Char): Depende de la naturaleza del valor, puede ser o no un tipo de dato ordinal. Se establece como un Subconjunto de:

- 1) El Conjunto de valores para dígito: [0, 9]
- 2) El Conjunto de valores para letra: [a - z, A - Z]

II. Datos Estructurados: Se trata de estructuras de datos complejas, definidas a partir de datos simples.

- i. Tipo Enumerado: Se trata de una lista de datos y valores hermética. Constituye un tipo ordinario finito y explícito en su declaración. a su declaración.

Ejemplo 3.3.4. (red, green, blue, yellow)

- ii. Tipo Subrango: Se trata de una subconjunto enumerado a partir otro meta-conjunto dónde sus valores son implícitos

Ejemplo 3.3.5. {1..80; -15..+15}

- iii. Tipo Array: Array se trata de una estructura de datos indexada con espacio en memoria estático. El “tipado” de la estructura Array viene determinado por la naturaleza de cada uno de sus datos (siendo estos obligatoriamente de la misma naturaleza).

Ejemplo 3.3.6. {ARRAY [1..1000] OF REAL; ARRAY [1..100] OF INTEGER}

- iv. Tipo Registro:

```

1 record-type = 'record' field-list 'end' .
2 field-list = [ ( fixed-part [';' variant-part ]
3               | variant-part ) [ ';' ] ] .
4 fixed-part = record-section { ';' record-section } .
5 record-section = identifier-list ':' type-denoter .
6 field-identifier = identifier .
7 variant-part = 'case' variant-selector 'of' variant
8               { ':' variant } .
9 variant-selector = [tag-field ':' ] tag-type .
10 tag-field = identifier .
11 variant = case-constant-list ':' '(' eld-list ')' .
12 tag-type = ordinal-type-identifier .
13 case-constant-list = case-constant { ',' case-constant } .
14 case-constant = constant .

```

Ejemplo 3.3.7. Registro:

```

1 RECORD
2     x,y: REAL;
3     area: REAL;
4     CASE shape OF
5         triangle:
6             (side: REAL; inclination, angle1, angle2: angle);
7         rectangle:
8             (side1, side2: REAL; skew: angle);
9         circle:
10            (diameter: REAL);
11     END

```

- v. Tipo Conjunto: Estructura heredada de la Teoría de Conjuntos (ver Apartado 1.1). Su declaración viene dada por: SET OF 'base-type'

Ejemplo 3.3.8. {SET OF CHAR; SET OF (red, green, blue, yellow)}

- vi. Tipo Fichero: Se trata de una estructura de “flujo” de valores. El tamaño de la misma viene dado por el conjunto de datos que serán leídos o escritos.

Ejemplo 3.3.9. {FILE OF REAL; FILE OF INTEGER}

- vii. Tipo Puntero: La estructura de Puntero tiene dos posibles valores:

- a. Null: Constituye el índice natural para las estructuras en memoria dinámica.
- b. Identificador de Valores: Se trata de un valor con naturaleza en memoria dinámico. A medida que se constituye como valor se reserva el espacio del tipo base. Dicho proceso se realiza a través del procedimiento `NEW()`. De igual manera, para “liberar” los recursos en memoria es necesario utilizar el procedimiento `DISPOSE()`.

Ejemplo 3.3.10. Puntero:

```

1  VAR
2      int_ptr : ^integer;
3  BEGIN
4      NEW(int_ptr);
5      int_ptr^ := 70;
6      WRITELN('Pointer value:', int_ptr^);
7      int_ptr^ := int_ptr^ + 7;
8      WRITELN('Pointer value:', int_ptr^);
9      DISPOSE(int_ptr^);
10 END;
```

3.3.1.3. Biblioteca

I. Procedimientos:

- i. `PROCEDURE REWRITE(f)` → Crea un fichero en modo escritura. En caso de existir el propio fichero es sobrescrito.
 - 1) Precondición: `True`
 - 2) Postcondición: $(f \uparrow)$ está indefinido.
- ii. `PROCEDURE PUT(f)` → Añade el valor del buffer del $(f \uparrow)$ al propio fichero.
 - 1) Precondición: $(f \uparrow)$ está definido.
 - 2) Postcondición: $(f \uparrow)$ está indefinido.
- iii. `PROCEDURE RESET(f)` → Abre un fichero en modo lectura con el puntero de fichero sobre el comienzo del mismo.
 - 1) Precondición: $(f \uparrow)$ está definido.
 - 2) Postcondición: $(f \uparrow)$ está indefinido.
- iv. `PROCEDURE GET(f)` → Avanza de descriptor de fichero y asigna el valor del buffer de al $(f \uparrow)$.
 - 1) Precondición: $(f \uparrow)$ está definido.
 - 2) Postcondición: $(f \uparrow)$ está indefinido.
- v. `PROCEDURE READ(f)` → Se encarga de leer el fichero (`var F: tipodeFichero`) y de asignar sus datos al conjunto de variables (`lista de variables`).
con la salvedad de que el (`var F: tipodeFichero`) es opcional, y en el caso de no especificarse explícitamente como parámetro se lee el fichero por defecto `input`.

Nota: (f) es equivalente: `begin read(ff, v1); begin read(ff, v2, ..., vn) end`

- vi. PROCEDURE WRITE(*f*) → Se encarga de escribir el fichero (var *F*: `tipodeFichero`) y de escribir en el mismo los datos de (lista de variables).

anterior, con la salvedad de que el (var *F*: `tipodeFichero`) es opcional, y en el caso de no especificarse explícitamente como parámetro se lee el fichero por defecto `input`.

Nota: (f) es equivalente: `begin write(ff, v1); begin write(ff, v2, ..., vn) end`

- vii. PROCEDURE NEW(*p*) → Reserva una variable *v* en memoria y asigna el puntero de *v* a *p*. El tipado de *v* viene dado explícitamente en la declaración de *p*.
- viii. PROCEDURE DISPOSE(*p*) → Libera el registro de memoria *v* asociado a *p*.

II. Funciones:

i. Funciones Aritméticas:

- a. FUNCTION ABS(*x*) → Se trata de un operador genérico para tipos de datos Entero y Real que a partir del parámetro (*x*:tipo) devuelve el valor absoluto de *x*.

Formalidad 3.3.11. FUNCTION ABS(*x*:*tipo*): *tipo*; $\equiv |x|$

- b. FUNCTION SQR(*x*) → Para el parámetro *x* de tipo INTEGER o REAL devuelve el valor de x^2 .

Formalidad 3.3.12. FUNCTION SQR(*x*:*tipo*): *tipo*; $\equiv x^2$

- c. FUNCTION SIN(*x*) → Para el tipo de datos REAL, devuelve el valor del seno del parámetro *x*.

Formalidad 3.3.13. FUNCTION SIN(*x*:*REAL*): *REAL*; $\equiv \text{sen}(x)$

- d. FUNCTION COS(*x*) → Para el tipo de datos REAL devuelve el coseno de *x*.

Formalidad 3.3.14. FUNCTION COS(*x*:*REAL*): *REAL*; $\equiv \text{cos}(x)$

- e. FUNCTION EXP(*x*) → Para el tipo de datos REAL devuelve el valor de e^x , siendo *x* el parámetro.

Formalidad 3.3.15. FUNCTION EXP(*x*:*REAL*): *REAL*; $\equiv e^x$

- f. FUNCTION LN(*x*) → Para el tipo de datos REAL devuelve $\text{Ln}(x)$, siendo *x* el parámetro.

Formalidad 3.3.16. FUNCTION LN(*x*:*REAL*): *REAL*; $\equiv \text{Ln}(x)$

- g. FUNCTION SQRT(*x*) → Para el parámetro *x* de tipo REAL devuelve el valor de \sqrt{x} .

Formalidad 3.3.17. FUNCTION SQRT(*x*:*REAL*): *REAL*; $\equiv \sqrt{x}$

- h. FUNCTION ARCTAN(*x*) → Para el tipo de dato Real devuelve el valor del arcotangente *x* en radianes.

Formalidad 3.3.18. FUNCTION ARCTAN(*x*:*REAL*): *REAL*; $\equiv \text{arctg}(x)$

ii. Funciones de Transferencia:

- a. FUNCTION TRUNC(*x*) → Para el tipo de dato REAL, obtiene la parte entera del parámetro *x*.

Formalidad 3.3.19. FUNCTION TRUNC(*x*:*REAL*): *LONGINT*; $\equiv \text{TRUNC}(a, b) = a$

- b. `FUNCTION ROUND(x)` → Para el tipo de datos `REAL`, redondea el parámetro `x` al valor entero mas próximo.

iii. Funciones Ordinales:

- a. `FUNCTION ORD(x)` → Para el tipo de datos `LONGINT` devuelve `true` en caso de que el parámetro `x` sea par. Siendo impar devuelve `false`.
- b. `FUNCTION CHR(x)` → Para el tipo de datos `BYTE` devuelve el carácter (ver tabla ASCII, Apéndice ...) del valor ordinal `x`.
- c. `FUNCTION SUCC(x)` → Para un valor ordinal devuelve el sucesor del parámetro `x`.
- d. `FUNCTION PRED(x)` → Para un valor ordinal devuelve el predecesor del parámetro `x`.

iv. Funciones Booleanas:

- a. `FUNCTION ODD(x)` → Devuelve si el valor `x` es par (`TRUE`) o IMPAR (`FALSE`).
Formalidad 3.3.20. `FUNCTION ODD(x:INTEGER): BOOLEAN; ≡ (abs(x) mod 2 = 1)`
- b. `FUNCTION EOF(f)` → Devuelve el valor `true` en caso de que sea final de fichero (el puntero de lectura/escritura se encuentra en el carácter de final de fichero). Caso contrario `false`.
- c. `FUNCTION EOLN(f)` → Devuelve el valor `true` en caso de que sea final de línea (el puntero de lectura/escritura se encuentra en el carácter de final de línea). Caso contrario `false`.

3.3.1.4. Estructura de un programa

Un programa en Pascal se divide en tres partes bien diferenciadas:

Programa 3.3.21. Plantilla de programa en Pascal

```

1 PROGRAM name (file-variables);
2 LABEL lab, lab, ... ;
3 CONST
4     name = CONSTANT;
5     {Other constant declarations}
6 TYPE
7     name = TYPE;
8     {Other type declarations}
9 VAR
10    name:TYPE;
11    {Other variable declarations}
12 {Procedure and Function declarations}
13 BEGIN
14     statements;
15 END.
```

- I. Program Heading (Cabecera del Programa): Se trata de una estructura clásica de Entrada/Salida del programa. Compuesto a su vez por:

- i. **program name**: Nombre principal y raíz del programa.
 - ii. **(file variables)** Lista de parámetros para enviar/recibir flujos de datos hacia en entorno de ejecución.
- II. **Declaration Part** (Apartado de Declaraciones): El bloque de declaración de programa es la estructura central aplicada y repetida a su vez, en los bloques PROCEDURE Y FUNCTION y se divide en los siguientes apartados:
- i. **label declaration** (sección de etiquetas).
 - ii. **(constant declaration)** (sección de constantes).
 - iii. **(type declaration)** (sección de estructuras de datos).
 - iv. **(variable declaration)** (sección de variables).
 - v. **(PROCEDURE | FUNCTION declaration)** (Declaración de Procedimientos o Funciones).
- III. **Statement Part** (Apartado de Sentencias): Se trata del bloque del programa principal que contiene el conjunto de sentencias y estructuras de control.

```

1 | BEGIN
2 |     statements ;
3 | END .

```

Programa 3.3.22. helloProgrammer.pas

```

1 | PROGRAM HelloProgrammer ;
2 | BEGIN
3 |     WRITELN('Hello Programmer!');
4 | END .

```

Notas sobre compilación: Para compilar el archivo fuente `helloProgrammer.pas` sobre GNU, usaremos el compilador GNU Pascal Compiler³⁰.

Las ordenes para compilarlo y ejecutarlo son las siguientes:

```

$ gpc -o helloProgrammer helloProgrammer.pas
$ ./helloProgrammer
Hello Programmer!

```

3.4. Evoluciones del Lenguaje Pascal

3.4.1. Modula/Modula-2

Modula-2 grew out of a practical need for a general, efficiently implementable systems programming language for minicomputers. Its ancestors are Pascal and Modula. From the latter it has inherited the name, the important module concept, and a systematic, modern syntax, from Pascal most of the rest. This includes in particular the data structures, i.e. arrays, records, variant records, sets, and pointers. Structured statements include the familiar if, case, repeat,

*while, for, and with statements. Their syntax is such that every structure ends with an explicit termination symbol.*³¹ [Wir80]

Modula/Modula-2 es un lenguaje multipropósito pensado originalmente para ser un lenguaje mucho más eficiente que Pascal. **De alguna manera Wirth ha buscado siempre la mejora en el rendimiento de todos sus lenguajes.**

Se presenta como un lenguaje basado en importantes conceptos de Programación Orientada a Objetos (POO), aunque no los implementa todos. La idea más destacada de POO en Modula/Modula-2 es la de encapsulación. Como sabemos crea una estructura de datos modular en base a unas propiedades y métodos (operaciones sobre los datos). La idea de cápsula³² deriva de que por sí las propiedades del módulo nunca son accesibles directamente como ocurre en la programación estructurada. Por contra, son los métodos (las acciones) las que dan acceso al contenido de estas variables bien sea para su lectura, escritura o ambas.

Como decimos, este es el aspecto más destacado de Modula/Modula-2 ya que, conceptos universales como la Herencia son inexistentes dentro del lenguaje. Dado que su propósito era más general y profesional que Pascal, Modula/Modula-2 tuvo cierto auge en entornos profesionales en los años ochenta del siglo XX. Una vez más sus “limitaciones por definición” lo han convertido en un “lenguaje para aprender a programar”.

3.4.1.1. Símbolos y Gramática

1. Tokens:

```
letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
        | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'
        | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' .
```

```
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
```

2. Símbolos especiales:

```
special-symbol = '+' | '-' | '*' | '/' | '=' | '<' | '>' | '[' | ']'
                | '.' | ',' | ':' | ';' | '^' | '(' | ')'
                | '<>' | '<=' | '>=' | ':=' | '..' | word-symbol .
```

```
word-symbol = 'and' | 'array' | 'begin' | 'case' | 'const' | 'div'
              | 'do' | 'downto' | 'else' | 'end' | 'file' | 'for'
              | 'function' | 'goto' | 'if' | 'in' | 'label' | 'mod'
              | 'nil' | 'not' | 'of' | 'or' | 'packed' | 'procedure'
              | 'program' | 'record' | 'repeat' | 'set' | 'then'
              | 'to' | 'type' | 'until' | 'var' | 'while' | 'with' .
```

3. Gramática: (ver Anexo C sección 2.)

Programa 3.4.1. helloProgrammer.mod

```
1 MODULE helloProgrammer;  
2  
3 FROM InOut IMPORT WriteString;  
4  
5 BEGIN  
6     WriteString('Hello programmer!');  
7 END helloProgrammer.
```

Notas sobre compilación: Para compilar el archivo fuente `helloProgrammer.adb` sobre GNU, usaremos el compilador GNU Modula-2³³.

Las ordenes para compilarlo y ejecutarlo son las siguientes:

```
$ gm2 -o helloProgrammer helloProgrammer.mod  
$ ./helloProgrammer  
Hello Programmer!
```

3.4.2. Ada

La idea conceptual del lenguaje de programación Ada nace por los enormes gastos que generaban compiladores, editores y otras herramientas de sistemas embebidos en el Departamento de Defensa de EEUU. En 1974 da comienzo un estudio que desvela el gran problema de tener un sistema de desarrollo muy caótico dónde eran frecuentes aplicaciones sobre un lenguaje determinado para un tipo de sistema concreto. Se llegó a la conclusión de que era necesaria una estandarización de desarrollo (lo que incluía un nuevo lenguaje general para estas ideas).

En 1975 se elaboró un documento técnico (Strawman) con las primeras especificaciones. Dicho documento sufrió distintas modificaciones gracias a la participación y comentarios de muchos desarrolladores. En el año 1976 se tenía una versión muy robusta del lenguaje que querían a nivel conceptual. Se hizo un concurso público para que distintas empresas dieran una forma Software a dichas especificaciones. Finalmente y tras un duro proceso de selección en 1979 se publicó ganadora la empresa *CII Honeywell Bull*. Al mismo tiempo se dio nombre al lenguaje que empezaba a ser una realidad. Se denominó Ada en honor a la primera programadora de la historia, Augusta Ada King (Ada Lovelace)³⁴, asistente y mecenas de Charles Babbage a su vez, creador de la primera máquina analítica.

En 1980 se publica la versión definitiva del lenguaje y es propuesta para su estandarización en ANSI. **En 1893 se tuvo la primera versión de Ada estándar (ANSI/MIL-STD 1815A) conocido como Ada 83.** El modelo se perfeccionó y por fin fue publicado por ISO (ISO-8652:1987).

El mismo Ada 83 desde el principio tuvo deficiencias prácticas por lo que se trabajó en una nueva versión (Ada 9X) que entre otras ideas, incorporaba el mecanismo de herencia. La nueva versión se llamó Ada 95 (ISO-8652:1995) y es uno de los estándares más usados hoy en día.

Ya en el año 2012, en el Congreso Ada-Europe celebrado en Stockholm, los organismos: Ada Resource Association (ARA) y Ada-Europe anunciaron el diseño de una nueva versión pendiente actualmente de aprobación por parte de ISO/IEC.

El lenguaje de programación Ada destaca sobre otros por lo siguiente:

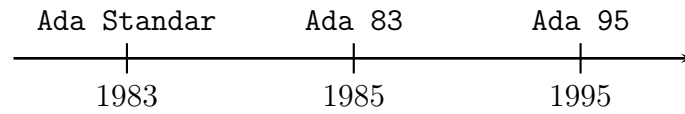


Figura 3.5: Evolución del Lenguaje Ada.

- i. Legibilidad: Ada en mayor medida que Pascal insiste en que los programas deben ser legibles (cualquier programador puede ser capaz de comprender el código fuente de un programa). Por ejemplo, un valor en aritmética flotante es expresado como: `(PART INT).(PART DECIMAL)`.
- ii. Tipificación fuerte: Es necesario y sabido que cualquier estado de datos (variable o propiedad) debe estar perfectamente tipificado es decir, desde el principio se debe aclarar a que familia de datos pertenece
- iii. Programación en gran escala: Es parte del concepto de ADA es ser un lenguaje que atienda a necesidades de computación masiva. Por ello se utilizan técnicas de encapsulación, unidades lógicas, parametrización de procesos con el objetivo final de que el programa sea siempre un conjunto de partes que se puedan auditar individualmente.
- iv. Manejo de excepciones: Las excepciones son imprescindibles para este tipo de lenguajes que puesto que trabajan muy cercanos al Hardware. ADA es un lenguaje ampliamente usado aplicaciones de alto rendimiento y es por esto, que hace muy necesario conocer como se comporta el programa en tiempo de ejecución.
- v. Unidades genéricas: Lo hemos comentado anteriormente y es que, la lógica de programación nos dice Divide and Conquer (D&C)³⁵ Tiende siempre a unidades lógicas (cápsulas) y funcionales. Con esto ADA siempre ha pretendido que los errores se corrijan desde lo más básico y procurando que afecte lo menos posible al resto de módulos.

Programa 3.4.2. helloProgrammer.adb

```

1 | with Ada.Text_IO; use Ada.Text_IO;
2 | procedure helloProgrammer is
3 | begin
4 |   Put_Line ("Hello programmer!");
5 | end helloProgrammer;
```

Notas sobre compilación: Para compilar el archivo fuente `helloProgrammer.adb` sobre GNU, usaremos el compilador GNAT (GNU NYU Ada Translator)³⁶.

Las ordenes para compilarlo y ejecutarlo son las siguientes:

```

$ gnatmake helloprogrammer.adb
$ ./helloprogrammer
```

```
Hello programmer!\begin{verbatim}
```

3.4.3. Oberon

*Oberon is a general-purpose programming language that evolved from Modula-2. Its principal new feature is the concept of type extension. It permits the construction of new data types on the basis of existing ones and to relate them.*³⁷ [Wir88]

Oberon es un lenguaje de programación orientado a objetos y procedimental creado por Niklaus Wirth (autor también de Pascal, Modula y Modula-2) y sus colaboradores del ETHZ (Suiza).

Oberon puede considerarse una evolución de Modula-2 con un soporte completo de orientación a objetos. De este lenguaje y de sus antecesores hereda buena parte de la sintaxis y de la filosofía. Wirth siempre ha intentado simplificar los lenguajes sin que por ello se pierda en potencia. También está diseñado con la seguridad en mente: tiene chequeos de rango en arrays, recolector de basura y es fuertemente tipado. Sin embargo, por su intento de simplicidad carece de enumeraciones y enteros restringidos en rango, los cuales pueden implementarse como objetos.

La sintaxis de orientación a objetos de Oberon no se parece a la de otros lenguajes más populares como C++ o Java, pero sí guarda similitud con la de Ada 95.

Oberon es también el nombre de un sistema operativo, escrito con y para este lenguaje. Oberon se ha portado a otros sistemas (incluyendo a Windows y sistemas Unix) e incluso se puede compilar en bytecodes para la máquina virtual de Java. También existe un proyecto para crear un compilador para la plataforma .NET.

3.4.3.1. Símbolos y Gramática

1. Tokens:
2. Símbolos especiales:
3. Gramática: (ver Anexo C sección 3.)

Programa 3.4.3. helloProgrammer.mod

```
1 MODULE Hello;
2     IMPORT Oberon, Texts;
3     VAR W: Texts.Writer;
4
5     PROCEDURE World*;
6     BEGIN
7         Texts.WriteString(W, "Hello World!");
8         Texts.WriteLine(W);
9         Texts.Append(Oberon.Log, W.buf);
10    END World;
11
12 BEGIN
13    Texts.OpenWriter(W);
```



```
14 | END Hello.
```

Notas sobre compilación: Para compilar el archivo fuente `helloProgrammer.adb` sobre GNU, usaremos el compilador `Oberon for GNU/Linux`³⁸.

Las ordenes para compilarlo y ejecutarlo son las siguientes:

```
$ helloprogrammer.mod
$ ./helloprogrammer
Hello programmer!
```


Notas

¹⁹El Lenguaje de Programación Pascal es descrito a partir del desarrollo de una versión de Algol 60. Comparado con Algol 60, el rango de aplicación es considerablemente mayor gracias a la variedad de estructura de datos. En principio es un intento para fundamentar las bases para enseñar programación con una herramienta eficiente para escribir grandes programas enfatizando en usar conceptos razonable sencillos, sistemáticos a la programación estructurada, y la eficiencia de la implementación. El compilador de es una sola pasada y ha sido construido para la familia CDC 6000.

²⁰El profesor Niklaus Wirth nació en Winterthur Suiza el 15 de febrero de 1934.

Se gradúa en 1959 como Ingeniero en Electrónica por la Escuela Politécnica Federal de Zúrich (ETH) en Suiza. Un año más tarde, se doctora (Ph.D.) en la Universidad de Berkley, California.

Trabajó a mediados de la década de los sesenta del siglo XX en la Universidad de Stanford y en la Universidad de Zúrich. Finalmente en 1968 se convierte en profesor de Informática en la ETH en Suiza.

²¹El lenguaje de programación FORTRAN intenta hacer posible la expresión de cualquier problema numérico. En particular, es ideal para formular conjuntos de múltiples variables, permitiendo que cualquier variable sea tratada desde un plano libre de contexto. Sin embargo, este tipo de ejercicios presenta inconsistencias con el repertorio de palabras de cada máquina de computo y su lógica numérica, lo que puede llevar a problemas a la hora de expresar algunos problemas numéricos. Mucha de la lógica que emplea FORTRAN no es directamente expresable por lo que se puede obtener incorporando bibliotecas.

²²La relación entre mayúsculas () y minúsculas () se delega en el fabricantes del compilador.

²³<http://gcc.gnu.org/fortran/>

²⁴El propósito de ALGOrithmic Lenguaje es la de describir procesos computacionales. El concepto básico usado en la descripción de reglas de cálculo es conocido expresiones aritméticas constituidas por: números, variables y funciones. Las expresiones son compuestas aplicando reglas de composición aritmética, constituyendo unidades del lenguaje, explícitamente formuladas, llamadas asignación de recursos.

²⁵<http://www.gnu.org/software/marst/>

²⁶Single-Pass.

²⁷British Standards Institution: <http://www.bsigroup.com/>

²⁸Standards Planning and Requirements Committee

²⁹Technical Advisory Group: <http://technicaladvisorygroup.com/>

³⁰ <http://www.gnu-pascal.de/gpc/>

³¹Modula-2 en un lenguaje de programación de propósito general, eficientemente implementado para para minicomputadoras. Sus antecesores son Pascal y Modula. Suscrito a su nombre está el concepto de módulo y el trato sistemático con una sintaxis moderna, por ejemplo: matrices, registros, registros variables, conjuntos y punteros. Sus estructuras incluye los familiares: if, case, repeat, while y símbolos de terminación explícita.

³²El concepto de cápsula como unidad estructural se debe a David Lorge Parnas (Canada, 10 de Febrero de 1941).

David Lorge Parnas es una de las figuras más representativas de la Ingeniería del Software. Graduado en Ingeniería (especialidad en Electricidad) a lo largo de su carrera ha sido profesor en Universidades como: Universidad de Carolina del Norte (EEUU), Universidad de Victoria (Canada), Universidad de Limerick (República de Irlanda) entre otras.

De todas sus aportaciones a las Ciencias de la Computación destaca su Diseño Modular donde establece el concepto de “Cápsula de Datos” como abstracción de un objeto en la vida real con una serie de propiedades y acciones.

³³<http://www.nongnu.org/gm2/homepage.html>

³⁴Augusta Ada King (Londres 10 de Diciembre de 1815, Londres, 27 de Noviembre 1852) es considerada la primera programadora de máquinas de la historia.

Hija del famosos poeta George Byron, su formación giró en torno a las Matemáticas y la Lógica. A pesar de ello, tuvo igualmente un interés notorio en las disciplinas humanísticas. Entro en el mundo de la programación gracias a su compañero y amigo Charles Babbage (creador de la Máquina Analítica Babbage).

En su obra *Notas* describe la Máquina Analítica y desarrolla un conjunto de instrucciones para realizar cálculos. Igualmente estableció las tarjetas perforadas como método para almacenar datos.

Anotar finalmente, que la Máquina Analítica de Charles Babbage nunca llegó a construirse por lo que Ada realizó todo su trabajo desde un punto de vista formal y lógico.

³⁵Divide y Vencerás.

³⁶<http://www.adacore.com/>

³⁷Oberon is un lenguaje de propósito general basado en Modula-2. Es principalmente nueva la característica de extensión de tipo.- Esto permite la construcción de nuevos tipos de datos basados en otros existentes.

³⁸<http://olymp.idle.at/tanis/oberon.linux.html>

Capítulo 4

Compiladores del Lenguaje Pascal

Resumen:

4.1. Pascal User's Group (PUG)	69
4.2. Pascal-P (The Portable Pascal Compiler)	70
4.3. UCSD Pascal	71
4.4. Pascaline	74
4.5. Borland Pascal	74
4.6. GNU Pascal Compiler (GPC)	77
4.7. FreePascal	78
Notas	81

4.1. Pascal User's Group (PUG)

*"This is the first issue of a newsletter sent to users and other interested parties about the programming language PASCAL. Its purpose is to keep the PASCAL community informed about the efforts of individuals to implement PASCAL on different computers and to report extensions made o the language. It will be published at infrequent intervals due to the limited manpower..."*³⁹

George H. Richmond. 1974 (Newsletter #1)

4.1.1. Historia

La Revista PUG fue publicada entre Enero de 1974 y Noviembre de 1983. Durante su actividad resultó un importante soporte para la evolución del Lenguaje Pascal.

Entre otros aspectos, se trató la estandarización de Pascal, la generación de compiladores base como P4 y aspectos de la evolución que sufría la computación en la década de los setenta.

Resalta el hecho, de que es sus últimas publicaciones se nombra un nuevo Lenguaje en desarrollo, ADA.

UCSD Pascal fue duramente criticado dado que era un proyecto que se ajustaba a las bases "de facto" de PUG.

El estándar propio de Pascal (propuesto por Tony Addyman) resultó ser la última gran disputa entre PUG y los institutos ANSI e ISO.

4.2. Pascal-P (The Portable Pascal Compiler)

El equipo de Wirth en la Universidad de Zurich creó dos familias de compiladores:

- i. CDC 6000: Código nativo para las propias máquinas CDC 6000. Se trataban de compiladores de una pasada que traducían el código fuente a código máquina directamente. Usaban o “Full Pascal”.
- ii. Pascal-P: Enfocado a la portabilidad y compatibilidad. Su idea era crear compilador/intérprete capaz de generar código intermedio para que después, sobre una arquitectura en concreto, se generase el ejecutable.

4.2.1. Historia Pascal CDC 6000

Fue implementado en Scallop (Lenguaje propio de las máquinas CDC) entre los años 1970 y 1971. Hubo también un intento de desarrollar el mismo compilador en Lenguaje Fortran pero debido al uso de que hacía el Lenguaje Pascal de estructuras recursivas, hizo imposible la tarea.

En el año 1972 Wirth y su equipo trabajan en una revisión del Lenguaje Pascal, un subconjunto del original ya que, se trabajaba intensamente en la idea de un compilador independiente de una arquitectura en concreto. La primera versión de Pascal Portable, P1 usaba la máquina “Stack” o pseudo-machine. Se trató de un prototipo que convivió con las versiones de CDC 6000.

4.2.2. Historia Pascal-P

Versión	Origen	Año	Hito
Pascal P1	Zurich	1973	Concepto de portabilidad entre arquitecturas.
Pascal P2	Zurich	1974	Implementación de “Full Pascal” y primer Compilador.
Pascal P3	Zurich	1976	Paso previo entre P2 y P4.
Pascal P4	Zurich	1976	“Estándar de facto” y base para UCSD Pascal.
Pascal P5	San Jose	2009	Compatible con ISO 7185 .
Pascal P6	Comunidad	En Desarrollo	Implementación de ISO 10206 y Pascaline.

Tabla 4.1: Versiones de Pascal-P.

- I. Pascal P2: Publicado en 1974, se trataba de una versión real del lenguaje. Fue acompañada de una revisión integral de “Full Pascal” o también llamado Pascal 1971. Sobre P2 se derivaron importantes compiladores como: UCSD Pascal a la vez que sirvió de prototipo para Borland Turbo Pascal.

- II. Pascal P3 y P4: Esta versión es la más importante de toda la familia dado que, aún hoy día sobrevive y es matriz para desarrollar nuevos compiladores. Data de 1976 aunque ha sido plenamente adaptada al estándar ISO Pascal 7185:1990. Decir que fue acompañada de una versión P3 que trataba de ser un intermediario entre P2 y P4, fue una implementación hipotética debido a que P4 se convirtió en el “estándar de facto”.
- III. Pascal P5: Dado los problemas de memoria sobre los que se desarrolló la versiones previas, en 2009 se propuso una revisión de Pascal-P4 que tiene como objetivo principal (sigue en desarrollo) mejorar el rendimiento y ser plenamente compatible con ISO 7185.
- IV. Revisiones:
 - i. Pascal P6: Pretendía implementar la versión extendida de Pascal. Finalmente se decidió desarrollar como una versión de Pascaline que añade a la ISO mecanismo de la Programación Paralela y Distribuida.
 - ii. Pascal P7: Hipotética versión exclusiva para Pascal Extendido. No se ha llegado a codificar debido a que dicha ISO es parte del proyecto P6.

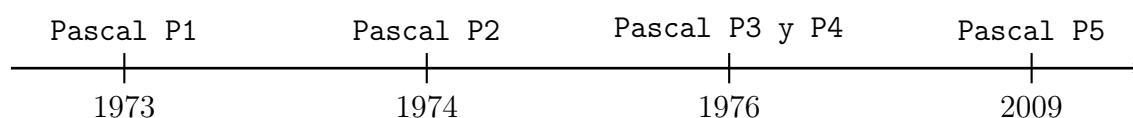


Figura 4.1: Evolución de Portable Pascal.

4.3. UCSD Pascal

UCSD Pascal o también, University of California, San Diego Pascal se trata de una revisión de Pascal-P2 que supuso una importante evolución conceptual en los lenguaje de programación.

Su característica más destacada era que uso instrucciones **p-code** con el propósito de ser multiplataforma, idea que era realidad a finales de los años setenta y que es parte hoy día de importantes Lenguajes de Programación como Java.

4.3.1. Historia

La idea original es de Kenneth Bowles quien en 1974 se percató de la gran cantidad de arquitecturas que existían y la incompatibilidad entre ellas. La síntesis de su idea era crear un dialecto de Pascal-P2 para que generase en la compilación el p-code que era fácilmente portable entre distintas arquitecturas con base en **p-code Operating Systems**.

La disputa surge por IBM y su política de instalaciones base, en concreto se ofrecía UCSD p-System, PC-DOS y CP/M-86 pero el rendimiento era muy distinto para los modelos de Hardware de la época. Por ello se ideó UCSD Pascal basado en una arquitectura p-code. El sistema se pasó a llamar The UCSD Pascal p-Machine que ya en sus orígenes era compatible para distintas máquinas.

Su estructura de compilación puso de base la necesidad de unidades de código (UNITS) y el uso de cadenas (STRING).

UCSD Pascal ha tenido cuatro versiones:

- I. Versión 1.0: Primer Software Base que fue distribuido junto al código fuente. Esta versión fue mejorada por los propios usuarios y derivaron en gran cantidad de mejoras.
- II. Versión 2.0: Revisión que trajo consigo compatibilidad con numerosas arquitecturas como: Apple II, DEC PDP-11, Zilog, MOS 6502, Motorola 68000 y primeros IBM-PC.
- III. Versión 3.0: Escrita desde Western Digital era parte de Pascal MicroEngine.
- IV. Versión 4.0: Desarrollada por SofTech, era una versión comercial orientada a la industria del desarrollo. Finalmente y tras ser improductiva paso a manos de Pecos Systems que a su vez estaba formada por entusiastas de p-System.

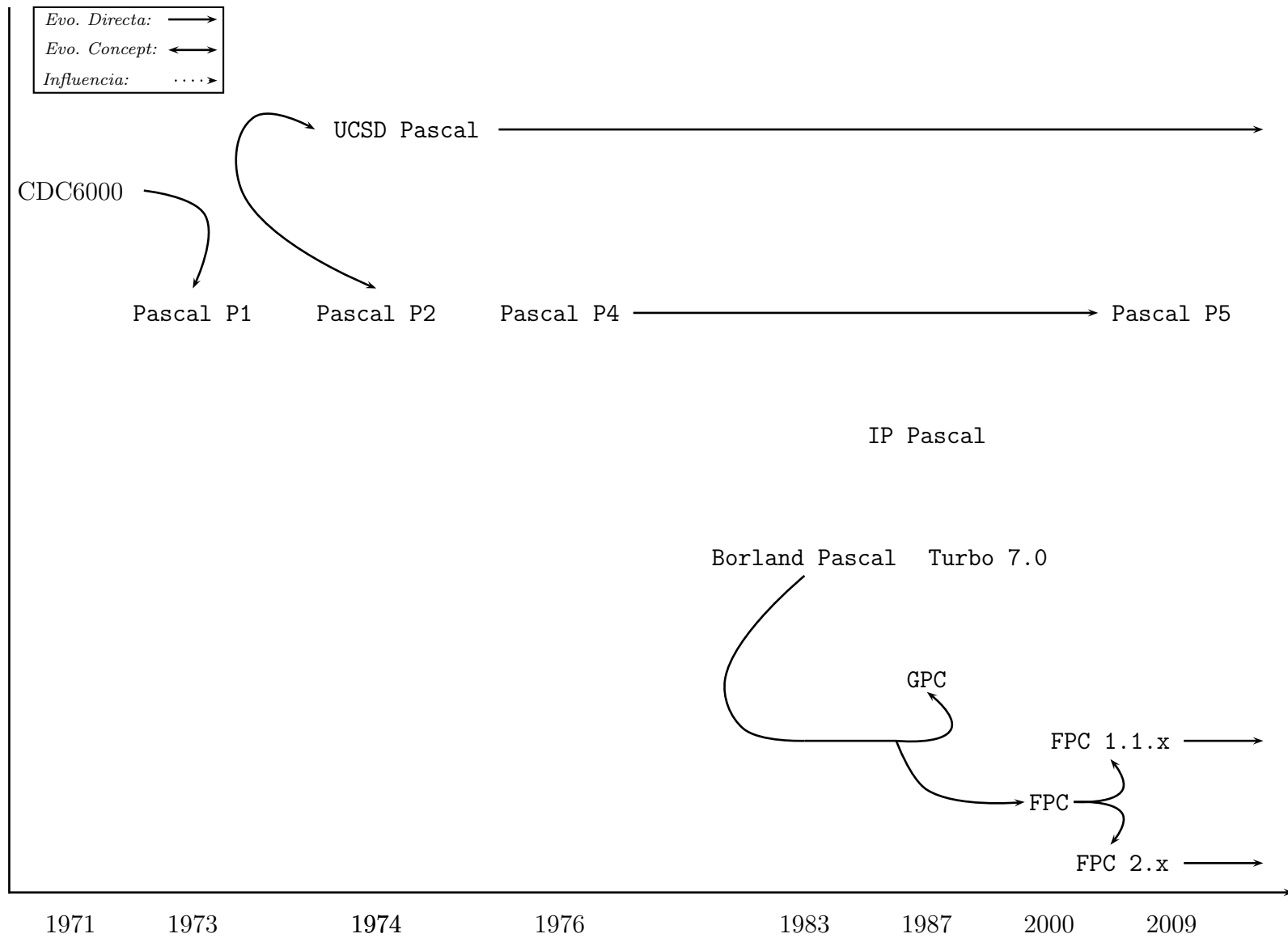


Figura 4.2: Evolución de compiladores para Pascal.

4.4. Pascaline

El dialecto Pascaline (Calculadora de Pascal) implementa el estándar ISO 7185 además de incorporar importantes funcionalidades como: Conceptos de Programación Orientada a Objetos, Arrays dinámicos o Monitores.

4.4.1. IP Pascal

IP Pascal Se trata un conjuntos de programas: IDE (Entorno de Desarrollo), compilador y codificador.

A lo largo de su desarrollo ha sufrido importantes mejoras y usando distintas plataformas de ejecución como:

- i. Z80: La implementación original (1980) fue escrita en Lenguaje Ensamblador de la propia máquina Z80. En 1985, IP Pascal fue completamente transcrito al propio Lenguaje Pascal. Ya en 1987, sufrió un importante cambio estructural tomando como base el Lenguaje C, dado que se estaba preparando la versión i386.
- ii. i386: Evolucionó a lo largo de las distintas versiones, donde originalmente se usaba código intermedio para IBM-PC. En 1994 se añadieron las funcionalidades de **Extended ISO 7185 Pascal**.
- iii. GNU/Linux: Creada en el año 2000 por la empresa Red Hat para su uso exclusivo a través de línea de comandos. En su diseño se utilizó un sistema GNU (glibc) y Syscalls para núcleo Linux.

4.5. Borland Pascal

Turbo Pascal se trata de un paquete Software compuesto por un compilador y un entorno de desarrollo (IDE).

El compilador fue desarrollado por la empresa Borland y que, gozó de gran popularidad a principios de los noventa del siglo XX dado su precio y compatibilidad con MS-DOS.

4.5.1. Historia

El desarrollo de Turbo Pascal estuvo liderado por Philippe Kahn, quien sentó su bases de su diseño. Entre sus hitos, destacan el de integra el proceso de: edición, compilación y enlazado. Por aquella época, era el propio programador y de manera explícita el que realizaba estas tareas. El concepto de “Kit de Desarrollo” unido a su precio de venta fueron los factores determinantes en su popularidad a los largo de mediados de los años ochenta y años noventa del siglo XX.

Su primera versión se basó Blue Label Pascal⁴⁰. Turbo Pascal se lanzó al mercado como Compas Pascal para CP/M con otras arquitecturas desarrolladas como: Apple II, máquinas DEC, CP/M-86 y MS-DOS. Su precio de mercado era de 49.99 USD. Hablamos del año 1983, por aquel entonces el Software y en particular los compiladores tenían precios mucho más elevados. Otro hito importante es que poco después fue lanzado la computadora personal IBM PC, dónde el propio compilador ofrecía resultados sorprendentes de rendimiento para estas máquinas tan limitadas.

Las versiones 2 y 3 del compilador ofrecieron cambios discretos, haciendo énfasis en la gestión de la memoria.

Por contra la versión 4 lanzada en 1987, fue prácticamente reescrita desde cero. Las versiones de 5 a 7 siguieron en la línea de añadir nuevos complementos al Software.

4.5.2. Valores internos para datos numéricos simples

I. Tipo Entero:

- i. SHORTINT: $[-128, 127]$ (1 Byte)
- ii. INTEGER: $[-32768, 32767]$ (2 Bytes)
- iii. LONGINT: $[-2147483648, 2147483647]$ (4 Bytes)
- iv. BYTE: $[0, 255]$ (1 Bytes)
- v. WORD: $[0, 65535]$ (2 Bytes)

II. Tipo Real:

- i. REAL: $[2.9 \cdot 10^{-39}, 1.7 \cdot 10^{38}]$ (de 11 a 12 dígitos representables, 6 Bytes)
- ii. SINGLE: $[1.5 \cdot 10^{-45}, 3.4 \cdot 10^{38}]$ (de 7 a 8 dígitos representables, 4 Bytes)
- iii. DOUBLE: $[5.0 \cdot 10^{-324}, 1.7 \cdot 10^{308}]$ (de 15 a 16 dígitos representables, 8 Bytes)
- iv. EXTENDED: $[1.9 \cdot 10^{-4851}, 1.1 \cdot 10^{4932}]$ (de 19 a 20 dígitos representables, 10 Bytes)
- v. COMP: $[-9.2 \cdot 10^{18}, 9.2 \cdot 10^{18}]$ (de 19 a 20 dígitos representables, 8 Bytes)

4.5.3. Biblioteca estándar

I. Procedimientos Estándar de Turbo Pascal (Descritas en el apartado 3.3.1.3):

- i. PROCEDURE APPEND(*var F:Text*); → Abre el archivo determinado como parámetro (*var F:Text*) para escribir a partir del final del fichero.
- ii. PROCEDURE DISPOSE(*var P:Pointer*); → Se encarga de liberar la memoria asignada al puntero (*var P:Pointer*).
- iii. PROCEDURE NEW(*var P:Pointer*); → Reserva memoria para el puntero (*var P:Pointer*).
- iv. PROCEDURE READ(*var F: tipodeFichero; {lista de variables}*); → *idem*.
- v. PROCEDURE READ([*var F: tipodeFichero;*] *{lista de variables}*); → *idem*.
- vi. PROCEDURE READLN([*var F: ficherodeTexto;*] *{lista de variables}*); → *idem* para la utilización de parámetros con el procedimiento anterior, con la salvedad de que se lee toda una línea del fichero, con el consiguiente avance del puntero de lectura.
- vii. PROCEDURE RESET(*var F: tipodeFichero*); → *idem*.
- viii. PROCEDURE REWRITE(*var F: tipodeFichero*); → *idem*.
- ix. PROCEDURE WRITE(*var F: tipodeFichero; {lista de variables}*); → *idem*
- x. PROCEDURE WRITE([*var F: tipodeFichero;*] *{lista de variables}*); → *idem*

- xi. PROCEDURE WRITELN(*[var F: fichero de Texto;]* {lista de variables}); $\rightarrow idem$ para la utilización de parámetros con el procedimiento anterior, con la salvedad de que se escribe toda una línea del fichero, con la consiguiente marca del puntero de escritura.

Función	Simbología
FUNCTION ABS	$ x $
FUNCTION ARCTAN	$arctg(x)$
FUNCTION COS	$cos(x)$
FUNCTION EXP	e^x
FUNCTION LN	$Ln x$
FUNCTION SIN	$sen(x)$
FUNCTION SQR	x^2
FUNCTION SQRT	\sqrt{x}
FUNCTION TRUNC	$TRUNC(a, b) = a$

Tabla 4.2: Relación entre la Biblioteca Estándar de Pascal y el Cálculo Matemático.

II. Funciones Estándar de Turbo Pascal:

- i. FUNCTION ABS(*x: tipo*): tipo; $\rightarrow idem$
- ii. FUNCTION ARCTAN(*x: REAL*): REAL; $\rightarrow idem$
- iii. FUNCTION CHR(*x: BYTE*): CHAR; $\rightarrow idem$
- iv. FUNCTION COS(*x: REAL*): REAL; $\rightarrow idem$
- v. FUNCTION EOF(*var F: tipo de Fichero*): BOOLEAN; $\rightarrow idem$
- vi. FUNCTION EOLN(*var F: tipo de Fichero*): BOOLEAN; $\rightarrow idem$
- vii. FUNCTION EXP(*x: REAL*): REAL; $\rightarrow idem$
- viii. FUNCTION LN(*x: REAL*): REAL; $\rightarrow idem$
- ix. FUNCTION ODD(*x: LONGINT*): BOOLEAN; $\rightarrow idem$
- x. FUNCTION ORD(*x: tipo Ordinal*): LONGINT; $\rightarrow idem$
- xi. FUNCTION PRED(*x: tipo Ordinal*): tipo Ordinal; $\rightarrow idem$
- xii. FUNCTION ROUND(*x: REAL*): LONGINT; $\rightarrow idem$
- xiii. FUNCTION SIN(*x: REAL*): REAL; $\rightarrow idem$
- xiv. FUNCTION SQR(*x: tipo*): tipo; $\rightarrow idem$
- xv. FUNCTION SQRT(*x: REAL*): REAL; $\rightarrow idem$
- xvi. FUNCTION SUCC(*x: tipo Ordinal*): tipo Ordinal; $\rightarrow idem$
- xvii. FUNCTION TRUNC(*x: REAL*): LONGINT; $\rightarrow idem$

Nombre	Instrucciones	Código Binario	Distribuido	IDE	Multiplataforma
CDC 6000	Full	Si	No	No	No
Pascal P1	1971	Ø	Ø	Ø	Ø
Pascal P2	1971	No	No	No	Si
Pascal P3	1971	Ø	Ø	Ø	Ø
Pascal P4	1971	No	No	No	Si
Pascal P5	ISO 7185	No	No	No	Si
Pascal P6	ISO 10206	Ø	Ø	Ø	Ø
UCSD Pascal	1971	No	No	No	Si
Pascaline	ISO 7185	Si	Si	Ø	Si
IP Pascal	ISO 7185	Si	Si	No	Si
Borland Pascal	Borland	Si	No	Si	Si
GPC	ISO 7185	No	No	No	Si
FPC	FPC	Si	Si	Si	Si

Tabla 4.3: Comparativa entre compiladores de Pascal.

4.6. GNU Pascal Compiler (GPC)

4.6.1. ¿Qué es GPC?

GPC (GNU Pascal Compiler) se trata de un compilador del lenguaje de programación Pascal perteneciente a la familia de compiladores de GNU GCC. Su primeras versiones datan de 1987. El compilador GPC se presenta como un autómata portable, rápido y flexible.

Es compatible con la ISO 7185 de Pascal e incorpora soporte para la ISO 10206 de Pascal Extendido.

Durante el año 2010 el grupo de desarrolladores debatió en torno al hipotético futuro de compilador y su integración con el “Front-End”⁴¹ de GCC. Finalmente en Julio de 2013 se congeló su desarrollo.

4.6.2. Estructura de GPC

Su portabilidad se basa en su estructura motor, es decir, en las herramientas con las que se ha creado.

- i. Flex: Determina utilizando expresiones regulares la pertenencia o no de a palabra al alfabeto (ver Apartado 5.2.1) Σ ⁴².
- ii. Bison: Trata la sintaxis en base a las especificaciones BNF⁴³.
- iii. Interfaz GAS: GNU Assembler (más conocido como GAS) se trata del ensamblador oficial del proyecto GNU. Es el “Back-End” para GCC. Se distribuye en el metapaquete Software Binutils. Actualmente tiene la licencia GPL v.3.0.

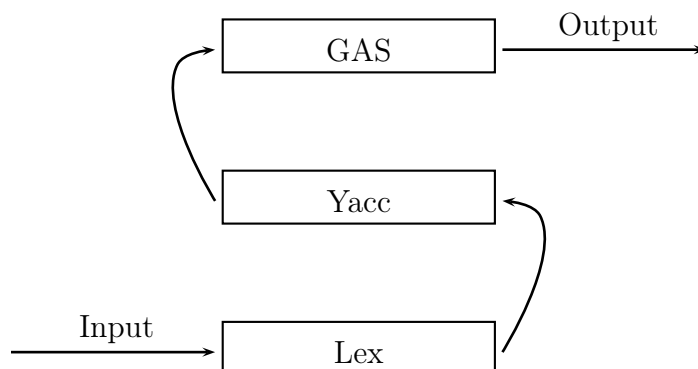


Figura 4.3: Arquitectura de GPC.

4.7. FreePascal

4.7.1. ¿Qué es FreePascal?

Inicialmente se conocía como FPC (Florian Paul Klämpf) acrónimo del propio autor.

Actualmente FPC está compuesto por: el propio Compilador, un conjunto de Bibliotecas y un IDE (Lazarus).

4.7.2. Historia

Su desarrollo comienza tras el anuncio de Borland en relación al abandono de su su familia de compiladores Borland Pascal (su sucesor natural sería Delphi).

Las primeras versiones fueron escritas por Florian Paul Klämpf en el propio dialecto de Borland Pascal. Del mismo modo, sus primeros ejecutables fueron para MS-DOS de 16 bits aunque, dos años después soportaba distintas arquitecturas de 32 bits.

4.7.2.1. Versiones

- I. Rama 0.x: La versión de 32 bits fue distribuida a través de Internet. Se hizo compatible con GNU/Linux y OS/2. 0.88.5 se convirtió en el primer producto estable de PFC. A pesar de esto, la mejora posterior (0,99,8) se hizo plenamente compatible con Win32 y añadía gran parte de las Bibliotecas de de Delphi.
- II. Rama 1.x: La primera versión estable de esta rama fue lanzada en Julio del año 2000 a la que siguió la 1.0.10 de Julio de 2003 donde se insistió en la corrección de errores. La misma se hizo compatible con procesadores de 68K, hecho que dejó palpables las notables deficiencias en el diseño del propio compilador.

Por ello se tomó la decisión de la reescritura del mismo con el claro objetivo de la limpieza del código y la idea de ser plenamente compatible con distintas plataformas.

Entre Noviembre de 2003 y principios de 2003 el nuevo diseño fue tomando forma y finalmente fue presentada como FPC 1.9.0 compatible para:

- i. Por Arquitectura: x86 y amd64 , Porwer-PC, ARM y Sparc v.8 y v.9.
 - ii. Por Sistema Operativo Base: Win2K y MS-DOS, GNU/Linux, Mac OS X, FreeBSD, OS/2.
- III. Rama 2.2.x: La motivación de está versión venía dada por que Lazarus necesitaba soporte pleno para: Win64, Windows CE y Mac OS X en x86. La primera versión estable se publicó en Septiembre de 2007 (2.2.2). Además se incorporó en lo sucesivo soporte para Active X/COM y OLE que lo convertía en un producto maduro para plataformas Win2k.
- IV. Rama 2.4.x: La versión 2.4 de FPC trajo consigo importantes cambios en el diseño del compilador. De nuevo la portabilidad fue el aspecto más relevante y en el que mayor esfuerzo realizó el equipo de desarrolladores- Las nuevas plataformas soportadas fueron:
- i. Mac PowerPC 64 y x86-amd64.
 - ii. iPhone.
 - iii. ARM.

Se añadió también soporte para Delphi y se reescribió “Unit System”.

- V. Rama 2.6 y 2.7: El lanzamiento en Enero de 2014 de PFC 2.6 aportó el soporte pleno del compilador en Mac OS X. La revisión 2.7 (actualmente en desarrollo) incorpora gran cantidad de cambio en el núcleo del compilador:
- i. Soporte para ISO 7185 y capacidad de compilar código de P4.
 - ii. Soporte para Delphi (aspectos avanzados de POO).
 - iii. Soporte para las arquitecturas y SSOO: MIPS, NetBSD, OpenBSD, AmigaOS (m68k) y JVM (algunas primitivas).

4.7.3. Estructura de FreePascal

El compilador FPC se divide en tres partes bien diferenciadas (siguiendo el esquema propio de un compilador):

- I. Analizador Léxico (*Scanner/Tokenizer*): El escáner (LEX) analiza el flujo de entrada de datos y prepara la lista de tokens que a su vez será utilizado por el *Paser*. Es el estado donde se analizan las directivas del Preprocesador. A su vez se divide en las siguientes unidades:
- i. Flujo de Entrada (*Input Stream*): Se encarga de normalizar el método de entrada (I/O) al fichero llavero `file.pas`
 - ii. Preprocesador: El escáner resuelve todas las directivas del preprocesador en el código fuente del programa. Se encarga de transformar dichas operaciones en sentencias de Pascal.

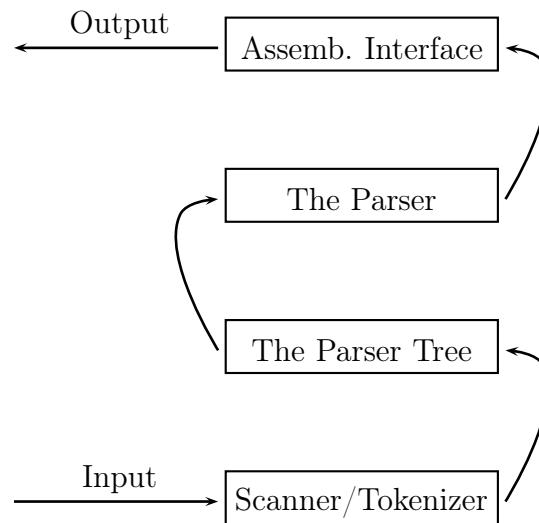


Figura 4.4: Arquitectura de FPC.

- II. Árbol Sintáctico (*The Parse Tree*): El árbol es la base del compilador. Tras el desglose de las sentencias y bloques, el código es traducido a una estructura de datos tipo Árbol *The Parse Tree*.
- III. Analizador Sintáctico (*Parser*): La tarea del *Parser* (Yacc) es la de analizar el flujo de tokens generado por el (*Scanner*) y comprobar que tiene un orden lógico es decir, que se ajustan a la sintaxis del lenguaje.

El mismo *Parser* utiliza una tabla de símbolos y genera un árbol de nodos para la interfaz de *Assembler*.

- IV. Generador de Código (*The Code Generator*): La interfaz *The Code Generator* es la encargada de generar el Output para el Lenguaje Ensamblador y posteriormente el enlace con las bibliotecas del SSOO.

En la versión 1.0 de FPC establecía código intermedio por cada nodo tras el primer análisis. A su vez se asociaba con las rutinas en código ensamblador tras la “segunda pasada” y finalmente generaba las instrucciones en Ensamblador.

Notas

³⁹“Este es el primer número de un boletín enviado a los usuarios y otras partes interesadas sobre el lenguaje de programación PASCAL. Su propósito es mantener a la comunidad informada sobre PASCAL los esfuerzos de las personas para poner en práctica PASCAL en equipos diferentes y que informe extensiones hechas o el idioma. Se publicará a intervalos poco frecuentes debido a la mano de obra limitada...”

⁴⁰Desarrollado por NasSys.

⁴¹El problema y motivo de la discusión era por la reimplementación del código intermedio, en este caso C++ a C.

⁴²Para la versión: <http://www.gnu-pascal.de/alpha/gpc-20060325.tar.bz2> el fichero `pascal-lex.l`

⁴³Para la versión: <http://www.gnu-pascal.de/alpha/gpc-20060325.tar.bz2> el fichero `parse.y`

Parte II

gp19901a (Analizador Léxico)

XI

Yo sé un himno gigante y extraño
que anuncia en la noche del alma una aurora,
y estas páginas son de ese himno
cadencias que el aire dilata en las sombras.

Yo quisiera escribirle, del hombre
domando el rebelde mezquino idioma,
con palabras que fuesen a un tiempo
suspiros y risas, colores y notas.

Pero en vano es luchar; que no hay cifra
capaz de encerrarle, y apenas ¡oh! ¡hermosa!
si teniendo en mis manos las tuyas
pudiera al oído cantártelo a solas.

Gustavo Adolfo Bécquer

Capítulo 5

Formalidades del Analizador Léxico

Resumen:

5.1. Introducción	87
5.2. Teoría de Lenguajes	89
5.3. Lenguajes Regulares	94
5.4. Expresiones Regulares	95
5.5. Autómatas	96
5.6. El Lenguaje LEX	103
5.7. Código fuente: gp1990la.l	104
Notas	107

5.1. Introducción

Definición 5.1.1. La función de un Analizador Léxico es la de leer el/los archivo/s de código fuente de un programa para relacionarlos con los Lexemas del lenguaje, producir los Tokens e informar de errores a nivel de lectura.

Definición 5.1.2. Un Lexema es un conjunto de uno o más caracteres que agrupan una determinada secuencia de caracteres del código fuente.

Nota: Los ejemplos se basan en Pascal ISO 7185:1990

Ejemplo 5.1.3. ‘99’ | ‘3.14’ | ‘IF’ | ‘q’ | “Hellow Programmer”

Definición 5.1.4. Un Token es un identificador para un determinado Lexema.

Ejemplo 5.1.5. Para el ejemplo anterior y en secuencia: INTEGER | FLOAT | WORD-SYMBOL | CHAR | STRING

Definición 5.1.6. Un Patrón es una descripción formal de un Lexema (Ver Tabla 5.1)

Definición 5.1.7. A la hora de trabajar con Errores Léxicos debemos tener en cuenta como queremos que sea de restrictivo nuestro Analizador. A grandes rasgos podemos establecer las siguientes categorías:

Lexema	Patrón	Token
Identificador	$[A - Z, a - z, 0 - 9]^*$	"tokIdent"
Entero	$[0 - 9]^*$	"tokInteger"
Real	$[0 - 9].[0 - 9]^*$	"tokReal"

Tabla 5.1: Ejemplo de relación entre: Lexema, Patrón y Token.

- i. Analizadores "Modo Pánico"
- ii. Analizadores con Recuperación

La idea de esta división en la arquitectura radica en lo que conocemos como "modo pánico" es decir, ante una palabra del Lenguaje que o bien no pertenece al mismo o bien, genera ambigüedad.

El modelo mayormente aceptado es el primero Analizadores "Modo Pánico", ante un error léxico, se termina el programa e informa al usuario/programador con el mayor detalle posible de dónde y por qué existe dicho error.

Por contra, la segunda categoría que hemos establecido: Analizadores con Recuperación asume que ante el error léxico se debe actuar de manera lógica e intentar que el análisis no se detenga.

Es lógico entonces, dotar a este programa de cierta "inteligencia sintáctica".

Las técnicas más comunes para recuperar el análisis son:

- i. Borrar un carácter extraño.
- ii. Reemplazar/Sustituir caracteres.

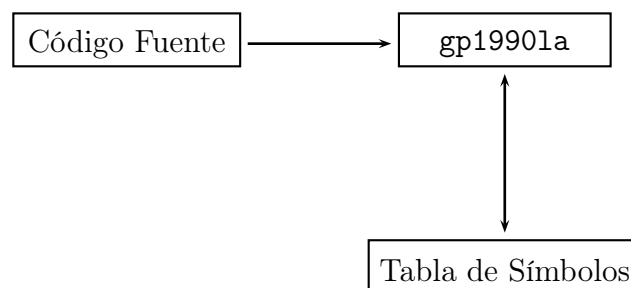


Figura 5.1: Relación entre el Analizador Léxico y el Programa Fuente.

5.2. Teoría de Lenguajes

5.2.1. Definiciones

Definición 5.2.1. Decimos que un **Alfabeto** es un conjunto de elementos finito y no vacío denotado como Σ .

Corolario 5.2.2. Cada uno de estos elementos recibe el nombre de **Símbolo**.

Definición 5.2.3. Existe una palabra común a todos los alfabetos que recibe el nombre de **Palabra Vacía**, denotada como: λ .

Corolario 5.2.4. El conjunto de todas las palabras de un alfabeto (incluida la palabra vacía) se denota como: Σ^* .

Ejemplo 5.2.5. El Alfabeto $\Sigma = \{0, 1\}$ es un conjunto de símbolos que a su vez forma parte de todas las palabras que utiliza cualquier sistema binario (por ejemplo un Disco Compacto).

Ejemplo 5.2.6. El Alfabeto $\Sigma = \{a, b, c, \dots, z\}$ es un conjunto de símbolos que a su vez forma parte de todas las palabras que utiliza la Lengua Castellana (que es un Lenguaje Humano).

5.2.2. Palabras

Definición 5.2.7. Decimos que una Palabra es una concatenación de símbolos de un Alfabeto dado.

$$v \in \Sigma \Leftrightarrow v_i \in \Sigma \quad (5.1)$$

5.2.2.1. Operaciones

Nota: Se trabajan con dos palabras: $u = [hola]$ y $v = [Mundo]$. La palabra vacía se denotara con el símbolo $\lambda \Rightarrow |\lambda| = 0$

- I. Concatenación: Entendemos que para un alfabeto dado Σ^* y dos palabras $u = \{u_1, u_2, \dots, u_n\}$ y $v = \{v_1, v_2, \dots, v_m\} \in \Sigma^*$ con $n, m \in \mathbb{N}$ por **concatenación** (denotado como \circ):

$$u \circ v = \{u_1, u_2, \dots, u_n, v_1, v_2, \dots, v_m\} \quad (5.2)$$

Ejemplo 5.2.8. Para las palabras $u \wedge v$ se tiene por concatenación:

$$uv = [holaMundo] \quad (5.3)$$

Propiedades:

- i. Dicha operación es asociativa: Para $w = [azul]$ tenemos:

$$(u \circ v) \circ w \equiv u \circ (v \circ w) = [holaMundoazul] \quad (5.4)$$

ii. Dicha operación tiene elemento neutro:

$$u \circ \lambda \equiv u = [\text{hola}] \quad (5.5)$$

iii. Dicha operación no es conmutativa:

$$uv = [\text{holaMundo}] \neq vu = [\text{Mundohola}] \quad (5.6)$$

II. Longitud:

Definición 5.2.9. Definimos **longitud** de una cadena $u \in \Sigma^*$, al número de símbolos que la componen (incluyendo los símbolos repetidos). Se denota normalmente como: $|u|$

$$|u| = \lambda + |u_1| + |u_2| + \dots + |u_n| \quad (5.7)$$

Ejemplo 5.2.10. Sea $u = [\text{hola}] \Rightarrow |u| = 4$

III. Potencia: Entendemos que para un alfabeto dado Σ^* y la palabras $u = \{u_1, u_2, \dots, u_n\} \in \Sigma^*$ con $n \in \mathbb{N}$ por **potencia** (denotado como u^n):

$$u^n = u^0 \circ u^1 \circ u^2 \circ \dots \circ u^n \quad (5.8)$$

Ejemplo 5.2.11. Para las palabras u se tiene como:

$$u_0 = \lambda, \quad u_1 = \text{hola}, \quad u_2 = \text{holahola}, \quad \dots \quad u_n = \text{CONCAT}_{i=0}^{i=n} u_i \quad (5.9)$$

Propiedades:

i. La potencia unidad de una palabra equivale a esa misma palabra:

$$a^1 = a \quad (5.10)$$

Ejemplo 5.2.12.

$$u^1 = [\text{hola}]^1 \equiv u = [\text{hola}] \quad (5.11)$$

El producto de distintas potencias de una palabra es igual a esa misma palabra con potencia resultado de la suma de los índices:

$$a^3 \cdot a^6 = a^9 \quad (5.12)$$

Ejemplo 5.2.13.

$$u^2 \cdot u^3 = [\text{holahola}] \cdot [\text{holaholahola}] \equiv u^5 = [\text{holaholaholaholahola}] \quad (5.13)$$

La potencia de una palabra sobre cero es igual a palabra vacía:

$$a^0 = \lambda \quad (5.14)$$

Ejemplo 5.2.14.

$$u^0 = [hola] = \lambda \quad (5.15)$$

El tamaño de una palabra sobre un índice cualquiera es igual al índice por el tamaño de la palabra original:

$$a^i = i \cdot |a| \quad (5.16)$$

Ejemplo 5.2.15.

$$u^5 = 5 \cdot \text{Length}(hola) = 20 \quad (5.17)$$

Nota: Ver ecuación: (5.13)

IV. Reflexión: Entendemos que para un alfabeto dado Σ^* y la palabras $u = \{u_1, u_2, \dots, u_n\} \in \Sigma^*$ con $n \in \mathbb{N}$ por **reflexión** (denotado como u^{-1}):

$$u^{-1} = \{u_n, \dots, u_2, u_1\} \quad (5.18)$$

Ejemplo 5.2.16. Para la palabra u se tiene como:

$$u^{-1} \equiv \frac{1}{u} = [aloh] \quad (5.19)$$

Propiedades:

- i. El tamaño de una palabra coincide con el de su inversa:

$$\text{Length}(a) = \text{Length}(a^{-1}) \quad (5.20)$$

Ejemplo 5.2.17.

$$\text{Length}(u) = \text{Length}(hola) = 4 \equiv \text{Length}(u^{-1}) = \text{Length}(aloh) = 4 \quad (5.21)$$

5.2.3. Lenguajes

Definición 5.2.18. Un **Lenguaje** es un subconjunto de palabras de algún alfabeto dado:

$$L \subseteq \wp(\Sigma) \equiv L \subseteq \Sigma^* \quad (5.22)$$

Definición 5.2.19. Existe el **Lenguaje Vacío** denotado como: $\varepsilon = \{\varepsilon\}$.

5.2.3.1. Operaciones

Nota: Se trabaja con un alfabeto $\Sigma = \{a, b\}$ y con dos lenguajes: $P = \{a, ab\}$ y $Q = \{a, b, bb\} \not\subseteq P, Q \wp(\Sigma)$.

- I. Unión: Para los alfabetos P y Q con el símbolo \cup , siendo r una palabra de la unión, se tiene que $P \cup Q = \{r \mid r \in P \vee r \in Q\}$

Ejemplo 5.2.20.

$$P \cup Q = \{a, ab, b, bb\} \quad (5.23)$$

Propiedades: Al ser cada alfabeto un conjunto en sí, esta operación conserva todas las propiedades de la Unión.

i. Conmutatividad:

$$P \cup Q \equiv Q \cup P = \{a, ab, b, bb\} \quad (5.24)$$

ii. Asociatividad:

Nota: Con $R = \{aaa, ba\}$.

$$(P \cup Q) \cup R = P \cup (Q \cup R) = \{a, aaa, ab, b, bb, ba\} \quad (5.25)$$

iii. Idempotencia

$$P \cup P \equiv P = \{a, ab\} \quad (5.26)$$

iv. Absorción:

$$P \cup W(P) \equiv W(P) \quad (5.27)$$

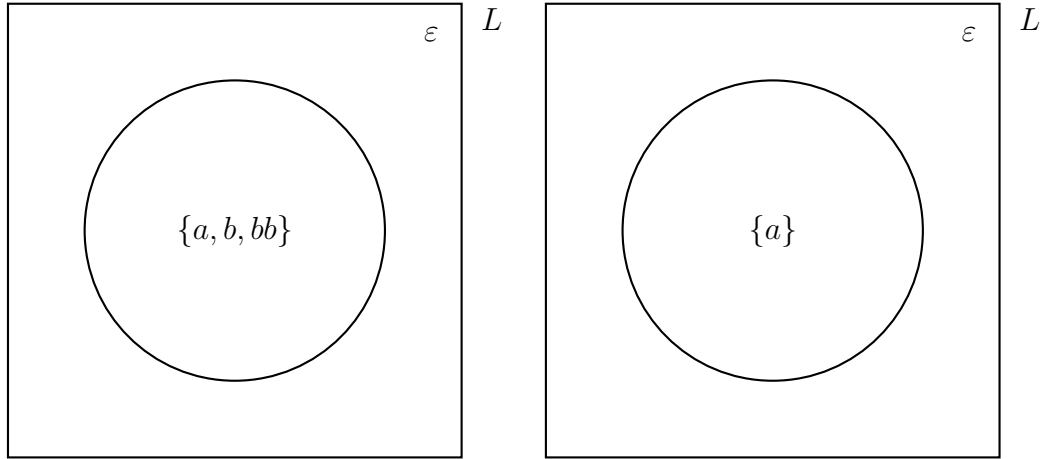
v. Neutralidad:

$$P \cup \varepsilon \equiv P = \{a, ab\} \quad (5.28)$$

II. Intersección: Para los alfabetos P y Q con el símbolo \cap , siendo r una palabra de la intersección, se tiene que $P \cap Q = \{r \in P \wedge r \in Q\}$

Ejemplo 5.2.21.

$$P \cap Q = \{a\} \quad (5.29)$$



(a) Diagrama de Venn para la operación: $P \cup Q$.

(b) Diagrama de Venn para la operación: $P \cap Q$.

III. Diferencia: Para los alfabetos P y Q con el símbolo $-$, siendo r una palabra de la diferencia, se tiene que $P - Q = \{r \in P \wedge r \notin Q\}$

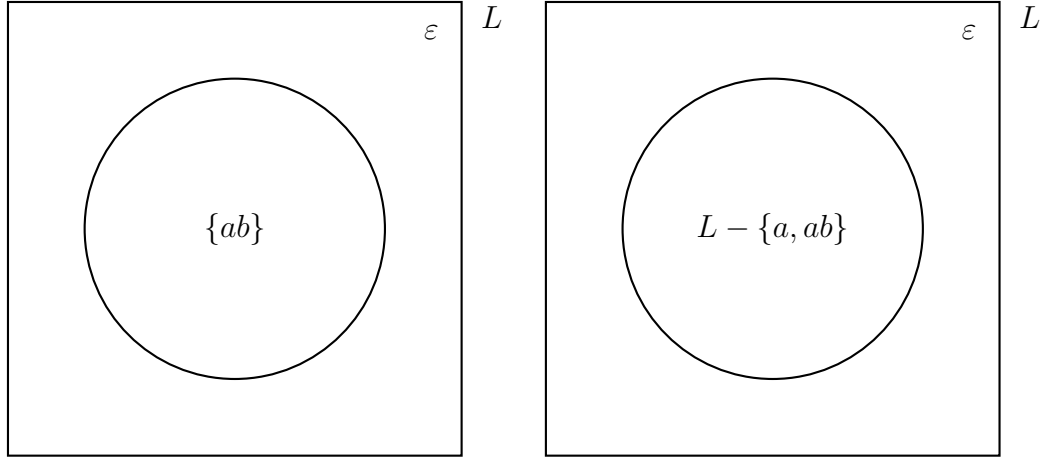
Ejemplo 5.2.22.

$$P - Q = \{ab\} \quad (5.30)$$

IV. Complemento: Para el alfabeto P se tiene por complemento de $P \Rightarrow \bar{P} = \Sigma^* - P$

Ejemplo 5.2.23.

$$\bar{P} = \{b\} \quad (5.31)$$



(a) Diagrama de Venn para la operación: $P - Q$.

(b) Diagrama de Venn para la operación: \bar{P} .

V. Concatenación: Para los conjuntos P y Q , siendo rs una palabra de la concatenación, se tiene que $PQ = \{ rs \mid r \in P \wedge s \in Q \}$

Ejemplo 5.2.24.

$$PQ = \{aa, ab, abb, ba, bab, bba, bbab\} \quad (5.32)$$

VI. Potencia: Se tiene por Potencia de un alfabeto L : L^n dónde L es un alfabeto y $n \in \mathbb{N}$ que representa en número de concatenaciones:

$$L^n = \{L_0, L_1, L_2, \dots, L_n\} \quad (5.33)$$

Ejemplo 5.2.25. Para el alfabeto $P = \{a, ab\}$, se tiene en iteración:

$$P^2 = \{aa, aab, aba, abab\} \quad (5.34)$$

VII. , también denominada: Cerradura de Kleene o Cerradura Estrella: Se denomina Clausura del Lenguaje, dónde L es un alfabeto y $n \in \mathbb{N}$ a todas las uniones de las Potencias del Lenguaje L (incluida L_0):

$$L^* = \bigcup_{i=0}^{i=n} L^i \quad (5.35)$$

Ejemplo 5.2.26. Para el alfabeto $P = \{a, ab\}$, se tiene:

$$P_0 \cup P^1 \cup P^2 \dots P^n = \{\varepsilon\} \cup \{a, ab\} \cup \{aa, aab, aba, abab\} \dots \quad (5.36)$$

VIII. Clausura Positiva: Se denomina Clausura del Lenguaje, dónde L es un alfabeto y $n \in \mathbb{N}$ a todas las uniones de las Potencias del Lenguaje L (excepto L_0):

$$L^+ = \bigcup_{i=1}^{i=n} L^i \quad (5.37)$$

Ejemplo 5.2.27. Para el alfabeto $P = \{a, ab\}$, se tiene:

$$P^1 \cup P^2 \dots P^n = \{a, ab\} \cup \{aa, aab, aba, abab\} \dots \quad (5.38)$$

Propiedades:

i. $L^* = L^+ \cup \{\lambda\}$

$$P^2 \cup \{\lambda\} \equiv \{aa, aab, aba, abab\} \cup \{\lambda\} = \{aa, aab, aba, abab\} \quad (5.39)$$

Demostración 5.2.28. Siendo $L^+ = \{L_1, L_2, \dots, L_n\}$ tenemos que:

$$L^+ \cup \{\lambda\} = \{L_0, L_1, L_2, \dots, L_n\} = L^*. \quad (5.40)$$

ii. $L^+ = LL^* = L^*L$

Demostración 5.2.29. Siendo $L^* = \{L_0, L_1, L_2, \dots, L_n\}$ tenemos que:

$$LL^* = L\{L_0, L_1, L_2, \dots, L_n\} = L\{L_1, L_2, \dots, L_n\} \quad (5.41)$$

IX. Reflexión: Se denota Reflexión del Lenguaje L a L^{-1}

$$L^{-1} = \{p^{-1} : p \in L\} \quad (5.42)$$

Ejemplo 5.2.30. Para el alfabeto P :

$$P^{-1} = \{ba, a\} \quad (5.43)$$

5.3. Lenguajes Regulares

Definición 5.3.1. Decimos que un **Lenguaje es Regular** si contiene los elementos: \emptyset , $\{\lambda\}$, $p \in \Sigma$ y está ligado a las operaciones: Unión (\cup), Concatenación (\circ) y Cerradura de Kleene (p^*).

Lenguaje Regular	Expresión Regular
\emptyset	\emptyset
$\{\lambda\}$	λ
$\{a\}, a \in \Sigma$	a
$U \cup V$	$(U \cup V)$
$U \circ V$	$(U)(V)$
U^*	$(U)^*$

Tabla 5.2: Relación de operadores entre Lenguajes Regulares y Expresiones Regulares.

Nota: \emptyset , $\{\lambda\}$, $\{p\}$, $p \in \Sigma$ son los denominados **Lenguajes Regulares Básicos**.

Ejemplo 5.3.2. Sea el $\Sigma = \{a, b, c\}$ tenemos:

$$U = \{a\} \circ \{b\}^* \cup \{c\}^*. \quad (5.44)$$

$$V = \{a\}^* \cup \{b\} \circ \{c\}. \quad (5.45)$$

i. Unión:

$$U \cup V = [\{a\} \circ \{b\}^* \cup \{c\}^* \cup \{a\}^* \cup \{b\} \circ \{c\}] \quad (5.46)$$

ii. Concatenación:

$$U \circ V = [\{a\} \circ \{b\}^* \cup \{c\}^* \circ \{a\}^* \cup \{b\} \circ \{c\}] \quad (5.47)$$

iii. Cerradura de Kleene:

$$U^* = [\{a\} \circ \{b\}^* \cup \{c\}^*]^* \quad (5.48)$$

Corolario 5.3.3. *Todo Lenguaje Finito es un Lenguajes Regular.*

5.4. Expresiones Regulares

Definición 5.4.1. Las **Expresiones Regulares** se forman recursivamente:

- i. Por medio de los símbolos: \emptyset y ε
- ii. $e \in \Sigma$
- iii. Siendo r y s el resultado, por medio de las operaciones $r \cup s, rs, r^*$

Corolario 5.4.2. *Las Expresiones Regulares son conceptualmente y operativamente lo mismo que los antes descritos Lenguajes Regulares, la diferencia radica en que estas últimas tienen el objetivo de ser lenguajes más legibles, es decir, las Expresiones Regulares son una simplificación de los Lenguajes Regulares para mejorar el entendimiento entre el hombre y la máquina.*

$$ER(E) := LR(E) \quad (5.49)$$

Expresión	Significado
'.'	Representa cualquier carácter excepto '\n'.
'*'	Ø o más copias de la expresión que le precede.
'['']'	Operador de conjunto de caracteres. También se usa para expresar rangos.
'^'	Operador para indicar el comienzo de la expresión.
'\$'	Indica el final de una expresión regular.
'{'}'	Indica el número de ocurrencias para el carácter que le precede.
'\'	Operador de “escapé” para caracteres restringidos.
'+'	Se utiliza con rangos, indica 1 o más copias de la expresión que le precede.
'?'	Indica Ø o una ocurrencia.
' '	Operador de disyunción.
'...'	Maraca el texto entrecomillado como literal.
'/'	Maraca como literal los caracteres que le preceden.
'()'	Operador para subconjuntos de expresiones regulares.

Tabla 5.3: Operadores comunes para Expresiones Regulares en UNIX.

5.5. Autómatas

5.5.1. ¿Qué es un Autómata?

Definición 5.5.1. Se conoce como Autómata Finito a máquinas abstractas que procesan cadenas para un determinado lenguaje.

Dicha máquina se compone de una serie de partes (Ver Figura 5.4):

- I. Cinta semi-infinita: Dividida a su vez en celdas donde se escribe la cadena de entrada.
- II. Unidad de Control (también llamada Cabeza Lectora): Que se encarga de procesar la citan.
- III. El Autómata propiamente dicho que mantiene la lógica del lenguaje a través de una serie de estados (de aceptación y finales).

Definición 5.5.2. Toda cadena necesariamente tiene una condición de parada, descrita como casilla vacía.

Definición 5.5.3. Todo Autómata M se expresa mediante una quintupla: $M = \{\Sigma, Q, q_0, F, t\}$

- i. Σ : Es el Lenguaje de aceptación: $\Sigma \subseteq \Sigma^*$
- ii. Q : Es el conjunto de estados: $\{q_0, q_1, q_2, \dots, q_n\} \in Q$
- iii. q_0 : Es el estado inicial.
- iv. F : Conjunto de estados finales o de aceptación.
- v. t : Se trata de la función de transición. Dependiendo del tipo de Autómata procesará o no determinados caracteres.

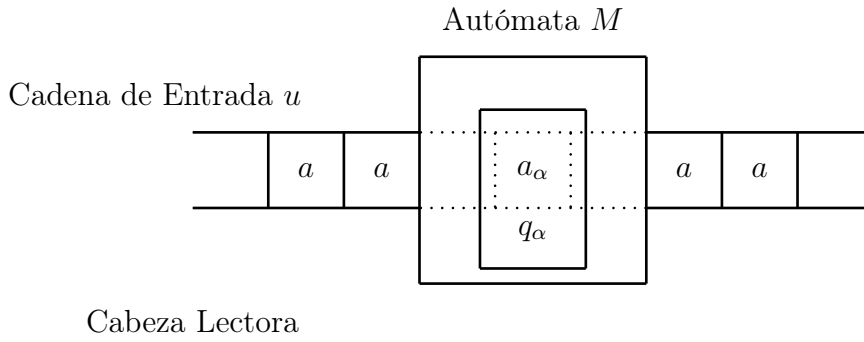


Figura 5.4: Representación de un Autómata Finito.

Nota: Dependiendo de la configuración de los estados internos del Autómata, diferenciamos tres tipos:

- i. Autómatas Finitos Determinista: Transiciones del tipo: $\delta(q, a)$. Procesan la palabra λ .
- ii. Autómatas Finitos No Determinista: Transiciones del tipo: $\Delta(q, a)$. No procesan la palabra λ .

5.5.2. Representación

La representación gráfica de un Autómata se realiza por medio de grafo dirigido y etiquetado o grafo etiquetado (Ver Sección 1.5.2).

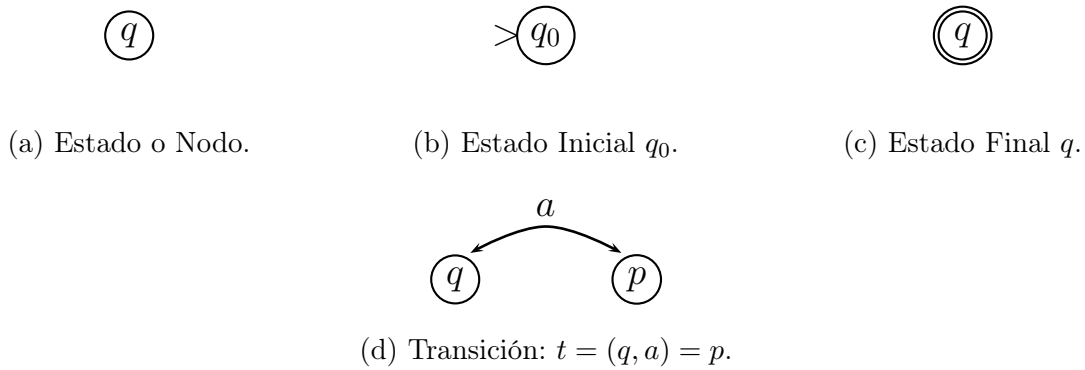


Figura 5.5: Representación de Autómatas.

5.5.3. Autómata Finito Determinista

Un **Autómata Finito Determinista** (AFD) se trata de un Autómata: $M_{AFD} = \{\Sigma, Q, q_0, F, \delta\}$ dónde la función de transición de estados se expresa como⁴⁴:

$$\delta : Q \times \Sigma \longrightarrow Q \text{ t.q. } (q, a) \longmapsto \delta(q, a) \quad (5.50)$$

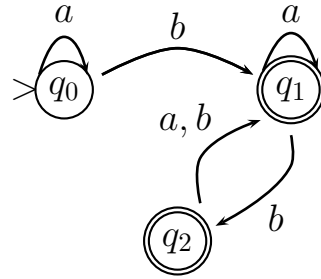
Corolario 5.5.4. Dado que q_0 es un estado de aceptación, la palabra λ aun siendo vacía es también procesada. Con el objetivo omitir este reconocimiento tenemos los AFnD.

Ejemplo 5.5.5. Dada la siguiente gramática:

- i. $\Sigma = \{a, b\}$
- ii. $Q = \{q_0, q_1, q_2\}$
- iii. q_0 : Es el estado inicial.
- iv. $F = \{q_1, q_2\}$
- v. $\delta =$ (Ver Figura 5.6)

δ	a	b
q_0	q_0	q_1
q_1	q_1	q_2
q_2	q_1	q_1

(a) Relación de Transiciones δ



(b) AFD para Tabla (a)

Figura 5.6: Ejemplo Autómata Finito Determinista.

5.5.4. Autómata Finito no Determinista

Un **Autómata Finito no Determinista** (AFnD) se trata de un Autómata: $M_{AFnD} = \{\Sigma, Q, q_0, F, \Delta\}$ dónde la función de transición entre estados se expresa como:

$$\Delta : Q \times \Sigma \longrightarrow \wp(Q) \text{ t.q. } (q, a) \longmapsto \Delta(q, a) = \{q_{i1}\} \quad (5.51)$$

Ejemplo 5.5.6. Dada la siguiente gramática:

- i. $\Sigma = \{a, b\}$
- ii. $Q = \{q_0, q_1, q_2, q_3\}$
- iii. q_0 : Es el estado inicial.
- iv. $F = \{q_1, q_3\}$
- v. $\delta =$ (Ver Figura 5.8)

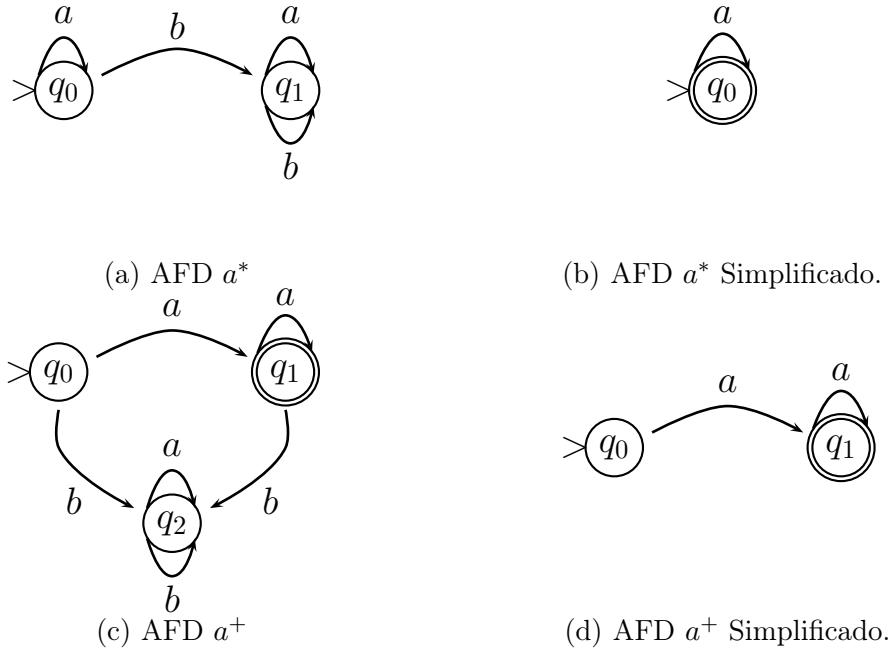


Figura 5.7: Ejemplos AFD.

5.5.5. Algoritmo: AFnD \Rightarrow AFD

Teorema 5.5.7. Para todo AFnD $M = (\Sigma, Q, q_0, F, \Delta)$ se puede obtener un AFD equivalente M' tal que $L(M) \equiv L(M')$.

Definición 5.5.8. La diferencia entre un AFnD y un AFD radica en la función de transición δ .

Programa 5.5.9. Pseudocódigo del Algoritmo AFnD \Rightarrow AFN:

I. Entrada: AFnD N

II. Salida: AFD N'

III. Método:

i. Correspondencia entre Estados.

Definición 5.5.10. Los únicos estados Q_D accesibles en AFD son subconjunto de Q_N cerrados respecto a λ .

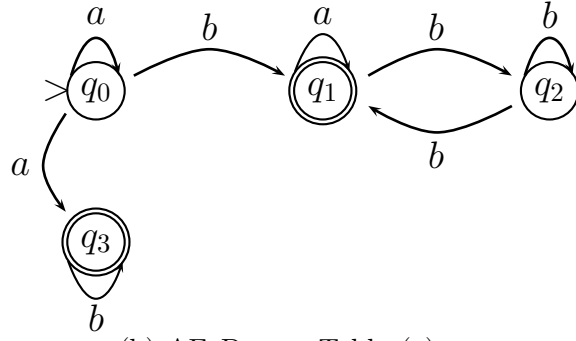
ii. Correspondencia del Estado Inicial.

Definición 5.5.11. El estado inicial de AFD es el resultado de calcular el cierre λ del estado inicial del AFND.

Corolario 5.5.12. El Cierre λ , también llamado $CLAUS_\lambda$ para un estado, es el conjunto de estados alcanzables mediante cero o más transiciones.

iii. Cálculo de Transiciones.

Δ	a	b
q_0	$\{q_0, q_1, q_2\}$	\emptyset
q_1	$\{q_1\}$	$\{q_2\}$
q_2	\emptyset	$\{q_1, q_2\}$
q_3	\emptyset	$\{q_3\}$

(a) Relación de Transiciones Δ 

(b) AFnD para Tabla (a)

Figura 5.8: Ejemplo Autómata Finito no Determinista.

Formalidad 5.5.13. $\delta_D(i, o) \setminus i \in Q_D$ para:

- $i = \{p_0, p_1, p_2, \dots, p_n\}$
- Obtener: $\bigcup_{i=0}^n \delta_N(p_i, a) = [r_0, r_1, r_2, \dots, r_m]$
- $\delta_D(i, o) = \bigcup_{j=1}^m CLAUS_\lambda(r_j)$

iv. Correspondencia del Estado Final.

Definición 5.5.14. En el AFD un estado será final si contiene algún estado final del AFND.

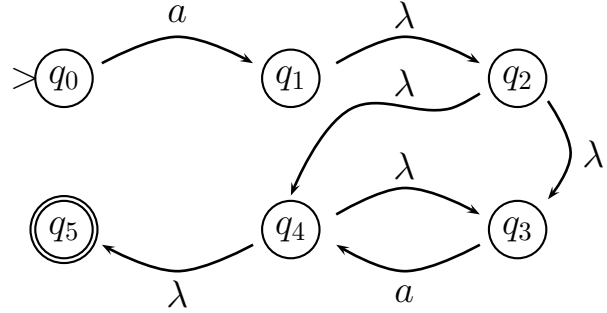
Ejemplo 5.5.15. Para el siguiente Automata:

- $\Sigma = \{a\}$
- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$
- q_0 : Es el estado inicial.
- $F = \{q_5\}$
- $\delta =$ (Ver Figura 5.9)

I. Correspondencia entre Estados (Cierre λ):

q_λ	$CLAUS_\lambda$
q_0	$CLAUS_\lambda(q_0) = \{q_0\}$
q_1	$\{q_1, q_2, q_3, q_4\}$
q_2	$\{q_2, q_3, q_4\}$
q_3	$\{q_3\}$
q_4	$\{q_3, q_4, q_5\}$
q_5	$\{q_5\}$

Δ	$\Delta(a)$	$\Delta(\lambda)$
q_0	$\{q_1\}$	$\{\emptyset\}$
q_1	$\{\emptyset\}$	$\{q_2\}$
q_2	$\{\emptyset\}$	$\{q_3, q_5\}$
q_3	$\{q_4\}$	$\{\emptyset\}$
q_4	$\{\emptyset\}$	$\{q_3, q_5\}$
q_5	$\{\emptyset\}$	$\{\emptyset\}$

(a) Relación de Transiciones Δ 

(b) AFnD para Tabla (a)

Figura 5.9: Autómata Finito no Determinista $a \cdot a^*$

II. Correspondencia del Estado Inicial.

$$qN_0 = CLAU S_\lambda(q_0) = \{q_0\} \equiv qD0 = \{q_1\} \quad (5.52)$$

III. Cálculo de Transiciones.

qN_i	$\delta(a)$	$\delta(\lambda)$
qN_0	$\{q_1\}$	$\{\emptyset\}$
qN_1	$\{\emptyset\}$	$\{q_2\}$
qN_2	$\{\emptyset\}$	$\{q_3, q_5\}$
qN_3	$\{q_4\}$	$\{\emptyset\}$
qN_4	$\{\emptyset\}$	$\{q_3, q_5\}$
qN_5	$\{\emptyset\}$	$\{\emptyset\}$

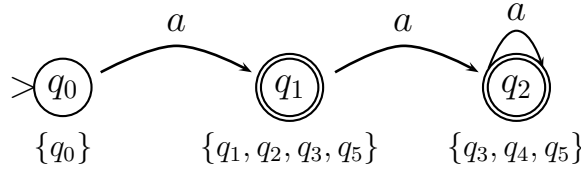
qDi	$\delta(a)$	$CLAU S_\lambda$
$qD0$	$\{q_1\}$	$\{q_1, q_2, q_3, q_5\}$
$qD3$	$\{q_4\}$	$\{q_3, q_4, q_5\}$

IV. Correspondencia del Estado Final.

- i. q_0 no contiene ningún estado final del AFnD.
- ii. q_1 contiene el estado q_5 del AFnD por lo que será estado final.
- iii. q_2 contiene el estado q_5 del AFnD por lo que será estado final.

Tipo	Criterio
AFD	$u \in \Sigma^* : M$ terminar de procesar u en un estado $q \in F$.
AFnD	$u \in \Sigma^* : M$ terminar de procesar u de manera completa en un estado $q \in F$.

Tabla 5.4: $L(M)$ para Autómatas.

Figura 5.11: Autómata Finito Determinista a partir de AFnD $a \cdot a^*$

5.5.6. Algoritmo: Expresión Regular \Rightarrow AFnD

Programa 5.5.16. Pseudocódigo del Algoritmo Expresión Regular \Rightarrow AFnD:

- I. Entrada: Expresión Regular $p \in \Sigma$
- II. Salida: AFnD M' que procesa $L(p)$
- III. Método:
 - i. Definir la Expresión Regular $p \in \Sigma$.
 - ii. Generar un AFnD para dicha Expresión Regular.
 - iii. Aplicar el Algoritmo: Algoritmo: AFnD \Rightarrow AFD (Ver Apartado 5.5.5)
 - iv. Minimizar el Número de Estados:

Definición 5.5.17. Para $\{p, q\} \in Q$ y $w \in \Sigma$, se dice que son Estados Equivalentes (no distinguibles):

$$w \longrightarrow \delta(p, w) = p_\lambda \text{ y } w \longrightarrow \delta(q, w) = q_\beta \Rightarrow p_\lambda = q_\beta \quad (5.53)$$

Corolario 5.5.18. La relación que se establece entre p y q es de Equivalencia.

Formalidad 5.5.19. Para $\{p, q\} \in Q$:

- i. Si p es un estado final y q no lo es, el par $\{p, q\}$ son distinguibles.
- ii. Si para dos estados $\{p, q\}$ se cumple:

$$\exists \delta \text{ t.q. } \delta(p, w) = p_\lambda, \exists \delta \text{ t.q. } \delta(q, w) = q_\beta \Rightarrow p_\lambda \neq q_\beta \quad (5.54)$$

El par $\{p, q\}$ no son distinguibles.

Ejemplo 5.5.20. Para el ejemplo de la Figura 5.9

- I. *idem*
- II. *idem*
- III. *idem*
- IV. Minimizar el Número de Estados:

- i. q_0 y q_1 no son distinguibles.
- ii. q_1 y q_2 no son distinguibles.
- iii. q_0 y q_2 si son distinguibles.

$$\delta(q_0, a) = \delta(q_2, a) \quad (5.55)$$

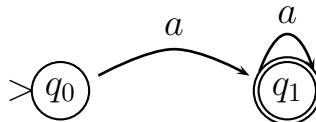


Figura 5.12: Autómata Finito Determinista Mínimo a partir de AFnD $a \cdot a^* \equiv a^+$

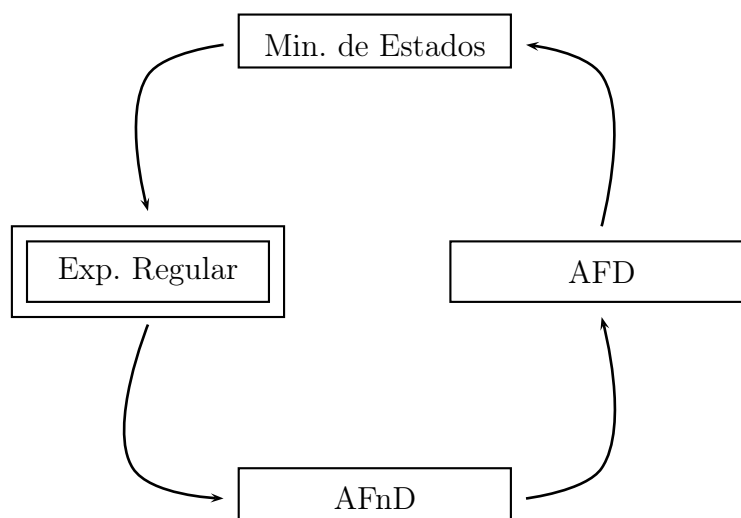


Figura 5.13: Ciclo de Thompson.

5.6. El Lenguaje LEX

LEX⁴⁵ o Lenguaje de Especificación para Analizadores Léxicos, se trata de un lenguaje que relaciona Expresiones Regulares con acciones determinadas.

Para generar un analizador LEX se han de seguir los siguientes pasos:

- i. Crear el programa fuente con las Expresiones Regulares `source.1`

- ii. Compilar el fichero `source.l` con LEX y generar (por defecto) `lex.yy.c`
- iii. Compilar `lex.yy.c` (con un compilador de Lenguaje C) y obtener el ejecutable.

La estructura de un programa LEX es la que sigue:

```

1 | Definitions
2 | %%
3 | Rules
4 | %%
5 | C Code

```

- I. Sección de Definiciones: En ella se definen variables, constantes y los patrones necesarios para el resto del programa.
- II. Sección de Reglas: Contiene el conjunto de reglas, definidas de la siguiente manera:

$$er_{\lambda} \quad \{sentencias\} \quad (5.56)$$

Donde:

- i. *er*: Es la Expresión Regular
- ii. *sentencias*: Es el conjunto de acciones a ejecutar cuando se estable la relación entre patrón y lexema.
- III. Sección de Código C: Consiste en una serie de sentencias auxiliares en Lenguaje C que permiten una mayor flexibilidad al desarrollador/programador.

5.7. Código fuente: gp1990la.l

5.7.1. Expresiones Regulares

El fichero fuente contiene las siguientes Expresiones Regulares:

- I. `NQUOTE` `[^']` \Rightarrow Toda palabra que comience por el carácter ‘ ’
- II. `IDENTIFIER` `[a-zA-Z]([a-zA-Z0-9_-])*` \Rightarrow Toda palabra que comience por una letra del alfabeto (mayúscula o minúscula) y seguido contenga los elementos de conjuntos repetidos de $[0, \infty]$:
 - i. Letras Minúsculas `[a-z]`
 - ii. Letras Mayúsculas: `[A-Z]`
 - iii. Dígitos: `[0-9]`
- III. `DIGITSECUENCE` `[0-9]+` \Rightarrow Toda palabra que contenga dígitos repetidos de $[1, \infty]$
- IV. `REALNUMBER` `[0-9]+". "[0-9]+` \Rightarrow Toda palabra que comience por un dígito de $[1, \infty]$ seguida del carácter ‘.’ y contenga de $[1, \infty]$ dígitos.

5.7.2. Tokens

Token	Valor	Token	Valor
AND	AND	TYPE	TYPE
ARRAY	ARRAY	UNTIL	UNTIL
CASE	CASE	VAR	VAR
CONST	CONST	WHILE	WHILE
DIV	DIV	WITH	WITH
DOWNTO	DOWNTO	{IDENTIFIER}	IDENTIFIER
ELSE	ELSE	":"	ASSIGNMENT
EXTERN or EXTERNAL	EXTERNAL	'({NQUOTE} '')+'	CHARACTER_STRING
ELSE	ELSE	":"	COLON
FOR	FOR	","	COMMA
FORWARD	FORWARD	{DIGITSECQUENCE}	DIGSEQ
FUNCTION	FUNCTION	."	DOT
GOTO	GOTO	"."	DOTDOT
IF	IF	"="	EQUAL
IN	IN	">="	GE
LABEL	LABEL	">"	GT
MOD	MOD	"["	LBRAC
NIL	NIL	"<="	LE
NOT	NOT	"("	LPAREN
OF	OF	"<"	LT
OR	OR	"_"	MINUS
OTHERWISE	OTHERWISE	"<>"	NOTEQUAL
PACKED	PACKED	"+"	PLUS
BEGIN	BEGIN	"]"	RBRAC
FILE	FILE	{REALNUMBER}	REALNUMBER
PROCEDURE	PROCEDURE	")"	RPAREN
PROGRAM	PROGRAM	";"	SEMICOLON
RECORD	RECORD	"/"	SLASH
REPEAT	REPEAT	"*"	STAR
SET	SET	"**"	STARSTAR
THEN	THEN	"->" or "^"	UPARROW
TO	TO		

Figura 5.14: Conjunto de Tokens para gp19901a

Notas

⁴⁴Usaremos en el texto para describir cualquier autómata la notación Backus-Naur Form (BNF).

⁴⁵LEX (Flex en su implementación GNU) nos permite generar un Analizador Léxico (AFD) mediante Expresiones Regulares.

Parte III

gp1990sa (Analizador Sintáctico)

Retrato

Mi infancia son recuerdos de un patio de Sevilla,
y un huerto claro donde madura el limonero;
mi juventud, veinte años en tierras de Castilla;
mi historia, algunos casos que recordar no quiero.

Ni un seductor Mañara, ni un Bradomín he sido
¿ya conocéis mi torpe aliño indumentario?,
más recibí la flecha que me asignó Cupido,
y amé cuanto ellas puedan tener de hospitalario.

Hay en mis venas gotas de sangre jacobina,
pero mi verso brota de manantial sereno;
y, más que un hombre al uso que sabe su doctrina,
soy, en el buen sentido de la palabra, bueno.

Adoro la hermosura, y en la moderna estética
corté las viejas rosas del huerto de Ronsard;
mas no amo los afeites de la actual cosmética,
ni soy un ave de esas del nuevo gay-trinar.

Desdeño las romanzas de los tenores huecos
y el coro de los grillos que cantan a la luna.
A distinguir me paro las voces de los ecos,
y escucho solamente, entre las voces, una.

¿Soy clásico o romántico? No sé. Dejar quisiera
mi verso, como deja el capitán su espada:
famosa por la mano viril que la blandiera,
no por el docto oficio del forjador preciada.

Converso con el hombre que siempre va conmigo
¿quien habla solo espera hablar a Dios un día?;
mi soliloquio es plática con ese buen amigo
que me enseñó el secreto de la filantropía.

Y al cabo, nada os debo; debéisme cuanto he escrito.

A mi trabajo acudo, con mi dinero pago
el traje que me cubre y la mansión que habito,
el pan que me alimenta y el lecho en donde yago.

Y cuando llegue el día del último viaje,
y esté al partir la nave que nunca ha de tornar,
me encontraréis a bordo ligero de equipaje,
casi desnudo, como los hijos de la mar.

Antonio Machado

Capítulo 6

Formalidades del Analizador Sintactico

Resumen:

6.1. Introducción a los Lenguajes Formales (LFs)	113
6.2. Gramáticas Independientes de Contexto	115
6.3. Jerarquía de Chomsky (JC)	117
6.4. Descripción de Gramáticas Formales	120
6.5. Analizadores Sintácticos	121
6.6. Análisis Sintáctico Descendente	122
6.7. Análisis Sintáctico Ascendente	124
6.8. Yacc (Yet another compiler-compiler)	124
Notas	127

6.1. Introducción a los Lenguajes Formales (LFs)

6.1.1. Definiciones

Definición 6.1.1. Un Lenguaje Formal se compone de un conjunto de signos finitos y unas leyes para operar con ellos.

Definición 6.1.2. Al conjunto de símbolos de un lenguaje se les denomina *Alfabeto*, denotado como Σ .

Definición 6.1.3. Al conjunto de leyes que describen al lenguaje se les denomina *sintaxis*.

Corolario 6.1.4. *Por tanto una palabra derivada de un alfabeto pertenecerá (será propio del lenguaje) si cumple las leyes formales del mismo.*

Definición 6.1.5. Para todos los lenguajes existe la palabra vacía, que se denota en este texto mediante el símbolo λ .

Corolario 6.1.6. *Por lo tanto:*

$$|\lambda| = 0 \tag{6.1}$$

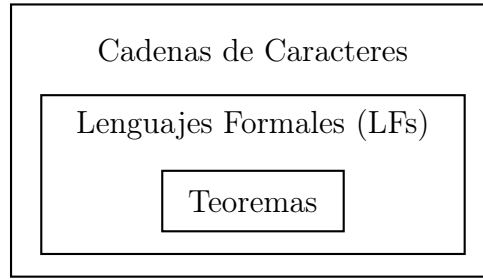


Figura 6.1: Relación entre: Teoremas, LFs y Cadenas de Caracteres.

Ejemplo 6.1.7. Para el alfabeto $O = \{0, 1\}$ y la palabra p , se dice que dicha palabra pertenece al alfabeto si cumple con la sintaxis:

$$p \in O \setminus p_0 = \lambda, p_1 = [01], p_2 = [0101], \dots, p_n = \text{CONCAT}_{i=0}^{i=n} [01]_i \equiv [01]^* \quad (6.2)$$

6.1.2. Especificación de los LFs

Los Lenguajes Formales se pueden describir por diversos métodos, sobre los que destacan:

- i. Mediante cadenas producidas por una gramática de Chomsky. Ver sección (6.3)
- ii. Por medio de una Expresión Regular. Ver sección (5.4)
- iii. Por cadenas aceptadas por un Autómata. Ver sección (5.5)

6.1.3. ¿Qué diferencia a un Lenguaje Natural (Humano) de un LF?

Para responder a esta pregunta, debemos aclarar que entendemos por Lenguaje Natural. Los Lenguajes Naturales tienen estructuras básicas en común con los Lenguajes Formales (de hecho la especificación formal se basa en el Lenguaje Humano).

El denominador común es la palabra como unidad estructural para construir oraciones. Por ello se tiene un alfabeto Σ para los Lenguajes Naturales, que es finito. La diferencia real entre estas dos formas de lenguajes radica en la polisemia (distintos significados) que tiene una palabra dentro de una oración (semántica) es decir, el significado varía según su posición y el contexto en el que se formula.

Ejemplo 6.1.8. Dados las siguientes palabras:

$$\{Javier, compró, una, casa\} \quad (6.3)$$

Se puede construir la frase:

$$Javier \text{ compró } una \text{ casa} \quad (6.4)$$

que sintáctica y semánticamente es correcta, pero la oración:

$$\text{Una casa compró Javier} \quad (6.5)$$

es sintácticamente correcta pero no semánticamente.

Por supuesto otra característica que diferencia a estos dos lenguajes es que un Lenguaje Formal como el Castellano ha sido perfeccionado a lo largo del tiempo. Con esto decimos que los Lenguajes Naturales evolucionan y están directamente relacionados con el tiempo.

6.2. Gramáticas Independientes de Contexto

Definición 6.2.1. Las Gramáticas Independientes de Contexto (GIC) se describen mediante una Tupla de 4 elementos:

$$G = (T, E, S, P) \quad (6.6)$$

Donde:

- i. T : Se trata de un alfabeto compuesto de símbolos no terminales.
- ii. E : Se trata de un alfabeto compuesto de símbolos terminales.

Formalidad 6.2.2. $T \cap E = \emptyset$

- iii. S : Variable símbolo inicial de la gramática $S \in T$.
- iv. P : Se trata del conjunto de reglas de producción del tipo:

$$A \longrightarrow w \quad (6.7)$$

Donde:

- a) A es la cabeza de la producción.
- b) w es el cuerpo de la producción.

Definición 6.2.3. El Lenguaje generado por una gramática se denota como:

$$L(G) : \{w \in \Sigma^* : S \xrightarrow{+} w\} \quad (6.8)$$

Ejemplo 6.2.4. Gramática para $L(G) = a^*$

$$T = \{a\}$$

$$E = \{S\}$$

$$S = \{\lambda\}$$

$$P = \begin{cases} S \longrightarrow aS \\ aS \xrightarrow{*} a \dots aS \\ aS \longrightarrow a \dots a \end{cases}$$

6.2.1. Derivaciones

Definición 6.2.5. Definimos derivación como el conjunto de producciones o generaciones donde una regla w_i iteran $[0, \infty]$ veces sobre si misma w_n

Formalidad 6.2.6. Existen dos tipos de producciones:

- i. $w \Rightarrow^+ w^*$: Derivaciones de $[1, \infty]$ veces.
- ii. $w \Rightarrow^* w^*$: Derivaciones de $[0, \infty]$ veces.

6.2.1.1. Representación mediante Árboles

Definición 6.2.7. Cualquier tipo de derivación puede ser representada gráficamente mediante un Árbol de Derivación.

Definición 6.2.8. Dicho Árbol se construye de la siguiente manera:

- i. La raíz se etiqueta con el símbolo inicial de la gramática S .
- ii. Cada nodo se etiqueta con un símbolo no terminal E .
- iii. Las hojas del árbol son etiquetadas con un símbolo terminal o λ .
- iv. Si la derivación es del tipo: $A \rightarrow s_1, s_2, \dots, s_k$ para $s_i \in (V \cup \Sigma^*)$:

A tiene k descendientes escritos de izquierda a derecha. (6.9)

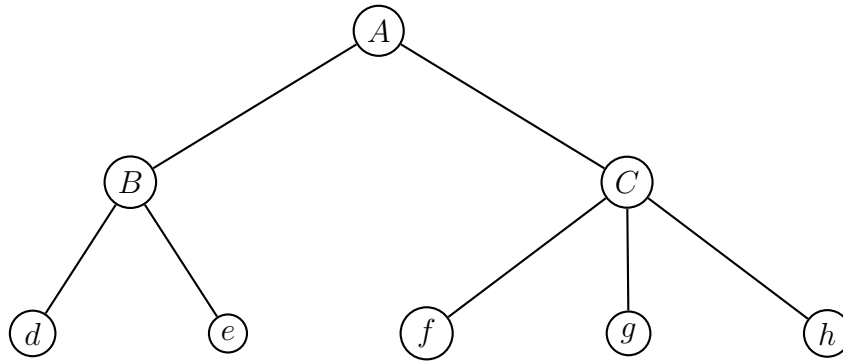
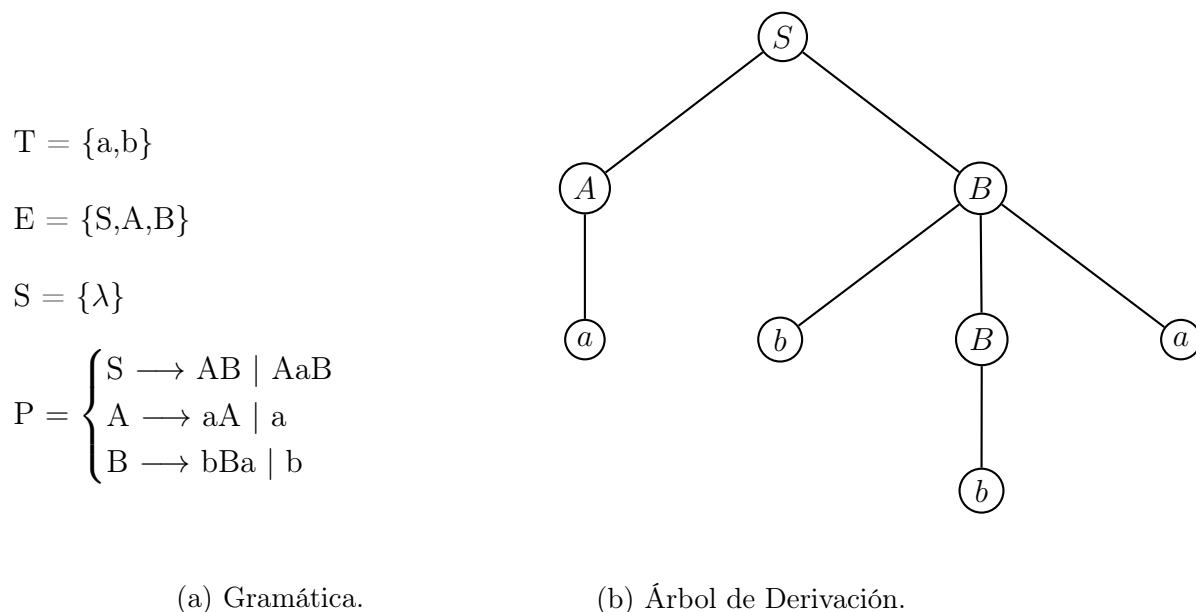


Figura 6.2: Ejemplo genérico de Árbol de Derivación.

Ejemplo 6.2.9. Para la siguiente gramática de la Figura (6.3) obtener la cadena $abba$

$$S \rightarrow AB \rightarrow AbBa \rightarrow abBa \rightarrow abba \quad (6.10)$$

Figura 6.3: Ejemplo Árbol de Derivación para obtener: *abba*

6.3. Jerarquía de Chomsky (JC)

Noam Avram Chomsky propuso en 1956 la formalidad de las gramáticas. A través de las reglas de producción en las que se basan las mismas, estableció un orden o jerarquía de las gramáticas y los lenguajes asociados.

Nota: Dichas gramáticas son subconjuntos unas de otras de modo que, el universo de las gramáticas son las de Tipo 0.

$$T_3 \subset T_2 \subset T_1 \subset T_0 \quad (6.11)$$

Nivel	Lenguaje	Autómata
0	Recursivamente Enumerable (LRE)	Máquina de Turing (MT)
1	Dependiente del Contexto (LSC)	Autómata Linealmente Acotado
2	Independiente del Contexto (LLC)	Autómata a Pila
3	Regular (LR)	Autómata Finito

Tabla 6.1: Relación entre: Nivel, Lenguaje y Autómata en la JC.

6.3.1. Niveles

La Jerarquía de Chomsky⁴⁶ contiene los siguientes niveles.

- I. Gramáticas de Tipo 0 (No Restrictivas): Estas gramáticas generan Lenguajes sin Restricciones.

Reglas de producción:

$$u \longrightarrow v \quad (6.12)$$

Donde:

- i. $u \in \Sigma^+$
- ii. $v \in \Sigma^*$
- iii. $u = xAy$
 - 1) $x, y \in \Sigma^*$
 - 2) $A \in E$

Notas:

- i. La cabeza de la producción no puede ser palabra vacía λ .
- ii. La cabeza de la regla debe contener al menos un símbolo no terminal.

Ejemplo 6.3.1. Sea la gramática G :

$$T = \{a, b\}$$

$$E = \{A, B, C\}$$

$$S = \{A\}$$

$$P = \begin{cases} A \longrightarrow aABC \mid abC \\ CB \longrightarrow BC \\ bB \longrightarrow bb \\ bC \longrightarrow b \end{cases}$$

II. Gramáticas de Tipo 1 (Sensibles al Contexto): Dichas gramáticas generan Lenguajes dependientes de Contexto.

Reglas de producción:

$$xAy \longrightarrow xvy \quad (6.13)$$

Donde:

- i. $v \in \Sigma^+$
- ii. $x, y \in \Sigma^*$

Nota: Se admite $S \rightarrow \lambda$

Ejemplo 6.3.2. Sea la gramática G :

$$T = \{S, B, C\}$$

$$E = \{a, b, c\}$$

$$S = \{\lambda\}$$

$$P = \begin{cases} S \rightarrow aSBc \mid aBC \\ bB \rightarrow bb \\ bC \rightarrow bc \\ CB \rightarrow BC \\ cC \rightarrow cc \\ aB \rightarrow ab \end{cases}$$

III. Gramáticas de Tipo 2 (Libres de Contexto): Las gramáticas de Tipo 2 generan Lenguajes independientes de Contexto.

Reglas de producción:

$$A \rightarrow v \tag{6.14}$$

Donde:

- i. $v \in \Sigma^*$
- ii. $A \in E$

Nota: La mayor parte de los Lenguajes de Programación pueden describirse a través de esta tipología.

Ejemplo 6.3.3. Ver Figura (6.3).

IV. Gramáticas de Tipo 3 (Regulares): Estas gramáticas generan Lenguajes Regulares.

Reglas de producción: Dichas gramáticas se clasifican en dos grupos:

- i. Gramáticas Lineales por la Izquierda:

$$P = \begin{cases} A \rightarrow a \\ A \rightarrow Va \\ S \rightarrow \lambda \end{cases}$$

- ii. Gramáticas Lineales por la Derecha:

$$P = \begin{cases} A \longrightarrow a \\ A \longrightarrow aV \\ S \longrightarrow \lambda \end{cases}$$

Donde:

- i. $a \in T$
 ii. $A, V \in E$

Ejemplo 6.3.4. Para G sobre:

- i. Gramáticas Lineales por la Izquierda:

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

$$E = \{A,B\}$$

$$S = \{A\}$$

$$P = \begin{cases} A \longrightarrow B1 \\ B \longrightarrow A0 \end{cases}$$

- ii. Gramáticas Lineales por la Derecha:

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

$$E = \{A,B\}$$

$$S = \{A\}$$

$$P = \begin{cases} A \longrightarrow 1B \\ B \longrightarrow 0A \end{cases}$$

6.4. Descripción de Gramáticas Formales

6.4.1. Backus-Naur Form

Backus⁴⁷-Naur⁴⁸ Form se trata una de las dos notaciones más importantes para Gramáticas Libres de Contexto.

John Backus, diseñador de lenguajes en IBM propuso para el Lenguaje de Programación IAL (conocido como ALGOL 58) un meta-lenguaje. Posteriormente con la publicación de ALGOL 60 la fórmula BNF se simplificó y perfeccionó.

BNF se trata de un conjunto de reglas derivativas del tipo: $\langle \text{symbol} \rangle ::= _expression_$

Donde:

- i. **symbol**: Es un símbolo No Terminal.
- ii. **_expression_**: Consiste en un conjunto de símbolos o de secuencias (separadas por el carácter '|') donde el símbolo “más a la izquierda” es el Terminal.
- iii. **'::='**: Operador de asignación. Indica que el símbolo de la izquierda es sustituido por la expresión de la derecha.

```

<syntax>          ::= <rule> | <rule> <syntax>
<rule>            ::= <opt-whitespace> "<" <rule-name> ">" <opt-whitespace> "::="
                   <opt-whitespace> <expression> <line-end>
<opt-whitespace> ::= " " <opt-whitespace> | ""
<expression>     ::= <list> | <list> "|" <expression>
<line-end>       ::= <opt-whitespace> <EOL> | <line-end> <line-end>
<list>           ::= <term> | <term> <opt-whitespace> <list>
<term>           ::= <literal> | "<" <rule-name> ">"
<literal>        ::= "'" <text> "'" | "\"" <text> "\""

```

EBNF: Existen distintas variantes sobre BNF. La más popular es Extended Backus-Naur Form (EBNF) que incorpora operadores de Expresiones Regulares como:

- i. a^+ : Repetir a de $[1, \infty]$ veces.
- ii. a^* : Repetir a de $[0, \infty]$ veces.

6.4.2. Wijngaarden Form

Var Wijngaarden Form (también conocida como vW-grammar p W-grammar) se trata de una técnica para definir Gramáticas Libres de Contexto en un número finito de reglas.

Las W-grammars se basan en la idea de que los símbolos No Terminales intercambian información entre los nodos y el árbol de “parseo”.

El primer uso de estas gramáticas fue en ALGOL 68.

6.5. Analizadores Sintácticos

Definición 6.5.1. La función del Analizador Sintáctico es la de relacionar el flujo de *tokens* elaborada por el Analizador Léxico y comprobar que la secuencia de estos *tokens* se corresponde con los patrones sintácticos (las reglas) del lenguaje.

Corolario 6.5.2. *El Analizador Sintáctico es el encargado de elaborar el árbol de análisis del código fuente sobre el que trabajaran el resto de fases del compilador.*

Definición 6.5.3. El Analizador Sintáctico es capaz de detectar errores en segunda fase, es decir, en la correspondencia entre *token* y *patrón sintáctico*.

Corolario 6.5.4. *Al contrario que ocurre con los errores léxicos, los errores sintácticos tienen una gran consistencia⁴⁹.*

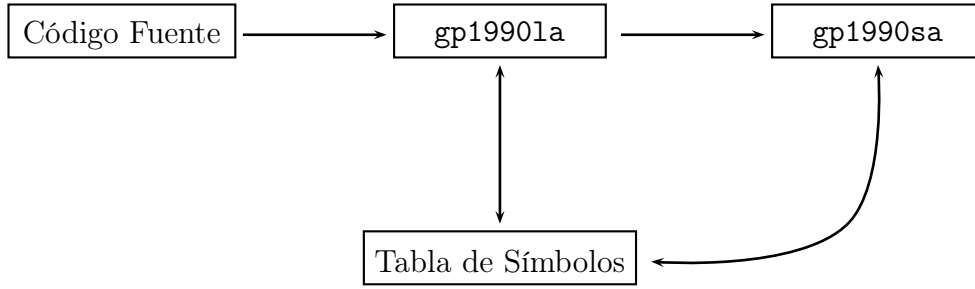


Figura 6.4: Relación entre el Analizador Léxico, Analizador Sintáctico y el Programa Fuente.

6.6. Análisis Sintáctico Descendente

Definición 6.6.1. Los Analizadores de Tipo Descendente (*Top-Down-Parser*) generan un árbol sintáctico a partir de una de entra $w \in L$.

Corolario 6.6.2. *Se trata de un recorrido en Preorden.* (Var Apartado 1.5.5.3)

Corolario 6.6.3. *Este análisis es análogo al proceso de derivación de una cadena por la izquierda.*

Tipos:

1. ASDR (Analizadores Sintácticos Descendentes Recursivos): Son analizadores ASD que basan su lógica en la recursión (Ver Apartado 6.6.1)
2. ASDnR (Analizadores Sintácticos Descendentes no Recursivos): Se trata de analizadores ASD dirigidos por pila o tabla. Su lógica de derivación es de la forma:

$$S \xrightarrow{*} w\alpha \quad (6.15)$$

Donde:

- i. w : Es la cadena de entrada.
- ii. α : Es la Tabla/Pila de símbolos gramaticales.

6.6.1. Autómatas LL(1)

Definición 6.6.4. Se trata de ASD con $k = 1$ tokens de predicción.

Este tipo de análisis se divide en tres fases o etapas:

- I. Conjunto de los PRIMEROS:

Definición 6.6.5. Si α es una forma sentencial compuesta por una concatenación de símbolos $PRIM(\alpha)$ es el conjunto de terminales o λ que pueden aparecer iniciando las cadenas que pueden derivar de α .

Formalidad 6.6.6. $a \in PRIM(\alpha)$ si $a \in (T \cup \{\lambda\}) \wedge \xrightarrow{*} a\beta$

Reglas 6.6.7. Para calcular el conjunto de los PRIMEROS tenemos:

- i. Si $\alpha \equiv \lambda \Rightarrow PRIM\{\lambda\} = \{\lambda\}$.
- ii. Si $\alpha \in (T \cup E)^+ \Rightarrow \alpha = a_1, a_2, \dots, a_n$ demuestra:
 - a. Si $a_1 \equiv a \in T \Rightarrow PRIM(\alpha) = \{a\}$.
 - b. Si $a_1 \equiv A \in E$ para:
 1. $PRIM(A) = \cup_{i=1}^n PRIM(\alpha_i) \wedge \alpha_i \in P$
 2. Si $PRIM(A) \wedge \lambda \in PRIM(A) \wedge A$ no es el último símbolo de $\alpha \Rightarrow PRIM(\alpha) = (PRIM(A) - \{\lambda\}) \cup PRIM(a_2, a_3, \dots, a_n)$
 3. Si A es el último símbolo de $\alpha \vee \lambda \notin PRIM(A) \Rightarrow PRIM(\alpha) = PRIM(A)$

Ejemplo 6.6.8. Dada la gramática:

$$G = (T, E, S, P) \quad (6.16)$$

$$T = \{ +, -, *, /, \text{num}, \text{id} \}$$

$$E = \{ E, E', T, T', F \}$$

$$S = \{E\}$$

$$P = \begin{cases} E \longrightarrow T E' \\ E' \longrightarrow + T E' \mid - T E' \mid \lambda \\ T \longrightarrow F T' \\ T' \longrightarrow * F T' \mid / F T' \mid \lambda \\ F \longrightarrow (E) \mid \text{num} \mid \text{id} \end{cases}$$

$$\text{example} \quad (6.17)$$

II. Conjunto de los SIGUIENTES:

Definición 6.6.9. Si A es un símbolo inicial no terminal de la gramática, $SIG(A)$ es el conjunto de terminales + $\{\$ \}$ que pueden aparecer a continuación de A en alguna forma sentencial derivada del símbolo inicial.

Formalidad 6.6.10. $a \in SIG(A)$ si $a \in (T \cup \{\$ \}) \wedge \exists \alpha, \beta \wedge S \xRightarrow{*} \alpha A a \beta$

Reglas 6.6.11. Para calcular el conjunto de los siguientes tenemos:

- i. Partimos de que: $SIG(A) = \emptyset$
- ii. Si A es símbolo inicial $\Rightarrow SIG(A) = SIG(A) \cup \{\$ \}$
- iii. Dada la regla: $B \rightarrow \alpha A \beta \Rightarrow SIG(A) = SIG(A) \cup (PRIM(\beta) - \{\lambda\})$
- iv. Dada la regla: $B \rightarrow \alpha A \vee B \rightarrow \alpha A \beta / \lambda \in PRIM(\beta) \Rightarrow SIG(A) = SIG(A) \cup SIG(B)$
- v. Repetir los pasos 3 y 4 hasta que no se puedan añadir más símbolos a $SIG(A)$

Ejemplo 6.6.12. Para la gramática (6.6.8)

example (6.18)

III. Conjunto de PREDICCIÓN:

Definición 6.6.13. Para una gramática ASDP con símbolo no terminal σ_N se debe consultar el símbolo de entrada y buscar en cada regla de dicho símbolo. Si los conjuntos de producción son disjuntos, el AS podrá construir una derivación hacia la izquierda de la cadena de entrada.

Formalidad 6.6.14. $PRED(A \rightarrow \alpha) \Rightarrow$

- i. Si $\alpha \in PRIM(\alpha) \Rightarrow (PRIM(\alpha) - \{\lambda\}) \cup SIG(A)$
- ii. Si no $\Rightarrow PRIM(\alpha)$

6.7. Análisis Sintáctico Ascendente

Definición 6.7.1. Los Analizadores de Tipo Ascendente (*Bottom-Up-Parser*) generan un árbol desde la hojas a la raíz.

Tipos:

- 1. LR ():
- 2. LALR ():

6.8. Yacc (Yet another compiler-compiler)

Yacc se trata de un popular “Front-End” para construir compiladores a nivel sintáctico diseñado originalmente por S.C. Johnson en 1970.

El análisis realizado por Yacc es del tipo LALR.

El proceso para generar un ejecutable con Yacc es el que sigue:

- i. Generar el fichero `source.y`
- ii. Compilar con Yacc `source.y` y generar por defecto `y.tab.c`
- iii. Compilar con CC (C Compiler) `y.tab.c` para obtener finalmente el ejecutable.

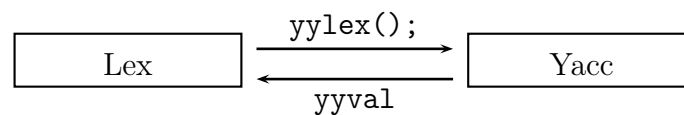


Figura 6.5: Relación entre el primitivas de Lex y Yacc.

La estructura de un fichero en Yacc es la que sigue:

```

1 | Definitions
2 | %%
3 | Rules
4 | %%
5 | C Code
  
```

I. Sección de Declaraciones: Dentro de este apartado existen a su vez, dos apartados:

- i. Apartado de rutinas en C: Delimitada por los símbolos { % (apertura) % } (cierre) contiene las directivas del preprocesador además, de variables y definiciones necesarias para el resto del programa.
- ii. Apartado de Tokens: Establece los Tokens a utilizar en el programa. Son necesarios desde el punto de vista global del programa. A su vez, en relación con Lex, dichos Tokens son intercambiados entre ambos programas.

II. Sección de Reglas de Traducción: Se definen en el mismo, las acciones semánticas que se corresponde a su vez con instrucciones en Código C. Las reglas de producción son de la forma:

$$E \longrightarrow E + T \mid T \quad (6.19)$$

Donde:

- i. E : Es un símbolo No Terminal.
- ii. T : Es un símbolo Terminal.

III. Apartado de Código en C: Se trata del conjunto de rutinas en C definidas por el desarrollador/programador. En el mismo apartado se establece o no la relación con LEX por medio de la función `yylex()`;

Dicha relación se describe en la Figura (6.5)

Notas

⁴⁶**Avram Noam Chomsky** es uno de los mayores lingüistas del siglo XX. Nació en Filadelfia el 7 de diciembre de 1928. A través de sus estudios sobre la formalidad de los lenguajes enuncia su teoría sobre “La adquisición individual” donde intenta dar explicación a las formalidades de los lenguajes naturales a través de representaciones formales.

⁴⁷**John Backus** fue un importante científico de computación nacido en el estado de Filadelfia (EEUU), el 3 de diciembre de 1924. Es prestigioso ganador del Premio Turing en el año 1977 debido en gran parte a sus trabajos sobre especificación de lenguajes de alto nivel.

Backus estuvo dentro del primer proyecto de FORTRAN, el primer lenguaje de alto nivel en la historia de la computación. Además su notación sobre gramáticas sentó las bases para ALGOL.

Su famosa notación es Backus Naur Form (BNF) que describe un autómata a partir de un conjunto de símbolos.

⁴⁸**Peter Naur** es un prestigioso científico danés nacido el 25 de octubre de 1928 ganador del Premio Turing en 2005. Su trabajo más representativo consiste en sentar junto a John Backus la notación para especificación de autómatas para lenguajes formales.

⁴⁹Están perfectamente definidos en el Lenguaje de Programación.


Parte IV

Anexos y Formalidades

Apéndice A

Blaise Pascal

“El corazón tiene razones que la razón ignora.”

 laise Pascal, nacido el 19 de Junio de 1623 en Clemont y fallecido el 19 de Agosto de 1662 en París, es un importantísimo pensador: matemático físico, filósofo y escritor. Podemos afirmar que es “un hombre e su época...”. Pascal fue un importante racionalista a la vez que, según el paso de los años dedico enormes esfuerzos en “racionalizar” el Cristianismo y la figura e Dios. Es considerado un importante teólogo.

Trabajó en el campo de las matemáticas y diseño una máquina de cálculo “Pascalina” capaz de realizar adiciones, con el tiempo, la propia máquina incorporó la operación de substracción.

Entre su más destacables estudios se encuentra la demostración del vacío. *Traité sur le vide* (Tratado sobre el vacío).

Bibliografía:

- i. *Essai pour les coniques* (1639)
- ii. *Experiences nouvelles touchant le vide* (1647)
- iii. *Traité du triangle arithmétique* (1653)
- iv. *Lettres provinciales* (1656–57)
- v. *De l'Ésprit géométrique* (1657 o 1658)
- vi. *Écrit sur la signature du formulaire* (1661)
- vii. *Pensées* (Sin terminar)

Apéndice B

gp1990sa.y

B.1. Yacc

```
sa.1  #include<stdio.h>
sa.2
sa.3  %}
sa.4
sa.5  %token AND ARRAY ASSIGNMENT CASE CHARACTER_STRING
sa.6  %token COLON COMMA CONST DIGSEQ DIV DO DOT DOTDOT
sa.7  %token DOWNT0 ELSE END EQUAL EXTERNAL FOR FORWARD
sa.8  %token FUNCTION GE GOTO GT IDENTIFIER IF IN LABEL LBRAC
sa.9  %token LE LPAREN LT MINUS MOD NIL NOT NOTEQUAL OF OR
sa.10 %token OTHERWISE PACKED PBEGIN PFILE PLUS PROCEDURE
sa.11 %token PROGRAM RBRAC REALNUMBER RECORD REPEAT RPAREN
sa.12 %token SEMICOLON SET SLASH STAR STARSTAR THEN
sa.13 %token TO TYPE UNTIL UPARROW VAR WHILE WITH
sa.14
sa.15 %%
sa.16 file      : program
sa.17           | module
sa.18           ;
sa.19
sa.20 program : program_heading semicolon block DOT
sa.21         ;
sa.22
sa.23 program_heading : PROGRAM identifier
sa.24                 | PROGRAM identifier LPAREN identifier_list RPAREN
sa.25                 ;
sa.26
sa.27 identifier_list : identifier_list COMMA identifier
sa.28                 | identifier
sa.29                 ;
sa.30
sa.31 block : label_declaration_part
```

```

sa.32      constant_definition_part
sa.33      type_definition_part
sa.34      variable_declaration_part
sa.35      procedure_and_function_declaration_part
sa.36      statement_part
sa.37      ;
sa.38
sa.39  module : constant_definition_part
sa.40      type_definition_part
sa.41      variable_declaration_part
sa.42      procedure_and_function_declaration_part
sa.43      ;
sa.44
sa.45  label_declaration_part : LABEL label_list semicolon
sa.46      |
sa.47      ;
sa.48
sa.49  label_list : label_list comma label
sa.50      | label
sa.51      ;
sa.52
sa.53  label : DIGSEQ
sa.54      ;
sa.55
sa.56  constant_definition_part : CONST constant_list
sa.57      |
sa.58      ;
sa.59
sa.60  constant_list : constant_list constant_definition
sa.61      | constant_definition
sa.62      ;
sa.63
sa.64  constant_definition : identifier EQUAL cexpression semicolon
sa.65      ;
sa.66
sa.67  /*constant : cexpression ;                      /* good stuff! */
sa.68
sa.69  cexpression : csimple_expression
sa.70      | csimple_expression relop csimple_expression
sa.71      ;
sa.72
sa.73  csimple_expression : cterm
sa.74      | csimple_expression addop cterm
sa.75      ;
sa.76
sa.77  cterm : cfactor

```

```

sa.78          | cterm mulop cfactor
sa.79          ;
sa.80
sa.81 cfactor : sign cfactor
sa.82          | cexponentiation
sa.83          ;
sa.84
sa.85 cexponentiation : cprimary
sa.86          | cprimary STARSTAR cexponentiation
sa.87          ;
sa.88
sa.89 cprimary : identifier
sa.90          | LPAREN cexpression RPAREN
sa.91          | unsigned_constant
sa.92          | NOT cprimary
sa.93          ;
sa.94
sa.95 constant : non_string
sa.96          | sign non_string
sa.97          | CHARACTER_STRING
sa.98          ;
sa.99
sa.100 sign : PLUS
sa.101         | MINUS
sa.102         ;
sa.103
sa.104 non_string : DIGSEQ
sa.105         | identifier
sa.106         | REALNUMBER
sa.107         ;
sa.108
sa.109 type_definition_part : TYPE type_definition_list
sa.110         |
sa.111         ;
sa.112
sa.113 type_definition_list : type_definition_list type_definition
sa.114         | type_definition
sa.115         ;
sa.116
sa.117 type_definition : identifier EQUAL type_denoter semicolon
sa.118         ;
sa.119
sa.120 type_denoter : identifier
sa.121         | new_type
sa.122         ;
sa.123

```

```

sa.124 new_type : new_ordinal_type
sa.125         | new_structured_type
sa.126         | new_pointer_type
sa.127         ;
sa.128
sa.129 new_ordinal_type : enumerated_type
sa.130         | subrange_type
sa.131         ;
sa.132
sa.133 enumerated_type : LPAREN identifier_list RPAREN
sa.134         ;
sa.135
sa.136 subrange_type : constant DOTDOT constant
sa.137         ;
sa.138
sa.139 new_structured_type : structured_type
sa.140         | PACKED structured_type
sa.141         ;
sa.142
sa.143 structured_type : array_type
sa.144         | record_type
sa.145         | set_type
sa.146         | file_type
sa.147         ;
sa.148
sa.149 array_type : ARRAY LBRAC index_list RBRAC OF component_type
sa.150         ;
sa.151
sa.152 index_list : index_list comma index_type
sa.153         | index_type
sa.154         ;
sa.155
sa.156 index_type : ordinal_type ;
sa.157
sa.158 ordinal_type : new_ordinal_type
sa.159         | identifier
sa.160         ;
sa.161
sa.162 component_type : type_denoter ;
sa.163
sa.164 record_type : RECORD record_section_list END
sa.165         | RECORD record_section_list semicolon variant_part END
sa.166         | RECORD variant_part END
sa.167         ;
sa.168
sa.169 record_section_list : record_section_list semicolon record_section

```



```

sa.170         | record_section
sa.171         ;
sa.172
sa.173 record_section : identifier_list COLON type_denoter
sa.174         ;
sa.175
sa.176 variant_part : CASE variant_selector OF variant_list semicolon
sa.177         | CASE variant_selector OF variant_list
sa.178         |
sa.179         ;
sa.180
sa.181 variant_selector : tag_field COLON tag_type
sa.182         | tag_type
sa.183         ;
sa.184
sa.185 variant_list : variant_list semicolon variant
sa.186         | variant
sa.187         ;
sa.188
sa.189 variant : case_constant_list COLON LPAREN record_section_list RPAREN
sa.190         | case_constant_list COLON LPAREN record_section_list semicolon
sa.191         variant_part RPAREN
sa.192         | case_constant_list COLON LPAREN variant_part RPAREN
sa.193         ;
sa.194
sa.195 case_constant_list : case_constant_list comma case_constant
sa.196         | case_constant
sa.197         ;
sa.198
sa.199 case_constant : constant
sa.200         | constant DOTDOT constant
sa.201         ;
sa.202
sa.203 tag_field : identifier ;
sa.204
sa.205 tag_type : identifier ;
sa.206
sa.207 set_type : SET OF base_type
sa.208         ;
sa.209
sa.210 base_type : ordinal_type ;
sa.211
sa.212 file_type : PFILE OF component_type
sa.213         ;
sa.214
sa.215 new_pointer_type : UPARROW domain_type

```

```

sa.216          ;
sa.217
sa.218 domain_type : identifier ;
sa.219
sa.220 variable_declaration_part : VAR variable_declaration_list semicolon
sa.221          |
sa.222          ;
sa.223
sa.224 variable_declaration_list :
sa.225          variable_declaration_list semicolon variable_declaration
sa.226          | variable_declaration
sa.227          ;
sa.228
sa.229 variable_declaration : identifier_list COLON type_denoter
sa.230          ;
sa.231
sa.232 procedure_and_function_declaration_part :
sa.233          proc_or_func_declaration_list semicolon
sa.234          |
sa.235          ;
sa.236
sa.237 proc_or_func_declaration_list :
sa.238          proc_or_func_declaration_list semicolon proc_or_func_declaration
sa.239          | proc_or_func_declaration
sa.240          ;
sa.241
sa.242 proc_or_func_declaration : procedure_declaration
sa.243          | function_declaration
sa.244          ;
sa.245
sa.246 procedure_declaration : procedure_heading semicolon directive
sa.247          | procedure_heading semicolon procedure_block
sa.248          ;
sa.249
sa.250 procedure_heading : procedure_identification
sa.251          | procedure_identification formal_parameter_list
sa.252          ;
sa.253
sa.254 directive : FORWARD
sa.255          | EXTERNAL
sa.256          ;
sa.257
sa.258 formal_parameter_list : LPAREN formal_parameter_section_list RPAREN ;
sa.259
sa.260 formal_parameter_section_list :
sa.261          formal_parameter_section_list semicolon formal_parameter_section

```

```

sa.262         | formal_parameter_section
sa.263         ;
sa.264
sa.265 formal_parameter_section : value_parameter_specification
sa.266         | variable_parameter_specification
sa.267         | procedural_parameter_specification
sa.268         | functional_parameter_specification
sa.269         ;
sa.270
sa.271 value_parameter_specification : identifier_list COLON identifier
sa.272         ;
sa.273
sa.274 variable_parameter_specification : VAR identifier_list COLON identifier
sa.275         ;
sa.276
sa.277 procedural_parameter_specification : procedure_heading ;
sa.278
sa.279 functional_parameter_specification : function_heading ;
sa.280
sa.281 procedure_identification : PROCEDURE identifier ;
sa.282
sa.283 procedure_block : block ;
sa.284
sa.285 function_declaration : function_heading semicolon directive
sa.286         | function_identification semicolon function_block
sa.287         | function_heading semicolon function_block
sa.288         ;
sa.289
sa.290 function_heading : FUNCTION identifier COLON result_type
sa.291         | FUNCTION identifier formal_parameter_list COLON result_type
sa.292         ;
sa.293
sa.294 result_type : identifier ;
sa.295
sa.296 function_identification : FUNCTION identifier ;
sa.297
sa.298 function_block : block ;
sa.299
sa.300 statement_part : compound_statement ;
sa.301
sa.302 compound_statement : PBEGIN statement_sequence END ;
sa.303
sa.304 statement_sequence : statement_sequence semicolon statement
sa.305         | statement
sa.306         ;
sa.307

```

```

sa.308 statement : open_statement
sa.309           | closed_statement
sa.310           ;
sa.311
sa.312 open_statement : label COLON non_labeled_open_statement
sa.313           | non_labeled_open_statement
sa.314           ;
sa.315
sa.316 closed_statement : label COLON non_labeled_closed_statement
sa.317           | non_labeled_closed_statement
sa.318           ;
sa.319
sa.320 non_labeled_closed_statement : assignment_statement
sa.321           | procedure_statement
sa.322           | goto_statement
sa.323           | compound_statement
sa.324           | case_statement
sa.325           | repeat_statement
sa.326           | closed_with_statement
sa.327           | closed_if_statement
sa.328           | closed_while_statement
sa.329           | closed_for_statement
sa.330           |
sa.331           ;
sa.332
sa.333 non_labeled_open_statement : open_with_statement
sa.334           | open_if_statement
sa.335           | open_while_statement
sa.336           | open_for_statement
sa.337           ;
sa.338
sa.339 repeat_statement : REPEAT statement_sequence UNTIL boolean_expression
sa.340           ;
sa.341
sa.342 open_while_statement : WHILE boolean_expression DO open_statement
sa.343           ;
sa.344
sa.345 closed_while_statement : WHILE boolean_expression DO closed_statement
sa.346           ;
sa.347
sa.348 open_for_statement : FOR control_variable ASSIGNMENT initial_value direction
sa.349                       final_value DO open_statement
sa.350           ;
sa.351
sa.352 closed_for_statement : FOR control_variable ASSIGNMENT initial_value direction
sa.353                       final_value DO closed_statement

```

```

sa.354         ;
sa.355
sa.356 open_with_statement : WITH record_variable_list DO open_statement
sa.357         ;
sa.358
sa.359 closed_with_statement : WITH record_variable_list DO closed_statement
sa.360         ;
sa.361
sa.362 open_if_statement : IF boolean_expression THEN statement
sa.363         | IF boolean_expression THEN closed_statement ELSE open_statement
sa.364         ;
sa.365
sa.366 closed_if_statement : IF boolean_expression THEN closed_statement
sa.367         ELSE closed_statement
sa.368         ;
sa.369
sa.370 assignment_statement : variable_access ASSIGNMENT expression
sa.371         ;
sa.372
sa.373 variable_access : identifier
sa.374         | indexed_variable
sa.375         | field_designator
sa.376         | variable_access UPARROW
sa.377         ;
sa.378
sa.379 indexed_variable : variable_access LBRAC index_expression_list RBRAC
sa.380         ;
sa.381
sa.382 index_expression_list : index_expression_list comma index_expression
sa.383         | index_expression
sa.384         ;
sa.385
sa.386 index_expression : expression ;
sa.387
sa.388 field_designator : variable_access DOT identifier
sa.389         ;
sa.390
sa.391 procedure_statement : identifier params
sa.392         | identifier
sa.393         ;
sa.394
sa.395 params : LPAREN actual_parameter_list RPAREN ;
sa.396
sa.397 actual_parameter_list : actual_parameter_list comma actual_parameter
sa.398         | actual_parameter
sa.399         ;

```

```

sa.400
sa.401  /*
sa.402   * this forces you to check all this to be sure that only write and
sa.403   * writeln use the 2nd and 3rd forms, you really can't do it easily in
sa.404   * the grammar, especially since write and writeln aren't reserved
sa.405   */
sa.406 actual_parameter : expression
sa.407         | expression COLON expression
sa.408         | expression COLON expression COLON expression
sa.409         ;
sa.410
sa.411 goto_statement : GOTO label
sa.412         ;
sa.413
sa.414 case_statement : CASE case_index OF case_list_element_list END
sa.415         | CASE case_index OF case_list_element_list semicolon END
sa.416         | CASE case_index OF case_list_element_list semicolon
sa.417                 otherwisepart statement END
sa.418         | CASE case_index OF case_list_element_list semicolon
sa.419                 otherwisepart statement semicolon END
sa.420         ;
sa.421
sa.422 case_index : expression ;
sa.423
sa.424 case_list_element_list : case_list_element_list semicolon case_list_element
sa.425         | case_list_element
sa.426         ;
sa.427
sa.428 case_list_element : case_constant_list COLON statement
sa.429         ;
sa.430
sa.431 otherwisepart : OTHERWISE
sa.432         | OTHERWISE COLON
sa.433         ;
sa.434
sa.435 control_variable : identifier ;
sa.436
sa.437 initial_value : expression ;
sa.438
sa.439 direction : TO
sa.440         | DOWNTO
sa.441         ;
sa.442
sa.443 final_value : expression ;
sa.444
sa.445 record_variable_list : record_variable_list comma variable_access

```

```

sa.446         | variable_access
sa.447         ;
sa.448
sa.449 boolean_expression : expression ;
sa.450
sa.451 expression : simple_expression
sa.452         | simple_expression relop simple_expression
sa.453         ;
sa.454
sa.455 simple_expression : term
sa.456         | simple_expression addop term
sa.457         ;
sa.458
sa.459 term : factor
sa.460         | term mulop factor
sa.461         ;
sa.462
sa.463 factor : sign factor
sa.464         | exponentiation
sa.465         ;
sa.466
sa.467 exponentiation : primary
sa.468         | primary STARSTAR exponentiation
sa.469         ;
sa.470
sa.471 primary : variable_access
sa.472         | unsigned_constant
sa.473         | function_designator
sa.474         | set_constructor
sa.475         | LPAREN expression RPAREN
sa.476         | NOT primary
sa.477         ;
sa.478
sa.479 unsigned_constant : unsigned_number
sa.480         | CHARACTER_STRING
sa.481         | NIL
sa.482         ;
sa.483
sa.484 unsigned_number : unsigned_integer | unsigned_real ;
sa.485
sa.486 unsigned_integer : DIGSEQ
sa.487         ;
sa.488
sa.489 unsigned_real : REALNUMBER
sa.490         ;
sa.491

```

```

sa.492  /* functions with no params will be handled by plain identifier */
sa.493  function_designator : identifier params
sa.494      ;
sa.495
sa.496  set_constructor : LBRAC member_designator_list RBRAC
sa.497      | LBRAC RBRAC
sa.498      ;
sa.499
sa.500  member_designator_list : member_designator_list comma member_designator
sa.501      | member_designator
sa.502      ;
sa.503
sa.504  member_designator : member_designator DOTDOT expression
sa.505      | expression
sa.506      ;
sa.507
sa.508  addop: PLUS
sa.509      | MINUS
sa.510      | OR
sa.511      ;
sa.512
sa.513  mulop : STAR
sa.514      | SLASH
sa.515      | DIV
sa.516      | MOD
sa.517      | AND
sa.518      ;
sa.519
sa.520  relop : EQUAL
sa.521      | NOTEQUAL
sa.522      | LT
sa.523      | GT
sa.524      | LE
sa.525      | GE
sa.526      | IN
sa.527      ;
sa.528
sa.529  identifier : IDENTIFIER
sa.530      ;
sa.531
sa.532  semicolon : SEMICOLON
sa.533      ;
sa.534
sa.535  comma : COMMA
sa.536      ;
sa.537

```



```

sa.538  %%
sa.539
sa.540  extern int line_no;
sa.541  extern char *yytext;
sa.542
sa.543  int yyerror(s)
sa.544  char *s;
sa.545  {
sa.546      fprintf(stderr, "***\n");
sa.547      fprintf(stderr, "*** %s: error at or before '%s', line %d\n",
sa.548                  s, yytext, line_no);
sa.549      fprintf(stderr, "***\n");
sa.550  }
sa.551
sa.552
sa.553  int main (void) {
sa.554      if(yyparse()==0){
sa.555          printf("%s\n","OK");
sa.556          return 0;
sa.557      }
sa.558
sa.559      return line_no;
sa.560  }
sa.561
sa.562
sa.563
sa.564

```


Apéndice C

Gramáticas

C.1. Pascal ISO 1990:7185

```
ps.1 <program> ::= program <identifier> ; <block> . <identifier> ::= <letter >
ps.2
ps.3 {<letter or digit>}
ps.4
ps.5 <letter or digit> ::= <letter> | <digit>
ps.6
ps.7 <block> ::= <label declaration part> <constant definition part> <type definition
ps.8 part> <variable declaration part>
ps.9
ps.10 <procedure and function declaration part> <statement part>
ps.11
ps.12 <label declaration part> ::= <empty> | label <label> {, <label>} ;
ps.13
ps.14 <label> ::= <unsigned integer>
ps.15
ps.16 <constant definition part> ::= <empty> | const <constant definition> { ;
ps.17 <constant definition>} ; <constant definition> ::= <identifier> = <constant>
ps.18 <constant> ::= <unsigned number> | <sign> <unsigned number> | <constant
ps.19 identifier> | <sign> <constant identifier> |
ps.20
ps.21 <string>
ps.22
ps.23 <unsigned number> ::= <unsigned integer> | <unsigned real>
ps.24
ps.25 <unsigned integer> ::= <digit> {<digit>}
ps.26
ps.27 <unsigned real> ::= <unsigned integer> . <unsigned integer> | <unsigned integer>
ps.28 .
ps.29
ps.30 <unsigned integer> E <scale factor>|
ps.31
```

```

ps.32 <unsigned integer> E <scale factor>
ps.33
ps.34 <scale factor> ::= <unsigned integer> | <sign> <unsigned integer>
ps.35
ps.36 <sign> ::= + | -
ps.37
ps.38 <constant identifier> ::= <identifier>
ps.39
ps.40 <string> ::= '<character> {<character>}'
ps.41
ps.42 <type definition part> ::= <empty> | type <type definition> {;<type
ps.43 definition>};
ps.44
ps.45 <type definition> ::= <identifier> = <type>
ps.46
ps.47 <type> ::= <simple type> | <structured type> | <pointer type>
ps.48
ps.49 <simple type> ::= <scalar type> | <subrange type> | <type identifier>
ps.50
ps.51 <scalar type> ::= (<identifier> {,<identifier>})
ps.52
ps.53 <subrange type> ::= <constant> .. <constant>
ps.54
ps.55 <type identifier> ::= <identifier>
ps.56
ps.57 <structured type> ::= <array type> | <record type> | <set type> | <file type>
ps.58
ps.59 <array type> ::= array [<index type>{,<index type>}] of <component type>
ps.60
ps.61 <index type> ::= <simple type>
ps.62
ps.63 <component type> ::= <type>
ps.64
ps.65 <record type> ::= record <field list> end
ps.66
ps.67 <field list> ::= <fixed part> | <fixed part> ; <variant part> | <variant part>
ps.68
ps.69 <fixed part> ::= <record section> {;<record section>}
ps.70
ps.71 <record section> ::= <field identifier> {, <field identifier>} : <type> |
ps.72 <empty>
ps.73
ps.74 <variant type> ::= case <tag field> <type identifier> of <variant> { ;
ps.75 <variant>}
ps.76
ps.77 <tag field> ::= <field identifier> : | <empty>

```

```

ps.78
ps.79 <variant> ::= <case label list> : ( <field list> ) | <empty>
ps.80
ps.81 <case label list> ::= <case label> {, <case label>}
ps.82
ps.83 <case label> ::= <constant>
ps.84
ps.85 <set type> ::= set of <base type>
ps.86
ps.87 <base type> ::= <simple type>
ps.88
ps.89 <file type> ::= file of <type>
ps.90
ps.91 <pointer type> ::= <type identifier>
ps.92
ps.93 <variable declaration part> ::= <empty> | var <variable declaration> {;
ps.94 <variable declaration>} ;
ps.95
ps.96 <variable declaration> ::= <identifier> {,<identifier>} : <type>
ps.97
ps.98 <procedure and function declaration part> ::= {<procedure or function
ps.99 declaration > ;}
ps.100
ps.101 <procedure or function declaration > ::= <procedure declaration > | <function
ps.102 declaration >
ps.103
ps.104 <procedure declaration> ::= <procedure heading> <block>
ps.105
ps.106 <procedure heading> ::= procedure <identifier> ; |
ps.107
ps.108 procedure <identifier> ( <formal parameter section> {;<formal parameter
ps.109 section>} );
ps.110
ps.111 <formal parameter section> ::= <parameter group> | var <parameter group> |
ps.112
ps.113 function <parameter group> | procedure <identifier> { , <identifier>}
ps.114
ps.115 <parameter group> ::= <identifier> {, <identifier>} : <type identifier>
ps.116
ps.117 <function declaration> ::= <function heading> <block>
ps.118
ps.119 <function heading> ::= function <identifier> : <result type> ; |
ps.120
ps.121 function <identifier> ( <formal parameter section> {;<formal parameter section>}
ps.122 ) : <result type> ;
ps.123

```

```

ps.124 <result type> ::= <type identifier>
ps.125
ps.126 <statement part> ::= <compund statement>
ps.127
ps.128 <statement> ::= <unlabelled statement> | <label> : <unlabelled statement>
ps.129
ps.130 <unlabelled statement> ::= <simple statement> | <structured statement>
ps.131
ps.132 <simple statement> ::= <assignment statement> | <procedure statement> | <go to
ps.133 statement> | <empty statement>
ps.134
ps.135 <assignment statement> ::= <variable> := <expression> | <function identifier> :=
ps.136 <expression>
ps.137
ps.138 <variable> ::= <entire variable> | <component variable> | <referenced variable>
ps.139
ps.140 <entire variable> ::= <variable identifier>
ps.141
ps.142 <variable identifier> ::= <identifier>
ps.143
ps.144 <component variable> ::= <indexed variable> | <field designator> | <file buffer>
ps.145
ps.146 <indexed variable> ::= <array variable> [<expression> {, <expression>}]
ps.147
ps.148 <array variable> ::= <variable>
ps.149
ps.150 <field designator> ::= <record variable> . <field identifier>
ps.151
ps.152 <record variable> ::= <variable>
ps.153
ps.154 <field identifier> ::= <identifier>
ps.155
ps.156 <file buffer> ::= <file variable>
ps.157
ps.158 <file variable> ::= <variable>
ps.159
ps.160 <referenced variable> ::= <pointer variable>
ps.161
ps.162 <pointer variable> ::= <variable>
ps.163
ps.164 <expression> ::= <simple expression> | <simple expression> <relational operator>
ps.165 <simple expression>
ps.166
ps.167 <relational operator> ::= = | <> | < | <= | >= | > | in
ps.168
ps.169 <simple expression> ::= <term> | <sign> <term> | <simple expression> <adding

```

```

ps.170 operator> <term>
ps.171
ps.172 <adding operator> ::= + | - | or
ps.173
ps.174 <term> ::= <factor> | <term> <multiplying operator> <factor>
ps.175
ps.176 <multiplying operator> ::= * | / | div | mod | and
ps.177
ps.178 <factor> ::= <variable> | <unsigned constant> | ( <expression> ) | <function
ps.179 designator> | <set> | not <factor>
ps.180
ps.181 <unsigned constant> ::= <unsigned number> | <string> | < constant identifier> <
ps.182 nil>
ps.183
ps.184 <function designator> ::= <function identifier> | <function identifier ( <actual
ps.185 parameter> {, <actual parameter>} )
ps.186
ps.187 <function identifier> ::= <identifier>
ps.188
ps.189 <set> ::= [ <element list> ]
ps.190
ps.191 <element list> ::= <element> {, <element> } | <empty>
ps.192
ps.193 <element> ::= <expression> | <expression> .. <expression>
ps.194
ps.195 <procedure statement> ::= <procedure identifier> | <procedure identifier>
ps.196 (<actual parameter> {, <actual parameter> })
ps.197
ps.198 <procedure identifier> ::= <identifier>
ps.199
ps.200 <actual parameter> ::= <expression> | <variable> | <procedure identifier> |
ps.201 <function identifier>
ps.202
ps.203 <go to statement> ::= goto <label>
ps.204
ps.205 <empty statement> ::= <empty>
ps.206
ps.207 <empty> ::=
ps.208
ps.209 <structured statement> ::= <compound statement> | <conditional statement> |
ps.210 <repetitive statement> | <with statement>
ps.211
ps.212 <compound statement> ::= begin <statement> {; <statement> } end;
ps.213
ps.214 <conditional statement> ::= <if statement> | <case statement>
ps.215

```

```

ps.216 <if statement> ::= if <expression> then <statement> | if <expression> then
ps.217 <statement> else <statement>
ps.218
ps.219 <case statement> ::= case <expression> of <case list element> {; <case list
ps.220 element> } end
ps.221
ps.222 <case list element> ::= <case label list> : <statement> | <empty>
ps.223
ps.224 <case label list> ::= <case label> {, <case label> }
ps.225
ps.226 <repetitive statement> ::= <while statement> | <repeat statement> | <for
ps.227 statement>
ps.228
ps.229 <while statement> ::= while <expression> do <statement>
ps.230
ps.231 <repeat statement> ::= repeat <statement> {; <statement>} until <expression>
ps.232
ps.233 <for statement> ::= for <control variable> := <for list> do <statement>
ps.234
ps.235 <control variable> ::= <identifier>
ps.236
ps.237 <for list> ::= <initial value> to <final value> | <initial value> downto <final
ps.238 value>
ps.239
ps.240 <initial value> ::= <expression>
ps.241
ps.242 <final value> ::= <expression>
ps.243
ps.244 <with statement> ::= with <record variable list> do <statement>
ps.245
ps.246 <record variable list> ::= <record variable> {, <record variable>}

```

C.2. Modula-2

```

ms.1 ident = letter {letter | digit}.
ms.2
ms.3 number = integer | real.
ms.4
ms.5 integer = digit {digit} | octalDigit {octalDigit} ("B"|"C") |
ms.6         digit {hexDigit} "H".
ms.7
ms.8 real = digit {digit} "." {digit} {ScaleFactor}.
ms.9
ms.10 ScaleFactor = "E" ["+"|"-"] digit {digit}.
ms.11

```



```

ms.12 hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
ms.13
ms.14 digit = octalDigit | "8" | "9".
ms.15
ms.16 octalDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7".
ms.17
ms.18 string = ''' {character} ''' | ''' {character} ''' .
ms.19
ms.20 qualident = ident { "." ident }.
ms.21
ms.22 ConstantDeclaration = ident "=" ConstExpression.
ms.23
ms.24 ConstExpression = expression.
ms.25
ms.26 TypeDeclaration = ident "=" type.
ms.27
ms.28 type = SimpleType | ArrayType | RecordType | SetType |
ms.29         PointerType | ProcedureType.
ms.30
ms.31 SimpleType = qualident | enumeration | SubrangeType.
ms.32
ms.33 enumeration = "(" IdentList ")".
ms.34
ms.35 IdentList = ident { "," ident }.
ms.36
ms.37 SubrangeType = [ident] "[" ConstExpression ".." ConstExpression "]" .
ms.38
ms.39 ArrayType = ARRAY SimpleType { "," SimpleType } OF type.
ms.40
ms.41 RecordType = RECORD FieldListSequence END.
ms.42
ms.43 FieldListSequence = FieldList { ";" FieldList }.
ms.44
ms.45 FieldList = [IdentList ":" type |
ms.46             CASE [ident] ":" qualident OF variant { "|" variant }
ms.47             [ELSE FieldListSequence] END].
ms.48
ms.49 variant = [CaseLabelList ":" FieldListSequence].
ms.50
ms.51 CaseLabelList = CaseLabels { "," CaseLabels }.
ms.52
ms.53 CaseLabels = ConstExpression [".." ConstExpression].
ms.54
ms.55 SetType = SET OF SimpleType.
ms.56
ms.57 PointerType = POINTER TO type.

```

```

ms.58
ms.59 ProcedureType = PROCEDURE [FormalTypeList].
ms.60
ms.61 FormalTypeList = "(" [ [VAR] FormalType
ms.62                  {",", [VAR] FormalType} ] ")" [":" qualident].
ms.63
ms.64 VariableDeclaration = IdentList ":" type.
ms.65
ms.66 designator = qualident { "." ident | "[" ExpList "]" | "^" }.
ms.67
ms.68 ExpList = expression {",", expression}.
ms.69
ms.70 expression = SimpleExpression [relation SimpleExpression].
ms.71
ms.72 relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN.
ms.73
ms.74 SimpleExpression = ["+" | "-"] term {AddOperator term}.
ms.75
ms.76 AddOperator = "+" | "-" | OR.
ms.77
ms.78 term = factor {MulOperator factor}.
ms.79
ms.80 MulOperator = "*" | "/" | DIV | MOD | AND.
ms.81
ms.82 factor = number | string | set | designator [ActualParameters] |
ms.83         "(" expression ")" | NOT factor.
ms.84
ms.85 set = [qualident] "{" [element {",", element}] "}".
ms.86
ms.87 element = expression [".." expression].
ms.88
ms.89 ActualParameters = "(" [ExpList] ")" .
ms.90
ms.91 statement = [assignment | ProcedureCall |
ms.92             IfStatement | CaseStatement | WhileStatement |
ms.93             RepeatStatement | LoopStatement | ForStatement |
ms.94             WithStatement | EXIT | RETURN [expression] ].
ms.95
ms.96 assignment = designator "==" expression.
ms.97
ms.98 ProcedureCall = designator [ActualParameters].
ms.99
ms.100 StatementSequence = statement {";", statement}.
ms.101
ms.102 IfStatement = IF expression THEN StatementSequence
ms.103             {ELSIF expression THEN StatementSequence}

```

```

ms.104             [ELSE StatementSequence] END.
ms.105
ms.106 CaseStatement = CASE expression OF case {"|" case}
ms.107             [ELSE StatementSequence] END.
ms.108
ms.109 case = [CaseLabelList ":" StatementSequence].
ms.110
ms.111 WhileStatement = WHILE expression DO StatementSequence END.
ms.112
ms.113 RepeatStatement = REPEAT StatementSequence UNTIL expression.
ms.114
ms.115 ForStatement = FOR ident ":=" expression TO expression
ms.116             [BY ConstExpression] DO StatementSequence END.
ms.117
ms.118 LoopStatement = LOOP StatementSequence END.
ms.119
ms.120 WithStatement = WITH designator DO StatementSequence END .
ms.121
ms.122 ProcedureDeclaration = ProcedureHeading ";" block ident.
ms.123
ms.124 ProcedureHeading = PROCEDURE ident [FormalParameters].
ms.125
ms.126 block = {declaration} [BEGIN StatementSequence] END.
ms.127
ms.128 declaration = CONST {ConstantDeclaration ";" } |
ms.129             TYPE {TypeDeclaration ";" } |
ms.130             VAR {VariableDeclaration ";" } |
ms.131             ProcedureDeclaration ";" | ModuleDeclaration ";".
ms.132
ms.133 FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" qualident].
ms.134
ms.135 FPSection = [VAR] IdentList ":" FormalType.
ms.136
ms.137 FormalType = [ARRAY OF] qualident.
ms.138
ms.139 ModuleDeclaration = MODULE ident [priority] ";" [import] [export] block ident.
ms.140
ms.141 priority = "[" ConstExpression "]".
ms.142
ms.143 export = EXPORT [QUALIFIED] IdentList ";".
ms.144
ms.145 import = [FROM ident] IMPORT IdentList ";".
ms.146
ms.147 DefinitionModule = DEFINITION MODULE ident ";"
ms.148
ms.149             {import} {definition} END ident "." .

```

```

ms.150
ms.151 definition = CONST {ConstantDeclaration ";" } |
ms.152             TYPE {ident ["=" type] ";" } |
ms.153             VAR {VariableDeclaration ";" } |
ms.154             ProcedureHeading ";".
ms.155
ms.156 ProgramModule = MODULE ident [priority] ";" {import} block ident ".".
ms.157
ms.158 CompilationUnit = DefinitionModule | [IMPLEMENTATION] ProgramModule.

```

C.3. Oberon

```

os.1  ident = letter {letter | digit}.
os.2
os.3  number = integer | real.
os.4
os.5  integer = digit {digit} | digit {hexDigit} "H".
os.6
os.7  real = digit {digit} "." {digit} [ScaleFactor].
os.8
os.9  ScaleFactor = ("E" | "D") ["+" | "-"] digit {digit}.
os.10
os.11 hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
os.12
os.13 digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
os.14
os.15 CharConstant = ''' character ''' | digit {hexDigit} "X".
os.16
os.17 string = ''' {character} ''' .
os.18
os.19 identdef = ident ["*"].
os.20
os.21 qualident = [ident "."] ident.
os.22
os.23 ConstantDeclaration = identdef "=" ConstExpression.
os.24
os.25 ConstExpression = expression.
os.26
os.27 TypeDeclaration = identdef "=" type.
os.28
os.29 type = qualident | ArrayType | RecordType | PointerType | ProcedureType.
os.30
os.31 ArrayType = ARRAY length {"," length} OF type.
os.32
os.33 length = ConstExpression.

```

```

os.34
os.35 RecordType = RECORD ["(" BaseType ")"] FieldListSequence END.
os.36
os.37 BaseType = qualident.
os.38
os.39 FieldListSequence = FieldList {";" FieldList}.
os.40
os.41 FieldList = [IdentList ":" type].
os.42
os.43 IdentList = identdef {"," identdef}.
os.44
os.45 PointerType = POINTER TO type.
os.46
os.47 ProcedureType = PROCEDURE [FormalParameters].
os.48
os.49 VariableDeclaration = IdentList ":" type.
os.50
os.51 designator = qualident {"." ident | "[" ExpList "]" | "(" qualident ")" | "^" }.
os.52
os.53 ExpList = expression {"," expression}.
os.54
os.55 expression = SimpleExpression [relation SimpleExpression].
os.56
os.57 relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
os.58
os.59 SimpleExpression = ["+"|"-"] term {AddOperator term}.
os.60
os.61 AddOperator = "+" | "-" | OR .
os.62
os.63 term = factor {MulOperator factor}.
os.64
os.65 MulOperator = "*" | "/" | DIV | MOD | "&" .
os.66
os.67 factor = number | CharConstant | string | NIL | set |
os.68 designator [ActualParameters] | "(" expression ")" | "~" factor.
os.69 set = "{" [element {"," element}] "}".
os.70
os.71
os.72 element = expression [".." expression].
os.73
os.74 ActualParameters = "(" [ExpList] ")" .
os.75
os.76 statement = [assignment | ProcedureCall |
os.77 IfStatement | CaseStatement | WhileStatement | RepeatStatement |
os.78 LoopStatement | ForStatement | WithStatement | EXIT | RETURN [expression] ].
os.79 assignment = designator ":=" expression.

```

```

os.80
os.81 ProcedureCall = designator [ActualParameters].
os.82
os.83 StatementSequence = statement {";" statement}.
os.84
os.85 IfStatement = IF expression THEN StatementSequence
os.86 {ELSIF expression THEN StatementSequence}
os.87 [ELSE StatementSequence] END.
os.88
os.89 CaseStatement = CASE expression OF case {"|" case}
os.90 [ELSE StatementSequence] END.
os.91
os.92 case = [CaseLabelList ":" StatementSequence].
os.93
os.94 CaseLabelList = CaseLabels {"," CaseLabels}.
os.95
os.96 CaseLabels = ConstExpression [".." ConstExpression].
os.97
os.98 WhileStatement = WHILE expression DO StatementSequence END.
os.99
os.100 RepeatStatement = REPEAT StatementSequence UNTIL expression.
os.101
os.102 LoopStatement = LOOP StatementSequence END.
os.103
os.104 ForStatement = FOR ident ":=" expression TO expression [BY ConstExpression] DO
os.105 StatementSequence END .
os.106
os.107 WithStatement = WITH qualident ":" qualident DO StatementSequence END .
os.108
os.109 ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.
os.110
os.111 ProcedureHeading = PROCEDURE ["*"] identdef [FormalParameters].
os.112
os.113 ProcedureBody = DeclarationSequence [BEGIN StatementSequence] END.
os.114
os.115 ForwardDeclaration = PROCEDURE "^" ident ["*"] [FormalParameters].
os.116
os.117 DeclarationSequence = {CONST {ConstantDeclaration ";" } |
os.118 TYPE {TypeDeclaration ";" } | VAR {VariableDeclaration ";" }}
os.119 {ProcedureDeclaration ";" | ForwardDeclaration ";"}.
os.120
os.121 FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" qualident].
os.122 FPSection = [VAR] ident {"," ident} ":" FormalType.
os.123 FormalType = {ARRAY OF} (qualident | ProcedureType).
os.124
os.125 ImportList = IMPORT import {"," import} ";" .

```

```
os.126
os.127  import = ident [":=" ident].
os.128
os.129  module = MODULE ident ";" [ImportList] DeclarationSequence
os.130  [BEGIN StatementSequence] END ident "." .
```


Apéndice D

Pascal Users group Newsletter

I. 1974

- i. Newsletter #2, Page 1: *History of Pascal documented.*
- ii. Newsletter #2, Page 6: *Wirth describes Pascal 6000-3.4.*
- iii. Newsletter #2, Page 18: *Wirth describes Pascal-P (the P-machine, probally p1).*

II. 1975

- i. Newsletter #3 , Page 1: *Pascal User Manual and Report published (assume first edition).*
- ii. Newsletter #3 , Page 4: *History of Pascal, revised..*
- iii. Newsletter #3 , Page 10: *Pascal-P2.*

III. 1976

- i. Newsletter #4 , Page 40: *Per Brinch Hansen discusses concurrent Pascal.*
- ii. Newsletter #4 , Page 81: *Pascal P4 released.*

IV. 1978

- i. Newsletter #11 Page 64: *ISO Standard Pascal progress discussion.*
- ii. Newsletter #11 Page 70: *Pascal P4 implementation notes (is P4 standard Pascal?).*
- iii. Newsletter #12 Page 7: *French/English Pascal keywords and identifiers.*
- iv. Newsletter #12 Page 17: *Application section appears with Pascal sources.*
- v. Newsletter #12 Page 33: *Analisis of Pasal design goals.*
- vi. Newsletter #13 Page 13: *Discussion of UCSD Pascal deviations/omissions from standard Pascal.*
- vii. Newsletter #13 Page 34: *Pascal prettyprinter (Hueras).*
- viii. Newsletter #13 Page 45: *Pascal prettyprinter (Condict).*

- ix. Newsletter #13 Page 83: *Letters from Wirth and others on the draft ISO standard.*
- x. Newsletter #13 Page 84: *Letter concerning test suite for ISO Pascal (Wichmann).*
- xi. Newsletter #13 Page 86: *Announcement of ANSI standards group.*
- xii. Newsletter #13 Page 92: *Lazy I/O first described.*

V. 1979

- i. Newsletter #14 Page 5: *Working draft of BSI/ISO Pascal standard.*
- ii. Newsletter #15 Page 7: *Comments on ADA.*
- iii. Newsletter #14 Page 31: *ID2ID identifier translator program.*
- iv. Newsletter #14 Page 35: *Text formatter program.*
- v. Newsletter #14 Page 62: *How to process scope in Pascal (A. Sale).*
- vi. Newsletter #14 Page 63: *Interactive Pascal-S.*
- vii. Newsletter #14 Page 90: *Pascal standards progress reports.*
- viii. Newsletter #14 Page 99: *Pascal validation suite available.*
- ix. Newsletter #14 Page 102: *Modula-2.*
- x. Newsletter #14 Page 112: *UCSD becomes commercial product.*

VI. 1980

- i. Newsletter #17 Page 12: *Report on Ada.*
- ii. Newsletter #17 Page 18: *Pascal cross reference program.*
- iii. Newsletter #17 Page 29: *Pascal macro processor.*
- iv. Newsletter #17 Page 54: *Conformant array parameters proposed.*

Apéndice E

GNU's Not UNIX!

E.1. UNIX

Era el año 1965 cuando los Laboratorios Bell⁵⁰ de AT&T⁵¹, el Instituto Tecnológico de Massachusetts⁵² y General Electric⁵³ comenzaron a trabajar en el Sistema Operativo Multics⁵⁴, acrónimo de “Multiplexed Information and Computing System”⁵⁵, diseñado para funcionar en una máquina Mainframe modelo GE-645.

El proyecto era base de las ideas de construir un Sistema Operativo que permitiese el acceso a distintos usuarios de modo simultáneo, con el objetivo principal de compartir los recursos de computación⁵⁶. Laboratorio Bell de AT&T deciden abandonar el proyecto tras distintas versiones fallidas de Multics.

Ken Thomson⁵⁷, programador de los propios laboratorios, continuó trabajando sobre la misma arquitectura original de Multics, la computadora GE-6354, desarrollando un juego espacial llamado “Space Travel”⁵⁸. El coste de cada partida se estimó en 75 dólares americanos de la época. El hecho del redimiento tan desfavorable originó que de nuevo, Ken Thomson, con la ayuda de Dennis Ritchie⁵⁹, reescribiese el juego para una computadora DEC PDP-7.

El trabajo invertido en esta nueva versión del juego, dio pie a la creación de un nuevo Sistema Operativo. Los dos programadores junto Rudd Canaday crearon gran cantidad de Software de soporte. Entre otras utilidades crearon un nuevo sistema de fichero distribuido y un potente intérprete de órdenes.

El nuevo proyecto se denominó UNICS, acrónimo de “Uniplexed Information and Computing System”⁶⁰, que era un juego de palabras sobre el citado MULTICS. Finalmente pasó a denominarse UNIX por influencia del hacker de MULTICS Brian Kernighan⁶¹.

El proyecto UNIX finalmente fue financiado por los Laboratorios Bell, debido a que se trabajó para máquinas superiores de la propia familia PDP⁶². En concreto sobre el conjunto [PDP-11, PDP-20].

Corría el año 1970 y UNIX era una realidad. Destaca de las primeras versiones el editor de textos `rundoff`⁶³. Todas las versiones de UNIX escritas antes de 1972 estaban codificadas en Lenguaje Ensamblador de las propias computadoras. Este año se produjo un hito histórico para el Sistema UNIX, y es la reescritura prácticamente completa del propio sistema a Lenguaje C⁶⁴.

La probabilidad de UNIX era una realidad, y el propio AT&T dotó a universidades y empresas de este nuevo sistema. Destaca de las versiones de la matriz UNIX, la distribución BSD (Berkley Software Distribution). El propio AT&T creó un departamento para comercializar UNIX. Su desarrollo, en distintas versiones, se discontinuó con la versión 7, en el año 1979. A

principios de la década de los 80, AT&T inició el proyecto Plan 9 del que ya era Software base el Sistema X-Window del MIT.

Le siguió un intento comercial sobre UNIX 7 denominada UNIX System III. Por aquella época la dispersión de fabricantes era enorme.

En 1993, la empresa Novell adquirió los laboratorios de UNIX de AT&T. En ese mismo año la versión BSD evolucionada en el tiempo, era constantemente demandada debido a problemas de patentes.

En 1995 Novell creó su propia versión de UNIX llamada UNIX-Ware.

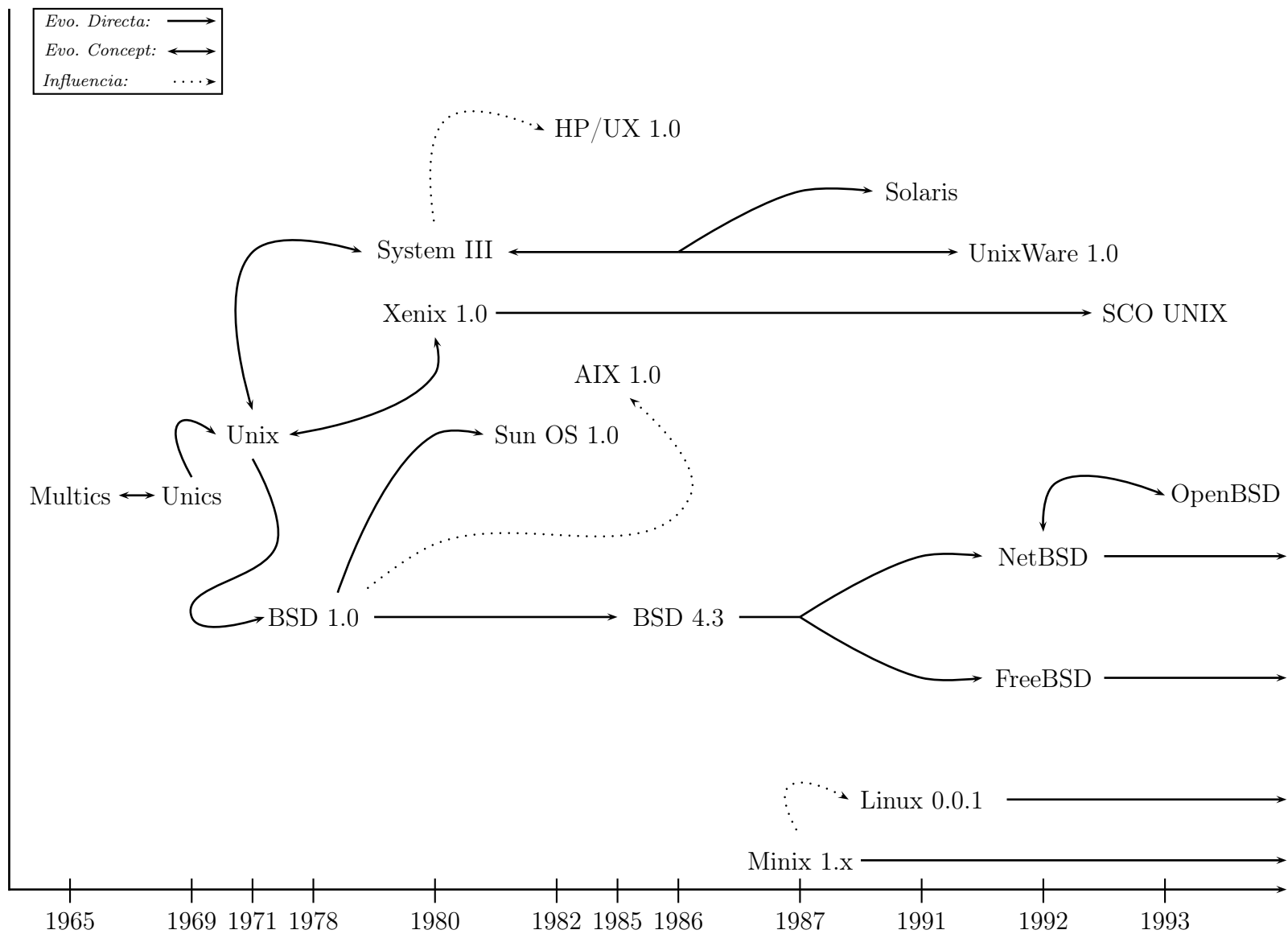


Figura E.1: Evolución de UNIX y sus distintas versiones en el tiempo.

E.2. Proyecto GNU

Lo cierto es que la historia que contamos nace en 1983 con la creación de GNU⁶⁵ [Sta85] por parte de Richard Stallman⁶⁶. Dos años después se crea la asociación “Free Software Foundation”⁶⁷ con el objetivo de promover el Software en torno a ciertas libertades, las cuales se consolidan con GPL⁶⁸, la licencia general y pública de GNU.

Gran cantidad de Software se desarrolló entre la última etapa de la década de los ochenta y principios de los noventa del siglo XX. Era tal, dicha cantidad, que se podría construir un clon de UNIX a falta de un núcleo en proyecto llamado Hurd⁶⁹. El problema surgía de una mala planificación y la insistencia por crear un MiniKernel, que conllevaba a una excesiva burocracia en la etapa de solución para cualquier posible fallo. Y es que, por entonces, un estudiante finlandés, Linus Torvalds⁷⁰, primero trabajando con grandes máquina y poco después con el 8386 dónde a partir de Minix⁷¹, adaptándolo a sus necesidades escribe y extiende funcionalidades que finalmente constituirían un nuevo núcleo muy discreto.

E.2.1. Licencia GPL

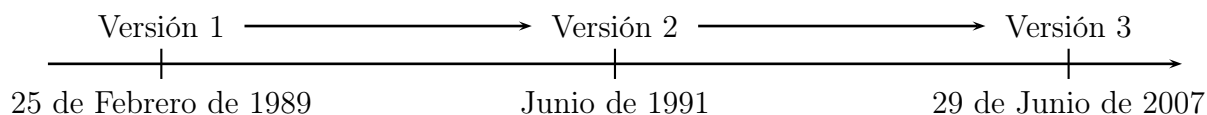


Figura E.2: Evolución de la Licencia GPL.

La General Public License (Licencia Pública de GNU) se trata de un marco legal para desarrolladores y Software creada por Free Software Foundation (Fundación de Software Libre).

Su primera versión se hizo pública el 25 de Febrero de 1989. Con este nuevo “marco legal” se pretendía consolidar el proyecto GNU que a su vez tenía su esencia espiritual en la cultura Hacker de los años sesenta del siglo XX. Los desarrollos que decidieran distribuir su Software bajo Copyright de GNU permitían que se derivasen copias del mismo sin tener por ello una remuneración. Al igual, se obligaba a que aquellos que mejorasen o ampliaran las características de un programa con la licencia GPL publicasen dichas mejoras para que la comunidad se siguiera enriqueciendo y creciendo.

La licencia GPL es compatible con otro tipo de licencias “Libres” partiendo de la base de que estas últimas se basan en los postulados de la propia GPL.

De igual manera se han creado licencias basadas en GPL que añaden restricciones para el uso y compartición del Software. Es la respuesta de la industria ante la proliferación de la cultura GNU.

La licencia GPL ha tenido tres versiones oficiales hasta la fecha:

- I. GPL versión 1: Como hemos citado anteriormente fue publicada el 25 de Febrero de 1989 bajo las siguientes ideas:
 - i. Dado el problema que plantea el Software comercial con su política de distribución de Software en formato ejecutable (binario), la licencia GPL determina que los programas que se distribuyan con su Copyrights deben ir acompañados del código fuente.

- ii. El segundo problema que intenta solucionar es el de la compatibilidad con Software comercial, no tanto para oponerse a el, sino para proteger los programas GPL. Por ello, la versiones modificadas de un Software GPL deben acerse públicas.

II. GPL versión 2: Fue publicada en Junio de 1991 con los objetivos:

- i. Hacer más restrictiva la publicación del Software, añadiendo la imperiosa necesidad de acompañar a un fichero ejecutable de su código fuente.
- ii. De igual manera y basado en los problemas que presentaban las Bibliotecas GNU con Software comercial, se creo una licencia específica para las misma; *Library General Public License* (Licencia Pública General de Bibliotecas) bajo la misma idea de que se deben abrir sus códigos fuente.

III. GPL versión 3: Tras un largo periodo de trabajo (el proyecto comienza en 2005) finalmente el 29 de Junio de 2007 se hizo pública la versión 3 de la licencia GPL.

Entre sus novedades destacan:

- i. Nuevas clausulas para formatos industriales restrictivos, como DRM (Gestión Digital de Derechos).
- ii. Resolución de ambigüedades.
- iii. Adaptar la GPL a los marcos jurídicos de los distintos países.
- iv. Proteger a desarrolles de Software Libre frente a las patentes.

Notas

⁵⁰<http://www.alcatel-lucent.com/wps/portal/BellLabs>

⁵¹<http://www.att.com/>

⁵²<http://web.mit.edu/>

⁵³<http://www.ge.com/>

⁵⁴<http://www.multicians.org/>

⁵⁵“Sistema de Computación e Información Multiplexada”

⁵⁶En aquella época el coste de computo era realmente elevado debido en gran parte al consumo energético.

⁵⁷**Ken Thomson**, nació en Nueva Orleans (New Orleans, EEUU) el 4 de Febrero de 1943. Se trata de un prestigioso científico de la computación conocido históricamente como uno de los creadores del Sistema Operativo UNIX. Thomson se gradúa en 1966 como Ingeniero Electrónico y de Ciencias de la Computación por la Universidad de Berkeley (California). Comenzó a trabajar para los Laboratorio Bell en el proyecto Multics bajo la supervisión de Elwyn Berlekamp.

A comienzos de los años sesenta del siglo XX, Thomson en compañía de Ritchie continuaron en solitario (sin financiación) una nueva versión de Multics. La base de esta nueva implementación gira en torno a las utilidades Software creadas para el juego "Space Travel". En 1969 se publica la primera versión de este Sistema Operativo llamado UNICS (posteriormente UNIX).

Thomson también ha trabajado en el editor QED, que hace un potente uso de las expresiones regulares, base de posteriores lenguajes como Perl.

Otro hito de la historia de Ken Thomson es su participación en el desarrollo de UTF-8 junto a Rob Pike en 1992.

En el año 2000 Thomson deja los Laboratorios Bell para en 2006 trabajar como ingeniero distinguido en la empresa Google Inc.

Entre todos sus reconocimientos por su trabajo, destacamos los siguientes:

- i. En 1983, junto a Ritchie recibe el prestigioso premio Turing.
- ii. En 1990, de nuevo con Denis Ritchie recibe el premio IEEE Richard W. Hamming Medal del instituto IEEE (Institute of Electrical and Electronics Engineers).
- iii. En 1997 entra a formar parte del Museo Histórico de la Computación (Computer History Museum) con su compañero Ritchie.
- iv. En 1999 Thomson es elegido por IEEE para recibir el primer premio Tsutomu Kanai Award.

⁵⁸“Viaje Espacial”

⁵⁹**Dennis MacAlistair Ritchie** (también conocido como drmm), nacido en 9 de Septiembre de 1941 en Bronxville Nueva York (New York, EEUU). Es conocido por ser el padre del Lenguaje de Programación C.

Ritchie trabajó para los Laboratorio Bell de AT&T donde junto a su compañero Ken Thomson creó el Sistema Operativo UNIX (1969).

Junto a Ken Thomson ha sido galardonado con prestigiosos premios como: el Premio Turing (1983) por ACM, la Medalla Hamming (1990) por IEEE y la Medalla de la Tecnología (1990) de manos del presidente Bill Clinton.

Ritchie se retiró en el año 2007. El 12 de Octubre de 2011, a la edad de 70 años, falleció por un fallo cardíaco (tras padecer un largo cáncer) en Nueva Jersey (New Jersey, EEUU).

⁶⁰“Sistema de Computación e Información Uniplexada”

⁶¹**Brian Wilson Kernighan** (también conocido como ‘K’), nació en Toronto (Canada) en 1942. Es uno de los científicos más influyentes en la computación de la última mitad del siglo XX.

Trabajó en los Laboratorios Bell de AT&T inicialmente en el Proyecto Multics para después colaborar activamente en el desarrollo de UNIX. A Kernighan se le debe el nombre de UNIX, puesto que el Sistema Operativo creado por Ritchie y Thomson originalmente recibía el nombre de UNICS, basándose en el acrónimo MULTICS.

Kernighan es un autor muy prolífico. Su firma está en obras y Software como:

- i. “C Programming Language” con Dennis Ritchie, aunque sus palabras dicen lo contrario: “it’s entirely Dennis Ritchie’s work”.
- ii. Es coautor de los lenguajes: AWK y AMPL.

- iii. Ha trabajado con Shen Lin en métodos de optimización de problemas NP-complete, gráficos particionados y problemas del viajero.
- iv. Es también conocido por la frase: (WYSIAYG) "What You See Is All You Get"(Lo que ves es todo lo que obtienes), que deriva de: (WYSIWYG) "What You See Is What You Get"(Lo que ves es lo que obtienes).

Actualmente Kernighan es profesor en el Departamento de Ciencias de la Computación de la Universidad de Princeton (Princeton University).

⁶²Programmed Data Processor (PDP) was a series of minicomputers made and marketed by the Digital Equipment Corporation from 1957 to 1990. The name 'PDP' intentionally avoided the use of the term 'computer' because, at the time of the first PDPs, computers had a reputation of being large, complicated, and expensive machines, and the venture capitalists behind Digital (especially Georges Doriot) would not support Digital's attempting to build a computer"; the word "minicomputer" had not yet been coined. So instead, Digital used their existing line of logic modules to build a Programmable Data Processor and aimed it at a market which could not afford the larger computers.

⁶³ <http://www.gnu.org/software/groff/>

⁶⁴Decir que partes de UNIX para aquel entonces y aun hoy en día se mantienen en código nativo de cada procesador.

⁶⁵<http://www.gnu.org/>

⁶⁶**Richard Matthew Stallman** nació el 16 de Marzo de 1953 en la Ciudad de Nueva York (New York City, EEUU). Stallman es un reconocido activista a favor de los derechos (en cuestión de ciertas libertades) de los usuarios de computadoras.

Richard Stallman se gradúa como físico por la Universidad de Harvard en 1974. Poco después trabaja y realiza un post-gradó en el prestigioso MIT. Allí conoció: "MIT's hacker culture"(La cultura Hacker del MIT) donde los códigos fuente de los programas eran compartidos por los desarrolladores. El problema de acceso a las fuentes del programa de cierto Hardware durante su trabajo en el MIT, incita a Stallman, a crear una organización que se basase en los principios de que el Software es .abierto. Dicho de otro modo, el Software debe ser accesible para los usuarios, respetando su/su autor/ autores, pero siempre con la idea de que teniendo este código, será mucho más sencillo adaptar el Hardware a las distintas necesidades de un usuario.

Con esta idea crea en 1983 "GNU Project"(Proyecto GNU) para crear un clon libre de UNIX. Con el objeto de asentar una cultura "para el movimiento libre del Software", en 1985 funda "The Free Software Foundation"(Fundación del Software Libre).

Durante estos años fue uno de los mayores contribuyentes del código para el Proyecto GNU. Dado que como dice textualmente: "Yo no puedo crear Leyes", decide que el Software de GNU sea distribuido bajo una licencia llamada "GNU General Public License"(Licencia Pública General de GNU).

El 25 de Febrero de 1989 publica la primera versión de la denominada, GPL, que ha sufrido a lo largo de su historia, tres revisiones.

⁶⁷<http://www.fsf.org/>

⁶⁸ <http://www.gnu.org/copyleft/gpl.html>

⁶⁹ <http://www.gnu.org/software/hurd/>

⁷⁰**Linus Benedict Torvalds** nació el 28 de Diciembre de 1969 en Helsinki (Finlandia). Linus es conocido en el mundo entero por ser el primer creador y actual máximo responsable del Kernel Linux.

Además Linus es cocreador del Software de Control de Versiones Git. Durante los años 1988 y 1996 se gradúa en Ciencias de la Computación por la Universidad de Helsinki.

Su formación académica fue interrumpida por distintos motivos, entre ellos el servicio militar. Poco después de finalizar su preparación obligatoria militar y ya de nuevo en la universidad tiene sus primeros contactos con el Sistema Operativo UNIX.

Primero se forma con grandes máquinas para después, y sobre una computadora PC-Compatible con un procesador 80386 de Intel, empezar a mejorar una copia de MINIX. Su Software base era el propio MINIX y el compilador GCC.

Finalmente el 5 de Junio de 1991 anuncia en el grupo comp.os.minix su nuevo sistema e invita a otros usuarios a participen en el para que a través de sus sugerencias, Linus haga evolucionar el primitivo Linux (en su famoso correo electrónico aclara que no promete incorporar en el Software todas las propuestas).

Linus originalmente llama a su trabajo "Freax" (un juego de palabras entre "Free" y X) en alusión a UNIX.

Torvalds decide licenciar al ya denominado Linux bajo el copyright de GNU (GPL) en base a la idea de que el compilador que usa es el popular GCC.

Linus es cocreador de Git, un popular sistema de control de versiones que tuvo su primera versión disponible para el público el 7 de Abril de 2005. El origen del proyecto es parte de las insistentes críticas de "Linux Foundation" y el propio Torvals de CVS (un popular Sistema de Control de Versiones gratuito).

⁷¹<http://www.minix3.org/>

Apéndice F

Linux

Parece ser que el punto de partida se fija el 25 de agosto de 1991, 20:57:08 GMT, con el siguiente correo electrónico⁷²:

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID:
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki
```

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-).

F.1. Historia

Linux nace a principio de los años noventa del siglo XX. Fue el resultado de enormes esfuerzos por parte de la cultura “Hacker” para tener la posibilidad de ejecutar un Sistema Operativo fiable y abierto como alternativa a los privativos y comerciales: MS-DOS de Microsoft y Mac OS de Apple.

La base de para Linux fue el código de Minix que era a su vez, un prototipo funcional de UNIX con alrededor de 2000 líneas de código y disponible para cualquier estudiante que tuviera acceso al manual “*The Design of Operating Systems*” del profesor Tanenbaum.

Además y como hemos citado anteriormente, el Proyecto GNU proporcionaba un potente compilador GCC (GNU C Compiler) además de otras herramientas para y el desarrollo y una Licencia que daba al propio proyecto un “marco legal”.

Se podría afirmar que Linux no era más que un hobby de un estudiante de segundo curso de Informática en la Universidad de Helsinki.

Tras el correo escribo por Linus a la propia comunidad de Minix ver apartado (F). fue publicada la versión 0.0.1 en Septiembre de 1991 que a su vez sirvió como recurso publicitario para una comunidad que empezaba a usar Internet y que era capaz de conectar grandes distancias en un tiempo muy corto. El trabajo inicialmente era una mejora de Minix, las versiones 0.10 y 0.11 (sobretudo está última) comenzador a dar forma a una pieza de un Sistema Operativo, un nuevo núcleo para Sistemas GNU.

Dichas versiones incluían soporte para: VGA, EGA, controladoras Floppy y múltiple “key-maps” fruto del trabajo de la comunidad.

El trabajo intenso de los desarrolladores provocó que la siguiente versión a la 0.12 fuera la 0.95. Linux fue duramente criticado por instituciones académicas dado que heredaba la arquitectura monolítica de Minix. Por ello y tras diversas discusiones se modificó la arquitectura del propio Kernel.

No debemos olvidar que gran parte de su éxito se debe al empuje comercial y la creación de importantes empresas en torno al propio Linux y otro Software GNU.

Debemos destacar importantes empresas como Red Hat o Caldera. De igual manera, programadores y aficionados de todo el mundo comenzaron a trabajar en sus propias versiones GNU/Linux por ejemplo: Ian Murdock con Debian o Peter Volkerding con Slackware.

Gracias a la participación de miles de desarrolladores, Linux es hoy día uno de los productos con capacidad de ejecutarse en múltiples dispositivos con arquitecturas muy distintas.

En Marzo de 1994 se publicó la versión 1.0 estable del Kernel.

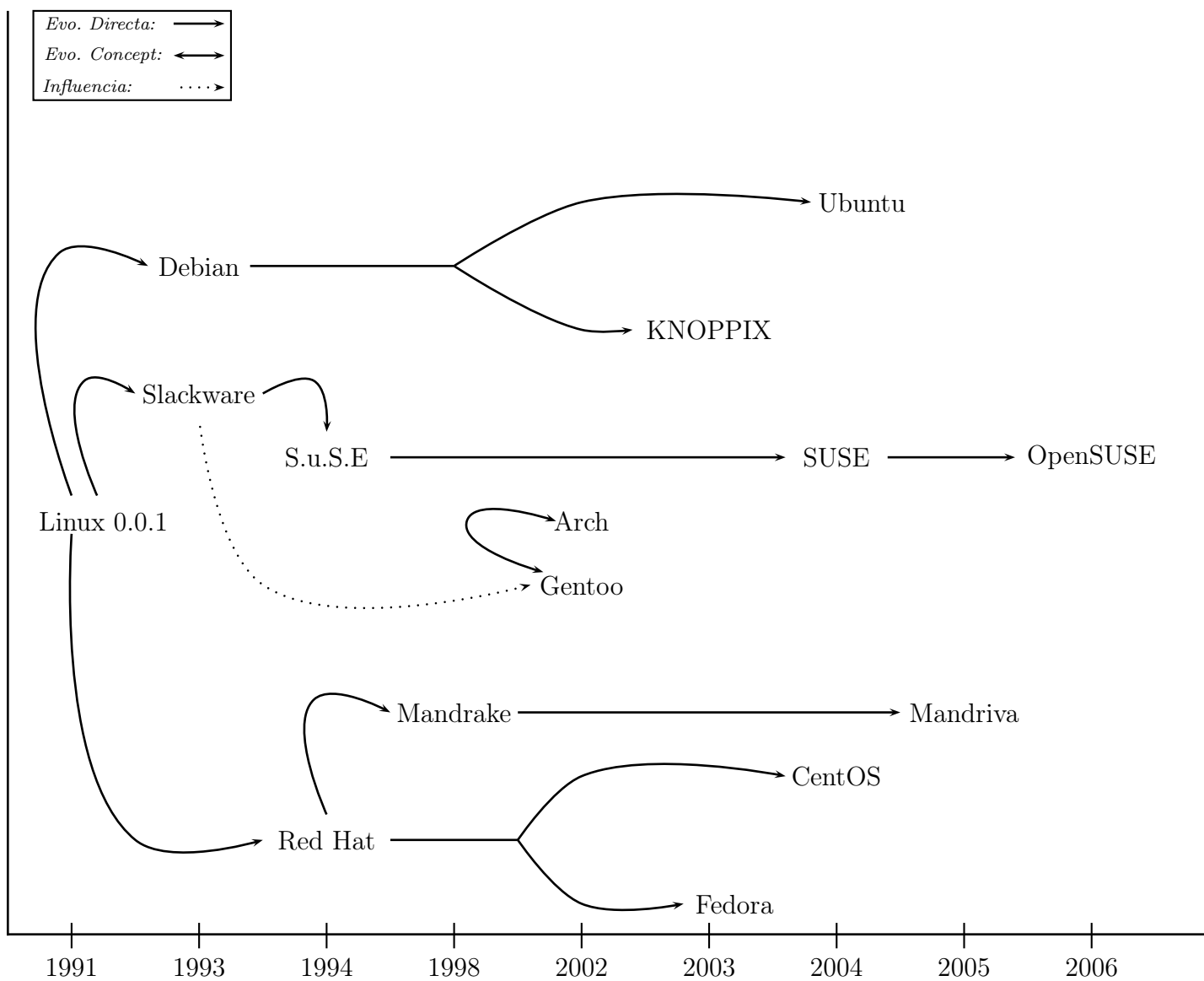


Figura F.1: Evolución de Linux y distintas distribuciones en el tiempo.

Notas

72

Hola a todos aquellos que usan Minix -

Estoy haciendo un sistema operativo (libre) (sólo un hobby, no será grande y profesional como GNU) para 386 (486) AT clones. Esta ha sido la cerveza desde abril, y está empezando a prepararse. Me gustaría algún comentario sobre cosas que la gente le gusta / disgusta de minix, como mi sistema operativo se asemeja un poco (la misma disposición física del sistema de archivos (por razones prácticas) entre otras cosas).

Me he portado bash (1.08) y gcc (1.40), y las cosas parecen funcionar.

Esto implica que tendré algo práctico dentro de unos meses, y

Me gustaría saber qué características la mayoría de la gente quiere. cualquier sugerencia son bienvenidos, pero no voy a prometer voy a ponerlas en práctica :-)

Linus (torvalds@kruuna.helsinki.fi)

PS. Sí - es libre de cualquier código de minix, y tiene un sistema de archivos multi-hilo. NO es protable (386 utiliza la conmutación de tareas, etc), y probablemente nunca apoyará nada que no sea AT-discos duros, ya que es todo lo que he :-).

Apéndice G

Tabla ASCII $[0, (2^8 - 1)]$

000d	00h	☐	(nul)	032d	20h	˘	064d	40h	@	096d	60h	‘
001d	01h	☉	(soh)	033d	21h	!	065d	41h	A	097d	61h	a
002d	02h	☼	(stx)	034d	22h	"	066d	42h	B	098d	62h	b
003d	03h	♥	(etx)	035d	23h	#	067d	43h	C	099d	63h	c
004d	04h	♦	(eot)	036d	24h	\$	068d	44h	D	100d	64h	d
005d	05h	♣	(enq)	037d	25h	%	069d	45h	E	101d	65h	e
006d	06h	♠	(ack)	038d	26h	&	070d	46h	F	102d	66h	f
007d	07h	•	(bel)	039d	27h	'	071d	47h	G	103d	67h	g
008d	08h	▣	(bs)	040d	28h	(072d	48h	H	104d	68h	h
009d	09h		(tab)	041d	29h)	073d	49h	I	105d	69h	i
010d	0Ah	▣	(lf)	042d	2Ah	*	074d	4Ah	J	106d	6Ah	j
011d	0Bh	♂	(vt)	043d	2Bh	+	075d	4Bh	K	107d	6Bh	k
012d	0Ch		(np)	044d	2Ch	,	076d	4Ch	L	108d	6Ch	l
013d	0Dh	♪	(cr)	045d	2Dh	-	077d	4Dh	M	109d	6Dh	m
014d	0Eh	♪	(so)	046d	2Eh	.	078d	4Eh	N	110d	6Eh	n
015d	0Fh	✱	(si)	047d	2Fh	/	079d	4Fh	O	111d	6Fh	o
016d	10h	►	(dle)	048d	30h	0	080d	50h	P	112d	70h	p
017d	11h	◄	(dc1)	049d	31h	1	081d	51h	Q	113d	71h	q
018d	12h	‡	(dc2)	050d	32h	2	082d	52h	R	114d	72h	r
019d	13h	‡	(dc3)	051d	33h	3	083d	53h	S	115d	73h	s
020d	14h	‡	(dc4)	052d	34h	4	084d	54h	T	116d	74h	t
021d	15h	§	(nak)	053d	35h	5	085d	55h	U	117d	75h	u
022d	16h	—	(syn)	054d	36h	6	086d	56h	V	118d	76h	v
023d	17h	‡	(etb)	055d	37h	7	087d	57h	W	119d	77h	w
024d	18h	↑	(can)	056d	38h	8	088d	58h	X	120d	78h	x
025d	19h	↓	(em)	057d	39h	9	089d	59h	Y	121d	79h	y
026d	1Ah		(eof)	058d	3Ah	:	090d	5Ah	Z	122d	7Ah	z
027d	1Bh	←	(esc)	059d	3Bh	;	091d	5Bh	[123d	7Bh	{
028d	1Ch	ℓ	(fs)	060d	3Ch	<	092d	5Ch	\	124d	7Ch	
029d	1Dh	↔	(gs)	061d	3Dh	=	093d	5Dh]	125d	7Dh	}
030d	1Eh	▲	(rs)	062d	3Eh	>	094d	5Eh	^	126d	7Eh	~
031d	1Fh	▼	(us)	063d	3Fh	?	095d	5Fh	_	127d	7Fh	△

Tabla G.1: Tabla ASCII $[0, (2^7 - 1)]$

128d	80h	€	160d	A0h	‰	192d	C0h	À	224d	E0h	à
129d	81h		161d	A1h	¡	193d	C1h	Á	225d	E1h	á
130d	82h	,	162d	A2h	¢	194d	C2h	Â	226d	E2h	â
131d	83h	f	163d	A3h	£	195d	C3h	Ã	227d	E3h	ã
132d	84h	„	164d	A4h	¤	196d	C4h	Ä	228d	E4h	ä
133d	85h	...	165d	A5h	¥	197d	C5h	Å	229d	E5h	å
134d	86h	†	166d	A6h	¦	198d	C6h	Æ	230d	E6h	æ
135d	87h	‡	167d	A7h	§	199d	C7h	Ç	231d	E7h	ç
136d	88h	ˆ	168d	A8h	¨	200d	C8h	È	232d	E8h	è
137d	89h	‰	169d	A9h	©	201d	C9h	É	233d	E9h	é
138d	8Ah	Š	170d	AAh	ª	202d	CAh	Ê	234d	EAh	ê
139d	8Bh	<	171d	ABh	«	203d	CBh	Ë	235d	EBh	ë
140d	8Ch	Ǝ	172d	ACH	¬	204d	CCh	Ì	236d	ECh	ì
141d	8Dh		173d	ADh		205d	CDh	Í	237d	EDh	í
142d	8Eh	Ž	174d	AEnh	®	206d	CEh	Î	238d	EEh	î
143d	8Fh		175d	AFh	¯	207d	CFh	Ï	239d	EFh	ï
144d	90h		176d	B0h	°	208d	D0h	Ð	240d	F0h	ð
145d	91h	‘	177d	B1h	±	209d	D1h	Ñ	241d	F1h	ñ
146d	92h	’	178d	B2h	²	210d	D2h	Ò	242d	F2h	ò
147d	93h	“	179d	B3h	³	211d	D3h	Ó	243d	F3h	ó
148d	94h	”	180d	B4h	´	212d	D4h	Ô	244d	F4h	ô
149d	95h	•	181d	B5h	µ	213d	D5h	Õ	245d	F5h	õ
150d	96h	—	182d	B6h	¶	214d	D6h	Ö	246d	F6h	ö
151d	97h	—	183d	B7h	·	215d	D7h	×	247d	F7h	÷
152d	98h	~	184d	B8h	¸	216d	D8h	Ø	248d	F8h	ø
153d	99h	™	185d	B9h	¹	217d	D9h	Ù	249d	F9h	ù
154d	9Ah	š	186d	BAh	º	218d	DAh	Ú	250d	FAh	ú
155d	9Bh	>	187d	BBh	»	219d	DBh	Û	251d	FBh	û
156d	9Ch	œ	188d	BCh	¼	220d	DCh	Ü	252d	FCh	ü
157d	9Dh		189d	BDh	½	221d	DDh	Ý	253d	FDh	ý
158d	9Eh	ž	190d	BEh	¾	222d	DEh	Þ	254d	FEh	þ
159d	9Fh	ÿ	191d	BFh	¿	223d	DFh	ß	255d	FFh	ÿ

Tabla G.2: Tabla ASCII $[2^7, (2^8 - 1)]$

Apéndice H

GNU General Public License (Version 2, June 1991)

Copyright © 1989, 1991 Free Software Foundation, Inc.

51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author’s protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors’ reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but

does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
 - a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

one line to give the program’s name and a brief idea of what it does.
Copyright (C) yyyy name of author

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) yyyy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands **show w** and **show c** should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than **show w** and **show c**; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Bibliografía

- [AVAU98] John E. Hopcroft Alfred V. Aho and Jeffrey D. Ullman. *Estructuras de datos y algoritmos*. Pearson, 1998.
- [Bac86] Maurice J. Bach. *The Desing of the Unix Operating System*. Prentice/Hall International, Inc, 1986.
- [Cho59] Noam Chomsky. On Certain Formal Properties of Grammars*. *Information and Control Volume 2, Issue 2, June 1959, Pages 137-167*, June 1959.
- [dB96] Juan de Burgos. *Calculo Infinitesimal de una Variable 2ed*. Mc Graw Hill, 1996.
- [dB06] Juan de Burgos. *Álgebra Lineal y Geometría Cartesiana 3ed*. Mc Graw Hill, 2006.
- [DW87] Neil Dale and Chip Weems. *Introduction to Pascal Structure Desing 2ed*. Hearth and Company, 1987.
- [ea60] J. W. Backus et al. Report on the algorithmic language ALGOL 60. *Numerische Mathematik Volume 2, Number 1, 106-136, DOI: 10.1007/BF01386216*, 1960.
- [ea97] Cristóbal Pareja et al. *Desarrollo de Algoritmos y Técnicas de Programación en Pascal*. Ra-Ma, Octubre 1997.
- [ea07a] Alfred V. Aho et al. *Compiladores: Principios, Técnicas y Herramientas 2ed*. Pearson Education, 2007.
- [ea07b] Alfred V. Aho et al. *Compilers: Principles, Techniques and Tools 2ed*. Pearson Education, 2007.
- [ISO91] ISO/IEC. Pascal ISO 90 (ISO/IEC 7185:1990). *ISO/IEC*, 1991.
- [ISO04] ISO/IEC. Fortran ISO 2003 (ISO/IEC DIS 1539-1:2004). *ISO/IEC*, 2004.
- [ISO10] ISO/IEC. Fortran ISO 2008 (ISO/IEC 1539-1:2010). *ISO/IEC*, 2010.
- [KP87] Brian W. Kernighan and Rob Pike. *El Entorno de Programación UNIX*. Prentice-Hall, 1987.
- [Mar04] Ricardo Peña Marí. *Diseño de Programas. Formalismos y Abstracción*. Pearson, 2004.
- [Mau08] Wolfgang Mauerer. *Professional Linux® Kernel Architecture*. Wiley Publishing, Inc, 2008.

- [Mer05] Félix García Merayo. *Matemática Discreta*. Thomson, 2005.
- [MNN05] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2005.
- [Roj10] Miguel Ángel Quintans Rojo. Apuntes de la asignatura: Lenguajes de Programación. *Ingeniería Técnica en Informática de Gestión*. Universidad de Alcalá, 2010.
- [Ros04] Kenneth H. Rosen. *Matemática Discreta y sus Aplicaciones 5ed.* Mc Graw Hill, 2004.
- [Sta66] American National Standard. FORTRAN 66 (USA Standard FORTRAN). *American National Standard*, March 7, 1966.
- [Sta85] Richard Stallman. The GNU manifesto. *j-DDJ*, 10(3):30–??, mar 1985.
- [Vah96] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice-Hall, 1996.
- [vdHea05] Jan-Jaap van der Heijden et al. *The GNU Pascal Manual*. GPC Web, 2005.
- [vH06] William von Hagen. *The Definitive Guide to GCC*. Apress, 2006.
- [WG05] N. Wirth and J. Gutknecht. *Project Oberon - The Design of an Operating System and Compiler*. Addison-Wesley, 2005.
- [Wir71] Nicklaus Wirth. The Programming Language Pascal. *Acta Informatica Volume 1, Number 1*, 35-63, DOI: 10.1007/BF00264291, 1971.
- [Wir80] Nicklaus Wirth. Modula-2. *Institut für Informatik ETH Zürich Technical Report 36*, 1980.
- [Wir88] Nicklaus Wirth. *The programming language Oberon*. Software: Practice and Experience Volume 18, Issue 7, pages 671–690, July 1988.
- [yDC04] José María Vall y David Camacho. *Programación Estructurada y Algoritmos en Pascal*. Pearson, 2004.

Índice alfabético

Símbolos

$CLAUS_{\lambda}$, 99

Álgebra de Boole, 19

Árbol, 27

Árbol Generador, 27

Árbol con Raíz, 30

Árboles Binarios, 31

25 de agosto de 1991, 20:57:08 GMT, 173

Números

8386, 166

A

ADA, 69

Ada, 62

Adriann van Wijgaarden, 51

Alfabeto, 89

ALGOL, 50

ALGOL 58, 50

ALGOL 60, 50

ALGOL 68, 51

ALGOL W, 51

ALGOL X, 53

Algoritmo de Kruskal, 29

Algoritmo de Prim, 27

Algoritmos de Búsqueda, 39

Algoritmos de Ordenación, 39

Analizador Léxico, 35, 87

Analizador Sintáctico, 36

Analizadores con Recuperación, 88

Augusta Ada King, 62

Autómata Finito Determinista, 37

Autómata Finito, 96

Autómata Finito Determinista, 97

Autómata Finito no Determinista, 37, 98

B

Back-End, 77

Backus-Naur Form, 50

Bipartito, 25

Bison, 35

Blaise Pascal, 45, 131

Boolean, 55

Borland, 74

Borland Turbo Pascal, 70

Bottom-Up-Parser, 38

bytecodes, 64

C

Calculadora de Pascal, 74

CDC 6000, 70

Cerradura de Klennee, 93, 95

Cerradura Estrella, 93

Char, 55

Charles Babbage, 62

Ciclo Hamiltoniano, 26

Cierre λ , 99

Circuito Eulero, 26

Clausura, 93

Clausura Positiva, 94

Complementario, 10

Complemento, 93

Concatenación, 89, 93, 95

Conjunto, 5

Conjunto de Variables Booleanas, 20

Conjunto por Compresión, 5

Conjunto por Extensión, 5

Conjunto Universal, 6

Conjunto Vacío, 6

D

Declaration Part, 60

Diferencia, 92

Diferencia Simétrica, 9

Disjuntos, 9

Divide and Conquer, 63

E

Entorno de Desarrollo, 74
Errores Léxicos, 87
Expresión Regular, 37
Expresiones Regulares, 95

F

Flex, 35
Fortran, 46
Fortran 2003, 48
Fortran 2008, 48
FORTRAN 66, 47
FORTRAN 77, 47
Fortran 90, 47
Fortran 95, 47
Free Pascal, 43
Free Software Foundation, 166
Front-End, 77
Full Pascal, 70
Función, 15
Funciones Aritméticas, 58
Funciones Biyectivas, 17
Funciones Booleanas, 59
Funciones de Transferencia, 58
Funciones Estándar, 76
Funciones Exhaustivas o Suprayectiva, 16
Funciones Inyectivas, 17
Funciones Ordinales, 59

G

GCC (GNU Compiler Collection), 43
GNU Assembler, 77
GNU Fortran, 49
GNU Modula-2, 62
GNU NYU Ada Translator, 63
GNU Pascal Compiler, 43, 77
GNU/Linux, 74
gp1990c, 35
gp1990la, 37
gp1990sa, 38
GPL, 166
Grafo, 21
Grafo Completo, 25
Grafo Dirigido, 24
Grafo Multigrafo, 23
Grafo no Simple, 24
Grafo Regular, 25
Grafo Simple, 23

H

Herencia, 61

I

i386, 74
IBM 360, 53
IBM 704, 47
IBM-PC, 72
IEEE, 54
Integer, 55
Intersección, 7, 92
IP Pascal, 74
ISO 10206, 77
ISO 7185, 77
ISO Pascal 7185:1990, 71
Isomorfos, 24

J

Java, 64

L

Lenguaje, 91
Lenguaje de Publicaciones, 50
Lenguaje de Referencia, 50
Lenguaje Ensamblador, 37
Lenguaje Finito, 95
Lenguaje Fortran, 70
Lenguaje Pascal, 70
Lenguaje Vacio, 91
Lenguajes C y C++, 37
Lenguajes Regular, 95
LEX, 103
Lex, 37
Lexema, 87
Longitud, 90
Lower-Case, 67

M

Milestone, 46
MiniKernel, 166
Modo Pánico, 88
Modula, 61
Motorola 68000, 72
MS-DOS, 74

N

Niklaus Wirth, 45

O

Oberon, 64

P

p-code, 71

p-code Operating Systems, 71

Palabra, 89

Palabra Vacía, 89

Par, 11

Pascal, 45

Pascal 1971, 70

Pascal P2, 70

Pascal P3 y P4, 71

Pascal P5, 71

Pascal P6, 71

Pascal P7, 71

Pascal-P, 70

Pascalina, 131

Pascaline, 74

Patrón, 87

PC-DOS, 71

Potencia, 90, 93

Procedimientos, 57

Procedimientos Estándar, 75

Producto Cartesiano, 11

Program Heading, 59

Programación Orientada a Objetos, 61

pseudo-machine, 70

R

Real, 55

Reflexión, 91, 94

Relación Binaria, 13

Relación Complementaria, 14

Relación Compuesta, 15

Relación Inversa, 14

Relación Transitiva, 14

Representaciones Hardware, 50

Resta, 9

Revista PUG, 69

S

Símbolo, 89

Standard 7185, 54

Statement Part, 60

T

Tipo Array, 56

Tipo Conjunto, 56

Tipo Entero, 75

Tipo Enumerado, 55

Tipo Fichero, 56

Tipo Puntero, 56

Tipo Real, 75

Tipo Registro, 56

Tipo Subrango, 56

Top-Down, 53

Top-Down-Parser, 38

Turbo Pascal, 74

U

UCSD Pascal, 69

Unión, 6, 91, 95

Unidad de Control, 96

Universidad de Zurich, 70

University of California, San Diego Pascal, 71

Upper-Case, 67

V

Variables Booleanas Binarias, 20

Y

Yacc, 38

Z

Z80, 74