Índice general

Índice de figuras

Índice de cuadros

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

Parte I

Introducción

Capítulo 1

Formalismos

Resumen:

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.*

i. 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:

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

ii. 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:

$$P = \{x \nearrow (x\,\%2 = 0) \wedge (x >= 0\ \&\ x < 10)\} \tag{1.2}$$

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:

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

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:

$$O \subseteq P \tag{1.4}$$

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:

$$U = \{1,2,\ldots,n,\ldots\} \tag{1.5}$$

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

$$P = \{1,2,3,5,\ldots,n,\ldots\} \subseteq U \tag{1.6}$$

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

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

$$\varnothing \equiv \{\ \} \equiv \{\varnothing\} \tag{1.7}$$

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:

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

Propiedades:

i. Propiedad Conmutativa:

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

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

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

iii. Propiedad de Absorción:

$$O \cup U = U \tag{1.11}$$

iv. Propiedad de Idempotencia:

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

v. Propiedad de Neutralidad:

$$O \cup \varnothing \equiv O = \{v, o, c, a, l, e, s\} \tag{1.13}$$

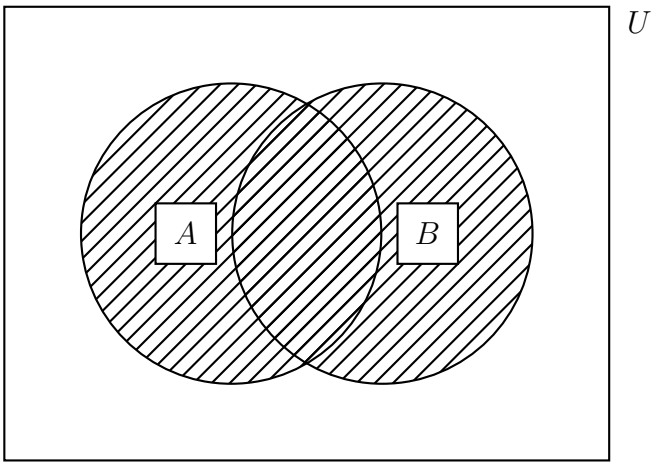


Figura 1.1: Relación de Unión.

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:

$$O \cap P = \{o, a, e\} \tag{1.14}$$

Propiedades:

i Propiedad Conmutativa:

$$O \cap P \equiv P \cap O = \{o, a, e\} \tag{1.15}$$

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

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

iii Propiedad de Absorción:

$$\varnothing \cap O \equiv \varnothing = \varnothing \tag{1.17}$$

iv Propiedad de Idempotencia:

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

v Propiedad de Neutralidad:

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

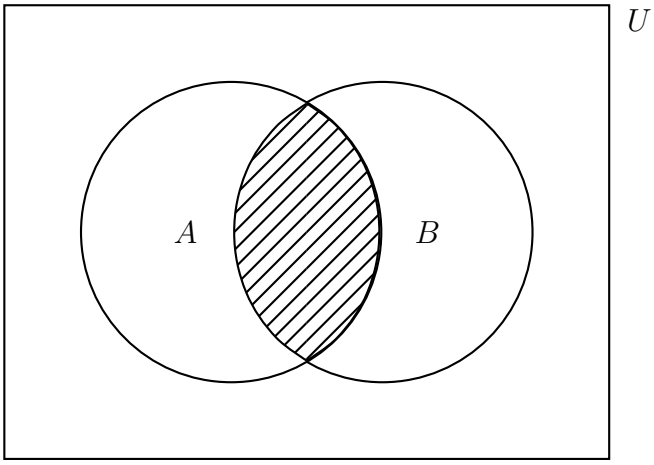


Figura 1.2: Relación de Intersección.

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

Definición 1.1.18. Primera Ley:

$$\overline{O \cup P} = \bar{O} \cap \bar{P} \tag{1.20}$$

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

Definición 1.1.19. Segunda Ley:

$$\overline{O \cap P} = \bar{O} \cup \bar{P} \tag{1.22}$$

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

$$O \cap P = \{v, c, l, s\} \tag{1.23}$$

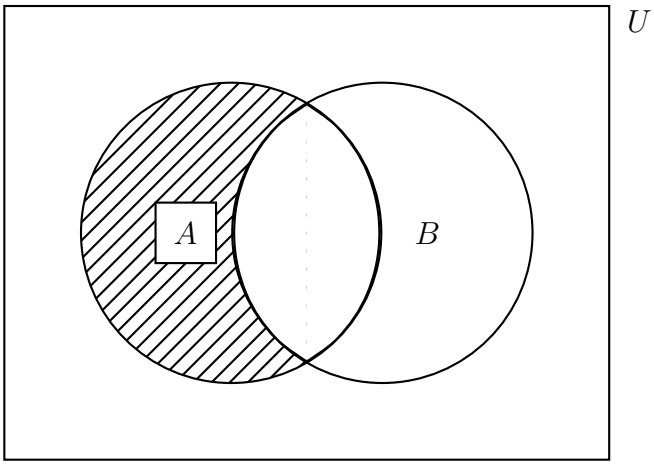


Figura 1.3: Relación de Resta.

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

$$P \cap Q = \varnothing \tag{1.24}$$

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:

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

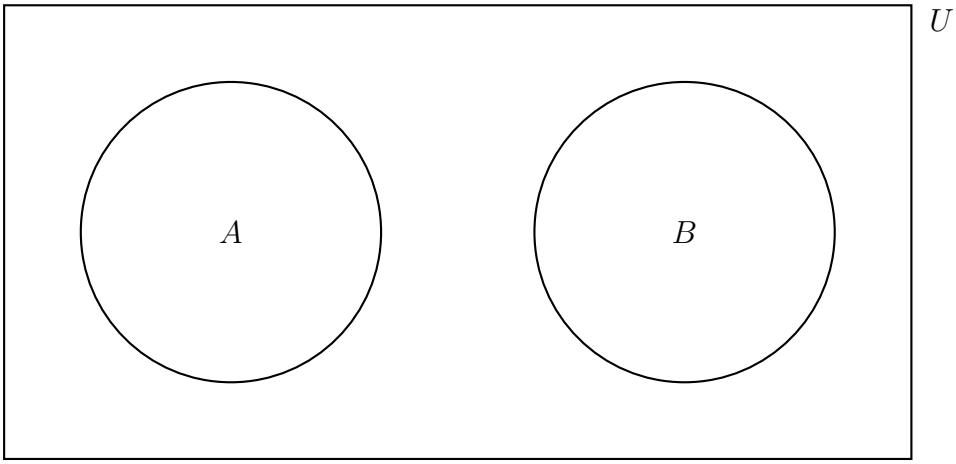


Figura 1.4: Relación de Disjunción.

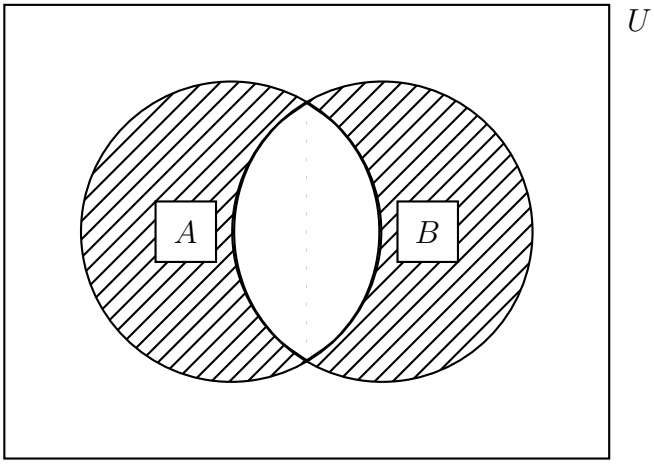


Figura 1.5: Relación de Diferencia Simétrica.

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:

$$\bar{P} = \{b, c, d, \ldots, x, y, z\} \tag{1.26}$$

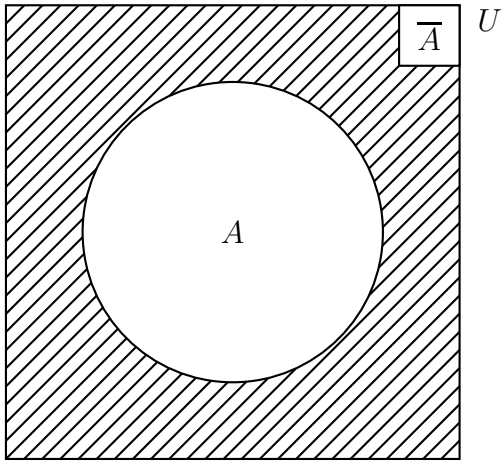


Figura 1.6: Relación de Complemento.

1.2. Relaciones

1.2.1. Definiciones

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

$$P = (a, b) \tag{1.27}$$

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:

$$a \div c \wedge b \div d \tag{1.28}$$

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

$$a \div b \tag{1.29}$$

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

$$(b, a) \tag{1.30}$$

1.2.2. Producto Cartesiano

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

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

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

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

Propiedades:

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

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

Por contra:

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

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

$$R \times S \times T = (R \times S) \times T = R \times (S \times T) \tag{1.35}$$

iii. **Distributivo**:

$$R \times (S \cap T) = (R \times S) \cap (R \times T) \tag{1.36}$$

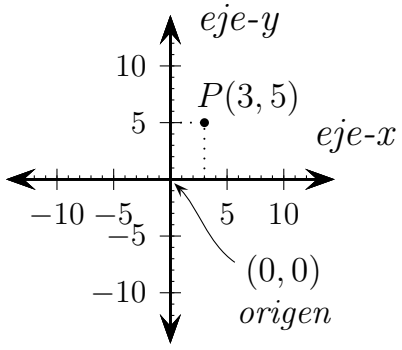
1.2.3. Representaciones

I. **Representación Mediante Tabla**: Para los conjuntos $O = (o_1, o_2, \ldots, o_m)$ y $P = (p_1, p_2, \ldots, p_n)$, 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**.

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

(a) Representación mediante tabla.

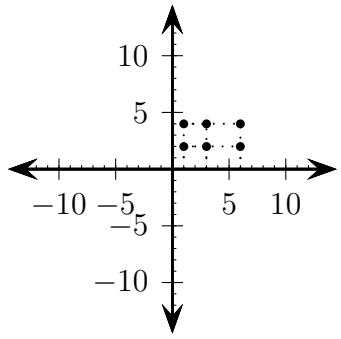


(b) Representación mediante Sistema Cartesiano.

Figura 1.7: Representaciones genéricas del Producto Cartesiano.

∅	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 $\Re$ decimos que: *"La Relación Binaria de A hacia B es de la forma"*:

$$\Re = \{(a, b) \diagup (((a, b) \in A \times B) \wedge (a \ \Re \ b)) \tag{1.37}$$

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

$$o_i \ \Re \ o_j = \{(1, 2), (2, 1), (2, 2), (3, 2)\} \tag{1.38}$$

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 las 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 $\Re^{-1}$) a aquella relación entre pares que establece:

$$\Re^{-1} = \{(a, b)|(b, a) \in \Re\} \tag{1.39}$$

Ejemplo 1.2.10. Para el Ejemplo (1.2.8):

$$o_i \ \Re^{-1} o_j = \{(2, 1), (1, 2), (2, 2), (2, 3)\} \tag{1.40}$$

III. **Relación Complementaria:**

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

$$\forall \ a \in A, b \in B; \ a \ \overline{\Re} \ b \Leftrightarrow a \ \Re \ b \notin (A \times B) \tag{1.41}$$

Ejemplo 1.2.12. Para el Ejemplo (1.2.8):

$$o_i \ \overline{\Re} \ o_j = \{(1, 1), (1, 3), (2, 3), (3, 1), (3, 3)\} \tag{1.42}$$

IV. **Relaciones Transitivas:**

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

$$\forall \ a \in A, b \in B, c \in C; a \ \Re \ b \wedge b \ \Re \ c \Rightarrow a \ \Re \ b \tag{1.43}$$

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

$$o_i \ \Re \ p_j = \tag{1.44}$$

V. **Relación Compuesta:**

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

$$\Re \circ \Im = \{(a,c) / \exists \ b \in B \Leftrightarrow \ a \ \Re \ b \wedge \ b \ \Im \ c\} \tag{1.45}$$

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

$$\Re \circ \Im = \{(1,2),(2,1)\} \tag{1.46}$$

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:

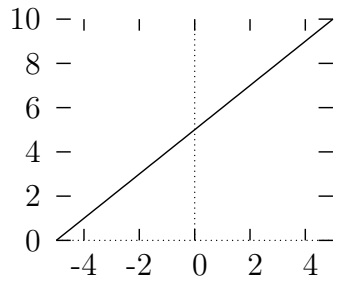
$$f : D_1 \longrightarrow R_1 \tag{1.47}$$

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

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

Dominio	*Recorrido*
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 $Dominio(x)$ le corresponden uno o varios valores en y que a su vez pertenece al conjunto $Recorrido(x)$

$$y = f(x) \tag{1.49}$$

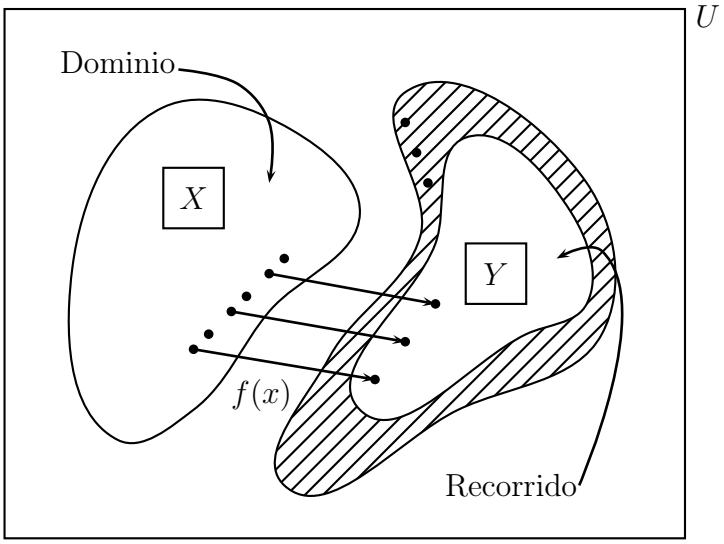


Figura 1.11: Relaciones entre los principales elementos de una Función.

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:

$$X' \subset X : f(X') = \{f(x') \nearrow x' \in X'\} \tag{1.50}$$

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

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

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

$$f : U \longrightarrow Y \nearrow \{u_i \in U, u_i \in X, y \in Y\} \tag{1.52}$$

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

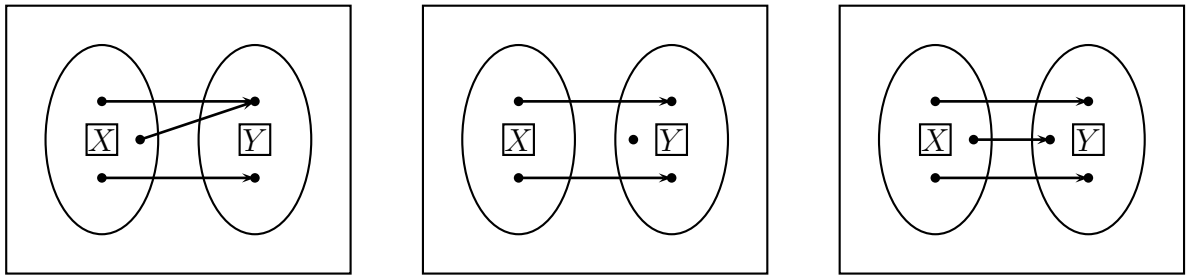
$$\forall\ y\ \in\ Y\ \exists\ x\ \in\ X \nearrow f(x) = y \tag{1.53}$$

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

$$\forall\ x\ \in\ X\ \exists\ y\ \in\ Y\ \diagup\ f(x) = y \tag{1.54}$$

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

$$\exists\ x\ \in\ X,\ \exists\ y\ \in\ Y\ \diagup\ f(x) = y \tag{1.55}$$



(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: $F(x) = (f_1, f_2x, \ldots, f_nx^{n-1})$ y $G(x) = (g_1, g_2x, \ldots, g_nx^{n-1})$ 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**:

$$F(x) + G(x) = (f_1 + g_1, f_2x + g_2x. \ldots, f_nx^{n-1} + g_nx^{n-1})\ \diagup\ \{n \in \mathbb{N}\} \tag{1.56}$$

Ejemplo 1.3.4.

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

II. **Resta**:

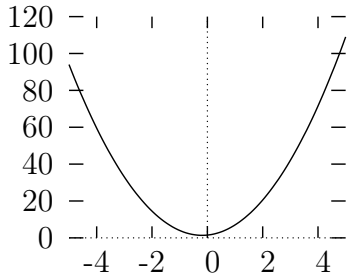
$$F(x) - G(x) = (f_1 - g_1, f_2x - g_2x. \ldots, f_nx^{n-1} - g_nx^{n-1})\ \diagup\ \{n \in \mathbb{N}\} \tag{1.58}$$

Ejemplo 1.3.5.

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

Dominio	*Recorrido*
0	$\frac{3}{2}$
1	7
$\vdots$	$\vdots$
n	$\frac{8n^2+3n+3}{2}$

(a) Representación mediante tabla.

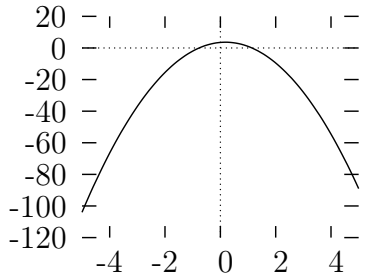


(b) Representación gráfica.

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

Dominio	*Recorrido*
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**:

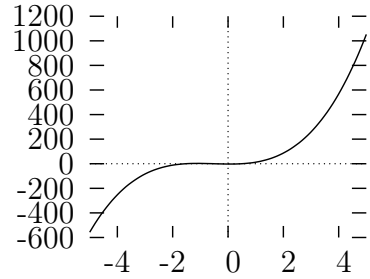
$$F(x) \cdot G(x) = \sum_{i=1}^{i=n} \prod_{j=1}^{j=n} f_i \cdot g_j \ / \ \{i, j \leqslant n\} \ e \ \{i, j \in \mathbb{N}\} \tag{1.60}$$

Ejemplo 1.3.6.

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

Dominio	*Recorrido*
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**:

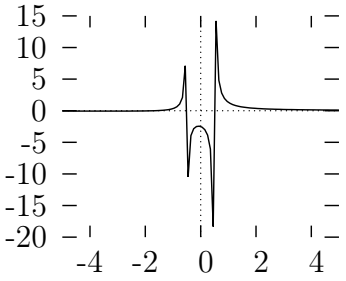
$$\frac{F(x)}{G(x)} = \frac{(f_1, f_2x, \ldots, f_nx^{n-1})}{(g_1, g_2x, \ldots, g_nx^{n-1})} = \frac{f_1}{g_1} + \frac{f_2x}{g_2x} + \ldots, \frac{f_nx^{n-1}}{g_nx^{n-1}} \diagup \{n \in \mathbb{N}\} \tag{1.62}$$

Ejemplo 1.3.7.

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

Dominio	*Recorrido*
0	$\varnothing$
1	$\varnothing$
$\vdots$	$\vdots$
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**:

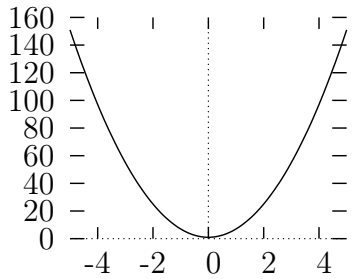
$$F(x) \circ G(x) = F(G(x)) = (f_1, f_2(g(x)), \ldots f_n(g(x))) \diagup \{n \in \mathbb{N}\} \tag{1.64}$$

Ejemplo 1.3.8.

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

Dominio	*Recorrido*
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):

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

Dónde:

i. $\mathfrak{B}$: Se trata del **Conjunto de Variables Booleanas**.

ii. $\sim$: Se trata de una **operación interna unitaria** ($\mathfrak{B} \to \mathfrak{B}$) que cumple:

$$a \to b = \sim a \ / \ \{a, b \in \mathfrak{B}\} \tag{1.67}$$

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

$$(a, b) \to c = a \oplus b \ / \ \{a, b, c \in \mathfrak{B}\} \tag{1.68}$$

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

$$(a, b) \to c = a \odot b \ / \ \{a, b, c \in \mathfrak{B}\} \tag{1.69}$$

Siendo las condiciones necesarias:

i. $a \oplus b = b$

ii. $a \odot b = a$

iii. $\sim a \oplus b = U$

iv. $a \odot \sim b = \varnothing$

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

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

1.4.2. Lógica Binaria

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

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

por lo que cumplen:

I. **Operación Complemento** también denominada Operación NOT (ver Figura 1.18):

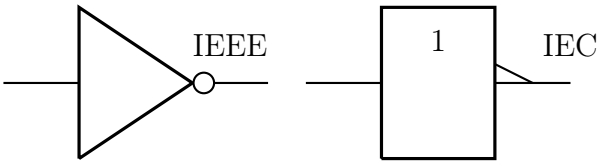
II. **Operación de Suma** también denominada Operación OR (ver Figura 1.19):

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):

x	$\bar{x}$	$x + \bar{x}$	$x \cdot \bar{x}$
0	1	1	0
1	0	1	0

(a) Tabla de la Verdad.

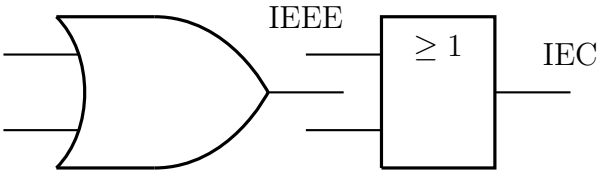


(b) Puertas Lógicas para IEEE e IEC.

Figura 1.18: Representaciones comunes del Operador Booleano NOT.

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

(a) Tabla de la Verdad.



(b) Puertas Lógicas para IEEE e IEC.

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**:

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

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

$$O = (u_1, u_2, \ldots, u_n) \Rightarrow u_i \in (\{0, 1\}, \bar{}, +, \cdot) \tag{1.73}$$

Para el Álgebra de Boole tenemos dos operaciones fundamentales:

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

$$O + P = (u_1, u_2, \ldots, u_n) + (v_1, v_2, \ldots, v_n) = (u_1 + v_1, u_2 + v_2, \ldots, u_n + v_n) \tag{1.74}$$

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

$$U \cdot V = (u_1, u_2, \ldots, u_n) \cdot (v_1, v_2, \ldots, v_n) = (u_1 \cdot v_1, u_2 \cdot v_2, \ldots, u_n \cdot v_n) \tag{1.75}$$

1.5. Nociones sobre Grafos

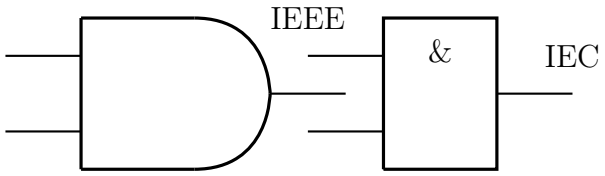
1.5.1. Definiciones

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.

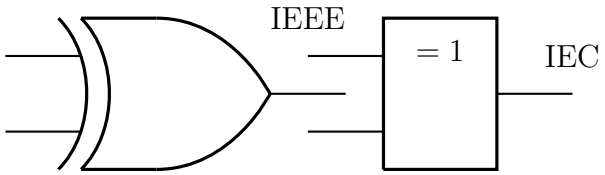


(b) Puertas Lógicas para IEEE e IEC.

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**.

$$V \; x \; V \to E \tag{1.76}$$

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

$$p : E \tag{1.77}$$

Por ello establecemos la siguiente notación para describir un Grafo:

$$G = (V, E, p) \tag{1.78}$$

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:

$$a_\lambda = (v_1, v_2) \; / \; v_1, v_2 \in V, \; a_\lambda \in E \tag{1.79}$$

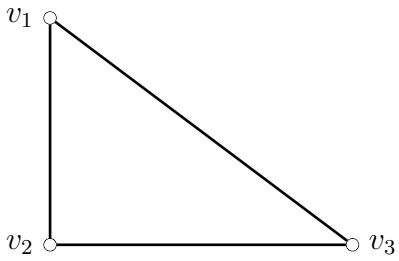
Definición 1.5.5. Un vértice v_1 es **adyacente** sobre otros vértices v_λ si dicho vértice forma parte de la relación:

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

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.

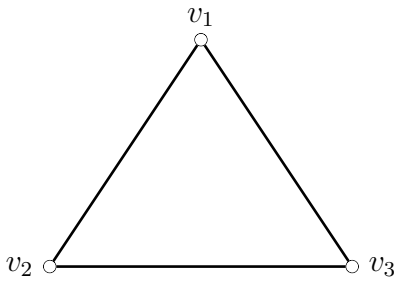
Definición 1.5.7. Si (a_1, a_2) inciden sobre el mismo vértice, se dice que son **aristas paralelas**.

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

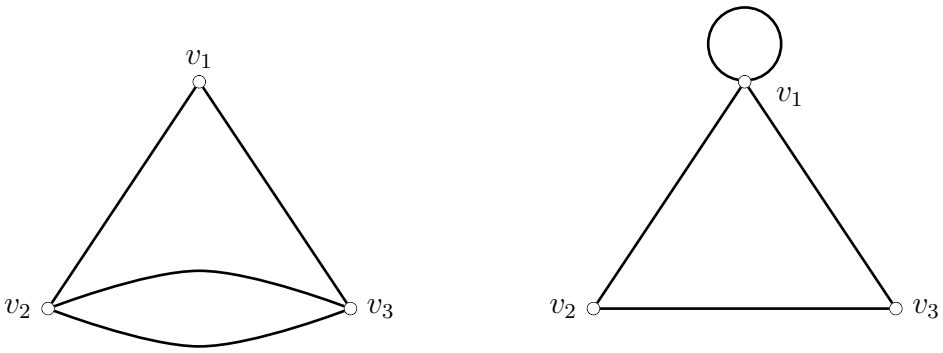
Definición 1.5.9. Un grafo $G = (V, E, p = \emptyset)$ que contiene más de un par de aristas para uno de sus vértices en un **Grafo Multigrafo**.

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**.

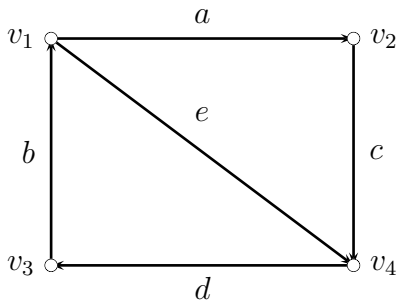
Corolario 1.5.12. *Si $p = \emptyset$ y existen bucles, el grafo se denomina* ***Grafo no Simple*** *.*

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



(a) Multigrafo G_4.

(b) Grafo no simple G_5.



(c) Grafo dirigido G_6.

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 **Biyección** entre ellos α.

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

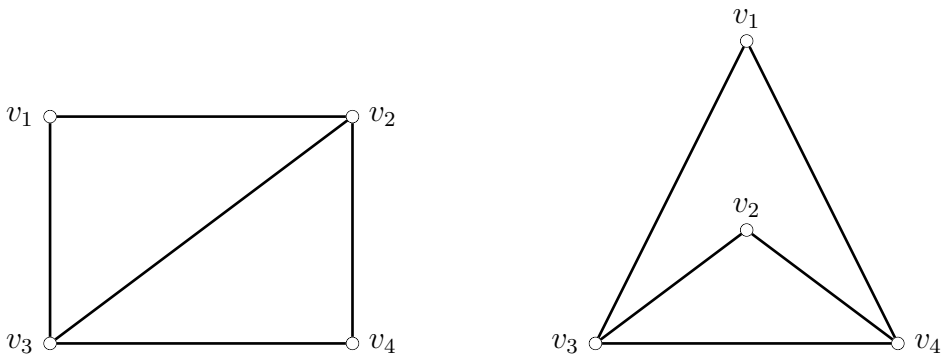
Ejemplo 1.5.15. Para $G_9 \Rightarrow V(G_9) = \alpha\ \{v_1, v_2, v_3, v_4\} \in V(G_8) = \{\alpha(v_1) = v_1',\ \alpha(v_2) = v_2',\ \alpha(v_3) = v_3',\ \alpha(v_4) = v_4'\}$

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

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

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.*



(a) Grafo G_8. (b) Grafo $G_9 \equiv \alpha G_8$.

Figura 1.24: Ejemplo de Grafos Isomorfos.

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

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:*

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

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. $\delta^+ \Rightarrow \delta(v_1)^- = 1;\ \delta(v_2)^- = 1;\ \delta(v_3)^- = 1;\ \delta(v_4)^- = 2$

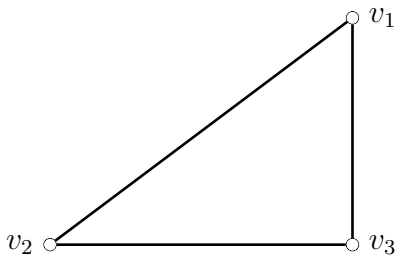
iii. $E = 5$

1.5.3. Tipos

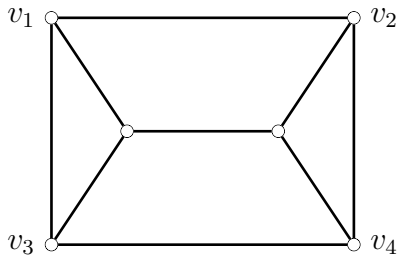
Definición 1.5.22. Se denomina **Grafo Completo** a aquel grafo simple de n vértices que tiene una sola arista entre cada par de vértices. Se denotan como K_n.

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

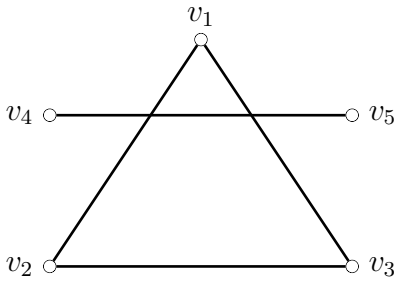
Definición 1.5.24. Se dice que un grafo es **Bipartito** si su número de vértices se pueden dividir en dos conjuntos $G = G_1 \cup G_2$ disjuntos $G_1 \cap G_2 = \emptyset$.



(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 Eureliano:

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 Eureliano**) 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 Eurler 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 Euler[3], es un grafo Eureliano.*

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 sí:*

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 el 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 = φ;
3    FOR i ← 2 TO n DO
4    maxProx[i] ← 1
5    absoluteMin[i] ← L[i,1]
6    WHILE n−1 DO
7       min ← ∞
8       FOR j ← 2 TO n DO
9          IF 0 ≤ absoluteMin[j] THEN min ← absoluteMin[j]
10            k ← j
11      T ← T ∪ maxProx[k]
12      absoluteMin[k] ← -1
13      FOR j ← 2 TO n DO
14         IF L[j,k] < absoluteMin[j] THEN absoluteMin[k] ← L[j,k]
15            maxProx[j] ← 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 $ii \ldots iv$.

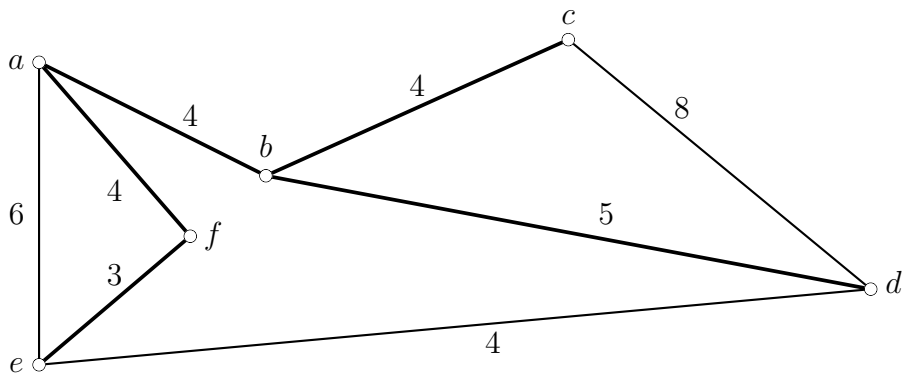


Figura 1.26: Grafo origen para Algoritmo de Prim.

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 arístas 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 ← numNodes;
3    T = φ;
4    DO
5       e ← {u,v}
6       nodeU ← find(u)
7       nodeV ← find(v)
8       IF (nodeU ≠ nodeV) THEN
9          union(nodeU,nodeV)
10         T ← T ∪ {e}
11   WHILE T = n − 1
12   RETURN T
```

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

i. Clasificar las arístas 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 $G(E) - 1$.

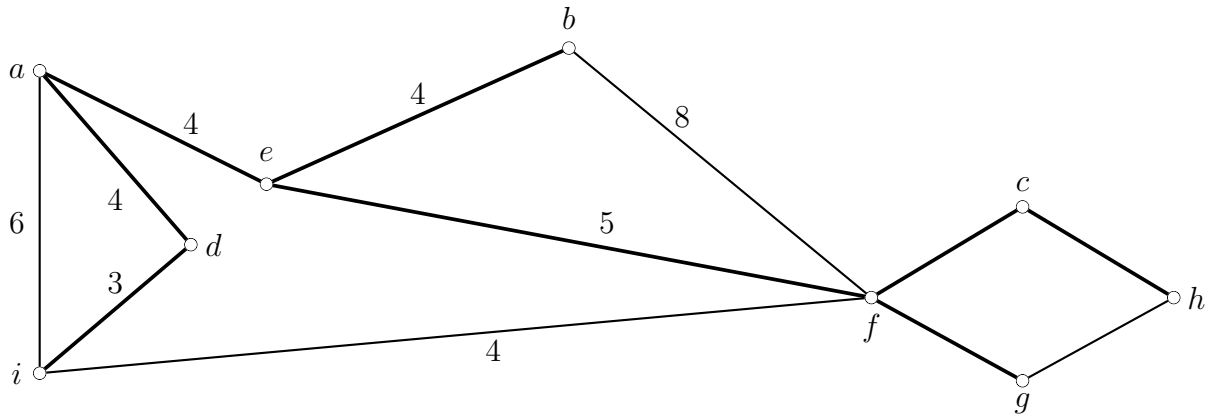


Figura 1.27: Grafo origen para Algoritmo de Kruskal.

Dónde:

i. $V = \{a, b, c, d, e, f, ...\}$

ii. $E = \{ab, bc, cd, de, db, ef, af, ...\}$

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

Cuadro 1.1: Tabla de Pesos Crecientes para Figura 1.27

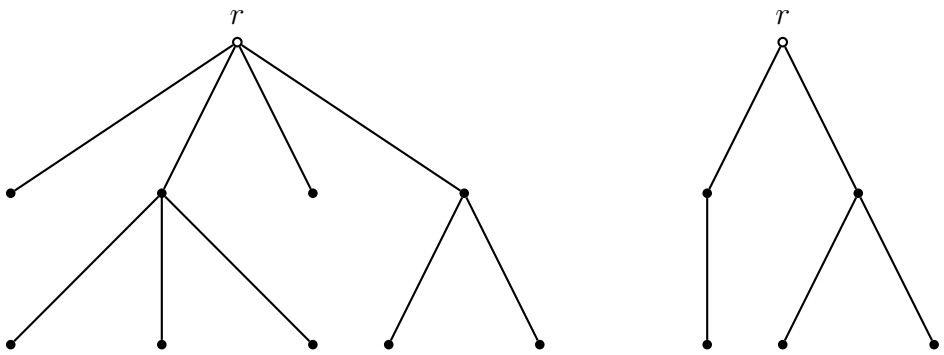
i. Clasificamos las arístas de en orden creciente (Tabla 1.1)

ii. Añadir a $T(E)$; $T(E) = \{a\}$; $T(E) = \{a, e\}$ ...

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 ńumero 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, ..., R_n subárboles de R de Izquierda a Derecha.

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

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

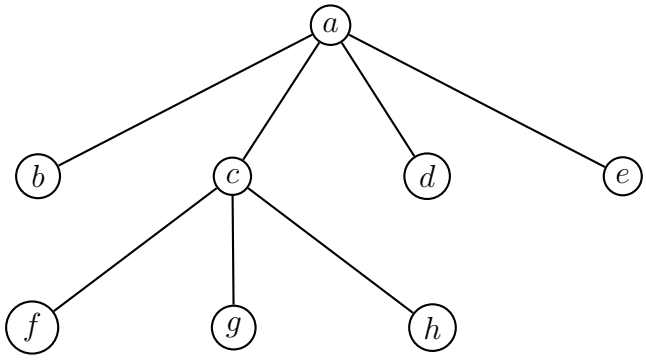


Figura 1.29: Ejemplo de Árbol con Raíz.

II. **Postordén** (Izquierda, Derecha, Raíz): Recorre los vértices: R_1, R_2, ..., R_n en Postordén 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. **Inordén** (Izquierda, Raíz, Derecha): Si r contiene: R_1, R_2, ..., R_n entonces recorre los nodos de izquierda a derecha R_i para volver a r y recorrer en Inordén 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 del capítulo

[1]La 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.

[2]**El 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.

[3]El origen de la **Teoría de Grafos parte de la famosa publicación "Los siete puentes de Königsberg"** dónde 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 cuidad de Königsberg (actual Kalingrad) 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:

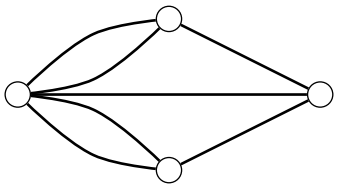


Figura 1.30: Grafo de Königsberg.

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.*

[4]La 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*

Bibliografía capitular

Capítulo 2

Resumen: Proyecto gp1990c

Resumen:

2.1. Objetivos

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

i. **Desarrollar el estándar ISO Pascal 7185:1990**[5] **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 (gp1990la): 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/Bison[7] sería:

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

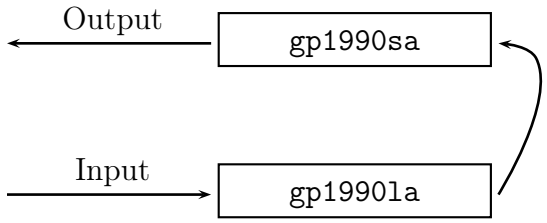


Figura 2.1: Síntesis y Partes de `gp1990c`.

2.3. Transformación de una Expresión Regular en Software

Dicho proceso comprende una serie de etapas:

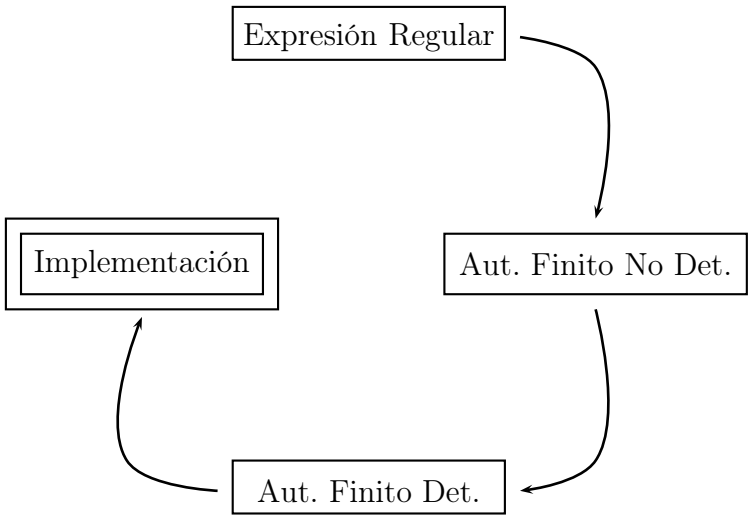


Figura 2.2: Transformación desde una Expresión Regular a su Implementación.

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

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

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

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

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

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* **??***).*

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 Contexto[8]

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

Cuadro 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 **??**.

2.5. Breve descripción de `gp1990sa`

Definición 2.5.1. `gp1990sa` en 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 una desarrollo de arriba hacía 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 hacía 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.l`: 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 (Autoconf[10]): 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 Compiler[11](`gpc`) y Free Pascal[12](`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.

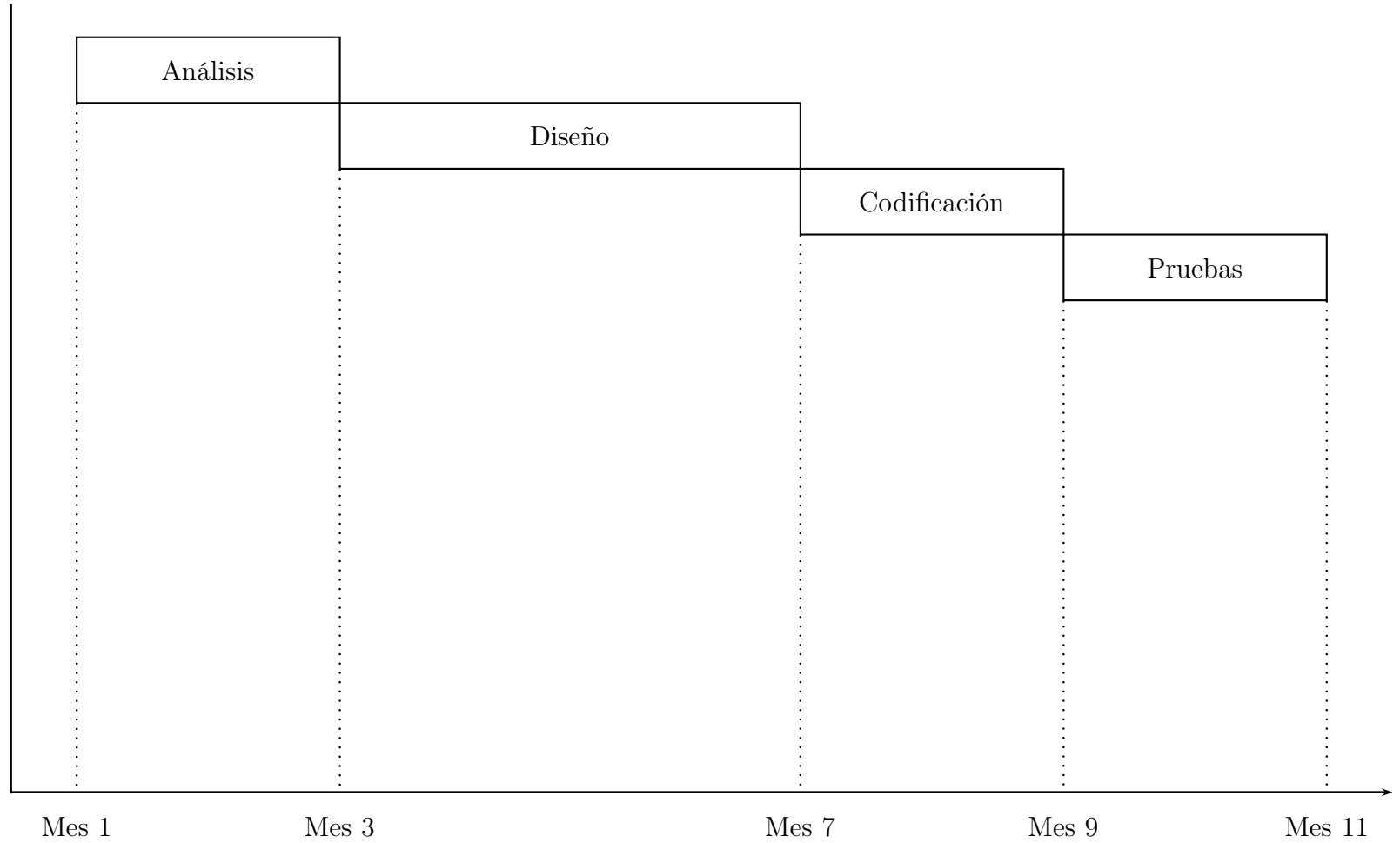


Figura 2.3: Diagrama de Gantt para desarrollo de `gp1990c`.

2.7. Entorno de desarrollo

I. GNU/Linux: Sistema Operativo base (Gentoo GNU/Linux[13] 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 LaTeX 2_ε). También es resaltable el hecho de que es compatible con otras familias UNIX como BSD.

II. BSD[14]: 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. GCC[15]: 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. Flex[16]: 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. Bison[17]: 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 2011[18]: 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 del capítulo

[5]http://www.moorecad.com/standardpascal/
[6]En inglés se denomina "parser"
[7]Compatible con Lex/Yacc.
[8]Gramática Chomskiana de Tipo 2: P = $\{(S \to \lambda) \vee (A \to p_2) \mid p_2 \in \Sigma^+; A \in N\}$
[9]Debido 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/

Bibliografía capitular

Capítulo 3

El Lenguaje de Programación Pascal

Resumen:

3.1. Introducción

A programming language called Pascal is described which was developed on the basis of Algol 60. Compared to Algol 60, its rango 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 contructed 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 Wirth[20] a finales de la década de los sesenta del siglo XX. En 1970 fue finalmente publicado, fijando dos objetivos en su diseño arquitectónico:

i. 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.

ii. 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 **??**).

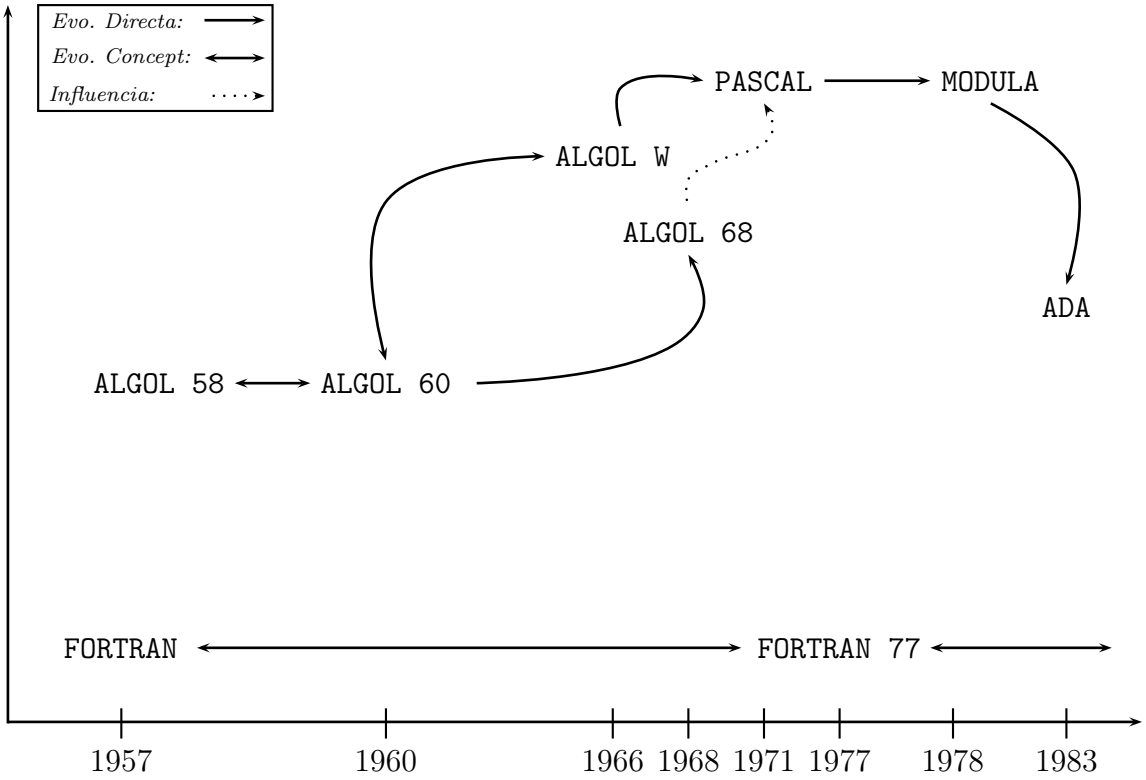


Figura 3.1: Relaciones entre los primeros Lenguajes de Programación.

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.

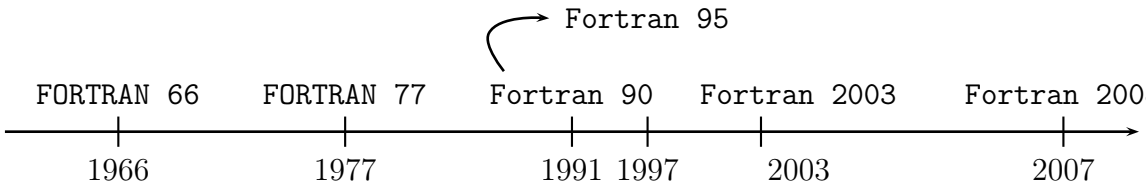


Figura 3.2: Evolución del Lenguaje Fortran.

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 son 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 letras[22]: `‘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`

Notas sobre compilación: Para compilar el archivo fuente `helloProgrammer.f95` sobre GNU, usaremos el compilador GNU Fortran[23].

Las ordenes para compilarlo y ejecutarlo son las siguientes:

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

3.2.2. ALGOL (ALGOrithmic 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 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.

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.

Cuadro 3.1: Convención entre *Reference Language* y otras publiaciones de ALGOL.

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.

ALGOL 60 es un 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 actualizas ALGOL 60 intentando conservar la esencia y cultura del lenguaje.

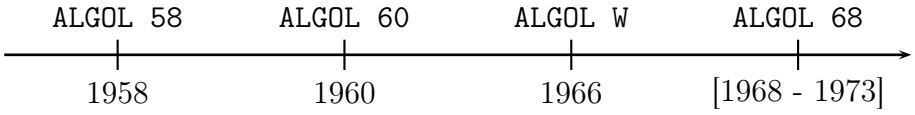


Figura 3.4: Evolución del Lenguaje ALGOL.

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`

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 tenia 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 pasada[26] basada en el diseño "Top-Down".

El compilador se completó finalmente a mediados de 1970.

Pascal después llego 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 compilares 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 BSI[27] [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 `SPARCH`[28] para la resolución de US TAG[29] 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 = ‘+’ | ‘-’ | ‘*’ | ‘/’ |‘=’ | ‘<’ | ‘>’ | ‘[’ | ‘]’| ‘.’ | ‘,’ | ‘:’ | ‘;’ | ’ˆ | ‘(’ | ‘)’| ‘<>’ |‘<=’ | ‘>=’ | ‘:=’ | ‘..’ | 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:

Ejemplo 3.3.7. Registro:

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:

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 sobreescrito.

1) Precondición: `True`
2) Postcondición: `(f↑)` está indefinido.

ii. `PROCEDURE PUT(`f`)` → Añade el valor del buffer del `(f↑)` al propio fichero.

1) Precondición: `(f↑)` está definido.
2) Postcondición: `(f↑)` 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↑)` está definido.
2) Postcondición: `(f↑)` está indefinido.

iv. `PROCEDURE GET(`f`)` → Avanza de descriptor de fichero y asigna el valor del buffer de al `(f↑)`.

1) Precondición: `(f↑)` está definido.
2) Postcondición: `(f↑)` 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,`v_1`); begin read(ff,`v_2`, ..., `v_n`) 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,`v_1`); begin write(ff,`v_2`, ..., `v_n`) 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;` $\equiv sen(x)$

d. `FUNCTION COS(`*x*`)` → Para el tipo de datos REAL devuelve el coseno de `x`.
Formalidad 3.3.14. `FUNCTION COS(`*x:REAL*`): REAL;` $\equiv cos(x)$

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 $Ln(x)$, siendo `x` el parámetro.
Formalidad 3.3.16. `FUNCTION LN(`*x:REAL*`): REAL;` $\equiv Ln(x)$

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;` $\equiv arctg(x)$

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;` $\equiv TRUNC(a,b) = a$

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;` $\equiv (abs(x)\ mod\ 2 = 1)$

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`

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 lo bloques PROCEDURE Y FUNCTION y se divide en los siguientes apartados:

i. `laber 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.

Programa 3.3.22. `helloProgrammer.pas`

Notas sobre compilación: Para compilar el archivo fuente `helloProgrammer.pas` sobre GNU, usaremos el compilador `GNU Pascal Compiler`[30].

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, modem 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 cápsula[32] 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. Gramatica: (ver Anexo **??** sección 2.)

Programa 3.4.1. `helloProgrammer.mod`

Notas sobre compilación: Para compilar el archivo fuente `helloProgrammer.adb` sobre GNU, usaremos el compilador GNU Modula-2[33].

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 público ganadora la empresa *CII Honeywell Bull*. Al mismo tiempo se dio nombre al lenguaje que empezaba a ser una realidad. Se denomino 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 perfecciono y por fin fue publicado por ISO (`ISO-8652:1987`).

El mismo Ada 83 desde el principio tuvo deficiencias prácticas por lo que se trabajo 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.

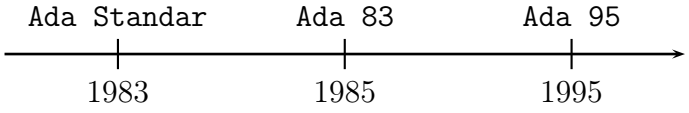


Figura 3.5: Evolución del Lenguaje Ada.

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

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 encapsuláción, 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`

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 **??** sección 3.)

Programa 3.4.3. `helloProgrammer.mod`

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 del capítulo

[19] *El 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.*

[20] El 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.

[21] *El 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.*

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

[23] http://gcc.gnu.org/fortran/

[24] *El 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/

[26] Single-Pass.

[27] *British Standards Institution*: http://www.bsigroup.com/

[28] *Standards Planning and Requirements Committee*

[29] *Technical Advisory Group*: http://technicaladvisorygroup.com/

[30] http://www.gnu-pascal.de/gpc/

[31] Modula-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 explicita.

[32] El concepto de cápsula como unidad estructural se de 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

[34] Augusta 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.

[35] Divide y Vencerás.

[36]http://www.adacore.com/

[37]Oberon 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

Bibliografía capitular

[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.

[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.

[Sta66] American National Standar. FORTRAN 66 (USA Standar FORTRAN). *American National Standar*, March 7, 1966.

[Wir71] Nickaus Wirth. The Programming Language Pascal. *Acta Informatica Volume 1, Number 1, 35-63, DOI: 10.1007/BF00264291*, 1971.

[Wir80] Nickaus Wirth. Modula-2. *Institut für Informatik ETH Zürich Technical Report 36*, 1980.

[Wir88] Nickaus Wirth. *The programming language Oberon*. Software: Practice and Experience Volume 18, Issue 7, pages 671–690, July 1988.

Capítulo 4

Compiladores del Lenguaje Pascal

Resumen:

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 ultima 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 creo 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 máquina 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 trato 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.

Cuadro 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 integra de "Full Pascal" o también llamado Pascal 1971. Sobre P2 se derivaron importantes compiladores como: UCDS Pascal a la vez que sirvió de prototipo para Borland Turbo Pascal.

II. Pascal P3 y P4: Está 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.

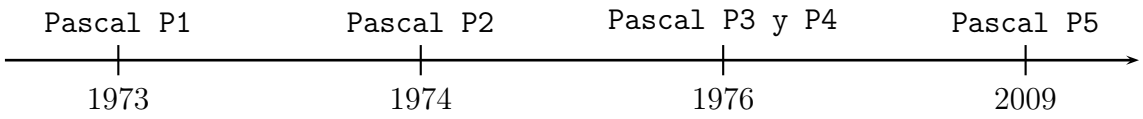


Figura 4.1: Evolución de Portable Pascal.

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 percato 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 ideo 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. Está 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 Pecan? Systems que a su vez estaba formada por entusiastas de p-System.

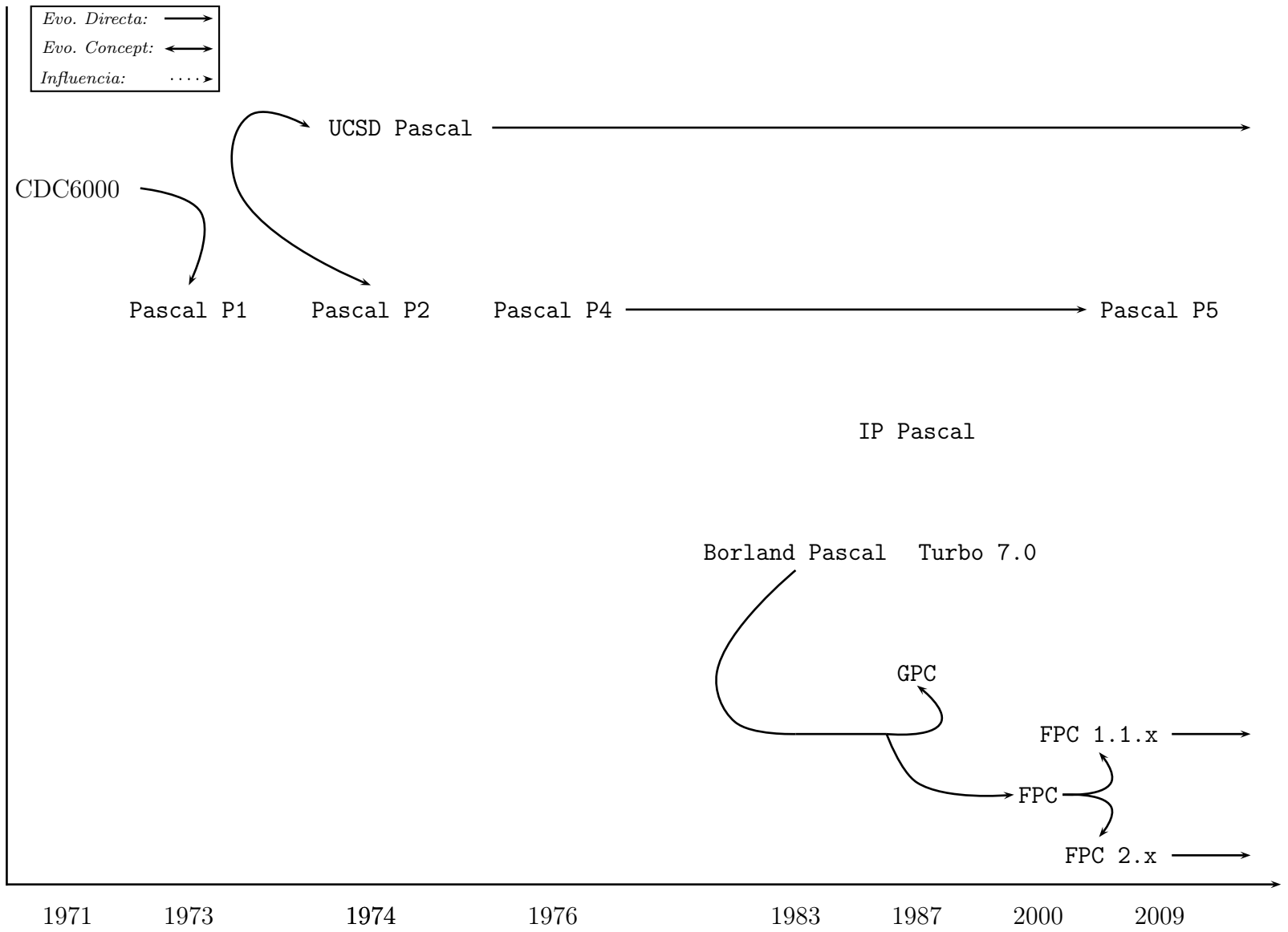


Figura 4.2: Evolución de compiladores para Pascal.

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 trascrito 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 explicita 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 Pascal[40]. 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: ficherodeTexto;] {lista de variables}`*`);` → *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	Lnx
FUNCTION SIN	$sen(x)$
FUNCTION SQR	x^2
FUNCTION SQRT	$\sqrt{x}$
FUNCTION TRUNC	$TRUNC(a,b) = a$

Cuadro 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;` → *idem*

ii. `FUNCTION ARCTAN(`*`x:REAL`*`): REAL;` → *idem*

iii. `FUNCTION CHR(`*`x:BYTE`*`): CHAR;` → *idem*

iv. `FUNCTION COS(`*`x:REAL`*`): REAL;` → *idem*

v. `FUNCTION EOF(`*`var F: tipodeFichero`*`): BOOLEAN;` → *idem*

vi. `FUNCTION EOLN(`*`var F: tipodeFichero`*`): BOOLEAN;` → *idem*

vii. `FUNCTION EXP(`*`x:REAL`*`): REAL;` → *idem*

viii. `FUNCTION LN(`*`x:REAL`*`): REAL;` → *idem*

ix. `FUNCTION ODD(`*`x:LONGINT`*`): BOOLEAN;` → *idem*

x. `FUNCTION ORD(`*`x:tipoOrdinal`*`): LONGINT;` → *idem*

xi. `FUNCTION PRED(`*`x:tipoOrdinal`*`): tipoOrdinal;` → *idem*

xii. `FUNCTION ROUND(`*`x:REAL`*`): LONGINT;` → *idem*

xiii. `FUNCTION SIN(`*`x:REAL`*`): REAL;` → *idem*

xiv. `FUNCTION SQR(`*`x:tipo`*`): tipo;` → *idem*

xv. `FUNCTION SQRT(`*`x:REAL`*`): REAL;` → *idem*

xvi. `FUNCTION SUCC(`*`x:tipoOrdinal`*`): tipoOrdinal;` → *idem*

xvii. `FUNCTION TRUNC(`*`x:REAL`*`): LONGINT;` → *idem*

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

Cuadro 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 **??**) Σ[42].

ii. Bison: Trata la sintaxis en base a las especificaciones BNF[43].

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.

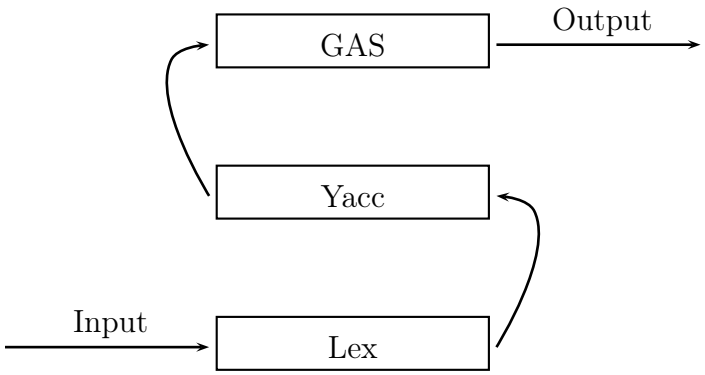


Figura 4.3: Arquitectura de GPC.

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 Prepocesador. 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.

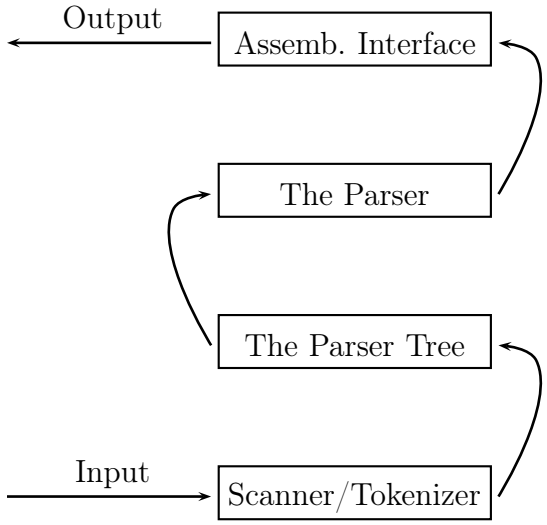


Figura 4.4: Arquitectura de FPC.

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 del capítulo

[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..."

[40]Desarrollado por NasSys.

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

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

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

Bibliografía capitular

Bibliografía general de la obra

[AVAU98] John E. Hopcroft Alfred V. Aho and Jefrey 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.

[Cru10] Francis D'Cruze. Pascal Programming Language. *Wath?*, 2010.

[CW12] Warren Buck Chi Woo, J. Pahikkala. Venn Diagram for PSTricks (Version 13). PlanetMath.org, 2012.

[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.

[Gaj10] Zarko Gajic. Introducing Borland Delphi. *Wath?*, Abril 2010.

[ibi11] ibiblio.org. A Brief History of FORTRAN/Fortran. http://www.ibiblio.org/, 2011.

[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. Kerninghan 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 Plublishing, Inc, 2008.

[Mer05a] Félix García Merayo. *Matemática Discreta*. Thomson, 2005.

[Mer05b] Félix García Merayo. *Matemática Discreta*. Thomson, 2005.

[MNN05] Marshall Kirk Mckusick and George V. Neville-Neil. *The Desing and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2005.

[rhc12] rhcsoftware.com. About RHC Software. http://www.rhcsoftware.com/, 2012.

[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 Standar. FORTRAN 66 (USA Standar FORTRAN). *American National Standar*, 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.

[Wik11a] WikipediaEN. Algol 58. http://en.wikipedia.org/, 2011.

[Wik11b] WikipediaEN. Algol 60. http://en.wikipedia.org/, 2011.

[Wik11c] WikipediaEN. Algol W. http://en.wikipedia.org/, 2011.

[Wik11d] WikipediaEN. Chomsky hierarchy. http://en.wikipedia.org/, 2011.

[Wik11e] WikipediaEN. Context-free grammar. http://en.wikipedia.org/, 2011.

[Wik11f] WikipediaEN. Context-sensitive grammar. http://en.wikipedia.org/, 2011.

[Wik11g] WikipediaEN. Formal language. http://en.wikipedia.org/, 2011.

[Wik11h] WikipediaEN. Regular grammar. http://en.wikipedia.org/, 2011.

[Wik11i] WikipediaEN. Turbo Pascal. http://en.wikipedia.org/, 2011.

[Wik11j] WikipediaEN. Unrestricted grammar. http://en.wikipedia.org/, 2011.

[Wir71] Nickaus Wirth. The Programming Language Pascal. *Acta Informatica Volume 1, Number 1, 35-63, DOI: 10.1007/BF00264291*, 1971.

[Wir80] Nickaus Wirth. Modula-2. *Institut für Informatik ETH Zürich Technical Report 36*, 1980.

[Wir88] Nickaus 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