Evolución de los lenguajes de programación(I)

Félix Saltor Soler

Este artículo (1) pretende dar una visión primordialmente evolutiva de los lenguajes de programación, intercalando cuestiones que han influido, a mi parecer, en esta evolución, y al mismo tiempo construyendo un *esquema clasificatorio* de los mismos.

Respecto a estas agrupaciones en clases (2) señalemos que:

a) emplearemos ciertos nombres de clases que son usuales, incluso si son discutibles;
b) la clasificación en *generaciones* es independiente de las generaciones del hardware, aunque mantengan cierta relación;
c) la mención de algunos lenguajes como ejemplo de una clase no implica un juicio de valor negativo para otros lenguajes no mencionados; y
d) algunos lenguajes muy particulares podrán quedar fuera del esquema clasificatorio.

LENGUAJES DE PROGRAMACION

Trataremos solamente de los *lenguajes de programación* de ordenadores, de los que no daremos una definición rigurosa, sino que los consideraremos en el sentido usual entre los informáticos, que podríamos concretar en «los vehículos formales (o lógicos) en que los profesionales de la programación especifican a los ordenadores lo que esperan de ellos». Quedan pues excluidos los restantes lenguajes de la informática (3).

Además, el énfasis estará en los lenguajes como tales, es decir, independientemente de sus traductores (compiladores, etc.) en cuanto ello sea posible, lo cual no siempre se dará, como veremos.

No describiremos ningún lenguaje específico: remitimos para ello al lector a los textos existentes sobre los distintos lenguajes (que no incluimos en la bibliografía que aparece al final), o a textos generales (4).

«Prehistoria»

Es discutible si a las máquinas que se programan cableando tableros de conexiones les es aplicable el nombre de ordenadores, y si los vehículos formales que se plasman en estos soportes cableados merecen el nombre de lenguajes. De todos modos, como algunos autores (5) los enumeran entre los lenguajes de programación, recordaremos aquí su existencia, si bien en una clase especial que etiquetaremos «*prehistoria*» por analogía, ya que los programas correspondientes no tienen un soporte escrito (6).

PRIMERA GENERACION: lenguajes de máquina

En la *clase 1* agruparemos los distintos *lenguajes de máquina* entendiendo por tal un lenguaje que es aceptado directamente por la unidad de mando o de gobierno (control unit) de la unidad central de un ordenador: uno o varios formatos de instrucción, con un juego de códigos de operación, y los correspondientes formatos (y valores válidos de los operandos).

Está claro que un lenguaje de máquina es muy estricto, muy rígido, en el formato de cada instrucción, y no permite ninguna agrupación explícita de instrucciones ni apenas relaciones entre ellas: por ello lo tomaremos como origen de un eje de *nivel sintáctico* (7). Además como el significado de las instrucciones es el mínimo posible en el ordenador de que se trate, es decir, la operación especificada por instrucción no es descomponible en operaciones parciales que se puedan ordenar a la unidad de mando, tomaremos también el lenguaje de máquina como cero del eje de *nivel semántico* (8). En este eje, que será también una escala de lo *imperativo* a lo *declarativo*, el lenguaje de máquina queda pues situado en el extremo como lenguaje totalmente imperativo.

Así pues, en el *sistema de coordenadas* nivel sintáctico-nivel semántico en que colocaremos clases y lenguajes, el lenguaje de máquina es el origen (figura 1). Respecto a este sistema de coordenadas es preciso indicar:

a) que no es cuantitativo (no se ha definido una *distancia*), sino realtivo; y
b) que a cada lenguaje correspondería en realidad un conjunto de puntos, aunque simplifiquemos representándolo por uno solo (que sería su centro de gravedad si el sistema fuera cuantitativo) (9).

Estas mismas pobrezas semántica, y, sobre todo, sintáctica (que dificultan no sólo la programación, sino también la modificación de programa) de los lenguajes de máquina son las que los hacen inadecuados para la escritura manual de programas, por lo que, ya en la primera generación de ordenadores, se trabajó en el diseño de lenguajes *simbólicos*, no sólo de los de la «segunda generación» de lenguajes que vamos a ver a continuación, sino apuntando a los de la clase que luego llamaremos funcionales algorítmicos (clase 3.ª), por obra, entre otros, de Grace Hopper.

SEGUNDA GENERACION: lenguajes uno-a-uno

Para solucionar, al menos en parte, las dificultades de la programación en lenguaje de máquina aparecen pues lenguajes que emplean *símbolos* nemotécnicos en vez de códigos de operación y *símbolos* variables sustituyendo a las direcciones *absolutas* de memoria, a los que denominaremos pues *lenguajes simbólicos* (en sentido estricto), o *lenguajes uno- a- uno* para indicar que a cada frase del programador corresponde generalmente una instrucción de máquina. Con ello el programador queda liberado de la asignación de direcciones absolutas a instrucciones y áreas de datos (lo que le permitirá en particular intercalar nuevas instrucciones sin reasignar direcciones), tarea que corresponde ahora al programa traductor del lenguaje uno-a-uno al lenguaje de máquina, el cual suele denominarse programa *ensamblador*.

Estos lenguajes forman pues parte de los lenguajes que requieren traducción, a los que llamaremos *simbólicos* en sentido amplio —y por oposición llamaremos *vernáculos* a los de máquina, siguiendo a Freixa (67)— y para los que se establece pues la distinción entre programa (y lenguaje) *fuente*, escrito por el programador, y programa (y lenguaje) de máquina (llamado a veces programa *objeto*, en cuanto resultado de la traducción —object program—, o también programa binario).

Esta clase de lenguajes la subdividiremos en dos, primero desde el punto de vista semántico y luego desde el sintáctico.

Lenguajes uno-a-uno básicos

Esta clase 2.ª está formada por los lenguajes simbólicos en los que la correspondencia una frase de programa fuente — una instrucción de máquina es total, y que reciben el nombre de lenguajes uno-a-uno *básicos.* Su nivel semántico, en cuanto a poder de las instrucciones, es pues el mismo que los lenguajes de máquina correspondientes, y sólo su facultad de efectuar declaraciones de diversos tipos nos permite hacerlos avanzar ligeramente en el eje semántico (figura 1).

Ejemplos: Los lenguajes de esta clase, si bien distintos para cada máquina, o sistema de explotación, han adoptado los nombres genéricos de SPS (Symbolic Programming System), Basic Autocoder o Basic Assembler.

Lenguajes uno-a-uno con macroinstrucciones

Los lenguajes de esta segunda generación en los que es posible definir y usar palabras clave (llamadas macroinstrucciones) tales que la frase que las contiene se traduce no en una sola, sino en toda una secuencia de instrucciones de máquina (*expansión* de la macroinstrucción), se llaman lenguajes uno-a-uno con *macroinstrucciones,* y aparecen posteriormente a los uno-a-uno básicos.

El poder semántico de una macroinstrucción es pues superior al de una instrucción de máquina (en particular si su expansión no es fija, sino variable según el contexto en que aparece), como queda reflejado en la figura 1.

Ejemplos: Estos lenguajes suelen llamarse Autocoder, Macro Assembly Language, Assembler Language o Assembler.

Formatos tabular o libre

Desde el punto de vista sintáctico los lenguajes de esta segunda generación pueden dividirse en dos clases:

a) los de *formato tabular,* en los que cada tipo de símbolo debe aparecer en unas posiciones determinadas del registro (por ejemplo en unas columnas fijas de la tarjeta perforada), de sintaxis pues, muy estricta; y

b) los de *formato libre,* en los que los símbolos no están sujetos a posiciones fijas, sino a la intercalación entre los mismos de *caracteres separadores* (generalmente la coma y el blanco), por lo que su nivel sintáctico es mayor (figura 1).

Señalemos también, como avances sintácticos a la vez que semánticos, los *literales,* el uso de ciertas expresiones como símbolos (del tipo de «PEPE + 3» o «* — 2»), y la aparición de la *subrutina* (10).

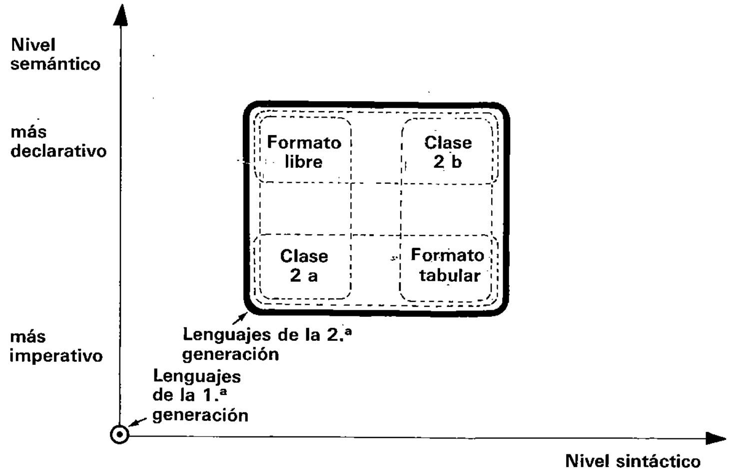

Figura 1

TERCERA GENERACION: lenguajes funcionales

Los lenguajes de la segunda generación son un esfuerzo importante por adaptar los lenguajes de máquina a las características humanas de los programadores, pero siguen estando lejos de ser el vehículo ideal para programar. Además, son propios de una máquina (o grupo de máquinas), lo que exige reprogramar al cambiar a un nuevo ordenador no compatible con el anterior. Es pues lógico que aparezcan lenguajes que no estén *«orientados a la máquina»*, como los de las dos primeras generaciones, sino *«orientados al problema»* (11), a los que también se conoce como *funcionales* y *«de alto nivel»* (high level languages o HLL). Las frases de estos lenguajes recuerdan muy poco, en general, a las instrucciones de máquina (en cierto modo todas ellas son macroinstrucciones). Al mismo tiempo estos lenguajes son potencialmente *de explotación universal*, pues podrá programarse en uno de ellos para todos los ordenadores, mientras se disponga de traductores de dicho lenguaje a los lenguajes vernáculos de los mismos (y en concreto los lenguajes Fortran y Cobol son prácticamente de explotación universal). Veamos además, antes de estudiarlos con más detalle, que en la adopción de esta clase de lenguajes hay también una razón económica.

Coste de programar y coste de ejecutar

Consideremos, en una primera hipótesis de trabajo, que un programa se establece de una vez para siempre, mientras que será ejecutado un número *n* de veces. El coste total relativo a dicho programa será pues:

Coste total = Coste de programación + n × Coste de ejecución,

donde *Coste de programación* incluye el diseño, la escritura, la depuración o puesta a punto, y la documentación, y *Coste de ejecución* engloba el consumo de recursos ligado a cada ejecución del programa (12). El coste de programación utilizando un lenguaje funcional Cp_3 (13), es menor que el correspondiente a un lenguaje de segunda generación Cp_2, ya que la duración de la programación —y por lo tanto el componente principal del coste, las horas de programador— es menor. En cambio, el coste de ejecución con un lenguaje funcional Ce_3 es mayor que su homónimo usando un lenguaje uno-a-uno, Ce_2, pues la ocupación de memoria y de tiempo son algo mayores. Representándolo gráficamente (figura 2), observaremos que las rectas cuya expresión analítica son los costes totales en uno y otro caso, se cortan en un punto, de abcisa N: así pues, según esta hipótesis, será menos costoso usar lenguajes funcionales si el programa ha de ejecutarse un número de veces inferior a N, y lenguajes de segunda generación en caso contrario.

Es interesante la evolución de N, cuya expresión es

$$N = \frac{Cp_2 - Cp_3}{Ce_3 - Ce_2}$$

Como el coste de la hora de programador —y por tanto el numerador— ha ido aumentando con los años, mientras que los costes tanto de ejecutar una instrucción como de una porción fija de memoria han ido disminuyendo —y con ellos el denominador— *el valor de N ha ido creciendo*. Así pues, ciertos programas para los que hubiera sido más rentable hace seis años usar lenguajes uno-a-uno, sería hoy preferible programarlos en un lenguaje funcional. Es más, si listamos algunas clases de programas por orden creciente de número de ejecuciones (teniendo en cuenta las ejecuciones en múltiples ordenadores):

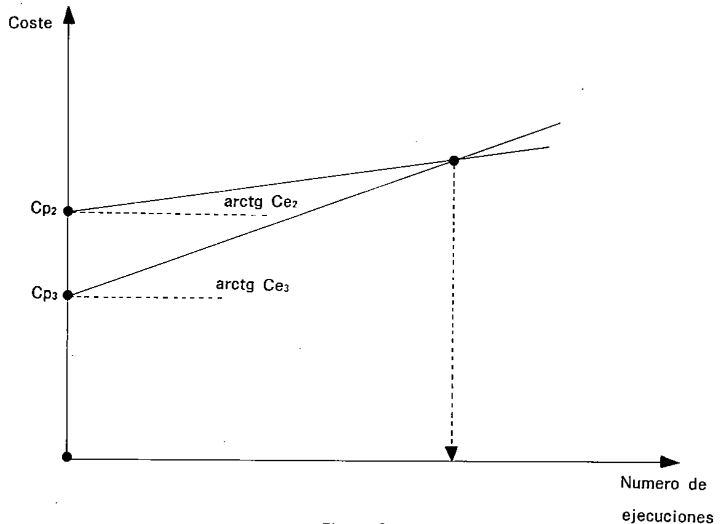

Figura 2

- programas de una sola ejecución (one shot programs);
- programas de aplicación particulares de una instalación;
- programas generalizados de aplicaciones particulares (por ejemplo seguros);
- programas generalizados de aplicaciones generales (por ejemplo nómina);
- traductores, programas de clasificación, de servicio, etc.;
- montadores (linkage editors);
- rutinas de gestión de trabajos de los sistemas de explotación; y
- rutinas de gestión de tareas de los sistemas de explotación,

encontraremos que hoy, al parecer, solamente las cuatro últimas clases debieran, siempre según esta hipótesis, programarse en lenguajes de segunda generación, pero no las dos primeras. Ahora bien, la hipótesis es falsa.

Donde aparecen el coste de mantener y otros factores

En efecto, un programa no se establece de una vez para todas, sino que debe ir siendo modificado con cierta frecuencia. Más concretamente, es bastante conocido —ver, por ejemplo, Teichroew (72) o Teichroew (74)— que los programas tienen, como las aplicaciones y los sistemas en que se integran, *un ciclo de vida*, a lo largo de la cual deben ser *mantenidos* (un programa de aplicación en uso que no haya sido modificado en todo un año es probablemente un programa *muerto:* nadie utiliza los resultados que produce).

Este mantenimiento de un programa será más fácil, y en consecuencia menos costoso, si su documentación es adecuada, y en particular si el lenguaje empleado ha sido funcional y no de segunda (y no digamos de primera) generación. Por lo tanto, aunque no demos una nueva expresión de N que tenga en cuenta los costes de mantenimiento, sí podremos decir que su valor será *mayor* que el dado por la fórmula suprascrita. No nos extrañará pues que compiladores, e incluso sistemas de explotación, se escriban, como luego veremos, en lenguajes funcionales.

Si a estas razones económicas en favor de los lenguajes de alto nivel añadimos, entre otras, la ya citada de su supervivencia a un cambio de ordenador, parece lógica la enorme disminución que ha tenido el uso de los de segunda generación en los últimos diez años. Y sin embargo, según la encuesta de Philippakis (73), los Assembler alcanzaban todavía el año 72 un índice de uso de 20 (frente a 78 para los otros lenguajes) entre las empresas estadounidenses (con una disminución del índice de 24 como promedio de los dos años precedentes).

NOTAS

(1) Se exponen aquí ideas personales, que no coinciden necesariamente con las de IBM.

Por otra parte, varias de estas ideas las he explicado en diversas ocasiones, en especial en el Simposio sobre «Estado actual y perspectivas de los lenguajes de proceso de datos (Software)» de la A.C.H.N.A. (Madrid, diciembre de 1970 —y agradezco los comentarios de D. Andrés Cristóbal—, y en el Seminario sobre «Diseño de Sistemas Informáticos» de la A.T.I. (Barcelona, febrero-marzo de 1972).

Se presupone en el lector conocimientos de programación de ordenadores.

(2) En el esquema clasificatorio (como asimismo en otras muchas ideas) no pretendo ser original, pues es solamente una ampliación de una clasificación clásica, que puede encontrarse por ejemplo en Camps & Martí (67), o en «Aplicación de la lingüística a la informática: Comunicación hombre-máquina: sus problemas. Tipos de lenguajes de programación», en Aladjem y otros (71).

(3) En particular, excluimos:

— Los lenguajes informáticos de comunicación hombre-máquina que no son para uso preferente por los programadores: a) lenguajes de encargo de trabajos a los sistemas de explotación (Job Control Languajes), en diferido (batch) o en inmediato (time sharing), y tanto si la explotación es de un solo ordenador como en los casos de varios (multiproceso, redes de ordenadores); b) lenguajes de acceso a bases de datos y ficheros por parte de no-programadores; c) lenguajes de análisis; d) lenguajes de microprogramación; etcétera.

— Los lenguajes de comunicación hombre-hombre, tales como ordinogramas, diagramas HIPO, etc.

— Los lenguajes de comunicación máquina-máquina, tales como los «protocolos» internos de las redes informáticas.

Así pues, en lo que sigue «lenguaje» deberá entenderse como «lenguaje de programación de ordenadores», salvo mención en contrario.

(4) La referencia usual es Sammet (69), si bien los socios de la A.T.I. tendrán más a mano Camps & Martí (67).

(5) Por ejemplo, Ph. Poré en el I Seminario sobre «Sistemas informáticos de dirección» de la A.T.I. (Barcelona, noviembre/diciembre de 1969).

(6) Los esquemas gráficos de estos cableados sobre impresos imagen de los tableros no son los programas, sino representaciones de los mismos a efectos de documentación, es decir, de comunicación hombre-hombre.

(7) Para una explicación de *sintaxis* y de *semántica* consúltese cualquier texto de lingüística; a los socios de la A.T.I. quizás les sea más asequible «Introducción a la semiótica para informáticos», en Aladjem y otros (71).

(8) Es claro que los *poderes* o niveles semánticos de los lenguajes de máquina son distintos para distintos ordenadores (mucho mayor en un lenguaje de doscientas instrucciones, que incluya la mayoría de operaciones, que en un lenguaje de dieciséis instrucciones, que requiera una subrutina de software para multiplicar, dividir, etc.), pero para cada caso tomamos el lenguaje de máquina como el origen de coordenadas.

Por otra parte, el nivel semántico de las unidades hardware de un ordenador puede ser inferior al nivel semántico de su lenguaje de máquina, siendo un *microprograma* (a veces llamado *firmware*) el intérprete que hace el puente entre ambos niveles; en este caso el «lenguaje de máquina» no es el «lenguaje del hardware», y éste último tendrá un nivel «negativo» en el eje semántico adoptado.

Notaremos aquí que la aparición posterior de prototipos y máquinas cuyos lenguajes de máquina son de los que luego llamaremos funcionales (Fortran, Euler, PL/I, APL, etc.) no nos hace catalogar a tales lenguajes como de primera generación, ya que dejamos estos casos fuera de nuestro esquema mientras no tengan, como posiblemente ocurrirá más adelante, un uso significativo.

Señalemos también que las máquinas de microprograma variable tendrán varios «lenguajes de máquina» (según sea el microprograma que se les cargue), aunque un solo «lenguaje del hardware».

(9) No hace falta decir que la adopción de un solo eje tanto en sintaxis como en semántica es ya una simplificación notable, pues para cada una podrían usarse varias dimensiones. Por otra parte la división entre aspectos sintácticos y aspectos semánticos (y entre ambos y los *pragmáticos*) no es terminante.

(10) La *co-rutina*, es decir, la rutina durante cuya ejecución no queda suspendida la de la rutina que la llamó, sino que ambas prosiguen «simultáneamente» mediante el *multitasking* o *subtasking*, aparece posteriormente, y no sólo en estos lenguajes, sino también en los que llamaremos funcionales. (La co-rutina, en general, puede tener además la característica de que acontezcan en momentos distintos su carga en memoria, su recepción de argumentos como parámetros, y su inicio de ejecución, y asimismo para sus homólogos de finalización).

(11) Otros denominan «problem oriented language» o POL, no a todo lenguaje funcional, sino sólo a los apropiados para una aplicación específica, como por ejemplo la simulación o la topografía.

(12) Ofrece dificultades la evaluación concreta de estos costes en cada caso, y en particular la del coste de ejecutar en el caso de multiprogramación, pero esta dificultad no implica la inexistencia de dichos costes, ni invalida pues la argumentación.

(13) No afirmamos que los costes sean iguales para todos los lenguajes funcionales —y luego para todos los de segunda generación—, sino que el razonamiento es válido cualquiera que sea el lenguaje concreto en uno y otro caso.

BIBLIOGRAFIA

Aladjem y otros (71): «Lingüística e Informática». A.T.I. Barcelona 1971.

Camps & Martí (67): «Visión general del software». A.T.I., Barcelona 1967. (Hay una edición revisada de 1972.)

Cheatham (72): «The recent evolution of programming languages» en «Information processing 71». North-Holland, Amsterdam 1972.

Freixa (67): «Cálculo digital». Real Academia de Ciencias y Artes, Barcelona 1967.

Higman (67): «A comparative study of programming languages» 1967.

Philippakis (73): «Programming language usage». Datamation, October 1973.

Rosen (67): «Programming systems and languages». McGraw-Hill, New York 1967.

Sammet (69): «Programming languages: History and fundamentals». Prentice-Hall 1969.

Teichroew (72): «Improvements in systems building». Seminario sobre «Diseño de sistemas informáticos». A.T.I., Barcelona 1972.

Teichroew (74): «Improvements in the system life cycle». En «Information Processing 74» preprints. Stockholm 1974