EVOLUCIÓN HISTÓRICA DE LOS LENGUAJES DE PROGRAMACIÓN

3

Jon Ander Gómez Adrián
Universitat Politècnica de València

Resumen: En la presente revisión de la evolución de los lenguajes de programación vamos a evitar un repaso exhaustivo de todos ellos, en parte porque no aportaría una mejor perspectiva pero sobre todo porque la lista es de tal magnitud que resulta imposible de abarcar. Remitimos al lector a otras publicaciones más completas donde podrá ver cómo la mayoría de estudios hablan de como máximo 50 lenguajes de programación de entre los aproximadamente 2.500 que se supone existen o han existido.

El propósito de nuestra aportación consiste en analizar la evolución de los lenguajes de programación a tenor de las necesidades de la industria y en algunos casos del mundo académico, como es el caso del Pascal, diseñado por el profesor Niklaus Wirth para enseñar a programar de manera estructurada a sus alumnos. Pero antes vamos a ver algunas definiciones, seguidas de una taxonomía de los lenguajes de programación y qué características definen cada paradigma de programación. Conocer previamente cómo pueden clasificarse los lenguajes de programación servirá de ayuda para entender mejor su evolución histórica.

¿QUÉ ES UN LENGUAJE DE PROGRAMACIÓN?

Existen varias definiciones de lenguaje de programación. Desde un punto de vista práctico se puede definir como el *instrumento que permite a los humanos comunicar instrucciones u órdenes a las computadoras*. Es obvio que no puede admitir ambigüedades como es el caso del lenguaje natural que utilizamos los seres humanos para comunicarnos. El hardware ejecuta exactamente lo que se le dice que ejecute y en el orden exacto que se le ha dicho. Con esta idea en mente se entenderá mejor la siguiente definición de lenguaje de programación: *lenguaje formal definido artificialmente para expresar algoritmos que, instrucción a instrucción, ejecutará una computadora*. Se utiliza el adjetivo "formal" porque un lenguaje de programación está sujeto a una serie de reglas sintácticas que definen sin ambigüedad la semántica de las expresiones que podemos utilizar para indicar las operaciones al ordenador. En base al carácter formal de los lenguajes de programación aportamos otra definición complementaria a las anteriores: *conjunto de reglas, símbolos, operadores y palabras reservadas que utilizaremos para escribir programas de ordenador*. Los símbolos, los operadores y las palabras reservadas tienen un significado asociado, por ejemplo el operador '+' se utiliza para indicar al ordenador que realice la suma de dos valores. Por último, otra definición más completa es *notación, conjunto de reglas y definiciones que determinan tanto lo*

que puede escribirse en un programa, y que el procesador puede interpretar, como el resultado de la ejecución de dicho programa por el procesador.

Algoritmo y expresión (en el contexto de la programación de ordenadores) son dos conceptos muy relacionados que servirán de apoyo para entender mejor qué es un lenguaje de programación y su utilidad. De entre las posibles definiciones de algoritmo hemos seleccionado la de Donald E. Knuth (1968), pues nos parece la más apropiada: *secuencia finita de instrucciones, reglas o pasos que describen de manera precisa las operaciones que un ordenador ha de ejecutar para llevar a cabo una tarea en un tiempo finito*. Una expresión es una *combinación de variables, constantes, operadores y nombres de función que tiene la propiedad de ser evaluada*. La definición de algoritmo está muy relacionada con las definiciones de lenguaje de programación vistas más arriba. En esta definición se insiste en la finitud de la secuencia de instrucciones y de llevar a cabo una tarea. No tendría sentido escribir un programa cuya secuencia de instrucciones fuese infinita, ni serviría de nada un algoritmo que no finalizase su ejecución para darnos la solución a un problema. Otro aspecto señalado es describir de manera precisa las operaciones a realizar, es decir, sin ambigüedad, pues el procesador no puede "interpretar" qué quiere decir el programador. La definición de expresión también ayuda a ver que las instrucciones debemos expresarlas de manera precisa combinando distintos elementos y siguiendo un conjunto de reglas y una notación específica, de manera que el resultado obtenido tras evaluar una expresión esté bien definido una vez conocidos los valores de las variables y las constantes que aparezcan en la expresión, así como las funciones que se invoquen. El resultado siempre será el mismo para los mismos valores de entrada.

En contraposición a lo estricto que resulta un lenguaje de programación, las personas utilizamos en muchas ocasiones lo que se conoce como pseudocódigo para compartir esbozos de algoritmos en las primeras etapas del diseño de aplicaciones. El pseudocódigo también resulta de mucha utilidad para explicar algoritmos en libros o en clase. Se caracteriza porque permite incluir lenguaje natural, dado que nunca será compilado o interpretado para ser ejecutado.

Los lenguajes de programación nos proporcionan mecanismos de abstracción para poder definir estructuras de datos y las operaciones o transformaciones que pueden realizarse sobre dichos datos. También incluyen mecanismos para el control del flujo de los programas que nos permiten establecer, con cierto nivel de abstracción, qué bloques de operaciones se ejecutarán repetidamente, o qué secuencias de instrucciones se ejecutarán en distintos momentos de la ejecución del programa habiéndolas escrito una sola vez. Los programadores utilizan abstracciones para representar conceptos y definir los cálculos que se realizarán en base a dichos conceptos. Los conceptos se representan en base a una serie de elementos básicos como pueden ser los datos numéricos y las operaciones que sobre ellos se pueden realizar. Programar, por tanto, consiste en combinar adecuadamente los elementos básicos que forman parte del lenguaje de programación para crear estructuras y cálculos más complejos, y más abstractos por

tanto, para poder obtener soluciones a problemas de cierta dificultad. Utilizando un símil de otro contexto, un escultor crea una obra a partir de ciertos materiales y utilizando sus herramientas; dependiendo de sus habilidades el resultado de su trabajo será más o menos valorado. Podemos decir que un lenguaje de programación proporciona las herramientas básicas para crear programas y, dependiendo del programador, el programa o aplicación resultante será de mejor o peor calidad.

LA LIBRERÍA ESTÁNDAR

Es común que los lenguajes de programación vayan acompañados de su librería estándar y, cada vez más, de su entorno de ejecución. Muchos lenguajes de programación llevan asociada una librería de funciones básicas como pueden ser las funciones matemáticas más habituales o las de entrada/salida. También se incluyen algunas estructuras de datos como soporte para utilizar algunas funciones de la librería. Este conjunto de funciones se conoce como librería estándar. Es difícil determinar si dichas funciones forman o no parte del lenguaje como tal, pero sin embargo, los programas necesitan tenerlas disponibles en el momento de su ejecución. Hasta mediados de los años 90 del siglo XX, la librería estándar o parte de ella se incluía en el fichero ejecutable resultado del proceso de compilación. La evolución de los sistemas operativos ha traído consigo que no sea necesario incluir dentro del fichero ejecutable el código compilado correspondiente a la librería estándar, por ello, desde hace unos años se habla de entorno de ejecución para un lenguaje de programación. Este entorno puede ser un conjunto de librerías dinámicas que el sistema operativo cargará en memoria cuando el programa lo necesite durante su ejecución, o una máquina virtual como es el caso de Java. Las librerías incluyen cada vez más funcionalidad, a beneficio de los programadores que ya no necesitan desarrollar ciertas funciones hoy consideradas como básicas. Durante los años 80 y 90 del siglo XX era habitual comprar ciertas librerías de manera adicional al compilador.

Volviendo a si la librería estándar puede considerarse parte o no del lenguaje, algo que siempre queda difuso, vamos a ver como ejemplo las cadenas de caracteres en Java. En realidad son objetos de la clase *String,* pero el compilador les da un tratamiento especial como si fuesen tipos de datos elementales. Utilizando las comillas dobles como delimitador se pueden manipular constantes de cadenas de caracteres que son, a todos los efectos, objetos de la clase *String*. De hecho, se permite el uso de operadores para estos objetos, algo que no es posible para objetos de otras clases salvo las clases que son una extensión de los tipos de datos nativos. En las últimas versiones de Java se permite que las constantes de tipo *String* se utilicen en la estructura de selección *switch* para el control de flujo de los programas. Formalmente, la clase *String* es una clase más de las predefinidas en la librería estándar de Java, pero no se concibe que un programa en Java prescinda de la clase *String*.

NIVELES DE LOS LENGUAJES DE PROGRAMACIÓN

Algunos intentos de clasificar los lenguajes de programación han seguido diversos criterios. Antes de pasar a la siguiente sección que versa sobre los paradigmas de programación, vamos a ver otros dos criterios utilizados a lo largo de la historia de los lenguajes de programación para clasificarlos: por niveles y por generaciones.

El nivel más bajo de programación es programar directamente utilizando el lenguaje máquina, codificando las instrucciones y los datos mediante ceros y unos. Esta práctica comenzó nada más aparecer los primeros ordenadores en las décadas de los 40 y los 50 del siglo XX. Nos podemos imaginar la cantidad de horas necesaria para escribir un simple programa y la cantidad de errores que se cometían. Era una labor altamente compleja y poco gratificante. Era la época de las tarjetas perforadas, aunque éstas comenzaron a utilizarse mucho antes para otros propósitos. Las instrucciones del código máquina se grababan en las tarjetas perforadas línea a línea, y cada línea contenía la correspondiente combinación de ceros y unos.

Con la aparición de los ensambladores la labor de programar resultó más eficiente aunque casi igual de compleja. Un lenguaje ensamblador es dependiente del hardware, de hecho, consiste en utilizar códigos nemotécnicos en lugar de unos y ceros para indicar las instrucciones y referenciar los registros. Cada modelo de procesador o microprocesador tiene su propio código máquina y por tanto su propio ensamblador.

Programar en ensamblador o en código máquina es lo que se conoce como **programar a bajo nivel**. El programador debe conocer muy bien la arquitectura del procesador para el cual programa, es decir, conocer qué registros tiene el procesador, qué propósito tiene cada registro, qué niveles de indirección se permiten para acceder a la memoria, cómo pueden ejecutarse subrutinas pasándoles argumentos y de qué mecanismos dispone el procesador para cambiar de contexto, es decir, para facilitar la multitarea.

A finales de la década de 1950 aparecieron los primeros lenguajes de programación que ya no eran dependientes del hardware sobre el cual se ejecutarían los programas. Los más representativos son COBOL, FORTRAN y ALGOL, aunque debemos señalar que hubo algunos diseños previos a partir de los cuales se concretaron estos tres, y que los tres sufrieron importantes cambios durante sus primeros años, de hecho el ALGOL 68 tenía importantes mejoras respecto de su antecesor el ALGOL 60. A estos lenguajes independientes del hardware se les conoce como **lenguajes de alto nivel** en contraposición a los lenguajes de bajo nivel. El concepto de compilador aparece ligado a los lenguajes de alto nivel. Un **compilador** es un programa que traduce los programas escritos en un lenguaje de programación de alto nivel, es decir independiente del procesador, al código máquina que sí ejecutará directamente el procesador. Aunque al principio los programadores debían conocer muchos detalles de la arquitectura sobre la que se ejecutarían sus programas, los lenguajes de alto nivel facilitaron, progresivamente, que los programadores pudiesen abstraerse de las características del hardware para concentrarse en los problemas a resolver.

Los lenguajes de alto nivel son mayoritariamente compilados, es decir, el código fuente se compila una sola vez para generar el código máquina. Después, el código máquina podrá ejecutarse tantas veces como sea necesario. Solo deberá compilarse de nuevo si se realizan modificaciones en el código fuente. Hay algunos lenguajes de programación que son interpretados, es decir, el programa fuente se va procesando y ejecutando línea a línea. Obviamente este proceso es más lento que ejecutar directamente el código máquina. Existe otra vía intermedia, puesta de moda gracias a Java, donde el código fuente se compila a un código intermedio independiente del hardware, conocido como *bytecode*, que una máquina virtual se encarga de ejecutar. Si el código intermedio se traduce a código máquina antes de ejecutarse o se ejecuta mediante una máquina virtual es algo que depende de la implementación.

En resumen, los lenguajes de programación pueden clasificarse en dos niveles, alto y bajo, según su dependencia del hardware. Para según qué tareas es bastante habitual combinar partes escritas en ensamblador dentro de un programa escrito en un lenguaje de alto nivel, por ejemplo para conseguir la versión más eficiente posible de algunas funciones que se utilizan repetidamente en un programa cuando la eficiencia es relevante.

LAS GENERACIONES DE LOS LENGUAJES

Aprovechando la división anterior en dos niveles, los lenguajes de programación también se clasifican por generaciones. El código máquina se considera representante de los lenguajes de **primera generación**. Cada diseño de procesador tiene su propio código máquina. Los ensambladores son considerados como lenguajes de **segunda generación**. Esta diferenciación comenzó a utilizarse cuando se acuñó el término **tercera generación** para identificar los lenguajes de programación de alto nivel que son independientes del hardware. Los lenguajes más representativos de la tercera generación son C, Ada, C++, C# y Java.

La propuesta de lenguajes de **cuarta generación** se desarrolló a lo largo de tres décadas, desde principios de los 70 a finales de los 90 del siglo XX. Nació con el propósito de diseñar nuevos lenguajes de programación que fuesen un refinamiento y una evolución de los lenguajes de tercera generación. Se buscaba que cada nueva generación de lenguajes de programación aumentase el nivel de abstracción con respecto a los detalles del hardware, es decir, que los programadores no necesitasen conocer detalles del sistema sobre el cual se ejecutarían sus programas. A cada nueva generación los lenguajes serían más cercanos al lenguaje natural de manera que la tarea de programar fuese más amigable, productiva y versátil.

La definición de lenguajes de cuarta generación fue evolucionando hasta quedar en desuso. El rasgo más relevante de esta generación es el de incluir soporte en la propia definición de los lenguajes para el acceso a bases de datos, generación de informes, funciones matemáticas más avanzadas, gestión del interfaz gráfico con el usuario o

desarrollo web. Todas estas funcionalidades tenían un denominador común: permitir manejar grandes colecciones de datos de manera sencilla para el programador, que le resultase transparente el tener que traer a la memoria del ordenador los datos con los que estaba trabajando. Por ejemplo, en el caso de acceder a una fuente de datos externa cuyo tamaño excede la capacidad de memoria del ordenador, en un lenguaje de cuarta generación sería transparente para el programador si en memoria solo hay disponible una porción de toda una lista de registros, se accedería como si todo estuviese en memoria y serían las funciones de la librería del propio lenguaje las que se encargarían de ir actualizando los datos en ambos sentidos (de la memoria a la fuente de datos externa y viceversa).

Lenguajes como Python, Ruby y Perl incluyen muchas características de los lenguajes de cuarta generación aunque en realidad son lenguajes de tercera generación algo más avanzados. Otros lenguajes como los orientados a objetos C++, C# y Java, incluyen una librería de funciones cada vez más completa que permite al programador trabajar con todas las funcionalidades previstas para los lenguajes de cuarta generación. Por ello, la frontera entre ambas generaciones de lenguajes ha quedado muy diluida. A pesar de ello, debemos recordar que hubo una explosión de nuevos lenguajes siguiendo la definición de lenguaje de cuarta generación, cada uno dedicado a un ámbito de aplicación. Pueden destacarse ABAP de SAP, Informix 4GL, Ingres 4GL, SQL, FORTH, Clipper, Data Flex, etc. En Wikipedia 4GL (2015) se encuentra una buena clasificación de los lenguajes de cuarta generación según ámbitos de aplicación, muchos de ellos en desuso actualmente.

Los lenguajes de **quinta generación** se definen como los lenguajes en los que programar consiste en definir los requisitos para resolver problemas en lugar de que un programador escriba un algoritmo. Muchos de los lenguajes pertenecientes al paradigma de programación declarativa son lenguajes de quinta generación, por ejemplo Prolog, OPS5 y Mercury. Los lenguajes de quinta generación están diseñados para que la computadora resuelva los problemas sin ayuda de programadores. De esta manera, el programador solo debe preocuparse de qué problemas pueden solucionarse y qué condiciones son necesarias que se cumplan para encontrar la solución a un determinado problema, sin necesidad de implementar algoritmos concretos que resuelvan los problemas.

La idea de quinta generación de lenguajes de programación se puso de moda en la década de 1980 considerándose como el futuro de la informática. Hubo quien se atrevió a vaticinar que reemplazarían a los lenguajes existentes incluso para el desarrollo de los sistemas operativos. Es destacable el hecho de que entre 1982 y 1993 en Japón se dedicaron muchas inversiones a la investigación en esta generación de lenguajes de programación con el objetivo de poner en funcionamiento una gran red de ordenadores utilizando esta tecnología. Pronto se vio que el paso que va desde definir los requisitos para desarrollar la solución a un problema hasta la implementación de los algoritmos era un salto cualitativo muy importante, labor que de momento sigue estando a cargo