LENGUAJES DE PROGRAMACIÓN
¿POR QUÉ HAY TANTOS Y APARECEN NUEVOS?

Por: Hanna Oktaba

La computadora, a diferencia de otras herramientas que en general apoyan el esfuerzo físico de los humanos, fue inventada para facilitar el trabajo intelectual. Si el hombre tiene algún problema, por ejemplo "sumar dos y dos", el diseñador define el algoritmo que resuelve el problema, el programador lo codifica en un lenguaje de programación, el cual la computadora es capaz de "entender", luego la computadora ejecuta el algorítmo expresado como programa en el lenguaje de programación en cuestión, y listo. La máquina le entrega al hombre la respuesta "4", sin que éste tuviera que esforzar sus neuronas.

¿Cuál es el papel del lenguaje de programación en este proceso? Es muy importante, el lenguaje de programación es el medio de comunicación entre el hombre y la máquina. El modelo general de las computadoras, desde que fue esbozado por von Neumann, no ha cambiado mucho, mientras que la invención humana para proponerse nuevos problemas a resolver, usando la computadora, parece no tener límites. En consecuencia, los lenguajes de programación tienen que adaptarse a éstas crecientes necesidades y aumentar la expresividad para poder resolver problemas muy diversos y cada vez más complejos. Además, tienen que ofrecer cierta eficiencia en la ejecución. Es un logro difícil de alcanzar y por lo tanto, se requiere una búsqueda constante de nuevos lenguajes para ello.

Permítanme exponer un breve panorama de los más importantes tipos de lenguajes de programación.

Lenguajes Imperativos

Comenzaré por los llamados lenguajes imperativos. En este tipo de lenguajes, cuyo origen está ligado a la propia arquitectura de von Neumann, la arquitectura consta de una secuencia de celdas, llamadas memoria, en la cual se pueden guardar en forma codificada, lo mismo datos que instrucciones; y de un procesador, el cual es capaz de ejecutar de manera secuencial una serie de operaciones, principalmente aritméticas y booleanas, llamadas comandos. En general, un lenguaje imperativo ofrece al programador conceptos que se traducen de forma natural al modelo de la máquina.

Los lenguajes imperativos más destacados de la historia han sido: FORTRAN, Algol, Pascal, C, Modula-2, Ada. Seguramente, los lectores conocen por lo menos uno de ellos.

El programador, al utilizar un lenguaje imperativo, por lo general tiene que traducir la solución abstracta del problema a términos muy primitivos, cercanos a la máquina. La distancia entre el nivel del razonamiento humano y lo expresable por los lenguajes imperativos causa que sus programas sean más "comprensibles" para la máquina que para el hombre. Esta desventaja para nosotros, reflejada en la dificultad que tenemos al construir programas en un lenguaje imperativo, se vuelve una ventaja en el momento de la generación del código. El programa está expresado en términos tan cercanos a la máquina, que el código generado es relativamente parecido al programa original, lo que permite cierta eficiencia en la ejecución.

Lenguajes Funcionales
Los matemáticos desde hace un buen tiempo están resolviendo problemas usando el concepto de función. Una función convierte ciertos datos en resultados. Si supiéramos cómo evaluar una función, usando la computadora, podríamos resolver automáticamente muchos problemas. Así pensaron algunos matemáticos, que no le tenían miedo a la máquina, e inventaron los lenguajes de programación funcionales. Además, aprovecharon la posibilidad que tienen las funciones para manipular datos simbólicos, y no solamente numéricos, y la propiedad de las funciones que les permite componer, creando de esta manera, la oportunidad para resolver problemas complejos a partir de las soluciones a otros más sencillos. También se incluyó la posibilidad de definir funciones recursivamente.

Un lenguaje funcional ofrece conceptos que son muy entendibles y relativamente fáciles de manejar para todos los que no se durmieron en las clases de matemáticas. El lenguaje funcional más antiguo, y seguramente el más popular hasta la fecha, es LISP, diseñado por McCarthy [1] en la segunda mitad de los años 50. Su área de aplicación es principalmente la Inteligencia Artificial. En la década de los 80 hubo una nueva ola de interés por los lenguajes funcionales, añadiendo la tipificación y algunos conceptos modernos de modularización y polimorfismo, como es el caso del lenguaje ML.

Programar en un lenguaje funcional significa construir funciones a partir de las ya existentes. Por lo tanto es importante conocer y comprender bien las funciones que conforman la base del lenguaje, así como las que ya fueron definidas previamente. De esta manera se pueden ir construyendo aplicaciones cada vez más complejas. La desventaja de este modelo es que resulta bastante alejado del modelo de la máquina de von Neumann y, por lo tanto, la eficiencia de ejecución de los intérpretes de lenguajes funcionales no es comparable con la ejecución de los programas imperativos precompilados. Para remediar la deficiencia, se está buscando utilizar arquitecturas paralelas que mejoren el desempeño de los programas funcionales, sin que hasta la fecha estos intentos tengan un impacto real importante.

Lenguajes Lógicos
Otra forma de razonar para resolver problemas en matemáticas se fundamenta en la lógica de primer orden. El conocimiento básico de las matemáticas se puede representar en la lógica en forma de axiomas, a los cuales se añaden reglas formales para deducir cosas verdaderas (teoremas) a partir de los axiomas. Gracias al trabajo de algunos matemáticos, de finales de siglo pasado y principios de éste, se encontró la manera de automatizar computacionalmente el razonamiento lógico --particularmente para un subconjunto significativo de la lógica de primer orden-- que permitió que la lógica matemática diera origen a otro tipo de lenguajes de programación, conocidos como lenguajes lógicos. También se conoce a estos lenguajes, y a los funcionales, como lenguajes declarativos, porque el programador, parar solucionar un problema, todo lo que tiene que hacer es describirlo vía axiomas y reglas de deducción en el caso de la programación lógica y vía funciones en el caso de la programación funcional.

En los lenguajes lógicos se utiliza el formalismo de la lógica para representar el conocimiento sobre un problema y para hacer preguntas que, si se demuestra que se pueden deducir a partir del conocimiento dado en forma de axiomas y de las reglas de deducción

estipuladas, se vuelven teoremas. Así se encuentran soluciones a problemas formulados como preguntas. Con base en la información expresada dentro de la lógica de primer orden, se formulan las preguntas sobre el dominio del problema y el intérprete del lenguaje lógico trata de encontrar la respuesta automáticamente. El conocimiento sobre el problema se expresa en forma de predicados (axiomas) que establecen relaciones sobre los símbolos que representan los datos del dominio del problema.

El PROLOG es el primer lenguaje lógico y el más conocido y utilizado. Sus orígenes se remotan a los inicios de la década de los 70 con los trabajos del grupo de A. Colmerauer [2] en Marsella, Francia. También en este caso, las aplicaciones a la Inteligencia Artificial mantienen el lenguaje vivo y útil.

En el caso de la programación lógica, el trabajo del programador se restringe a la buena descripción del problema en forma de hechos y reglas. A partir de ésta se pueden encontrar muchas soluciones dependiendo de como se formulen las preguntas (metas), que tienen sentido para el problema. Si el programa está bien definido, el sistema encuentra automáticamente las respuestas a las preguntas formuladas. En este caso ya no es necesario definir el algoritmo de solución, como en la programación imperativa, en cambio, lo fundamental aquí es expresar bien el conocimiento sobre el problema mismo. En programación lógica, al igual que en programación funcional, el programa, en este caso los hechos y las reglas, están muy alejados del modelo von Neumann que posee la máquina en la que tienen que ser interpretados; por lo tanto, la eficiencia de la ejecución no puede ser comparable con la de un programa equivalente escrito en un lenguaje imperativo. Sin embargo, para cierto tipo de problemas, la formulación del programa mismo puede ser mucho más sencilla y natural (para un programador experimentado, por supuesto).

Lenguajes Orientados a Objetos

A mediados de los años 60 se empezó a vislumbrar el uso de las computadoras para la simulación de problemas del mundo real. Pero el mundo real está lleno de objetos, en la mayoría de los casos complejos, los cuales difícilmente se traducen a los tipos de datos primitivos de los lenguajes imperativos. Así es que a dos noruegos, Dahl y Nygaard [3], se les ocurrió el concepto de *objeto* y sus colecciones, llamadas *clases de objetos*, que permitieron introducir abstracciones de datos a los lenguajes de programación. La posibilidad de reutilización del código y sus indispensables modificaciones, se reflejaron en la idea de las jerarquías de *herencia de clases*. A ellos también les debemos el concepto de polimorfismo introducido vía procedimientos virtuales. Todos estos conceptos fueron presentados en el lenguaje Simula 67, desde el año 1967. Aunque pensado como lenguaje de propósito general, Simula tuvo su mayor éxito en las aplicaciones de simulación discreta, gracias a la clase SIMULATION que facilitaba considerablemente la programación.

La comunidad informática ha tardado demasiado en entender la utilidad de los conceptos básicos de Simula 67, que hoy identificamos como conceptos del modelo de objetos. Tuvimos que esperar hasta los años 80 para vivir una verdadera ola de propuestas de lenguajes de programación con conceptos de objetos encabezada por Smalltalk, C++, Eiffel, Modula-3, Ada 95 y terminando con Java. La moda de objetos se ha extendido de los lenguajes de programación a la Ingeniería de *Software*. Si alguien desarrolla sistemas y

todavía no sabe qué es un objeto, mejor que no lo confiese en público y lea cuanto antes los números 3, 4 y 20 de la revista Soluciones Avanzadas.

El modelo de objetos, y los lenguajes que lo usan, parecen facilitar la construcción de sistemas o programas en forma modular. Los objetos ayudan a expresar programas en términos de abstracciones del mundo real, lo que aumenta su comprensión. La clase ofrece cierto tipo de modularización que facilita las modificaciones al sistema. La reutilización de clases previamente probadas en distintos sistemas también es otro punto a favor. Sin embargo, el modelo de objetos, a la hora de ser interpretado en la arquitectura von Neumann conlleva un excesivo manejo dinámico de memoria debido a la constante creación de objetos, así como a una carga de código fuerte causada por la constante invocación de métodos. Por lo tanto, los programas en lenguajes orientados a objetos siempre pierden en eficiencia, en tiempo y memoria, contra los programas equivalentes en lenguajes imperativos. Para consolarnos, los expertos dicen que les ganan en la comprensión de código.

Lenguajes Concurrentes, Paralelos y Distribuidos

La necesidad de ofrecer concurrencia en el acceso a los recursos computacionales se remonta a los primeros sistemas operativos. Mientras que un programa realizaba una operación de entrada o salida otro podría gozar del tiempo del procesador para sumar dos números, por ejemplo. Aprovechar al máximo los recursos computacionales fue una necesidad apremiante, sobre todo en la época en que las computadoras eran caras y escasas; el sistema operativo tenía que ofrecer la ejecución concurrente y segura de programas de varios usuarios, que desde distintas terminales utilizaban un solo procesador, y así surgió la necesidad de introducir algunos conceptos de *programación concurrente* para programar los sistemas operativos.

Posteriormente, cuando los procesadores cambiaron de tamaño y de precio, se abrió la posibilidad de contar con varios procesadores en una máquina y ofrecer el *procesamiento en paralelo*, es decir, procesar varios programas al mismo tiempo. Esto dio el impulso a la creación de lenguajes que permitían expresar el paralelismo. Finalmente, llegaron las redes de computadoras, que también ofrecen la posibilidad de ejecución en paralelo, pero con procesadores distantes, lo cual conocemos como la *programación distribuida*.

En resumen, el origen de los conceptos para el manejo de concurrencia, paralelismo y distribución está en el deseo de aprovechar al máximo la arquitectura von Neumann y sus modalidades reflejadas en conexiones paralelas y distribuidas.

Históricamente encontramos en la literatura soluciones conceptuales y mecanismos tales como: semáforos, regiones críticas, monitores, envío de mensajes (CSP), llamadas a procedimientos remotos (RPC), que posteriormente se incluyeron como partes de los lenguajes de programación en Concurrent Pascal, Modula, Ada, OCCAM, y últimamente en Java.

Uno de los ejemplos más importantes es el modelo de envío de mensajes de CSP de Hoare [4], para las arquitecturas paralelas y distribuidas, el cual no solamente fructificó en una propuesta del lenguaje de programación OCCAM, sino dio origen a una nueva familia de

procesadores, llamados "transputers", que fácilmente se componen en una arquitectura paralela.

Es difícil evaluar las propuestas existentes de lenguajes para la programación concurrente, paralela y distribuida. Primero, porque los programadores están acostumbrados a la programación secuencial y cualquier uso de estos mecanismos les dificulta la construcción y el análisis de programas. Por otro lado, este tipo de conceptos en el pasado fue manejado principalmente a nivel de sistemas operativos, protocolos de comunicación, etcétera, donde la eficiencia era crucial, y por lo tanto no se utilizaban lenguajes de alto nivel para la programación. Hoy en día, la programación de sistemas complejos tiene que incluir las partes de comunicaciones, programación distribuida y concurrencia. Esto lo saben los creadores de los lenguajes más recientes, que integran conceptos para manejar: los hilos de control, comunicación, sincronización y no determinismo; el hardware y las aplicaciones se los exigen.

A manera de conclusiones

A quien pensaba que en la materia de los lenguajes de programación ya se había dicho todo, espero haberlo convencido de que estaba equivocado. Como dice Hoare [4]: "El diseño de un lenguaje de programación es siempre un compromiso, en el cual un buen diseñador tiene que tomar en cuenta: el nivel de abstracciones deseado, la arquitectura del *hardware*, y el rango propuesto de las aplicaciones".

Lo malo, o lo bueno, según lo veamos, es que queremos construir sistemas a niveles cada vez más abstractos, con *hardware* cada vez más complicado y con aplicaciones cada vez más ambiciosas (véase la reciente efervescencia con Java y sus aplicaciones en Internet). Por lo tanto, el trabajo para los diseñadores de lenguajes existe para un buen rato.

Una versión completa de este artículo aparecerá en un número próximo a salir de la revista Soluciones Avanzadas, dedicado a los lenguajes de programación.

Referencias

[1] *J. Mcarthy et al., LISP 1.5 "Programming Manual Cambridge" MA. MIT Press, 1965.*

[2] *A. Colmeraure and P. Roussel, " The Birth of PROLOG", ACM SlgPlan Notices, Vol. 28, No. 3, March 1993.*

[3] *O.J. Dahl, B. Myhrhaug, K. Nygaard, "Simula 67 Common Base Language", Norwegian Computer Center 1970.*

[4] *C.A.R. Hoare, "Communicating Sequential Processes", CACM, Vol. 21, No. 8, August 1978.*

Dra. Hanna Oktaba, Investigadora de la UNAM.
Dirección electrónica: oktaba@servidor.unam.mx