Introducción a los lenguajes de programación

Pedro O. Pérez M., PhD.

Implementación de métodos computacionales
Tecnológico de Monterrey

pperezm@tec.mx

01-2022

Contenido I

Contenido II

Probar los programas es fundamental. Sin embargo, la realización de pruebas sólo llega hasta cierto punto, ya que no es posible probar los programas para todas las posibles entrada. Aún más importante, si el programa es complejo, por ejemplo contiene recursiones o iteraciones, entonces si no se comprende qué es lo que ocurre al ejecutar un ciclo o una llamada a una función en forma recursiva, es poco probable que podamos escribir el código correctamente. Si al probar el código resulta ser incorrecto, será necesario corregirlo.

Para conseguir iteraciones o recursiones correctas, es necesario establecer hipótesis inductivas, y resulta útil razonar, formal o informalmente, que la hipótesis es coherente con la iteración o recursión. Este proceso sirve para compreder que el trabajo que realiza un programa correcto es esencialmente el mismo que el proceso de demostrar teoremas por inducción.

Como mencionamos anteriormente, una demostración deductiva consta de una secuencia de proposiciones cuya veracidad se comprueba partiendo de una proposición inicial, conocida como hipótesis o de una serie de proposiciones dadas, hasta llegar a una conclusión. Cada uno de los pasos de la demostración, hay que deducirlos mediante algún principio lógico aceptado, bien a partir de los postulados o de algunas de las proposiciones anteriores de la demostración deductiva o de una combinación de éstas.

El teorema que se demuestra partiendo de una hipótesis H para llegar a una conclusión C es la proposición "si H entonces C". Decimos entonces que C *se deduce de* H.

Por ejemplo, demostrar que:

- Teorema 1. Si $x \geq 4$, entonces $2^x \geq x^2$.
- Teorema 2. Si x es la suma de los cuadrados de cuatro enteros positivos, entonces $2^x \geq x^2$.

En los dos ejemplos anteriores, la hipótesis emplea términos familiares: enteros, suma y multiplicación, por ejemplo. En muchos otros teoremas, incluyendo los de la teoría de autómatas, el término utilizado en la proposición puede tener implicaciones que son menos obvias. Una forma útil de proceder en muchas demostraciones es: "Si no estamos seguros de cómo comenzar una demostración, convertimos todos los términos de la hipótesis a sus definiciones".

Demostrar, Teorema 3: Sea S un subconjunto finitos de un determinado conjunto infinito U. Sea T el conjunto complementario de S con respecto a U. Entonces T es infinito.

1. Un conjunto S es finito si existe un entero n tal que S tiene exactamente n elementos. Escribimos $||S||$ se utiliza para designar el número de elementos de un conjunto S. Si el conjunto S no es finito, decimos que S es infinito. Intuitivamente, un conjunto infinito es un conjunto que contiene más que cualquier número entero de elementos.
2. Si S y T son subconjuntos de algún conjunto U, entonces T es el complementario de S (con respecto a U) si $S \cup T = U$ y $S \cap T = \emptyset$. Es decir, cada elemento de U es exactamente uno de S y otro de T; dicho de otra manera, T consta exactamente de aquellos elementos de U que no pertenecen a S.
3. Demostración por reducción a lo absurdo.

Demostración formal

Sabemos que $S \cup T = U$ y que S y T son disjuntos, por lo que $||S|| + ||T|| = ||U||$. Dado que S es finito, $||S|| = n$ para algún entero n, y como U es infinito, no existe ningún entero p tal que $||U|| = p$. Por tanto, suponemos que T es infinito; es decir, $||T|| = m$ para algún m. Entonces $||U|| = ||S|| + ||T|| = n + m$, lo que contradice la proposición dada de que no existe ningún entero p que sea igual a $||U||$.

la forma “si-entonces” del teorema es la más común den las áreas típicas de las matemáticas. Sin embargo, vamos a ver otros tipos de proposiciones que también resultan ser teoremas.

En primer lugar, existen diversos tipos de enunciados de teoremas que parecen diferentes de la forma simple "si H entonces C", pero de hecho expresan lo mismo: si la hipótesis H es verdadera para un valor determinado del (o de los) parámetro(s), entonces la conclusión C es verdadera para el mismo valor. A continuación se enumeran varias formas en las que puede expresarse "si H entonces C".

1. H implica C.
2. H sólo si C.
3. C si H.
4. Si se cumple H, se cumple C.

Expresemos el Teorema 1 de las cuatro formas antes mencionadas:

1. $x \geq 4$ implica $2^x \geq x^2$.
2. $x \geq 4$ sólo si $2^x \geq x^2$.
3. $x \geq 4$ si $2^x \geq x^2$.
4. Si $x \geq 4$, entonces $2^x \geq x^2$.

Además de las formas antes mencionadas, podemos emplear el operador $\Longrightarrow$, $H \Longrightarrow C$.

Proposiciones "Si y sólo si"

En ocasiones, encontraremos proposiciones de la forma "A si y sólo B" ("A if and only if B"). Otras formas de esta proposición son "A iff b", "A es equivalente a B" o "A exactamente si B". Realmente, esta proposición se corresponde con dos proposiciones si-entonces: "si A entonces B" y "si B entonces A". En estos casos, demostraremos la proposición "A si y sólo si B" demostrando las dos proposiciones siguientes:

1. La *parte si*: "si B entonces A" y
2. La *sólo si*: "si A entonces B", lo que a menudo se escribe equivalente "A sólo si B".

Las demostraciones pueden presentarse en cualquier orden. Se puede emplear los operadores $\iff$ or $\equiv$.

Al demostrar una proposición si-y-solo-si, es importante recordar que es necesario probar tanto la parte "si" como la parte "sólo-si". En ocasiones, resultará útil dividir una proposición si-y-solo-si en una sucesión de varias equivalencias. Es decir, para demostrar "A si y sólo si B", primero hay que demostrar "A si y sólo si C" y luego demostrar "C si y sólo si B".

A continuación veremos un ejemplo de este tipo de proposiciones con una demostración sencilla. Para ello, emplearemos la siguiente notación:

1. $\lfloor x \rfloor$, el *suelo* del número real x, es el mayor entero igual o menor que x.
2. $\lceil x \rceil$, el *techo* del número real x, es el menor entero igual o mayor que x.

Demostrar que, Teorema 4. Sea x un número real. Entonces $\lfloor x \rfloor = \lceil x \rceil$ si y sólo si x es un entero.

Teoremas que parecen no ser proposiciones si-entonces

En ocasiones, nos encontraremos con teoremas que no parecen contener una hipótesis. Un ejemplo muy conocido de esto lo encontraremos en el campo de la trigonometría.

Teorema 5. $\sin^2\theta + \cos^2\theta = 1 \rightarrow$ si θ es un ángulo, entonces $\sin^2\theta + \cos^2\theta = 1$.

Otras formas de demostración

Existen varias formas en que podemos construir demostraciones:

1. Empleando conjuntos.
2. Por reducción al absurdo.
3. Mediante contraejemplo.

Demostración de equivalencias entre conjuntos

En la teoría de autómatas, frecuentemente es necesario demostrar un teorema que establece que los conjuntos construidos de dos formas diferentes son el mismo conjunto. A menudo, se trata de conjuntos de cadenas de caracteres y se denominan “lenguajes”.

Veamos un ejemplo. Si E y F son dos expresiones que representan conjuntos, la proposición $E = F$ quiere decir que los dos conjuntos representados son iguales. De forma má precisa, cada uno de los elementos del conjunto representado por E está en el conjunto representado por F, y cada uno de los elementos del conjunto representado por F está en el conjunto representado por E.
Demostrar, Teorema 6. $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$.

Figura: Pasos correspondientes a la parse "si" del Teorema 6

	Proposición	Justificación
1.	x pertenece a $R \cup (S \cap T)$	Postulado
2.	x pertenece a R o x pertenece a $S \cap T$	(1) y la definición de unión
3.	x pertenece a R o x pertenece a S y T	(2) y la definición de intersección
4.	x pertenece a $R \cup S$	(3) y la definición de unión
5.	x pertenece a $R \cup T$	(3) y la definición de unión
6.	x pertenece a $(R \cup S) \cap (R \cup T)$	(4), (5), y la definición de intersección

Figura: Pasos correspondientes a la parse "sólo si" del Teorema 6

	Proposición	Justificación
1.	x pertenece a $(R \cup S) \cap (R \cup T)$	Postulado
2.	x pertenece a $R \cup S$	(1) y la definición de intersección
3.	x pertenece a $R \cup T$	(1) y la definición de intersección
4.	x pertenece a R o x pertenece a S y T	(2), (3), y el razonamiento sobre la unión
5.	x pertenece a R o x pertenece a $S \cap T$	(4) y la definición de intersección
6.	x pertenece a $R \cup (S \cap T)$	(5) y la definición de unión

Toda proposición si-entonces tiene una forma equivalente que, en algunas circunstancias, es más fácil de demostrar. La conversión contradictoria de la proposición "si H entonces C" es "si no C entonces no H". Una proposición y su contradictoria son ambas verdaderas o ambas falsas, por lo que podemos demostrar una u otra.

Para ver por qué "si H entonces C" y "si no C entonces no H" son lógicamente equivalente, en primer lugar, observamos que hay que considerar cuatro casos:

1. $H = \textit{verdadero}$, $C = \textit{verdadero}$.
2. $H = \textit{verdadero}$, $C = \textit{falso}$.
3. $H = \textit{falso}$, $C = \textit{verdadero}$.
4. $H = \textit{falso}$, $C = \textit{falso}$.

Solo existe una manera de hacer que una proposición si-entonces sea falsa: la hipótesis tiene que ser verdadera y la conclusión falsa (inciso 2). En los otros tres casos, la proposición si-entonces es verdadera.

Consideremos ahora en qué casos la conversión contradictoria "si no C entonces no H" es falsa. Para que la esta proposición sea falsa, su hipótesis (que es "no C") tiene que ser verdadera y su conclusión (que es "no H") tiene que ser falta. Pero "no C" es verdadera cuando C es falsa y "no H" es falsa justamente cuando H es verdadera. Estas dos condiciones corresponden al inciso 2, lo que demuestra que en cada uno de los cuatros casos, la proposición original y su conversión contradictoria son ambas verdaderas o falsas; es decir, son lógicamente equivalentes.

Demostración por reducción al absurdo

Otra forma de demostrar una proposición de la forma "si H entonces C" consiste en demostrar la proposición: "H y no C implica falsedad". Es decir, comenzamos suponiendo que tanto la hipótesis H como la negación de la conclusión C son verdaderas. La demostración se completa probando que algo que se sabe que es falso se deduce lógicamente a partir de H y C. Esta forma de demostración se conoce como demostración por reducción al absurdo.

Recuerda la forma en que demostramos el Teorema 3: Sea S un subconjunto finitos de un determinaod conjunto infinito U. Sea T el conjunto complementario de S con respecto a U. Entonces T es infinito.

Una de las pruebas más importantes y reconocidas que utiliza la reducción al absurdo es "The Halting Problem - Prueba de que las computadoras no pueden hacer todo (El Problema de la Parada)"

En grupos de tres personas, discute las siguientes preguntas:

- ¿Porqué crees que es importante esta demostración?
- ¿Qué implicaciones tiene?

Revisemos el siguiente vídeo. En él, encontraremos respuestas a las preguntas antes planteadas: “The Halting Problem - An Impossible Problem to Solve”

En la práctica, no se habla de demostrar un teorema, sino que tenemos que enfrentarnos a algo que parece que es cierto, por ejemplo, una estrategia para implementar un programa y tenemos que decidir si el "teorema" es o no verdadero. Para resolver este problema, podemos intentar demostrar el teorema, y si no es posible, intentar demostrar que la proposición es falsa.

Generalmente, los teoremas son proposiciones que incluyen un número infinito de casos, quizás todos los valores de sus parámetros.

Suele ser más fácil demostrar que una proposición no es un teorema que demostrar que sí lo es.

1. Demostrar, Supuesto Teorema 1. Si un entero x es un número primo, entonces x es impar.
2. Demostrar, Supuesto Teorema 2. No existe ninguna pareja de enteros a y b tal que $a \mod b = b \mod a$.

Existe una forma especial de demostración, denominada "inductiva", que es esencial a la hora de tratar con objetos definidos de forma recursiva. Muchas de las demostraciones inductivas más habituales trabajan con enteros, pero en la teoría de autómatas, también necesitamos demostraciones inductivas, por ejemplo, para conceptos definidos recursivamente como pueden ser árboles y expresiones de diversas clases, como expresiones regulares.

Supón que tenemos que demostrar una proposición $S(n)$ acerca de un número entero n. Un enfoque que se emplea habitualmente consiste en demostrar dos cosas:

1. El *caso base*, donde demostramos $S(i)$ para un determinado entero i. Normalmente, $i = 0$ o $i = 1$, pero habrá ejemplos en los que desearemos comenzar en cualquier valor mayor de i, quizá porque la proposición S sea falsa para los enteros más pequeños.
2. El *paso de inducción*, donde suponemos que $n \geq i$, siendo i el entero empleado en el caso base, y demostramos que "si $S(n)$ entonces $S(n+1)$".

Intuitivamente, estas dos partes deberían convencernos de que $S(n)$ es verdadera para todo entero n que sea igual o mayor que el entero de partida i.

Para algunos de los ejemplos que vamos a desarrollar, nos basaremos en alguno de los siguientes teoremas:

- Si $A \leq B$, entonces $A + C \leq B + C$.
- Si $A \leq B$ y $B \leq C$, entonces $A \leq C$.

Demostrar, para todo $n \geq 4$:

$$n^2 \leq 2^n$$

Demostrar, para todo $n \geq 3$:

$$2n + 1 \leq 2^n$$

Demostrar, para todo $n \geq 1$:

$$\sum_1^n i = \frac{n(n+1)}{2}$$

Supón una sucesión de números $a_1, a_2, \ldots$ que cumplen las siguientes reglas:

1. $a_1 = 1$
2. $a_{n+1} = 2a_n + 1$ para toda $n \geq 1$.

Demostrar, para todo $n \geq 1$:

$$a_n = 2^n - 1$$

¿Qué es un lenguaje?

Alfabeto

Un alfabeto es un conjunto finito, no vacío de símbolos. Por convención, usaremos el símbolo Σ. Entre los ejemplos más comunes de alfabetos podemos encontrar:

1. $\Sigma = 0, 1$, alfabeto binario.
2. $\Sigma = a, b, c, ..., z$, el conjunto de todas las letras minúsculas.
3. El conjunto de todos los caracteres ASCII, o el conjunto de todos los caracteres imprimibles ASCII.

Cadenas (strings)

- Una cadena (o algunas veces palabras) es una secuencia finita de símbolos elegidos de algún alfabeto. Por ejemplo, 01101 es una cadena del alfabeto binarios $\Sigma = 0, 1$.
- La cadena vacía, ε, es un cadena con cero ocurrencia de símbolos y, por lo mismo, puede ser elegido de cualquier alfabeto.
- La longitud de una cadena es el número de posiciones para símbolos dentro de la misma. Por ejemplo, 01101 tiene una longitud de 5. La notación estándar para la longitud de una cadena w es $|w|$. Por ejemplo, $|011| = 3$ y $|\varepsilon| = 0|$.

Potencias de un alfabeto

- Si Σ es un alfabeto, podemos expresar el conjunto de todas las cadenas de una cierta longitud usando la notación exponencial. Con esto, podemos definir Σ^k como el conjunto de cadenas de longitud k, seleccionados de los símbolos de Σ.
- Es importante hacer notar que $\Sigma^0 = \{\varepsilon\}$, sin importar a que alfabeto nos refiramos.
- De tal forma que si $\Sigma = \{0, 1\}$, entonces:
  - $\Sigma^1 = \{0, 1\}$,
  - $\Sigma^2 = \{00, 01, 10, 11\}$,
  - $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

- El conjunto de todos las cadenas (de cualquier longitud) que existen sobre un alfabeto Σ, convencionalmente, se denomina Σ^*. Por ejemplo, si $\Sigma = \{0, 1\}$, $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, ...\}$. Este conjunto, de hecho, resulta ser: $\Sigma^* = \Sigma^0 \cap \Sigma^1 \cap \Sigma^2 ...$
- Algunas veces, quizás queremos excluir la cadena vacía del conjunto de cadenas. Para ello, usaremos Σ^+. Lo cual es igual a $\Sigma^+ = \Sigma^1 \cap \Sigma^2 \cap \Sigma^3 ...$ o $\Sigma^* = \Sigma^+ \cup \varepsilon$.

Concatenación de cadenas

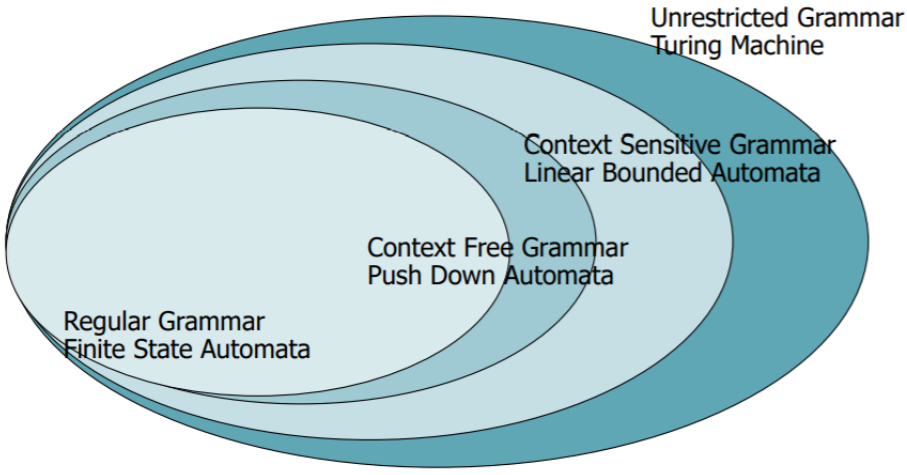
- Sean x y y cadenas. Entonces, xy indica la concatenación de x y y. De forma más precisa, si x es una cadena compuesta por i símbolos $x = a_1 a_2 ... a_i$ y y es una cadena compuesta por j símbolos $y = b_1 b_2 ... b_j$, entonces xy es una cadena de longitud $i + j$, $xy = a_1 a_2 ... a_i b_1 b_2 ... b_j$.
- Para cualquier cadena w, la ecuación $\varepsilon w = w\varepsilon = w$ es verdad. Esto es porque ε es identidad para concatenación.

- Un conjunto de cadenas, todas la cuales son seleccionadas de algún Σ^*, donde Σ es un alfabeto particular, es llamada lenguaje. Dicho de otra forma, si Σ es un alfabeto, y $L \subseteq \Sigma^*$, entonces L es un lenguaje sobre Σ. Por ejemplo,
  - El lenguaje Español, la colección legal de palabras españolas, es un conjunto de cadenas sobre el alfabeto que consiste de todas las letras.
  - El lenguaje C, o cualquier otro lenguaje de programación, es el conjunto de programas válidos que son un subconjunto de todas las posibles cadenas que pueden ser formadas sobre el conjunto de los caracteres ASCII.
- También podemos definir otros lenguajes:
  - El lenguaje de todas las cadenas consistentes de n 0's seguidos de n 1's para alguna $n \geq 0$: $\{\varepsilon, 01, 0011, 000111, ...\}$.

- También podemos definir otros lenguajes:
  - El lenguaje de todas las cadenas consistentes de n 0's seguidos de n 1's para alguna $n \geq 0$: $\{\varepsilon, 01, 0011, 000111, ...\}$.
  - El lenguaje de todas las cadenas que tiene un igual número de 0's y 1's: $\{\varepsilon, 01, 10, 0011, 1100, 0101, 1010, 1001, ...\}$.
  - El conjunto de números binarios cuyo valor es primo: $\{10, 11, 101, 111, 1011, ...\}$.
  - Σ^* es un lenguaje para cualquier alfabeto Σ.
  - $\emptyset$, el lenguaje vacío, es un lenguaje sobre cualquier alfabeto.
  - $\{\varepsilon\}$, lenguaje conformado por solo una cadena vacía, es, también, un lenguaje sobre cualquier alfabeto. Importante, $\emptyset \neq \{\varepsilon\}$.

Jerarquía de Chomsky-Schützenberger

- La jerarquía de Chomsky-Schützenberger es la base formal para describir un lenguaje (natural o artificial). “How Complex is Natural Language? The Chomsky Hierarchy”.


Unrestricted Grammar
Turing Machine
Context Sensitive Grammar
Linear Bounded Automata
Context Free Grammar
Push Down Automata
Regular Grammar
Finite State Automata

Definición de gramática

Una grámatica formal es un cuadrupla $G = (N, \Sigma, P, S)$ donde

- N es un conjunto finito de No Terminales.
- Σ es un conjunto finito de terminales y es disjunto de N.
- P es un conjunto finito de reglas de producción de la forma $w \epsilon (N \cup \Sigma)* \rightarrow w \epsilon (N \cup \Sigma)*$.
- $S \epsilon N$ es el símbolo inicial.

Tipo-0, Sin restricciones

- Los lenguajes definidos por gramáticas Tipo-0 son aceptadas por Maquinas de Turing.
- Las reglas son de la forma: $\alpha \rightarrow \beta$, donde α y β son cadenas arbitrarias sobre N y $\alpha \neq \varepsilon$.
- Ejemplos de producciones: Sab $\rightarrow$ ba, A $\rightarrow$ S.

Tipo-1, Sensibles al contexto

- Los lenguajes definidos por gramáticas Tipo-1 son aceptadas por Autómatas Delimitados Linealmente.
- Sintaxis de algunos lenguajes naturales (Alemán).
- Las reglas son de la forma: $\alpha A \beta \rightarrow \alpha B \beta$, $S \rightarrow \varepsilon$ donde A y $S \epsilon N$, α, β, $B \epsilon (N \cup \Sigma)*$ y $B \neq \varepsilon$.

- Los lenguajes definidos por gramáticas Tipo-2 son aceptadas por Autómatas Push-Down.
- El lenguaje natural es casi enteramente definible por árboles sintácticos de Tipo-2.
- Las reglas son de la forma: $A \rightarrow \alpha$ donde $A \epsilon N$, $\alpha \epsilon (N \cup \Sigma)*$.

- Los lenguajes definidos por gramáticas Tipo-3 son aceptadas por Autómatas de Estados Finitos.
- La mayoría de la sintaxis de dialogo hablado informal.
- Las reglas son de la forma: $A \rightarrow \alpha$, $A \rightarrow \varepsilon$, $A \rightarrow \alpha B$ donde A, $B \epsilon N$, $\alpha \epsilon \Sigma$.

Si quieres conocer un poco más, revisa estos dos vídeos:

- "TYPES OF GRAMMAR- Type 0, Type 1, Type 2, Type 3 (CHOMSKY HIERARCHY)|| THEORY OF COMPUTATION".
- "Noam Chomsky, Fundamental Issues in Linguistics (April 2019 at MIT) - Lecture 1".

Antes de comenzar a discutir conceptos de los lenguajes de programación, debemos considerar algunos preliminares.

1. Analizaremos algunas razones por las que es importante estudiar conceptos generales de diseño y evaluación de lenguajes.
2. Describiremos brevemente los principales dominios de programación. A continuación, presentamos una lista de criterios que pueden servir como base para tales juicios.
3. Discutiremos las dos principales influencias en el diseño de lenguajes: la arquitectura de la máquina y las metodologías de diseño de programas.
4. Por último, presentaremos las diversas categorías de lenguajes de programación. Describiendo algunas de las principales ventajas y desventajas que deben tenerse en cuenta durante el diseño del lenguaje.

Razones para estudiar conceptos de lenguajes de programación

- De manera individual, revisa el documento: "A History of Computer Programming Languages" de Andrew Ferguson.
- En grupos pequeños, de tres personas, comenten:
  - ¿Porqué crees que debemos aprender varios lenguajes de programación?

- **Mayor capacidad para expresar ideas.** Se cree ampliamente que la profundidad a la que las personas pueden pensar está influenciada por el poder expresivo del lenguaje en el que comunican sus pensamientos. El estudio de los conceptos del lenguaje de programación genera una apreciación de las características y construcciones valiosas del lenguaje y alienta a los programadores a usarlas, incluso cuando el lenguaje que están usando no admite directamente tales características y construcciones.
- **Transfondo mejorado para elegir los lenguajes apropiados.** Un mejor entendiemiento del funcionamiento de los lenguajes, es posible elegir mejor el lenguaje con las características que mejor solucionan el problema.

- **Mayor capacidad para aprender nuevos lenguajes.** La programación es todavía una disciplina relativamente joven, y las metodologías de diseño, las herramientas de desarrollo de software y los lenguajes de programación aún se encuentran en un estado de evolución continua. Esto hace que el desarrollo de software sea una profesión emocionante, pero también significa que el aprendizaje continuo es esencial. El proceso de aprender un nuevo lenguaje de programación puede ser largo y difícil, especialmente para alguien que se siente cómodo con solo uno o dos lenguajes y nunca ha examinado los conceptos del lenguaje de programación en general. Una vez que se adquiere una comprensión profunda de los conceptos fundamentales de los lenguajes, se vuelve mucho más fácil ver cómo estos conceptos se incorporan al diseño del idioma que se está aprendiendo.

- **Mejor comprensión de la importancia de la implementación.** Al aprender los conceptos de los lenguajes de programación, es interesante y necesario tocar los problemas de implementación que afectan esos conceptos. En algunos casos, la comprensión de los problemas de implementación conduce a la comprensión de por qué los lenguajes están diseñados de la forma en que lo están. A su vez, este conocimiento conduce a la capacidad de usar un lenguaje de manera más inteligente. Podemos convertirnos en mejores programadores al comprender las opciones entre las construcciones del lenguaje de programación y las consecuencias de esas opciones. Otro beneficio de comprender los problemas de implementación es que nos permite visualizar cómo una computadora ejecuta varias construcciones de lenguaje.

- **Mejor uso de lenguajes que ya se conocen.** Muchos lenguajes de programación contemporáneos son grandes y complejos. En consecuencia, es poco común que un programador esté familiarizado y utilice todas las funciones de un lenguaje que utiliza. Al estudiar los conceptos de los lenguajes de programación, los programadores pueden aprender sobre partes previamente desconocidas y no utilizadas de los lenguajes que ya usan y comenzar a usar esas funciones.

- **Avance general de la computación.** Finalmente, existe una visión global de la computación que puede justificar el estudio de los conceptos del lenguaje de programación. Aunque generalmente es posible determinar por qué un lenguaje de programación en particular se volvió popular, muchos creen, al menos en retrospectiva, que los lenguajes más populares no siempre son los mejores disponibles.

Dominios del área de programación

Las computadoras se han aplicado a una miríada de áreas diferentes, desde controlar plantas de energía nuclear hasta proporcionar videojuegos en teléfonos móviles. Debido a esta gran diversidad en el uso de las computadoras, se han desarrollado lenguajes de programación con objetivos muy diferentes. En esta sección, analizamos brevemente algunas de las áreas de las aplicaciones de software y sus lenguajes asociados.

- **Aplicaciones científicas.** Las primeras computadoras digitales, que aparecieron a fines de la década de 1940 y principios de la de 1950, se inventaron y utilizaron para aplicaciones científicas. Por lo general, las aplicaciones científicas de esa época usaban estructuras de datos relativamente simples, pero requerían una gran cantidad de cálculos aritméticos de punto flotante. Las estructuras de datos más comunes fueron arreglos y matrices; las estructuras de control más comunes eran bucles de conteo y selecciones. Los primeros lenguajes de programación de alto nivel inventados para aplicaciones científicas fueron diseñados para satisfacer esas necesidades. El primer lenguaje para aplicaciones científicas fue Fortran. ALGOL 60 y la mayoría de sus descendientes también estaban destinados a usarse en esta área, aunque también fueron diseñados para usarse en áreas relacionadas.

- **Aplicaciones de negocios.** El uso de computadoras para aplicaciones comerciales comenzó en la década de 1950. Se desarrollaron computadoras especiales para este propósito, junto con lenguajes especiales. El primer lenguaje de alto nivel para negocios exitoso fue COBOL (ISO/IEC, 2002), cuya versión inicial apareció en 1960. Sigue siendo el lenguaje más utilizado para estas aplicaciones. Los lenguajes comerciales se caracterizan por la facilidad para producir informes elaborados, formas precisas de describir y almacenar números decimales y datos de caracteres, y la capacidad de especificar operaciones aritméticas decimales.

- **Inteligencia Artificial.** La inteligencia artificial (IA) es un área amplia de aplicaciones de software caracterizada por el uso de cálculos simbólicos en lugar de numéricos. La computación simbólica significa que se manipulan los símbolos, que consisten en nombres en lugar de números. Además, el cálculo simbólico se realiza más convenientemente con listas enlazadas de datos en lugar de matrices. Este tipo de programación a veces requiere más flexibilidad que otros dominios de programación. Por ejemplo, en algunas aplicaciones de IA es conveniente la capacidad de crear y ejecutar segmentos de código durante la ejecución. El primer lenguaje de programación ampliamente utilizado desarrollado para aplicaciones de IA fue el lenguaje funcional LISP (McCarthy et al., 1965), que apareció en 1959. in embargo, a principios de la década de 1970 apareció un enfoque alternativo para algunas de estas aplicaciones: la programación lógica utilizando el lenguaje Prolog (Clocksin y Mellish, 2003).

- **Desarrollo de herramientas.** El sistema operativo y las herramientas de soporte de programación se conocen colectivamente como software de sistemas. El software de sistemas se usa casi continuamente, por lo que debe ser eficiente. Además, debe tener características de bajo nivel que permitan escribir las interfaces de software a dispositivos externos. El sistema operativo UNIX está escrito casi en su totalidad en C (ISO, 1999), lo que ha hecho que sea relativamente fácil de portar o mover a diferentes máquinas.

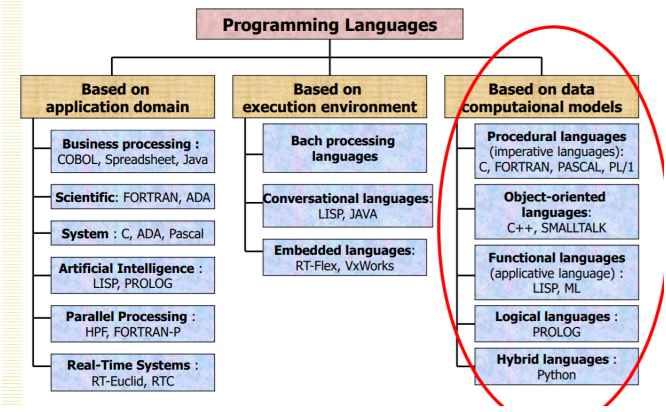
- **Desarrollo Web.** La World Wide Web está respaldada por una colección ecléctica de lenguajes, que van desde lenguajes de marcado, como HTML, que no es un lenguaje de programación, hasta lenguajes de programación de propósito general, como Java.

- **Legibilidad.** Uno de los criterios más importantes para juzgar un lenguaje de programación es la facilidad con la que se pueden leer y comprender los programas. Antes de 1970, la principal característica positiva de los lenguajes de programación fue la eficiencia. Las construcciones del lenguaje se diseñaron más desde el punto de vista de la computadora que de los usuarios de la computadora. La legibilidad debe considerarse en el contexto del dominio del problema. Por ejemplo, si un programa que describe un cálculo está escrito en un lenguaje que no está diseñado para ese uso, el programa puede resultar poco natural y complicado, se dificultará su lectura.

- **Ortogonalidad.** La ortogonalidad en un lenguaje de programación significa que un conjunto relativamente pequeño de construcciones primitivas se puede combinar en un número relativamente pequeño de formas para construir las estructuras de control y datos del lenguaje. Además, toda combinación posible de primitivas es legal y significativa. Por ejemplo, considera los tipos de datos. Supongamos que un lenguaje tiene cuatro tipos de datos primitivos (entero, flotante, doble y carácter) y dos tipos de operadores (arreglo y puntero). Si los dos operadores de tipo se pueden aplicar a sí mismos y a los cuatro tipos de datos primitivos, se puede definir una gran cantidad de estructuras de datos. La ortogonalidad está estrechamente relacionada con la simplicidad: cuanto más ortogonal es el diseño de un lenguaje, menos excepciones requieren las reglas del lenguaje. Menos excepciones significan un mayor grado de regularidad en el diseño, lo que hace que el idioma sea más fácil de aprender, leer y comprender.

- **Fácilidad de escritura.** La capacidad de escritura es una medida de la facilidad con la que se puede usar un lenguaje para crear programas para un dominio de problema elegido.
- **Fiabilidad.** Se dice que un programa es confiable si se desempeña de acuerdo con sus especificaciones en todas las condiciones.
  - Revisión de tipos.
  - Manejo de excepciones.

Modelos de lenguajes de programación



- Lenguajes imperativos
  - Un programa consiste de una secuencia de enunciados.
  - La ejecución de cada declaración hace que el intérprete cambie el estado del sistema.
  - Los sucesivos estados de la máquina conducen a la solución deseada.
  - Ejemplos: C, FORTRAN, Algol, PL/I, Pascal.
- Lenguajes basados en objetos
  - Los objetos complejos están diseñados como extensiones de objetivos simples.
  - Herencia de objetos simples.
  - Ejemplos: C++, Java, Smalltalk.

- Lenguajes aplicativos (lenguajes funcionales)
  - Realizar una función con sus argumentos para generar resultados.
  - Los resultados de una función pueden ser argumentos de otra función.
  - Ejemplos: LISP, ML, Scheme, Erlang y Clojure.
- Lenguajes basados en reglas (lenguajes lógicos)
  - Ejecutar una acción cuando se cumple una determinada condición de habilitación (predicado).
  - Construir una matriz o tabla de posibles condiciones y acciones apropiadas para pruebas o búsqueda en bases de datos.
  - El programa consta de hechos, reglas y consultas.
  - Ejemplos: Prolog
- Lenguajes híbridos: Python.

Bibliografía

- Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2014). Introduction to Automata Theory, Languages, and Computation: Pearson New International Edition: Vol. 3rd ed. Pearson.