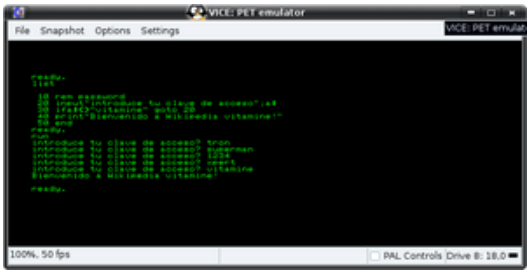
WIKIPEDIA
La enciclopedia libre

Lenguaje de programación

Un **lenguaje de programación** es un lenguaje formal (o artificial, es decir, un lenguaje con reglas gramaticales bien definidas) que proporciona a una persona, en este caso el programador, la capacidad y habilidad de escribir (o programar) una serie de instrucciones o secuencias de órdenes en forma de algoritmos con el fin de controlar el comportamiento físico o lógico de un sistema informático, para que de esa manera se puedan obtener diversas clases de datos o ejecutar determinadas tareas. A todo este conjunto de órdenes escritas mediante un lenguaje de programación se le denomina programa informático.[1] [2] [3] [4]



Captura de la microcomputadora Commodore PET-32 mostrando un programa en el lenguaje de programación de alto nivel BASIC, bajo el emulador VICE, en una distribución GNU/Linux.

Características

Programar viene a ser el proceso de crear un software fiable mediante la escritura, prueba, depuración, compilación o interpretación, y mantenimiento del código fuente de dicho programa informático. Básicamente, este proceso se define aplicando lógicamente los siguientes pasos:

- El desarrollo lógico del programa para resolver un problema en particular.
- Escritura de la lógica del programa empleando un lenguaje de programación específico (codificación del programa).
- Compilación o interpretación del programa hasta convertirlo en lenguaje de máquina.
- Prueba y depuración del programa.
- Desarrollo de la documentación.

```java
// class declaration
public class ProgrammingExample {

    // method declaration
    public void sayHello() {

        // method output
        System.out.println("Hello World!");
    }
}
```

Un ejemplo de código fuente escrito en el lenguaje de programación Java, que imprimirá el mensaje "Hello World!" a la salida estándar cuando es compilado y ejecutado

Los lenguajes de programación están formados por un conjunto de símbolos (llamado alfabeto), reglas gramaticales (léxico/morfológicas y sintácticas) y semánticas, que en conjunto definen las estructuras válidas del lenguaje y su significado. Existe el error común de tratar como sinónimos los términos 'lenguaje de programación' y 'lenguaje informático'. Los lenguajes informáticos engloban a los lenguajes de programación y a otros más, como por ejemplo HTML (lenguaje para el marcado de páginas web, que no es propiamente un lenguaje de programación, sino un conjunto de instrucciones que permiten estructurar el contenido de los documentos).

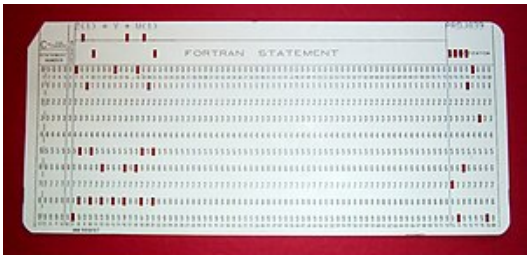
El lenguaje de programación permite especificar de *manera precisa* sobre qué datos debe operar un software específico, cómo deben ser almacenados o transmitidos dichos datos, y qué acciones debe tomar el software bajo una variada gama de circunstancias. Todo esto, a través de un lenguaje que intenta estar *relativamente* próximo al lenguaje humano o natural. Una característica relevante de los lenguajes de programación es precisamente que más de un programador pueda usar un conjunto común de instrucciones que sean comprendidas entre ellos para realizar la construcción de un programa de forma colaborativa.

Historia

Para que la computadora entienda nuestras instrucciones debe usarse un lenguaje específico conocido como código máquina, que la máquina lee fácilmente, pero que es excesivamente complicado para las personas. De hecho, solo consiste en cadenas extensas de números 0 y 1(números binarios).

Para facilitar el trabajo, los primeros operadores de computadoras decidieron crear un traductor para reemplazar los 0 y 1 por palabras o abstracción de palabras y letras provenientes del inglés; este se conoce como lenguaje ensamblador. Por ejemplo, para sumar se usa la letra A de la palabra inglesa *add* (sumar). El lenguaje ensamblador sigue la misma estructura del lenguaje máquina, pero las letras y palabras son más fáciles de recordar y entender que los números.

La necesidad de recordar secuencias de programación para las acciones usuales llevó a denominarlas con nombres fáciles de memorizar y asociar: `ADD` (sumar), `SUB` (restar), `MUL` (multiplicar), `CALL` (ejecutar subrutina), etc. A esta secuencia de posiciones se le denominó "instrucciones", y a este conjunto de instrucciones se le llamó lenguaje ensamblador. Posteriormente aparecieron diferentes lenguajes de programación, los cuales reciben su denominación porque tienen una estructura sintáctica semejante a la de los lenguajes escritos por los humanos, denominados también lenguajes de alto nivel.[5]



Código Fortran en una tarjeta perforada, mostrando el uso especializado de las columnas 1-5, 6 y 73-80

A finales de 1953, John Backus sometió una propuesta a sus superiores en IBM para desarrollar una alternativa más práctica al lenguaje ensamblador, para programar la computadora central IBM 704. El histórico equipo Fortran de John Backus consistió en los programadores Richard Goldberg, Sheldon F. Best, Harlan Herrick, Peter Sheridan, Roy Nutt, Robert Nelson, Irving Ziller, Lois Haibt y David Sayre.[6]

El primer manual para el lenguaje Fortran apareció en octubre de 1956, con el primer compilador Fortran entregado en abril de 1957. Esto era un compilador optimizado, porque los clientes eran reacios a usar un lenguaje de alto nivel a menos que su compilador pudiera generar código cuyo desempeño fuera comparable al de un código hecho a mano en lenguaje ensamblador.

En 1960 se creó COBOL, uno de los lenguajes usados aún en la actualidad, en informática de gestión.

A medida que la complejidad de las tareas que realizaban las computadoras aumentaba, se hizo necesario disponer de un método más eficiente para programarlas. Entonces se crearon los lenguajes de alto nivel, como lo fue BASIC en las versiones introducidas en los microordenadores de la década de 1980. Mientras que una tarea tan sencilla como sumar dos números puede necesitar varias instrucciones en lenguaje ensamblador, en un lenguaje de alto nivel bastará una sola sentencia.

Clasificación de los lenguajes de programación

Los lenguajes de programación han sido históricamente clasificados atendiendo a distintos criterios:

- **Clasificación histórica**

  A medida que surgían nuevos lenguajes que permitían nuevos estilos de programación más expresiva, se distinguieron dichos estilos en una serie de **generaciones**, cada una representando lenguajes de programación surgidos en una época similar y con características genéricas comunes.

- **Lenguajes de alto y de bajo nivel**

  Los lenguajes de programación se suelen clasificar dentro de dos amplias categorías que se refieren a su "nivel de abstracción", es decir, en cuanto a lo específico o general que es respecto a la arquitectura de computación inherente al sistema que se está utilizando.

- **Clasificación por paradigmas**

  Los paradigmas de programación distinguen distintos modelos de cómputo y de estilos de estructurar y organizar las tareas que debe realizar un programa. Un lenguaje de programación puede ofrecer soporte a uno o varios paradigmas de programación, total o parcialmente.

- **Clasificación por propósito**

  Se distinguen los lenguajes de programación de **propósito general** de aquellos de **propósito específico**.

En algunas ocasiones los lenguajes de programación son también clasificados en **familias** que comparten ciertas características comunes como el estilo general de la sintaxis que emplean. Habitualmente estas características suelen ser heredadas de lenguajes de programación anteriores que sirvieron de inspiración a los creadores de dicho lenguaje.

Clasificación histórica o por generaciones

Los equipos de ordenador (el hardware) han pasado por cuatro generaciones, de las que las tres primeras (ordenadores con válvulas, transistores y circuitos integrados) están muy claras; la cuarta (circuitos integrados a gran escala) es más discutible.

Algo parecido ha ocurrido con la programación de los ordenadores (el software), que se realiza en lenguajes que suelen clasificarse en cinco generaciones, de las que las tres primeras son evidentes, mientras no todo el mundo está de acuerdo en las otras dos. Estas generaciones no coincidieron exactamente en el tiempo con las de hardware, pero sí de forma aproximada, y son las siguientes:

- **Primera generación**: los primeros ordenadores se programaban directamente en código de máquina (basado en sistema binario), que puede representarse mediante secuencias de 0 y 1. No obstante, cada modelo de ordenador tiene su propia estructura interna a la hora de programarse. A estos lenguajes se les denominaba Lenguajes de bajo nivel, porque sus instrucciones ejercen un control directo sobre el hardware y están condicionados por la estructura física de las computadoras que lo soportan. Dado que este tipo de lenguaje se acerca mucho más a la lógica de la máquina que a la humana, es mucho más complicado programar con él. El uso de la palabra *bajo* en su denominación no implica que el lenguaje sea menos potente que un lenguaje de alto nivel, sino que se refiere a la reducida abstracción entre el lenguaje y el hardware. Por ejemplo, se utiliza este tipo de lenguajes para programar tareas críticas de los sistemas operativos, de aplicaciones en tiempo real o controladores de dispositivos. Otra limitación de estos lenguajes es que se requiere de ciertos conocimientos de programación para realizar las secuencias de instrucciones lógicas.
- **Segunda generación**: los lenguajes simbólicos, asimismo propios de la máquina, simplifican la escritura de las instrucciones y las hacen más legibles. Se refiere al lenguaje ensamblador ensamblado a través de un macroensamblador. Es el lenguaje de máquina combinado con una serie de poderosas macros que permiten declarar estructuras de datos y de control complejas.
- **Tercera generación**: los lenguajes de alto nivel sustituyen las instrucciones simbólicas por códigos independientes de la máquina, parecidas al lenguaje humano o al de las Matemáticas. Se crearon para que el usuario común pudiese solucionar un problema de procesamiento de datos de una manera más fácil y rápida. Son usados en ámbitos computacionales donde se logra un alto rendimiento con respecto a lenguajes de generaciones anteriores. Entre ellos se encuentran C, Fortran, Smalltalk, Ada, C++, C#, Cobol, Delphi, Java y PHP, entre otros. Algunos de estos lenguajes pueden ser de propósito general, es decir, que el lenguaje no está enfocado a una única especialidad, sino que puede usarse para crear todo tipo de programas. Para ciertas tareas más comunes, existen bibliotecas para facilitar la programación que permiten la reutilización de código.
- **Cuarta generación**: se ha dado este nombre a ciertas herramientas que permiten construir aplicaciones sencillas combinando piezas prefabricadas. Hoy se piensa que estas herramientas no son, propiamente hablando, lenguajes. Cabe mencionar que, algunos proponen reservar el nombre de cuarta generación para la programación orientada a objetos. Estos últimos tienen una estructura muy parecida al idioma inglés. Algunas de sus características son: acceso a base de datos, capacidades gráficas, generación de código automáticamente, así como poder programar visualmente (como por ejemplo Visual Basic o SQL). Entre sus ventajas se cuenta una mayor productividad y menor agotamiento del programador, así como menor concentración por su parte, ya que las herramientas proporcionadas incluyen secuencias de instrucciones. El nivel de concentración que se requiere es menor, ya que algunas instrucciones, que le son dadas a las herramientas, a su vez, engloban secuencias de instrucciones a otro nivel dentro de la herramienta. Cuando hay que dar mantenimiento a los programas previamente elaborados, es menos complicado por requerir menor nivel de concentración. Por otro lado, sus desventajas consisten en que estas herramientas prefabricadas son generalmente menos flexibles que las instrucciones directas en los lenguajes de bajo nivel. Además, se suelen crear dependencias con uno o varios proveedores externos, lo que se traduce en pérdida de autonomía. Asimismo, es frecuente que dichas herramientas prefabricadas contengan librerías de otros proveedores, que conlleva instalar opciones adicionales que son consideradas opcionales. A menos que existan acuerdos con otros proveedores, son programas que se ejecutan únicamente con el lenguaje que lo creó. Tampoco suelen cumplir con los estándares internacionales ISO y ANSI, lo cual conlleva un riesgo futuro por desconocerse su tiempo de permanencia en el mercado. Algunos ejemplos son: NATURAL y PL/SQL.
- **Quinta generación**: La quinta generación de lenguajes de programación [(5GL)] es un término que se refiere a un conjunto de lenguajes de programación de alto nivel que se centran en la resolución de problemas utilizando inteligencia artificial y técnicas de programación declarativa. Estos lenguajes de programación utilizan paradigmas de programación no convencionales para ayudar a los desarrolladores a resolver problemas complejos.

Algunos ejemplos de lenguajes de programación de quinta generación son:

- **Mercury**: Un lenguaje de programación funcional basado en lógica que utiliza un enfoque declarativo para la programación.
- **OPS5**: Un lenguaje de programación basado en reglas que se utiliza en sistemas expertos.
- **Prolog**: Un lenguaje de programación lógico que se utiliza para la programación de inteligencia artificial y sistemas expertos.
- **Haskell**: Un lenguaje de programación funcional que se utiliza en la programación de inteligencia artificial y aprendizaje automático.
- **Lisp**: Un lenguaje de programación funcional que se utiliza en la programación de inteligencia artificial y sistemas expertos.

Paradigma de programación

Un paradigma de programación consiste en un método para llevar a cabo cómputos y la forma en la que deben estructurarse y organizarse las tareas que debe realizar un programa.[7] Se trata de una propuesta tecnológica adoptada por una comunidad de programadores, y desarrolladores cuyo núcleo central es incuestionable en cuanto que únicamente trata de resolver uno o varios problemas claramente delimitados; la resolución de estos problemas debe suponer consecuentemente un avance significativo en al menos un parámetro que afecte a la ingeniería de software. Representa un enfoque particular o filosofía para diseñar soluciones. Los paradigmas difieren unos de otros, en los conceptos y la forma de abstraer los elementos involucrados en un problema, así como en los pasos que integran su solución del problema, en otras palabras, el cómputo. Tiene una estrecha relación con la formalización de determinados lenguajes en su momento de definición. Es un estilo de programación empleado.

Un paradigma de programación está delimitado en el tiempo en cuanto a aceptación y uso, porque nuevos paradigmas aportan nuevas o mejores soluciones que lo sustituyen parcial o totalmente.

El paradigma de programación que actualmente es más utilizado es la "orientación a objetos" (OO). El núcleo central de este paradigma es la unión de datos y procesamiento en una entidad llamada "objeto", relacionable a su vez con otras entidades "objeto".

Tradicionalmente, datos y procesamiento se han separado en áreas diferente del diseño y la implementación de software. Esto provocó que grandes desarrollos tuvieran problemas de fiabilidad, mantenimiento, adaptación a los cambios y escalabilidad. Con la OO y características como el encapsulado, polimorfismo o la herencia, se permitió un avance significativo en el desarrollo de software a cualquier escala de producción. La OO parece estar ligada en sus orígenes con lenguajes como Lisp y Simula, aunque el primero que acuñó el título de "programación orientada a objetos" fue Smalltalk.

Clasificación por paradigmas

En general, la mayoría de paradigmas son variantes de los dos tipos principales de programación, imperativa y declarativa. En la programación imperativa se describe paso a paso un conjunto de instrucciones que deben ejecutarse para variar el estado del programa y hallar la solución, es decir, un algoritmo en el que se describen los pasos necesarios para solucionar el problema.

En la programación declarativa las sentencias que se utilizan lo que hacen es describir el problema que se quiere solucionar; se programa diciendo lo que se quiere resolver a nivel de usuario, pero no las instrucciones necesarias para solucionarlo. Esto último se realizará mediante mecanismos internos de inferencia de información a partir de la descripción realizada.

A continuación se describen algunas de las distintas variantes de paradigmas de programación:

- **Programación imperativa o por procedimientos**: es el más usado en general, se basa en dar instrucciones al ordenador de como hacer las cosas en forma de algoritmos, en lugar de describir el problema o la solución. Las recetas de cocina y las listas de revisión de procesos, a pesar de no ser programas de computadora, son también conceptos familiares similares en estilo a la programación imperativa; donde cada paso es una instrucción. Es la forma de programación más usada y la más antigua, el ejemplo principal es el lenguaje de máquina. Ejemplos de lenguajes puros de este paradigma serían el C, BASIC o Pascal.
- Programación orientada a objetos: está basada en el imperativo, pero encapsula elementos denominados objetos que incluyen tanto variables como funciones. Está representado por C# o Java entre otros, pero el más representativo sería el Smalltalk que está completamente orientado a objetos.

- Programación dirigida por eventos: la programación dirigida por eventos es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen.
- **Programación declarativa**: está basada en describir el problema declarando propiedades y reglas que deben cumplirse, en lugar de instrucciones. Hay lenguajes para la programación funcional, la programación lógica, o la combinación lógico-funcional. La solución es obtenida mediante mecanismos internos de control, sin especificar exactamente cómo encontrarla (tan solo se le indica a la computadora qué es lo que se desea obtener o qué es lo que se está buscando). No existen asignaciones destructivas, y las variables son utilizadas con transparencia referencial. Los lenguajes declarativos tienen la ventaja de ser razonados matemáticamente, lo que permite el uso de mecanismos matemáticos para optimizar el rendimiento de los programas.[8] Unos de los primeros lenguajes funcionales fueron Lisp y Prolog.
- Programación funcional: basada en la definición los predicados y es de corte más matemático, está representado por Scheme (una variante de Lisp) o Haskell. Python también representa este paradigma.[9]
- Programación lógica: basado en la definición de relaciones lógicas, está representado por Prolog.
- Programación con restricciones: similar a la lógica usando ecuaciones. Casi todos los lenguajes son variantes del Prolog.
- **Programación multiparadigma**: es el uso de dos o más paradigmas dentro de un programa. El lenguaje Lisp se considera multiparadigma. Al igual que Python o PHP que son orientados a objetos, reflexivos, imperativos y funcionales.[9] Según lo describe Bjarne Stroustrup, esos lenguajes permiten crear programas usando más de un estilo de programación. El objetivo en el diseño de estos lenguajes es permitir a los programadores utilizar el mejor paradigma para cada trabajo, admitiendo que ninguno resuelve todos los problemas de la forma más fácil y eficiente posible. Por ejemplo, lenguajes de programación como C++, Genie, Delphi, Visual Basic o D[10] combinan el paradigma imperativo con la orientación a objetos. Incluso existen lenguajes multiparadigma que permiten la mezcla de forma natural, como en el caso de Oz, que tiene subconjuntos (particularidad de los lenguajes lógicos), y otras características propias de lenguajes de programación funcional y de orientación a objetos. Otro ejemplo son los lenguajes como Scheme de paradigma funcional o Prolog (paradigma lógico), que cuentan con estructuras repetitivas, propias del paradigma imperativo.
- **Programación reactiva**: este paradigma se basa en la declaración de una serie de objetos emisores de eventos asíncronos y otra serie de objetos que se "suscriben" a los primeros (es decir, quedan a la escucha de la emisión de eventos de estos) y *reaccionan* a los valores que reciben. Es muy común usar la librería Rx de Microsoft (Acrónimo de Reactive Extensions), disponible para múltiples lenguajes de programación.
- Lenguaje específico del dominio o DSL: se denomina así a los lenguajes desarrollados para resolver un problema específico, pudiendo entrar dentro de cualquier grupo anterior. El más representativo sería SQL para el manejo de las bases de datos, de tipo declarativo, pero los hay imperativos, como el Logo.

Elementos

Variables y vectores

Las variables son títulos asignados a espacios en memoria para almacenar datos específicos. Son contenedores de datos y por ello se diferencian según el tipo de dato que son capaces de almacenar. En la mayoría de lenguajes de programación se requiere especificar un tipo de variable concreto para guardar un dato específico. Por ejemplo, en Java, si deseamos guardar una cadena de texto debemos especificar que la variable es del tipo *String*. Por otra parte, en lenguajes como PHP o JavaScript este tipo de especificación de variables no es necesario. Además, existen variables compuestas llamadas vectores. Un vector no es más que un conjunto de bytes consecutivas en memoria y del mismo tipo guardadas dentro de una variable contenedor. A continuación, un listado con los tipos de variables y vectores más comunes:

```
1  ' Globales ----------
2  Var Variable0:Booleano
3  Var Variable1:Cadena
4  ' Fin Globales -------
5  Proc Procedimiento ' <- Procedimiento sin retorno.
6     Var Variable2:Entero ' Locales
7     Var Variable3:Real
8
9     Si Variable0 = Falso Entonces ' Condición "If"
10       Contar Variable2 = 0 a 9 ' Bucle "For"
11          Variable1 = Variable1 + "1"
12       Seguir ' "End For"
13    FinSi ' "End If"
14
15    Variable3 = 5.13
16 FinProc
```

Imagen tomada de Pauscal, lenguaje de programación en español creado en Argentina

Tipo de dato	Breve descripción
Char	Estas variables contienen un único carácter, es decir, una letra, un signo o un número.
Int	Contienen un número entero.
Float	Contienen un número decimal.
String	Contienen cadenas de texto, o lo que es lo mismo, es un vector con varias variables del tipo Char.
Boolean	Solo pueden contener un cero o un uno.

En el caso de variables booleanas, el cero es considerado para muchos lenguajes como el literal falso ("**False**"), mientras que el uno se considera verdadero ("**True**").

Condicionales

Las sentencias condicionales son estructuras de código que indican que, para que cierta parte del programa se ejecute, deben cumplirse ciertas premisas; por ejemplo: que dos valores sean iguales, que un valor exista, que un valor sea mayor que otro... Estos condicionantes por lo general solo se ejecutan una vez a lo largo del programa. Los condicionantes más conocidos y empleados en programación son:

- **If**: Indica una condición para que se ejecute una parte del programa.
- **Else if**: Siempre va precedido de un "If" e indica una condición para que se ejecute una parte del programa siempre que no cumpla la condición del if previo y sí se cumpla con la que el "else if" especifique.
- **Else**: Siempre precedido de "If" y en ocasiones de "Else If". Indica que debe ejecutarse cuando no se cumplan las condiciones previas.

Bucles

Los bucles son parientes cercanos de los condicionantes, pero ejecutan constantemente un código mientras se cumpla una determinada condición. Los más frecuentes son:

- **For**: Ejecuta un código mientras una variable se encuentre entre 2 determinados parámetros.
- **While**: Ejecuta un código mientras que se cumpla la condición que solicita.

Hay que decir que a pesar de que existan distintos tipos de bucles, todos son capaces de realizar exactamente las mismas funciones. El empleo de uno u otro depende, por lo general, del gusto del programador.

Funciones

Las funciones se crearon para evitar tener que repetir constantemente fragmentos de código. Una función podría considerarse como una variable que encierra código dentro de si. Por lo tanto, cuando accedemos a dicha variable (la función) en realidad lo que estamos haciendo es ordenar al programa que ejecute un determinado código predefinido anteriormente.

Todos los lenguajes de programación tienen algunos elementos de formación primitivos para la descripción de los datos y de los procesos o transformaciones aplicadas a estos datos (tal como la suma de dos números o la selección de un elemento que forma parte de una colección). Estos elementos primitivos son definidos por reglas sintácticas y semánticas que describen su estructura y significado respectivamente.

Sintaxis

A la forma visible de un lenguaje de programación se la conoce como sintaxis. La mayoría de los lenguajes de programación son puramente textuales, es decir, utilizan secuencias de texto que incluyen palabras, números y puntuación, de manera similar a los lenguajes naturales escritos. Por otra parte, hay algunos lenguajes de programación que son más gráficos en su naturaleza, utilizando relaciones visuales entre símbolos para especificar un programa.

La sintaxis de un lenguaje de programación describe las combinaciones posibles de los símbolos que forman un programa sintácticamente correcto. El significado que se le da a una combinación de símbolos es manejado por su semántica (ya sea formal o como parte del código duro de la referencia de implementación). Dado que la mayoría

de los lenguajes son textuales, este artículo trata de la sintaxis textual.

La sintaxis de los lenguajes de programación es definida generalmente utilizando una combinación de expresiones regulares (para la estructura léxica/morfológica) y la Notación de Backus-Naur (para la estructura sintáctica). Este es un ejemplo de una gramática simple, tomada del lenguaje Lisp:

```
expresión ::= átomo | lista
átomo ::= número | símbolo
número ::= [+-]? ['0'-'9']+
símbolo ::= ['A'-'Z'] ['a'-'z'].*
lista ::= '(' expresión* ')'
```

Con esta gramática se especifica lo siguiente:

```
def add5(x):
   return x+5

def dotwrite(ast):
   nodename = getNodename()
   label=symbol.sym_name.get(int(ast[0]),ast[0])
   print '    %s [label="%s"' % (nodename, label),
   if isinstance(ast[1], str):
      if ast[1].strip():
         print '= %s"];' % ast[1]
      else:
         print '"]'
   else:
      print '"];'
      children = []
      for n, child in enumerate(ast[1:]):
         children.append(dotwrite(child))
      print '    %s -> {' % nodename,
      for name in children:
         print '%s' % name,
```

Con frecuencia se resaltan los elementos de la sintaxis con colores diferentes para facilitar su lectura. Este ejemplo está escrito en Python.

- una *expresión* puede ser un *átomo* o una *lista*;
- un *átomo* puede ser un *número* o un *símbolo*;
- un *número* es una secuencia continua de uno o más dígitos decimales, precedido opcionalmente por un signo más o un signo menos;
- un *símbolo* es una letra seguida de cero o más caracteres (excluyendo espacios); y
- una *lista* es un par de paréntesis que abren y cierran, con cero o más expresiones en medio.

Algunos ejemplos de secuencias bien formadas de acuerdo a esta gramática:

`'12345', '()', '(a b c232 (1))'`

No todos los programas sintácticamente correctos son semánticamente correctos. Muchos programas sintácticamente correctos tienen inconsistencias con las reglas del lenguaje; y pueden (dependiendo de la especificación del lenguaje y la solidez de la implementación) resultar en un error de traducción o ejecución. En algunos casos, tales programas pueden exhibir un comportamiento indefinido. Además, incluso cuando un programa está bien definido dentro de un lenguaje, todavía puede tener un significado que no es el que la persona que lo escribió estaba tratando de construir.

Usando el lenguaje natural, por ejemplo, puede no ser posible asignarle significado a una oración gramaticalmente válida o la oración puede ser falsa:

- "Las ideas verdes y descoloridas duermen furiosamente" es una oración bien formada gramaticalmente pero no tiene significado comúnmente aceptado.
- "Juan es un soltero casado" también está bien formada gramaticalmente pero expresa un significado que no puede ser verdadero.

El siguiente fragmento en el lenguaje C es sintácticamente correcto, pero ejecuta una operación que no está definida semánticamente (dado que `p` es un apuntador nulo, las operaciones `p->real` y `p->im` no tienen ningún significado):

```
complex *p = NULL;
complex abs_p = sqrt (p->real * p->real + p->im * p->im);
```

Si la declaración de tipo de la primera línea fuera omitida, el programa dispararía un error de compilación, pues la variable "p" no estaría definida. Pero el programa sería sintácticamente correcto todavía, dado que las declaraciones de tipo proveen información semántica solamente.

La gramática necesaria para especificar un lenguaje de programación puede ser clasificada por su posición en la Jerarquía de Chomsky. La sintaxis de la mayoría de los lenguajes de programación puede ser especificada utilizando una gramática Tipo-2, es decir, son gramáticas libres de contexto. Algunos lenguajes, incluyendo a Perl y a Lisp, contienen construcciones que permiten la ejecución durante la fase de análisis. Los lenguajes que permiten construcciones que permiten al programador alterar el comportamiento de un analizador hacen del

análisis de la sintaxis un problema sin decisión única, y generalmente oscurecen la separación entre análisis y ejecución. En contraste con el sistema de macros de Lisp y los bloques BEGIN de Perl, que pueden tener cálculos generales, las macros de C son meros reemplazos de cadenas, y no requieren ejecución de código.

Semántica estática

La semántica estática define las restricciones sobre la estructura de los textos válidos que resulta imposible o muy difícil expresar mediante formalismos sintácticos estándar. Para los lenguajes compilados, la semántica estática básicamente incluye las reglas semánticas que se pueden verificar en el momento de compilar. Por ejemplo el chequeo de que cada identificador sea declarado antes de ser usado (en lenguajes que requieren tales declaraciones) o que las etiquetas en cada brazo de una estructura *case* sean distintas. Muchas restricciones importantes de este tipo, como la validación de que los identificadores sean usados en los contextos apropiados (por ejemplo no sumar un entero al nombre de una función), o que las llamadas a subrutinas tengan el número y tipo de parámetros adecuado, pueden ser implementadas definiéndolas como reglas en una lógica conocida como sistema de tipos. Otras formas de análisis estáticos, como los análisis de flujo de datos, también pueden ser parte de la semántica estática. Otros lenguajes de programación como Java y C# tienen un análisis definido de asignaciones, una forma de análisis de flujo de datos, como parte de su semántica estática.

Sistema de tipos

Un sistema de tipos de datos define la manera en la cual un lenguaje de programación clasifica los valores y expresiones en *tipos*, cómo pueden ser manipulados dichos tipos y cómo interactúan. El objetivo de un sistema de tipos es verificar y normalmente poner en vigor un cierto nivel de exactitud en programas escritos en el lenguaje en cuestión, detectando ciertas operaciones inválidas. Cualquier sistema de tipos decidible tiene sus ventajas y desventajas: mientras por un lado rechaza muchos programas incorrectos, también prohíbe algunos programas correctos aunque poco comunes. Para poder minimizar esta desventaja, algunos lenguajes incluyen *lagunas de tipos*, conversiones explícitas no verificadas que pueden ser usadas por el programador para permitir explícitamente una operación normalmente no permitida entre diferentes tipos. En la mayoría de los lenguajes con tipos, el sistema de tipos es usado solamente para verificar los tipos de los programas, pero varios lenguajes, generalmente funcionales, llevan a cabo lo que se conoce como inferencia de tipos, que le quita al programador la tarea de especificar los tipos. Al diseño y estudio formal de los sistemas de tipos se le conoce como *teoría de tipos*.

Lenguajes tipados versus lenguajes no tipados

Se dice que un lenguaje es tipado si la especificación de cada operación debe definir los tipos de datos para los cuales es aplicable, con la implicación de que no es aplicable a otros tipos. Por ejemplo, "`este texto entre comillas`" es una cadena de caracteres. En la mayoría de los lenguajes de programación, dividir un número por una cadena de caracteres no tiene ningún significado. Por tanto, la mayoría de los lenguajes de programación modernos rechazarían cualquier intento de ejecutar dicha operación por parte de algún programa. En algunos lenguajes, estas operaciones sin significado son detectadas cuando el programa es compilado (validación de tipos "estática") y son rechazadas por el compilador, mientras en otros son detectadas cuando el programa es ejecutado (validación de tipos "dinámica") y se genera una excepción en tiempo de ejecución.

Un caso especial de lenguajes de tipo son los lenguajes de *tipo sencillo*. Estos son con frecuencia lenguajes de marcado o de scripts, como REXX o SGML, y solamente cuentan con un tipo de datos; comúnmente cadenas de caracteres que luego son usadas tanto para datos numéricos como simbólicos.

En contraste, un lenguaje *sin tipos*, como la mayoría de los lenguajes ensambladores, permiten que cualquier operación se aplique a cualquier dato, que por lo general se consideran secuencias de bits de varias longitudes. Lenguajes de alto nivel *sin datos* incluyen BCPL y algunas variedades de Forth.

En la práctica, aunque pocos lenguajes son considerados con tipo desde el punto de vista de la teoría de tipos (es decir, que verifican o rechazan *todas* las operaciones), la mayoría de los lenguajes modernos ofrecen algún grado de manejo de tipos. Si bien muchos lenguajes de producción proveen medios para evitar o rodear el sistema de tipado.

Tipos estáticos versus tipos dinámicos

En lenguajes con *tipos estáticos* se determina el tipo de todas las expresiones antes de la ejecución del programa (típicamente al compilar). Por ejemplo, `1` y `(2+2)` son expresiones enteras; no pueden ser pasadas a una función que espera una cadena, ni pueden guardarse en una variable que está definida como fecha.

Los lenguajes con tipos estáticos pueden manejar tipos *explícitos* o tipos *inferidos*. En el primer caso, el programador debe escribir los tipos en determinadas posiciones textuales. En el segundo caso, el compilador *infiere* los tipos de las expresiones y las declaraciones de acuerdo al contexto. La mayoría de los lenguajes populares con tipos estáticos, tales como C++, C# y Java, manejan tipos explícitos. Inferencia total de los tipos suele asociarse con lenguajes menos populares, tales como Haskell y ML. Sin embargo, muchos lenguajes de tipos explícitos permiten inferencias parciales de tipo; tanto Java y C#, por ejemplo, infieren tipos en un número limitado de casos.

Los lenguajes con tipos dinámicos determinan la validez de los tipos involucrados en las operaciones durante la ejecución del programa. En otras palabras, los tipos están asociados con *valores en ejecución* en lugar de *expresiones textuales*. Como en el caso de lenguajes con tipos inferidos, los lenguajes con tipos dinámicos no requieren que el programador escriba los tipos de las expresiones. Entre otras cosas, esto permite que una misma variable se pueda asociar con valores de tipos distintos en diferentes momentos de la ejecución de un programa. Sin embargo, los errores de tipo no pueden ser detectados automáticamente hasta que se ejecuta el código, dificultando la depuración de los programas, no obstante, en lenguajes con tipos dinámicos se suele dejar de lado la depuración en favor de técnicas de desarrollo como por ejemplo BDD y TDD. Ruby, Lisp, JavaScript y Python son lenguajes con tipos dinámicos.

Tipos débiles y tipos fuertes

Los lenguajes *débilmente tipados* permiten que un valor de un tipo pueda ser tratado como de otro tipo, por ejemplo una cadena puede ser operada como un número. Esto puede ser útil a veces, pero también puede permitir ciertos tipos de fallas que no pueden ser detectadas durante la compilación o a veces ni siquiera durante la ejecución.

Los lenguajes *fuertemente tipados* evitan que pase lo anterior. Cualquier intento de llevar a cabo una operación sobre el tipo equivocado dispara un error. A los lenguajes con tipos fuertes se les suele llamar *de tipos seguros*.

Lenguajes con tipos débiles como Perl y JavaScript permiten un gran número de conversiones de tipo implícitas. Por ejemplo en JavaScript la expresión `2 * x` convierte implícitamente `x` a un número, y esta conversión es exitosa inclusive cuando `x` es `null`, `undefined`, un `Array` o una cadena de letras. Estas conversiones implícitas son útiles con frecuencia, pero también pueden ocultar errores de programación.

Las características de *estáticos* y *fuertes* son ahora generalmente consideradas conceptos ortogonales, pero su trato en diferentes textos varía. Algunos utilizan el término *de tipos fuertes* para referirse a *tipos fuertemente estáticos* o, para aumentar la confusión, simplemente como equivalencia de *tipos estáticos*. De tal manera que C ha sido llamado tanto lenguaje de tipos fuertes como lenguaje de tipos estáticos débiles.

Implementación

La implementación de un lenguaje es la que provee una manera de que se ejecute un programa para una determinada combinación de software y hardware. Existen básicamente dos maneras de implementar un lenguaje: compilación e interpretación.

- Compilación: es el proceso que traduce un programa escrito en un lenguaje de programación a otro lenguaje de programación, generando un programa equivalente que la máquina será capaz de interpretar. Los programas traductores que pueden realizar esta operación se llaman compiladores. Estos, como los programas ensambladores avanzados, pueden generar muchas líneas de código de máquina por cada proposición del programa fuente.
- Interpretación: es una asignación de significados a las fórmulas bien formadas de un lenguaje formal. Como los lenguajes formales pueden definirse en términos puramente sintácticos, sus fórmulas bien formadas pueden no ser más que cadenas de símbolos sin ningún significado. Una interpretación otorga significado a esas fórmulas.

```
/**
 * Simple HelloButton() method.
 * @version 1.0
 * @author john doe <doe.j@example.com>
 */
HelloButton()
{
  JButton hello = new JButton( "Hello, wor
  hello.addActionListener( new HelloBtnLis

  // use the JFrame type until support for
  // new component is finished
  JFrame frame = new JFrame( "Hello Button
  Container pane = frame.getContentPane();
  pane.add( hello );
  frame.pack();
  frame.show();              // display the f
}
```

Código fuente de un programa escrito en el lenguaje de programación Java

Se puede también utilizar una alternativa para traducir lenguajes de alto nivel. En lugar de traducir el programa fuente y grabar en forma permanente el código objeto que se produce durante la compilación para utilizarlo en una ejecución futura, el programador solo carga el programa fuente en la computadora junto con los datos que se van a procesar. A continuación, un programa intérprete, almacenado en el sistema operativo del disco, o incluido de manera permanente dentro de la máquina, convierte cada proposición del programa fuente en lenguaje de máquina conforme vaya siendo necesario durante el procesamiento de los datos. El código objeto no se graba para utilizarlo posteriormente.

La siguiente vez que se utilice una instrucción, se la deberá interpretar otra vez y traducir a lenguaje máquina. Por ejemplo, durante el procesamiento repetitivo de los pasos de un ciclo o bucle, cada instrucción del bucle tendrá que volver a ser interpretada en cada ejecución repetida del ciclo, lo cual hace que el programa sea más lento en tiempo de ejecución (porque se va revisando el código en tiempo de ejecución) pero más rápido en tiempo de diseño (porque no se tiene que estar compilando a cada momento el código completo). El intérprete elimina la necesidad de realizar una compilación después de cada modificación del programa cuando se quiere agregar funciones o corregir errores; pero es obvio que un programa objeto compilado con antelación deberá ejecutarse con mucha mayor rapidez que uno que se debe interpretar a cada paso durante una ejecución del código.

La mayoría de lenguajes de alto nivel permiten la programación multipropósito, aunque muchos de ellos fueron diseñados para permitir programación dedicada, como lo fue el Pascal con las matemáticas en su comienzo. También se han implementado lenguajes educativos infantiles como Logo mediante una serie de simples instrucciones. En la actualidad son muy populares algunos lenguajes especialmente indicados para aplicaciones web, como Perl, PHP, Ruby, Python o JavaScript.

Dialectos

Un **dialecto** de un lenguaje de programación es una variación o extensión (relativamente pequeña) del lenguaje que no cambia su naturaleza intrínseca. Con lenguajes como Scheme y Forth, los implementadores pueden considerar que los estándares son insuficientes, inadecuados o ilegítimos, por lo que a menudo se desviarán del estándar, haciendo un nuevo *dialecto*. En otros casos, se crea un dialecto para su uso en un lenguaje específico de dominio, a menudo un subconjunto. En el mundo Lisp, la mayoría de los lenguajes que utilizan la sintaxis básica de una expresión S y la semántica similar a Lisp se consideran dialectos Lisp, aunque varían enormemente, al igual que, digamos, Raqueta y Clojure. Como es común que un lenguaje tenga varios dialectos, puede resultar bastante difícil para un programador sin experiencia encontrar la documentación correcta. El lenguaje de programación BASIC tiene muchos dialectos.

Técnica

Programming language

Para escribir programas que proporcionen los mejores resultados, cabe tener en cuenta una serie de detalles.

- *Corrección*. Un programa es correcto si hace lo que debe hacer tal y como se estableció en las fases previas a su desarrollo. Para determinar si un programa hace lo que debe, es muy importante especificar claramente qué debe hacer el programa antes de desarrollarlo y, una vez acabado, compararlo con lo que realmente hace.
- *Claridad*. Es muy importante que el programa sea lo más claro y legible posible, para facilitar así su desarrollo y posterior mantenimiento. Al elaborar un programa se debe intentar que su estructura sea sencilla y coherente, así como cuidar el estilo en la edición; de esta forma se ve facilitado el trabajo del programador, tanto en la fase de creación como en las fases posteriores de corrección de errores, ampliaciones, modificaciones, etc. Fases que pueden ser realizadas incluso por otro programador, con lo cual la claridad es aún más necesaria para que otros programadores puedan continuar el trabajo fácilmente. Algunos programadores llegan incluso a utilizar Arte ASCII para delimitar secciones de código. Otros, por diversión o para impedir un análisis cómodo a otros programadores, recurren al uso de código ofuscado.
- *Eficiencia*. Se trata de que el programa, además de realizar aquello para lo que fue creado (es decir, que sea correcto), lo haga gestionando de la mejor forma posible los recursos que utiliza. Normalmente, al hablar de eficiencia de un programa, se suele hacer referencia al tiempo que tarda en realizar la tarea para la que ha sido creado y a la cantidad de memoria que necesita, pero hay otros recursos que también pueden ser de

consideración al obtener la eficiencia de un programa, dependiendo de su naturaleza (espacio en disco que utiliza, tráfico de red que genera, etc.).

- *Portabilidad*. Un programa es portable cuando tiene la capacidad de poder ejecutarse en una plataforma, ya sea hardware o software, diferente a aquella en la que se elaboró. La portabilidad es una característica muy deseable para un programa, ya que permite, por ejemplo, a un programa que se ha desarrollado para sistemas GNU/Linux ejecutarse también en la familia de sistemas operativos Windows. Esto permite que el programa pueda llegar a más usuarios más fácilmente.

Paradigmas

Los programas se pueden clasificar por el paradigma del lenguaje que se use para producirlos. Los principales paradigmas son: imperativos, declarativos y orientación a objetos.

Los programas que usan un lenguaje imperativo especifican un algoritmo, usan declaraciones, expresiones y sentencias.[11] Una declaración asocia un nombre de variable con un tipo de dato, por ejemplo: `var x: integer;`. Una expresión contiene un valor, por ejemplo: `2 + 2` contiene el valor 4. Finalmente, una sentencia debe asignar una expresión a una variable o usar el valor de una variable para alterar el flujo de un programa, por ejemplo: `x := 2 + 2; if x == 4 then haz_algo();`. Una crítica común en los lenguajes imperativos es el efecto de las sentencias de asignación sobre una clase de variables llamadas "no locales".[12]

Los programas que usan un lenguaje declarativo especifican las propiedades que la salida debe conocer y no especifican cualquier detalle de implementación. Dos amplias categorías de lenguajes declarativos son los lenguajes funcionales y los lenguajes lógicos. Los lenguajes funcionales no permiten asignaciones de variables no locales, así, se hacen más fácil, por ejemplo, programas como funciones matemáticas.[12] El principio detrás de los lenguajes lógicos es definir el problema que se quiere resolver (el objetivo) y dejar los detalles de la solución al sistema.[13] El objetivo es definido dando una lista de sub-objetivos. Cada sub-objetivo también se define dando una lista de sus sub-objetivos, etc. Si al tratar de buscar una solución, una ruta de sub-objetivos falla, entonces tal sub-objetivo se descarta y sistemáticamente se prueba otra ruta.

La forma en la cual se programa puede ser por medio de texto o de forma visual. En la programación visual los elementos son manipulados gráficamente en vez de especificarse por medio de texto.

Véase también

- Anexo:Lenguajes de programación
- Programación estructurada
- Programación modular
- Programación funcional
- Programación orientada a aspectos
- Programación con restricciones
- Programación a nivel funcional
- Programación a nivel de valores
- Lenguaje de programación esotérico
- Anexo:Cronología de los lenguajes de programación

Referencias

1. *Tecnicos de Soporte Informatico de la Comunidad de Castilla Y Leon. Temario Volumen i Ebook* (https://books.google.es/books?id=SUjFswQk1_4C&pg=PA312&dq=lenguaje++programaci%C3%B3n+es&hl=es&sa=X&ved=0ahUKEwiq6rWskdjlAhUJkhQKHWzbCGQQ6AEINjAC#v=onepage&q=lenguaje%20%20programaci%C3%B3n%20es&f=false). MAD-Eduforma. ISBN 9788466551021. Consultado el 7 de noviembre de 2019.
2. Juganaru Mathieu, Mihaela (2014). *Introducción a la programación* (https://editorialpatria.com.mx/pdffiles/9786074384154.pdf). Grupo Editorial Patria. ISBN 978-607-438-920-3. Consultado el 21 de mayo de 2021.
3. Yáñez, Luis Hernández (2013-2014). *Fundamentos de la programación* (https://www.fdi.ucm.es/profesor/luis/fp/fp.pdf). Consultado el 21 de mayo de 2021.
4. Joyanes Aguilar, Luis (2008). José Luis García y Cristina Sánchez, ed. *FUNDAMENTOS DE PROGRAMACIÓN Algoritmos, estructura de datos y objetos*. McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U. ISBN 978-84-481-6111-8.