

Introducción a la programación > 2. Linguaxes de programación

2. LINGUAXES DE PROGRAMACIÓN



Unha linguaxe de programación é un **conjunto de regras** e sintaxe que permiten aos desenvolvedores **escribir instrucións que un computador pode entender e executar**.

O obxectivo principal dunha linguaxe de programación é permitir a creación de programas que poden resolver problemas, automatizar tarefas, procesar datos e crear aplicacions.

[14 lenguajes de programación para partirse la caja pero que funcionan](#)

2.1. SINTAXE E SEMÁNTICA

2.1.1. Sintaxe

A sintaxe dunha linguaxe de programación é o **conxunto de regras e estruturas** que determinan como deben ser escritos os programas nunha determinada linguaxe.

Estas regras especifican a forma correcta de organizar e combinar palabras, símbolos, e outros elementos léxicos para crear instrucións válidas que un compilador ou intérprete poida entender e procesar. A sintaxe **define a forma externa do código**, sen entrar no seu significado (que é tratado pola semántica).

2.1.2. Semántica

A semántica refírese **ao significado das instrucións escritas na linguaxe**. Mientras que a sintaxe se encarga da estrutura e a forma correcta de escribir o código, a semántica **describe o que fai ese código cando é executado**. A semántica define o comportamento do programa, os efectos das instrucións e as relacións entre diferentes elementos do código.

2.2. TIPO DE LINGUAXES DE PROGRAMACIÓN SEGUNDO O NIVEL

O **nivel** dunha linguaxe de programación refírese ao **grao de abstracción que ofrece sobre o hardware e o sistema operativo**, e pode influír na complexidade, a flexibilidade e a eficiencia do desenvolvemento de software.

2.2.1. LINGUAXES DE ALTO NIVEL

Unha linguaxe de alto nivel é un **tipo de linguaxe de programación que está máis próxima á linguaxe humana** e á lóxica matemática, en comparación coas linguaxes de baixo nivel. Estas linguaxes están **deseñadas para ser fáceis** de ler e escribir para os seres humanos, abstraendo as complexidades do hardware subxacente e permitindo que os programadores se concentren na lóxica e na resolución de problemas sen preocuparse polos detalles da implantación do hardware.

2.2.1.1. Características

- **Abstracción de hardware:** As linguaxes de alto nivel ocultan os detalles específicos do hardware, como a administración de memoria e os códigos de máquina. Isto permite aos programadores escribir código sen preocuparse pola arquitectura específica do procesador.
- **Sintaxe clara e flexible:** A sintaxe das linguaxes de alto nivel está deseñada para ser semellante á gramática das linguaxes naturais ou para seguir unha lóxica matemática, o que facilita a súa comprensión e escritura.
- **Librerías e frameworks:** As linguaxes de alto nivel poden vir acompañadas de bibliotecas e *frameworks* que proporcionan funcionalidades avanzadas e ferramentas para facilitar o desenvolvemento de aplicacións complexas.
- **Portabilidade:** O código escrito en linguaxes de alto nivel tende a ser más portátil, xa que o mesmo código pode executarse en diferentes plataformas con poucas ou ningunhas modificacións, grazas á presenza de compiladores ou intérpretes específicos para cada plataforma.

2.2.1.2. Vantaxes

- **Facilidade de Desenvolvemento:** A sintaxe simplificada e a abstracción do hardware permiten un desenvolvemento más rápido e menos propenso a erros.
- **Lectura e mantemento:** O código en linguaxes de alto nivel é xeralmente más flexible e más fácil de manter e modificar, grazas á súa sintaxe clara e á súa estrutura organizada.
- **Productividade:** Permiten aos programadores ser más produtivos, xa que non teñen que preocuparse tanto polos detalles de baixo nivel e poden centrarse na resolución de problemas e na creación de funcionalidades.
- **Depuración e Probas:** As ferramentas de depuración e as bibliotecas de probas son más sofisticadas e están más dispoñibles para linguaxes de alto nivel, facilitando a detección e corrección de erros.

2.2.1.3. Desvantaxes

- **Rendemento:** As linguaxes de alto nivel poden ser menos eficientes en termos de rendemento en comparación con linguaxes de baixo nivel, debido á sobrecarga introducida pola abstracción e a interpretación ou compilación do código.
- **Menor Control sobre o Hardware:** A abstracción pode limitar o control directo sobre o hardware, o que pode ser importante en aplicacións de tempo real.

2.2. LINGUAXES DE BAIXO NIVEL

Unha linguaxe de baixo nivel **é un tipo de linguaxe de programación que está máis próxima á arquitectura do hardware** do computador, proporcionando menos abstracción en comparación coas linguaxes de alto nivel. Estas linguaxes permiten un **maior control sobre os recursos do sistema** e as operacións de baixo nivel, pero tamén requiren unha comprensión máis profunda da arquitectura do hardware e a súa operación.

2.2.2.1. Características

- **Control detallado sobre o hardware:** As linguaxes de baixo nivel proporcionan un acceso directo á memoria e ás operacións de procesador, permitindo aos programadores controlar con precisión o funcionamento do hardware.
- **Menor abstracción:** Estas linguaxes ofrecen pouca ou ningunha abstracción en comparación coas linguaxes de alto nivel. Isto significa que os programadores deben manexar detalles como a administración de memoria e as operacións aritméticas a nivel de bits e bytes.
- **Eficiencia:** O código escrito en linguaxes de baixo nivel tende a ser máis eficiente en termos de rendemento, xa que se executa más cerca do hardware e pode ser optimizado para tarefas específicas.
- **Sintaxe complexa:** A sintaxe das linguaxes de baixo nivel é xeralmente más complexa e menos lexible para os seres humanos en comparación coas linguaxes de alto nivel.

2.2.2.2. Exemplos

- **Linguaxe Máquina:** É a linguaxe más baixa e próxima ao hardware, composta por instrucións en **código binario que o procesador pode executar directamente**. Non require tradución adicional, pero é extremadamente difícil de ler e escribir para os humanos. O código binario específico para unha instrución, como `10110000 01100001`, que pode corresponder a unha operación de movendo datos na memoria.
- **Linguaxe Ensamblador:** É unha linguaxe de baixo nivel que usa abreviaturas e símbolos para representar instrucións de código máquina. Cada instrución de Ensamblador corresponde a unha operación de baixo nivel que o procesador pode realizar. Menos abstracta que as linguaxes de alto nivel, pero máis comprensible para os humanos que a linguaxe máquina.

2.2.2.3. Vantaxes

- **Eficiencia e Rendemento:** O código escrito en linguaxes de baixo nivel pode ser máis eficiente e rápido debido á súa proximidade co hardware.
- **Control Completo:** Permiten un control total sobre as operacións do procesador e a memoria, o que é útil para tarefas que requieren un alto rendemento ou que están moi próximas ao hardware, como sistemas operativos e controladores de dispositivos.
- **Optimización de Recursos:** A programación en baixo nivel pode optimizar o uso dos recursos do sistema para aplicacións específicas.

2.2.2.3. Desvantaxes

- **Complexidade:** A programación en linguaxes de baixo nivel é complexa e require unha comprensión detallada do hardware. O código é menos lexible e máis difícil de manter e depurar.

- **Portabilidade:** O código escrito en linguaxes de baixo nivel pode ser menos portátil entre diferentes arquitecturas de hardware, xa que está fortemente vinculado a un tipo específico de procesador ou sistema.
 - **Tempo de Desenvolvimento:** O desenvolvemento e depuración de código en baixo nível pode levar máis tempo debido á súa complexidade e ao nivel de detalle requirido.
-

2.2.3. CASO ESPECIAL: LINGUAXE C

A linguaxe de programación **C** é unha linguaxe que se sitúa nun **punto intermedio entre as linguaxes de alto nivel e as linguaxes de baixo nivel**. A súa versatilidade faiña útil tanto para tarefas de alto nivel como para programacións que requieren un acceso máis próximo ao hardware.

2.2.3.1. Características como linguaxe de alto nível

- **Sintaxe clara e flexible:** C ten unha sintaxe relativamente clara que permite escribir código de maneira eficiente e organizada, semellante a outras linguaxes de alto nível.
- **Bibliotecas e funcións standard:** Inclúe unha biblioteca estándar rica que proporciona unha ampla gama de funcións para manipulación de cadeas, entrada/saída, matemáticas, e manipulación de memoria.
- **Portabilidade:** O código C pode ser portado entre diferentes sistemas operativos e arquitecturas de hardware con poucas modificacións, debido á súa ampla aceptación e soporte.

2.2.3.2. Características como linguaxe de baixo nível

- **Acceso directo a memoria:** C permite manipular directamente a memoria mediante o uso de **punteiros**, o que ofrece un control detallado sobre a memoria e as operacións do hardware.
- **Manipulación de bits e bytes:** Permite operacións a nivel de bits e bytes, o que é útil para tarefas que requieren unha manipulación precisa dos datos.
- **Control sobre a xestión de recursos:** Permite un control detallado sobre o fluxo de execución e a xestión de recursos do sistema. Isto é especialmente útil para programar sistemas operativos, controladores de dispositivos e software de alto rendemento.

2.2.3.3. Vantaxes

- **Eficiencia e rendemento:** C é coñecida pola súa eficiencia e capacidade de xerar código que se executa rapidamente. Isto a fai ideal para aplicacións de alto rendemento e programacións de sistema.
- **Versatilidade:** Pode ser usada para desenvolvemento de sistemas operativos, compiladores, controladores de dispositivos, e software de aplicación.
- **Control e flexibilidade:** Ofrece un alto grao de control sobre a memoria e o hardware, permitindo optimizacións e personalizacións precisas.

2.2.3.4. Desvantaxes

- **Seguridade e Robustez:** A manipulación directa da memoria e a falta de comprobacións automáticas de tipos poden levar a errores difíciles de detectar, como violacións de acceso a memoria e fugas de memoria.
- **Complexidade de Depuración:** O código en C pode ser máis difícil de depurar debido á súa complexidade e a posibilidade de errores sutís asociados coa manipulación de punteiros e memoria.

2.3. PARADIGMAS DE PROGRAMACIÓN

Un **paradigma de programación** é un estilo ou enfoque fundamental que guía **a maneira en que se desenvolve e organiza o código nun programa informático**. Cada paradigma ten as súas propias regras, conceptos e técnicas para resolver problemas e estruturar o código, influíndo na forma en que se escriben e interpretan os programas. Os paradigmas axudan a definir a forma na que os problemas son modelados e resoltos mediante a programación.

2.3.1. PROGRAMACIÓN ESTRUTURADA

A **programación estruturada** é un paradigma de programación que se centra na **organización e control do fluxo de execución do programa mediante o uso de estruturas de control** claras e ben definidas. O obxectivo principal da programación estruturada é mellorar a lexibilidade, a mantebilidade e a eficiencia do código mediante unha estrutura ordenada e modular.

Este paradigma estudarase en profundidade nunha unidade específica.

2.3.2. PROGRAMACIÓN ORIENTADA A OBXECTOS (POO)

A **programación orientada a obxectos (POO)** é un paradigma de programación que organiza o código en **obxectos** e **clases**, facilitando a modelaxe e a xestión da complexidade en programas de gran tamaño. POO está baseado en varios conceptos clave que axudan a estruturar o código de forma más natural e modular, permitindo que os obxectos representen entidades do mundo real ou conceptos abstractos.

Este paradigma estadarase en profundidade nunha unidade específica.

2.3.3. PROGRAMACIÓN FUNCIONAL

A **programación funcional** é un paradigma de programación que **trata as funcións como elemento principal e enfatiza o uso da composición de funcións** para resolver problemas.

Ao contrario de paradigmas como a programación imperativa ou orientada a obxectos, que se enfocan en manipular o estado e usar obxectos, a programación funcional pon o foco na transformación de datos e en definir cálculos a través de funcións matemáticas.

Este paradigma estudarase en profundidade nunha unidade específica.

2.3.4. PROGRAMACIÓN LÓXICA

A **programación lóxica** é un paradigma de programación **baseado na lóxica formal e na dedución para resolver problemas**. Este enfoque está centrado en expresar o que se quere lograr mediante *regras* e *feitos* e permitir que o sistema deduza as solúções a partir destas declaracións.

É fundamentalmente diferente dos paradigmas imperativos e orientados a obxectos, xa que non especifica unha secuencia explícita de pasos para resolver un problema, senón que define *relacións* e deixa que o sistema deduza as respostas.

Os conceptos clave da programación lóxica son:

- **Feitos:** Declaracións que describen información ou verdadeiros enunciados sobre o dominio do problema. Representan coñecementos básicos que se aceptan como verdadeiros.
 - **Regras:** Declaracións que definen como derivar novos feitos a partir de feitos existentes. As regras consisten en unha cabeza (o feito derivado) e un corpo (condicións necesarias para que a cabeza sexa verdadeira).
 - **Consultas:** Preguntas ou consultas feitas ao sistema para deducir información baseada en feitos e regras. O sistema usa un proceso de dedución para responder ás consultas.
 - **Inferencia:** O proceso mediante o cal o sistema deduce novas informacións ou respostas baseadas en feitos e regras. A inferencia pode ser automática e está baseada en algoritmos de resolución, como o algoritmo de *resolución*.
 - **Unificación:** O proceso de igualar termos e substituír variables para que as regras e feitos se apliquen correctamente durante a inferencia.
 - **Resolución:** O método de dedución usado para determinar se unha consulta é verdadeira ou falsa ao buscar unha coincidencia entre a consulta e os feitos ou regras existentes.
-

2.4. COMPILEDORES E INTÉPRETES

Como dixemos unha **linguaxe de programación está deseñado para ser sínxelo** de ler e escribir por persoas. Pero o **hardware** dun equipo **non é capaz de comprender** esta linguaxe.

Un equipo, e máis concretamente unha **CPU**, entende únicamente unha linguaxe chamado **código máquina**. Este código máquina é moi simple e francamente pesado para escribir, xa que só consta de ceros e uns:

```
001010001110100100101010000001111  
11100110000011101010010101101101  
...  
.
```

O código máquina parece sínxelo a simple vista, pero a súa sintaxe é máis complexa e entravesada que a dunha linguaxe de programación de alto nivel. Moi poucos programadores escriben nesta linguaxe e só a utilizan en propósitos moi concretos.

En lugar de iso, **creáronse programas tradutores** que permiten aos programadores escribir en linguaxes de alto nivel e a través destes tradutores **converter os programas en código máquina**.

Dado que o **código máquina** está ligado ao hardware da máquina que o executa, este código **non é portable entre equipos de diferente tipo**. Os programas escritos en linguaxes de alto nivel poden ser traducidos entre **distintas máquinas usando tradutores distintos**.

Estes tradutores pódense dividir en dous grandes bloques: **compiladores e intérpretes**.

2.4.1. INTÉPRETES

Os **intérpretes** len o código fonte (o programa escrito na linguaxe de alto nivel) dos programas tal e como o escribiu o programador, analízao e interpreta as instrucións sobre a marcha. Polo tanto este é capaz de manter unha conversación interactiva.

O seu funcionamento é o seguinte:

- Lectura do código fonte:** O intérprete recibe o código fonte escrito na linguaxe de programación que soporta e **lé o código línea por línea** ou en bloques.
- Análise léxica e sintáctica:** O intérprete analiza o código para identificar os *tokens* (elementos básicos do código como palabras clave, operadores, e identificadores) e verifica que o código segue a sintaxe correcta da linguaxe.
- Execución:** O intérprete executa as instrucións directamente, realizando as operacións descritas no código fonte.
- Producción de resultados:** O intérprete produce os resultados da execución.

ⓘ Conversión a código intermedio

En algúns casos, o intérprete pode converter o código fonte en código intermedio ou en representación interna que facilita a execución despois da análise léxica e sintáctica. Este paso non é necesario en todos os intérpretes.

2.4.1.1. Vantaxes

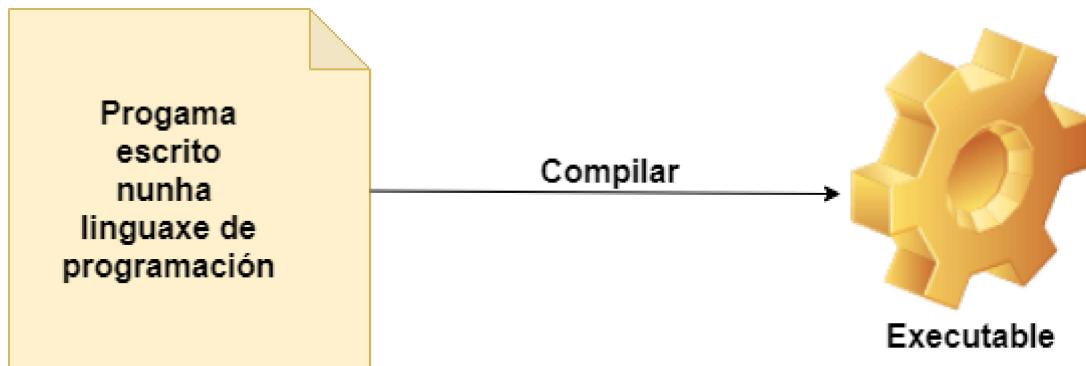
- Facilidade de uso:** Permiten a execución e probas de código de forma rápida e interactiva. Ideal para desenvolvemento e depuración onde o desenvolvedor pode executar e probar código en pequenos fragmentos.
- Portabilidade:** O código fonte interpretado pode ser executado en diferentes plataformas sen necesidade de recompilar, sempre que haxa un intérprete dispoñible para a plataforma desexada.
- Flexibilidade:** Facilita a modificación e a execución dinámica de código. Pode ser útil en sistemas que requieren *scripts* ou configuracións que cambian frecuentemente.

2.4.1.2. Desvantaxes

- Desempeño:** A execución pode ser máis lenta en comparación co código compilado, xa que o intérprete debe procesar e interpretar o código en tempo real.
- Uso de recursos:** A interpretación en tempo real pode consumir máis recursos, como CPU e memoria, en comparación co código previamente compilado.
- Distribución de código:** O código fonte interpretado pode ser máis fácil de modificar e, por tanto, pode presentar riscos de seguridade se se distribúe sen protexer adequadamente.

2.4.2. COMPILEDORES

Os **compiladores** necesitan o programa completo nun ficheiro e logo executa o proceso de tradución do código fonte. O código máquina resultante colócase nun novo arquivo para a súa execución posterior. Un exemplo de linguaxe que utiliza compilador é C.



En sistemas Windows estes executables teñen a extensión “.exe” ou “.dll”.

Se abrimos un arquivo con código máquina nun ficheiro de texto veríamos algo totalmente ilexible:

O seu funcionamento é o seguinte:

- Análise léxico:** O compilador lee o código fonte e o divide en unidades lexicais chamadas *tokens* (como palabras clave, identificadores, operadores, etc.). Por exemplo, a instrución `int x = 5;` é dividida en *tokens* como `int`, `x`, `=`, `5`, e `;`.
 - Análise sintáctico:** O compilador analiza a estrutura do código para asegurarse de que segue a sintaxe da linguaxe. Este paso constrúe unha árbore de análise sintáctica que representa a estrutura do programa. Por exemplo, a instrución `int x = 5;` é analizada para verificar que a declaración é válida dentro do contexto da linguaxe.
 - Análise semántico:** O compilador verifica o significado do código e asegúrese de que o uso das variables, tipos de datos e operacións sexa coherente e correcto. Por exemplo, verifica que `x` sexa unha variable que pode ser asignada e que `5` é un valor compatible co tipo `int`.
 - Optimización:** O compilador aplica diversas técnicas para mellorar o código, reducir o tamaño do código e mellorar o desempeño. Esta etapa pode optimizar tanto o código fonte como o código intermediario. Por exemplo, pode simplificar operacións matemáticas ou eliminar código innecesario.
 - Xeración de código:** O compilador traduce a representación interna do código a código de máquina ou a un código intermediario que pode ser executado por un procesador. Por exemplo, a instrución `int x = 5;` é traducida a un conxunto de instrucións que a CPU pode executar para asignar o valor `5` á variable `x`.

- 6. **Enlazado:** O compilador ou un *enlazador* externo combina o código obxecto xerado coas bibliotecas e módulos necesarios para crear o arquivo executábel final.

2.4.2.1. Vantaxes

- **Desempeño:** O código compilado pode ser executado moito máis rápido que o código interpretado, xa que está convertido en código de máquina que a CPU pode executar directamente.
- **Optimización:** Os compiladores aplican diversas técnicas de optimización que melloran a eficiencia do código executado.
- **Detección de errores:** Os compiladores realizan unha análise exhaustiva do código fonte e poden detectar erros e inconsistencias antes da execución, o que facilita a corrección de problemas.
- **Independencia do código fonte:** Unha vez compilado, o código obxecto ou executábel non require a presenza do compilador para ser executado, permitindo a distribución e execución do software sen dependencias adicionais.

2.4.2.2. Desvantaxes

- **Tempo de compilación:** A compilación pode levar tempo, especialmente para programas grandes ou complexos, o que pode ralentizar o proceso de desenvolvemento e depuración.
 - **Flexibilidade:** A modificación e proba de código poden ser menos flexibles en comparación coa interpretación, xa que cada cambio require recompilación.
 - **Complexidade:** O proceso de compilación pode ser complexo e require unha comprensión profunda do funcionamento do compilador e das técnicas de optimización.
-

2.4.3. CASO ESPECIAL DE JAVA

Java é un exemplo de linguaxe de programación que utiliza tanto un **compilador como un intérprete** para a execución do código. Este enfoque combina os melhores aspectos da compilación e da interpretación, permitindo a portabilidade e a eficiencia.

O funcionamento é o seguinte:

1. Compilación en Java

- **Código fuente (Java):** O programador escribe o código fuente en Java, que é un código de alto nivel, entendible para os humanos.
- **Compilador:** O compilador Java (`javac`) toma o código fuente escrito en Java e compílalo a **bytecode** de Java, que é un código intermediario e independente da plataforma.

2. Execución en Java

- **Máquina Virtual de Java (JVM):** A *Máquina Virtual de Java* (JVM, Java Virtual Machine) é un intérprete que executa o *bytecode* xerado polo compilador `javac`. A JVM é responsable de interpretar e executar o *bytecode*, e está deseñada para ser multiplataforma, o que significa que o mesmo *bytecode* pode ser executado en diferentes sistemas operativos e arquitecturas de hardware, sempre que haxa unha JVM dispoñible para esa plataforma.
- **Interpretación e Compilación Xusto a Tempo (JIT):** A JVM usa un mecanismo chamado *Compilación Xusto a Tempo* (JIT, Just-In-Time Compilation) para mellorar o desempeño. O JIT compila o *bytecode* en código de máquina nativo durante a ejecución, o que permite a ejecución más rápida.

2.4.3.1 Beneficios

- **Portabilidad:** O *bytecode* de Java pode ser executado en calquera plataforma que teña unha JVM. Isto permite que o mesmo código fuente se poida executar en diferentes sistemas operativos e arquitecturas de hardware sen modificacións.
 - **Seguridade:** O *bytecode* e a JVM proporcionan unha capa adicional de seguridade, xa que o *bytecode* é executado nun ambiente controlado que pode verificar e protexer contra certas vulnerabilidades.
 - **Desempeño mellorado:** A compilación Xusto a Tempo (JIT) permite que a ejecución do código sexa optimizada e mellorada en tempo real, aumentando o desempeño do programa.
-

2.5. TIPADO

A clasificación das linguaxes de programación segundo o seu **tipado** refírese a **como se manexan e restrinxen os tipos de datos dentro dunha linguaxe**. O tipado dunha linguaxe pode afectar como se realizan as operacións, a robustez do código e a facilidade de detección de erros. Aquí están os principais tipos de tipado en linguaxes de programación:

2.5.1. Tipado Estático e Tipado Dinámico

- No **Tipado Estático**, o **tipo** de cada variable e expresión **é coñecido** e verificado no momento da **compilación**. Isto significa que os errores de tipo son detectados antes de que o programa se execute.
- En cambio, no **Tipado Dinámico** os **tipos** de datos son verificados e determinados durante a **execución do programa**. Isto significa que o tipo dunha variable pode cambiar a medida que o programa se executa.

2.5.2. Tipado Débil e Tipado Forte

- No **Tipado Débil**, o sistema de tipos **permite conversións automáticas** ou implícitas entre tipos de datos. Isto significa que a linguaxe pode realizar conversións entre tipos de forma implícita, o que pode levar a comportamentos inesperados.
- En cambio no **Tipado Forte**, o sistema de tipos require que as **conversións** entre tipos sexan explícitas e **controladas**. As conversións implícitas non se permiten, e as incompatibilidades de tipo son tratadas como erros.

2.5.3. Tipado Explícito e Tipado Implícito

- No **Tipado Explícito**, o **tipo** de cada variable debe ser **especificado** claramente polo **programador** no momento da declaración ou **definición**.
 - Por outro lado, no **Tipado Implícito**, o **tipo** das variables é **inferido polo compilador ou intérprete** en función do contexto. O **programador non ten que especificar o tipo** explícitamente.
-

2.6. ELEMENTOS COMÚNS DA SINTAXE

Aínda que as linguaxes de programación poden diferir en sintaxe e características, todas **comparten algúns elementos comúns fundamentais** que permiten a construcción de programas efectivos e eficientes. Aquí teñes os elementos comúns a todas as linguaxes de programación:

- **Identificadores:** Son nomes definidos polo programador para representar **variables, funcións, clases, etc.**
- **Variables:** As variables son **espazos na memoria que almacenan datos e teñen un nome asociado**. Poden conter diferentes tipos de datos, como números, cadeas de texto, ou obxectos más complexos.
- **Funcións:** As funcións son **bloques de código** que realizan unha **tarefa específica** e poden ser chamadas desde outras partes do programa. Poden recibir parámetros e devolver valores.
- **Palabras reservadas:** Son **palabras** predefinidas pola linguaxe que teñen **significados especiais** e non poden ser usadas como identificadores.
- **Delimitadores e separadores:** Son **caracteres** que se usan para **estruturar o código e separar diferentes partes do mesmo**. Inclúen parénteses, corchetes, chaves, punto e coma, comas, etc.
- **Tipos de Datos:** Os tipos de datos definen a **clase de datos que unha variable pode almacenar**. Os tipos de datos comúns inclúen:
 - **Enteiros:** Números enteros.
 - **Flotantes:** Números con decimais.
 - **Cadeas de texto:** Secuencias de caracteres.
 - **Booleanos:** Valores verdadeiros ou falsos.
 - **Estructuras de Datos:** As linguaxes de programación permiten almacenar e manipular coleccións de datos usando diferentes estruturas, como:
 - **Arrays ou vectores:** Coleccións de elementos do mesmo tipo.
 - **Listas:** Coleccións dinámicas que permiten elementos de diferentes tipos.
 - **Mapas ou Dicionarios:** Pares clave-valor.
- **Operadores:** Os operadores **utilízanse sobre variables e valores para realizar diferentes tipos de operacións**. Hai diferentes categorías de operadores:
 - **Aritméticos (+, -, *, /, %):** Realizan operacións matemáticas básicas.
 - **De comparación (==, !=, >, <, >=, <=):** Comparan dous valores.
 - **Lóxicos (&&, ||, !):** Realizan operacións lóxicas.
- **Estructuras de Control:** As estruturas de control permiten **cambiar o fluxo de ejecución do programa** baseándose en condicións ou repeticións. As principais estruturas de control inclúen:
 - **Condicionais:** Executan bloques de código baseados en condicións.
 - **Bucles:** Repiten un bloque de código varias veces.
- **Entrada e Saída (I/O):** Os programas precisan interactuar co mundo exterior a través da entrada e saída de datos. Isto inclúe ler datos do usuario, ficheiros, ou enviar datos a unha pantalla ou ficheiro.

- **Comentarios:** Son **anotacións** no código que **non son executadas**, usadas para explicar o código ou deixar notas. Poden ser de liña única ou de varias liñas.
-

2.6. HISTORIA E EVOLUCIÓN

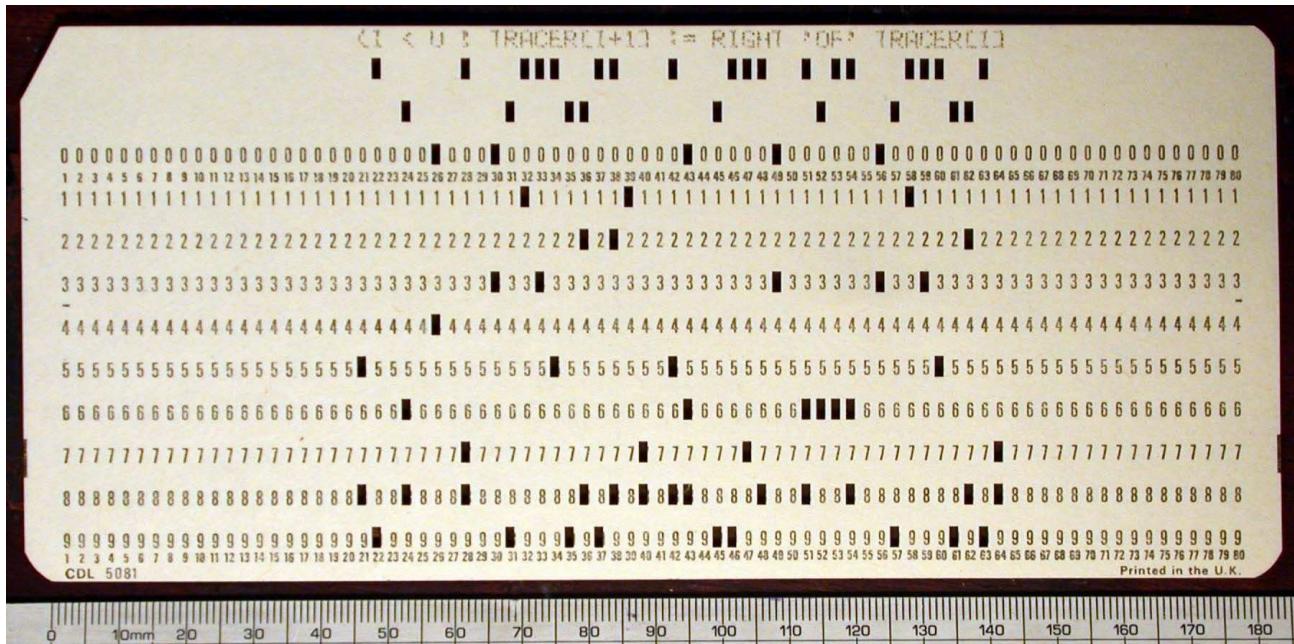
Desde os primeiros días da informática, a capacidade de instruir e controlar as máquinas computacionais foi fundamental para o desenvolvimento tecnológico. No comezo, os programas eran escritos en código máquina, unha forma de programación que directamente utiliza os códigos binarios que os ordenadores entenden. Este enfoque, mentres que extremadamente eficiente, era tamén notoriamente complexo e propenso a erros.

O desexo de simplificar a programación e facer o proceso más accesible levou á aparición das primeiras linguaxes de alto nivel. Estas linguaxes abstraían a complexidade do hardware, permitindo aos programadores escribir código usando estruturas e conceptos más comprensibles para os humanos. Co tempo, estas linguaxes evolucionaron, introducindo novos paradigmas, técnicas e funcionalidades que permitiron aos desenvolvedores afrontar problemas cada vez más complexos e diversificados.

 [2024 Stack Overflow Developer Survey](#)

2.6.1. 40S: OS PRIMEIROS PASOS

A programación con tarxetas perforadas foi un dos **primeiros métodos de entrada de datos e instrucións** aos computadores, utilizado amplamente desde os anos 1940 ata os anos 1970. As **tarxetas perforadas** son pezas de cartón cun tamaño estándar, que conteñen perforacións organizadas en filas e columnas.



Cada perforación ou ausencia desta representa un bit de información. Estas tarxetas eran programadas polo tanto en **linguaxe máquina**.

Para facilitar o proceso, durante esta época desenvolvéronse as linguaxes de Ensamblador (*assembly language*), que permitían usar abreviaturas en lugar de números binarios.

2.6.2. 50S: OS PRIMEIROS PARADIGMAS

2.6.2.1. Fortran



En 1957 aparece **FORTRAN**. O proxecto foi liderado por John Backus e o seu equipo na IBM. O obxectivo era crear unha linguaxe de programación que permitise escribir **programas para cálculos numéricos e científicos** máis facilmente que con código máquina ou Ensamblador. Foi unha das **primeiras linguaxes de alto nivel**.

```

!-----main program
use devObject
implicit none
!-----Executables
call open_devObjects
!
call usersub
!
call close_devObjects
end
!-----end of main

subroutine usersub
use devObject
implicit none

integer,parameter:: size=1024
type(devVar) dv1,dv2,dv3,dv4

real ar1d1(size),ar1d2(size),ar1d3(size), ar1dresult(size)

call random_number(ar1d1)
call random_number(ar1d2)
call random_number(ar1d3)

dv1=allocate_dv('real',size)
dv2=allocate_dv('real',size)
dv3=allocate_dv('real',size)

call transfer_r4(ar1d1,dv1,.true.)
call transfer_r4(ar1d2,dv2,.true.)
call transfer_r4(ar1d3,dv3,.true.)

!pointwise multiplication division addition subtraction test
dv4=(3.14159*dv1*(dv2+dv1)*(dv1-(.553+dv3))*dv2+(-.244))/dv1

call transfer_r4(ar1dresult,dv4,.false.)

call deallocate_dv(dv1)
call deallocate_dv(dv2)
call deallocate_dv(dv3)
call deallocate_dv(dv4)
end subroutine usersub

```

2.6.2.2. Cobol



2 anos máis tarde desenvolveuse para programar **software para negocios e administración**, **COBOL** (*Common Business-Oriented Language*), cunha sintaxe próxima ao inglés.

Debido a gran cantidad de programas desenvolvidas nesta linguaxe aínda **está en uso hoxe en día**.

```
COMMAND ==> PROC-REG-ENTRADA.
000974      MOVE  CLI-TIPCTA-ENT  TO  CLI-TIPCTA-SAL.
000975      MOVE  NUMCTA-ENT   TO  NUMCTA-SAL.
000976      MOVE  NOMREF-ENT   TO  NOMREF-SAL.
000977      MOVE  NOMCLIE-ENT  TO  NOMCLIE-SAL.
000978      MOVE  DOMIC-CLI-ENT TO  DOMIC-CLI-SAL.
000979      MOVE  SALDOMAX-ENT TO  SALDOMAX-SAL.
000980      MOVE  EL-FECHA-ENT <IND-FECHA-ENT-FIN> TO  FECHA-ENTRADA.
000981      *
000982      MOVE  NUM-MOUS-ENT GREATER SALDOMAX-ENT
000983      IF    NUM-MOUS-ENT GREATER SALDOMAX-ENT
000984          MOVE  B      TO  IND-VALOR-1
000985      ELSE
000986          COMPUTE IND-VALOR-1 ROUNDED = 1 -
000987          <NUM-MOUS-ENT / SALDOMAX-ENT>.
000988      *
000989      MOVE  FEC-VALOR-ENT  TO  FEC-VALOR-SAL.
000990      MOVE  FEC-OPER-ENT  TO  FEC-OPER-SAL.
000991      IF    FEC-OPER-ENT = FEC-VALOR-ENT
000992          MOVE  ZERO   TO  NUM-DIAS-DIF
000993      ELSE
000994          CALL  'DIFERDIA' USING FEC-VALOR-ENT.
000995
```

El mundo depende del lenguaje COBOL y casi no hay desarrolladores que lo conozcan. IBM decía tener la solución... pero no

2.6.3. 60S: DIVERSIFICACIÓN

BASIC PROGRAMMING

BASIC (*Beginner's All-purpose Symbolic Instruction Code*) é unha linguaxe de programación que foi deseñada para ser sinxela e **accesible para principiantes**. Foi desenvolvida a súa primeira versión en 1964.

Nos anos 80, unha versión de BASIC desenvolvida por **Bill Gates e Paul Allen** foi licenciada a varios fabricantes de computadores persoais, converténdose nun estándar de facto.

2.6.4. 70S: REVOLUCIÓN DA PROGRAMACIÓN ESTRUTURADA



No ano 1972, **C** foi desenvolvida por Dennis Ritchie nos laboratorios Bell. A linguaxe foi creada como unha mellora sobre a linguaxe B. O obxectivo principal era crear unha linguaxe que permitise a **programación de sistemas operativos** e que fose portátil entre diferentes máquinas.

Un dos primeiros grandes logros de C foi que **o sistema operativo UNIX foi escrito nesta linguaxe**, orixinalmente escrito en Ensamblador. A reescrita en C permitiu que UNIX se convertese nun sistema operativo portátil e más fácil de manter.

The screenshot shows a window titled 'FIBONACI.C' in a Borland C++ IDE. The menu bar includes File, Edit, Search, Run, Compile, Debug, Project, Options, Window, and Help. The code in the editor is:

```
#include <stdio.h>
#include <conio.h>

int i, j, inpt;
ar[20];

main()
{
    clrscr();

    printf("Enter number (i to 20) ? ");
    scanf("%d", & inpt);

    ar[0] = ar[1] = 1;
    printf("\n1 1");

    for (i = 2; i <= inpt; i++)
    {
        ar[i] = ar[i-1] + ar[i-2];
        printf(" %d", ar[i]);
    }
}
```

The status bar at the bottom shows '1:1' and the keyboard shortcut 'F1 Help F2 Save F3 Open Alt-F9 Compile F9 Make F10 Menu'.

2.6.5. 80S: PROGRAMACIÓN ORIENTADA A OBXECTOS



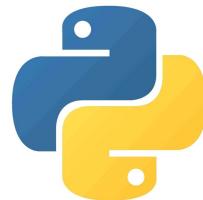
En 1983, publicouse **C++**. Esta **amplía a linguaxe C** con características de **programación orientada a obxectos** e outras melloras. Foi creada por Bjarne Stroustrup e, desde entón, converteuse nunha das linguaxes de programación más populares e utilizadas no mundo.

Actualmente é moi utilizada no desenvolvemento de videoxogos.

```
#include <iostream>
using namespace std;

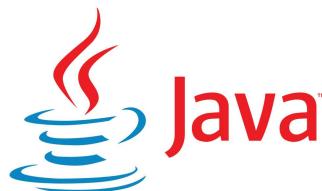
int main() {
    cout << "Hello World!";
    return 0;
}
```

2.6.6. 90S: INTERNET E A EXPLOSIÓN DAS LINGUAXES



Python é unha linguaxe de programación de **alto nivel, interpretada** e de propósito xeral, coñecida pola súa sinxeleza e facilidade de lectura. Foi creada por Guido van Rossum e lanzada por primeira vez en 1991. Python ten unha ampla gama de aplicacións, desde desenvolvemento web ata análise de datos e intelixencia artificial.

Python 3.0, lanzada en decembro de 2008, foi unha **revisión significativa da linguaxe**, deseñada para solucionar problemas fundamentais e mellorar a coherencia e sinxeleza.



Java foi desenvolvido por James Gosling, Mike Sheridan e Patrick Naughton no proxecto Green de Sun Microsystems. O proxecto comezou en 1991 con obxectivo de crear unha linguaxe que permitira escribir software para dispositivos electrónicos e outras plataformas e a súa primeira versión foi lanzada en 1995.

Esta é unha linguaxe de programación de **alto nivel, orientada a obxectos** e baseada en clases. É amplamente utilizada para desenvolvemento de software en diversas plataformas, incluíndo sistemas empresariais, aplicacións móveis e web.



PHP foi desenvolvido en 1995 por Rasmus Lerdorf. Este liberou o que chamou “Personal Home Page Tools”. A comunidade de desenvolvedores comezou a contribuír e mellorar o proxecto, e así naceu PHP como unha linguaxe de programación do lado do servidor, específica para o desenvolvemento web.

PHP, (*PHP: Hypertext Preprocessor*), é unha linguaxe de programación de propósito xeral amplamente **utilizada para o desenvolvemento web**. O seu deseño permite a creación dinámica de páxinas web e a interacción con bases de datos.

PHP é especialmente coñecido pola súa facilidade de uso e a súa integración eficiente con HTML.

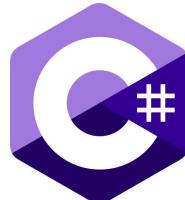


JavaScript é unha linguaxe de programación de **alto nivel, interpretada** e dinámica, que se utiliza principalmente para **desenvolver e mellorar páxinas web**. A pesar do seu nome, **non ten ningunha relación** coa linguaxe de programación **Java**.

A primeira versión pública foi incluída no navegador Netscape Navigator 2.0 en 1996.

JavaScript é unha das **linguaxes más importantes no desenvolvemento web moderno** e xogou un papel crucial na evolución da web interactiva e dinámica.

2.6.7. 2000S EN ADIANTE: MODERNIDADE E ESPECIALIZACIÓN



C# é unha linguaxe de programación desenvolvida por **Microsoft como parte da súa plataforma .NET**. Lanzada en 2000, C# foi deseñada para ser unha linguaxe moderna, segura e fácil de usar, cun enfoque na produtividade do desenvolvedor e na interoperabilidade con outras linguaxes e tecnoloxías.

A linguaxe combina características de linguaxes como C++, Java e Delphi, proporcionando unha base sólida para o desenvolvemento de aplicacións en diversos contextos.



Ruby é unha linguaxe de programación dinámica, orientada a obxectos e de alto nivel, que foi deseñada para ser simple e divertida de usar. Desenvolvida por Yukihiro Matsumoto, tamén coñecido como Matz.

Aínda que Ruby foi lanzada en 1995, Ruby **gañou a súa popularidade coa aparición de Ruby on Rails**, un framework de desenvolvemento web lanzado en 2004 por David Heinemeier Hansson. Rails proporcionou un enfoque de “convención sobre configuración” que facilitou o **desenvolvemento rápido e sinxelo de aplicacións web**.



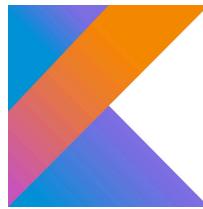
Go, tamén coñecida como Golang, é unha linguaxe de programación **desenvolvida por Google** e lanzada oficialmente en 2009.

Deseñada para ser simple, eficiente e con boa interoperabilidade, Go destaca polas súas características que facilitan o desenvolvemento de software de alto rendemento e a escalabilidade, especialmente en contornos de sistemas e redes.



Rust é unha linguaxe de programación moderna deseñada para ofrecer seguridade en memoria e rendemento elevado sen sacrificando a produtividade do desenvolvedor.

Desenvolvida por **Mozilla Research** e lanzada oficialmente en 2010, Rust es conocida por proporcionar garantías de seguridad en tiempo de compilación que evitan muchos errores comunes en lenguajes de bajo nivel, como errores de acceso a memoria e condiciones de carrera.



Kotlin é unha linguaxe de programación moderna desenvolvida por **JetBrains**, unha empresa coñecida por ferramentas de desenvolvemento de software como IntelliJ IDEA. Lanzada en 2011, Kotlin está deseñada para ser unha linguaxe concisa, segura e interoperable, proporcionando unha alternativa moderna e mellorada a Java.

En 2017, **Google anunciou que Kotlin sería unha linguaxe oficial para o desenvolvemento de aplicacóns**

Android. Isto incrementou significativamente a súa popularidade e adoptación na comunidade de desenvolvedores Android.



Swift é unha linguaxe de programación desenvolvida por **Apple** e lanzada oficialmente en 2014.

Deseñada para ser segura, rápida e fácil de usar, Swift é a linguaxe principal para o desenvolvemento de aplicacóns para **iOS, macOS, watchOS e tvOS**.

Substituí a Objective-C como a linguaxe preferida para o desenvolvemento de software na plataforma Apple,



TypeScript foi desenvolvido por Microsoft e lanzado por primeira vez en outubro de 2012. O principal impulsor do proxecto foi Anders Hejlsberg, coñecido tamén por ser o arquitecto principal de linguaxes como C# e Turbo Pascal.

O desenvolvemento de TypeScript naceu da **necesidade de mellorar a produtividade dos desenvolvedores ao traballar con JavaScript**, especialmente en proxectos de gran escala.

TypeScript es un superconjunto de JavaScript que engade tipado estático opcional ao linguaxe. Isto significa que todo código válido de JavaScript tamén é válido en TypeScript, pero con TypeScript pódense engadir tipos ás variables, funcións e obxectos, entre outras cousas.

Ademais, **TypeScript compíllase a JavaScript**, o que permite que se execute en calquera entorno onde JavaScript sexa soportado, como navegadores web ou servidores con Node.js.