TEMA 24

LENGUAJES DE PROGRAMACIÓN. TIPOS. CARACTERÍSTICAS.

ÍNDICE

INTRODUCCIÓN

Para que un ordenador funcione es necesario utilizar programas. El programa indica al ordenador qué tiene que hacer, y éste únicamente realiza aquellas operaciones que aquel incluye.

Un programa y sus sentencias se construyen o redactan con unos símbolos, y de acuerdo con unas reglas que constituyen la gramática del lenguaje de programación.

Un **lenguaje de programación** es un conjunto de símbolos y de reglas para combinarlos, que se usan para expresar algoritmos. Los lenguajes de programación, al igual que los lenguajes que usamos para comunicarnos, poseen un **léxico** (vocabulario o conjunto de símbolos permitidos), una **sintaxis** (reglas que indican cómo realizar las construcciones del lenguaje) y una **semántica** (reglas que permiten determinar el significado de las construcciones correctas del lenguaje).

Para que un ordenador pueda procesar un programa escrito en un determinado lenguaje de programación, es necesario realizar previamente una traducción del programa al lenguaje que entienda dicho ordenador, siguiendo una serie de fases.

1. TIPOS DE LENGUAJES DE PROGRAMACIÓN

1.1. Lenguaje máquina

Es el único lenguaje que entiende directamente el ordenador. Por esta razón, su estructura está totalmente adaptada a los circuitos de la máquina y muy alejada de la forma de expresión y análisis de los problemas propia de los humanos. Esto hace que la programación en este lenguaje resulte tediosa y complicada, requiriéndose un conocimiento profundo de la arquitectura física del ordenador. Frente a esto, el código máquina hace posible que el programador utilice la totalidad de recursos que ofrece el ordenador, obteniéndose programas muy eficientes (que aprovechan al máximo los recursos existentes) en tiempo de ejecución y en ocupación de memoria.

Las principales **características** del lenguaje máquina son las siguientes:

- Las instrucciones se expresan en el alfabeto binario (están codificadas en binario como cadenas de ceros y unos), pudiéndose utilizar códigos intermedios (octal y hexadecimal). Esta característica hace que un programa en lenguaje máquina sea difícil de entender y, como consecuencia, difícil de modificar.

- Los datos se referencian por medio de las direcciones de memoria donde se encuentran (no aparecen nombres de variables o de constantes).

- Las instrucciones realizan operaciones muy simples. El programador debe ingeniárselas para expresar cada una de las operaciones que desea realizar en función de las instrucciones elementales que dispone.

- Existe muy poca versatilidad para la redacción de las instrucciones, ya que, tienen un formato rígido en cuanto a la posición de los distintos campos (código de operación seguido de los campos dedicados a los operandos).

- El lenguaje máquina depende y está íntimamente ligado a la CPU del ordenador. Esto hace que los programas en dicho lenguaje no sean transferibles de un modelo de ordenador a otro, es decir, existe baja portabilidad.

- En un programa en lenguaje máquina no pueden incluirse comentarios que faciliten la legibilidad del mismo.

1.2. Lenguaje ensamblador

Constituye el primer intento de sustitución del lenguaje máquina por uno más cercano al usado por los humanos. Este acercamiento se plasma en las siguientes aportaciones:

- Uso de una *notación simbólica o nemotécnica* para representar los códigos de operación. De esta forma se evitan los códigos numéricos, tan difíciles de manejar. Normalmente, los códigos nemotécnicos están constituidos por abreviaturas de las operaciones en inglés. Así, por ejemplo, la suma se representa en la mayoría de los ensambladores por ADD.

- *Direccionamiento simbólico*. En lugar de utilizar direcciones binarias absolutas, los datos pueden identificarse con nombres.

- Se permite el *uso de comentarios* entre las líneas de instrucciones, haciendo posible la redacción de programas más legibles.

Aparte de esto, el lenguaje ensamblador presenta la mayoría de los inconvenientes del lenguaje máquina, como son su repertorio muy reducido de instrucciones, el rígido formato de las instrucciones, la baja portabilidad y la fuerte dependencia del hardware. Por otro lado, mantiene la ventaja del uso óptimo de los recursos hardware, permitiendo la obtención de un código muy eficiente.

Este tipo de lenguajes hacen corresponder a cada instrucción en ensamblador una instrucción en código máquina. Esta traducción es llevada a cabo por un programa traductor denominado **ensamblador**.

Para solventar en cierta medida la limitación que supone poseer un repertorio de instrucciones tan reducido, se han desarrollado unos ensambladores especiales denominados **macroensambladores**. Los lenguajes que traducen los macroensambladores disponen de macroinstrucciones cuya traducción da lugar a varias instrucciones máquina y no a una sola. Por ejemplo, existen macroinstrucciones para multiplicar, dividir, transferir bloques de memoria principal a disco, etc.

Dado que el lenguaje ensamblador está fuertemente condicionado por la arquitectura del ordenador que soporta, los programadores no suelen escribir programas de tamaño considerable en ensamblador. Más bien usan este lenguaje para afinar partes importantes de programas escritos en lenguajes de más alto nivel. El lenguaje ensamblador sigue siendo importante, ya que, da al programador el control total de la máquina, y como resultado genera un código compacto, rápido y eficiente.

Algunos ejemplos de instrucciones en ensamblador son:

```
MOV CX, 50      ; Poner el registro CX a 50

CMP CX, 0       ; ¿Es CX igual a cero?
```

1.3. Lenguajes de alto nivel

Los esfuerzos encaminados a hacer la labor de programación independiente de la máquina dieron como resultado la aparición de los lenguajes de programación de alto nivel. Estos lenguajes, más evolucionados, utilizan palabras y frases relativamente fáciles de entender y proporcionan también facilidades para expresar alteraciones del flujo de control de una forma bastante sencilla e intuitiva.

Las **características** fundamentales de los lenguajes de alto nivel son las siguientes:

- *Son independientes de la arquitectura física del ordenador*. Esto permite utilizar los mismos programas en ordenadores de arquitecturas diferentes (**portabilidad**) y, además, no es necesario conocer el hardware específico de la máquina.

- Normalmente, *una sentencia da lugar, al ser traducida, a varias instrucciones en lenguaje máquina.*

- Utilizan *notaciones cercanas a las habituales* en un determinado ámbito. Las operaciones se expresan con sentencias muy parecidas al lenguaje matemático o al lenguaje natural. Se utilizan, por lo general, palabras o términos en inglés.

La utilización de conceptos habituales suele implicar las siguientes cualidades:

a) Las instrucciones se expresan por medio de *texto*, conteniendo caracteres alfanuméricos y caracteres especiales.

b) Se puede asignar un *nombre simbólico* a determinados componentes del programa, para facilitar su comprensión. En los lenguajes imperativos, el programador puede *definir las*

variables que desee, dándoles los nombres que considere oportuno, y siendo las reglas para la denominación de las mismas poco restrictivas.

c) Se suelen incluir *instrucciones potentes de uso frecuente* que son ofrecidas por el lenguaje de programación. Por ejemplo, se suelen ofrecer funciones matemáticas de uso común (seno, coseno, etc.), operadores específicos de entrada/salida, operadores de tratamiento de cadenas de caracteres, etc.

d) Pueden incluirse *comentarios* en las líneas de instrucciones, o en líneas específicas de comentarios. Esto facilita la **legibilidad** de los programas, tanto para el propio programador, como para otras personas.

Como consecuencia de este alejamiento de la máquina y acercamiento a las personas, los programas escritos en lenguajes de programación no pueden ser interpretados directamente por el ordenador, siendo necesario realizar previamente su traducción a lenguaje máquina. Hay dos tipos de traductores, que vamos a considerar en la siguiente sección, los compiladores y los intérpretes.

2. TRADUCTORES: COMPILADORES E INTÉRPRETES

Un **traductor** es un metaprograma que toma como entrada un programa (o parte de un programa) escrito en lenguaje simbólico, alejado de la máquina, denominado **programa fuente** y proporciona como salida otro programa, semánticamente equivalente, escrito en un lenguaje comprensible por el hardware del ordenador, denominado **programa objeto**. Veremos dos tipos de traductores, los compiladores e intérpretes, que representan dos aproximaciones muy distintas a la tarea de permitir el funcionamiento de los programas escritos en un determinado lenguaje de programación de alto nivel.

Un **compilador** traduce completamente un programa fuente, escrito en un lenguaje de alto nivel, a un programa objeto, escrito en lenguaje ensamblador o máquina. El programa fuente suele estar contenido en un archivo, y el programa objeto puede almacenarse como archivo en memoria masiva para ser procesado posteriormente, sin necesidad de volver a realizar la traducción. Una vez traducido el programa, su ejecución es independiente del compilador, así, por ejemplo, cualquier interacción con el usuario sólo estará controlada por el sistema operativo. Como parte importante de este proceso de traducción, el compilador informa al usuario de la presencia de errores en el programa fuente, pasándose a crear el programa objeto sólo en el caso de que no hayan sido detectados errores (por lo general, suele cancelarse la compilación al detectar un error).

La traducción por un compilador (la **compilación**) consta de dos etapas fundamentales, que a veces no están claramente diferenciadas a lo largo del proceso: la etapa de análisis del programa fuente y la etapa de síntesis del programa objeto. Cada una de estas etapas conlleva la realización de varias fases. El análisis del texto fuente implica la realización de un análisis del léxico, de la sintaxis y de la semántica. La síntesis del programa objeto conduce a la generación de código y su optimización.

Un **intérprete** permite que un programa fuente escrito en un determinado lenguaje vaya traduciéndose y ejecutándose directamente, sentencia a sentencia, por el ordenador. El intérprete capta una sentencia fuente, la analiza e interpreta, dando lugar a su ejecución inmediata, no creándose, por tanto, un archivo o programa objeto almacenaje en memoria masiva para posteriores ejecuciones. La ejecución del programa estará supervisada por el intérprete.

En la práctica, el usuario crea un archivo con el programa fuente. Esto suele realizarse con un editor específico del propio intérprete del lenguaje. Según se van almacenado las instrucciones simbólicas, se analizan y se producen los mensajes de error correspondientes; así, el usuario puede proceder inmediatamente a su corrección. Una vez creado el archivo fuente, el usuario puede dar la orden de ejecución y el intérprete lo ejecuta línea a línea. Siempre el análisis antecede inmediatamente a la ejecución.

Si utilizamos un intérprete para traducir un programa, cada vez que necesitemos ejecutar el programa se volverá a analizar, ya que, no se genera un fichero objeto. En cambio, con un compilador, aunque sea más lenta, la traducción sólo debe realizarse una vez. Además los traductores no permiten realizar optimizaciones del código (que eliminan órdenes innecesarias compactando el código) más allá del contexto de cada sentencia del programa.

La principal ventaja de los intérpretes frente a los compiladores es que resulta más fácil localizar y corregir errores de los programas, ya que la ejecución de un programa bajo un intérprete puede interrumpirse en cualquier momento para conocer los valores de las distintas variables y la instrucción fuente que acaba de ejecutarse. Con un compilador esto no se puede realizar, salvo que el programa se ejecute bajo el control de un programa especial de ayuda denominado depurador ("**debugger**"). Por este

motivo, los intérpretes resultan más pedagógicos para aprender a programar, ya que el alumno puede detectar y corregir más fácilmente sus errores.

3. EL PROCESO DE COMPILACIÓN

Por ser los compiladores el tipo de traductor más utilizado en la actualidad, desarrollaremos el proceso de compilación, que consiste en la traducción de un programa fuente, escrito en lenguaje de alto nivel, a su correspondiente programa objeto, escrito en lenguaje máquina, dejándolo listo para la ejecución con poca o ninguna preparación adicional.

Antes del proceso de compilación, se crea el programa fuente utilizando cualquier aplicación disponible con capacidades de edición de textos.

Para llevar a cabo la compilación, el programa debe residir en memoria principal simultáneamente con el compilador. El resultado de la compilación puede dar lugar a la aparición de errores, en cuyo caso no se genera el programa objeto, sino que se realiza un **listado de compilación**, un informe indicando la naturaleza y situación de los errores detectados por el compilador en el programa fuente. Con el programa editor se corrigen y se comienza de nuevo el proceso.

Una vez finalizada la compilación y obtenido el programa objeto, se somete a un proceso de montaje donde se enlazan los distintos módulos que lo componen, en caso de tratar con programas que poseen subprogramas (que pueden ser compilados separadamente). Además, se incorporan las denominadas **rutinas de biblioteca** en caso de solicitarlas el propio programa. Este proceso de montaje es realizado por un programa denominado **montador** o **encuadernador**, que también recibe el nombre de *editor de enlace* o *linker*.

La compilación es un proceso complejo que consume a veces un tiempo muy superior a la propia ejecución del programa. Este proceso consta, en general, de dos etapas fundamentales: la etapa de **análisis** del programa fuente y la etapa de **síntesis** del programa objeto. Cada una de estas etapas conlleva la realización de varias fases.

El compilador utiliza internamente una **tabla de símbolos** para introducir determinados datos que necesita. Esta tabla interviene prácticamente en todas las fases del proceso de compilación; así mismo, el compilador posee un **módulo de tratamiento de errores** que permite determinarlas reacciones que se deben producir ante la aparición de cualquier tipo de error.

3.1. Análisis del programa fuente

Durante esta etapa, el programa fuente se divide en sus elementos componentes, creándose una representación intermedia del mismo. Durante este proceso, es posible detectar posibles errores en la escritura del programa.

3.1.1. Análisis lexicográfico

Se examina el programa fuente de izquierda a derecha, reconociendo las unidades básicas de información pertenecientes al lenguaje. Estas unidades básicas se denominan **unidades léxicas** o *tokens*. Un **token** es un elemento o cadena (secuencia de caracteres) que tiene un significado propio en el lenguaje.

Además de reconocer cada token, almacena en la tabla de símbolos aquella información del símbolo que pueda ser necesaria para las restantes fases de compilación.

En el caso de que no existan errores en este primer análisis, se obtiene una representación del programa formada por:

- La descripción de símbolos en la tabla de símbolos.
- Una secuencia de símbolos en la que se incluye, junto a cada símbolo, una referencia a la ubicación de dicho símbolo en la tabla de símbolos. Esta secuencia es denominada tira de tokens.

En esta nueva representación, se ha eliminado previamente toda la información superflua, como comentarios, tabulaciones y espacios en blanco no significativos.

Los errores que pueden ser reconocidos se deben a la detección de cadenas de caracteres del programa fuente que no se ajustan al patrón de ningún token, como identificadores de variables incorrectos, números expresados incorrectamente, cadenas de caracteres mal delimitadas, etc.

3.1.2. Análisis sintáctico

La sintaxis de un lenguaje de programación especifica cómo deben escribirse los programas basándose en un conjunto de reglas sintácticas que determinan la gramática del lenguaje. Un programa es sintácticamente correcto cuando sus estructuras (expresiones, sentencias, asignaciones, etc.) aparecen dispuestas de forma correcta de acuerdo a la gramática del lenguaje.

El análisis sintáctico recibe la tira de tokens del analizador lexicográfico e investiga en ella los posibles errores sintácticos que aparezcan, como expresiones estructuradas incorrectamente, falta de algún token dentro de una sentencia, formato incorrecto en una asignación, etc.

En caso de no encontrar errores, se agrupan los componentes léxicos en frases gramaticales que serán utilizadas en la síntesis del programa objeto. Estas frases gramaticales se suelen representar mediante un árbol sintáctico que pone de manifiesto la estructura jerárquica de los componentes de un programa.

3.1.3. Análisis semántico

Una construcción de un lenguaje de programación puede ser sintácticamente correcta y carecer de significado. Por tanto, si queremos generar código en lenguaje máquina con el mismo significado que el código fuente, tendremos que determinar la validez semántica de las construcciones del código fuente.

El análisis semántico se encarga de estudiar la coherencia semántica del código fuente a partir de la identificación de las construcciones sintácticas y de la información almacenada en la tabla de símbolos. El análisis semántico debe ser capaz de detectar construcciones *sin un significado correcto*. Por ejemplo, asignar a una variable de tipo entero un valor de tipo cadena de caracteres, sumar dos valores booleanos, etc.

3.2. Síntesis del programa objeto

Construye el programa objeto a partir de la representación obtenida en la etapa de análisis.

3.2.1. Generación de código intermedio

Se traduce el resultado de la etapa de análisis (en el caso de ausencia de errores) a un **lenguaje intermedio** propio del compilador. Esta representación intermedia (código intermedio) debe poseer las siguientes características:

- Debe ser independiente de la máquina.
- Debe ser fácil de producir.
- Debe ser fácil de traducir a un lenguaje máquina.

Esta fase aparece, generalmente, en aquellos compiladores construidos para su implantación en distintas máquinas, ya que su existencia hace que todos los módulos de análisis del compilador sean comunes para todas las máquinas (aumenta la portabilidad del compilador y se abaratan sus costes de construcción).

3.2.2. Optimización de código

Esta fase se suele introducir para mejorar la eficiencia del código producido por el compilador. Se toma el código intermedio de forma global y se optimiza, adaptándolo a las características del procesador al que va dirigido para mejorar los resultados de su ejecución.

La optimización se realiza en cualquier parte del programa, pero es en los bucles donde se consiguen los mejores resultados; por ejemplo, sacar una sentencia que se repite fuera del bucle, consiguiendo que sólo se ejecute una vez. De esta forma, se consigue un programa igualmente correcto, pero que consume menos tiempo de ejecución.

Un problema de la optimización es que incrementa el tamaño y la complejidad del compilador.

3.2.3. Generación de código objeto

Se traduce el código intermedio optimizado a código final, es decir, al lenguaje máquina (en algunos casos a ensamblador) del procesador al que va dirigido el compilador. Para ello, se reserva la memoria necesaria para cada elemento del código intermedio, y cada instrucción en lenguaje intermedio es traducida a varias instrucciones en lenguaje máquina.

3.3. Detección y tratamiento de errores

Aunque es en la etapa de análisis donde se detectan la mayoría de los errores, cada etapa del proceso puede encontrar errores. Para tratar adecuadamente un error, interviene el **módulo de tratamiento de errores** que comprende generalmente la realización de dos acciones:

- *Diagnóstico del error*: Trata de buscar su localización exacta y la posible causa del mismo, para ofrecer al programador un mensaje de diagnóstico, que será incluido en el listado de compilación.
- *Recuperación del error*: Después de detectado un error, cada fase debe tratarlo de alguna forma para poder continuar con la compilación y permitir la detección de más errores, aunque no se genere código objeto.

Los **tipos de errores** que puede tener un programa son los siguientes:

- *Errores lexicográficos*: Se producen por la aparición de tokens no reconocibles, es decir, cadenas que no se ajustan al patrón de ningún símbolo elemental del lenguaje.
- *Errores sintácticos*: Aparecen cuando se violan las reglas de sintaxis del lenguaje.
- *Errores semánticos*: Se detectan cuando aparece una secuencia sintácticamente correcta, pero que carece de sentido dentro del contexto del programa fuente.
- *Errores lógicos*: Son debidos a la utilización de un algoritmo o expresión incorrecta para el problema que se trata de resolver.
- *Errores de ejecución*: Son errores relacionados con desbordamientos, operaciones matemáticamente irresolubles, etc.

En ocasiones, se indican determinados errores que pueden existir pero que no perjudican al resto del proceso de compilación e incluso pueden permitir el funcionamiento del programa final. Estos mensajes de error se denominan **advertencias** o *warnings*. Un ejemplo típico puede ser la declaración de una variable que no se utiliza en ningún bloque del programa.

4. CLASIFICACIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

Los lenguajes de programación se pueden clasificar de acuerdo con diversos criterios. El criterio más simple que se puede considerar hace referencia a la **proximidad** del lenguaje con la máquina o con el lenguaje natural. De acuerdo con este criterio, existen tres niveles:

- *Lenguajes de bajo nivel*: Lenguajes máquina.
- *Lenguajes de nivel medio*: Lenguajes ensambladores y macroensambladores.
- *Lenguajes de alto nivel*: El resto de los lenguajes, a los que ya hemos hecho referencia.

Dado que los lenguajes de programación, en cierto modo, han tenido un desarrollo paralelo a la evolución de los ordenadores, se pueden clasificar atendiendo a su **desarrollo histórico**. Esta clasificación distingue cinco generaciones de lenguajes:

- *Primera generación*: Lenguajes máquina.
- *Segunda generación*: Ayudas a la programación, como los ensambladores.
- *Tercera generación*: Lenguajes de alto nivel imperativos, que siguen vigentes en la actualidad, como Pascal, Modula-2, Fortran, Cobol, C y Ada.
- *Cuarta generación*: Lenguajes o entornos de programación orientados básicamente a aplicaciones de gestión y bases de datos, como SQL, Natural, etc.
- *Quinta generación*: Lenguajes orientados a aplicaciones en Inteligencia Artificial, como Lisp y Prolog.

4.1. Clasificaciones de los lenguajes de alto nivel

De forma muy general, los lenguajes de alto nivel se pueden dividir en **lenguajes de propósito general**, que pueden ser empleados en cualquier tipo de aplicación y **lenguajes de propósito especial**.

Desde el punto de vista del campo de aplicación al que pertenece el lenguaje podemos considerar la siguiente clasificación:

Aplicaciones científicas: En las que predominan las operaciones numéricas o matriciales propias de algoritmos matemáticos. Lenguajes adecuados para estas aplicaciones son Fortran y Pascal.

Aplicaciones de procesamiento de datos: Donde son frecuentes las operaciones de creación, mantenimiento y consulta sobre ficheros y bases de datos. Dentro de este campo estarían aplicaciones de gestión empresarial, como programas de nóminas, contabilidad, facturación, control de inventario, etc. Lenguajes aptos para este tipo de aplicaciones son Cobol y SQL.

Aplicaciones de tratamiento de textos: Asociadas al manejo de textos en lenguaje natural. Lenguajes adecuados para ello son el Snobol y el C.

Aplicaciones en inteligencia artificial: Realización de programas que emulan el comportamiento inteligente: algoritmos de juegos, programas de comprensión del lenguaje natural, visión artificial, robótica y sistemas expertos. Los lenguajes que se suelen utilizar en estos casos son el Lisp y el Prolog.

Aplicaciones de programación de sistemas: Programación de módulos de sistemas operativos, compiladores, ensambladores, intérpretes y, en general, aquellos de interfaz entre el hardware y los usuarios. Tradicionalmente se utilizaba el lenguaje ensamblador, pero en la actualidad se muestran muy adecuados los lenguajes Ada, C y Modula-2.

Otra forma de clasificar los lenguajes de alto nivel tiene en cuenta el **estilo de programación** que fomentan, es decir, la filosofía de construcción de programas:

Lenguajes imperativos o procedurales. Estos lenguajes se fundamentan en el uso de variables para almacenar valores y el uso de instrucciones que indican las operaciones a realizar sobre los datos. La mayoría de los lenguajes de alto nivel son de este tipo.

Lenguajes declarativos. En este caso, el proceso por el cual se ejecuta el programa no aparece de forma explícita. Los programas se construyen mediante la definición de funciones (lenguajes funcionales como Lisp) o expresiones lógicas que indican las relaciones entre determinadas estructuras de datos (lenguajes de programación lógica como Prolog).

Lenguajes orientados a objetos. El diseño de los programas se centra más en los datos y su estructura. Los programas consisten en descripciones de unidades denominadas objetos que encapsulan los datos y las operaciones que actúan sobre ellos. Uno de los lenguajes más usados en esta filosofía es el C++.

Lenguajes orientados al problema. Este tipo de lenguajes están diseñados para problemas específicos, principalmente de gestión. Los programas están formados por sentencias que indican qué se quiere hacer. Generalmente, suelen ser generadores de aplicaciones que permiten cierta automatización de la tarea de desarrollo de software de gestión.

5. EJEMPLOS DE LENGUAJES DE ALTO NIVEL

5.1. Fortran

Fue desarrollado en 1954 por el equipo de John Backus bajo el control de IBM. El objetivo principal era la producción de un lenguaje que pudiera traducirse de forma eficaz a lenguaje máquina.

FORTRAN significa FORmula TRANslator y está considerado como el primer lenguaje de alto nivel. Alcanzó gran popularidad desde su primera versión en 1957. Se estandarizó y mejoró en 1966 y nuevamente en 1977 y en 1990.

Está diseñado para su uso en aplicaciones científicas y técnicas. Se caracteriza por su potencia en los cálculos matemáticos, pero está limitado en todo lo relativo al tratamiento de datos no numéricos. Por esta razón, no ha sido usado extensamente en el ámbito del microordenador, pero sigue siendo un lenguaje común en aplicaciones de investigación, ingeniería y educación.

5.2. Cobol

Su nombre proviene de la frase "Common Business Oriented Language" (lenguaje general para los negocios). Es el lenguaje más utilizado en aplicaciones de gestión y fue creado en 1960 por un comité patrocinado por el departamento de defensa de los Estados Unidos.

Ha experimentado diversas actualizaciones hasta su versión COBOL ANS-85. Se ganó una amplia aceptación como lenguaje estandarizado, siendo uno de los más populares.

Las características más interesantes de este lenguaje son: se asemeja al lenguaje natural (haciendo uso abundante del inglés sencillo), es autodocumentado y ofrece grandes facilidades en el manejo de ficheros, así como en la edición de informes escritos.

Entre sus inconvenientes están sus rígidas reglas de formato de escritura, la necesidad de escribir todos los elementos al máximo detalle, la extensión excesiva en sus sentencias y la inexistencia de funciones matemáticas.

5.3. Basic

Se diseñó en 1965 para proporcionar a los principiantes un lenguaje fácil de aprender (Beginner's All-purpose Symbolic Instruction Code).

El principal objetivo que se pretendía al diseñar BASIC era la facilidad de uso y aprendizaje, incluso a costa de la ineficiencia.

Sus principales aportaciones son las de ser un lenguaje interpretado e interactivo. Esto, unido a la popularización de los microordenadores, ha hecho que este lenguaje se extienda con rapidez e incluso que aparezcan gran diversidad de versiones que extienden y adaptan el lenguaje original, haciéndolo útil para aplicaciones técnicas y de gestión. La versión más moderna, Visual Basic, soporta las características y métodos orientados a objetos.

5.4. Pascal

El PASCAL recibe su nombre en honor al filósofo y matemático francés Blaise Pascal, que inventó la primera máquina mecánica para sumar. Este lenguaje fue desarrollado en 1970 por el matemático suizo Nicklaus Wirth. Los motivos fundamentales de su creación son, por un lado, proporcionar un lenguaje adecuado para la enseñanza de los conceptos y técnicas de programación, y por otra parte, desarrollar implementaciones del lenguaje que funcionen de forma fiable y eficiente sobre los ordenadores disponibles. Estos objetivos han sido alcanzados en gran medida y, además, con el tiempo ha llegado a ser un lenguaje muy utilizado en todo tipo de aplicaciones.

Está diseñado para ilustrar conceptos clave de programación, como los tipos de datos, programación estructurada y diseño descendente. Trata de proporcionar un mecanismo para implementar esos conceptos. Se ha convertido en el predecesor de otros lenguajes más modernos, como Modula-2 y Ada.

5.5. C

El lenguaje C fue creado en 1972 por Dennis Ritchie, que junto con Ken Thompson había diseñado anteriormente el sistema operativo Unix, y su intención era conseguir un lenguaje idóneo para la programación de sistemas que fuese independiente de la máquina. Desde entonces, tanto el Unix como el

C han tenido un enorme desarrollo y proliferación, hasta convertirse en un estándar industrial para el desarrollo de software.

Es un lenguaje moderno de propósito general que combina las características de un lenguaje de alto nivel (programación estructurada, tipos y estructuras de datos, recursividad, etc.) con una serie de características más propias de los lenguajes de más bajo nivel. Esta cualidad del C hace posible que se utilice la programación estructurada para resolver tareas de bajo nivel, obteniendo un código ejecutable veloz y eficiente. Debido a sus especiales características, muchas personas consideran al C como un lenguaje de nivel medio.

Se ha vuelto muy popular y es el lenguaje más utilizado entre los desarrolladores profesionales de software comercial. Además, es un lenguaje pequeño (pocas instrucciones) y conciso (no tiene instrucciones redundantes). El coste de un lenguaje tan potente y útil es que no es particularmente fácil de aprender. La programación segura y fiable en este lenguaje requiere un conocimiento bastante profundo del mismo.

5.6. C++

Es el sucesor del lenguaje C, fue desarrollado por Bjarne Stroustup en los laboratorios Bell a principios de la década de los 80.

Introduce la programación orientada a objetos en C. Los objetos proporcionan una forma completamente nueva de ver los programas, una nueva filosofía de programación.

Es un lenguaje muy poderoso y eficiente. Sin embargo, es aún más difícil de aprender que C. Dado que C es un subconjunto de C++, es necesario aprender todo acerca de C y luego asimilar la filosofía de la programación orientada a objetos y el uso que C++ hace de la misma. No obstante, cada vez más programadores de C adoptan C++.

5.7. Modula-2

A finales de los años 70, Nicklaus Wirth dirige el desarrollo del lenguaje MODULA-2, con la intención de incluir las necesidades de la programación de sistemas y dar respuesta a las críticas recibidas con respecto a las carencias de lenguaje Pascal.

Además de incluir las características del lenguaje Pascal, el nuevo lenguaje soluciona las principales carencias del mismo, como son la posibilidad de compilación separada, creación de bibliotecas, programación concurrente, mejora en el manejo de cadenas de caracteres, procedimientos de entrada/salida y de gestión de la memoria, etc. Además posee grandes facilidades para la programación de sistemas.

Este lenguaje también posee cualidades didácticas, por lo cual, ha sido ampliamente aceptado en la comunidad universitaria como herramienta idónea para la enseñanza de la programación.

5.8. Ada

Constituye el último intento de obtener un único lenguaje para todo tipo de aplicaciones e incluye los últimos avances en técnicas de programación. Su diseño fue encargado por el departamento de defensa de Estados Unidos y su estandarización fue publicada en 1983. El nombre de ADA se debe a Augusta Ada Byron, condesa de Lovelace, considerada la primera programadora de la historia.

Entre las características del lenguaje se encuentran la compilación separada, la programación concurrente, la programación estructurada, su buena mantenibilidad, características de tiempo real, etc. El principal inconveniente de este lenguaje es su gran extensión, que puede complicar su uso.

5.9. Lisp

Es un lenguaje de programación pequeño y conciso, diseñado en 1959 por John McCarthy en el MIT par el trabajo con inteligencia artificial. Este lenguaje toma su nombre del procesamiento de listas, en inglés LISt Processing.

Está pensado para resolver problemas de manipulación de símbolos, que son los elementos básicos de este lenguaje, y representan objetos arbitrarios del dominio de interés que se esté tratando.

El LISP es un lenguaje funcional, ya que, todo programa se puede ver como una función de alto nivel que se aplica sobre otras funciones de más bajo nivel para obtener determinados resultados. Para realizar operaciones elementales pueden utilizarse funciones de una biblioteca.

Por sus características, este lenguaje no se parece en nada a otros lenguajes de programación. A pesar de ello, es un lenguaje fácil de aprender y es el más común dentro de las aplicaciones en inteligencia artificial.

Un problema inicial fue que no se podía ejecutar eficientemente en muchos ordenadores, por lo que han aparecido una serie de terminales dedicados al Lisp, con hardware y software de bajo nivel diseñado para soportar este lenguaje de forma eficiente. En la actualidad, existen versiones estándar de Lisp, como Common Lisp y DG Common Lisp.

5.10. Prolog

El PROLOG (PROgramming Logic) es un lenguaje basado en la lógica, apropiado para un gran número de aplicaciones en bases de datos e inteligencia artificial. Ha sido el lenguaje más utilizado en Europa para tal propósito.

Permite al programador expresar una serie de tareas basándose en la descripción de los objetos que intervienen en la misma (hechos y reglas) y las relaciones lógicas que existen entre ellos (predicados), en lugar de hacerlo mediante un algoritmo. Lleva incorporada la programación de operaciones y todo el esfuerzo de programación consiste en especificar adecuadamente los hechos y las reglas para después establecer las preguntas que podrán ser inferidas de forma automática.

Frente al resto de los lenguajes empleados en sistemas expertos, permite desarrollarlos sin demasiados conocimientos de programación, ya que no requiere programar ningún algoritmo. Puede utilizarse en educación, para enseñar lógica y técnicas de resolución de problemas.

6. BIBLIOGRAFÍA

Alberto Prieto
Introducción a la Informática
Mc Graw-Hill, 2ª edición, 1997

Alfonso Ureña López
Fundamentos de Informática
Ra-ma, 1997