LENGUAJES DE PROGRAMACIÓN

**OBJETIVOS DEL CURSO**

Que el estudiante conozca y comprenda los Componentes Fundamentales de los Lenguajes de Programación y la forma en cómo se Diseñan e Implementan.

Que el estudiante conozca y comprenda los Dominios del Problema o Áreas de Aplicación en dónde se utilizan los diferentes Tipos de Lenguajes.

Que el estudiante conozca y comprenda las Características y Elementos Estructurales Básicos de las diversas Familias de Lenguajes de Programación como son los Lenguajes Estructurados en Bloques, Orientados a Objetos, Funcionales, Lógicos, de Definición y Manipulación de Datos, de Programación Concurrente, de Programación Paralela y Distribuida, etc..

Que el estudiante conozca algunos Ejemplos Específicos de las Familias de Lenguajes de Programación como son: FORTRAN, C, Pascal, Modula, Ada, Small Talk, C++, C#, Java, Lisp, Prolog, SQL, XML, MPI, Occam, etc.

Que el estudiante tenga lo conocimientos básicos con el fin de que sea capaz de identificar y elegir el Lenguaje de Programación adecuado al Dominio del Problema de una aplicación que tenga que desarrollar.

# EL ESTUDIO DE LOS LENGUAJES DE PROGRAMACIÓN

**Lenguaje de Programación.** Notación que se usa para describir Algoritmos y Estructuras de Datos y que se puede implementar en una Computadora.

**¿Por qué estudiar Lenguajes de Programación?**

**1.-*Mejorar la habilidad para desarrollar Algoritmos Eficaces.*** Las características especiales de un lenguaje pueden ser usadas para desarrollar Algoritmos de forma Elegante y Eficiente; pero sí se usan incorrectamente pueden generar Programas ineficientes o con Errores Lógicos difíciles de detectar. Un ejemplo es la *Recursión* que cuando se usa correctamente permite desarrollar Algoritmos Cortos y Eficientes, pero sí se usa de forma inadecuada puede causar que el tiempo de ejecución crezca enormemente.

**2.-*Mejorar el uso del Lenguaje de Programación disponible.*** Si se conoce y entiende cómo se implementan las características del lenguaje, se pueden escribir Programas más Eficientes.

**3.- *Ampliar las posibilidades de solución de un Problema.*** El Lenguaje es una herramienta para el desarrollo del pensamiento, pero también puede ser una limitante al no contar con una expresión para ciertas ideas. “La familiaridad con un único lenguaje de programación tiende a tener un efecto similar de restricción. Al buscar datos y estructuras de programa adecuados para la solución de un problema, uno tiende a pensar sólo en estructuras que son susceptibles de expresión inmediata en lenguajes con los que uno está familiarizado. Através del estudio de las construcciones suministrados por una amplia gama de lenguajes, así como de la manera en que se aplican estas construc­ciones, el programador aumenta su "vocabulario" de programación”.

**4. *Permitir una mejor elección del Lenguaje de Programación*.** El conocimiento de diversos lenguajes puede ayudar a la elección de un lenguaje que sea el adecuado para un proyecto particular, con lo cual se reduce el esfuerzo de codificación requerido. Por ejemplo, el desarrollo de aplicaciones de apoyo en la Toma de Decisiones, como en tareas de inteligencia artificial, se pueden escribir con más facilidad en LISP, ML o Prolog.

**5. *Facilitar el aprendizaje de un nuevo lenguaje*.** El conocimiento de los componentes estructurales de diversos lenguajes de programación y técnicas de ejecución permite aprender fácilmente un nuevo lenguaje de programación.

**6. *Facilitar el diseño de un nuevo lenguaje*.** “Todo programa tiene una *Interfaz* *de Usuario* que es, de hecho, una forma de lenguaje de programación”. La interfaz de usuario consiste en un conjunto de comandos y formatos de datos destinados a permitir que el usuario se comunique con el programa, aspectos que también se presentan en el diseño de un lenguaje de programación de propósito genera. Así el diseño de programas se simplifica si el programador está familiarizado con diversas estructuras y métodos de implementación de los lenguajes de programación.

# BREVE HISTORIA DE LOS LENGUAJES DE PROGRAMACIÓN

Los primeros lenguajes de alto nivel aparecieron en la década de los 50’s, el mas representativo fue FORTRAN. Ada, C, Pascal, Prolog y Smalltalk datan de los años setenta, C++ y ML de los años ochenta y por último Java data de los años noventa.

***Lenguajes basados en el cálculo numérico.*** Las computadoras más antiguas se construyeron de finales de los años treinta a principios de los cuarenta. Estas primeras máquinas estaban proyectadas para resolver problemas numéricos y se pensaba en ellas como en calculadoras electrónicas. Por tanto, los cálculos numéricos eran el área de aplicación dominante para estas primeras computadoras.

En el inicio de los años cincuenta comenzaron a aparecer notaciones simbólicas. Grace Hopper encabezó un grupo en Univac para desarrollar el lenguaje A-0, y John Backus desarrolló Speedcoding para la IBM 701. Ambos lenguajes se proyectaron para compilar expresiones aritméticas sencillas en un lenguaje de máquina ejecutable.

Un importante avance tuvo lugar de 1955 a 1957, cuando Backus encabezó un equipo para desarrollar FORTRAN, o FORmula TRANslator (traductor de fórmulas). Como en el caso de los primeros esfuerzos, los datos de FORTRAN se orientaban en tomo a cálculos numéricos, pero la meta era un lenguaje de programación que incluyera estructuras de control, condicionales y enunciados de entrada y de salida. Puesto que pocos pensaban que el lenguaje resultante pudiera competir con el lenguaje ensamblador codificado a mano, se concentraron todos los esfuerzos en su ejecución eficiente y se diseñaron diversos enunciados específicamente para la IBM 704.

FORTRAN tuvo un éxito extraordinario --tanto así que cambió para siempre la programación y probablemente preparó el escenario para su reemplazo final por otros lenguajes--. Se hizo una revisión de FORTRAN en 1958 (FORTRAN II) y otra más unos cuantos años después (FORTRAN IV). Casi todos los fabricantes implementaron una versión del lenguaje y reinaba el caos. Finalmente, en 1966, FORTRAN IV se convirtió en estándar con el nombre de FORTRAN 66 y ha sido actualizado dos veces desde entonces, a FORTRAN 77 y FORTRAN 90. Sin embargo, el número extremadamente grande de programas escritos en estos primeros dialectos ha sido causa de que las generaciones subsiguientes de traductores sean casi todas compatibles con estos antiguos programas e inhibe el uso de características modemas de programación.

La GAMM (la Sociedad Alemana de Matemáticas Aplicadas) organizó un comité para diseñar un lenguaje universal. En Estados Unidos, la Association for Computing Machinery (ACM; asociación para maquinaria de cómputo) también organizó un comité similar. Aunque inicialmente los europeos temían ser dominados por los estadounidenses, los comités se fusionaron. Bajo el liderazgo de Peter Naur, el comité desarrolló el Intemational Algorithmic Language (IAL; lenguaje algorítmico intemacional). Después se cambió de nombre por ALGOrithinic Language (ALGOL; lenguaje algorítmico). Una revisión tuvo lugar en 1960 y el ALGOL 60 (con una revisión menor en 1962) se convirtió en el lenguaje "académico" de cómputo estándar desde los años sesenta hasta principios de los setenta.

En tanto que una meta para el FORTRAN era la eficiencia en la IBM 704, ALGOL tenía objetivos muy distintos:

1. La notación de ALGOL debería aproximarse a las matemáticas normales.

2. ALGOL debería ser útil para la descripción de algoritmos.

3. Los programas en ALGOL deberían ser compilables a lenguaje de máquina.

4. ALGOL no debería estar ligado a una sola arquitectura de computadora.

Para lograr la independencia de ALGOL de las máquinas, no se incluyó entrada ni salida en el lenguaje; dejando a que se escribieran procedimientos especiales para estas operaciones en cada plataforma. Aunque esto permitió hacer programas independientes de un hardware particular, al incorporar los procedimientos de entrada y/o salida de cada implementación se volvían incompatibles.

Backus fue el editor de] informe de ALGOL que definía el lenguaje [BACKUS 1960]. Usó

una notación sintáctica comparable con el concepto de lenguaje libre de contexto desarrollado por Chomsky [CHOMSKY 1959]. Esto constituyó la introducción de la teoría formal de la gramática al mundo de los lenguajes de programación. A causa de su propio papel y el de Naur en el desarrollo de ALGOL, la notación se conoce ahora como BNF, o

Backus Naur Form (forma de Backus Naur). I

Como un ejemplo más de la influencia de ALGOL, Burroughs, un vendedor de computadoras que, más tarde, se fusionó con Sperry Univac para formar Unisys, descubrió la obra de un matemático polaco llamado Lukasiewicz. Él había desarrollado una técnica que permitía escribir expresiones matemáticas sin paréntesis (notación postfija), con un proceso eficiente de evaluación con base en pilas. Aunque no se trataba de un resultado matemático importante, esta técnica tuvo un efecto profundo sobre la teoría de compiladores. A través del uso de métodos basados en la técnica de Lukasiewicz, Burroughs desarrolló el hardware de computadora B5500 con base en una arquitectura de pilas y pronto tuvo un compilador de ALGOL mucho más rápido que cualquier compilador de FORTRAN.

El concepto de tipos definidos por el usuario se desarrolló en los años sesenta, y ni FORTRAN ni ALGOL disponían de esta clase de características. Simula 67, desarrollado por Nygaard y Dahl de Noruega, introdujo el concepto de clases en ALGOL. Esto le proporcionó a Stroustrup la idea para sus clases de C++ como una extensión de C más tarde en los años ochenta. Wirth desarrollo ALGOL W a mediados de la década de 1960 como una extensión de ALGOL. Este diseño tuvo sólo un éxito menor; sin embargo, entre 1968 y 1970 Wirth desarrolló Pascal., el cual se convirtió en el lenguaje de la ciencia de la computación en los años setenta.

Con la introducción de su nueva línea de computadoras 360 en 1963, la IBM desarrolló NPL (New Programming Language; nuevo lenguaje de programación), luego el nombre se cambió a MPPL (MultiPurpose Programming Language; lenguaje de programación de usos múltiples) y más tarde se acortó a sólo PL/I. El PL/I fusionó los atributos numéricos de FORTRAN con las características de programación para negocios de COBOL. PL/I alcanzó un éxito moderado en los años setenta, pero su uso actual ha disminuido al ser reemplazado por C y Ada. El subconjunto educativo. El BASIC se desarrolló para satisfacer las necesidades de cálculo numérico no científico, pero se ha extendido mucho más allá de su meta original.

***Lenguajes para negocios***. El procesamiento de datos de negocios fué uno de los primeros dominios de aplicación por desarrollar después de los cálculos numéricos. Grace Hopper encabezó un grupo en Univac para desarrollar FLOWMATIC en 1955. La meta era desarrollar aplicaciones de negocios usando una forma de texto parecido al inglés.

En 1959, el Departamento de Defensa de Estados Unidos. patrocinó una reunión para desarrollar el Common Business Language (CBL; lenguaje común para negocios), el cual habría de ser un lenguaje orientado a negocios que usara el inglés tanto como fuera posible para su notación. A causa de la divergencia en las actividades de muchas compañías, se foimó un Comité para desarrollar rápidamente el lenguaje. Aunque pensaban que estaban proyectando un lenguaje provisional, las especificaciones, publicadas en 1960, eran los diseños para el COBOL (COmmon Business Oriented Language; lenguaje común orientado a negocios). COBOL fue revisado en 1961 y 1962, estandarizado en 1968 y revisado una vez más en 1974 y 1978.

***Lenguajes para inteligencia artificial***. El interés en los lenguajes para inteligencia artificial se inició en los años cincuenta con el IPL (Information Processing Language; lenguaje de procesamiento de información) de la Rand Corporation. El IPL-V se conocía en forma bastante amplia, pero su uso estaba limitado por su diseño de bajo nivel. El avance importante ocurrió cuando John McCarthy, del MIT, diseñó LISP (LISt Processing; procesamiento de listas) para la IBM 704. LISP 1.5 se convirtió en la implementación "estándard" de LISP durante muchos años. Más recientemente, Scheme y Common LISP han continuado esa evolución

LISP fue proyectado como un lenguaje funcional para procesamiento de listas. El dominio ordinario de problemas para LISP comprendía la búsqueda. Como la mayoría de los programas en LISP desarrollaban un árbol de posibles trayectorias (como una lista enlazada) y luego recorría el árbol en busca del recorrido óptimo, pronto los juegos se convirtieron en un campo de pruebas natural para LISP.

En los años cincuenta Yngve en el MIT, desarrollo COMIT como una forma de manejar procesamiento de lenguajes naturales usando reglas BNF. Aquí el objetivo era el procesamiento de cadenas para la transformación de texto de un formato a otro. La traducción automática, donde se podían sustituir cadenas de símbolos por otras cadenas, era el dominio natural de aplicación.

En 1972 un grupo de los Laboratorios Bell de AT&T encabezado por Ralph Griswold inició el desarrollo de SNOBOL (StriNg Oriented symBOlic Languaje) con la meta de crear un lenguaje de procesamiento de cadenas para la manipulación de fórmulas y análisis de gráficas.

Alternativamente a LISP que fue proyectado para aplicaciones de procesamiento de listas para usos generales, se creó Prolog como un lenguaje para usos especiales cuya estructura básica de control y su estrategia de implementación se basaban en conceptos de lógica matemática.

***Lenguajes para sistemas***. Con el desarrollo del sistema operativo UNIX escrito principalmente en C durante los primeros años de la década de 1970, se demostró la eficacia de los lenguajes de alto nivel en esta área, derrivando la creencia de que se tenía que usar el de lenguaje ensamblador para lograr sistemas operativos eficientes.

# ATRIBUTOS DE UN BUEN LENGUAJE

**1*. Claridad, sencillez y unidad.*** Un lenguaje de programación proporciona a la vez un marco conceptual para pensar acerca de los algoritmos y un medio de expresar esos algoritmos. El lenguaje debe constituir una ayuda para el programador mucho antes de la etapa misma de codificación. Debe proveer un conjunto claro, sencillo y unificado de conceptos que se puedan usar como primitivas en el desarrollo de algoritmos. Para ello, es deseable contar con un número mínimo de conceptos distintos cuyas reglas de combinación sean tan sencillas y regulares como sea posible. Llamamos a este atributo *integridad conceptual*.

La sintaxis de un lenguaje afecta la facilidad con la que un programa se puede escribir, poner a prueba, y más tarde entender y modificar. La legibilidad de los programas en un lenguaje es aquí una cuestión medular. Muchos lenguajes contienen construcciones sintácticas que favorecen una lectura errónea al hacer que dos enunciados casi idénticos signifiquen cosas distintas. Un lenguaje debe tener la propiedad de que las construcciones que signifiquen cosas distintas se vean diferentes; es decir, las diferencias semánticas deberán reflejarse en la sintaxis del lenguaje.

**2. *Ortogonalidad.*** El término ortogonalidad se refiere al atributo de ser capaz de combinar varias características de un lenguaje en todas las combinaciones posibles, de manera que todas ellas tengan significado.

Un ejemplo se encuentra en los conceptos de tipos y funciones. Un tipo describe la estructura la estructura de los elementos de datos. Una función es un procedimiento por el que pasa un número finito de valores de parámetro y devuelve un único valor hacia el procedimiento que la invoca. En un lenguaje ortogonal, los tipos son independientes de las funciones, y no se aplican restricciones a los tipos de parámetros que pueden ser pasados o al tipo de valor que puede ser devuelto. Así, podríamos ser capaces de pasar una función a una función, y recibir una función de regreso.

Cuando las características de un lenguaje son ortogonales, entonces es más fácil aprender el lenguaje y escribir los programas porque hay menos excepciones y casos especiales que recordar. El aspecto negativo de la ortogonalidad es que un programa suele compilar sin errores a pesar de contener una combinación de características que son lógicamente incoherentes o cuya ejecución es en extremo ineficiente.

**3. *Naturalidad para la aplicación.*** Un lenguaje necesita una sintaxis que, al usarse correctamente, permita que la estructura del programa refleje la estructura lógica subyacente del algoritmo. Idealmente, deberá ser posible traducir directamente un diseño de programa de este tipo a enunciados de programa adecuados que reflejen la estructura del algoritmo.

El lenguaje deberá suministrar estructuras de datos, operaciones, estructuras de control y una sintaxis natural apropiada para el problema que se va a resolver. Un lenguaje particularmente adecuado para una cierta clase de aplicaciones puede simplificar grandemente la creación de programas individuales en esa área. Prolog, con su predisposición hacia propiedades deductivas, y C++, para diseño orientado a objetos, nos dan un ejemplo en este sentido.

**4. *Apoyo para la abstracción.*** Incluso con el lenguaje de programación más natural para una aplicación, siempre queda una brecha considerable entre las estructuras de datos y operaciones abstractas que caracterizan la solución de un problema y las estructuras de datos primitivos y operaciones particulares integradas en un lenguaje. Por ejemplo, C puede ser un lenguaje apropiado para construir un programa para organizar los horarios de clases en una universidad, pero las estructuras de datos abstractos de "estudiante", "sección de clase, instructor, salón de clases y las operaciones abstractas de asignación de un estudiante a una sección de clase, "organizar una sección de clase en un salón de clases," etc., que son naturales para la aplicación no son suministradas directamente por C.

**5. *Facilidad para verificar programas***. La confiabilidad de los programas escritos en un lenguaje es siempre una preocupación medular. Existen muchas técnicas para verificar que un programa ejecuta correctamente la función requerida. Se puede probar que un programa es correcto a través de un método formal de verificación, se puede poner a prueba ejecutándolo con los datos de entrada de prueba y comparando los resultados de salida con las especificaciones, etc. La sencillez de la estructura semántica y sintáctica es un aspecto primordial que tiende a simplificar la verificación de programas.

**6*. Entorno de programación.*** La existencia de un entorno de programación adecuado puede facilitar el trabajo con un lenguaje. Se podría incluir una larga lista de factores como parte del entorno de programación. La disponibilidad de una implementación confiable, eficiente y bien documentada del lenguaje debe encabezar la lista. Los editores especiales y paquetes de prueba hechos a la medida para el lenguaje pueden acelerar mucho la creación y puesta a prueba de programas. Los recursos para el mantenimiento y modificación de múltiples versiones de un programa pueden simplificar mucho el trabajo con programas grandes.

**7. *Portabilidad de programas*.** Un criterio importante para muchos proyectos de programación es el de la portabilidad de los programas resultantes de la computadora en la cual se desarrollaron hacia otros sistemas de computadoras. Un lenguaje que está ampliamente disponible y cuya definición es independiente de las características de una máquina particular constituye una base útil para la producción de programas transportables.

**8. *Costo de uso*.** El costo es un elemento importante en la evaluación de cualquier lenguaje de programación, pero son factibles diferentes medidas del mismo:

*(a) Costo de ejecución del programa*. En los inicios de la computación, era importante el diseño de compiladores que optiman, la asignación eficiente de registros y el diseño de mecanismos eficientes de apoyo al tiempo de ejecución. El costo de ejecución del programa, aunque siempre ha tenido cierta importancia en el diseño de lenguajes, es de importancia primordial para grandes programas de producción que se van a ejecutar con frecuencia.

*(b) Costo de traducci6n de programas*. Cuando un lenguaje como FORTRAN, C o Pascal se utiliza en proyectos de desarrollo donde los programas se compilan muchas veces cuando se están depurando, contar con una traducción (compilación) eficiente, más que una ejecución eficiente, puede ser muy importancia.

(c) *Costo de creación, prueba y uso de programas*. Un tercer aspecto del costo de un lenguaje de programación queda ejemplificado por el lenguaje SmalItalk. Para una cierta clase de problemas se puede diseñar, codificar, probar, modificar y usar una solución con una inversión mínima en tiempo y energía del programador. SmalItalk es económico en cuanto a que se minimizan el tiempo y esfuerzo totales que se invierten en la solución de un problema en la computadora.

(d) *Costo de mantenimiento de los programas*. Muchos estudios han demostrado que el costo más grande de un programa que se utilizará varios años, es su costo de mantenimiento. El mantenimiento incluye la reparación de los errores que se descubren después de que se comienza a usar el programa, los cambios que requiere el programa cuando se actualiza el hardware subyacente o el sistema operativo, y las extensiones o mejoras al programa que se requieren para satisfacer nuevas necesidades. Un lenguaje que facilita la modificación, reparación y extensión frecuente del programa por parte de diferentes programadores durante un periodo de varios años puede ser, a la larga, mucho menos costoso de usar que cualquier otro.

# DOMINIOS DE APLICACIÓN

El lenguaje apropiado frecuentemente, depende del dominio de la aplicación que resuelve el problema. Los lenguaje adecuados para diversos dominios de aplicación han evolucionado a lo largo de los últimos 30 años. Algunos de ellos se resumen en la tabla 1.1.

***De procesamiento de negocios.*** En este dominio, COBOL es todavía el lenguaje importante para aplicaciones de procesamiento de datos, aunque a veces se utilizan C y PL/I. Sin embargo, la necesidad de analizar los datos en tiempo real ha hecho que la hoja de cálculo haya reformado este dominio de aplicación. Mientras que en otra época le tomaba al programador varios meses elaborar un programa típico de planeación de negocios, en la actualidad un analista puede desarrollar muchas hojas de cálculo en unas pocas horas.

Los lenguajes de cuarta generación (4GL) se han apoderado también de parte de este mercado. Los 4GL son lenguajes adaptados para dominios específicos de aplicaciones de negocios y suministran típicamente una interfaz de programador con base en ventanas, fácil acceso a registros de bases de datos y capacidades especiales para generar formas de entrada y reportes. Muchas herramientas de Desarrollo Rápido de Aplicaciones (RAD) incorporan lenguajes 4GL. A ésta unión a veces se les nombra Lenguajes Visuales.

|  |  |  |  |
| --- | --- | --- | --- |
| Época | Aplicación | Lenguajes principales | Otros lenguajes |
| Años  Sesenta | Negocios | COBOL | Ensamblador |
| Científica | FORTRAN | ALGOL, BASIC, APL |
| Sistemas | Ensamblador | JOVIAL, Forth |
| IA | LISP | SNOBOL |
| Hoy | Negocios | COBOL., Hoja de Cálculo | C, PL/I, 4GL |
| Científica | FORTRAN, C, C++ | BASIC, Pascal |
| Sistemas | C, C++, Pascal, Ada, | BASIC, Modula |
| IA | LISP, Prolog |  |
| Edición | TeX, Postscript, Procesadores de Texto |  |
| Proceso | Shell de UNIX, TCL, PERL | Marvel |
| POO | C++, Java | SmalItalk, Eiffel, C# |

Tabla 1.1. Lenguajes para diversos dominios de aplicación.

***Científicas.*** Aunque el FORTRAN 77 enfrenta reto de lenguajes como C, el FORTRAN 90 es una actualización reciente del estándar del lenguaje, la cual incorpora muchas características que se encuentran en Ada y otros lenguajes.

***De Sistemas***. El C, desarrollado hacia finales de los años sesenta, y su nueva variante C++, imperan en este dominio de aplicación. El C proporciona una ejecución muy eficiente y permite al programador tener pleno acceso al sistema operativo y al hardware subyacente. Aunque destinado a esta área, el Ada nunca ha alcanzado su objetivo de convertirse en un lenguaje importante en este dominio. La programación en lenguaje ensamblador se ha vuelto anacrónica.

Con el advenimiento de procesadores baratos que gobiernan automóviles, hornos de microondas, juegos de video y relojes digitales, ha aumentado la necesidad de contar con lenguajes para tiempo real. C, Ada, Pascal y Java se suelen usar para este tipo de procesamiento en tiempo real.

***De IA***. Todavía se utiliza LISP, aunque las versiones modernas como Scheme y Common LISP han reemplazado al LISP 1.5 del MIT de principios de los años sesenta. Prolog ha desarrollado un buen número de seguidores. Ambos lenguajes son muy aptos para aplicaciones de "búsqueda".

***Edición.*** Los sistemas de procesamiento de texto tienen su propia sintaxis para mandatos de entrada y archivos de salida. La composición de los capítulos de un libros se hace utilizando el sistema de procesamiento de texto TEX y, capítulos se "compilaron" conforme se escriben para introducir referencias de figuras y tablas, colocar figuras y componer párrafos. El traductor TEX produce un programa en el lenguaje Postscript de descripción de páginas, el cual tiene sintaxis y semántica y se puede compilar por medio de un procesador adecuado como el que se encuentra en una impresora láser.

***De proceso.*** Durante los años sesenta el programador era el agente activo en cuanto al uso de una computadora. Para llevar a cabo una tarea, el programador escribía un comando apropiado que la computadora ejecutaba luego. Sin embargo, en la actualidad solemos usar un programa para controlar otro, por ejemplo, para hacer el respaldo de archivos todas las noches, para sincronizar el tiempo cada hora, para enviar una respuesta automática al correo electrónico que llega cuando estamos de vacaciones, para poner automáticamente a prueba un programa cada vez que compila satisfactoriamente, etc. Llamamos a estas actividades procesos, y existe un interés considerable en el desarrollo de lenguajes donde estos procesos se puedan especificar y luego traducir para ejecutarlos en forma automática.

Dentro de UNIX, el lenguaje de comandos de usuario se conoce como shell (concha) y a los programas se les llama guiones de shell. Estos guiones se pueden invocar siempre que ocurren ciertas, condiciones habilitadoras. Han aparecido varios lenguajes de guiones; tanto el TCL como el PERL se usan para fines parecidos. El archivo ".BAT" de MS-DOS es una forma sencilla de guión.

***Nuevos paradigmas.*** Siempre hay nuevos modelos de aplicación en estudio. El ML se ha utilizado en la investigación de lenguajes de programación para investigar la teoría de tipos. Aunque no es un lenguaje muy extendido en la industria, su popularidad va en aumento. Smalltalk es otro lenguaje importante. Aunque el uso comercial de Smalltalk no es muy grande, ha tenido un profundo efecto sobre el diseño de lenguajes. Muchas de las características orientadas a objetos de C++ y Ada tuvieron su origen en Smalltalk.

Los lenguajes para diversos dominios de aplicación son una fuente continua de nueva investigación y desarrollo. En la medida en que el conocimiento de las técnicas de compilación mejora, y conforme evoluciona el conocimiento de cómo construir sistemas complejos; constantemente se encuentran nuevos dominios de aplicación y se requieren lenguajes que satisfagan las necesidades de los mismos.

# EFECTOS DE LOS ENTORNOS SOBRE LOS LENGUAJES

El entorno externo que apoya la ejecución de un programa se conoce como *entorno operativo* u *objetivo*. El entorno en el cual un programa se diseña, se codifica, se pone a prueba y se depura es el *entorno anfitrión*, y puede ser diferente del entorno operativo en el cual se usará el programa.

Cuatro clases generales de entornos objetivo cubren casi todas las aplicaciones de programación: *de procesamiento por lotes, interactivo, de sistema incrustado* y *de programación* (como caso especial de entorno interactivo). Cada uno plantea distintos requerimientos sobre los lenguajes adaptados a esos entornos.

## Entornos de procesamiento por lotes

El primero y más simple entorno operativo se compone sólo de archivos externos de datos. Un programa toma un cierto conjunto de archivos de datos como entrada, procesa los datos y produce un conjunto de archivos de datos de salida; por ejemplo, un programa de nómina procesa dos archivos de entrada que contienen registros maestros de nómina y tiempos de periodos de paga semanales, y produce dos archivos de salida que contienen registros maestros actualizados y cheques de sueldo. Este entorno operativo se designa como de procesamiento por lotes porque los datos de entrada se reúnen en "lotes" de archivos y son procesados en lotes por el programa.

***Efectos sobre el diseño de lenguajes.*** La influencia del entorno se aprecia en cuatro áreas principales: *características de entrada/salida, características de manejo de errores y excepciones, recursos de regulación del tiempo* y *estructura de programas*.

En un,lenguaje proyectado para procesamiento por lotes, por lo común los archivos constituyen la base para casi toda la estructura de entrada/salida.

En un entorno de procesamiento por lotes, un error que termine la ejecución del programa es aceptable, debido a que se puede repetir la ejecución después de corregir el error. Además, en este entorno, no es posible la ayuda externa por parte del usuario para manejar o corregir el error de inmediato. Así pues, las facilidades del lenguaje para manejo de errores y excepciones enfatizan el manejo de éstos en el programa para que éste se pueda recuperar de casi todos los errores y continúe con el procesamiento sin suspender su ejecución.

La estructura de programa representativa de este entorno consiste en un programa principal y un conjunto de subprogramas. Esta estructura típica de programa está presente en FORTRAN, C y Pascal. Los subprogramas, en su totalidad, son procesados por el compilador y éste interactúa rara vez con el programador durante el proceso de compilación.

## Entornos interactivos

En un entorno interactivo, un programa interactúa durante su ejecución directamente con un usuario en una consola de visualización, enviando alternativamente salidas hacia ésta y recibiendo entradas desde el teclado o ratón. Son ejemplos los sistemas de procesamiento de texto, hojas de cálculo, juegos de video, sistemas de gestión de bases de datos y sistemas de instrucción asistida por computadora.

***Efectos sobre el diseño de lenguajes***. Las características de la entrada/salida interactiva son lo suficientemente diferentes de las operaciones ordinarias con archivos como para que casi todos los lenguajes proyectados para un entorno de procesamiento por lotes experimenten cierta dificultad para adaptarse a un entorno interactivo. El C, por ejemplo, incluye funciones para tener acceso a líneas de texto desde un archivo y otras funciones que alimentan directamente cada carácter conforme lo digita el usuario en una terminal. La introducción directa de texto desde una terminal en Pascal, sin embargo, suele ser muy engorrosa. Por esta razón, el C (y su derivado C++) se ha vuelto mucho más popular como lenguaje para escribir programas interactivos.

El manejo de errores en un entorno interactivo recibe un tratamiento diferente. Si se introducen mal los datos de entrada desde un teclado, el programa puede desplegar un mensaje de error y solicitar una corrección al usuario. Las características del lenguaje para manejar el error dentro del programa (por ejemplo, no tomándolo en cuenta e intentando continuar) tienen menos importancia. Sin embargo, la terminación del programa como respuesta a un error no es ordinariamente aceptable (a diferencia del procesamiento por lotes).

En un entorno interactivo el concepto de un programa principal suele estar ausente. En su lugar, el programa se compone de un conjunto de subprogramas y el usuario introduce el "programa principal" como una serie de comandos en la terminal. Así, la interacción con el usuario adopta la forma de una solicitud de un comando, seguida de la ejecución del mismo, seguida de la solicitud de otro comando y así sucesivamente. ML, LISP y Prolog exhiben este tipo de comportamiento.

## Entornos de sistemas incrustados

Un sistema de computadora que se usa para controlar parte de un sistema más grande como una planta industrial, una aeronave, una máquina herramienta, un automóvil o incluso un tostador doméstico se conoce como un *sistema de computadora incrustado*. Aquí el sistema de computadora es parte integral del sistema más grande, y la falla de este significa también comúnmente la falla del sistema mayor y por lo tanto la aplicación no puede fallar. La seguridad de funcionamiento y corrección son atributos primarios de los programas que se usan en estos dominios. Ada, C y C++ se usan extensamente para satisfacer algunos de los requerimientos especiales de los entornos de sistema incrustado.

***Efectos sobre el diseño de lenguajes***. Los programas escritos para sistemas incrustados suelen operar sin un sistema operativo subyacente y sin los archivos de entorno y dispositivos de E/S usuales. En vez de ello, el programa debe interactuar directamente con dispositivos de E/S distintos de los normales a través de procedimientos especiales que toman en cuenta las peculiaridades de cada dispositivo. Por esta razón, los lenguajes para sistemas incrustados suelen hacer mucho menos énfasis en archivos y operaciones de entrada/salida orientadas a archivos. El acceso a dispositivos especiales se suele facilitar a través de características del lenguaje que proporcionan acceso a registros particulares del hardware, localidades de memoria, manejadores de interrupciones o subprogramas escritos en lenguaje ensamblador u otros lenguajes de bajo nivel.

El manejo de errores en sistemas incrustados tiene una importancia medular. Ordinariamente, cada programa debe estar preparado para manejar todos los errores en forma interna, adoptando acciones apropiadas para recuperarse y continuar. La interrupción, excepto en el caso de una avería catastrófica del sistema, no suele ser una alternativa aceptable, y por lo común no hay un usuario en el entorno que pueda proporcionar una corrección interactiva del error. El manejo de errores debe ser capaz de dar cuenta de las faltas de los componentes del sistema, además de los tipos comunes de errores causados por valores de datos erróneos. En el Ada, esta preocupación por el manejo interno de errores se manifiesta en las extensas capacidades para el manejo de excepciones.

Los sistemas incrustados deben operar casi siempre en tiempo real; es decir, la operación del sistema mayor dentro del cual está incrustada la computadora requiere que el sistema de computadora sea capaz de responder a entradas y producir salidas dentro de intervalos de tiempo estrechamente restringidos. Por ejemplo, una computadora que controle el vuelo de un avión debe responder con rapidez a los cambios de altitud o velocidad. La operación de estos programas en tiempo real requiere capacidades del lenguaje para monitorear intervalos de tiempo, responder a demoras de más de una cierta duración (que pueden indicar la avería de un componente del sistema), e iniciar y concluir acciones en tiempos determinados.

Por último, un sistema de computadora incrustado suele ser un sistema distribuido, compuesto de más de una computadora. El programa que se ejecuta en un sistema de esta clase se compone por lo general de un conjunto de tareas que operan simultáneamente, donde cada una controla a una parte del sistema. El programa principal, si lo hay, existe sólo para iniciar la ejecución de las tareas. Una vez iniciadas, estas tareas se ejecutan por lo común en forma simultánea e indefinida, puesto que sólo deben concluir cuando falla el sistema completo o se apaga por alguna razón.

## Entornos de programación

Es el entorno en el cual los programas se crean y se ponen a prueba, y tiende a tener menos influencia sobre el diseño de lenguajes que el entorno operativo en el cual se espera que los programas se ejecuten. Sin embargo, se reconoce ampliamente que la producción de programas que operan de manera confiable y eficiente se simplifica mucho a través de un buen entorno de programación y de un lenguaje que permita el uso de buenas herramientas y prácticas de programación.

Un entorno de programación consiste primordialmente en un conjunto de herramientas de apoyo y un lenguaje de para invocarlas. Cada herramienta de apoyo es otro programa que el programador puede utilizar como ayuda durante una o más de las etapas de creación de un programa. Las herramientas típicas en un entorno de programación incluyen editores, depuradores, verificadores, generadores de datos de prueba e impresoras.

***Efectos sobre el diseño de lenguajes***. Los entornos de programación han afectado el diseño de los lenguajes principalmente en dos áreas importantes: las características que facilitan la compilación por separado y el ensamblado de un programa a partir de componentes, así como las características que ayudan a poner a prueba y depurar los programas.

*Compilación por separado*. Por lo común, en la construcción de cualquier programa grande es deseable que distintos programadores o grupos de programación proyecten, codifiquen y pongan a prueba partes del programa antes del ensamblado final de los componentes en un programa completo. Esto requiere que el lenguaje esté estructurado de manera que los subprogramas individuales u otras partes se puedan compilar y ejecutar por separado, sin las demás partes, y luego fusionarse sin cambio en el programa final.

La compilación por separado se dificulta por el hecho de que, al compilar un subprograma, el compilador puede necesitar información acerca de otros subprogramas u objetos de datos compartidos, como:

1. *La especificación del número, orden y tipo de parámetros* que espera cualquier subprograma llamado, permite al compilador comprobar si una invocación del subprograma externo es válida. También puede ser necesario conocer el lenguaje en el que el otro subprograma está codificado, para que el compilador pueda establecer la adecuada "secuencia de llamado" de instrucciones para transferir datos e información de control al subprograma externo durante la ejecución en la forma esperada por ese subprograma.

2. *La declaración de tipo de datos* para cualquier variable referida es necesaria para que el compilador pueda determinar la representación de almacenamiento de la variable externa, de modo que la referencia se pueda compilar usando la fórmula de acceso apropiada para la variable (por ejemplo, el desplazamiento correcto dentro del bloque de entorno común).

3. *La definición de un tipo de datos* que se define externamente pero se usa para declarar cualquier variable local dentro del subprograma se necesita para permitir al compilador asignar almacenamiento y computar fórmulas de acceso para datos locales.

Para suministrar esta información acerca de subprogramas compilados por separado, objetos de datos compartidos y definiciones de tipos, (1) el lenguaje puede requerir que la información se redeclare dentro del subprograma (como en FORTRAN), (2) puede prescribir un orden de compilación particular para requerir que la compilación de cada subprograma vaya precedida de la compilación de la especificación de todos los subprogramas invocados y datos compartidos (como en Ada y hasta cierto punto en Pascal), o (3) puede requerir la presencia de una biblioteca que contenga las especificaciones pertinentes durante la compilación, de modo que el compilador pueda recuperarlas según lo necesite (como en Ada y C++).

El término *compilación independiente* se usa por lo común para la opción (1). Cada subprograma se puede compilar de manera independiente sin información externa alguna, el subprograma es totalmente autosuficiente. La compilación independiente tiene la desventaja de que ordinariamente no hay manera de comprobar la congruencia de la información acerca de subprogramas y datos externos que se redeclaran en el subprograma. Si las declaraciones dentro del subprograma no concuerdan con la estructura real de los datos o subprograma externos, entonces aparece un error sutil en la etapa final de ensamblado que no habría sido detectado durante la puesta a prueba de las partes del programa compiladas en forma independiente.

Las opciones (2) y (3) requieren algún medio para dar o colocar en una biblioteca especificaciones de subprogramas, definiciones de tipos y entornos comunes antes de la compilación de un subprograma. Ordinariamente, es deseable permitir que se omita el cuerpo (variables y enunciados locales) de un subprograma y dar, únicamente la especificación. El cuerpo se puede compilar por separado, más tarde. En Ada, por ejemplo, todos los subprogramas, tareas, o paquetes se dividen en dos partes, una especificación y un cuerpo, las cuales se pueden compilar por separado o colocarse en una biblioteca, según se requiera, para permitir la compilación de otros subprogramas. Una llamada de subprograma hecha a un subprograma que todavía no ha sido compilado se conoce como un fragmento. Un subprograma que contiene fragmentos se puede ejecutar y, cuando se llega a un fragmento, la llamada causa que se imprima un mensaje de diagnóstico de sistema (o que se adopte otra acción) en vez de una invocación real del subprograma. Así pues, un subprograma compilado por separado se puede ejecutar para fines de prueba, a pesar de que no está todavía disponible el código para algunas de las rutinas que él invoca.

Otro aspecto de la compilación por separado que afecta el diseño de lenguajes es en cuanto al uso de nombres compartidos. Si varios grupos están escribiendo partes de un programa

grande, suele ser difícil asegurar que los nombres que usa cada grupo para subprogramas, entornos comunes y definiciones de tipos compartidas sean distintos. Un problema común es encontrar, durante el ensamblado del programa final completo, que varios subprogramas y otras unidades de programa tienen nombres iguales. Esto significa a menudo una revisión tediosa y lenta de código ya verificada. Los lenguajes emplean tres métodos para evitar este problema: 1

1. Todo nombre compartido, como en un enunciado externo en C, debe ser único, y es obligación del programador asegurar que así sea. Por ejemplo, los nombres que se usan dentro de los archivos #include del C estándar llevan ordinariamente "\_" como prefijo, de modo que los programadores deberán evitar nombres variables que comiencen con la línea de subrayado.

2. Los lenguajes suelen emplear reglas de definición de ámbito para ocultar nombres. Si un subprograma está contenido dentro de otro subprograma, los otros subprogramas compilados por separado sólo conocen los nombres del subprograma más exterior. Los lenguajes como Pascal, C y Ada emplean este mecanismo

3. Los nombres se pueden conocer agregando explícitamente sus definiciones desde una biblioteca externa. Éste es el mecanismo básico de herencia en lenguajes orientados a objetos. Al incluir en un subprograma una definición de clase externamente definida, se vuelven conocidos otros objetos que define esa clase, como en Ada y C++. En Ada, los nombres también pueden ser homónimos, de manera que varios objetos pueden tener el mismo nombre. En tanto el compilador pueda decidir a cuál objeto se hace efectivamente referencia, no es necesario hacer cambios en el programa de llamado.

**Puesta a prueba y depuración.** Casi todos los lenguajes contienen ciertas características que facilitan la puesta a prueba y la depuración de programas. Unos cuantos ejemplos representativos son:

1**. *Características para rastreo de ejecución***. Prolog, LISP y muchos otros lenguajes interactivos suministran características que permiten marcar enunciados y variables particulares para "rastrearlos" durante su ejecución. Siempre que se ejecuta un enunciado marcado o se asigna un nuevo valor a una variable marcada, la ejecución del programa se interrumpe y se invoca un subprograma de rastreo designado (que típicamente imprime información de depuración apropiada).

2. ***Puntos de interrupción***. En un entorno interactivo de programación, los lenguajes suelen suministrar una característica donde el programador puede especificar puntos del programa como puntos de interrupción. Cuando se alcanza un punto de interrupción durante la ejecución del programa, la misma se interrumpe y el control se traslada al programador en una terminal. El programador puede inspeccionar y modificar valores de variables y luego reiniciar el programa a partir del punto de interrupción.

3. ***Asertos.*** Un aserto es una expresión condicional que se inserta como un enunciado independiente en un programa, por ejemplo:

*aserto:* ( X > 0 y A = 1 ) ó ( X = 0 y A > B +10 )

El aserto expresa las relaciones que deben cumplirse entre los valores de las variables en ese punto del programa. Cuando el aserto se habilita, el compilador inserta código en el programa compilado para poner a prueba las condiciones expresadas. Durante la ejecución, si las condiciones no se cumplen, entonces la misma se interrumpe y se invoca un manejador de excepciones para imprimir un mensaje o adoptar otra acción. Se trata de un concepto sencillo que existe en varios lenguajes, entre ellos C++.

PROBLEMAS

1. Para un lenguaje que se utilice, evalúe las razones de su éxito de acuerdo con la lista de 8 atributos de un buen lenguaje.¿Se debería ampliar la lista?

2. Considere el lenguaje simple siguiente. a y b representan nombres de variables enteras. Cada enunciado puede tener un rótulo como prefijo. Los enunciados en el lenguaje incluyen:

a = b Asignar a a el valor de b

a = a + l Sumar 1 a a

a = a - 1 Restar 1 de a

si a = 0 entonces ir a L Si a = 0 transferir el control al enunciado L

si a > 0 entonces ir a L Si a > 0 transferir el control al enunciado L

ir a L Transferir el control al enunciado L

alto Detener la ejecución

Por ejemplo, el programa que computa a = a + b puede estar dado por:

L: a = a + 1

B = b - 1

si b > 0 entonces ir a L

alto

(a) Escriba los programas siguientes en este lenguaje:

(1) Dados a y b, computar X = a + b

(2) Dados a y b, computar X = a \* b

(3) Dados a, b, c y d, computar X = a \* b y Y = e \* d

(b) Exponga un conjunto mínimo de extensiones necesario para que este lenguaje sea fácil de usar. Tome en cuenta conceptos como subprogramas, enunciados nuevos, declaraciones, etcétera.

3. Las características del C permiten expresar el mismo significado de muchas maneras ¿Cuántos enunciados diferentes puede usted escribir en C que sumen 1 a la variable X, es decir, que sean equivalentes a X = X + 1 ? Analice las ventajas y desventajas de este aspecto del diseño del C.

4. Suponga que un nuevo diseño de lenguaje provee tres tipos básicos de datos: *entero, real* y de *carácter*. También proporciona la capacidad para declarar *arreglos* y *registros* de datos. Los arreglos tienen elementos del mismo tipo y los registros los tienen de tipos mixtos. Use el concepto de *ortogonalidad* para criticar estas dos variantes del nuevo diseño:

(a) Los elementos de arreglos y registros pueden ser de cualquiera de estos tipos básicos de datos y ellos mismos también pueden ser arreglos o registros (por ejemplo, un elemento de un registro puede ser un arreglo).

(b) Los elementos de arreglos y registros pueden ser del tipo entero o real. Los arreglos de caracteres se llaman cadenas y reciben un tratamiento especial. Los elementos de registros pueden ser del tipo de carácter. Los registros pueden tener arreglos o elementos, pero los arreglos no pueden tener registros como elementos. Los arreglos no pueden tener arreglos como elementos, pero se suministran arreglos multidimensionales para obtener el mismo efecto.