novática
Revista de la Asociación de Técnicos de Informática

***Novática***, revista fundada en 1975 y decana de la prensa informática española, es el órgano oficial de expresión y formación continua de **ATI** (Asociación de Técnicos de Informática), organización que edita también la revista **REICIS** (Revista Española de Innovación, Calidad e Ingeniería del Software).

**<http://www.ati.es/novatica/>**
**<http://www.ati.es/reicis/>**

**ATI** es miembro fundador de **CEPIS** (*Council of European Professional Informatics Societies*) y es representante de España en **IFIP** (*International Federation for Information Processing*); tiene un acuerdo de colaboración con **ACM** (*Association for Computing Machinery*), así como acuerdos de vinculación o colaboración con **AdaSpain**, **AI2**, **ASTIC**, **RITSI** e **Hispalinux**, junto a la que participa en **ProInnova**.



**Coordinación Editorial, Redacción Central y Redacción ATI Madrid**
Padilla 66, 3º, dcha., 28006 Madrid
Tlfn.914029391; fax.913093685 <novatica@ati.es>
**Composición, Edición y Redacción ATI Valencia**
Av. del Reino de Valencia 23, 46005 Valencia
Tlfn. 963740173 <novatica_prod@ati.es>
**Administración y Redacción ATI Cataluña**
Via Laietana 46, ppal. 1ª, 08003 Barcelona
Tlfn.934125235; fax 934127713 <secregen@ati.es>
**Redacción ATI Aragón**
Lagasca 9, 3-B, 50006 Zaragoza.
Tlfn./fax 976235181 <secreara@ati.es>
**Redacción ATI Andalucía** <secreand@ati.es>
**Redacción ATI Galicia**<secregal@ati.es>
**Suscripción y Ventas** <novatica.subscripciones@atinet.es>
**Publicidad** Padilla 66, 3º, dcha, 28006 Madrid.
Tlnf.914029391; fax.913093685 <novatica@ati.es>
**Imprenta:** Derra S.A., Juan de Austria 66, 08005 Barcelona.
**Depósito legal:** B 15.154-1975 -- ISSN: 0211-2124; CODEN NOVAEC
**Portada:** Lenguaje primario - Concha Arias Pérez / © ATI
**Diseño:** Fernando Agresta / © ATI 2003

## sumario

Nº 222, marzo-abril 2013, año XXXIX

**editorial**
**Una iniciativa de creación de empleo para los profesionales TIC** > 02

**en resumen**
**Estudiantes antiguos y jóvenes profesionales** > 02
*Llorenç Pagés Casas*

**noticias de IFIP**
**TC2: Grupos de trabajo y llamamiento a la participación** > 03
*Antonio Vallecillo Moreno*

**monografía**
**Lenguajes de programación**
Editores invitados: *Óscar Belmonte Fernández y Carlos Granell Canut*
**Presentación. Lenguajes de programación en perspectiva** > 04
*Óscar Belmonte Fernández, Carlos Granell Canut*
**Los lenguajes de programación en perspectiva** > 09
*Ricardo Peña Marí*
**La programación funcional** > 14
*Manuel Montenegro Montes*
**Estándares en la web** > 20
*Carlos Blé Jurado*
***Laudatio* a Antony R. Hoare** > 24
*Ricardo Peña Marí*
**Respuesta a la *Laudatio*** > 26
*Antony R. Hoare*

**secciones técnicas**

**Enseñanza Universitaria de la Informática**
**Vídeo-ejercicios didácticos para el aprendizaje de la programación** > 28
*Germán Moltó*

**Seguridad**
**Análisis de Bitcoin: Sistema P2P de pago digital descentralizado con moneda criptográfica virtual** > 34
*Javier Areitio Bertolín*

**Software Libre**
**Monitorización de PostgresSQL: Plugin para Pandora FMS** > 42
*Luis Caballero Cruz*

**Tecnologías para la Educación**
**Animaciones adaptativas de programas: una propuesta basada en estilos de aprendizaje** > 49
*Francisco Manso-González, Jaime Urquiza Fuentes, Estefanía Martín Barroso, Marta Gómez-Gómez*

**TIC y Turismo**
**Extracción automática de fichas de recursos turísticos de la web** > 55
*Iker Manterola Isasa, Xabier Saralegi Urizar, Sonia Bilbao Arechabala*
**Referencias autorizadas** > 60

**Sociedad de la Información**

**Privacidad y nuevas tecnologías**
**Privacidad y vigilancia: Una guía básica** > 67
*Aaron Martin*

**Programar es crear**
**El problema del CUIT (Competencia UTN-FRC 2012, problema D, enunciado)** > 74
*Julio Javier Castillo, Diego Javier Serrano, Marina Elizabeth Cárdenas*
**El problema del Buscaminas Cuadrado en 3D (Competencia UTN-FRC 2012, problema F, solución)** > 75
*Julio Javier Castillo, Diego Javier Serrano, Marina Elizabeth Cárdenas*

**Asuntos Interiores**
**Coordinación editorial / Programación de Novática / Socios Institucionales** > 77

**Tema del próximo número: "Minería de procesos"**

# Los lenguajes de programación en perspectiva

Ricardo Peña Marí
*Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid*

<ricardo@sip.ucm.es>

***Resumen:*** *En este trabajo se presenta una panorámica de los lenguajes de programación, a la vez desde el punto de vista histórico y desde el de su clasificación en las diferentes familias. En realidad, muchos aspectos de los lenguajes, tales como la concurrencia o la modularidad, son transversales a más de un familia, por lo que se comentan específicamente varios de esos aspectos y se toman ejemplos de diferentes lenguajes para ilustrarlos. Se cubren también lenguajes de propósito específico tales como los que tratan restricciones o los lenguajes de* "script"*. El artículo finaliza recapitulando los logros alcanzados por los lenguajes en su corta historia y tratando de extrapolar las tendencias futuras.*

***Palabras clave:*** *Lenguajes funcionales, lenguajes imperativos, lenguajes lógicos, lenguajes de* script*, modelos de cómputo, orientación a objetos, paradigmas de programación, programación concurrente, programación paralela, programación con restricciones, reutilización de código, sistemas de tipos.*

**Autor**

**Ricardo Peña Marí** es Catedrático del Departamento de Sistemas Informáticos y Computación de la Universidad Complutense de Madrid. Ha escrito un libro sobre el tema del artículo, "*De Euclides a Java: Historia de los algoritmos y de los lenguajes de programación*" (Nivola, 2006), en el que detalla los orígenes y el desarrollo de la Informática. También es autor del libro de texto "*Diseño de programas: formalismo y abstracción*" (Pearson, 2005) para estudiantes de Ingeniería Informática. Sus áreas de investigación son los lenguajes funcionales, el análisis estático de programas, y la generación de código con certificado. Es co-autor de más de cincuenta publicaciones revisadas por pares en revistas y congresos internacionales.

## 1. Orígenes de los lenguajes de programación y familias principales

Podemos definir algoritmo como *un conjunto de reglas, que aplicadas sistemáticamente a unos datos de entrada apropiados, resuelven un cierto problema en un número finito de pasos.* Los algoritmos han acompañado desde los tiempos históricos al desarrollo de las matemáticas, pues no otra cosa son los procedimientos para operar números en los diferentes sistemas de numeración, o los métodos para resolver sistemas de ecuaciones.

Sin embargo los lenguajes de programación (en adelante, LP), cuyo propósito es describir algoritmos, solo aparecen en el momento en que existe una máquina capaz de ejecutarlos. En ese sentido, el primer LP de la historia sería el conjunto de códigos utilizados en las tarjetas perforadas que gobernaban el telar de Jacquard (año 1800), que fue la primera máquina programable. Centrándonos en el siglo XX, e ignorando la pléyade de lenguajes ensambladores con que se programaban los primeros computadores electrónicos, nos ocuparemos aquí de los llamados *lenguajes de alto nivel*, a partir de Fortran (1957).

Aunque se han desarrollado miles de lenguajes, resulta relativamente fácil clasificarlos en familias para una mejor comprensión de sus analogías y diferencias. Así, se admiten tres familias principales:

- Lenguajes imperativos
- Lenguajes lógicos
- Lenguajes funcionales

Las dos últimas se agrupan bajo el nombre de lenguajes *declarativos*.

La primera es, sin duda, la más numerosa y la que incluye los lenguajes más utilizados en la práctica. A ella pertenecen los LP más populares a lo largo de la historia de la programación, tales como Fortran, Cobol, Pascal, C, C++ y Java.

La segunda contiene un solo lenguaje popular, Prolog, en sus diferentes versiones y con variados aditamentos. Existen muchos otros lenguajes en esta categoría, pero generalmente no han pasado del estadio de prototipos de investigación.

La tercera tiene varios representantes que han logrado amplias comunidades de usuarios, tales como Lisp y sus variantes, Erlang, Standard ML (SML), y Haskell.

Sin embargo, esta clasificación no es suficiente en la actualidad, ya que no recoge numerosas características de los LP que juegan un papel fundamental y que en cierto modo son transversales a dichas familias, en el sentido de que cualquiera de ellas puede incluir una o varias de estas características. Les llamaré *aspectos*, y una enumeración sin ningún orden especial podría ser la siguiente:

- Modularidad y ocultamiento
- Orientación a objetos
- Concurrencia, paralelismo y facilidades para la programación distribuida
- Polimorfismo, sobrecarga y genericidad
- Tratamiento de restricciones
- Creación dinámica de textos HTML o XML

En las siguientes líneas trataré de poner un poco de orden en esta enumeración, explicando en qué forma estos aspectos se han ido incorporando a los diferentes lenguajes. Repasando la historia de los LP, es fácil apreciar que aunque un aspecto aparezca por primera vez en un paradigma, y siempre que haya tenido éxito, luego se difunde a lenguajes de otros paradigmas.

## 2. Los modelos de cómputo

Este es el aspecto principal que determina la adscripción a una u otra familia. Desde la invención de las Máquinas de Turing, se han ideado numerosos mecanismos de cómputo, todos ellos con la misma potencia que aquellas, en el sentido de que permiten calcular *exactamente* las mismas funciones, o si se quiere, expresar los mismos algoritmos. De ellos, tres han sido adoptados por los lenguajes de programación:

En la familia imperativa, el cómputo se concibe como una *transformación incremental del estado*, representado por el valor que en cada instante tiene el conjunto de las variables del programa. La instrucción "estrella" es la de asignación, de la forma `x := exp`, donde una variable *x*, posiblemente subindicada, es modificada con el valor de una expresión. El resto de las instrucciones se dedican a indicar el orden preciso en que se realizan estas asignaciones y quizás a variar los subíndices. De esta forma el cómputo avanza modificando incrementalmente el estado hasta alcanzar el estado final deseado. Este modelo es el más cercano al hardware subyacente, un computador convencional, en el que el estado se corresponde con el contenido de la memoria y la instrucción de asignación se corresponde con el trasiego de datos entre la memoria y la unidad de proceso.

En los lenguajes lógicos, el paso elemental de

> "Desde la invención de las Máquinas de Turing, se han ideado numerosos mecanismos de cómputo, todos ellos con la misma potencia que aquellas, en el sentido de que permiten calcular *exactamente* las mismas funciones, o si se quiere, expresar los mismos algoritmos"

cómputo es la *unificación*, que en esencia es un mecanismo de deducción lógica. Supongamos la siguiente cláusula de un programa Prolog: `append ([X|Xs],Ys,[X|Zs]) :- append (Xs,Ys,Zs)` y un objetivo a resolver `append ([0,1],[2,3],Ws).`

La unificación del mismo con la cabeza de la cláusula, permite deducir el objetivo más sencillo `append ([1],[2,3],Zs)` y la sustitución `X = 0, Xs = [1], Ys = [2,3], Ws=[0|Zs].`

El cómputo progresa realizando nuevas unificaciones mientras queden objetivos por resolver y el resultado final, o bien es un fallo, o bien es una sustitución de variables por términos.

El primer caso representa una respuesta negativa a la pregunta inicial, mientras que el segundo representa una respuesta afirmativa siempre que las variables se sustituyan tal como indica la respuesta. Los programas lógicos pueden producir varias respuestas alternativas a una misma pregunta, y un mismo programa puede ser interrogado con diferentes preguntas. Es decir, los conceptos de entrada y salida de un programa no son los convencionales en otros paradigmas.

En los lenguajes funcionales, el paso elemental es la *β-reducción*, que consiste en reemplazar una parte izquierda por la correspondiente parte derecha de una definición, previa sustitución de los parámetros formales por los reales. Dada la definición en Haskell,

```
map f []     = []
map f (x:xs) = (f x) : (map f xs)
```

la expresión `3:(map (+1) (4:5:6:[]))` se reduce en un paso a `3:(4+1):(map (+1) (5:6:[]))`.

El cómputo prosigue aplicando β-reducciones hasta que no es posible reducir más la expresión de partida. El resultado es dicha expresión irreducible.

A diferencia de otros paradigmas, el funcional admite dos modelos alternativos de cómputo: el *impaciente* y el *perezoso*. El primero se corresponde con el mecanismo de paso por valor de la mayoría de los lenguajes imperativos: primero se reducen los parámetros y luego se llama a la función. En el segundo, los parámetros que sean expresiones se pasan sin evaluar a la función y solo se reducen si son requeridos por esta. Por ejemplo, si se tiene la definición `f x = 3` y `noterm` es una expresión cuya evaluación no termina, la llamada `f noterm` devuelve 3 con evaluación perezosa y no termina con impaciente.

Los lenguajes funcionales perezosos permiten definir listas infinitas, tal como la siguiente:

```
nats = 0 : (map (+1) nats)
```

Si se desarrollase esta definición de forma impaciente se obtendría la lista de los infinitos naturales.

Otro aspecto del modelo de cómputo funcional es el *orden superior*, es decir el hecho de que las funciones son tratadas como valores a todos los efectos: pueden ser parámetros o resultados de otras funciones, almacenadas en estructuras de datos, definidas dinámicamente, o utilizadas para construir dinámicamente otras más complejas. El paradigma imperativo admite un uso restringido de esta idea y el lógico no lo contempla en absoluto. Existen lenguajes lógico-funcionales que sí lo soportan, si bien se trata de lenguajes poco difundidos.

## 3. Encapsulamiento, modularidad y orientación a objetos

Los primeros LP incluían tan solo procedimientos y funciones con parámetros como único mecanismo de modularidad. Eso permite un cierto grado de abstracción ya que es posible utilizar una función sin conocer cómo está implementada.

La abstracción llegó algo más tarde a las representaciones de datos con la aparición del concepto de *tipo abstracto* de datos hacia 1974. Ello permite un tipo de módulo más interesante en el que se compilan a la vez varios procedimientos junto con la representación de un tipo complejo.

Por un lado, el módulo oculta dicha representación a sus usuarios y por otro permite la manipulación de variables de dicho tipo a través de los procedimientos del módulo. Lenguajes populares que incluyen este mecanismo son Modula-2 y Ada. Por ejemplo, podríamos definir un tipo abstracto *pila* con las operaciones habituales y el siguiente código de usuario:

```
type p: pila;
pilaVacia (p); apilar(p,3); ...
```

La *orientación a objetos* se inspiró en estas ideas para elaborar su noción de *clase*. Una clase define un tipo de *objetos*, cuya representación es privada, y un conjunto de *métodos* que pueden manipular objetos de dicho tipo.

El siguiente código utiliza una clase Pila en C++:

```
Pila p = Pila();      // asigna inicialmente a p la pila vacía
p.apilar(3); …        // apila en p un 3
```

La sintaxis elegida permite considerar implícitamente el objeto manipulado a la vez como parámetro de entrada y de salida del método llamado.

La orientación a objetos introdujo el concepto adicional de *herencia*, mediante el cual se pueden declarar versiones especializadas de una clase, llamadas subclases, de tal forma que no es necesario repetir en cada subclase las declaraciones de la clase de partida. Esta idea está ya presente en Simula-67 (1967), el lenguaje precursor del paradigma. La herencia necesita unas reglas adicionales de visibilidad según las cuales las declaraciones de una clase son privadas para sus usuarios pero son visibles para sus subclases.

El concepto de subclase se puede iterar y construir así amplias jerarquías de subclases a partir de una clase dada. De hecho las librerías de los LP orientados a objetos están estructuradas de este modo. Así, la herencia se convierte en un importante factor de *reutilización* de código, pues el programador puede tomar una clase ya existente y realizarle ligeros retoques (es decir construir una subclase) para obtener el componente que necesita. A la vez que se disminuye el esfuerzo de programación, también se aumenta la fiabilidad del código por el hecho de reutilizar componentes ya probados.

La mayoría de los LP modernos incluyen conceptos de *módulo* lo suficientemente generales para englobar una sola clase, varias relacionadas, una colección de procedimientos independientes, un conjunto de definiciones de constantes o/y de variables globales, o cualquier otra combinación. El programador puede seleccionar qué parte de esa información se hace visible a los usuarios del módulo.

## 4. Paradigmas propiciados por el hardware

> “La abstracción llegó algo más tarde a las representaciones de datos con la aparición del concepto de *tipo abstracto* de datos hacia 1974. Ello permite un tipo de módulo más interesante en el que se compilan a la vez varios procedimientos junto con la representación de un tipo complejo”

La gran disparidad entre la velocidad de la unidad de proceso y la de los periféricos condujo en los años 1960 a la invención de la *interrupción*, y con ella al paradigma de *programación concurrente*. El objetivo inicial era tener varios procesos activos en un mismo procesador, si bien el paradigma fue más tarde aplicable también a máquinas multiprocesador con memoria compartida, y a conjuntos de computadores comunicados por algún tipo de red.

Se tardó muchos años en comprender y dominar el fenómeno de la concurrencia. Los primeros mecanismos incorporados a los LP (semáforos, regiones críticas, monitores) presuponían la existencia de una memoria común. Los siguientes conceptos (mensajes asíncronos, mensajes síncronos, *rendezvous*) podían aplicarse indistintamente a memoria compartida o distribuida. El primer lenguaje de amplio uso en incorporar algunos de ellos fue Ada (1980).

Actualmente todos los LP imperativos modernos incorporan alguna noción de hebra concurrente y mecanismos para sincronizarlas y comunicarlas entre sí. Los lenguajes funcionales también tienen versiones concurrentes (Concurrent Haskell, Concurrent ML, etc...) que permiten este tipo de programación. Algunos de ellos, como Erlang, han nacido directamente con la pretensión de programar sistemas distribuidos con miles de procesos y docenas de máquinas para albergarlos, como suelen ser los sistemas de telefonía, o los programas que implementan las redes sociales en Internet.

La creciente popularización de máquinas multinúcleo ha dado lugar al mecanismo de *memoria transaccional*, que implementa las regiones críticas de una forma inspirada en los sistemas de bases de datos: se permiten varios procesos al tiempo en la región, pero solo uno de ellos tiene éxito en completarla. Los demás han de hacer *roll-back* de sus cambios (es decir, deshacerlos) y volver a intentarlo. Todo el mecanismo es transparente al programador y en promedio mejora el paralelismo de los programas.

A diferencia de la programación concurrente, la *programación paralela* tiene como objetivo la explotación al máximo de la capacidad de cálculo de un conjunto de procesadores, que puede ir de unas pocas decenas a muchos miles. Aquí los mecanismos de sincronización y comunicación se dan por resueltos y muchas veces quedan ocultos al programador. Los objetivos principales son repartir la carga del modo más uniforme posible, y disminuir el tiempo de proceso dedicado a comunicaciones.

Hay dos modelos básicos de cómputo, el *paralelismo de tareas* y el *paralelismo de datos*.

En el primero, el programador crea una estructura de procesos (p.e. esquemas divide y vencerás, en tubería, de trabajadores replicados, etc.) y no todas las tareas realizan el mismo trabajo, o en todo caso lo realizan asíncronamente.

En el segundo, el programador es responsable de distribuir los datos entre los procesadores, pero existe un solo código secuencial que es ejecutado síncronamente por todos ellos. En el modelo de paralelismo de datos el LP estándar es *High Performance Fortran (1993)*, una variante de Fortran 90 con directivas para la distribución de los datos. En el paradigma funcional, *Data Parallel Haskell*, también implementa dicho modelo. En el modelo de paralelismo de tareas hay dos estándares, PVM (*Parallel Virtual Machine*, 1990) y MPI (*Message Passing Interface*, 1993) de librerías de paso de mensajes.

El paralelismo, además de un paradigma con lenguajes propios, también es un *mecanismo de implementación* de los lenguajes, especialmente en el ámbito declarativo. En estos lenguajes, el programador no necesita especificar el orden de las acciones, por lo que el compilador tiene gran libertad para decidir dicho orden. Una posibilidad es ejecutar en paralelo partes del programa que no interfieran entre sí. Las oportunidades son muchas debido a la ausencia de efectos laterales de estos paradigmas.

## 5. Los sistemas de tipos en los lenguajes

Los tipos de los primeros lenguajes (enteros, coma flotante y matrices) se correspondían casi exactamente con los soportados por el hardware. Más adelante, se formaron dos corrientes de opinión con respecto a los tipos: los que defendían una *disciplina de tipos*, en la que toda variable se declaraba con un tipo y el compilador forzaba su uso de un modo compatible con el mismo, y los que defendían un lenguaje *sin tipos* en tiempo de compilación. A la primera pertenecen por ejemplo Algol-60, Pascal, Ada y Java. A la segunda, Lisp, Prolog y Erlang. Lenguajes como C y C++ se situarían en una posición intermedia entre ambas corrientes. Los defensores de la primera alegaban seguridad y los de la segunda, flexibilidad.

Los lenguajes funcionales a partir de SML (1980) encontraron un sistema de tipos, llamado *polimorfismo paramétrico*, que parece reunir lo mejor de ambos mundos. Por un lado, no es necesario declarar el tipo de las variables, o de los argumentos de las funciones, lo que elimina uno de los inconvenientes de la disciplina de tipos. Por otro, una variable o argumento puede tener muchos tipos, incluso todos los tipos posibles, lo que va en favor de la flexibilidad. Por ejemplo, el tipo de la función *map* definida más arriba es *(a->b)-[a]->[b]*, donde *a* y *b* representan "cualquier tipo" y no necesariamente el mismo. Lo más importante es que toda variable o función recibe un tipo en compilación, que es deducido por el compilador a partir de su definición, y el propio compilador vigila que cada entidad se usa de acuerdo con su tipo. Es decir, los defensores de la seguridad también quedan satisfechos.

Los lenguajes orientados a objetos soportan otro tipo de polimorfismo, llamado de *subtipos*, según el cual un objeto de una subclase es admisible en cualquier parte del texto donde sea admisible uno de la superclase. Con ello es posible tener algunas de las ventajas del polimorfismo paramétrico, e incluso otras adicionales como por ejemplo crear una lista con objetos de varios tipos distintos, aunque en este caso a costa de la seguridad o de introducir comprobaciones dinámicas.

Otros dos aspectos relacionados con los tipos son la sobrecarga y la genericidad definidas por el programador. Ambas se incorporan por primera vez en el lenguaje Ada, y lenguajes posteriores de diferentes paradigmas también las incluyen. Mediante la primera se puede dar el mismo nombre a operaciones con distinta implementación (un ejemplo muy usado es sobrecargar el operador = con distintas relaciones de orden), y mediante la segunda se pueden definir procedimientos o clases en los que algunos tipos, constantes y operaciones son parámetros que pueden ser

> "Otros dos aspectos relacionados con los tipos son la sobrecarga y la genericidad definidas por el programador. Ambas se incorporan por primera vez en el lenguaje Ada, y lenguajes posteriores de diferentes paradigmas también las incluyen"

variados de una concreción a otra. Eso permite por ejemplo crear algoritmos de ordenación o estructuras de datos muy generales que es posible concretar de formas distintas, incluso dentro del mismo programa, proporcionando distintos tipos de elementos y de relaciones de orden.

Con la suma de estas características (polimorfismo, subtipos, sobrecarga y genericidad), los sistemas de tipos han evolucionado de ser solo un mecanismo de seguridad, a convertirse además en un potente factor de reutilización de código. En efecto, cuanto más general sea el tipo de un componente, en más contextos puede ser aceptado como válido y por tanto más puede ser reutilizado.

## 6. La programación con restricciones

Existen lenguajes especializados para expresar y resolver problemas con restricciones desde los años 1970. Podríamos considerar estos lenguajes como un paradigma de *propósito específico*. Un problema de este tipo consta de los siguientes elementos:

- Un conjunto finito de variables incógnita $x_1,...,x_n$.
- Un conjunto de dominios $D_1,...D_n$ en los que toman valores dichas variables.
- Un conjunto de restricciones. Cada una representa una relación entre los valores admisibles de las variables involucradas.

Ejemplos de restricciones son expresiones que involucren variables, constantes y los operadores binarios $=, \neq, \leq, \geq$, u operadores n-arios tales como "todas distintas".

Ejemplos de dominios son los booleanos, los subrangos de los enteros (por ejemplo {0...9}), los enteros, los racionales o los reales.

Una solución al problema viene dada por una asignación de valores a las variables de forma que los valores pertenezcan a los dominios correspondientes y satisfagan todas las restricciones. A veces se busca una solución cualquiera, otras veces se buscan todas y otras se busca una que sea óptima en algún sentido. En este último caso, el programador ha de definir una función de coste cuyo valor hay que hacer máximo o mínimo.

Dos ejemplos de problemas, que se expresan de forma muy sucinta con restricciones, son la resolución de *sudokus* y la colocación de *n* reinas en un tablero de ajedrez $n \times n$ de forma que no se den jaque. La programación con restricciones es relevante en muchas áreas de gran importancia práctica, tales como la planificación de plantillas, de horarios, de tareas, problemas de transporte, de almacenamiento, etc. Los algoritmos de resolución están muy optimizados, y aunque la mayoría de los problemas que resuelven son NP-completos, en la práctica se comportan con razonable eficiencia.

A partir de 1987 aparecen los lenguajes lógicos con restricciones (*Constraint Logic Programming*, o CLP), en los que las restricciones están integradas de forma elegante. El programador puede usar predicados convencionales mezclados con restricciones y el sistema de soporte a la ejecución contiene un resolutor para las mismas. Actualmente todas las versiones de Prolog soportan distintos tipos de restricciones.

No hay nada que impida generar y resolver restricciones desde otros paradigmas, lo cual siempre podría hacerse invocando las librerías apropiadas. Sin embargo parece que la programación lógica ha conseguido integrarlas "sin costuras" como una característica más del propio lenguaje.

## 7. Los lenguajes de script

La aparición de Internet y la necesidad de un creciente trasiego de páginas HTML y XML entre servidores y clientes, desató toda una pléyade de nuevas tecnologías y lenguajes. Casi desde el principio hubo un deseo de dotar a dichas páginas de animación e interactividad, y de hecho eso propició la enorme difusión del lenguaje Java en los años 90, cuyo nacimiento estuvo asociado al del navegador de Netscape. Muy poco después surgieron los llamados lenguajes de *script*, lenguajes todo-terreno cuyo objetivo inicial era crear páginas HTML dinámicamente, pero a los que también se dotó de acceso a las órdenes del sistema operativo y de las bases de datos subyacentes con el fin de manipular ficheros, ejecutar programas, y obtener o modificar información, bien en el lado del servidor, o bien en el del cliente.

Aunque podríamos considerar estos lenguajes como de propósito especifico, de hecho engloban en su seno lenguajes de programación completos y se escriben en ellos aplicaciones de creciente complejidad. Por haber sido "los últimos" en llegar a la escena, han podido aprovecharse de todos los aciertos, y también de los errores, que se han dado en los LP convencionales. Y sin embargo ese no ha sido exactamente el caso. Tomaré como ejemplo JavaScript, aunque la mayoría de los comentarios son aplicables a otros lenguajes de *script*.

A pesar del nombre, JavaScript solo tiene en común con Java que nació un poco después del éxito de éste, y pareció una buena decisión comercial utilizar su nombre. Se trata de un lenguaje interpretado por los navegadores para ser ejecutado en el lado del cliente, y por tanto con acceso limitado a los recursos del computador que lo alberga.

Dejando aparte sus capacidades para crear formularios web y acceder a su contenido, su orientación como LP es un híbrido interesante entre la orientación a objetos y la programación funcional.

Se pueden crear fácilmente funciones de orden superior como *map* y *fold*, y también clausuras, es decir funciones que arrastran consigo el entorno local en el que fueron declaradas.

Estas características se pueden mezclar libremente con las típicas del paradigma imperativo (estado, acceso a variables no locales, es decir efectos laterales) y las de la orientación a objetos (herencia y polimorfismo de subtipos). El resultado es que se requiere bastante pericia y formación informática para manejar el lenguaje con fiabilidad.

Por si fuera poca complicación, no hay sistema de tipos estático, y el dinámico realiza conversiones implícitas para adaptarse al tipo de los operandos.

Por ejemplo, `"5"*5` devuelve `25`, pero `"a"*5` es un error. La instrucción **`if`** `(exp) {...}` **`else`** {…} ejecuta la rama del **`else`** si la expresión `exp` es de tipo `string` y resulta ser la cadena vacía, o es un puntero nulo, o es un valor indefinido o erróneo. En cambio, `e1 == e2` devuelve `true` si ambas expresiones están indefinidas. Hay otros muchos ejemplos en este y en otros lenguajes de *script* que horrorizarían a los defensores de la disciplina de tipos.

Parecería que en aras de la flexibilidad se ha vuelto a los tiempos de PL/I, lenguaje que la

> “Los programas son entidades endiabladamente complejas, pero a diferencia de otros productos de la ingeniería, son absolutamente predecibles y no sufren desgaste por el uso. Nuestra obligación es dominarlos e impedir que ellos nos dominen”

propia IBM reconoció como un gran error y cuyos programas fallidos propiciaron la llamada "crisis del software" a finales de los años 60.

Sin duda los lenguajes de *script* son enormemente útiles para el propósito que fueron concebidos, pero en mi opinión es una insentatez privarles de mecanismos de comprobación estática que han probado su utilidad en otros lenguajes, que pueden hacer la programación más predecible, más segura, y reducir largas y costosas depuraciones.

## 8. Balance y perspectivas

Cincuenta años de evolución de los lenguajes de programación han dado mucho de sí. El balance no puede ser más positivo. Se han acuñado muchos conceptos y mecanismos que hacen posible la creación de enormes programas, en muchos casos ejecutándose simultáneamente en diferentes máquinas e interaccionando entre sí, y no obstante con un número razonablemente bajo de errores. No podemos decir que el software esté en crisis actualmente, pero quizás corremos el riesgo de entrar en una fase de "optimismo desmesurado" como la que precedió a la crisis de los 60, si no aprendemos de los errores del pasado.

Resumiendo brevemente el panorama descrito en este artículo, diría que la evolución de los LP ha ido en las siguientes direcciones:

- **Un creciente nivel de abstracción**. Ejemplos de esto son los tipos definidos por el usuario, incluyendo en ellos los tipos abstractos y las clases, las funciones de orden superior, las construcciones para gobernar la concurrencia, etc. Cada una de estas construcciones acercan más al programador a su problema y le independizan del hardware subyacente.
- **Adopción de mecanismos de modularidad**, que permiten independizar unas partes del programa de otras y posibilitan repartir el trabajo entre varias personas.
- **Adopción de mecanismos de seguridad estática**. En efecto, los sistemas de tipos actuales protegen de errores triviales que de otro modo pasarían inadvertidos, y a la vez tienen la suficiente flexibilidad para no constreñir excesivamente al programador.
- **Potenciamiento de la reutilización**. Tanto los sistemas de tipos polimórficos, sean estos paramétricos o de subtipos, como la sobrecarga y la genericidad, la modularidad y el orden superior ya mencionados, inciden en potenciar la creación de componentes reutilizables, bien tal como han sido creados, bien previa una ligera adaptación a un nuevo contexto.

En mi opinión estas tendencias se van a mantener en el futuro porque han demostrado ser capaces de mantener bajo control los siempre crecientes complejidad y volumen de los sistemas software. Pero no se debe bajar la guardia y volver a escribir los programas de modo artesanal, como está ocurriendo parcialmente con las tecnologías asociadas a Internet.

Tratando de hacer alguna predicción futura, quisiera comentar en estas últimas líneas el papel que estimo jugarán los futuros compiladores. Hasta ahora estos se han ocupado fundamentalmente de garantizar la corrección sintáctica y de tipos de los programas, además de haber puesto un énfasis comprensible en generar un código lo más eficiente posible. Pero les aguardan tareas de mayor responsabilidad.

Los avances de los últimos años en las técnicas de análisis estático hacen posible que los compiladores deduzcan muchas propiedades útiles a partir del texto de un programa. Algunos ejemplos son:

- Demostrar la terminación de bucles
- Sintetizar invariantes para los mismos
- Estimar cotas superiores al coste en tiempo y en memoria de los programas
- Demostrar la ausencia de bloqueo de los programas concurrentes

Todas estas propiedades son indecidibles en general, pero cada año se encuentran nuevos algoritmos capaces de tratar con éxito más casos particulares. De este modo, una parte de la corrección podría ser garantizada por el compilador, y solo las partes más difíciles serían responsabilidad del programador. Aún en esos casos, el programador podría proporcionar ciertos asertos que ayudaran al compilador a completar la tarea.

La investigación en metodologías de programación, y en los lenguajes a los que ha dado lugar, ha sido tremendamente fructífera, y así debe seguir. Los programas son entidades endiabladamente complejas, pero a diferencia de otros productos de la ingeniería, son absolutamente predecibles y no sufren desgaste por el uso. Nuestra obligación es dominarlos e impedir que ellos nos dominen.

Vendrán nuevos métodos y nuevos lenguajes, probablemente más seguros y fiables, pero los retos y el volumen de nuestros programas también serán mayores. La batalla contra la complejidad seguirá activa por mucho tiempo.