Interpretación de programas de computadora

INTRODUCCIÓN

Los programadores expertos controlan la complejidad de sus diseños con las mismas técnicas generales utilizadas por los diseñadores de todos los sistemas complejos. Combinan elementos primitivos para formar objetos compuestos, abstraen objetos compuestos para formar bloques de construcción de nivel superior y conservan la modularidad adoptando vistas a gran escala de la estructura del sistema. Como programador, uno utiliza lenguajes para construir objetos de datos y procesos computacionales para modelar fenómenos complejos en el mundo real. Sin embargo, a medida que se enfrente a problemas cada vez más complejos, encontrará que cualquier lenguaje de programación fijo no es suficiente para sus necesidades. Deberá recurrir constantemente a nuevos lenguajes para expresar sus ideas de manera más eficaz. En el diseño de ingeniería, el establecimiento de nuevos lenguajes es una poderosa estrategia para controlar la complejidad; a menudo uno puede mejorar su capacidad para hacer frente a un problema complejo mediante la adopción de un nuevo lenguaje que le permita describir el problema de una forma diferente, utilizando primitivas, medios de combinación y medios de abstracción que sean particularmente adecuados para el problema en cuestión.

La programación abarca multitud de lenguajes. Hay lenguajes físicos, como los lenguajes de máquina para computadoras concretas. En estos lenguajes, la representación de datos y control se realiza en términos de bits de almacenamiento individuales e instrucciones de máquina primitivas. El programador de lenguaje de máquina erige sistemas y utilidades eficientes para llevar a cabo cálculos con recursos de hardware limitados. Los lenguajes de alto nivel, erigidos sobre un sustrato de lenguaje de máquina, ocultan la representación de datos como colecciones de bits y la representación de programas como secuencias de instrucciones primitivas. Estos lenguajes tienen medios de combinación y abstracción, como la definición de procedimientos, que son adecuados para la organización de sistemas a gran escala.

La abstracción metalingüística —el establecimiento de nuevos lenguajes— desempeña un papel importante en todas las ramas del diseño de ingeniería. Es particularmente importante para la programación informática, ya que en la programación no solo es posible formular nuevos lenguajes, sino que estos lenguajes también pueden ser implementados mediante la construcción de evaluadores. Un *evaluador* (o *intérprete)* para un lenguaje de programación es un procedimiento que, cuando se aplica a una expresión del lenguaje, realiza las acciones necesarias para evaluarla.

No es exagerado considerar esto como la idea más fundamental en la programación: *El evaluador, que determina el significado de las expresiones en un lenguaje de programación, no es más que otro programa*.

Suena obvio, ¿no? Pero las implicaciones son profundas. Si uno es un informático teórico, la idea del intérprete recuerda el descubrimiento de las limitaciones de los sistemas lógicos formales por parte de Gödel, el concepto de una computadora universal por Turing y la noción básica de la máquina de programa almacenado por von Neumann. Si uno es un programador, dominar la idea de un intérprete

es una fuente de gran poder. Provoca un cambio mental real, un cambio básico en la forma de pensar en la programación. Sin entender los intérpretes, todavía puede escribir programas; incluso puede ser un programador competente. Pero no puede ser un programador completo.

Hay tres razones por las que, como programador, uno debería aprender acerca de los intérpretes.

En primer lugar, en algún momento necesitará implementar sistemas que tal vez no sean intérpretes para lenguajes de propósito general, pero que igual son intérpretes. Casi todos los sistemas informáticos complejos con los que las personas interactúan de manera flexible —una herramienta computarizada de dibujo o un sistema de recuperación de información, por ejemplo— incluyen algún tipo de intérprete que estructura la interacción. Estos programas pueden incluir operaciones individuales complejas —sombrear una región en la pantalla o realizar una búsqueda en una base de datos—, pero es el intérprete el pegamento que le permite combinar operaciones individuales para formar patrones útiles. ¿Se puede utilizar el resultado de una operación como entrada para otra operación? ¿Se le puede poner un nombre a una secuencia de operaciones? ¿El nombre es local o global? ¿Se puede parametrizar una secuencia de operaciones y ponerles nombres a sus entradas? Y así sucesivamente. No importa cuán complejas y pulidas sean las operaciones individuales, a menudo es la calidad del pegamento la que determina más directamente la potencia del sistema. Es fácil encontrar ejemplos de programas con buenas operaciones individuales, pero pésimo pegamento.

En segundo lugar, incluso los programas que no son intérpretes tienen importantes piezas similares a intérpretes. Si se busca dentro de un sofisticado sistema de diseño asistido por computador, es probable que se encuentre, trabajando juntos, un lenguaje de reconocimiento geométrico, un intérprete de gráficos, un intérprete de control basado en reglas y un intérprete de lenguaje orientado a objetos. Una de las formas más poderosas de estructurar un programa complejo es como una colección de lenguajes, cada uno de los cuales proporciona una perspectiva diferente, una forma diferente de trabajar con los elementos del programa. Elegir el tipo de lenguaje adecuado para el propósito correcto y comprender las soluciones de compromiso involucradas en la implementación: de eso se trata el estudio de los intérpretes.

La tercera razón para aprender acerca de los intérpretes es que las técnicas de programación que involucran explícitamente la estructura del lenguaje son cada vez más importantes. El actual empeño en diseñar y manipular jerarquías de clases en sistemas orientados a objetos es solo un ejemplo de esta tendencia. Tal vez esta sea una consecuencia inevitable del hecho de que los programas son cada vez más complejos —pensar más explícitamente en los lenguajes puede ser la mejor herramienta para hacer frente a esta complejidad. Obsérvese de nuevo la idea básica: el intérprete en sí no es más que un programa. Pero ese programa está escrito en algún lenguaje, cuyo intérprete en sí mismo no es más que un programa escrito en algún lenguaje, cuyo intérprete en sí mismo... Tal vez toda la distinción entre programa y lenguaje de programación sea una idea engañosa, y los futuros programadores no se verán a sí mismos escribiendo programas en particular, sino creando nuevos lenguajes para cada nueva aplicación.

El dominio de los intérpretes no es fácil, y por una buena razón. El diseñador de lenguajes está un nivel más alejado del usuario final que el programador de aplicaciones normal. Al diseñar un programa de aplicación, uno piensa en las tareas específicas que se deben realizar y tiene en cuenta qué características incluir. Pero, al diseñar un lenguaje, se tienen en cuenta las distintas aplicaciones que las personas podrían querer implementar y las formas en que podrían implementarlas. ¿Debería el lenguaje tener un ámbito estático o dinámico, o una mezcla? ¿Debería tener herencia? ¿Debería pasar parámetros por referencia o por valor? Todo dependerá de cómo uno espere que se use su lenguaje, de qué tipos de programas deban ser fáciles de escribir y de cuáles uno pueda permitirse que sean más difíciles.

Además, los intérpretes son realmente programas *sutiles.* Un simple cambio en una línea de código en un intérprete puede tener un enorme impacto en el comportamiento del lenguaje resultante. Uno no podrá entender estos programas mirándolos superficialmente —muy pocas personas en el mundo pueden echar un vistazo a un nuevo intérprete y predecir cómo se comportará incluso con programas relativamente simples. Así que hay que estudiarlos. Mejor aún, *ejecutarlos* —son código que funciona. Lo ideal es interpretar algunas expresiones simples y, a continuación, otras más complejas. Agregar mensajes de error. Modificar los intérpretes. Que cada quien diseñe sus propias variaciones. Tratar de dominarlos realmente, no de obtener solo una vaga sensación de cómo funcionan.

Si uno hace esto, cambiará su visión sobre cómo escribe programas y su visión de sí mismo como programador. Llegará a verse a sí mismo como un diseñador de lenguajes en lugar de un mero usuario de lenguajes, como una persona que elige las reglas según las cuales se arman los lenguajes, en lugar de solo un seguidor de las reglas que otras personas han elegido.

En la última década, las aplicaciones y los servicios de información han entrado en la vida de las personas de todo el mundo de maneras que difícilmente parecían posibles en los años 90. Están impulsados por una colección cada vez mayor de lenguajes y *frameworks*, todos erigidos sobre una plataforma cada vez mayor de intérpretes.

¿Y si uno desea crear páginas web? En 1995, eso significaba dar formato a texto estático y gráficos, algo equivalente a crear un programa que correrían los navegadores ejecutando una sola sentencia `print`. Las páginas web dinámicas de hoy en día hacen pleno uso de lenguajes de *scripting* (otro nombre para los lenguajes interpretados) como JavaScript. Los programas escritos para correr en los navegadores pueden ser complejos e incluir llamadas asincrónicas a un servidor Web que normalmente ejecuta un programa en un *framework* completamente diferente, posiblemente con una gran cantidad de servicios, cada uno con su propio lenguaje individual.

O tal vez esté programando un clúster informático masivo para realizar búsquedas a escala global. Si es así, es posible que esté escribiendo sus programas utilizando el paradigma *map-reduce* de la programación funcional para librarse de tener que tratar explícitamente con los detalles de cómo se planifican los procesadores individuales.

O tal vez esté desarrollando nuevos algoritmos para redes de sensores y explorando el uso de la evaluación diferida para enfrentar mejor el paralelismo y la agregación de datos. O esté explorando sistemas de transformación como XSLT para controlar páginas web. O diseñando *frameworks* para remezclar secuencias multimedia. O...

¡Tantas aplicaciones nuevas! ¡Tantos lenguajes nuevos! ¡Tantos nuevos intérpretes! Como siempre, los programadores principiantes, incluso los *aprendices* más capaces, pueden salirse bien viendo cada nuevo *framework* por separado y trabajando dentro de su conjunto fijo de reglas. Pero la creación de nuevos *frameworks* requiere habilidades del *maestro*: entender los principios que fluyen a través de los lenguajes, apreciar qué características de cada lenguaje son las más adecuadas para qué tipo de aplicación, y saber cómo crear los intérpretes que dan vida a esos lenguajes.

1. EL CICLO DE VIDA DE LOS LENGUAJES DE PROGRAMACIÓN

Cada lenguaje de programación tiene un ciclo de vida, que tiene algunas similitudes con el conocido ciclo de vida del software. Primeramente, el lenguaje se *diseña* para cumplir con algún requisito. Se escribe una *especificación* formal o informal del lenguaje con el fin de comunicar el diseño a otras personas. A continuación, el lenguaje se implementa mediante procesadores de lenguaje. Inicialmente, se podría desarrollar un *prototipo* para que los programadores puedan probar el lenguaje rápidamente. Más adelante, se desarrollarán *compiladores* de alta calidad (de fuerza industrial) que permitan llevar a cabo una programación de aplicaciones realista.

Como sugiere el término, el ciclo de vida de los lenguajes de programación es un proceso iterativo. El diseño de un lenguaje es un esfuerzo altamente creativo y desafiante, y ningún diseñador hace un trabajo perfecto en el primer intento. La experiencia de especificar o implementar un nuevo lenguaje tiende a exponer irregularidades en el diseño. Los implementadores y programadores pueden descubrir defectos en la especificación, tales como la ambigüedad, la incompletitud o la inconsistencia. También pueden descubrir características desagradables del lenguaje en sí, las cuales hacen que el lenguaje sea excesivamente difícil de implementar de manera eficiente o que sea insatisfactorio para la programación.

En cualquier caso, el lenguaje podría tener que ser diseñado, especificado e implementado nuevamente, tal vez varias veces. Esto seguramente será costoso, es decir, consumirá mucho tiempo y dinero. Por lo tanto, es necesario planificar el ciclo de vida para minimizar los costos.

La Figura 1 ilustra un modelo de ciclo de vida muy recomendable. El diseño es seguido inmediatamente por la especificación. Esto es necesario para comunicar el diseño a los implementadores y programadores. Le sigue el desarrollo de un prototipo y, luego, el desarrollo de compiladores. La especificación, la creación de prototipos y el desarrollo del compilador tienen costos crecientes, por lo que tiene sentido ordenarlos de esta manera. El diseñador obtiene un *feedback* lo más rápido posible y el costoso desarrollo del compilador se deja para cuando el diseño del lenguaje se haya estabilizado.

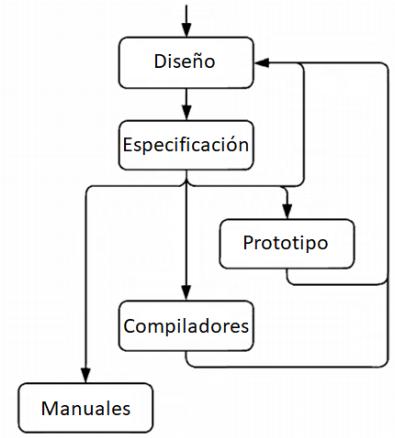


Figura 1. Un modelo de ciclo de vida de un lenguaje de programación

1.1. Diseño

La esencia del diseño de los lenguajes de programación es que el diseñador selecciona conceptos y decide cómo combinarlos. Esta selección está, por supuesto, determinada en gran medida por el uso previsto del lenguaje. Una variedad de conceptos se han incorporado en los lenguajes de programación: conceptos básicos como valores y tipos, almacenamiento, ligaciones y abstracción; y conceptos más avanzados como encapsulamiento, polimorfismo, excepciones y concurrencia. Un único lenguaje que admita todos estos conceptos es probable que sea muy grande y complejo (y sus implementaciones serán grandes, complejas y costosas). Por lo tanto, es necesaria una selección criteriosa de conceptos.

El diseñador debe buscar la simplicidad y la regularidad. La simplicidad implica que el lenguaje debe admitir solo los conceptos esenciales para las aplicaciones para las que está destinado. La regularidad implica que el lenguaje debe combinar estos conceptos de forma sistemática, evitando restricciones que puedan sorprender a los programadores o dificultar su tarea. Las irregularidades del lenguaje también tienden a dificultar la implementación.

Se ha descubierto una serie de principios que proporcionan una guía útil al diseñador:

- El *principio de completitud de tipos* sugiere que ninguna operación debería restringirse arbitrariamente en los tipos de sus operandos. Por ejemplo, las operaciones como la asignación y el paso de parámetros deberían, idealmente, ser aplicables a todos los tipos del lenguaje.
- El *principio de abstracción* sugiere que, para cada frase del programa que especifica algún tipo de cálculo, debería haber una manera de abstraer esa frase y parametrizarle las entidades con que opera. Por ejemplo, debería ser posible abstraer cualquier expresión para realizar una función o (en un lenguaje imperativo) abstraer cualquier comando para realizar un procedimiento.
- El *principio de correspondencia* sugiere que, para cada forma de declaración, debería haber un mecanismo de parámetros correspondiente. Por ejemplo, se debería poder transformar un bloque con una definición de constante en un procedimiento (o función) con un parámetro constante.

Estos son principios, no dogmas. Los diseñadores a menudo tienen que hacer compromisos (por ejemplo, para evitar construcciones que serían excesivamente difíciles de implementar). Pero al menos los principios ayudan al diseñador a tomar las decisiones de diseño difíciles de una manera racional y plenamente conscientes de sus consecuencias.

El propósito principal de este breve debate ha sido dar una idea de por qué el diseño de los lenguajes es tan difícil.

1.2. Especificación

Son varios los grupos de personas que tienen un interés directo en un lenguaje de programación: en primer lugar, el *diseñador* que inventó el lenguaje; los *implementadores*, cuya tarea es escribir los procesadores de lenguaje; y la comunidad de usuarios finales: los *programadores.* Todas estas personas deben apoyarse en una comprensión común del lenguaje, para lo cual es necesario que se refieran a una *especificación* acordada del mismo.

Se deben especificar varios aspectos de un lenguaje de programación:

- La incumbencia de la *sintaxis* es la forma de los programas. La sintaxis de un lenguaje define qué *tokens* (símbolos) se usan en los programas y cómo se componen las frases a partir de *tokens* y subfrases. Ejemplos de frases son los comandos, las expresiones, las declaraciones y los programas completos.
- Las *restricciones contextuales* (a veces denominadas *semánticas estáticas*) son reglas como las siguientes. Las *reglas de ámbito* determinan el ámbito (*scope*) de cada declaración y nos permiten ubicar la declaración de cada identificador. Las *reglas de tipo* nos permiten inferir el tipo de cada expresión y asegurarnos de que a cada operación se le proporcionan operandos de los tipos correctos. Las restricciones contextuales se llaman así porque el hecho de que una frase (tal como una expresión) esté bien formada depende de su contexto.
- A la *semántica* le incumben los significados de los programas. Hay varios puntos de vista sobre cómo se debe especificar la semántica. Desde un punto de vista, se puede tomar el significado de un programa como una función matemática, mapeando las entradas del programa a sus salidas. Esta es la base de la *semántica denotacional*. Desde otro punto de vista, es posible tomar el significado de un programa como su comportamiento cuando se ejecuta en una máquina. Esta es la base de la *semántica operacional*. Casi siempre, al estudiar procesadores de lenguaje, es decir, sistemas que ejecutan programas o los preparan para ser ejecutados, será preferible basarse en el punto de vista operacional.

Cuando se especifica un lenguaje de programación, existe la opción de hacerlo de una manera informal o una formal:

- Una *especificación informal* es una especificación escrita en castellano o en algún otro lenguaje natural. Tal especificación puede ser fácilmente entendida por cualquier usuario del lenguaje de programación, si está bien escrita. La experiencia demuestra, sin embargo, que es muy difícil hacer una especificación informal lo suficientemente precisa para todas las necesidades de los implementadores y los programadores; las interpretaciones erróneas son muy frecuentes. Incluso para el diseñador del lenguaje, una especificación informal es insatisfactoria porque fácilmente puede ser inconsistente o quedar incompleta.
- Una *especificación formal* es una especificación escrita en una notación precisa. Es más probable que tal especificación sea no ambigua, consistente y completa, y menos probable que se la malinterprete. Sin embargo, una especificación formal será inteligible solo para las personas que entienden la notación en la que se la escribe.

En la práctica, la mayoría de las especificaciones de los lenguajes de programación son híbridas. Casi todos los diseñadores de lenguajes especifican su *sintaxis* formalmente, utilizando BNF, EBNF o diagramas de sintaxis. Estos formalismos son ampliamente comprendidos y fáciles de usar. Algunos lenguajes más antiguos, como Fortran y Cobol, no tenían su sintaxis formalizada, y es digno de mención que su sintaxis es bastante irregular. La especificación formal de la sintaxis tiende a fomentar la simplicidad sintáctica y la regularidad, como se ve en Algol (el lenguaje para el que se inventó la BNF) y sus muchos sucesores. Por ejemplo, las primeras versiones de Fortran tenían varias clases de expresión diferentes, permitidas en diferentes contextos (asignación, indexación de matrices, parámetros de bucle); mientras que Algol desde el principio tenía solo una clase de expresión, permitida en todos los contextos.

Del mismo modo, la especificación formal de la *semántica* tiende a fomentar la simplicidad semántica y la regularidad. Desafortunadamente, pocos diseñadores de lenguajes todavía intentan esto. Los formalismos semánticos son mucho más difíciles de dominar que la BNF. Incluso si lo hacen, escribir una especificación semántica de un lenguaje de programación real (a diferencia de un lenguaje de juguete) es una tarea sustancial. Lo peor de todo es que el diseñador no tiene que especificar un lenguaje estable y bien entendido, sino uno que, poco a poco, se irá diseñando y rediseñando. La mayoría de los formalismos semánticos no son adecuados para satisfacer los requisitos de los diseñadores de lenguajes, por lo que no es de extrañar que casi todos ellos se contenten con escribir especificaciones semánticas informales. Por lo general, también las restricciones contextuales se especifican informalmente, porque su especificación formal es más difícil y las notaciones disponibles aún no se comprenden ampliamente.

Sin embargo, no se deben subestimar las ventajas de la formalidad y las desventajas de la informalidad. Las especificaciones informales tienen una fuerte tendencia a ser inconsistentes,

incompletas o ambas. Estos errores de especificación provocan confusión cuando el diseñador de un lenguaje busca comentarios de colegas, cuando se implementa el nuevo lenguaje y cuando los programadores intentan aprenderlo. Por supuesto, con una inversión suficiente de esfuerzo, la mayoría de los errores de especificación se pueden detectar y corregir, pero una especificación informal probablemente nunca estará completamente libre de errores. La misma cantidad de esfuerzo bien podría producir una especificación formal cuya precisión, al menos, está garantizada.

El acto mismo de escribir una especificación tiende a enfocar la mente del diseñador en aspectos del diseño que son incompletos o inconsistentes. Por lo tanto, el ejercicio de especificación proporciona comentarios valiosos y oportunos al diseñador. Una vez completado el diseño, la especificación (ya sea formal o informal) se utilizará para guiar las implementaciones posteriores del nuevo lenguaje.

1.3. Prototipos

Un prototipo es una implementación de bajo costo de un nuevo lenguaje de programación. El desarrollo de un prototipo ayuda a resaltar las características del lenguaje que son difíciles de implementar. Los prototipos también les ofrecen a los programadores oportunidades para probar tempranamente los lenguajes. Así, los diseñadores de lenguajes pueden obtener una retroalimentación valiosa. Además, dado que los prototipos se pueden desarrollar con relativa rapidez, este *feedback* es lo suficientemente oportuno como para hacer revisiones. Los prototipos podrían carecer de velocidad y de un buen reporte de errores, dado que estas cualidades se sacrifican en aras de una rápida implementación.

Para ciertos lenguajes de programación, un *intérprete* bien podría ser un prototipo útil. Un intérprete es mucho más fácil y rápido de implementar que un compilador para el mismo lenguaje. El inconveniente de un intérprete es que un programa interpretado se ejecutará tal vez 100 veces más despacio que un programa equivalente en código de máquina. Una vez que pasen la etapa de probar el lenguaje y comiencen a usarlo para construir aplicaciones reales, los programadores se cansarán rápidamente de esta enorme ineficiencia.

Una forma más duradera de prototipo es un *compilador interpretativo*. Consiste en un sistema que traduce programas escritos en cierto lenguaje de programación a algún código de máquina abstracto, junto con un intérprete para la máquina abstracta. El código objeto interpretado se ejecutará 'solo' unas 10 veces más despacio que un código de máquina concreto. Desarrollar este compilador y el intérprete juntos es mucho menos costoso que desarrollar un compilador que traduzca programas escritos en el lenguaje de alto nivel a un código de máquina real. De hecho, una máquina abstracta adecuada podría estar siempre disponible, lista para usar, ahorrando el costo de escribir el intérprete.

Otro método para desarrollar la implementación del prototipo es implementar un sistema que traduzca desde el nuevo lenguaje a un lenguaje de alto nivel existente. Tal traducción suele ser sencilla (siempre y cuando el lenguaje meta se elija con cuidado). Obviamente, el lenguaje meta existente ya debe

estar respaldado por una implementación adecuada. Este fue precisamente el método elegido para la primera implementación de C++, que utilizaba el traductor `cfront` para convertir el programa fuente a C.

El desarrollo del prototipo debe guiarse por la especificación del lenguaje, sea esta formal o informal. La especificación le indica al implementador qué programas están bien formados (es decir, se ajustan a la sintaxis del lenguaje y a las restricciones contextuales) y qué deberían hacer estos programas cuando se ejecutan.

1.4. Compiladores

Un prototipo no es adecuado para ser usado por un gran número de programadores en la construcción de aplicaciones reales durante un período prolongado. Cuando haya cumplido su propósito de permitirles a los programadores probar el nuevo lenguaje para proporcionarle comentarios al diseñador del lenguaje, el prototipo debería ser reemplazado por una implementación de mayor calidad. Esta será invariablemente un compilador o, más probablemente, una familia de compiladores, que generen código objeto para una serie de máquinas. Esta implementación de alta calidad se conoce como un *compilador de fuerza industrial.*

El trabajo que se dedicó al desarrollo de un prototipo no tiene por qué desperdiciarse. Si el prototipo era un compilador interpretativo, por ejemplo, se lo puede usar como punto de partida para crear un compilador que genere código de máquina.

El desarrollo de compiladores debe guiarse por la especificación del lenguaje. Un analizador sintáctico se puede desarrollar sistemáticamente a partir de la especificación sintáctica del lenguaje fuente. El desarrollo de un analizador contextual debe guiarse por una especificación de las reglas de ámbito y las reglas de tipo del lenguaje fuente. Por último, una especificación de la semántica del lenguaje fuente debe guiar el desarrollo de una especificación de código, que a su vez debe utilizarse para desarrollar sistemáticamente un generador de código.

En la práctica, las restricciones contextuales y la semántica rara vez se especifican formalmente. Si se comparan compiladores desarrollados por separado para el mismo lenguaje, a menudo se encontrará que son consistentes con respecto a la sintaxis, pero inconsistentes con respecto a las restricciones contextuales y la semántica. Esto no es un accidente, porque la sintaxis se suele especificar formalmente, y por lo tanto precisamente, y todo lo demás informalmente, lo que conduce inevitablemente a diferentes interpretaciones.

2. NIVELES DE LENGUAJE DE PROGRAMACIÓN

Los lenguajes de bajo nivel (*código de máquina, lenguaje ensamblador*) se llaman así porque obligan a que los algoritmos sean expresados en términos de instrucciones primitivas, del tipo que se pueden

ejecutar directamente en el hardware electrónico. Los lenguajes de alto nivel (*C*, *Python*, *Java*, *Clojure*) se llaman así porque permiten que los algoritmos se expresen en términos que están más cerca de la forma en que uno conceptualiza estos algoritmos en su cabeza. Los siguientes conceptos son típicos de los lenguajes de alto nivel, pero solo se admiten de forma rudimentaria (o directamente no se admiten en absoluto) en los lenguajes de bajo nivel:

- *Expresiones:* una expresión es una regla para calcular un valor. El programador que usa un lenguaje de alto nivel puede escribir expresiones similares a la notación matemática usual, utilizando operadores como ' + ', ' - ', ' * ' y ' / '.
- *Tipos de datos:* los programas manipulan datos de muchos tipos: tipos primitivos como valores de verdad, caracteres y enteros, y tipos compuestos como registros y matrices. El programador que usa un lenguaje de alto nivel puede definir tales tipos explícitamente y declarar constantes, variables, funciones y parámetros de estos tipos.
- *Estructuras de control*: Las estructuras de control permiten al programador que usa un lenguaje de alto nivel programar cómputos selectivos (por ejemplo, por medio de los comandos `if` y `case`) y cómputos iterativos (por ejemplo, mediante los comandos `while` y `for`).
- *Declaraciones*: Las declaraciones permiten al programador que usa un lenguaje de alto nivel utilizar identificadores para denotar entidades como valores constantes, variables, procedimientos, funciones y tipos.
- *Abstracción*: Una herramienta mental esencial del programador es la abstracción, o separación de incumbencias: separar la noción de *qué* cómputo se debe realizar de los detalles de *cómo* se va a realizar. El programador puede enfatizar esta separación mediante el uso de procedimientos y funciones con nombre. Además, se pueden parametrizar las entidades con que operan.
- *Encapsulamiento* (o *abstracción de datos*)*:* los paquetes y las clases le permiten al programador agrupar declaraciones relacionadas y ocultar selectivamente algunas de ellas. Un uso particularmente importante de este concepto es agrupar las variables ocultas junto a las operaciones con estas variables, lo cual es la esencia de la programación orientada a objetos.

2.1. Lenguajes de alto nivel

Incluso en los primeros días de la computación electrónica en la década de 1940 estaba claro que había una necesidad de herramientas de software para apoyar el proceso de programación. La programación se hacía en *código de máquina*, requería una habilidad considerable y era un trabajo duro, lento y propenso a errores. Se desarrollaron lenguajes ensambladores, lo que liberó al programador de tener que lidiar con gran parte de los detalles de bajo nivel, pero que requieren un software *ensamblador* para traducir de lenguaje ensamblador a código de máquina. Darle nombres simbólicos a las instrucciones, los valores, las ubicaciones de almacenamiento, los registros, etc. le permite al programador

concentrarse en la codificación de los algoritmos en lugar de hacerlo en los detalles de la representación binaria requerida por el hardware y ser, por lo tanto, más productivo. La *abstracción* proporcionada por el lenguaje ensamblador le permite al programador ignorar los detalles necesarios para interactuar directamente con el hardware.

El desarrollo de lenguajes de alto nivel se aceleró a partir de la década de 1950. Paralelamente, surgió la necesidad de contar con compiladores y otras herramientas para la implementación de estos lenguajes. Se reconoció la importancia de las especificaciones formales de los lenguajes y se entendió la correspondencia entre determinados tipos de gramática y la posibilidad de la implementación directa. El uso extensivo de lenguajes de alto nivel motivó el rápido desarrollo de una amplia gama de nuevos lenguajes, algunos diseñados para áreas de aplicación específicas, como el COBOL para aplicaciones empresariales y el FORTRAN para el cálculo numérico. Otros como PL/I (entonces llamado NPL) trataron de ser mucho más de propósito general. Muchos equipos desarrollaron compiladores para estos lenguajes en una época en que las arquitecturas de las máquinas de destino también estaban cambiando rápidamente.

2.1.1. Ventajas de los lenguajes de alto nivel

Las dificultades de la programación en lenguajes de bajo nivel son fáciles de ver y es obvia la necesidad de contar con lenguajes más fáciles de usar. Se requiere una notación de programación mucho más cercana a la especificación del problema. Se necesitan abstracciones de nivel superior para que el programador pueda concentrarse más en el problema que en los detalles de la implementación de la solución.

Los lenguajes de alto nivel pueden ofrecer tales abstracciones, y ofrecen muchas ventajas potenciales sobre los lenguajes de bajo nivel, incluyendo:

- La resolución de problemas es significativamente más rápida. Pasar de la especificación del problema al código es más sencillo usando un lenguaje de alto nivel. Depurar código en un lenguaje de alto nivel es mucho más fácil. Algunos lenguajes de alto nivel son adecuados para la creación rápida de prototipos, por lo que es particularmente fácil probar nuevas ideas y agregar código de depuración.
- Los programas escritos en lenguajes de alto nivel son generalmente más fáciles de leer, entender y, por lo tanto, mantener. El mantenimiento del código es ahora una gran industria donde los programadores modifican código que es poco probable que haya sido escrito por ellos mismos. Los programas escritos en lenguajes de alto nivel pueden ser, hasta cierto punto, autodocumentados, reduciendo la necesidad de comentarios extensos y documentación separada. El lector no se ve abrumado por el detalle necesario para entender los programas escritos en lenguajes de bajo nivel.
- Los lenguajes de alto nivel son más fáciles de aprender.

- Los programas escritos en lenguajes de alto nivel se pueden estructurar para reflejar más fácilmente la estructura del problema original. La mayoría de los lenguajes de alto nivel actuales admiten una amplia gama de características de estructuración de programas y datos, como la orientación a objetos, la compatibilidad con procesos asincrónicos y el paralelismo.
- Los lenguajes de alto nivel pueden ofrecer portabilidad de software. Esto exige cierto grado de estandarización del lenguaje. La mayoría de los lenguajes de alto nivel ahora están bastante bien definidos. Por ejemplo, portar un programa en Java de una máquina a otra con diferentes arquitecturas y sistemas operativos debería ser una tarea fácil.
- La comprobación en tiempo de compilación puede eliminar muchos errores en una etapa temprana, antes de que el programa se ejecute realmente. La comprobación de declaraciones de variables, la comprobación de tipos, la garantía de que las variables se inicializan correctamente, la comprobación de compatibilidad de argumentos, etc., a menudo son características ofrecidas por los lenguajes de alto nivel. Además, el compilador puede insertar código que realice comprobaciones en tiempo de ejecución, como la comprobación de índices de matrices. El pequeño costo adicional en tiempo de ejecución puede ser un pequeño precio a pagar por la eliminación temprana de errores.

2.1.2. Desventajas de los lenguajes de alto nivel

A pesar de estas ventajas significativas, puede haber circunstancias en las que el uso de un lenguaje de bajo nivel (normalmente un lenguaje ensamblador) puede ser más apropiado. Se pueden identificar algunas ventajas de los lenguajes de bajo nivel.

- Es posible que el programa necesite realizar algunas operaciones de bajo nivel específicas de hardware que no correspondan a ninguna característica del lenguaje de alto nivel. Por ejemplo, el hardware puede almacenar información de estado del dispositivo en una ubicación de almacenamiento determinada —y en la mayoría de los lenguajes de alto nivel no hay manera de expresar el direccionamiento directo de la máquina. Puede ser necesario realizar E/S de bajo nivel, o hacer uso de una instrucción específica de la máquina —de nuevo es probable que sea difícil expresar esto en un lenguaje de alto nivel.
- El uso de lenguajes de bajo nivel a menudo se justifica por la eficiencia en términos de velocidad de ejecución o de requisitos de almacenamiento en tiempo de ejecución. Hay muchas aplicaciones de la programación donde la eficiencia es una preocupación principal. Podría tratarse de cálculos a gran escala que requieran días o semanas de tiempo de procesador o incluso cálculos realmente cortos con restricciones severas en tiempo real. La eficiencia suele estar relacionada con la minimización del tiempo de cómputo, pero otras restricciones como el uso de la memoria o el consumo de energía podrían ser más importantes. En el desarrollo de las implementaciones de los primeros lenguajes, el tema de la eficiencia influyó fuertemente en el diseño de compiladores. Su baja eficiencia era

considerada la principal desventaja de los lenguajes de alto nivel. Se suponía que el código generado por la máquina nunca podría ser tan eficaz como el código escrito a mano. Pero a medida que la tecnología de los compiladores mejoró constantemente, a medida que los procesadores se volvieron más rápidos y a medida que sus arquitecturas se volvieron más adecuadas para ejecutar código generado por los compiladores a partir de programas escritos en lenguajes de alto nivel, el argumento de la eficiencia se volvió mucho menos significativo. Hoy en día, es probable que el código generado por un compilador para una amplia gama de lenguajes de programación y máquinas de destino sea igual de eficaz, si no más, que el código escrito a mano. ¿Implica esto que justificar el uso de los lenguajes de bajo nivel basándose en que permiten producir código eficiente es ahora erróneo? La realidad es que puede haber circunstancias en las que la codificación manual en código de máquina o en lenguaje ensamblador, con mucho cuidado, conducirá a mejores resultados. Pero, al desarrollar software, una regla valiosa que hay que recordar es que no es necesario optimizar el código que ya es lo suficientemente rápido.

3. UNA TAXONOMÍA DE LOS PROCESADORES DE LENGUAJE

Un *programming language processor* o *procesador de lenguaje* es cualquier sistema que manipule programas expresados en algún lenguaje de programación en particular. Con la ayuda de los procesadores de lenguaje es posible ejecutar programas, o prepararlos para ser ejecutados.

Esta definición de procesadores de lenguaje es muy general. Abarca una gran variedad de sistemas, incluyendo los siguientes:

- *Editores*. Un editor permite ingresar, modificar y guardar el texto de un programa en un archivo. Cualquier editor de texto nos permite editar documentos textuales (no necesariamente el texto de un programa). Un tipo de editor más sofisticado es uno diseñado para editar programas expresados en un lenguaje en particular.
- *Traductores* y *compiladores*. Un traductor traduce un texto de un lenguaje a otro. En particular, un compilador traduce un programa de un lenguaje de alto nivel a un lenguaje de bajo nivel, preparándolo así para que se ejecute en una computadora. Al realizar esta traducción, un compilador comprueba si el programa tiene errores sintácticos y contextuales.
- *Intérpretes*. Un intérprete toma un programa expresado en un lenguaje determinado y lo ejecuta inmediatamente. Este modo de ejecución, omitiendo una etapa de compilación en favor de la respuesta inmediata, es el preferido en los entornos interactivos. Normalmente los lenguajes de comandos y los lenguajes de consulta de base de datos son lenguajes interpretados.

En la práctica, durante el desarrollo de programas se utilizan todos los tipos anteriores de procesadores de lenguaje. En un sistema de programación convencional, estos procesadores de lenguaje suelen ser herramientas separadas; esta es la filosofía de las “herramientas de software”. Sin embargo, la

mayoría de los sistemas ahora ofrecen procesadores de lenguaje integrados, en los que la edición, la compilación y la interpretación son solo opciones dentro de un único sistema. Los siguientes ejemplos contrastan estos dos enfoques.

Ejemplo 1. Procesadores de lenguaje como herramientas de software

La filosofía de las "herramientas de software" está bien ejemplificada en el sistema operativo UNIX. De hecho, esta filosofía fue fundamental para el diseño del sistema.

Considere la posibilidad de que un usuario de UNIX desarrolle un juego de ajedrez en Java, utilizando el Java Development Kit (JDK). El usuario invoca un editor, como el editor de pantalla vi, para ingresar y almacenar el texto del programa en un archivo denominado (por ejemplo) `Chess.java`:

```
vi Chess.Java
```

A continuación, el usuario invoca el compilador de Java, `javac`:

```
javac Chess.Java
```

Esto convierte el programa almacenado en código objeto y lo almacena en un archivo denominado `Chess.class`. El usuario ahora puede probar el código objeto ejecutándolo con el intérprete, `java`:

```
java Chess
```

Si el programa no se puede compilar, o se comporta mal cuando se ejecuta, el usuario puede volver a invocar el editor para modificar el programa; a continuación, puede volver a invocar el compilador; y así sucesivamente. Por lo tanto, el desarrollo del programa es un ciclo de edición-compilación-ejecución.

No hay comunicación directa entre estos procesadores de lenguaje. Si el programa no se puede compilar, el compilador generará uno o varios informes de errores, indicando en cada uno la posición del error. El usuario debe tener en cuenta estos informes de errores y, al volver a invocar el editor, debe encontrar los errores y corregirlos. Esto es muy inconveniente, especialmente en las primeras etapas del desarrollo del programa, cuando los errores pueden ser numerosos.

La esencia de la filosofía de las "herramientas de software" es proporcionar un pequeño número de herramientas comunes y sencillas, que se pueden combinar de diversas maneras para realizar una gran variedad de tareas. Por lo tanto, solo es necesario proporcionar un único editor, uno que se pueda utilizar para editar programas en una gran variedad de lenguajes, y de hecho otros documentos textuales también. Lo que se ha descrito hasta aquí es la filosofía de las "herramientas de software" en su forma más pura. En la práctica, esta filosofía se ve comprometida con el fin de facilitar el desarrollo de programas. El editor puede tener una función que le permita al usuario compilar el programa (o de hecho ejecutar cualquier comando del sistema) sin salir del editor. Algunos compiladores van más allá: si el programa no se puede compilar, el editor se vuelve a invocar automáticamente y se posiciona en el primer error.

Estas son soluciones *ad hoc*. Un enfoque nuevo parece preferible: un procesador de lenguaje totalmente integrado, diseñado específicamente para admitir el ciclo de edición-compilación-ejecución.

Ejemplo 2. Procesador de lenguaje integrado

Apache NetBeans es un procesador de lenguaje totalmente integrado para Java, que consta de un editor, un compilador y otras aplicaciones. El usuario ejecuta comandos para abrir, editar, compilar y correr los programas. Estos comandos se pueden seleccionar en los menús desplegables o desde el teclado.

El editor está adaptado a Java. Utiliza sangrías para ayudar con el diseño del programa y colores para distinguir entre palabras clave de Java, literales y comentarios. El editor también está totalmente integrado con las aplicaciones para la construcción de interfaces visuales de Apache NetBeans.

El compilador está integrado con el editor. Cuando el usuario ejecuta el comando 'compile' y se detecta, en tiempo de compilación, que el programa contiene un error, la frase errónea es resaltada y queda lista para su edición inmediata. Si el programa contiene varios errores, el compilador los enumera, y el usuario puede seleccionar un mensaje de error en particular y la frase correspondiente será resaltada.

El programa objeto también se integra con el editor. Si el programa falla en tiempo de ejecución, se resalta la frase errónea. Por supuesto, esta frase no es necesariamente la que contiene el error lógico. Pero sería irrazonable esperar que el procesador de lenguaje depure el programa automáticamente!

3.1. Implementaciones de procesadores de lenguaje

Una vista simplista pero no inexacta del proceso de implementación de un lenguaje sugiere que se requiere algún tipo de programa traductor (un *compilador*) que tome los programas escritos en un lenguaje de alto nivel y los transforme en programas expresados en código de máquina semánticamente equivalentes que se puedan ejecutar en la computadora de destino. Es probable que se requiera también otro software, como las bibliotecas. Conforme la complejidad del lenguaje fuente aumenta a medida que el lenguaje se vuelve cada vez "de más y más alto nivel", más cercano a la expresión humana, uno esperaría que la complejidad del traductor también aumentara.

Muchas rutas son posibles, y pueden ser las características del lenguaje de alto nivel las que fuercen los diferentes enfoques. Por ejemplo, la forma tradicional de implementar Java hace uso de la *Java Virtual Machine* (JVM), donde el compilador traduce el código fuente escrito en Java a código JVM y un programa independiente (un *intérprete*) lee estas instrucciones de máquina virtual y emula las acciones de la máquina virtual, ejecutando efectivamente el programa escrito en Java. Este método de implementación aparentemente ilógico tiene beneficios significativos. En particular, es compatible con la carga dinámica de clases de Java. Sin un código de máquina virtual tan neutro en la arquitectura, implementar la carga dinámica de clases sería mucho más difícil. Además, permite el soporte de la

reflexión, mediante la cual un programa escrito en Java puede, en tiempo de ejecución, examinar o modificar sus propiedades internas.

Los enfoques interpretados son muy apropiados para la implementación de algunos lenguajes de programación. Los costos generales de la compilación se reducen, pero los tiempos de ejecución son más largos. En el campo de la implementación de lenguajes de programación abundan las compensaciones.

Para hacer un uso eficaz de un lenguaje de alto nivel, es esencial conocer su implementación. En algunas áreas de aplicación exigentes, como los *sistemas integrados,* donde un sistema informático con una función fija controla algún dispositivo electrónico o mecánico, puede haber estrictas exigencias sobre el controlador integrado y el código ejecutable. Puede haber restricciones en tiempo real (por ejemplo, cuando se controla el tiempo de encendido de un motor de automóvil, donde un conjunto predefinido de operaciones tiene que completarse en la duración de una chispa), restricciones de memoria (¿puede todo el programa caber en los 64 KiB disponibles en la versión barata del chip microcontrolador?) o restricciones de consumo de energía (¿con qué frecuencia tengo que cargar las baterías en mi teléfono celular?). Estas restricciones imponen exigencias sobre el rendimiento del hardware, pero también sobre la forma en que se implementa realmente el lenguaje de alto nivel que implementa la funcionalidad del sistema. Los diseñadores necesitan tener un conocimiento profundo de la implementación para poder abordar estos problemas.

3.1.1. Compiladores

El compilador es un programa que traduce de un *lenguaje fuente* a un *lenguaje meta*, y que está implementado en algún *lenguaje de implementación.* Como se ha visto, la perspectiva tradicional de un compilador es tomar como entrada algún lenguaje de alto nivel y generar código de máquina para alguna máquina de destino.

El campo de la compilación no se limita a la generación de código en un lenguaje de bajo nivel. Es posible desarrollar tecnología de compilación para traducir de un lenguaje de alto nivel a otro. Por ejemplo, algunos de los primeros compiladores de C++ generaban código en C en lugar de código de máquina. Los compiladores de C existentes estaban disponibles para realizar el paso final.

La complejidad de un compilador no solo se ve influenciada por las complejidades de los lenguajes fuente y meta, sino también por el requisito de que el código objeto esté optimizado. Hay un peligro real de ser abrumado por las complejidades y los detalles de las tareas abarcadas en la construcción de un compilador. Un enfoque bien estructurado para el desarrollo del compilador es esencial.

El diseño de los compiladores está influenciado por las características de las especificaciones formales del lenguaje, por la teoría de autómatas, por el análisis de algoritmos, por el diseño del procesador, por las estructuras de datos y el diseño del algoritmo, por los servicios del sistema operativo, por el conjunto de instrucciones de la máquina de destino y otras características del hardware, por las

características del lenguaje de implementación, así como también por las necesidades de los usuarios de los compiladores. Codificar un compilador puede ser una tarea desalentadora, pero el proceso se simplifica en gran medida haciendo uso de los enfoques, experiencias, recomendaciones y algoritmos de otros escritores de compiladores.

Es útil decir que un compilador se puede componer a partir de dos fases distintas. La primera es la fase de *análisis*: la lectura del programa fuente y la creación de estructuras de datos internas que reflejen su estructura sintáctica y semántica según la definición del lenguaje. La segunda es la fase de *síntesis*: la generación de código para una máquina a partir de las estructuras de datos creadas por la fase de análisis. Pensar en un compilador en términos de estas dos fases distintas puede simplificar en gran medida su diseño y su implementación.

Esta subdivisión, aunque simple, tiene consecuencias importantes para el diseño de compiladores. La interfaz entre estas dos fases es algún lenguaje intermedio, una vía entre los lenguajes fuente y meta. Si este lenguaje intermedio es diseñado con cuidado, puede ser posible estructurar el compilador para que la fase de análisis se vuelva independiente de la máquina de destino y la fase de síntesis se vuelva independiente del lenguaje fuente. Esto, en teoría, permite grandes ahorros potenciales en el esfuerzo de implementación invertido durante el desarrollo de nuevos compiladores. Si un compilador estructurado de esta manera debe reorientarse a una nueva arquitectura de máquina, solo es necesario modificar o reescribir la fase de síntesis. No es necesario tocar la fase de análisis. De forma similar, si es necesario modificar el compilador para compilar un lenguaje fuente diferente, pero destinado a la misma máquina, solo es necesario modificar o reescribir la fase de análisis.

El analizador léxico y el analizador sintáctico hacen cosas similares. Ambos agrupan caracteres o *tokens* para formar unidades sintácticas más grandes. Por lo tanto, una cuestión a resolver es si una estructura gramatical determinada debe ser reconocida por el analizador léxico o por el analizador sintáctico. El enfoque tradicional, y es un enfoque que funciona bien, consiste en reconocer las estructuras más simples en el analizador léxico, específicamente aquellas que se pueden expresar en términos de una gramática de Chomsky de tipo 3. Las estructuras sintácticas especificadas por una gramática de Chomsky de tipo 2 o más compleja son resueltas por el analizador sintáctico. En teoría, el analizador sintáctico también podría tratar con los *tokens* léxicos mediante un enfoque de análisis propio de las gramáticas de Chomsky de tipo 2, pero esto aumentaría significativamente la complejidad del analizador sintáctico. Además, al hacer que el analizador léxico se encargue de estos *tokens* mejora la eficiencia del compilador porque se pueden usar técnicas de análisis más simples y rápidas, propias de las gramáticas de Chomsky de tipo 3.

Los análisis léxico, sintáctico y semántico y las fases de optimización independientes de la máquina forman el *front end* del compilador. Las fases de generación de código y las dependientes de la máquina

forman el *back end*. Todas estas fases tienen roles específicos y distintos. Y, para apoyar el diseño de estos módulos individuales, las interfaces entre ellos tienen que ser definidas con cuidado.

La implementación de un lenguaje no se detiene en el compilador. Además, siempre se requiere proporcionar el acceso a bibliotecas de rutinas que proporcionen un entorno en el que se pueda ejecutar el código generado por el compilador. También se necesitan otras herramientas como depuradores, enlazadores (*linkers*), asistentes de documentación y entornos de desarrollo interactivos. No es una tarea fácil.

Para hacer frente a esta complejidad se requiere estructurar el proyecto de construcción del compilador siguiendo un estricto enfoque de diseño. Las técnicas tradicionales de ingeniería de software se adaptan bien a los proyectos de construcción de compiladores, asegurando que sean adecuados la modularización, las pruebas, el diseño de la interfaz, etc. Las pruebas exhaustivas, etapa por etapa, son vitales para un compilador.

Para facilitar la tarea de producir una implementación de un lenguaje de programación, se han desarrollado muchas herramientas de software que ayudan a generar partes de un compilador o intérprete automáticamente. Por ejemplo, los analizadores léxicos y los analizadores sintácticos (correspondientes a las dos etapas iniciales del proceso de compilación) a menudo se crean con la ayuda de herramientas que toman, como entrada, la especificación formal de la sintaxis del lenguaje de programación y generan, como salida, código fuente que se incorporará en el compilador. La modularización de los compiladores también ha ayudado a reducir la carga de trabajo.

Por ejemplo, se han creado muchos compiladores utilizando un *front end* independiente de la máquina de destino, un *back end* independiente del lenguaje fuente y una representación intermedia estándar como interfaz entre ellos. Así, los *front ends* y *back ends* se pueden mezclar y combinar para producir una gran variedad de nuevos compiladores completos. Hoy en día, los proyectos de construcción de compiladores rara vez comienzan desde cero.

3.1.2. Intérpretes

La ejecución de un programa escrito en un lenguaje de alto nivel mediante un compilador es un proceso de dos etapas. En la primera etapa, el *código fuente* se traduce a *código de máquina* y, en la segunda etapa, el *hardware* ejecuta o una *máquina virtual* interpreta este código para producir resultados. En otro enfoque bastante popular para la implementación de lenguajes, directamente no se genera ningún código. En su lugar, un *intérprete* lee el código fuente y lo "ejecuta" inmediatamente. Por lo tanto, si el intérprete encuentra la instrucción `a=b+1`, analiza los caracteres en el código fuente para determinar que la entrada es una instrucción de asignación, extrae de sus propias estructuras de datos el valor de `b`, le suma uno a ese valor y almacena el resultado en su propio registro de `a`.

Esto es análogo a lo que uno haría si le entregaran un programa en C y le dijeran que lo ejecutara a mano. Leería cada sentencia en el orden adecuado y haría mentalmente lo que dice el código. Probablemente realizaría un seguimiento de los valores de las variables en una hoja de papel y escribiría cada salida del programa, hasta que se haya completado su ejecución. Esencialmente, eso sería lo que hace un intérprete de C. Un intérprete de C lee un programa escrito en C y lo ejecuta. No hay que generar ni cargar ningún código objeto. En su lugar, un intérprete traduce un programa como las acciones que resultan de ejecutarlo inmediatamente. La *inmediatez* es la característica clave de la interpretación; el programa fuente no se traduce previamente a una representación de bajo nivel.

En un entorno interactivo, la inmediatez es muy ventajosa. Por ejemplo, el usuario de un lenguaje de comandos espera una respuesta inmediata cuando ingresa cada comando; sería absurdo esperar que el usuario introduzca una secuencia completa de comandos antes de ver, de golpe, todas las respuestas desde la primera. De forma similar, el usuario de un lenguaje de consulta de base de datos espera una respuesta inmediata a cada consulta. En este modo de trabajo, el 'programa' se introduce una vez y luego se descarta.

Sin embargo, es mucho más probable que el usuario de un lenguaje de programación retenga el programa para su uso posterior y, posiblemente, para continuar su desarrollo. Aun así, la traducción del lenguaje de programación a un lenguaje intermedio seguido de la interpretación del lenguaje intermedio (es decir, la *compilación interpretativa*) es una buena alternativa a la *compilación completa,* especialmente durante las primeras etapas del desarrollo del programa.

¿Cómo decide uno cuándo usar un intérprete y cuándo usar un compilador? Cuando se ingresa un código fuente en el intérprete, el intérprete toma el control para comprobar y ejecutar el programa. Un compilador también comprueba el código fuente, pero en su lugar genera código objeto. Después de ejecutar el compilador, se debe ejecutar el enlazador para generar el código objeto y, a continuación, se debe cargar el código objeto en la memoria para ejecutarlo. Si el compilador genera un código objeto en lenguaje ensamblador, también se debe ejecutar un programa ensamblador. Por lo tanto, con un intérprete definitivamente se requiere menos esfuerzo para ejecutar un programa.

Los intérpretes pueden ser más versátiles que los compiladores. Se puede utilizar Java para escribir un intérprete de C que se ejecute en una PC basada en Microsoft Windows, en una Macintosh de Apple y en un sistema basado en Linux, de modo que el intérprete pueda ejecutar código fuente escrito en C en cualquiera de esas plataformas. Un compilador, sin embargo, genera código objeto para una computadora determinada. Por lo tanto, incluso si se usó un compilador de C escrito originalmente para la PC y se lo ejecutó en la Mac, igual generaría código objeto para la PC. Para que el compilador genere código objeto para la Mac, habría que volver a escribir partes sustanciales del compilador. Se puede evitar este problema haciendo que el compilador genere código objeto para la máquina virtual de Java. Esta máquina virtual se ejecuta en varias plataformas informáticas.

¿Qué sucede si el código fuente contiene un error lógico que no se muestra hasta el tiempo de ejecución, como, por ejemplo, un intento de dividir un número entero por una variable cuyo valor es cero?

Puesto que un intérprete tiene el control cuando está ejecutando el código fuente, puede detenerse e indicar el número de línea de la instrucción en infracción y el nombre de la variable. Incluso puede solicitar alguna acción correctiva antes de reanudar la ejecución del programa, como cambiar el valor de la variable a algo distinto de cero. Un intérprete puede incluir un depurador interactivo de *nivel de código fuente,* también conocido como *depurador simbólico*. Se lo denomina *simbólico* porque el depurador permite utilizar símbolos del código fuente, como nombres de variables.

Por otro lado, el código objeto generado por un compilador y un enlazador generalmente no incluye información del código fuente, como los números de línea y los nombres de variables. Cuando se produce un error en tiempo de ejecución, el programa puede simplemente abortar la ejecución e imprimir un mensaje con la dirección de la instrucción del código objeto incorrecta. A continuación, depende del usuario del programa averiguar en el código fuente cuál es la instrucción correspondiente y la variable cuyo valor era cero.

Así que, cuando se trata de depuración, un *intérprete* suele ser el camino a seguir. Algunos compiladores pueden generar información adicional en el código objeto para que, si se produce un error en tiempo de ejecución, el código objeto pueda imprimir el número de línea de la instrucción en infracción o el nombre de la variable. A continuación, se puede corregir el error, volver a compilar y volver a ejecutar el programa. La generación de información adicional en el código objeto puede hacer que este se ejecute más lentamente. Esto puede inducir a que el programador desactive las características de depuración cuando esté a punto de compilar la versión de "producción" del programa. Esta situación se ha comparado con usar un chaleco salvavidas cuando se está aprendiendo a navegar en un lago, y luego quitárselo cuando se está a punto de salir al océano.

Una vez que se ha depurado correctamente el programa, la principal preocupación es la rapidez con la que se ejecuta. Dado que una computadora puede ejecutar un programa en su lenguaje de máquina nativo a la máxima velocidad, un programa compilado puede correr varios órdenes de magnitud más rápido que un código fuente interpretado. Un compilador es definitivamente el ganador cuando se trata de velocidad. Esto ciertamente es verdad en el caso de un compilador optimizador que sabe cómo generar código especialmente eficaz.

Por lo tanto, si se debe usar un compilador o un intérprete depende de cuáles sean los aspectos importantes del desarrollo y la ejecución de un programa. Lo mejor de ambos mundos incluiría un intérprete con un depurador simbólico interactivo para usar durante el desarrollo del programa y un compilador para generar código de máquina para una ejecución rápida, después de haber depurado el programa.

La interpretación tiene sentido cuando existe la mayoría de las siguientes circunstancias:

- El programador está trabajando en modo interactivo, y desea ver los resultados de cada instrucción antes de entrar en la siguiente instrucción.
- El programa debe ser utilizado una vez y luego descartado, y por lo tanto la velocidad de funcionamiento no es muy importante.
- Se espera que cada instrucción se ejecute una sola vez (o al menos no con mucha frecuencia).
- Las instrucciones tienen formatos simples, y por lo tanto se pueden analizar de forma fácil y eficiente.

La interpretación es muy lenta. La interpretación de un programa fuente, escrito en un lenguaje de alto nivel, puede ser hasta 100 veces más lenta que ejecutar un programa equivalente en código de máquina. Por lo tanto, la interpretación no tiene sentido cuando:

- El programa se ejecutará en modo de producción, por lo que la velocidad es importante.
- Se espera que las instrucciones se ejecuten con frecuencia.
- Las instrucciones tienen formatos complicados y, por lo tanto, su análisis requiere mucho tiempo. Este es el caso en la mayoría de los lenguajes de alto nivel.

4. PROBLEMAS PRÁCTICOS DE LA INTERPRETACIÓN

El proceso de interpretación del código fuente suena atractivo porque no hay necesidad de la implementación de la generación de código, que es potencialmente compleja. Pero hay problemas prácticos. El primer problema se refiere al rendimiento. Si una instrucción del código fuente se ejecuta repetidamente, es analizada cada vez, antes de cada ejecución. El costo del posible múltiple análisis de instrucciones llevado a cabo por el intérprete al emular la acción de la instrucción será muchas veces mayor que el costo de ejecutar algunas instrucciones de máquina obtenidas mediante una compilación. Sin embargo, este costo se puede reducir con bastante facilidad haciendo el análisis del programa solo una vez, traduciéndolo a una forma intermedia que posteriormente se interpreta. Muchos lenguajes han sido implementados de esta manera, utilizando una forma intermedia interpretada, a pesar de la sobrecarga de la interpretación.

El segundo problema se refiere a la necesidad de que un intérprete esté presente en el tiempo de ejecución. Cuando un programa está "corriendo", se encuentra en la memoria del sistema en forma de código fuente o en una forma intermedia posterior al análisis, junto con el programa intérprete. Es probable que la memoria ocupada total sea mucho mayor que la del código compilado equivalente. Para sistemas pequeños e integrados con una memoria muy limitada, esto puede ser una desventaja decisiva.

En cierto sentido, todas las implementaciones de los lenguajes de programación se interpretan. En la interpretación del código fuente, el intérprete es complejo porque tiene que analizar las instrucciones del código fuente y, a continuación, emular su ejecución. En la interpretación de código intermedio, el

intérprete es más sencillo porque el análisis del código fuente se ha realizado de antemano. En el enfoque compilado tradicional donde se genera código de máquina de un hardware concreto, la interpretación es realizada enteramente por el hardware: no hay interpretación de software y, por lo tanto, no hay sobrecarga. Mirando estos tres niveles de interpretación con mayor detalle, se pueden identificar fácilmente sus puntos fuertes y débiles:

- *Interpretación de código fuente*: la complejidad del intérprete es alta, la eficiencia de la ejecución (*runtime efficiency*) es baja (análisis repetido y emulación de las instrucciones del código fuente), el costo de compilación inicial es cero porque no hay ningún compilador independiente y, por lo tanto, el retraso para iniciar la ejecución del programa también es cero.
- *Interpretación de código intermedio*: la complejidad del intérprete es menor, la eficiencia de la ejecución (*runtime efficiency*) se mejora (el análisis y la emulación de las instrucciones de código intermedios es comparativamente simple), hay un costo de compilación inicial y, por lo tanto, hay un retraso para iniciar la ejecución del programa.
- *Interpretación del código objeto / compilación completa*: no hay necesidad de un intérprete en software por lo que la complejidad del intérprete es cero, la eficiencia de la ejecución (*runtime efficiency*) es alta (la interpretación del código de máquina la realiza el hardware), hay un costo de compilación inicial significativo y, por lo tanto, puede haber un retraso importante para iniciar la ejecución del programa.

Los diferentes requisitos de memoria de los tres enfoques son algo más difíciles de cuantificar y dependen de los detalles de implementación. En el caso de la interpretación de código fuente, una implementación simplista requeriría que tanto el código fuente del programa como el intérprete (que es complejo) estén en la memoria principal. El caso de la interpretación de código intermedio requeriría que la versión del programa en código intermedio y el intérprete (que es más simple) estén en la memoria principal. Y en el caso de la compilación completa, solo el código objeto compilado tendría que estar en la memoria principal. Esto, por supuesto, no tiene en cuenta los requisitos de memoria del programa en ejecución: espacio para variables, estructuras de datos, búferes, código de bibliotecas, etc.

Hay más comparaciones entre puntos fuertes y débiles. Por ejemplo, cuando se modifica el código fuente, en el caso de interpretación de código fuente no hay ninguna sobrecarga de compilación adicional, mientras que en el caso de compilación completa, es habitual que se vuelva a compilar todo el programa o módulo. En el caso de interpretación de código intermedio, puede ser posible volver a compilar las instrucciones del código fuente que han cambiado, evitando una recompilación completa.

Por último, cabe destacar que esta cuestión de la menor eficiencia de las implementaciones interpretadas rara vez es una razón válida para descartar el uso de un intérprete. La sobrecarga de interpretación en el tiempo y el espacio suele ser irrelevante, particularmente en los sistemas informáticos más grandes, y los beneficios ofrecidos bien pueden compensar cualquier problema de eficiencia.

5. CONSIDERACIONES TERMINOLÓGICAS

Solía ser más fácil explicar las diferencias entre un intérprete y un compilador. Con la creciente popularidad de las máquinas virtuales, la imagen se vuelve un poco borrosa.

Una máquina virtual (VM o *virtual machine*) es un programa que simula una computadora. Este programa puede ejecutarse en diferentes plataformas informáticas reales. Por ejemplo, la *Java Virtual Machine* (JVM) se puede ejecutar en una PC basada en Microsoft Windows, en una Macintosh de Apple, en un sistema basado en Linux y en muchas plataformas más.

La VM tiene su propio lenguaje de máquina, y las instrucciones de este lenguaje las interpreta la computadora real donde la VM es ejecutada. Por lo tanto, si uno escribe un traductor que traduce código fuente en C al lenguaje de máquina interpretado por la VM, ¿es esto un compilador o un intérprete?

En lugar de buscarle la quinta pata al gato, pongámonos de acuerdo en que si un sistema recibe un código fuente y lo traduce a código de máquina, ya sea para una computadora real o para una VM, el sistema es un *compilador*. Un sistema que ejecuta el código fuente sin traducirlo primero a código de máquina es un *intérprete*.

Hay muchas implementaciones de lenguajes de programación de alto nivel donde el *compilador* genera código de máquina para una VM y, a continuación, un programa independiente, el *intérprete*, lee ese código de máquina y emula su ejecución en la VM, instrucción por instrucción. A primera vista, puede parecer que hacer esto carece de sentido: ¿por qué no generar directamente código de máquina de la computadora real? Pero este enfoque tiene varias ventajas significativas, incluyendo las siguientes:

- El diseño del código generado por el compilador no está restringido por la arquitectura de la máquina de destino. Esto puede simplificar el diseño del compilador porque este no tiene que lidiar con las peculiaridades y restricciones del hardware real. Por ejemplo, puede ser conveniente utilizar una arquitectura basada en una pila para la VM, aunque esta característica no esté directamente disponible en el hardware de destino.
- Se mejora la portabilidad. Si el intérprete está escrito en un lenguaje portátil, tanto él como el código de máquina generado para la VM se pueden distribuir y ejecutar fácilmente en computadoras con diferentes arquitecturas o sistemas operativos. Esto se lleva bien con la prevalencia actual de los entornos heterogéneos en red.
- El código de máquina de la VM se puede diseñar para que sea especialmente compacto. Hay áreas de aplicación donde esto puede ser muy importante.
- Las características de depuración y supervisión en tiempo de ejecución se pueden incorporar en el intérprete de la VM, lo que permite mejorar la seguridad en la ejecución del programa.
- El código de máquina de la VM se puede ejecutar en un entorno supervisado (*sandbox*), para impedir que esta realice operaciones ilegales. Por ejemplo, la VM puede funcionar con datos tipados y la comprobación de tipos de tiempo de ejecución puede proporcionar información de depuración útil.

La desventaja obvia de este enfoque es la cuestión de la eficiencia. Es probable que el código interpretado sea más lento que la ejecución nativa. Pero, para la mayoría de las aplicaciones, esto no resulta ser verdaderamente significativo. Por lo general, las ventajas compensan ampliamente esta desventaja, y muchos lenguajes de programación modernos están implementados de esta manera.

La naturaleza de las VM plantea preguntas interesantes. El diseño de una VM no debe ser demasiado cercano al diseño del hardware porque, de lo contrario, la ventaja de la simplificación del compilador esencialmente desaparecerá. En cambio, si el diseño de la VM se hace muy cercano o idéntico al lenguaje que se está implementando, el compilador se volverá muy simple, pero el intérprete tendrá que lidiar con el detalle de la decodificación y el análisis de este código potencialmente complejo. Las funciones del compilador se estarán desplazando al intérprete. Sin embargo, varios lenguajes han sido implementados correctamente de esta manera, donde el intérprete hace todo el trabajo, eliminando la necesidad del compilador independiente. Algunas implementaciones del lenguaje BASIC se han llevado a cabo de esta manera. Pero debido a la necesidad de que el intérprete realice un análisis repetido de alguna sentencias del código fuente, esto rara vez es un enfoque práctico para un sistema de producción.

6. FUNCIONAMIENTO DE LOS INTÉRPRETES

Los intérpretes son mágicos. En la superficie se ven engañosamente simples: entra el texto y sale algo. Son programas que toman como entrada otros programas y producen algo. Simple, ¿verdad? Pero cuanto más uno lo piensa, más fascinante se vuelven. Los caracteres aparentemente aleatorios —letras, números y caracteres especiales— se introducen en el intérprete y, de repente, se vuelven *significativos.* ¡El intérprete les da sentido! Genera sentido a partir de algo sin sentido. Y la computadora, una máquina que supuestamente solo entiende ceros y unos, ahora entiende y actúa según este extraño lenguaje con que ha sido alimentada, gracias a un intérprete que traduce este lenguaje mientras lo va leyendo.

Es difícil hacer afirmaciones genéricas sobre los intérpretes ya que su variedad es muy alta y ninguno es igual a otro. Lo que se puede decir es que el único atributo fundamental que comparten es que todos reciben código fuente y lo evalúan sin producir ningún resultado intermedio visible que se pueda ejecutar más adelante. Esto contrasta con los compiladores, que reciben código fuente y producen una salida en otro lenguaje que el sistema subyacente puede comprender.

Estos son algunos ejemplos bien conocidos de intérpretes:

- *Intérprete de BASIC*: el lenguaje BASIC tiene expresiones y comandos de asignación como otros lenguajes de alto nivel. Pero sus estructuras de control son de bajo nivel: un programa es solo una secuencia de comandos enlazados mediante saltos condicionales e incondicionales. Un intérprete de BASIC busca, analiza y ejecuta un comando a la vez.
- *Intérprete de Lisp*: Lisp es un lenguaje muy inusual en el sentido de que utiliza, tanto para el código como para los datos, una estructura de datos común (la lista). De hecho, ¡un programa en Lisp puede

generar nuevo código en tiempo de ejecución! La estructura de los programas en Lisp se presta a la interpretación.

- *Intérprete del lenguaje de comandos (shell) de UNIX*: el usuario de UNIX le da órdenes al sistema operativo introduciendo comandos textuales. El programa *shell* lee una línea de texto desde la entrada, la analiza para extraer un nombre de comando junto con algunos argumentos y ejecuta el comando mediante una llamada del sistema. El usuario puede ver los resultados del comando antes de introducir el siguiente. Los comandos constituyen un lenguaje de comandos y el *shell* es un intérprete para ese lenguaje de comandos.
- *Intérprete de SQL*: SQL es un lenguaje de consulta de base de datos. El usuario extrae información de la base de datos introduciendo una consulta escrita en SQL, que se analiza y ejecuta inmediatamente. Esto lo hace un intérprete de SQL dentro del sistema de administración de bases de datos.

Algunos intérpretes son realmente diminutos y ni siquiera incluyen un paso de análisis. Simplemente interpretan la entrada de inmediato. Basta observar alguno de los muchos intérpretes de *Brainfuck* que hay en Internet para ver esto.

En el otro extremo del espectro hay tipos mucho más elaborados de intérprete, que están altamente optimizados y utilizan técnicas avanzadas de análisis y evaluación. Algunos de ellos no solo evalúan su entrada, sino que la traducen a una representación interna denominada *bytecode* y, a continuación, evalúan esta última. Aún más avanzados son los intérpretes JIT (*just-in-time* o *justo a tiempo*) que, en tiempo de ejecución, traducen la entrada a código de máquina nativo que luego se ejecuta.

Pero también, entre esas dos categorías, hay intérpretes que analizan el código fuente, partiendo de él crean un árbol de sintaxis abstracta (*abstract syntax tree* o AST) y, a continuación, evalúan ese árbol. Este tipo de intérprete a veces se llama "intérprete recorredor de árbol" (*tree-walking interpreter*), porque "camina" por el AST y lo interpreta.

Para trabajar con el código fuente, debe convertírselo a un formato más accesible. Por más fácil que sea trabajar con un texto plano en un editor, cuando se trata de interpretarlo, en un lenguaje de programación, como otro lenguaje de programación, esto se vuelve rápidamente engorroso.

Por lo tanto, antes de evaluar el código fuente, su representación debe cambiarse dos veces (Fig. 2):

Figura 2. Cambios del código fuente antes de la evaluación

La primera transformación, que convierte el código fuente en *tokens*, se denomina "análisis léxico" (o "lexing", para abreviar). La lleva a cabo un analizador léxico o *lexer* (también denominado *tokenizador*

o *scanner* —algunos usan una palabra u otra para denotar diferencias sutiles en el comportamiento), que lee los caracteres del código fuente y los agrupa en una secuencia de *tokens* léxicos. Cada *token* léxico es una estructura de datos pequeña y fácilmente categorizable que representa un componente básico del lenguaje de programación que se está procesando. Hay *tokens* como números, identificadores, signos de puntuación, operadores, cadenas, palabras reservadas, etc. Los comentarios se pueden omitir a menos que el lenguaje defina componentes especiales, codificados en los comentarios, que puedan ser necesarios más adelante. El espacio en blanco (caracteres de espacio, tabulaciones, saltos de línea, etc.) puede omitirse excepto, de nuevo, cuando tenga algún significado sintáctico (las tabulaciones a veces indican la estructura de bloques, los espacios en blanco pueden estar ubicados dentro de cadenas de caracteres, etc.). La sintaxis de los *tokens* léxicos suele ser simple, y la expectativa es que su estructura se pueda especificar formalmente en términos de una gramática de Chomsky de tipo 3 (es decir, mediante expresiones regulares). Esto simplifica considerablemente la codificación del analizador léxico.

A continuación, los *tokens* se le pasan al analizador sintáctico o *parser*. Esto suele hacerse de dos maneras diferentes: por un lado, el analizador léxico puede escanear todo el archivo de entrada y, a continuación, pasarle toda la lista de *tokens* al analizador sintáctico. Como alternativa, los *tokens* se le pueden ir pasando al analizador sintáctico individualmente, cada vez que este solicite uno. A continuación, el analizador sintáctico realiza la segunda transformación y convierte los *tokens* en un "árbol de sintaxis abstracta".

Quienes han programado alguna vez, probablemente hayan oído hablar de los *parsers*, sobre todo al encontrar un "error del *parser*". O tal vez incluso hayan dicho algo como "hay que *parsear* esto". La palabra "*parser*" es tan común como "intérprete" y "lenguaje de programación". Todo el mundo sabe que ellos *existen*. Tienen que hacerlo, ¿verdad? Porque, ¿quién más sería responsable de los "errores del *parser*"?

Pero, ¿qué es exactamente un *parser*? ¿Cómo hace su trabajo? Esto es lo que Wikipedia tiene que decir al respecto:

> Un analizador sintáctico (*parser*) es un programa informático que analiza una cadena de símbolos de acuerdo a las reglas de una gramática formal... El análisis sintáctico convierte el texto de entrada en otras estructuras (comúnmente árboles), que son más útiles para el posterior análisis y capturan la jerarquía implícita de la entrada. Un *analizador léxico* crea *tokens* de una secuencia de caracteres de entrada y son estos *tokens* los que son procesados por el analizador sintáctico para construir la estructura de datos, por ejemplo un árbol de análisis o árboles de sintaxis abstracta.

Para un artículo de Wikipedia sobre un tema de ciencias de la computación, este extracto es notablemente fácil de entender. ¡Incluso el *lexer* puede ser reconocido allí!

Un *parser* convierte su entrada en una estructura de datos que la representa. Eso suena bastante abstracto, así que se lo ilustrará aquí con un ejemplo en JavaScript:

```
> var input = '{"name": "Luke", "age": 23}';
> var output = JSON.parse(input);
> output
{ name: "Luke", age: 23 }
> output.name
"Luke"
> output.age
23
```

La entrada se guarda en `input`. No es más que un texto, una cadena, pasada a un *parser* oculto detrás de la función `JSON.parse` que retorna un valor de salida recibido por `output`. Esta salida es la estructura de datos que representa la entrada: un objeto de JavaScript con dos campos denominados `name` y `age`, cuyos valores también corresponden a la entrada. Un *parser* de JSON toma texto como entrada y construye una estructura de datos que representa la entrada. Eso es exactamente lo que hace el *parser* de un lenguaje de programación. La diferencia está en el hecho de que, en el caso de un *parser* de JSON, al observar la entrada se puede ver la estructura de datos. Mientras que observando esto

```
if ((5 + 2 * 3) == 91) { return computeStuff(input1, input2); }
```

no es inmediatamente obvio cómo se lo podría representar con una estructura de datos. Los usuarios de lenguajes de programación rara vez pueden ver o interactuar con la representación interna del código fuente analizado. Los programadores de Lisp son la excepción a la regla: en Lisp las estructuras de datos utilizadas para representar el código fuente son las mismas utilizadas por cualquier programador de Lisp. El código fuente analizado es fácilmente accesible como datos en el programa. "El código es dato, los datos son código" es algo que los programadores de Lisp dicen con mucha frecuencia.

En la mayoría de los intérpretes y compiladores, la estructura de datos utilizada para la representación interna del código fuente se denomina "árbol de sintaxis abstracta" (*abstract syntax tree* o AST, para abreviar). La sintaxis es *abstracta* porque ciertos detalles del código fuente se omiten en el AST. Puntos y comas, saltos de línea, espacios en blanco, comentarios —dependiendo del lenguaje y del *parser*, estos no se representan en el AST, sino que simplemente guían al *parser* durante su construcción.

Un hecho a tener en cuenta es que no hay un formato de AST universal que sea utilizado por todos los *parsers*. La implementación concreta dependerá del lenguaje de programación que se esté analizando.

Bien, entonces esto es lo que hacen los *parsers*. Toman los *tokens* del código fuente como entrada y producen una estructura de datos que representa ese código. Al crear la estructura de datos, inevitablemente analizan la entrada, comprobando que se ajuste a la estructura sintáctica esperada. Por ello, el análisis que lleva a cabo el *parser* también se denomina *análisis sintáctico*.

Pero la fase de análisis no está del todo completa incluso después de que se haya construido el árbol de sintaxis. Las gramáticas de Chomsky de tipo 2 tradicionales utilizadas para crear el analizador sintáctico no pueden tratar directamente con problemas contextuales como la comprobación de los tipos, la declaración y los ámbitos de los nombres, la elección de los operadores sobrecargados, etc. Este es el papel de la fase de *análisis semántico.* Aquí se atraviesa el árbol, insertando y comprobando la información de los tipos. Los lenguajes tipados pueden requerir que todos o casi todos los nodos del árbol estén etiquetados con un tipo de datos. La complejidad aumenta cuando el lenguaje que se compila permite tipos definidos por el usuario. Las reglas para la compatibilidad de tipos también deben aplicarse aquí. Por ejemplo, ¿permite el lenguaje que se asigne un valor entero a una variable real (de punto flotante)?

El proceso de evaluación de un intérprete es donde el código se vuelve significativo. Sin evaluación, una expresión como `1 + 2` es solo una serie de caracteres, *tokens* o una estructura de árbol que representa esta expresión. No significa nada. Evaluada, por supuesto, `1 + 2` se convierte en `3`, `5 > 1` se convierte en `true`, `5 < 1` se convierte en `false`, y `puts("Hola mundo!")` se convierte en el mensaje amistoso que todo el mundo conoce.

El proceso de evaluación define cómo funciona el lenguaje de programación que se está interpretando. Por ejemplo:

```
let num = 5;
if (num) {
    return a;
} else {
    return b;
}
```

Esto devolverá `a` o `b` dependiendo de que el proceso de evaluación del intérprete decida si el entero `5` es equivalente a *verdadero* o no. En algunos lenguajes lo es, en otros solo sería válido usar entre los paréntesis una expresión que produzca un valor *booleano*, como, por ejemplo, `5 != 0`.

Considere esto:

```
let one = fn() {
    printLine("one");
    return 1;
};

let two = fn() {
    printLine("two");
    return 2;
};

add(one(), two());
```

¿Se imprime primero `uno` y luego `dos`, o es al revés? Depende de la especificación del lenguaje y, en última instancia, de la implementación de su intérprete y del orden en que se evalúan los argumentos en una sentencia de llamada.

La *evaluación* es donde las implementaciones de intérpretes (independientemente del lenguaje que estén interpretando) también divergen más. Hay muchas estrategias diferentes para elegir al evaluar el código fuente. También vale la pena señalar de nuevo que la línea entre intérpretes y compiladores es borrosa. La noción de un intérprete como algo que no va dejando artefactos ejecutables a su paso (a diferencia de un compilador, que hace precisamente eso) se vuelve difusa muy rápido cuando se observan las implementaciones de lenguajes de programación altamente optimizados del mundo real.

El llamado *modelo de evaluación basado en ambientes* consta de dos partes básicas:

1. Para *evaluar* una combinación (una expresión compuesta), se evalúan las subexpresiones y luego se le aplica el valor de la subexpresión *operador* a los valores de las subexpresiones *operandos*.
2. Para *aplicar* un procedimiento compuesto a un conjunto de argumentos, se evalúa el cuerpo del procedimiento en un nuevo ambiente.

Estas dos reglas describen la esencia del proceso de evaluación, un ciclo básico en el que las expresiones a evaluar en ambientes son reducidas a procedimientos para aplicar a argumentos, que a su vez se reducen a nuevas expresiones a evaluar en nuevos ambientes, etc., hasta llegar a los símbolos, cuyos valores se buscan en el ambiente, y a los procedimientos primitivos, que se aplican directamente. Este ciclo de evaluación estará representado por la interacción entre los dos procedimientos críticos en el evaluador: `eval` y `apply`. La implementación del evaluador dependerá de procedimientos que definan la sintaxis de las expresiones a evaluar.

El *modelo de evaluación basado en ambientes* es similar a cómo los seres humanos evalúan una expresión matemática: se insertan los valores de las variables en la expresión y se la evalúa poco a poco, comenzando con el paréntesis más interno y avanzando hasta obtener el resultado de la expresión. Luego se puede repetir el proceso, dándoles a las variables otros valores.

Sin embargo, existen algunas diferencias. Mientras que el ser humano copiará el texto de la fórmula con las variables reemplazadas por valores, y luego escribirá una secuencia de copias cada vez más reducidas de la fórmula hasta que esta se reduzca a un solo valor, el intérprete mantendrá la fórmula (o, mejor dicho, el árbol de sintaxis abstracta de una expresión) sin cambios y utilizará una tabla de símbolos para realizar un seguimiento de los valores de las variables. En lugar de reducir una fórmula, el intérprete es una función que toma un árbol de sintaxis abstracta y una tabla de símbolos como argumentos y devuelve el valor de la expresión representada por ese árbol de sintaxis abstracta. La función puede llamarse a sí misma de forma recursiva con partes del árbol de sintaxis abstracta para encontrar los valores de subexpresiones y, cuando evalúa una variable, puede buscar su valor en la tabla de símbolos.

Este proceso puede extenderse para manejar también sentencias y declaraciones, pero la idea básica es la misma: una función recibe el AST del programa y, posiblemente, alguna información adicional sobre el contexto (como una tabla de símbolos o los datos que entran al programa) y devuelve la salida resultante. El intérprete también puede efectuar algunas operaciones de E/S como efectos secundarios.

Dicho esto, la opción más obvia y clásica de qué hacer con el AST es simplemente atravesarlo (a menudo en *post-orden)*, visitando cada nodo y haciendo lo que este significa: imprimir una cadena, sumar dos números, ejecutar el cuerpo de una función, todo *on the fly* (sobre la marcha). A los intérpretes que trabajan de esta manera se los denomina "intérpretes recorredores de árbol" (*tree-walking interpreters*) y son el arquetipo de los intérpretes. A veces, su paso de evaluación está precedido por pequeñas optimizaciones que reescriben el AST (por ejemplo, eliminando todas las variables declaradas pero no utilizadas) o lo convierten en otra *representación intermedia* (RI) más adecuada para la evaluación recursiva y repetida.

Otros intérpretes también atraviesan el AST, pero en lugar de interpretar el propio AST, primero lo convierten a *bytecode*. El *bytecode* es una RI mucho más densa del AST. El formato exacto del *bytecode* y qué *opcodes* (códigos de operación) lo componen son características muy variables y dependen de los lenguajes de programación huésped y anfitrión. En general, sin embargo, los *opcodes* son bastante similares a los mnemónicos de la mayoría de los lenguajes ensambladores; es una apuesta segura decir que la mayoría de las definiciones de *bytecode* contienen los *opcodes* `push` y `pop` para hacer operaciones de pila. Pero el *bytecode* no es un código de máquina nativo, ni es lenguaje ensamblador. No puede y no será ejecutado por el sistema operativo y la CPU de la máquina en la que se ejecuta el intérprete. En cambio, es interpretado por una *máquina virtual* que es parte del intérprete. De la misma forma que VMware y VirtualBox emulan máquinas y CPU reales, estas máquinas virtuales emulan una máquina que comprende cierto formato de *bytecode* en particular. Este enfoque puede generar grandes beneficios de rendimiento.

Existe una variación de esta estrategia que no involucra un AST. En lugar de construir un AST, el *parser* emite *bytecode* directamente. Ahora bien, ¿se trata aún de un intérprete o ya sería un compilador? Emitir un *bytecode* que luego se interpreta (¿o habría que decir "se ejecuta"?), ¿no es una forma de compilación? La línea se vuelve borrosa. Y para hacerlo aún más confuso, también existe esto: algunas implementaciones de lenguajes de programación analizan el código fuente, construyen un AST y convierten este AST en *bytecode*. Pero en lugar de ejecutar las operaciones especificadas por el *bytecode* directamente en una máquina virtual, la máquina virtual luego traduce el *bytecode* a código de máquina nativo, justo antes de su ejecución —justo a tiempo. Este sistema se denomina intérprete/compilador JIT (por "*just in time*" o "justo a tiempo").

Otros intérpretes se saltean la compilación a *bytecode*. Atraviesan de forma recursiva el AST, pero antes de ejecutar una rama particular del mismo, el nodo se traduce a código de máquina nativo. Y luego es ejecutado. Nuevamente, "justo a tiempo".

Una ligera variación de estos últimos casos es un modo mixto de interpretación en el que el intérprete evalúa de forma recursiva el AST y solo después de detectar que se ha evaluado una rama particular del AST varias veces, se la traduce a código de máquina.

Increíble, ¿no? Tantas maneras diferentes de implementar la evaluación, tantos giros y variaciones.

La elección de qué estrategia elegir depende en gran medida de las necesidades de rendimiento y portabilidad, del lenguaje de programación que se está interpretando y de hasta dónde se está dispuesto a llegar. Un "intérprete recorredor de árbol" (*tree-walking interpreter*) que evalúe de forma recursiva un AST probablemente sea el más lento de todos los enfoques, pero es fácil de construir, ampliar y estudiar, y será tan portátil como el lenguaje en el que se lo implemente.

Un intérprete que traduzca a *bytecode* y use una máquina virtual para evaluar dicho *bytecode* será mucho más rápido. Pero también más complicado y más difícil de construir. Si a la mezcla se le agrega la compilación a código de máquina JIT, también será necesario soportar varias arquitecturas de máquina, para que el intérprete, por ejemplo, funcione tanto en una CPU ARM como en una x86.

Todos estos enfoques se pueden encontrar en los lenguajes de programación del mundo real. Y, la mayoría de las veces, el enfoque elegido va cambiando a lo largo de la vida del lenguaje. Ruby es un gran ejemplo de ello. Hasta la versión 1.8 incluida, el intérprete era un "intérprete recorredor de árbol" (*tree-walking interpreter*), ejecutando el AST mientras lo atravesaba. Pero, con la versión 1.9, se adoptó una arquitectura de máquina virtual. Ahora, el intérprete de Ruby analiza el código fuente, construye un AST y luego compila ese AST a *bytecode*, que luego se ejecuta en una máquina virtual. El aumento de rendimiento fue enorme.

El motor de JavaScript de WebKit, *JavaScriptCore*, y su intérprete llamado *Squirrelfish* también utilizaron como enfoque el recorrido del AST y la ejecución directa. Luego, en 2008, se produjo el cambio a una máquina virtual y la interpretación de *bytecode*. Hoy en día, el motor tiene cuatro (!) etapas diferentes de compilación JIT, que se activan en diferentes momentos durante la vida del programa interpretado, dependiendo de qué parte del programa necesita el mejor rendimiento.

Otro ejemplo es Lua. La implementación principal del lenguaje de programación Lua comenzó como un intérprete que traduce a *bytecode* y ejecuta el *bytecode* en una máquina virtual basada en registros. 12 años después de su lanzamiento, nació una nueva implementación del lenguaje: LuaJIT. El objetivo claro de Mike Pall, el creador de LuaJIT, era crear la implementación de Lua más rápida posible. Y lo hizo. Un formato de *bytecode* denso se traduce JIT a código de máquina altamente optimizado para diferentes arquitecturas, y así la implementación de LuaJIT supera a la implementación original de Lua en todos los puntos de referencia. Y no por poco: a veces es 50 veces más rápida.

7. DISEÑO DE INTÉRPRETES

El diseño conceptual de un programa es una vista de alto nivel de su arquitectura de software. El diseño conceptual incluye los componentes principales del programa, cómo están organizados y cómo interactúan entre sí. No dice necesariamente cómo se implementarán estos componentes. Más bien, permite primero examinar y comprender los componentes sin preocuparse por cómo eventualmente se los desarrollará.

Tanto los compiladores como los intérpretes son traductores de lenguajes de programación. Un compilador recibe un código fuente y lo traduce a código de máquina y un intérprete traduce un programa en acciones. Tales traductores, vistos en el nivel más alto, constan de un *front end* y un *back end*. Siguiendo el principio de reutilización de software, un compilador de C y un intérprete de C pueden compartir el mismo *front end*, pero cada uno tendrá un *back end* diferente.

La Figura 3 muestra un diseño conceptual para compiladores e intérpretes. Si todo está diseñado correctamente, solo en el *front end* se necesita saber en qué lenguaje están escritos los programas fuente, y solo en el *back end* se necesita distinguir entre un compilador y un intérprete.

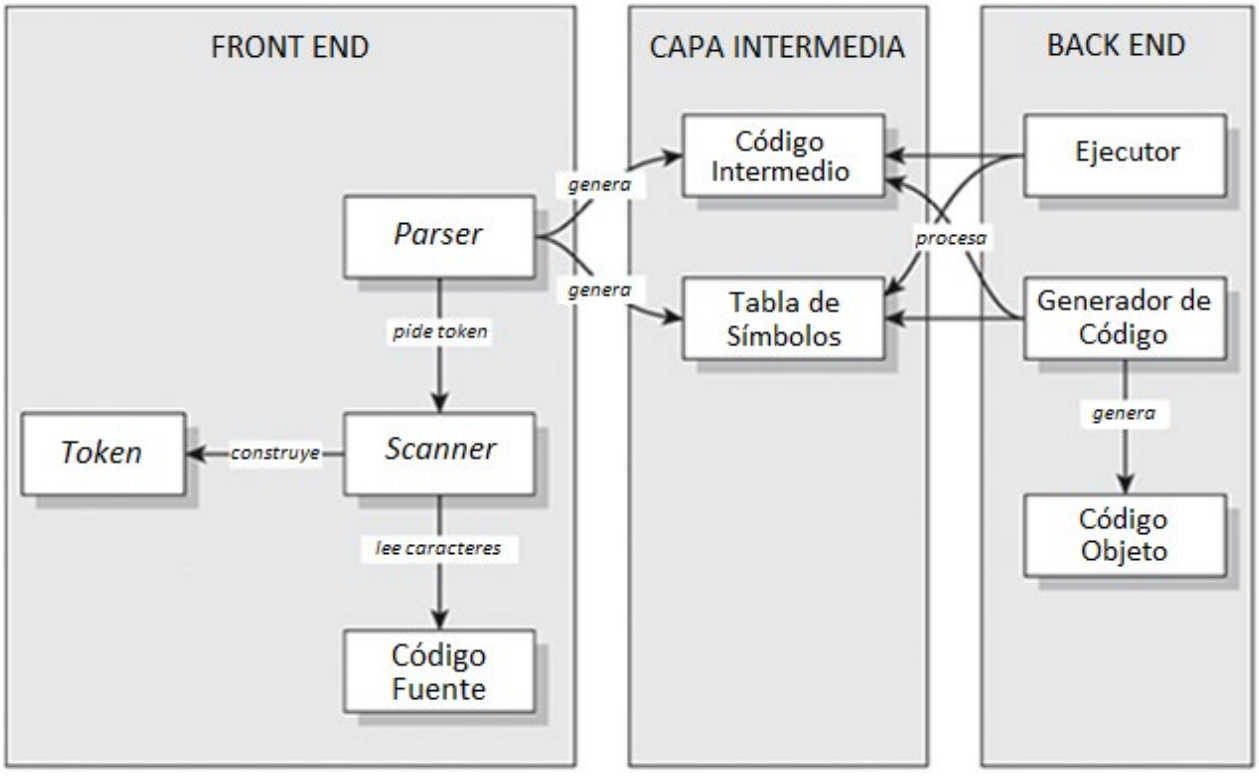


Figura 3. Un diseño conceptual para compiladores e intérpretes

En esta figura, una flecha representa una orden dada por un componente a otro. El *parser* le dice al *scanner* que obtenga el siguiente *token*. Luego, el *scanner* lee caracteres del código fuente y construye un nuevo *token*.

El *front end* de un traductor lee el programa fuente y realiza la etapa de traducción inicial. Sus componentes principales son el *parser,* el *scanner,* el *token* y el *código fuente.*

El *parser* controla el proceso de traducción en el *front end*. Solicita repetidamente al *scanner* el siguiente *token* y analiza las secuencias de *tokens* para determinar qué elementos del lenguaje de alto nivel está traduciendo, como expresiones aritméticas, sentencias de asignación o declaraciones de procedimientos. El *parser* verifica que lo que está escrito en el programa fuente sea sintácticamente correcto; en otras palabras, el *parser* detecta y marca cualquier error de sintaxis. Lo que hace el *parser* se llama *parsing* (análisis sintáctico) y suele decirse que el *parser* recibe el código fuente y lo *parsea* para traducirlo.

El *scanner* lee los caracteres del programa fuente secuencialmente y construye *tokens,* que son los elementos de bajo nivel del lenguaje fuente. Por ejemplo, los *tokens* de C incluyen *palabras reservadas* como `if`, `switch`, `case`, `for`, `do`, y `while`, *identificadores* que son nombres de variables y funciones, y *símbolos especiales* tales como `= + - *` y `/`. Lo que hace el *scanner* se llama *scanning* (análisis léxico) y suele decirse que el *scanner* recibe el programa fuente y lo *escanea* para dividirlo en *tokens*.

Al final, el compilador traduce un programa a un código objeto que es un lenguaje de máquina, por lo que el componente principal de su *back end* es un *generador de código.* Por el contrario, el intérprete ejecuta el programa, por lo que el componente principal de su *back end* es un *ejecutor.*

Si el compilador y el intérprete van a compartir el mismo *front end*, sus diferentes *back ends* necesitan una interfaz intermedia en común con el *front end*. No hay que olvidar que el *front end* realiza la etapa de traducción inicial. En el *nivel intermedio* que sirve como una interfaz en común, el *front end* genera código intermedio y una *tabla de símbolos*.

El código intermedio es una forma predigerida del programa fuente que el *back end* puede procesar de manera eficiente. El código intermedio es una estructura de datos (a menudo un árbol en memoria) que representa las sentencias del programa fuente. Este árbol de sintaxis anotado todavía contiene muchos vestigios del lenguaje fuente y del paradigma de programación al que este pertenece: las construcciones de nivel superior como los ciclos `for`, las llamadas a métodos, las listas de comprensión, las variables lógicas y las sentencias de selección todavía están representadas directamente por nodos y subárboles. Vale la pena señalar que los métodos utilizados para obtener el árbol de sintaxis anotado son en gran medida independientes del lenguaje y del paradigma.

La tabla de símbolos contiene información sobre los símbolos contenidos en el programa fuente (como, por ejemplo, los identificadores). El *back end* de un compilador procesa el código intermedio y la tabla de símbolos para generar la versión, en lenguaje de máquina, del programa fuente. El *back end* de un intérprete procesa el código intermedio y la tabla de símbolos para ejecutar directamente el programa. Al procesar el código intermedio, la elección es entre poco preprocesamiento, seguido de ejecución en un intérprete, y mucho preprocesamiento (en forma de generación de código de máquina), seguido de

ejecución en hardware. Vale la pena señalar que los detalles de estas transformaciones y las características de tiempo de ejecución que requieren dependen en gran medida del lenguaje y del paradigma —aunque las técnicas mediante las cuales se aplican a menudo serán similares.

Para una mayor reutilización del software, el código intermedio y las estructuras de la tabla de símbolos se pueden diseñar para que sean independientes del lenguaje. En otras palabras, las mismas estructuras se pueden utilizar para traducir código fuente escrito en diferentes lenguajes. Por lo tanto, el *back end* también será independiente del lenguaje; cuando procesa estas estructuras, no necesita saber ni importarle cuál era el lenguaje en que estaba escrito el código fuente.

8. REPRESENTACIONES INTERMEDIAS

El funcionamiento de los compiladores e intérpretes está organizado normalmente como una serie de pasadas. A medida que el compilador o el intérprete extrae conocimiento del código que procesa, debe transmitir esa información de una pasada a otra. Por lo tanto, el compilador o intérprete necesita una representación de todos los hechos que conoce sobre el programa. A esta representación se la llama *representación intermedia* o RI. Un compilador o intérprete puede tener una sola RI, o puede tener una serie de RI que utiliza para transformar el código fuente en código objeto. Durante la traducción, la forma RI del programa es la única forma que se utiliza: el compilador o intérprete no vuelve al código fuente; en su lugar, busca lo que necesite en la forma RI del código. Las propiedades de las RI de un compilador o intérprete tienen un efecto directo sobre lo que este puede hacerle al código.

Casi todas las fases del compilador o intérprete manipulan el programa en su forma RI. Por lo tanto, las propiedades de la RI, como los mecanismos para leer y escribir campos específicos, para encontrar hechos o anotaciones específicas y para navegar por un programa en forma de RI, tienen un impacto directo en la facilidad de escritura de las pasadas individuales y en el costo de ejecutarlas.

La mayoría de las pasadas en el compilador o intérprete consultan una RI; el *scanner* es una excepción. La mayoría de las pasadas generan una RI; las pasadas en el generador de código del compilador o en el ejecutor del intérprete pueden ser excepciones. Muchos compiladores e intérpretes modernos utilizan múltiples RI. En un compilador o intérprete estructurado en pasadas, la RI es la representación primaria y definitiva del código.

Una RI debe ser lo suficientemente expresiva para registrar todos los datos útiles que el compilador o intérprete necesite transmitir entre las pasadas. El código fuente es insuficiente para este propósito; el compilador o intérprete extrae muchos hechos que no tienen representación en el código fuente. Para registrar todos los detalles que el compilador o intérprete debe codificar, la mayoría de los constructores de compiladores o intérpretes aumentan la RI con tablas y conjuntos que registran información adicional. Estos pueden ser considerados parte de la RI.

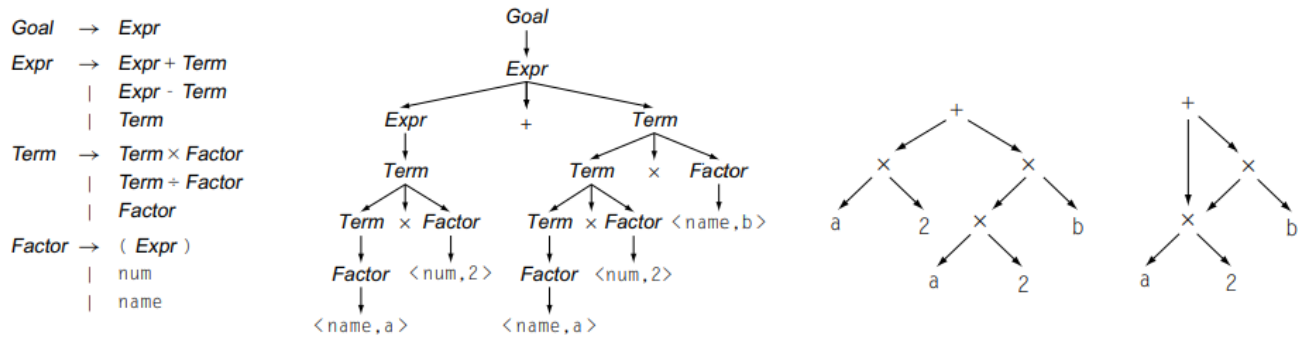
La selección de una RI apropiada para un proyecto requiere una comprensión del lenguaje en el que está escrito el código fuente, de la computadora de destino y de las propiedades de las aplicaciones que el compilador o intérprete procesarán. Por ejemplo, un traductor de fuente a fuente (*source-to-source translator*) podría usar una RI que se parezca mucho al código fuente, mientras que un compilador que produzca código ensamblador para un microcontrolador podría obtener mejores resultados con una RI similar al código ensamblador. De manera similar, un compilador de C podría necesitar anotaciones sobre valores de puntero que son irrelevantes en un compilador de Perl, y un compilador de Java debería mantener registros sobre la jerarquía de clases que no tienen contrapartida en un compilador de C.

La implementación de una RI obliga al constructor de un compilador o intérprete a centrarse en cuestiones prácticas. El compilador o intérprete necesita formas económicas de realizar las operaciones que realiza con frecuencia. Necesita formas concisas de expresar la gama completa de estructuras que pueden surgir durante la compilación o interpretación. El constructor de un compilador o intérprete también necesita mecanismos que les permitan a los seres humanos examinar, de manera fácil y directa, el programa expresado en la RI. El interés propio debe garantizar que los constructores de compiladores o intérpretes le presten atención a este punto. Por último, los compiladores o intérpretes que utilizan una RI casi siempre realizan múltiples pasadas sobre la RI. En el caso de los compiladores, la capacidad de reunir información en una pasada y utilizarlo en otra mejora la calidad del código que puede generar.

Históricamente, los compiladores e intérpretes han utilizado muchos tipos de RI. En términos generales, las RI se dividen en tres categorías estructurales:

- Las *RI lineales* se asemejan al pseudocódigo de alguna máquina abstracta. Los algoritmos iteran sobre secuencias simples y lineales de operaciones. Existen ventajas para el uso de una RI lineal. Por ejemplo, traduciendo sentencia por sentencia debería ser fácil generar código objeto (no necesariamente optimizado) a partir de un código lineal. Una amplia gama de RI lineales se utiliza hoy en día, pero también es posible adoptar lenguajes de alto nivel existentes. Por ejemplo, C ha sido usado en muchos compiladores como RI. La generación de C a partir de un árbol de sintaxis debería ser fácil, y el código resultante se podría compilar con un compilador de C existente para producir un programa ejecutable, sin tener que escribir un generador de código convencional. Generar la RI a partir del árbol de sintaxis abstracta simplemente usando un recorrido prefijo o postfijo también producirá una forma de RI lineal. A veces, estas RI eran utilizadas en los primeros compiladores, especialmente cuando el hardware de destino estaba basado en pilas. Estas representaciones tienen ventajas de simplicidad, pero no son tan populares hoy en día porque no funcionan tan bien con los potentes algoritmos de optimización que se utilizan en los compiladores actuales. Aunque todavía se utilizan algunas RI basadas en pila, que se interpretan o traducen a un código de máquina de destino (como en la máquina virtual de Java), las RI lineales no basadas en pila sino en instrucciones que contienen un operador, hasta dos argumentos y un resultado se han vuelto más populares.

- Las *RI gráficas* codifican en un gráfico el conocimiento del compilador. Los algoritmos se expresan en términos de objetos gráficos: nodos, listas o árboles. Las RI basadas en gráficos son bastante populares y, también en este caso, existe una variedad de diseños. Algunos de estos diseños de RI están relacionados con lenguajes o tipos de lenguaje particulares, algunos son buenos para ciertos tipos de optimizaciones y otros intentan ser de propósito general, apropiados para muchos lenguajes. Aunque es conveniente tener una RI lineal legible por los seres humanos, el uso de estructuras de datos más potentes puede ofrecer ventajas. El propio *árbol de sintaxis* producido por el *parser* se puede considerar como una RI. La Figura 4 muestra una gramática clásica de *expresión* (a) junto con un *árbol de sintaxis* (b) para `a × 2 + a × 2 × b`. El árbol de sintaxis es grande comparado con el código fuente porque representa la derivación completa, con un nodo para cada símbolo gramatical. El correspondiente AST (*abstract syntax tree* o *árbol de sintaxis abstracta,* fig. 4c) es fácil de construir y se puede usar directamente para la generación de código objeto. Para la generación de código es posible recorrer el árbol en *post-orden*, generando código para los hijos de un nodo antes de procesar la operación especificada en el nodo. Aunque este enfoque pueda parecer muy atractivo para la generación de código, tiene una clara desventaja: es extremadamente difícil generar código de alta calidad a partir de esta representación. Si bien el AST es más conciso que un árbol de sintaxis, conserva fielmente la estructura del código fuente original. Por ejemplo, el AST anterior contiene dos copias de la expresión `a × 2`. Un DAG (*directed acyclic graph* o *grafo acíclico dirigido,* fig. 4d) es una contracción del AST que evita esta duplicación. En un DAG, los nodos pueden tener múltiples padres y los subárboles idénticos se reutilizan. Esto hace que el DAG sea más compacto que el AST correspondiente. En sistemas reales, los DAG se utilizan por dos razones. Si las limitaciones de memoria restringen el tamaño de los programas que puede manejar el compilador, un DAG puede ayudar a reducir el uso de la memoria. Otros sistemas utilizan DAG para exponer redundancias. Aquí, el beneficio es que se produce en un código compilado mejor. Estos últimos sistemas tienden a usar el DAG como una RI temporaria —se construye el DAG, se transforma la RI definitiva eliminando las redundancias y se descarta el DAG.



(a) Gramática de *expresión* (b) Árbol de sintaxis (c) AST (d) DAG

Figura 4. Árbol de sintaxis, AST y DAG para `a × 2 + a × 2 × b` usando la gramática clásica de *expresión*

- Las *RI híbridas* combinan elementos tanto de las *RI* gráficas como de las lineales, en un intento de capturar las fortalezas y evitar las debilidades de cada una. Una representación híbrida común utiliza una RI lineal de bajo nivel para representar bloques de código lineal y un grafo para representar el flujo de control entre esos bloques. En los últimos años se han desarrollado muchas otras RI, algunas diseñadas para la optimización de propósito general y otras orientadas a tipos específicos de optimización.

9. UNA TAXONOMÍA DE LOS INTÉRPRETES

Un intérprete simula en software una computadora idealizada. Tales "computadoras" están formadas por el procesador, la memoria del código, la memoria de los datos y (generalmente) una pila. El procesador extrae instrucciones de la memoria del código, las decodifica y las ejecuta, realizando las acciones prescritas para esas instrucciones por la semántica del lenguaje. Una instrucción puede leer o escribir en la memoria de los datos o en la pila. Las llamadas a funciones guardan en la pila las direcciones de retorno para que se pueda volver a la instrucción que sigue a la llamada a la función. Hay que tener en cuenta que, a diferencia de la compilación, la interpretación requiere la presencia de los datos de entrada necesarios para el programa. Hay tres cosas a considerar al construir un intérprete: cómo almacenar datos, cómo y cuándo rastrear símbolos y cómo ejecutar instrucciones.

Los intérpretes vienen en dos variedades: iterativos y recursivos. Un intérprete iterativo trabaja con una versión linealizada del AST y requiere más preprocesamiento que un intérprete recursivo, que trabaja directamente con el AST.

La interpretación iterativa es adecuada cuando las instrucciones del lenguaje fuente son todas primitivas. Las instrucciones del programa fuente se obtienen, analizan y ejecutan, una tras otra. La interpretación iterativa es adecuada para códigos de máquina reales y abstractos, para algunos lenguajes de programación muy simples y para lenguajes de comandos.

La interpretación recursiva es necesaria si el lenguaje fuente tiene instrucciones compuestas. En este contexto, las 'instrucciones' podrían ser sentencias, expresiones y/o declaraciones. La interpretación de una instrucción puede desencadenar la interpretación de las instrucciones que la componen. Un intérprete de un lenguaje de programación de alto nivel o un lenguaje de consulta debe ser recursivo. Sin embargo, la interpretación recursiva es más lenta y compleja que la iterativa, por lo que normalmente los lenguajes de alto nivel es preferible compilarlos, o al menos traducirlos a lenguajes intermedios de más bajo nivel que sean adecuados para la interpretación iterativa.

9.1. Intérpretes iterativos

La estructura de un *intérprete iterativo* es mucho más cercana a la de una CPU que a la de un intérprete recursivo. Consiste en un bucle plano para iterar sobre una sentencia condicional múltiple

(un `switch`) que contiene un segmento de código por cada tipo de nodo; el segmento de código de un tipo de nodo dado implementa la semántica de ese tipo de nodo, como se describe en el manual de definición del lenguaje. Requiere un AST completamente anotado y enhebrado (*threaded*), y mantiene un *puntero de nodo activo,* que apunta al nodo a interpretar: el *nodo activo.* El intérprete iterativo ejecuta el segmento de código del nodo apuntado por el puntero de nodo activo; al final, este código hace que el puntero de nodo activo apunte a otro nodo, su sucesor, lo que dirige al intérprete a ese nodo, cuyo código se ejecuta a continuación, etc. El puntero de nodo activo es comparable al registro contador de programa o puntero de instrucción en una CPU. La principal diferencia es que el puntero de nodo activo se establece explícitamente en lugar de incrementarse implícitamente.

Las estructuras de datos dentro de un intérprete iterativo se parecen mucho más a las de un programa compilado que a las de un intérprete recursivo. Habrá un arreglo para contener los datos globales del programa fuente, si el lenguaje fuente los permite. Si el lenguaje fuente está basado en una pila, el intérprete iterativo mantendrá una pila para almacenar las variables locales. Las variables y otras entidades tienen direcciones, que son desplazamientos dentro de estos arreglos de memoria. La información de apilamiento y alcance, si corresponde, se coloca en la pila. No se utiliza una tabla de símbolo, excepto quizás para dar mejores mensajes de error. La pila se puede implementar convenientemente como un arreglo extensible.

Los intérpretes convencionales son iterativos: trabajan en un ciclo de búsqueda-análisis-ejecución. Esto está plasmado en el siguiente *esquema de interpretación iterativa* :

```
inicializar
hacer {
    buscar la siguiente instrucción
    analizar esa instrucción
    ejecutar esa instrucción
} mientras (aún esté en ejecución);
```

Primero, se busca una instrucción de la memoria del código o, en algunos casos, el usuario la ingresa directamente. En segundo lugar, se analiza la instrucción, descomponiéndola en sus partes. En tercer lugar, se ejecuta la instrucción. Luego se repite todo el ciclo. Normalmente, el lenguaje fuente tiene varias formas de instrucción, por lo que la ejecución de una instrucción se descompone en varios casos, un caso para cada forma de instrucción.

Comparados con los intérpretes recursivos, los intérpretes iterativos suelen ser un poco más fáciles de construir y son mucho más rápidos, pero producen diagnósticos en tiempo de ejecución menos extensos. Comparados con los compiladores, los intérpretes iterativos son mucho más fáciles de construir y, en general, permiten diagnósticos en tiempo de ejecución muy superiores. Sin embargo, ejecutar un

programa utilizando un intérprete es mucho más lento que ejecutar la versión compilada de ese programa en una máquina real. Se puede esperar que el uso de un intérprete iterativo sea entre 100 y 1000 veces más lento que ejecutar un programa compilado, pero un intérprete optimizado para aumentar la velocidad puede reducir la pérdida a quizás un factor de 30 o incluso menos, en comparación con un programa generado con un compilador optimizador. Las ventajas de la interpretación no relacionadas con la velocidad son una mayor portabilidad y una mayor seguridad, aunque estas propiedades también se pueden lograr en programas compilados. Un intérprete iterativo es el mejor medio si se necesita ejecutar programas para los que se desean diagnósticos extensos o para los que no se dispone de un compilador adecuado.

En las siguientes subsecciones, se estudiará este esquema (la interpretación iterativa) aplicado a la interpretación de código de máquina y a la interpretación de lenguajes de programación y comandos simples.

9.1.1. Interpretación iterativa de código de máquina

Un intérprete de código de máquina a menudo se llama *emulador*. Vale la pena recordar aquí que una máquina real *M* es funcionalmente equivalente a un emulador del código de máquina de *M*. La única diferencia es que una máquina real usa hardware electrónico (y quizás paralelo) para buscar, analizar y ejecutar instrucciones y, por lo tanto, es mucho más rápida que un emulador.

Una instrucción de código de máquina es esencialmente un registro, que consta de un primer campo que es la operación (generalmente llamado *código de operación* u *opcode*) seguido de cero o más campos que son los operandos. El análisis de la instrucción (o *decodificación)* consiste simplemente en descomprimir estos campos. La ejecución de la instrucción está determinada por el código de operación.

Para implementar un emulador, se emplean las siguientes técnicas simples:

- Se representa la memoria de la máquina mediante una arreglo. Si la memoria está dividida, por ejemplo, en espacios separados para el código y los datos, entonces se representa cada memoria mediante un arreglo separado.
- Se representan los registros de la máquina mediante variables. Esto se aplica igualmente a los registros visibles y ocultos. Uno de los registros, el *registro contador de programa* o *puntero de instrucción,* contendrá la dirección de la siguiente instrucción que se ejecutará. Otro, el *registro de estado,* se utilizará para controlar la terminación del programa.
- Se busca cada instrucción de la memoria del código.
- Se analiza cada instrucción aislando su código de operación y sus campos de operandos.
- Se ejecuta cada instrucción por medio de una sentencia condicional múltiple (un `switch`), con un caso por cada posible valor de *opcode*. En cada caso, emular la instrucción por medio de la actualización de la memoria y/o los registros.

Cuando se construye un intérprete de código de máquina, no importa si se está interpretando un código de máquina real o un código de máquina abstracto. Para un código de máquina abstracto, el intérprete será la única implementación. Para un código de máquina real, estarán disponibles un intérprete implementado en hardware (el *procesador)* y un intérprete implementado en software (el *emulador)*. De estos, el procesador será mucho más rápido. Pero un emulador es mucho más flexible que un procesador: se puede adaptar de forma económica para una variedad de propósitos. Se puede usar un emulador para experimentar antes de que se construya el procesador. Un emulador también se puede ampliar fácilmente con fines de diagnóstico. De todos modos, incluso cuando hay un procesador disponible, un emulador para el mismo código de máquina lo complementa muy bien.

Un *compilador interpretativo* es una combinación de compilador e intérprete, lo que brinda algunas de las ventajas de cada uno. La idea clave es traducir el programa fuente a un *lenguaje intermedio,* diseñado para cumplir los siguientes requisitos:

- debe estar en un nivel intermedio entre el lenguaje fuente y el código de máquina normal;
- sus instrucciones deben tener formatos sencillos, para que se pueden analizar de forma fácil y rápida;
- la traducción del lenguaje fuente al lenguaje intermedio debe ser fácil y rápida.

Por lo tanto, un compilador interpretativo combina una compilación rápida con una velocidad de ejecución tolerable.

Ejemplo 3. Compilación interpretativa

El *Java Development Kit* (JDK) de Oracle es una implementación de un compilador interpretativo para Java. En su corazón se encuentra la *Java Virtual Machine* (JVM), una potente máquina abstracta.

El *bytecode* o código JVM es un lenguaje intermedio orientado a Java. Proporciona instrucciones poderosas que corresponden directamente a las operaciones de Java, como la creación de objetos, la invocación de métodos y la indexación de arreglos. Por lo tanto, la traducción de Java a código JVM es fácil y rápida. A pesar de ser poderosas, las instrucciones de código JVM tienen formatos simples como las instrucciones de código de máquina, con campos de operación y campos de operandos, por lo que son fáciles de analizar. Por lo tanto, la interpretación del código JVM es relativamente rápida: "solo" unas diez veces más lenta que el código de máquina.

El JDK consta de un traductor de Java a código JVM y un intérprete de código JVM, los cuales se ejecutan en cualquier máquina real donde estén disponibles.

Los compiladores interpretativos son procesadores de lenguaje muy útiles. En las primeras etapas del desarrollo del programa, el programador bien podría pasar más tiempo compilando que ejecutando el programa, ya que está descubriendo y corrigiendo repetidamente errores sintácticos, contextuales y lógicos simples. En esa etapa, la compilación rápida es más importante que la ejecución rápida, por lo que un compilador interpretativo es ideal. Más adelante, y especialmente cuando se lo ponga en uso en

producción, el programa será ejecutado muchas veces, pero rara vez se volverá a compilar. En esa etapa, la ejecución rápida asumirá una importancia primordial, por lo que se requerirá un compilador que genere código de máquina eficiente. En Java, este problema generalmente se resuelve mediante una técnica denominada *compilación JIT (just-in-time)*.

9.1.2. Interpretación iterativa de lenguajes de comandos y de programación sencillos

Los lenguajes de comandos (como algunos lenguajes de gráficos, protocolos de red, lenguajes de procesamiento de texto, lenguajes de control de tareas y lenguajes de *shell scripting*, como el lenguaje del *shell* de UNIX) son lenguajes bastante simples. Normalmente, el usuario va ingresando una secuencia de comandos, uno por uno, y recibe cada vez una respuesta inmediata a cada uno. Cada comando se ejecuta solo una vez. Estos factores sugieren que ocurre la interpretación de cada comando tan pronto como se ingresa. De hecho, los lenguajes de comandos están diseñados específicamente para ser interpretados.

La interpretación iterativa también es posible para ciertos lenguajes de programación, siempre que un programa fuente no sea más que una secuencia de comandos primitivos. El lenguaje de programación no debe incluir comandos compuestos, es decir, comandos que contengan subcomandos.

La interpretación iterativa es llevada a cabo mediante un *intérprete dirigido por la sintaxis* que imita lo que uno hace cuando lleva a cabo el seguimiento de un código fuente manualmente. A medida que se avanza en el código, se analizan, validan y ejecutan instrucciones. Todo sucede en el *parser* porque un intérprete dirigido por la sintaxis no crea un AST ni traduce el código fuente a *bytecode* o a código de máquina. El intérprete se guía directamente a través de la sintaxis para ejecutar las sentencias.

Ejemplo 4. Intérprete de Mini-BASIC

Considere un lenguaje de programación simple con la siguiente sintaxis (expresada en EBNF):

```
Program            ::= Command*
Command            ::= Variable = Expression |
                       read Variable |
                       write Variable |
                       go Label |
                       if Expression Relational-Op Expression go Label |
                       stop
Expression         ::= primary-Expression |
                       Expression Arithmetic-Op primary-Expression
primary-Expression ::= Numeral | Variable | (Expression)
Arithmetic-Op      ::= + | - | * | /
Relational-Op      ::= = | <> | < | <= | >= | >
Variable           ::= a | b | c |... | z
Label              ::= Digit Digit*
```

En el esquema de interpretación iterativa, los comandos del lenguaje de programación corresponden a las "instrucciones". El análisis de un comando consiste en realizar el análisis sintáctico y quizás el contextual. Esto hace que el análisis sea mucho más lento y complejo que la decodificación de una instrucción en código de máquina. La ejecución está controlada por la forma de comando, según lo determinado por el análisis sintáctico.

Un programa escrito en Mini-BASIC no es más que una secuencia de comandos. Los comandos están etiquetados *implícitamente* como `0`, `1`, `2`, etc., y es posible hacer referencia a estas etiquetas en los comandos `go` e `if`. El programa puede utilizar hasta veintiséis variables, que están predeclaradas.

La semántica de los programas escritos en Mini-BASIC debería ser intuitivamente clara. Todos los valores son números reales. El programa que se muestra en la Figura 5 lee un número (en la variable `a`), calcula su raíz cuadrada con precisión de dos lugares decimales (en la variable `b`) y la muestra.

Es fácil imaginar una *máquina abstracta para Mini-BASIC.* El programa escrito en Mini-BASIC se carga en la memoria del código, con comandos sucesivos en las direcciones `0`, `1`, `2`, etc. El *registro contador de programa*, `CP`, siempre contiene la dirección del comando que se debe ejecutar a continuación.

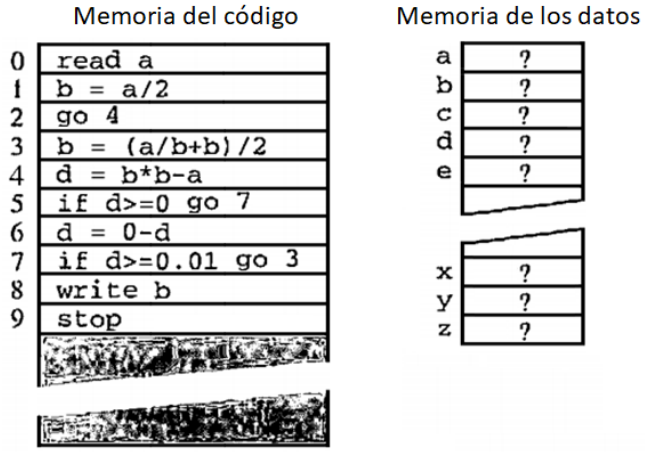


Figura 5. Máquina abstracta para Mini-BASIC: memoria del código y memoria de los datos

Los datos del programa se guardan en la memoria de los datos, compuesta por 26 celdas, una para cada variable. La Figura 5 ilustra la memoria del código y la memoria de los datos. Es necesario decidir cómo representar los comandos de Mini-BASIC en la memoria del código. Las opciones y sus consecuencias son las siguientes:

(a) El *texto* del código fuente: cada comando se debe escanear y también analizar sintácticamente (*parsear*) en tiempo de ejecución (es decir, cada vez que se lo obtiene de la memoria del código).

(b) Una secuencia de *tokens:* cada comando debe escanearse en el momento de la carga del programa y analizarse sintácticamente (*parsearse*) en tiempo de ejecución.

(c) Un *AST:* Todos los comandos se deben escanear y también analizar sintácticamente (*parsear*) en el momento de la carga del programa.

La opción (a), ilustrada en la Figura 5, ralentizaría drásticamente el intérprete. La opción (c) es mejor pero ralentizaría un poco el cargador. La opción (b) es un compromiso razonable.

Si bien una secuencia de *tokens* es una representación conveniente de un comando en la memoria, sería inconveniente para su ejecución. El análisis de un comando escaneado debería consistir en analizarlo sintácticamente (*parsearlo*) y traducirlo a una representación interna adecuada para su ejecución. En Mini-BASIC, todas las variables están predeclaradas, por lo que no hace falta detectar variables no declaradas; y solo hay un tipo de datos, por lo que tampoco es necesario realizar una verificación de tipos.

Para formas particulares de comandos y expresiones, deben implementarse las funciones/métodos `eval` y `apply` (los llamados *métodos de interpretación)*. A ambos métodos de interpretación se les debe permitir el acceso al estado de la máquina abstracta de Mini-BASIC.

Dado que un intérprete es mucho más fácil de implementar que un compilador, muchos lenguajes de programación han tenido un intérprete como su primera implementación. El BASIC original fue uno de los pocos lenguajes de programación "de alto nivel" para los que lo normal era la interpretación. Un procesador de BASIC típico permitía ingresar, editar y ejecutar programas de forma incremental. Un procesador de lenguaje de este tipo podía ejecutarse en una microcomputadora con un almacenamiento muy limitado, de ahí su popularidad en los primeros días de las microcomputadoras. Pero esto solo fue posible porque el lenguaje era muy primitivo. Sus estructuras de control eran más típicas de un lenguaje de bajo nivel, lo que lo hacía poco atractivo para los programadores serios. Más recientemente, los dialectos "estructurados" de BASIC se han vuelto más populares y la compilación se ha convertido en una alternativa a la interpretación.

9.2. Intérpretes recursivos

Los lenguajes de programación modernos son de mayor nivel que un lenguaje de programación simple. En particular, los comandos pueden ser compuestos: pueden contener subcomandos, subsubcomandos, etc.

Es posible interpretar lenguajes de programación de alto nivel. Sin embargo, el esquema de interpretación iterativa es inadecuado para dichos lenguajes. El análisis de cada comando en el programa fuente implica el análisis recursivo de sus subcomandos. Asimismo, la ejecución de cada comando implica la ejecución recursiva de sus subcomandos. Por lo tanto, uno se ve conducido inexorablemente a un proceso de dos etapas, mediante el cual se analiza todo el programa fuente antes de que pueda comenzar la interpretación propiamente dicha. Esto da lugar al siguiente *esquema de interpretación recursiva:*

```
buscar y analizar el programa
ejecutar el programa
```

donde tanto el análisis como la ejecución son recursivos.

En necesario decidir cómo se representará el programa en cada etapa. Si se proporciona en forma de código fuente, 'buscar y analizar el programa' debe realizar un análisis sintáctico y contextual del programa. Por lo tanto, un AST decorado es una representación adecuada como resultado de la etapa de análisis. Por lo tanto, 'ejecutar el programa' operará con el AST decorado del programa.

La interpretación recursiva es llevada a cabo por un *intérprete basado en árbol* que construye un árbol de alcance (*scope tree*) completo antes de ejecutar un programa. Eso significa que puede admitir tanto lenguajes de tipado estático (como Java) como lenguajes de tipado dinámico (como Python). Puede resolver todos los símbolos de forma estática antes de la ejecución. Un intérprete basado en árbol es como el *front end* de un compilador pero con un intérprete en el *back end* en lugar de un generador de código.

Un *intérprete recursivo* tiene una rutina de interpretación para cada tipo de nodo en el AST. Tal rutina de interpretación llama a otras rutinas similares, dependiendo de sus hijos; esencialmente hace lo que establece el manual de definición del lenguaje. Esta arquitectura es posible porque el significado de cada construcción del lenguaje se define en función de los significados de sus componentes. Por ejemplo, el significado de una sentencia `if` está definido por los significados de la condición, la parte `then` y la parte `else` que contiene, además de un breve párrafo en el manual que los une. Esta estructura se refleja fielmente en un intérprete recursivo, que primero interpreta la condición y luego, dependiendo del resultado, interpreta la parte `then` o la parte `else`; dado que las partes `then` y `else` pueden contener nuevamente sentencias `if`, la rutina del intérprete para las sentencias `if` será recursiva, al igual que muchas otras rutinas del intérprete. La interpretación de todo el programa comienza llamando a la rutina de interpretación principal con el nodo superior del AST como parámetro.

Otra característica importante es el *indicador de estado* que se utiliza para dirigir el flujo de control. Su componente principal es el modo de funcionamiento del intérprete. Este es un valor enumerado; su valor normal es algo así como *NormalMode,* que indica un flujo secuencial de control, pero hay otros valores disponibles para indicar saltos, excepciones, retornos de funciones y posiblemente otras formas de flujo de control. Su segundo componente es un valor en el sentido más amplio de la palabra, para proporcionar más información sobre el flujo de control no secuencial. Este puede ser un valor para el modo *ReturnMode,* un nombre de excepción más posibles valores para el modo *ExceptionMode* o una etiqueta para *JumpMode.* El indicador de estado también debe contener el nombre del archivo y el número de línea del texto en el que fue creado el indicador de estado, y posiblemente otra información de depuración.

Cada rutina de interpretación comprueba el indicador de estado después de cada llamada a otra rutina, para ver cómo continuar. Si el indicador de estado es *NormalMode,* la rutina continúa normalmente. De lo contrario, comprueba si el modo es uno que debería manejar; si lo es, lo maneja; de lo contrario, la rutina regresa inmediatamente, para permitir que la rutina responsable maneje el modo.

Las variables, constantes con nombre y otras entidades con nombre se manejan ingresándolas en la tabla de símbolos, de la forma en que son descriptas en el manual.

El intérprete recursivo típico analiza todo el programa fuente antes de que comience la ejecución. Por lo tanto, renuncia a una de las ventajas habituales de la interpretación, que es la inmediatez. Esto explica por qué la interpretación recursiva no se usa generalmente para lenguajes de programación de alto nivel. Para tales lenguajes, una mejor alternativa es recibir el código fuente del programa y traducirlo a una representación intermedia simple, que se pueda ejecutar mediante una interpretación iterativa. Dos notables excepciones a la regla general son Lisp y Prolog.

La interpretación recursiva es menos común que la iterativa. Sin embargo, esta forma de interpretación se ha asociado durante mucho tiempo con Lisp. Un programa escrito en Lisp no solo está representado por un árbol: ¡es un árbol! Varias características del lenguaje (enlace dinámico, tipado dinámico y la posibilidad de construir, en tiempo de ejecución, código de programa adicional) hacen que la interpretación de Lisp sea mucho más adecuada que la compilación. Lisp siempre ha tenido una banda devota de seguidores, pero no todos están preparados para tolerar una ejecución lenta. Un dialecto más reciente, *Scheme*, ha descartado exitosamente las características problemáticas de Lisp para hacer factible la compilación.

Cabe señalar que dos lenguajes de programación populares, BASIC y Lisp, ambos adecuados para la interpretación pero completamente diferentes, han evolucionado a lo largo de líneas de cierto modo paralelas, ¡generando dialectos estructurados adecuados para la compilación! Otro ejemplo de lenguaje de alto nivel adecuado para la interpretación es Prolog. Este lenguaje tiene una sintaxis muy simple: un programa es una colección plana de cláusulas, no tiene reglas de alcance y solo tiene pocas reglas de tipo de las que preocuparse. La interpretación es casi obligatoria por la capacidad de un programa de modificarse a sí mismo agregando y eliminando cláusulas en tiempo de ejecución.

Un intérprete recursivo se puede escribir con relativa rapidez y es útil para la creación rápida de prototipos, pero no es la arquitectura a elegir para un intérprete de trabajo pesado. Una ventaja secundaria pero importante es que puede ayudar al diseñador del lenguaje a depurar el diseño del lenguaje y su descripción. Las desventajas son la velocidad de ejecución, que puede ser inferior a lo que se podría lograr con un compilador (un factor de 1000 o más), y la falta de verificación de contexto estático: el código que no se ejecuta no se probará. La velocidad se puede mejorar realizando una memoización (del inglés: *memoization*) juiciosa: si a partir de las reglas de identificación del lenguaje se sabe, por ejemplo, que un identificador en una expresión dada siempre se identificará con el mismo tipo (lo cual es cierto en casi todos los lenguajes), entonces el tipo de un identificador se puede memoizar en su nodo en el árbol de sintaxis. En caso de ser necesario, se puede lograr una verificación completa del contexto estático haciendo una evaluación completa de los atributos antes de comenzar la interpretación; los resultados también se pueden utilizar generalmente para acelerar la interpretación.

Los intérpretes recursivos que operan con programas de alto nivel con poco o ningún preprocesamiento antes de la ejecución son más adecuados para la implementación de DSL (*domain-specific languages*) que para implementar lenguajes de programación de propósito general. Son el camino más rápido para poner en marcha un lenguaje, no son demasiado difíciles de construir y son muy flexibles. Es posible agregar nuevas instrucciones sin muchos problemas. Un DSL no es más un lenguaje (típicamente pequeño) diseñado para hacer que los usuarios sean particularmente productivos en un dominio específico, es decir, un lenguaje diseñado para una clase limitada de problemas. Algunos ejemplos son Mathematica, lenguajes de *shell scripting*, wikis, UML, XSLT, *makefiles*, gramáticas formales, lenguajes de consulta de bases de datos, lenguajes de formato de texto, lenguajes de descripción de escenas para trazadores de rayos (*ray-tracers*), lenguajes para configurar simulaciones de economía e incluso formatos de archivos de datos como CSV y XML. Lo opuesto a un DSL es un lenguaje de programación de propósito general como C, Java o Python. En el uso común, los DSL también suelen tener la connotación de ser más pequeños debido a su foco. Sin embargo, este no es siempre el caso. SQL, por ejemplo, es mucho más grande que la mayoría de los lenguajes de programación de propósito general. El lenguaje meta de un compilador para un DSL puede ser un código de máquina tradicional, pero también puede ser otro lenguaje de alto nivel para el que ya existan compiladores, una secuencia de señales de control para una máquina, o texto y gráficos formateados en algún lenguaje de control de impresoras (por ejemplo, PostScript). Los DSL a menudo son interpretados en lugar de compilados. Aun así, para leer y analizar el código fuente, los compiladores e intérpretes de DSL tendrán *front ends* que son similares a los utilizados en los compiladores e intérpretes para lenguajes de propósito general.

El principal inconveniente de los intérpretes recursivos es la eficiencia en tiempo de ejecución: no son particularmente eficientes en tiempo de ejecución. Si realmente existe una preocupación por la eficiencia de un DSL en particular o es necesario implementar un lenguaje de programación más general, estos intérpretes no son apropiados. Para reducir los recursos de memoria y acelerar la velocidad de ejecución, hace falta que el código fuente de entrada sea procesado aún más. Para ello, se necesita otra categoría de intérpretes que sea mucho más eficiente. Desafortunadamente, la eficiencia tiene el costo de una implementación más complicada. En esencia, se necesita una herramienta que reciba el código fuente de alto nivel y lo traduzca a instrucciones de bajo nivel llamadas *instrucciones de bytecode* (instrucciones cuyo código de operación cabe en 8 bits). Luego, es posible ejecutar ese *bytecode* en un intérprete eficiente llamado 'máquina virtual' (*virtual machine* o VM). Estas máquinas no deben confundirse con las máquinas virtuales que ejecutan un sistema operativo dentro de otro. La mayoría de los lenguajes actualmente populares están basados en VM (Java, JavaScript, C#, Python, Ruby 1.9).

Si se está traduciendo código fuente a *bytecode* de bajo nivel, uno se puede preguntar por qué no traducir directamente a código de máquina y saltear el intérprete por completo. El código de máquina crudo sería más eficiente. Los intérpretes de *bytecode* tienen una serie de características útiles, incluida la

portabilidad. El código de máquina es específico de una CPU, mientras que el *bytecode* se ejecuta en cualquier computadora con un intérprete compatible. Pero la razón principal por la que se evita generar código de máquina es que es bastante difícil.

Las instrucciones de la CPU tienen que ser muy simples para poder ser implementadas de manera fácil y eficiente en el hardware. El resultado suele ser un conjunto de instrucciones irregulares y muy alejadas del código fuente de alto nivel. Los intérpretes de *bytecode*, por otro lado, están diseñados específicamente para ser fácilmente tomados como metas de la generación de código. Al mismo tiempo, sus instrucciones son de un nivel lo suficientemente bajo como para poder ser interpretadas rápidamente.

10. ACERCA DEL APRENDIZAJE SOBRE COMPILADORES E INTÉRPRETES

¿Por qué aprender sobre compiladores e intérpretes? A pocas personas se les pedirá que escriban un compilador o un intérprete para un lenguaje de propósito general como C, Java o Python. Entonces, ¿por qué la mayoría de las instituciones de educación superior en informática ofrecen cursos sobre este tema y, a menudo, los hacen obligatorios? Algunas razones típicas son :

(a) Se considera un tema que uno debe conocer para tener una "buena cultura" en informática.
(b) Un buen artesano debe conocer sus herramientas, y los compiladores e intérpretes son herramientas importantes para los profesionales informáticos en general y los programadores en particular.
(c) Las técnicas utilizadas para construir un compilador o un intérprete también son útiles para otros propósitos.
(d) Existe una buena posibilidad de que un programador u otro profesional informático necesite escribir un compilador o un intérprete para un DSL (lenguaje específico de dominio).

La primera de estas razones es algo dudosa, aunque se puede decir algo a favor de "conocer sus raíces", incluso en un campo tan cambiante como lo es la informática.

La razón "b" es más convincente: los compiladores e intérpretes son piezas de código complejas y el conocimiento de cómo funcionan puede ser muy útil. Comprender cómo se traduce un programa escrito en un lenguaje de alto nivel a una forma que pueda ser ejecutada por el hardware brinda una buena idea de cómo se comportará un programa cuando se ejecute, dónde estarán los cuellos de botella de rendimiento y los costos de ejecución de las sentencias de alto nivel individuales, es decir, les permitirá a los programadores ajustar sus programas de alto nivel para conseguir una mejor eficiencia. Además, los informes de errores que proporcionan los compiladores e intérpretes suelen ser más fáciles de entender cuando uno conoce y comprende las diferentes fases de compilación e interpretación: es importante conocer la diferencia entre errores léxicos, errores de sintaxis, errores de tipo, etc.

La tercera razón también es bastante válida. Los algoritmos usados en un compilador o intérprete son relevantes para muchas otras áreas de aplicación, como para la decodificación y el análisis de textos

y el desarrollo de interfaces controladas por comandos. En particular, las técnicas utilizadas para leer el texto de un programa (*lexing* y *parsing*) y convertirlo en una forma que es fácilmente manipulable por una computadora (sintaxis abstracta), se pueden utilizar para leer y manipular cualquier tipo de texto estructurado, como documentos XML, listas de direcciones, etc. Por ejemplo, la transformación de datos de una forma sintáctica a otra se puede abordar considerando la gramática de la estructura de los datos de origen y utilizando técnicas de *parsing* tradicionales para leerlos y luego devolverlos expresados según la nueva gramática. Además, puede ser apropiado desarrollar un lenguaje simple que actúe como interfaz de usuario para un programa. La simplicidad y elegancia de algunos algoritmos de *parsing* y el hecho de basar el análisis sintáctico en una gramática especificada formalmente ayudan a producir herramientas de software confiables y sin complicaciones.

La construcción de un compilador o un intérprete brinda un área de aplicación práctica para muchos algoritmos y estructuras de datos fundamentales, permite la construcción de una pieza de software a gran escala e inherentemente modular, lo que es ideal para que su desarrollo sea llevado a cabo por un equipo de programadores. Brinda una visión de los lenguajes de programación, lo que ayuda a mostrar por qué algunos lenguajes de programación son como son y facilita el aprendizaje de nuevos lenguajes. De hecho, una de las mejores formas de aprender un lenguaje de programación es escribir un compilador o un intérprete para ese lenguaje, preferiblemente escribiéndolo en el propio lenguaje. Escribir un compilador también da una idea del diseño de las máquinas de destino, tanto reales como virtuales. Vale la pena señalar que los métodos necesarios para construir el *front end* de un compilador o intérprete son más ampliamente aplicables que los métodos necesarios para construir el *back end*, pero este último es más importante para comprender cómo se ejecuta un programa en una máquina.

Cuantos más procesadores de lenguaje uno construya, más patrones verá. La verdad es que la arquitectura de la mayoría de los procesadores de lenguaje es tremendamente similar. Un disco rayado le empezará a tocar en la cabeza cada vez que empiece a desarrollar un nuevo procesador de lenguaje: “Primero hay que construir el reconocedor de sintaxis que creará una estructura de datos en la memoria. Luego hay que recorrer la estructura de datos, recopilando información o alterando la estructura. Por último, se debe crear un generador de código o de reportes que se alimente de la estructura de datos”. Incluso uno empezará a ver patrones dentro de las propias tareas. Las tareas comparten muchos algoritmos y estructuras de datos comunes.

Una vez que uno tiene estos patrones de diseño de lenguajes y la arquitectura general en su cabeza, puede construir prácticamente lo que quiera. La construcción de un nuevo lenguaje no requiere muchos conocimientos de informática teórica. Uno puede ser escéptico al respecto, porque todos los libros que ha leído sobre el desarrollo de lenguajes se han centrado en los compiladores. Sí, la creación de un compilador para un lenguaje de programación de propósito general requiere una sólida formación en informática. Pero no todos los lenguajes usados en informática son lenguajes de programación de

propósito general. Y uno frecuentemente necesita desarrollar lectores de archivos de configuración, lectores de datos, generadores de código basados en modelos, traductores de fuente a fuente (*source-to-source translators*), analizadores de código e intérpretes.

Cuando uno habla de procesadores de lenguaje, no solo se refiere a *implementar* lenguajes con un compilador o un intérprete. Se trata de cualquier programa que procese, analice o traduzca un archivo de entrada. *Implementar* un lenguaje significa construir una aplicación que ejecute o realice tareas siguiendo órdenes escritas como sentencias en ese lenguaje. Esa es solo una de las cosas que se pueden hacer con una definición de lenguaje determinada. Por ejemplo, a partir de la definición de C, es posible construir un compilador de C, un traductor de C a Java o una herramienta para aislar fugas de memoria (*memory leaks*) en el código en C. Del mismo modo, basta pensar en todas las herramientas integradas en Eclipse, el entorno de desarrollo para Java. Más allá de compilar, Eclipse también puede refactorizar, reformatear, buscar, resaltar sintaxis, etc.

Con el fin de construir un intérprete para un lenguaje de programación de propósito general, uno necesitará implementar la recolección de basura (*garbage collection*), la ejecución de código ubicado en más de un archivo de código (*linking*), clases, bibliotecas y depuradores (*debuggers*). La mejor manera de averiguar cómo funciona todo esto es buscar en el código fuente de un intérprete existente. El código fuente de los intérpretes está disponible para casi todos los lenguajes comunes, tanto los de tipado dinámico como los de tipado estático. También se puede leer la documentación de los intérpretes de Smalltalk y Self, así como también sobre la recolección de basura y el despacho dinámico de métodos. Un buen artículo sobre máquinas de *bytecode* basadas en registros es *The Implementation of Lua 5.0*. Una de las cosas que uno descubrirá rápidamente al leer el código fuente o al crear su propio intérprete es que no es tan fácil hacer que un intérprete funcione realmente rápido. Cada ciclo de reloj de la CPU y cada acceso a la memoria deben tenerse en cuenta. Es por eso que el núcleo de la mayoría de los intérpretes rápidos es un código escrito en lenguaje ensamblador y ajustado a mano. El conocimiento de la arquitectura de la computadora, como la memoria caché y la segmentación de instrucciones por parte de la CPU, es esencial. En enero de 2009, Dan Bornstein, quien diseñó Dalvik (una VM basada en registros que ejecuta en la plataforma móvil Android de Google programas desarrollados en Java, aunque con un *bytecode* diferente al de la JVM —*Java Virtual Machine*— de Oracle, que está basada en pila), describió las increíbles acrobacias que él y su equipo realizaron para exprimir hasta la última gota de eficiencia de la CPU ARM y la memoria *flash* del teléfono. La memoria *flash* es mucho más lenta que la memoria dinámica, pero no le da amnesia cuando se corta la energía. Aparte de ser lenta, no hay mucha memoria *flash* en un teléfono. La máquina virtual Dalvik intenta compartir estructuras de datos y comprimirlas siempre que sea posible. Para representar un método dado, Dalvik usa, en promedio, un poco más de memoria que la JVM. Pero, como explica Bornstein, hacerlo vale la pena. Dalvik ejecuta muchas menos instrucciones para lograr el mismo resultado. Puede reutilizar registros para evitar

movimientos innecesarios hacia y desde la pila de operandos. Debido al costo de ejecutar cada instrucción, menos instrucciones significan mucho menos costo y un mayor rendimiento. Es posible obtener más información sobre los detalles de implementación de la máquina virtual Dalvik consultando su código fuente. También se pueden aprender algunos de los principios subyacentes buscando *threaded interpreter* en la Web.

Volviendo a las razones por las que la mayoría de las instituciones de educación superior en informática ofrecen cursos sobre compiladores e intérpretes, la razón "d" se está volviendo cada vez más importante a medida que los lenguajes específicos de dominio (DSL) van ganando popularidad. La necesidad de un DSL se presenta con frecuencia, y el conocimiento del diseño de compiladores e intérpretes puede facilitar su rápida implementación.

En resumen, escribir un compilador o intérprete es un excelente proyecto educativo y mejora las habilidades de comprensión y diseño de lenguajes de programación, diseño de algoritmos y estructuras de datos y una amplia gama de técnicas de programación. El estudio de compiladores e intérpretes hace que uno sea un mejor programador.

Bibliografía

Abelson, H., Sussman, G. J. y Sussman, J. (1996). *Structure and interpretation of computer programs* (2ª Ed.). MIT Press.

Ball, T. (2017). *Writing an interpreter in Go*. Edición del autor.

Cooper, K. D. y Torczon, L. (2012). *Engineering a compiler* (2ª Ed.). Elsevier/Morgan Kaufmann.

Friedman, D. P. y Wand, M. (2008). *Essentials of programming languages* (3ª Ed.). MIT Press.

Grune, D., van Reeuwijk, K., Bal, H., Jacobs, C. y Langendoen, K. (2012). *Modern compiler design* (2ª Ed.). Springer.

Mak, R. (2009). *Writing compilers and interpreters: A modern software engineering approach using Java* (3ra Ed.). Wiley.

Mogensen, T. Æ. (2017). *Introduction to compiler design*. Springer.

Parr, T. (2010). *Language implementation patterns: Create your own domain-specific and general programming languages*. Pragmatic Bookshelf.

Watson, D. (2017). *A practical approach to compiler construction*. Springer.

Watt, D. A. y Brown, D. F. (2000). *Programming language processors in Java: Compilers and interpreters*. Pearson Education Ltd.