Lenguajes de Programación: Principios y Paradigmas

Maurizio Gabbrielli and Simone Martini

INDICE

CAPÍTULO 1 | MÁQUINAS ABSTRACTAS

Los mecanismos de abstracción desempeñan un papel crucial en la informática porque nos permiten gestionar la complejidad inherente a la mayoría de los sistemas informáticos al aislar los aspectos importantes en un contexto específico. En el campo de los lenguajes de programación, estos mecanismos son fundamentales, tanto desde un punto de vista teórico (muchos conceptos importantes se pueden formalizar adecuadamente usando abstracciones) como en el sentido práctico, porque los lenguajes de programación hoy en día usan construcciones comunes de creación de abstracciones.

Uno de los conceptos más generales que emplean la abstracción es la *máquina abstracta*. En este capítulo, veremos cómo este concepto está estrechamente relacionado con los lenguajes de programación. También veremos cómo, sin exigirnos que entremos en detalles específicos de una implementación en particular, nos permite describir qué es una implementación de un lenguaje de programación. Para hacer esto, describiremos en términos generales lo que significa el *intérprete* y el *compilador* de un lenguaje. Finalmente, veremos cómo las máquinas abstractas pueden estructurarse en jerarquías que describen e implementan sistemas de software complejos.

1.1 LOS CONCEPTOS DE UNA MÁQUINA ABSTRACTA Y UN INTÉRPRETE

En el contexto de este libro, el término "máquina" se refiere claramente a una máquina informática. Como sabemos, una computadora electrónica y digital es una máquina física que ejecuta algoritmos que están formalizados adecuadamente para que la máquina pueda "comprenderlos". Intuitivamente, una máquina abstracta no es más que una abstracción del concepto de una computadora física.

Para la ejecución real, los algoritmos deben formalizarse adecuadamente utilizando las construcciones proporcionadas por un lenguaje de programación. En otras palabras, los algoritmos que queremos ejecutar deben representarse utilizando las instrucciones de un lenguaje de programación, L. Este lenguaje se definirá formalmente en términos de una sintaxis específica y una semántica precisa. Por el momento, la naturaleza de L no nos preocupa. Aquí, es suficiente saber que la sintaxis de L nos permite usar un conjunto finito determinado de construcciones, llamadas instrucciones,

para construir programas. Por lo tanto, un programa en L (o un programa escrito en L) no es más que un conjunto finito de instrucciones de L. Con estas observaciones preliminares, ahora presentamos una definición que es central en este capítulo.

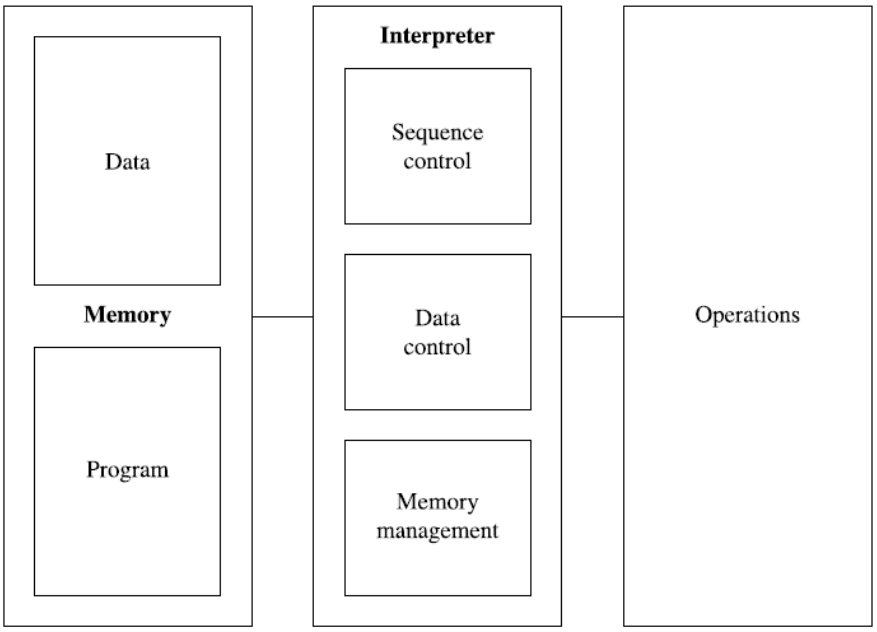


Fig. 1.1 The structure of an abstract machine

Definición 1.1 (Máquina abstracta) Suponga que se nos da un lenguaje de programación, L. Una máquina abstracta para L, denotada por M_L, es cualquier conjunto de estructuras de datos y algoritmos que pueden realizar el almacenamiento y la ejecución de programas escritos en L.

Cuando elegimos no especificar el lenguaje, L, simplemente hablaremos de la máquina abstracta, M, omitiendo el subíndice. Pronto veremos algunos ejemplos de máquinas abstractas y cómo se pueden implementar. Por el momento, detengámonos y consideremos la estructura de una máquina abstracta. Como se muestra en la figura 1.1, una máquina abstracta genérica M_L se compone de un almacén y un *intérprete*. El almacén sirve para almacenar datos y programas, mientras que el intérprete es el componente que ejecuta las instrucciones contenidas en los programas. Veremos esto más claramente en la siguiente sección.

1.1.1 EL INTÉRPRETE

Claramente, el intérprete debe realizar las operaciones que son específicas del lenguaje que está interpretando, L. Sin embargo, incluso dada la diversidad de lenguajes, es posible discernir los tipos de operación y un "método de ejecución" común

a todos los intérpretes. El tipo de operación ejecutada por el intérprete y las estructuras de datos asociadas se dividen en las siguientes categorías:

1. Operaciones para procesar datos primitivos;
2. Operaciones y estructuras de datos para controlar la secuencia de ejecución de operaciones;
3. Operaciones y estructuras de datos para controlar las transferencias de datos;
4. Operaciones y estructuras de datos para la gestión de la memoria.

Consideremos estos 4 puntos en detalle:

1. La necesidad de operaciones como las del punto uno es clara. Una máquina, incluso una abstracta, se ejecuta ejecutando algoritmos, por lo que debe tener operaciones para manipular elementos de datos primitivos. Estos artículos pueden ser representados directamente por una máquina. Por ejemplo, para las máquinas abstractas físicas, así como para las máquinas abstractas utilizadas por muchos lenguajes de programación, los números (enteros o reales) son casi siempre datos primitivos. La máquina implementa directamente las diversas operaciones requeridas para realizar operaciones aritméticas (suma, multiplicación, etc.). Estas operaciones aritméticas son, por lo tanto, operaciones primitivas en lo que respecta a la máquina abstracta.

2. Las operaciones y estructuras para el "control de secuencia" permiten controlar el flujo de ejecución de instrucciones en un programa. La ejecución secuencial normal de un programa podría tener que modificarse cuando se cumplen algunas condiciones. Por lo tanto, el intérprete utiliza estructuras de datos (por ejemplo, para mantener la dirección de la siguiente instrucción a ejecutar) que son manipuladas por operaciones específicas que son diferentes de las utilizadas para la manipulación de datos (por ejemplo, operaciones para actualizar la dirección de la siguiente instrucción a ejecutarse).

3. Las operaciones que controlan las transferencias de datos se incluyen para controlar cómo se deben transferir los operandos y los datos de la memoria al intérprete y viceversa. Estas operaciones tratan con los diferentes modos de direccionamiento de almacenamiento y el orden en que los operandos deben recuperarse desde el almacenamiento. En algunos casos, las estructuras de datos auxiliares pueden ser necesarias para manejar las transferencias de datos. Por ejemplo, algunos tipos de máquinas usan pilas (implementadas en hardware o software) para este propósito.

4. Finalmente, hay gestión de memoria. Esto se refiere a las operaciones utilizadas para asignar datos y programas en la memoria. En el caso de máquinas abstractas que son similares a las máquinas de hardware, la gestión del almacenamiento es relativamente simple. En el caso límite de una máquina basada en registros físicos que no está multiprogramada, un programa y sus datos asociados podrían asignarse en

una zona de memoria al comienzo de la ejecución y permanecer allí hasta el final, sin mucha necesidad real de administración de memoria. Las máquinas abstractas para lenguajes de programación comunes, en cambio, como se verá, utilizan técnicas de gestión de memoria más sofisticadas. De hecho, algunas construcciones en estos lenguajes directa o indirectamente causan que la memoria sea asignada o desasignada. La implementación correcta de estas operaciones requiere estructuras de datos adecuadas (por ejemplo, pilas) y operaciones dinámicas (que, por lo tanto, corren en tiempo de ejecución).

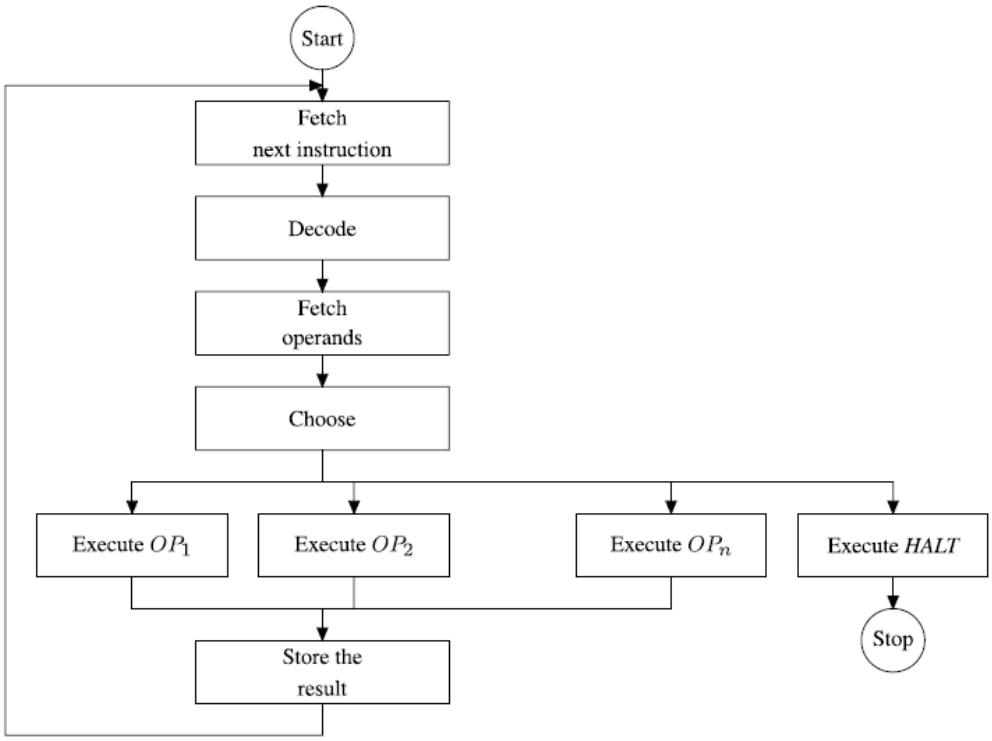


Fig. 1.2 The execution cycle of a generic interpreter

El ciclo de ejecución del intérprete, que es sustancialmente el mismo para todos los intérpretes, se muestra en la Fig. 1.2. Está organizado en términos de los siguientes pasos. Primero, obtiene la siguiente instrucción para ejecutar desde la memoria. La instrucción se decodifica para determinar la operación a realizar, así como sus operandos. Todos los operandos requeridos por la instrucción se obtienen de la memoria utilizando el método descrito anteriormente. Después de esto, se ejecuta la instrucción, que debe ser una de las primitivas de la máquina. Una vez que se ha completado la ejecución de la operación, se almacenan los resultados. Luego, a menos que la instrucción recién ejecutada sea una instrucción de detención, la ejecución pasa a la siguiente instrucción en secuencia y el ciclo se repite.

Ahora que hemos visto al intérprete, podemos definir el lenguaje que interpreta de la siguiente manera:

Definición 1.2 (Lenguaje de máquina) Dada una máquina abstracta, M_L, el lenguaje L "entendido" por el intérprete de M_L se llama *lenguaje de máquina* de M_L.

Los programas escritos en el lenguaje de máquina de M_L se almacenarán en las estructuras de almacenamiento de la máquina abstracta para que no se puedan confundir con otros datos primitivos sobre los que opera el intérprete.

Lenguajes de "Bajo Nivel" y de "Alto Nivel"

En el campo de los lenguajes de programación, los términos "bajo nivel" y "alto nivel" se utilizan a menudo para referirse, respectivamente, a la distancia del usuario humano y de la máquina.

Llamemos, por lo tanto, de *bajo nivel*, aquellos lenguajes cuyas máquinas abstractas están muy cerca o coinciden con la máquina física. A partir de finales de la década de 1940, estos lenguajes se usaron para programar las primeras computadoras, pero resultaron ser extremadamente incómodos de usar. Debido a que las instrucciones en estos lenguajes debían tener en cuenta las características físicas de la máquina, los asuntos que eran completamente irrelevantes para el algoritmo debían considerarse al escribir programas o al codificar algoritmos. Debe recordarse que, a menudo, cuando hablamos genéricamente de "lenguaje de máquina", nos referimos al lenguaje (de bajo nivel) de una máquina física. Un lenguaje particular de bajo nivel para una máquina física es su lenguaje ensamblador, que es una versión simbólica de la máquina física (es decir, que usa símbolos como ADD, MUL, etc., en lugar de sus códigos binarios de hardware asociados). Los programas en lenguaje ensamblador se traducen al código de máquina utilizando un programa llamado *ensamblador*.

Los llamados lenguajes de programación de *alto nivel* son, por otro lado, aquellos que admiten el uso de construcciones que utilizan mecanismos de abstracción apropiados para garantizar que sean independientes de las características físicas de la computadora. Por lo tanto, los lenguajes de alto nivel son adecuados para expresar algoritmos de una manera que sea relativamente fácil de entender para el usuario humano. Claramente, incluso las construcciones de un lenguaje de alto nivel deben corresponder a las instrucciones de la máquina física porque debe ser posible ejecutar programas.

1.1.2 UN EJEMPLO DE UNA MÁQUINA ABSTRACTA: LA MÁQUINA DE HARDWARE

Por lo que se ha dicho hasta ahora, debe quedar claro que el concepto de máquina abstracta se puede utilizar para describir una variedad de sistemas diferentes, que van desde máquinas físicas hasta la World Wide Web.

Como primer ejemplo de una máquina abstracta, consideremos el caso concreto de una máquina física convencional como la de la figura 1.3. Se implementa físicamente utilizando circuitos lógicos y componentes electrónicos. Llamemos a tal máquina MH_{LH} y dejemos que LH sea su lenguaje de máquina.

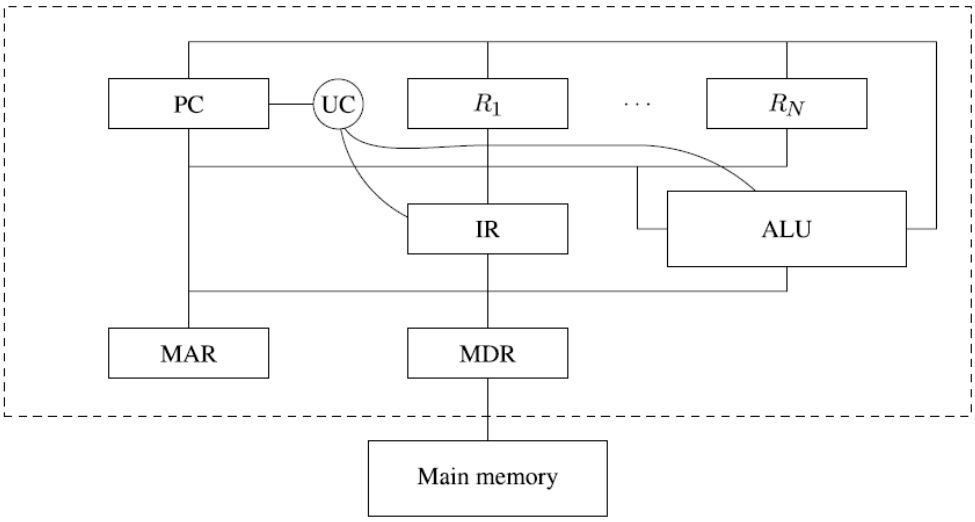


Fig. 1.3 The structure of a conventional calculator

Para este caso específico, podemos, utilizando lo que ya hemos dicho sobre los componentes de una máquina abstracta, identificar las siguientes partes.

Memoria: el componente de almacenamiento de una computadora física se compone de varios niveles de memoria. Memoria secundaria implementada utilizando componentes ópticos o magnéticos; memoria primaria, organizada como una secuencia lineal de celdas, o palabras, de tamaño fijo (generalmente un múltiplo de 8 bits, por ejemplo 32 o 64 bits); caché y los registros que son internos a la Unidad Central de Procesamiento (CPU).

La memoria física, ya sea primaria, caché o archivo de registro, permite el almacenamiento de datos y programas. Como se dijo, esto se hace usando el alfabeto binario. Los datos se dividen en unos pocos "tipos" primitivos: generalmente, tenemos números enteros, los llamados números "reales" (en realidad, un subconjunto de los racionales), caracteres y secuencias de bits de longitud fija. Dependiendo del tipo de datos, se utilizan diferentes representaciones físicas, que usan una o más palabras de memoria para cada elemento del tipo. Por ejemplo, los enteros se pueden representar con números complementarios de 1 o 2 contenidos en una sola palabra, mientras que los reales deben representarse como números de coma flotante utilizando una o dos palabras, dependiendo de si son de precisión simple o doble. Los caracteres alfanuméricos también se implementan como secuencias de números binarios codificados en un código de representación apropiado (por ejemplo, los formatos ASCII o UNICODE).

Aquí no entraremos en detalles de estas representaciones. Debemos enfatizar el hecho de que, aunque todos los datos están representados por secuencias de bits, a nivel de hardware podemos distinguir diferentes categorías, o tipos más apropiados, de datos primitivos que pueden ser manipulados directamente por las operaciones proporcionadas por el hardware. Por esta razón, estos tipos se denominan *tipos predefinidos*.

El lenguaje de la máquina física: El lenguaje, LH que ejecuta la máquina física, se compone de instrucciones relativamente simples. Una instrucción típica con dos operandos, por ejemplo, requiere una palabra de memoria y tiene el formato:

```
OpCode Operand1 Operand2
```

donde `OpCode` es un código único que identifica una de las operaciones primitivas definidas por el hardware de la máquina, mientras que `Operand1` y `Operand2` son valores que permiten ubicar los operandos haciendo referencia a las estructuras de almacenamiento de la máquina y sus modos de direccionamiento. Por ejemplo,

```
ADD R5, R0
```

podría indicar la suma del contenido de los registros R0 y R5, con el resultado almacenado en R5, mientras que

```
ADD (R5), (R0)
```

podría significar que se calcula la suma del contenido de las celdas de memoria cuyas direcciones están contenidas en R0 y R5 y el resultado se almacena en la celda cuya dirección está en R5. Cabe señalar que, en estos ejemplos, por razones de claridad, estamos utilizando códigos simbólicos como ADD, R0, (R0). En el lenguaje considerado, por otro lado, tenemos valores numéricos binarios (las direcciones se expresan en modo "absoluto"). Desde el punto de vista de la representación interna, las instrucciones no son más que datos almacenados en un formato particular.

Al igual que las instrucciones y las estructuras de datos utilizadas en la ejecución de programas, el conjunto de instrucciones posibles (con sus operaciones asociadas y modos de direccionamiento) depende de la máquina física en particular. Es posible discernir clases de máquinas con características similares. Por ejemplo, podemos distinguir entre los procesadores convencionales CISC (Computadora de conjunto de instrucciones complejas) que tienen muchas instrucciones de máquina (algunas de las cuales son bastante complejas) y las arquitecturas RISC (Computadoras de conjunto de instrucciones reducidas) en las que tienden a haber menos instrucciones que, en particular, lo suficientemente simple como para ejecutarse en unos pocos (posiblemente uno) ciclo de reloj y en forma canalizada.

Intérprete: con la estructura general de una máquina abstracta como modelo, es posible identificar los siguientes componentes de una máquina física (hardware):

1. Las operaciones para procesar datos primitivos son las operaciones aritméticas y lógicas habituales. Los implementa la ALU (Unidad Aritmética y Lógica). Se proporcionan operaciones aritméticas en números enteros y números de punto flotante, booleanos, así como turnos, pruebas, etc.

2. Para el control de la ejecución de la secuencia de instrucciones, existe el registro del Contador de programas (PC), que contiene la dirección de la siguiente instrucción a ejecutar. Es la estructura de datos principal de este componente. Las operaciones de control de secuencia utilizan específicamente este registro y típicamente incluyen la operación de incremento (que maneja el flujo normal de control) y las operaciones que modifican el valor almacenado en el registro de la PC (saltos).

3. Para manejar la transferencia de datos, se utilizan los registros de la CPU que interactúan con la memoria principal. Ellos son: el registro de dirección de datos (el MAR o el registro de direcciones de memoria) y el registro de datos (MDR o registro de datos de memoria). Además, hay operaciones que modifican el contenido de estos registros y que implementan varios modos de direccionamiento (directo, indirecto, etc.). Finalmente, hay operaciones que acceden y modifican los registros internos de la CPU.

4. El procesamiento de la memoria depende fundamentalmente de la arquitectura específica. En el caso más simple de una máquina de registro que no es multiprogramada, la gestión de la memoria es rudimentaria. El programa se carga e inmediatamente comienza a ejecutarse; permanece en la memoria hasta que termina. Para aumentar la velocidad de cálculo, todas las arquitecturas modernas utilizan técnicas de administración de memoria más sofisticadas. En primer lugar, hay niveles de memoria intermedios entre los registros y la memoria principal (es decir, memoria caché), cuya gestión necesita estructuras de datos y algoritmos especiales. En segundo lugar, casi siempre se implementa alguna forma de programación múltiple (la ejecución de un programa puede suspenderse para dar la CPU a otros programas, a fin de optimizar la gestión de los recursos). Como regla general, estas técnicas (que son utilizadas por los sistemas operativos) generalmente requieren soporte de hardware especializado para administrar la presencia de más de un programa en la memoria en cualquier momento (por ejemplo, reubicación de direcciones dinámicas).

Todas las técnicas descritas hasta ahora necesitan que el hardware proporcione estructuras y operaciones de datos de gestión de memoria específicas. Además, hay otros tipos de máquinas que corresponden a arquitecturas menos

convencionales. En el caso de una máquina que utiliza una pila (hardware) en lugar de registros, existe la estructura de datos de la pila junto con las operaciones push y pop.

El intérprete para la máquina de hardware se implementa como un conjunto de dispositivos físicos que comprenden la Unidad de Control y que admiten la ejecución del denominado ciclo de *obtención-decodificación-ejecución*, utilizando las operaciones de control de secuencia. Este ciclo es análogo al del intérprete genérico como el representado en la figura 1.2. Consiste en las siguientes fases.

En la fase de *obtención*, la siguiente instrucción a ejecutar se recupera de la memoria. Esta es la instrucción cuya dirección se encuentra en el registro de la PC (el registro de la PC se incrementa automáticamente después de que se haya obtenido la instrucción). La instrucción, que debe recordarse, está formada por un código de operación y quizás algunos operandos, luego se almacena en un registro especial, llamado registro de instrucciones.

En la fase de *decodificación*, la instrucción almacenada en el registro de instrucciones se decodifica utilizando circuitos lógicos especiales. Esto permite la interpretación correcta tanto del código de operación de la instrucción como de los modos de direccionamiento de sus operandos. Los operandos se recuperan mediante operaciones de transferencia de datos utilizando los modos de dirección especificados en la instrucción.

Finalmente, en la fase de *ejecución*, la operación de hardware primitiva se ejecuta realmente, por ejemplo, utilizando los circuitos de la ALU si la operación es aritmética o lógica. Si hay un resultado, se almacena de la manera especificada por el modo de direccionamiento y el código de operación actualmente contenido en el registro de instrucciones. El almacenamiento se realiza mediante operaciones de transferencia de datos. En este punto, la ejecución de la instrucción se completa y es seguida por la siguiente fase, en la que se busca la siguiente instrucción y el ciclo continúa (siempre que la instrucción que se acaba de ejecutar no sea una instrucción de detención).

Cabe señalar que, aunque solo sea conceptualmente, la máquina de hardware distingue los datos de las instrucciones. A nivel físico, no hay distinción entre ellos, dado que ambos están representados internamente en términos de bits. La distinción deriva principalmente del estado de la CPU. En el estado de búsqueda, cada palabra obtenida de la memoria se considera una instrucción, mientras que, en la fase de ejecución, se considera como datos. Debe observarse que, finalmente, una descripción precisa del funcionamiento de la máquina física requeriría la introducción de otros estados además de buscar, decodificar y ejecutar. Nuestra descripción solo pretende mostrar cómo una máquina física instancia el concepto general de un intérprete.

1.2 IMPLEMENTACIÓN DE UN LENGUAJE

Hemos visto que una máquina abstracta, M_L , es por definición un dispositivo que permite la ejecución de programas escritos en L. Por lo tanto, una máquina abstracta corresponde únicamente a un lenguaje, su lenguaje de máquina. Por el contrario, dado un lenguaje de programación, L, hay muchas (un número infinito) de máquinas abstractas que tienen L como lenguaje de máquina. Estas máquinas difieren entre sí en la forma en que se implementa el intérprete y en las estructuras de datos que utilizan; todos están de acuerdo, sin embargo, en el lenguaje que interpretan: L.

Implementar un lenguaje de programación L significa implementar una máquina abstracta que tiene L como lenguaje de máquina. Antes de ver qué técnicas de implementación se utilizan para los lenguajes de programación actuales, primero veremos cuáles son las diversas posibilidades teóricas para una máquina abstracta.

1.2.1 IMPLEMENTACIÓN DE UNA MÁQUINA ABSTRACTA

Cualquier implementación de una máquina abstracta, M_L debe tarde o temprano usar algún tipo de dispositivo físico (mecánico, electrónico, biológico, etc.) para ejecutar las instrucciones de L. Sin embargo, el uso de dicho dispositivo puede ser explícito o implícito. De hecho, además de la implementación "física" (en hardware) de las construcciones de M_L , incluso podemos pensar en lugar de una implementación (en software o firmware) en niveles intermedios entre M_L y el dispositivo físico subyacente. Por lo tanto, podemos reducir las diversas opciones para implementar una máquina abstracta a los siguientes tres casos y a combinaciones de ellos:

• implementación en hardware;

• simulación utilizando software;

• simulación (emulación) usando firmware.

Microprogramación

Las técnicas de microprogramación se introdujeron en la década de 1960 con el objetivo de proporcionar una amplia gama de computadoras diferentes, desde las más lentas y económicas hasta las de mayor velocidad y precio, con el mismo conjunto de instrucciones y, por lo tanto, el mismo lenguaje ensamblador (el IBM 360 fue la computadora más famosa en la que se utilizó la microprogramación). El lenguaje de máquina de las máquinas microprogramadas está en un nivel extremadamente bajo y

consiste en microinstrucciones que especifican operaciones simples para la transferencia de datos entre registros, desde y hacia la memoria principal y quizás también a través de los circuitos lógicos que implementan operaciones aritméticas. Cada instrucción en el lenguaje que se va a implementar (es decir, en el lenguaje de máquina que ve el usuario de la máquina) se simula utilizando un conjunto específico de microinstrucciones. Estas microinstrucciones, que codifican la operación, junto con un conjunto particular de microinstrucciones que implementan el ciclo de interpretación, constituyen un *microprograma* que se almacena en una memoria especial de solo lectura (que requiere un equipo especial para escribir). Este microprograma implementa el intérprete para el lenguaje (ensamblador) común a diferentes computadoras, cada una de las cuales tiene un hardware diferente. Las máquinas físicas más sofisticadas (y costosas) se construyen utilizando hardware más potente, por lo tanto, pueden implementar una instrucción utilizando menos pasos de simulación que los modelos menos costosos, por lo que funcionan a una mayor velocidad.

Es necesario introducir cierta terminología: el término utilizado para la simulación mediante microprogramación es *emulación*; el nivel al que ocurre la microprogramación se llama *firmware*.

Observemos, finalmente, que una máquina microprogramable constituye un ejemplo simple y simple de una *jerarquía* compuesta por dos máquinas abstractas. En el nivel superior, la máquina de ensamblaje está construida sobre lo que hemos llamado la máquina microprogramada. El intérprete de lenguaje ensamblador se implementa en el lenguaje del nivel inferior (como microinstrucciones), que, a su vez, es interpretado directamente por la máquina física microprogramada. Discutiremos esta situación con más profundidad en la Secta. 1.3.

IMPLEMENTACIÓN EN HARDWARE

La implementación directa de M_L en hardware siempre es posible en principio y es conceptualmente bastante simple. De hecho, se trata de utilizar dispositivos físicos como memoria, circuitos aritméticos y lógicos, buses, etc., para implementar una máquina física cuyo lenguaje de máquina coincida con L. Para hacer esto, es suficiente implementarlo en el hardware Las estructuras de datos y los algoritmos que constituyen la máquina abstracta.

La implementación de una máquina M_L en hardware tiene la ventaja de que la ejecución de programas en L será rápida porque serán ejecutados directamente por el hardware. Sin embargo, esta ventaja se compensa con varias desventajas que predominan cuando L es un lenguaje genérico de alto nivel. De hecho, las construcciones de un lenguaje de alto nivel, L, son relativamente complicadas y están muy lejos de las funciones elementales proporcionadas a nivel del circuito electrónico. Una implementación de M_L requiere, por lo tanto, un diseño más complicado para la máquina física que queremos implementar. Además, en la práctica, tal máquina, una vez implementada, sería casi imposible de modificar. No sería posible implementar en él ninguna modificación futura de L sin incurrir en costos prohibitivos. Por estas razones, en la práctica, cuando se implementa M_L , en hardware, solo se usan lenguajes de bajo nivel porque sus construcciones están muy cerca de las operaciones que pueden definirse naturalmente usando solo dispositivos físicos. Sin embargo, es posible implementar lenguajes "dedicados" desarrollados para aplicaciones especiales directamente en hardware donde se necesitan enormes velocidades de ejecución. Este es el caso, por ejemplo, de algunos lenguajes especiales utilizados en sistemas en tiempo real.

El hecho es que hay muchos casos en los que la estructura de la máquina abstracta de un lenguaje de alto nivel ha influido en la implementación de una arquitectura de hardware, no en el sentido de una implementación directa de la máquina abstracta en hardware, sino en la elección de operaciones primitivas y estructuras de datos que permiten una implementación más simple y eficiente del intérprete de lenguaje de alto nivel. Este es el caso, por ejemplo, con la arquitectura del B5500, una computadora de la década de 1960 que fue influenciada por la estructura del lenguaje Algol.

Simulación utilizando software

La segunda posibilidad para implementar una máquina abstracta consiste en implementar las estructuras de datos y algoritmos requeridos por M_L usando programas escritos en otro lenguaje, L', que, podemos suponer, ya ha sido implementado. Usando la máquina del lenguaje L, M'_L, , podemos, de hecho, implementar la máquina M_L usando programas apropiados escritos en L' que interpretan las construcciones de L simulando la funcionalidad de M_L .

En este caso, tendremos la mayor flexibilidad porque podemos cambiar fácilmente los programas que implementan los constructos de M_L . Sin embargo, veremos un rendimiento menor que en el caso anterior porque la implementación de M_L utiliza otra máquina abstracta M'_L,, que, a su vez, debe implementarse en hardware, software o firmware, agregando un nivel adicional de interpretación.

Emulación usando Fimware

Finalmente, la tercera posibilidad es intermedia entre la implementación de hardware y software. Consiste en la simulación (en este caso, también se llama emulación) de las estructuras de datos y algoritmos para M_L en microcódigo (que presentamos brevemente en el recuadro de la página 10).

Funciones parciales

Una función f: A → B es una correspondencia entre los elementos de A y los elementos de B de tal manera que, para cada elemento *a* de A, existe un único elemento de B. Lo llamaremos f (a).

Una *función parcial*, f: A → B, también es una correspondencia entre los dos conjuntos A y B, pero puede estar indefinida para algunos elementos de A. Más formalmente: es una relación entre A y B tal que, para cada a ∈ A, si existe un elemento correspondiente b ∈ B, es único y se escribe f (a). La noción de función parcial, para nosotros, es importante porque, de forma natural, los programas definen funciones parciales. Por ejemplo, el siguiente programa (escrito en un lenguaje con sintaxis y semántica obvias y cuyo núcleo se definirá en la figura 2.11):

```
read(x);
if (x == 1) then print(x);
     else while (true) do skip
```

calcula la función parcial:

$$f(n) = \begin{cases} 1 & \text{if } x = 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Conceptualmente, esta solución es similar a la simulación en software. En ambos casos, M_L se simula utilizando programas apropiados que son ejecutados por una máquina física. Sin embargo, en el caso de la emulación de firmware, estos programas son microprogramas en lugar de programas en un lenguaje de alto nivel.

Como vimos en la caja, los microprogramas usan un lenguaje especial de muy bajo nivel (con operaciones primitivas extremadamente simples) que se almacenan en una memoria especial de solo lectura en lugar de en la memoria principal, por lo que pueden ser ejecutados por la máquina física en alta velocidad. Por esta razón, esta implementación de una máquina abstracta nos permite obtener una velocidad de ejecución que es más alta que la que se puede obtener de la simulación de software, incluso si no es tan rápida como la solución de hardware equivalente. Por otro lado, la flexibilidad de esta solución es menor que la de la simulación de software, ya que, si bien es fácil modificar un programa escrito en un lenguaje de alto nivel, la modificación del

microcódigo es relativamente complicada y requiere un hardware especial para volver a escribir La memoria en la que se almacena el microcódigo. La situación es de todos modos mejor que en el caso de implementación de hardware, dado que los microprogramas pueden modificarse.

Claramente, para que esta solución sea posible, la máquina física en la que se usa debe ser micro programable. En resumen, la implementación de M_L en hardware ofrece la mayor velocidad, pero no flexibilidad. La implementación en el software ofrece la mayor flexibilidad y la menor velocidad, mientras que el que usa el firmware es intermedio entre los dos.

1.2.2 IMPLEMENTACIÓN: EL CASO IDEAL

Consideremos un lenguaje genérico, L, que queremos implementar, o más bien, para el cual se requiere una máquina abstracta, M_L . Suponiendo que podemos excluir, por las razones que acabamos de mencionar, la implementación directa en hardware de M_L , podemos suponer que, para nuestra implementación de M_L , tenemos disponible una máquina abstracta, Mo_{Lo} , que llamaremos la máquina host, que ya está implementada (no nos importa cómo) y, por lo tanto, nos permite utilizar las construcciones de su lenguaje de máquina Lo directamente.

Intuitivamente, la implementación de L en la máquina host Mo_{Lo} se lleva a cabo utilizando una "traducción" de L a Lo. Sin embargo, podemos distinguir dos modos de implementación conceptualmente muy diferentes, dependiendo de si hay una traducción "implícita" (implementada por la simulación de construcciones de M_L por programas escritos en Lo) o una traducción explícita de programas en L a los programas correspondientes en lo. Ahora consideraremos estas dos formas en sus formas ideales. Llamaremos a estas formas ideales:

1. implementación puramente interpretada, y

2. implementación puramente compilada.

NOTACIÓN

A continuación, como se mencionó anteriormente, usamos el subíndice $_L$ para indicar que una construcción particular (máquina, intérprete, programa, etc.) se refiere al lenguaje L. Usaremos el superíndice ^{L} para indicar que un programa está escrito en el lenguaje L. Nosotros usará $Prog^L$para denotar el conjunto de todos los programas posibles que se pueden escribir en el lenguaje L, mientras que D denota el conjunto de

datos de entrada y salida (y, para simplificar el tratamiento, no hacemos distinción entre los dos).

Un programa escrito en L puede verse como una función parcial (ver el recuadro):

$$P^L: D \rightarrow D$$

tal que

$$P^L(Input) = (Output)$$

si la ejecución de P^Len la entrada de datos de *Input* termina y produce *Output* como resultado. La función no está definida si la ejecución de P^Len sus datos de entrada, *Input*, no finaliza.

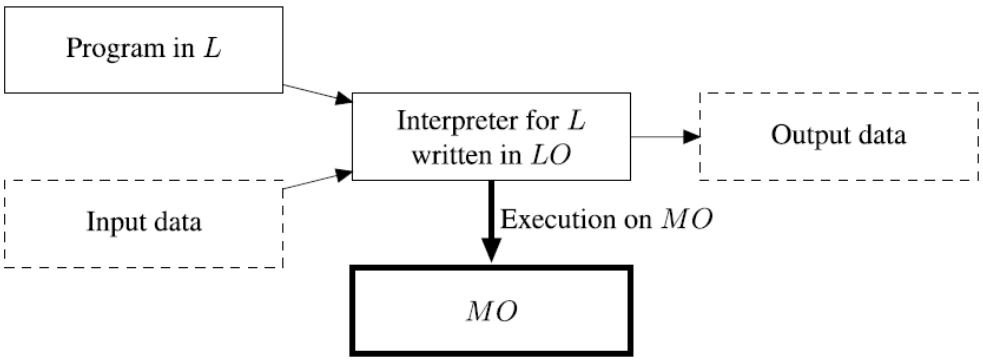


Fig. 1.4 Purely interpreted implementation

Implementación puramente interpretada: en una implementación puramente interpretada (que se muestra en la Fig. 1.4), el intérprete para M_L se implementa utilizando un conjunto de instrucciones en Lo. Es decir, se implementa un programa en Lo que interpreta todas las instrucciones de L; Este es un intérprete. Lo llamaremos I_L^{Lo}.

Una vez que se implementa dicho intérprete, ejecutando un programa P^L (escrito en el lenguaje L) en los datos de entrada especificados D ∈ D, solo necesitamos ejecutar el programa I_L^{Lo} en la máquina Mo_{Lo}, con P^L y D como datos de entrada. Más precisamente, podemos dar la siguiente definición.

Definición 1.3 (Intérprete): Un intérprete para el lenguaje L, escrito en el lenguaje Lo, es un programa que implementa una función parcial:

I_L^{Lo}: ($Prog^L$× D) → D tal que I_L^{Lo} (P^L, *Input*) = P^L (*Input*) (1.1)

El hecho de que un programa pueda considerarse como dato de entrada para otro programa no debería sorprender, dado que, como ya se dijo, un programa es solo un conjunto de instrucciones que, en el análisis final, están representadas por un cierto conjunto de símbolos (y por lo tanto por secuencias de bits). En la implementación puramente interpretada de L, por lo tanto, los programas en L no se traducen explícitamente. Solo hay un procedimiento de "decodificación". Para ejecutar una instrucción de L, el intérprete I_L^{Lo} utiliza un conjunto de instrucciones en Lo que corresponde a una instrucción en lenguaje L. Dicha decodificación no es una traducción real porque el código correspondiente a una instrucción de L se ejecuta, no se emite, por el intérprete. Cabe señalar que deliberadamente no hemos especificado la naturaleza de la máquina Mo_{Lo} . Por lo tanto, el lenguaje Lo puede ser un lenguaje de alto nivel, un lenguaje de bajo nivel o incluso un firmware.

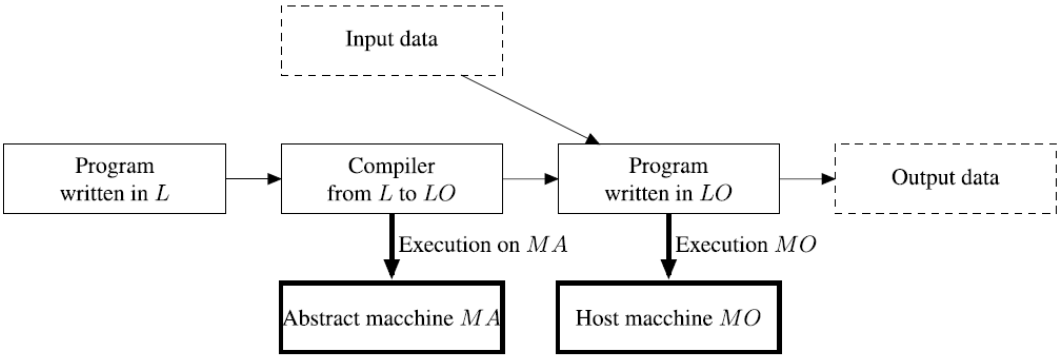


Fig. 1.5 Pure compiled implementation

Implementación puramente compilada: con la implementación puramente compilada, como se muestra en la figura 1.5, la implementación de L se lleva a cabo traduciendo explícitamente los programas escritos en L a los programas escritos en Lo. La traducción es realizada por un programa especial llamado compilador; se denota por $C_{L,Lo}$. En este caso, el lenguaje L generalmente se llama lenguaje fuente, mientras que el lenguaje Lo se llama lenguaje objeto. Para ejecutar un programa P^L (escrito en el lenguaje L) en los datos de entrada D, primero debemos ejecutar $C_{L,Lo}$, y darle Pc^{Lo}como entrada. Esto producirá un programa compilado Pc^{Lo}como su salida (escrito en Lo). En este punto, podemos ejecutar Pc^{Lo}en la máquina Mo_{Lo} proporcionándole datos de entrada D para obtener el resultado deseado.

Definición 1.4 (Compilador): Un compilador de L a Lo es un programa que implementa una función:

$C_{L,Lo}$: $Prog^{L} \rightarrow Prog^{Lo}$

tal que, dado un programa P^{L}, si

$C_{L,Lo}$ (P^{L}) = Pc^{Lo}, (1.2)

entonces, para cada *Input* ∈ D^{4}:

P^{L} (*Input*) = Pc^{Lo} (*Input*) (1.3)

Tenga en cuenta que, a diferencia de la interpretación pura, la fase de traducción descrita en (1.2) (llamada compilación) está separada de la fase de ejecución, que, por otro lado, es manejada por (1.3). De hecho, la compilación produce un programa como salida. Este programa puede ejecutarse en cualquier momento que queramos. Cabe señalar que si Mo_{Lo} es la única máquina disponible para nosotros y, por lo tanto, si Lo es el único lenguaje que podemos usar, el compilador también será un programa escrito en Lo. Sin embargo, esto no es necesario, ya que el compilador podría de hecho ejecutarse en otra máquina abstracta y esta última máquina podría ejecutar un lenguaje diferente, a pesar de que produce código ejecutable para Mo_{Lo} .

Comparando las dos técnicas

Habiendo presentado las técnicas de implementación puramente interpretadas y compiladas, ahora discutiremos las ventajas y desventajas de estos dos enfoques.

En cuanto a la implementación puramente interpretada, la principal desventaja es su *baja eficiencia*. De hecho, dado que no hay fase de traducción, para ejecutar el programa P^{L}, el intérprete I_{L}^{Lo} debe realizar una decodificación de las construcciones de L mientras se ejecuta. Por lo tanto, como parte del tiempo requerido para la ejecución de P^{L}, también es necesario agregar el tiempo requerido para realizar la decodificación. Por ejemplo, si lenguaje L contiene la construcción iterativa **`for`** y si esta construcción no está presente en lenguaje Lo, ejecutar un comando como:

```
P1: for (I = 1, I<=n, I=I+1) C;
```

el intérprete I_{L}^{Lo} debe decodificar este comando en tiempo de ejecución y, en su lugar, ejecutar una serie de operaciones que implementan el bucle. Esto podría parecerse al siguiente fragmento de código:

```
P2:
     R1 = 1
     R2 = n
L1: if R1 > R2 then goto L2
     translation of C
     ...
     R1 = R1 + 1
     goto L1
L2: ...
```

Es importante repetir que, como se muestra en (1.1), el intérprete no genera código. El código que se muestra inmediatamente arriba no es producido explícitamente por el intérprete, sino que solo describe las operaciones que el intérprete debe ejecutar en tiempo de ejecución una vez que ha decodificado el comando **`for`**. También se puede ver que por cada aparición del mismo comando en un programa escrito en L, el intérprete debe realizar una descodificación por separado; Esto no mejora el rendimiento. En nuestro ejemplo, el comando *C* dentro del bucle debe decodificarse n veces, claramente con la consiguiente ineficiencia. Como suele suceder, las desventajas en términos de eficiencia se compensan con ventajas en términos de *flexibilidad*. De hecho, interpretar las construcciones del programa que queremos ejecutar en tiempo de ejecución permite la interacción directa con lo que sea que esté ejecutando el programa.

Esto es particularmente importante, por ejemplo, porque hace que la definición de herramientas de depuración de programas sea relativamente fácil. En general, además, el desarrollo de un intérprete es más simple que el desarrollo de un compilador; Por esta razón, se prefieren soluciones interpretativas cuando es necesario implementar un nuevo lenguaje en poco tiempo. Cabe señalar, finalmente, que una implementación interpretativa permite una reducción considerable en el uso de memoria, dado que el programa se almacena solo en su versión de origen (es decir, en el lenguaje L) y no se produce ningún código nuevo, incluso si esta consideración No es particularmente importante hoy. Las ventajas y desventajas de los enfoques de compilación e interpretación de los lenguajes son duales entre sí.

La traducción del programa fuente, P^L, a un programa objeto, Pc^{Lo}, ocurre por separado de la ejecución de este último. Si descuidamos el tiempo necesario para la compilación, por lo tanto, la ejecución de Pc^{Lo}resultará más eficiente que una implementación interpretativa porque la primera no tiene la sobrecarga de la fase de decodificación de instrucciones. En nuestro primer ejemplo, el compilador traducirá el fragmento de programa `P1` al fragmento `P2`. Más tarde, cuando sea necesario, `P2` se ejecutará sin tener que decodificar nuevamente la instrucción `for`. Además, a diferencia del caso de un intérprete, el compilador realiza una decodificación de una instrucción del lenguaje L una vez, independientemente del número de veces que esta instrucción se produce en tiempo de ejecución. En nuestro ejemplo, el comando C se decodifica y traduce una vez solo en tiempo de compilación y el código producido por

este se ejecuta n veces en tiempo de ejecución. En la secta. 2.4, describiremos la estructura de un compilador, junto con las optimizaciones que se pueden aplicar al código que produce. Una de las principales desventajas del enfoque de compilación es que pierde toda la información sobre la estructura del programa fuente. Esta pérdida dificulta la interacción en tiempo de ejecución con el programa. Por ejemplo, cuando se produce un error en tiempo de ejecución, puede ser difícil determinar qué comando del programa fuente lo causó, dado que el comando se habrá compilado en una secuencia de instrucciones de lenguaje de objetos. En tal caso, puede ser difícil, por lo tanto, implementar herramientas de depuración; en general, hay menos flexibilidad que la que ofrece el enfoque interpretativo.

1.2.3 IMPEMENTACIÓN: EL CASO REAL Y LA MÁQUINA INTERMEDIA

Las implementaciones puramente compiladas e interpretadas de Burly pueden considerarse como los dos casos extremos de lo que sucede en la práctica cuando se implementa un lenguaje de programación. En realidad, en implementaciones de lenguaje real, ambos elementos casi siempre están presentes. En lo que respecta a la implementación interpretada, observamos de inmediato que cada intérprete "real" opera en una representación interna de un programa que siempre es diferente de la externa. La traducción de la notación externa de L a su representación interna se realiza mediante traducción real (compilación, en nuestra terminología) de L a un lenguaje intermedio. El lenguaje intermedio es el que se interpreta. Análogamente, en cada implementación de compilación, se simulan algunas construcciones particularmente complejas. Por ejemplo, algunas instrucciones para entrada / salida podrían traducirse al lenguaje de la máquina física, pero requerirían unos cientos de instrucciones, por lo que es preferible traducirlas en llamadas a algún programa apropiado (o directamente a las operaciones del sistema operativo), que simula tiempo de ejecución (y por lo tanto interpreta) las instrucciones de alto nivel.

¿Se pueden implementar siempre el intérprete y el compilador?

En este punto, el lector podría preguntar si la implementación de un intérprete o un compilador siempre será posible. O más bien, dado el lenguaje, L, que queremos implementar, ¿cómo podemos estar seguros de que es posible implementar un programa particular I_L^{Lo} en el lenguaje Lo que realiza la interpretación de todas las construcciones de L? ¿Cómo, además, podemos estar seguros de que es posible traducir programas de L en programas en Lo usando un programa adecuado, $C_{L,Lo}$?

La respuesta precisa a esta pregunta requiere nociones de la teoría de la computabilidad que se introducirá en el capítulo. 3. Por el momento, solo podemos responder que la existencia del intérprete y compilador está garantizada, siempre que el lenguaje, Lo, que estamos utilizando para la implementación sea lo suficientemente expresivo con respecto al lenguaje, L, que queremos para implementar. Como veremos, cada lenguaje de uso común, y por lo tanto también nuestro Lo, tiene el mismo poder expresivo (máximo) y esto coincide con un modelo abstracto particular de computación que llamaremos Máquina de Turing. Esto significa que cada algoritmo posible que pueda formularse puede implementarse mediante un programa escrito en Lo. Dado que el intérprete para L no es más que un algoritmo particular que puede ejecutar las instrucciones de L, claramente no hay dificultad teórica en la implementación del intérprete I_L^{Lo}. En lo que respecta al compilador, suponiendo que también se escriba en Lo, el argumento es similar. Dado que L no es más expresivo que Lo, debe ser posible traducir programas en L a otros en Lo de una manera que conserve su significado. Además, dado que, por supuesto, Lo permite la implementación de cualquier algoritmo, también permitirá la implementación del programa de compilación particular $C_{L,Lo}$ que implementa la traducción.

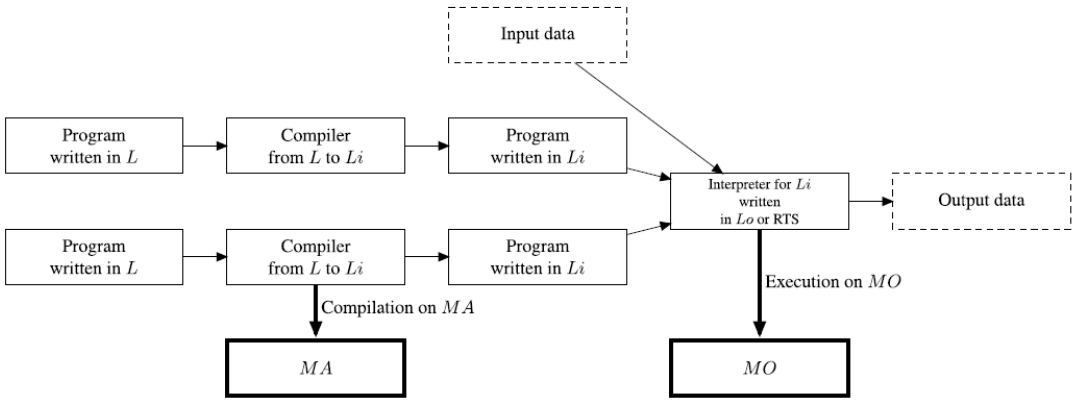


Fig. 1.6 Implementation: the real case with intermediate machine

Por lo tanto, la situación real para la implementación de un lenguaje de alto nivel es la que se muestra en la figura 1.6. Supongamos, como anteriormente, que tenemos un lenguaje L que debe implementarse y supongamos también que existe una máquina host Mo_{Lo} que ya se ha construido. Entre la máquina M_L que queremos implementar y la máquina host, existe un nivel adicional caracterizado por su propio lenguajeL_i , y por su máquina abstracta asociada, Mi_{Li} , que llamaremos lenguaje intermedio y máquina intermedia, respectivamente.

Como se muestra en la Fig. 1.6, tenemos un compilador $C_{L,Li}$ que traduce L a L_i y un intérprete I_{Li}^{Lo} que se ejecuta en la máquina Mo_{Lo} (que simula la máquina Mi_{Li}). Para ejecutar un programa genérico, P^L, el compilador primero debe traducir el programa a un programa de lenguaje intermedio, Pi^{Li}. A continuación, este programa lo ejecuta el intérprete I_{Li}^{Lo}. Cabe señalar que, en la figura, hemos escrito "intérprete o soporte de tiempo de ejecución (RTS)" porque no siempre es necesario implementar todo el intérprete I_{Li}^{Lo}. En el caso en que el lenguaje intermedio y el lenguaje de la máquina host no estén demasiado distantes, podría ser suficiente utilizar el intérprete de la máquina host, ampliado por programas adecuados, que se conocen como soporte de tiempo de ejecución, para simular la máquina intermedia.

Dependiendo de la distancia entre el nivel intermedio y el nivel de origen o host, tendremos diferentes tipos de implementación. Resumiendo, podemos identificar los siguientes casos:

1. M_L = Mi_{Li} : implementación puramente interpretada.

2. $M_L \neq Mi_{Li} \neq Mo_{Lo}$.

   (a) Si el intérprete de la máquina intermedia es sustancialmente diferente del intérprete de Mo_{Lo} , diremos que tenemos una implementación de un tipo interpretativo.

   (b) Si el intérprete de la máquina intermedia es sustancialmente el mismo que el intérprete de Mo_{Lo} (del cual se extiende parte de su funcionalidad), diremos que tenemos una implementación de un tipo compilado.

3. Mi_{Li} = Mo_{Lo} , tenemos una implementación puramente compilada.

El primer y el último caso corresponden a los casos límite ya encontrados en la sección anterior. Estos son los casos en los que las máquinas intermedias coinciden, respectivamente, con la máquina para el lenguaje a implementar y con la máquina host.

Por otro lado, en el caso en que la máquina intermedia está presente, tenemos un tipo de implementación interpretada cuando el intérprete para la máquina intermedia es sustancialmente diferente del intérprete para Mo_{Lo} . En este caso, por lo tanto, el intérprete I_{Li}^{Lo} debe implementarse utilizando el lenguaje L_o . La diferencia entre esta solución y la puramente interpretada radica en el hecho de que no todas las construcciones de L necesitan ser simuladas. Para algunas construcciones hay directamente correspondientes en el lenguaje de la máquina host, cuando se traducen de L al lenguaje intermedio L_i , por lo que no se requiere simulación. Además, la distancia entre Mi_{Li} y Mo_{Lo} es tal que las construcciones para las cuales esto sucede son pocas en número y, por lo tanto, el intérprete para la máquina intermedia debe tener muchos de sus componentes simulados. En la implementación compilada, por otro lado, el lenguaje intermedio está más cerca de la máquina host y el intérprete lo

comparte sustancialmente. En este caso, entonces, la máquina intermedia, Mi_{Li}, se implementará utilizando la funcionalidad de Mo_{Lo}, extendida adecuadamente para manejar esas construcciones de lenguaje fuente de L que, cuando también se traducen al lenguaje intermedio L_i, no tienen un equivalente inmediato en la máquina host. Este es el caso, por ejemplo, en el caso de algunas operaciones de E/S que, incluso cuando se compilan, generalmente son simuladas por programas adecuados escritos en L_o. El conjunto de dichos programas, que amplían la funcionalidad de la máquina host y que simulan en tiempo de ejecución parte de la funcionalidad del lenguaje L_i, y por lo tanto también del lenguaje L, constituyen el llamado *soporte de tiempo de ejecución* para L.

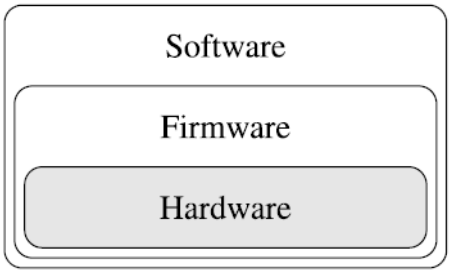
Como se puede deducir de esta discusión, la distinción entre los casos intermedios no está clara. Existe todo un espectro de tipos de implementación que van desde aquel en el que todo se simula hasta el caso en el que todo, en cambio, se traduce al lenguaje de máquina host. Qué simular y qué traducir depende en gran medida del lenguaje en cuestión y de la máquina host disponible. Está claro que, en principio, uno tenderá a interpretar las construcciones de lenguaje que están más alejadas del lenguaje máquina host y a compilar el resto. Además, como es habitual, se prefieren las soluciones compiladas en los casos en que se desea una mayor eficiencia de ejecución de los programas, mientras que el enfoque interpretado será cada vez más preferido cuando se requiera una mayor flexibilidad.

También debe tenerse en cuenta que la máquina intermedia, incluso si siempre está presente en principio, a menudo no está realmente presente. Las excepciones son casos de lenguaje que tienen definiciones formalmente establecidas de sus máquinas intermedias, junto con sus lenguajes asociados (que se hace principalmente por razones de portabilidad). La implementación compilada de un lenguaje en una nueva plataforma de hardware es una tarea bastante grande que requiere un esfuerzo considerable. La implementación interpretativa es menos exigente, pero requiere cierto esfuerzo y a menudo plantea problemas de eficiencia. A menudo, se desea implementar un lenguaje en muchas plataformas diferentes, por ejemplo, al enviar programas a través de una red para que puedan ejecutarse en máquinas remotas (como sucede con los llamados aplets). En este caso, es extremadamente conveniente primero compilar los programas en un lenguaje intermedio y luego implementar (interpretar) el lenguaje intermedio en las diversas plataformas.

Claramente, la implementación del código intermedio es mucho más fácil que la implementación del código fuente, dado que la compilación ya se ha llevado a cabo. Esta solución para la portabilidad de las implementaciones fue adoptada por primera vez a gran escala por el lenguaje Pascal, que se definió junto con una máquina intermedia (con su propio lenguaje, código P) que fue diseñada específicamente para este propósito. El lenguaje Java utilizó una solución similar, cuya máquina intermedia (llamada JVM — Java Virtual Machine) tiene como lenguaje de máquina el llamado Java Byte Code. Ahora se implementa en cada tipo de computadora.

Como última nota, hagamos hincapié en el hecho, que debe quedar claro a partir de lo que hemos dicho hasta ahora, que no se debe hablar de un "lenguaje interpretado" o un "lenguaje compilado", porque cada lenguaje se puede implementar utilizando cualquiera de Estas técnicas. En cambio, se debe hablar de implementaciones interpretativas o compiladas de un lenguaje.

Fig. 1.7 The three levels of a microprogrammed computer



1.3 JERARQUÍAS DE MÁQUINAS ABSTRACTAS

Sobre la base de lo que hemos visto, una computadora microprogramada, en la que se implementa un lenguaje de programación de alto nivel, se puede representar como se muestra en la figura 1.7. Cada nivel implementa una máquina abstracta con su propio lenguaje y su propia funcionalidad.

Este esquema puede extenderse a un número arbitrario de niveles y, por lo tanto, se genera una jerarquía, incluso si no siempre es explícita. Esta jerarquía se usa en gran medida en el diseño de software. En otras palabras, las jerarquías de máquinas abstractas se usan a menudo en las que cada máquina explota la funcionalidad del nivel inmediatamente inferior y agrega una nueva funcionalidad propia para el nivel inmediatamente superior. Hay muchos ejemplos de jerarquías de este tipo. Por ejemplo, existe la simple actividad de programar. Cuando escribimos un programa P en un lenguaje, L, en esencia, no estamos haciendo más que definir un nuevo lenguaje, L_P (y, por lo tanto, una nueva máquina abstracta) compuesto por las (nuevas) funcionalidades que P proporciona al usuario a través de su interfaz. Por lo tanto, dicho programa puede ser utilizado por otro programa, que definirá nuevas funcionalidades y, por lo tanto, un nuevo lenguaje, etc. Cabe señalar que, en términos generales, también podemos hablar de máquinas abstractas cuando se trata de un conjunto de comandos que, estrictamente hablando, no constituyen un lenguaje de programación real. Este es el caso de un programa, con la funcionalidad de un sistema operativo o con la funcionalidad de un nivel de middleware en una red informática.

En el caso general, por lo tanto, podemos imaginar una jerarquía de máquinas M_{L0}, M_{L1},. . . , M_{Ln}. La máquina genérica, M_{Li}, se implementa explotando la funcionalidad (que es el lenguaje) de la máquina inmediatamente debajo (M_{Li-1}). Al mismo tiempo, M_{Li} proporciona su propio lenguaje L_i a la máquina por encima de M_{Li+1}, que, al explotar ese lenguaje, utiliza la nueva funcionalidad que M_{Li} proporciona con respecto a los niveles inferiores. A menudo, dicha jerarquía también tiene la tarea de enmascarar los niveles inferiores. M_{Li} no puede acceder directamente a los recursos proporcionados por las máquinas que se encuentran debajo de él, pero solo puede utilizar cualquier lenguaje que proporcione L_{i-1}.

La estructuración de un sistema de software en términos de capas de máquinas abstractas es útil para controlar la complejidad del sistema y, en particular, permite un grado de independencia entre las diversas capas, en el sentido de que cualquier modificación interna de la funcionalidad de una capa no tiene (o no debería tener) ninguna influencia en las otras capas. Por ejemplo, si usamos un lenguaje de alto nivel, L, que usa los mecanismos de manejo de archivos de un sistema operativo, cualquier modificación a estos mecanismos (mientras la interfaz sigue siendo la misma) no tiene ningún impacto en los programas escritos en L.

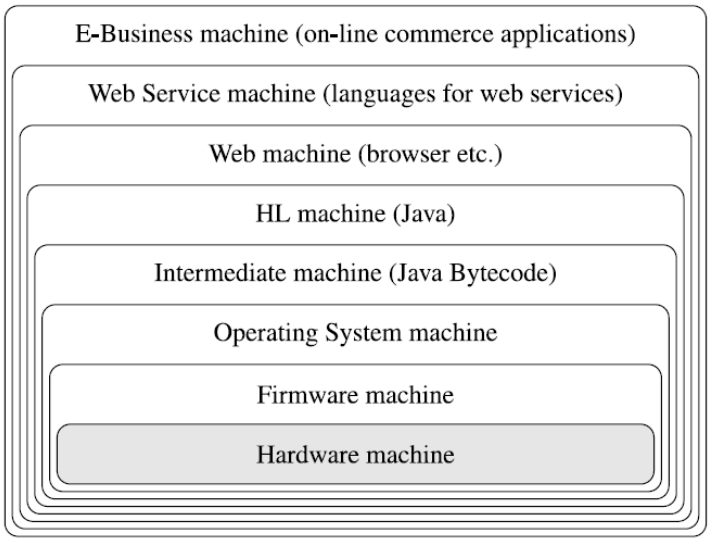


Fig. 1.8 A hierarchy of abstract machines

Un ejemplo canónico de una jerarquía de este tipo en un contexto aparentemente distante de los lenguajes de programación es la jerarquía5 de protocolos de comunicación en una red de computadoras, como, por ejemplo, el estándar ISO / OSI.

En un contexto más cercano al tema de este libro, podemos considerar el ejemplo que se muestra en la figura 1.8.

En el nivel más bajo, tenemos una computadora de hardware, implementada utilizando dispositivos electrónicos físicos (al menos, en la actualidad; en el futuro, la posibilidad

de dispositivos biológicos será algo que debe considerarse activamente). Por encima de este nivel, podríamos tener el nivel de una máquina abstracta y microprogramada. Inmediatamente arriba (o directamente arriba del hardware si el nivel de firmware no está presente), existe la máquina abstracta proporcionada por el sistema operativo que se implementa mediante programas escritos en lenguaje máquina. Tal máquina puede, a su vez, verse como una jerarquía de muchas capas (núcleo, administrador de memoria, administrador periférico, sistema de archivos, intérprete de lenguaje de comandos) que implementan funcionalidades que son progresivamente más remotas de la máquina física: comenzando con el núcleo, que interactúa con el hardware y gestiona los cambios de estado del proceso, al intérprete de comandos (o shell) que permite a los usuarios interactuar con el sistema operativo. En su complejidad, por lo tanto, el sistema operativo, por un lado, extiende la funcionalidad de la máquina física, proporcionando funcionalidades que no están presentes en la máquina física (por ejemplo, primitivas que operan en archivos) a niveles superiores. Por otro lado, enmascara algunas primitivas de hardware (por ejemplo, primitivas para manejar E/S) en las que los niveles más altos de la jerarquía no tienen interés en ver directamente. La máquina abstracta proporcionada por el sistema operativo forma la máquina host en la que se implementa un lenguaje de programación de alto nivel utilizando los métodos que discutimos en las secciones anteriores. Normalmente usa una máquina intermedia, que, en el diagrama (Fig. 1.8), es la máquina virtual Java y su lenguaje de código de bytes.

El nivel proporcionado por la máquina abstracta para el lenguaje de alto nivel que hemos implementado (Java en este caso) normalmente no es el último nivel de la jerarquía. En este punto, de hecho, podríamos tener una o más aplicaciones que juntas brinden nuevos servicios. Por ejemplo, podemos tener un nivel de "máquina web" en el que se implementan las funciones requeridas para procesar las comunicaciones web (protocolos de comunicación, visualización de código HTML, ejecución de applet, etc.). Por encima de esto, podríamos encontrar el nivel de "Servicio web" que proporciona las funciones necesarias para que los servicios web interactúen, tanto en términos de protocolos de interacción como del comportamiento de los procesos involucrados. En este nivel, se pueden implementar lenguajes verdaderamente nuevos que definan el comportamiento de los llamados "procesos empresariales" basados en servicios web (un ejemplo es el lenguaje de ejecución de procesos empresariales). Finalmente, en el nivel superior, encontramos una aplicación específica, en nuestro caso, comercio electrónico, que, si bien proporciona una funcionalidad altamente específica y restringida, también se puede ver en términos de una máquina abstracta final.

Transformación del programa y Evaluación parcial

Además de la “traducción” de programas de un lenguaje a otro, como lo hace un compilador, existen numerosas técnicas de transformación que involucran solo un

lenguaje que opera en los programas. Estas técnicas se definen principalmente con el objetivo de mejorar el rendimiento. La evaluación parcial es una de estas técnicas y consiste en evaluar un programa contra una entrada para producir un programa especializado con respecto a esta entrada y que sea más eficiente que el programa original. Por ejemplo, supongamos que tenemos un programa P (X, Y) que, después de procesar los datos X, realiza operaciones en los datos en Y dependiendo del resultado de trabajar en X. Si los datos, X, entrada al programa son siempre De la misma manera, podemos transformar este programa en P (Y), donde los cálculos con X ya se han realizado (antes del tiempo de ejecución) y, por lo tanto, obtener un programa más rápido.

Más formalmente, un evaluador parcial para el lenguaje L es un programa que implementa la función:

$$Peval_L: (Prog^L \times D) \rightarrow Prog^L$$

que tiene las siguientes características. Dado un programa genérico, P, escrito en L, tomando dos argumentos, el resultado de evaluar parcialmente P con respecto a una de sus *primeras entradas* D_1es:

$$Peval_L(P, D1) = P'$$

donde el programa P' (el resultado de la evaluación parcial) acepta un único argumento y es tal que, para cualquier dato de entrada, Y, tenemos:

$$I_L\big(P, (D_1, Y)\big) = I_L(P', Y)$$

donde I_L es el intérprete de lenguajes.

1.4 RESUMEN DE CAPÍTULO

El capítulo ha introducido los conceptos de máquina abstracta y los métodos principales para implementar un lenguaje de programación. En particular, hemos visto:

• La máquina abstracta: una formalización abstracta para un ejecutor genérico de algoritmos, formalizada en términos de un lenguaje de programación específico.
• El intérprete: un componente esencial de la máquina abstracta que caracteriza su comportamiento, relacionando en términos operativos el lenguaje de la máquina abstracta con el mundo físico incrustado.
• El lenguaje máquina: el lenguaje de una máquina abstracta genérica.
• Distintas tipologías lingüísticas: caracterizadas por su distancia de la máquina física.

• La implementación de un lenguaje: en sus diferentes formas, desde puramente interpretado hasta puramente compilado; El concepto de compilador es particularmente importante aquí.
• El concepto de lenguaje intermedio: esencial en la implementación real de cualquier lenguaje; Hay algunos ejemplos famosos (máquina de código P para Pascal y la máquina virtual Java).
• Jerarquías de máquinas abstractas: las máquinas abstractas pueden estar compuestas jerárquicamente y muchos sistemas de software pueden verse en tales términos.