

Capítulo 2: Conceitos fundamentais

2.1 Processadores de linguagens: compiladores, interpretadores, máquinas virtuais.

Usuários distintos entendem um computador de forma distinta, de acordo com a interface definida pelo software que costumam utilizar. Nos casos extremos, é possível que dois usuários da mesma máquina não consigam sequer uma base comum para trocar impressões sobre a máquina que ambos utilizam, simplesmente porque cada um conhece apenas uma face, ou interface, distinta da máquina, definida pelo software e pelos periféricos que utiliza.

Exemplos de interfaces comuns são sistemas de acesso a contas bancárias, linguagens de acesso a bases de dados, processadores de texto, sistemas de reserva de passagens, planilhas eletrônicas, programas de jogos, e naturalmente, no caso que mais nos interessa aqui, as diversas linguagens de programação.

Para cada linguagem L, a máquina pode ser vista como um sistema exclusivamente dedicado à execução de programas em L. Diferentes linguagens fazem com que aparentemente tenhamos diferentes máquinas virtuais, cuja implementação não interessa ao usuário, na maioria dos casos. Em princípio, pelo menos, para quem executa um programa escrito em Pascal não faz diferença se o hardware da máquina executa o código Pascal diretamente, ou se uma tradução é feita para o código que é finalmente executado (a linguagem da máquina), possivelmente em vários passos.

A máquina virtual de uma linguagem de programação L pode sempre ser vista como a implementação de uma interface entre a máquina e o usuário, interface essa definida através de L. Essa implementação pode ser, até certo ponto, estendida ou modificada pelo usuário: se um programador de Pascal dispõe das declarações de um tipo chamado complexo, e de um conjunto de rotinas suficientes para as ações que pretende executar com valores do novo tipo, esse tipo pode ser considerado como implementado pela nova máquina virtual, obtida por extensão da anterior: um Pascal com complexos.

Na prática, a implementação de uma máquina virtual nunca é totalmente transparente (invisível): alguns aspectos da forma pela qual foi feita podem se tornar aparentes para seus usuários, através do tempo gasto para a execução, ou de mensagens provenientes de etapas intermediárias da implementação, aspectos esses que deveriam ser invisíveis.

Para implementar uma linguagem de programação diretamente numa máquina, esta deve dispor de circuitos que, para cada instrução da linguagem, se encarregam das seguintes ações:

1. busca da próxima instrução a ser executada;
2. análise da instrução, determinando a ação que deve ser executada, como devem ser obtidos os dados de entrada para a execução dessa ação, e a forma de tratamento dos seus resultados;
3. busca dos dados necessários;
4. execução da ação correspondente sobre esses dados;
5. armazenamento (ou outro tratamento adequado) dos resultados.

Mesmo para linguagens de máquina, esta implementação não precisa ser feita diretamente, e pode ser feita através de um passo intermediário de microprogramação. Neste caso, as instruções são simuladas por programas embutidos no próprio processador, ou em chips de memória exclusiva para leitura (*read-only memory*).

As duas formas básicas de implementação de uma linguagem de programação são a compilação e a interpretação, que são frequentemente usadas em combinação. Em princípio, um **compilador**¹ implementa uma linguagem fonte *traduzindo* programas escritos nessa linguagem para a linguagem objeto da máquina alvo, onde os programas vão ser executados. Em oposição, um **interpretador** examina o programa fonte, e *simula* a execução de cada instrução ou comando de forma que o seu efeito seja reproduzido corretamente, à medida que essa execução se torna necessária. Essencialmente, o funcionamento de um interpretador pode ser descrito pelas ações (1) - (5) acima. De certa maneira, podemos considerar que o hardware de um computador, ao executar um programa armazenado em sua memória, faz a interpretação desse programa, instrução a instrução.

Mostramos a seguir, de forma esquemática, o funcionamento de compiladores e interpretadores puros.

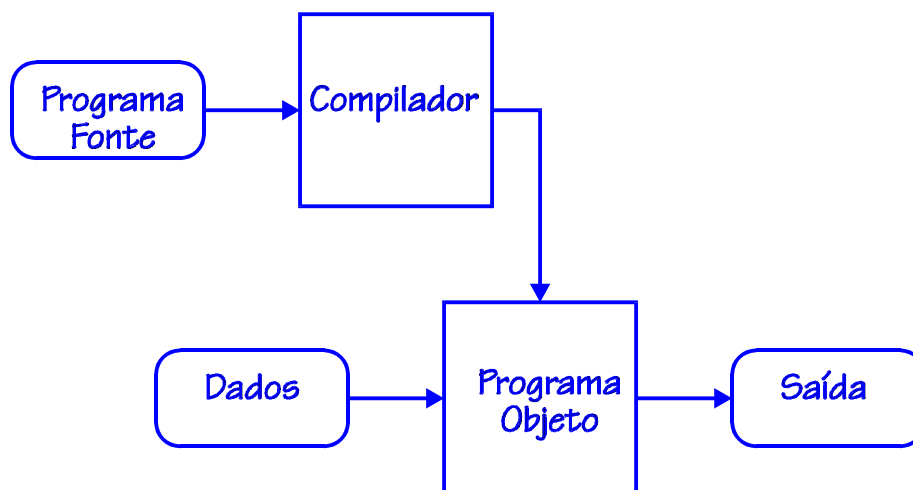


Fig. 1 — Compilação Pura

¹O nome *compilador*, na realidade, se refere à necessidade de usar rotinas de uma ou mais bibliotecas, que devem ser compiladas (encontradas e reunidas), e que vão constituir uma parte mais ou menos substancial do código objeto. Historicamente, entretanto, o nome foi associado ao processo de tradução.

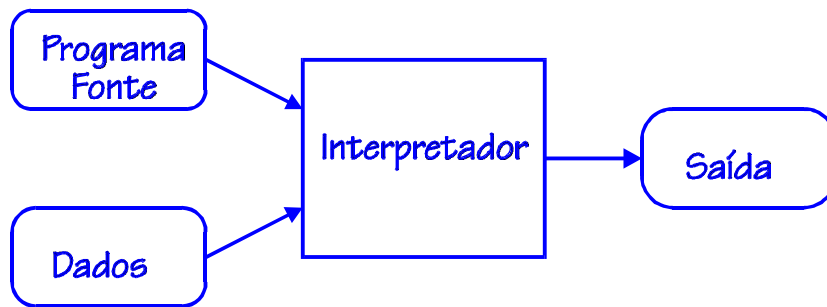


Fig. 2 — Interpretação Pura

Na prática, não existem compiladores ou interpretadores puros: cada compilador ou interpretador recebe o seu nome em função da forma de implementação que melhor o descreve.

Por exemplo, seria excessivo que um compilador traduzisse cada acesso a um arquivo em disco feito em um programa inserindo no programa objeto várias vezes o (longo) código necessário para obrigar o hardware do disco a executar a sequência de ações correspondentes. O que se faz na prática é interpretar essas instruções através de chamadas a rotinas (conhecidas como *serviços*) de um sistema operacional, que se tornam responsáveis por essas ações.

Por outro lado, muitos interpretadores efetuam uma tradução (compilação) do código fonte para uma representação interna ou código intermediário, cuja interpretação pode então ser feita com maior facilidade (Fig. 3).

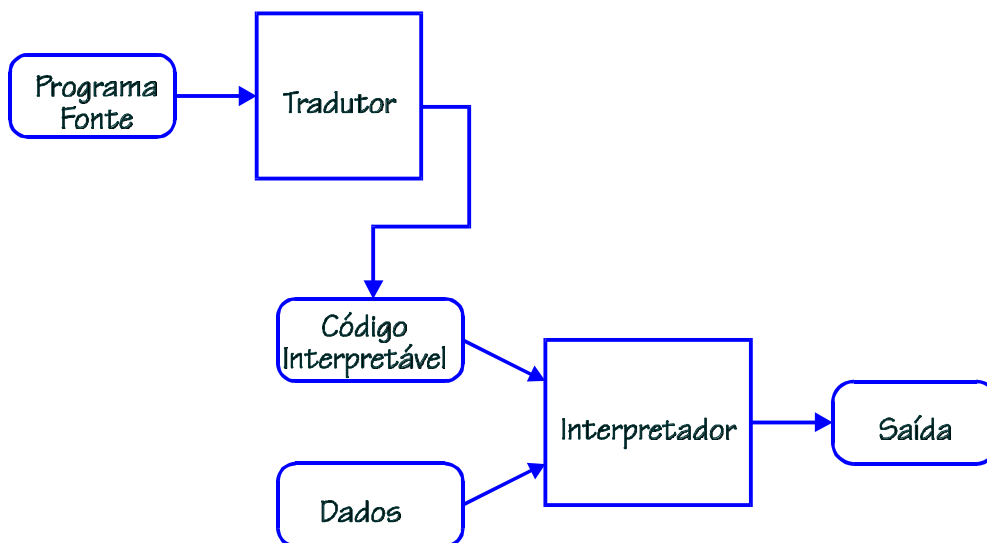


Fig. 3 — Interpretação com tradução prévia

Uma pequena confusão se pode fazer neste último caso: o programa responsável pela tradução acima mencionada é às vezes chamado de interpretador, às vezes de compilador. Quando se constrói um programa objeto executável a partir do código intermediário gerado, o programa realmente responsável pela interpretação é incluído automaticamente, e a forma de implementação pode não ficar clara. Esta situação pode ser vista na Fig. 4.

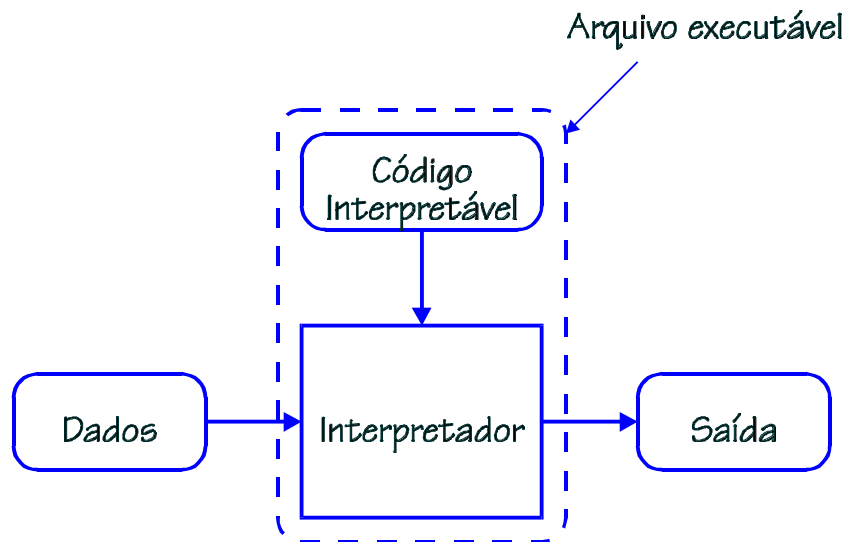


Fig. 4 — Compilação simulada por interpretação

Frequentemente, se nenhuma menção é feita, as diferenças entre interpretadores e compiladores não podem ser reconhecidas externamente com facilidade. Entretanto, em geral o interpretador é mais lento, uma vez que alguns passos (essencialmente, os passos (1) e (2) acima) são repetidos cada vez que uma instrução deve ser executada; no caso do compilador, isso não é necessário. Adicionalmente, é possível adaptar o código gerado por um compilador para cada comando de acordo com seu contexto, tirando proveito de diferenças que não podem ser tratadas com a mesma facilidade pelo interpretador. Por outro lado, em geral o controle que o interpretador tem sobre o programa que está sendo executado é maior, e é possível detectar durante a execução situações de erro invisíveis (ou imprevisíveis) para o compilador durante a fase de tradução.

2.2 O conceito de ligação.

Durante o processo de programação, é necessário estabelecer uma correspondência entre os objetos² em termos dos quais a especificação do programa foi construída, e objetos da linguagem de programação. Essa correspondência é feita através de nomes definidos pelo programador, que pela implementação devem corresponder a objetos "concretos", ou "reais", da máquina virtual, que de alguma forma vão posteriormente ser associados a posições de memória e/ou trechos de código durante a execução. A definição dessa correspondência é feita de acordo com regras especificadas para cada linguagem, e pode ser complicada, porque

1. o objeto "concreto" associado ao nome pode não existir, ou pode estar invisível (isto é, existe, mas as regras da linguagem proíbem o acesso ao objeto) em certos trechos da execução do programa;
2. o objeto pode ser "instanciado" mais de uma vez, de forma que vários objetos "concretos" existem (ou seja, tem vida) simultaneamente, correspondendo ao mesmo nome declarado pelo programador, e é necessário que a instância correta seja acessada.

²Exceto onde explicitamente mencionado, a palavra *objeto* é usada em sentido geral, e não no sentido específico associado com a expressão "orientação a objetos".

Fala-se em ligação (*binding*) para definir essa correspondência entre nomes e objetos. Conforme o caso, a correspondência pode ser definida por ocasião da programação, da tradução (compilação) ou da execução. Diz-se que a correspondência é feita *em tempo de programação*, tradução (compilação) ou execução.

Um objeto "1" de uma linguagem tem a ele associado o valor inteiro 1, e pode ser considerado pré-definido. Um objeto de nome "i" é provavelmente uma variável de tipo inteiro, e o nome "i" estará associado a uma ou mais posições da memória onde se pode armazenar um valor inteiro. Dependendo da linguagem, a situação varia: em FORTRAN, a alocação de espaço é feita estaticamente, e uma posição é escolhida para "i", durante a compilação, ficando definitivamente estabelecida a ligação entre o nome "i" e aquela posição de memória.

Em outras linguagens, como Pascal, essa ligação só poderia ser feita dinamicamente, durante a execução do programa: como podemos ter procedimentos recursivos, uma variável local a um procedimento pode ter várias instâncias, cujo número e localização vão depender da forma da execução. Note, entretanto, que, para o programador, neste caso, o nome "i" sempre corresponde a um único objeto ("a" variável "i"), e, durante a programação, as ações a serem executadas com qualquer instância da variável "i" são especificadas uma única vez.

Veremos nas seções seguintes, para diversos objetos, em diversas linguagens, as diversas formas de ligação que podem ser feitas.

2.3 Abstração: dados, controle.

O processo de programação exige do programador uma tradução entre duas linguagens:

- a *linguagem do problema*, isto é, a linguagem em que as ações a programar e os objetos a que essas ações se referem estão descritos, em alguma forma de representação do problema que se busca resolver;
- a *linguagem de programação*, que oferece ao programador um conjunto de mecanismos de programação que foram julgados adequados pelo projetista da linguagem, de acordo com a sua finalidade prevista.

A tradução entre as duas linguagens pode ser mais simples ou mais complicada em função da correspondência que possa ser feita entre os mecanismos oferecidos pela linguagem de programação e os objetos e ações usados na descrição do problema. Normalmente, a correspondência mencionada acima não é completa, e torna-se necessário fazer corresponder um objeto ou ação da linguagem do problema a um conjunto de objetos ou seqüência de ações da linguagem de programação, que o programador deve procurar manipular como unidades para que a tradução entre as duas linguagens seja corretamente efetuada. Isto pode ser facilitado quando a linguagem permite a definição e a manipulação de objetos e de ações, através de combinação de objetos e ações mais simples.

Este processo pelo qual os objetos ou ações compostos podem ser considerados como unidades é denominado **abstração**. Diz-se que uma linguagem de programação tem facilidades para abstração se é possível definir objetos e ações compostos (abstratos) e tratá-los como se sempre tivessem feito parte da linguagem, fazendo referência a eles através de seus nomes.

Praticamente todas as linguagens oferecem algum tipo de abstração. Por exemplo, a reunião de uma série de declarações e comandos, em um procedimento,

permite que se faça referência ao conjunto de ações especificadas pelos comandos (como uma ação composta), através de uma chamada do procedimento. Este é um mecanismo de abstração que, de uma forma ou de outra, faz parte de todas as linguagens de programação, e é de uso relativamente simples.

Outro mecanismo de abstração extremamente comum é o grupamento de várias variáveis em uma única, como por exemplo é feito pela declaração DIMENSION em FORTRAN ou pelo uso de tipos *array* em linguagens como Pascal ou Algol-68. Em FORTRAN, uma declaração DIMENSION X(100) corresponde à declaração de 100 variáveis reais X(1), X(2), ... X(100), que, de forma equivalente, pode ser entendida como a declaração de uma única variável X, efetivamente composta pelas 100 variáveis reunidas. Para algumas operações, fazemos referência à I-ésima componente X(I); para outras, fazemos referência a toda a variável X.

De uma forma geral, todos os mecanismos que permitem a definição de novos tipos são também mecanismos de abstração. Outros exemplos de mecanismos de abstração serão apresentados no restante deste trabalho, à medida que formos discutindo as diversas facilidades típicas oferecidas pelas linguagens de programação.

2.4 Correção de programas; a programação como uma ciência exata.

Programas (e outros objetos definidos através de linguagens de programação) são objetos definidos de maneira bastante precisa, e por essa razão se prestam à análise matemática rigorosa. Assim, uma representação do comportamento esperado de um programa pode ser construída através de objetos matematicamente bem definidos. Por exemplo, um programa pode ser representado por uma função: a correspondência entre os valores dados como entrada e os valores obtidos como saída, em todos os casos possíveis. Podemos também, diretamente a partir do programa, construir uma representação do comportamento realmente implementado. Dispondo desse modelo matemático, para demonstrar, como um teorema, a correção do programa, basta mostrar a equivalência entre as duas representações, a especificada (aquela esperada) e a implementada (aquela obtida). Isto pode ser efetivamente feito, diretamente, no caso de programas ou trechos de programas pequenos, usando técnicas específicas para isso. No caso de sistemas maiores, podem ser usadas técnicas baseadas na decomposição de um sistema grande em partes menores, dividindo o problema:

- prova-se a correção das partes componentes do sistema;
- supondo que todas as partes componentes estão corretas, prova-se a correção do sistema, isto é, prova-se a correção da interligação das partes componentes.

Parte da solução do problema pode ser automatizada, lançando-se mão de ferramentas de software auxiliares, baseadas em provadores automáticos de teoremas.

Uma técnica muito utilizada para programação (principalmente de programas ou sistemas de porte relativamente pequeno) é a de programação por *refinamentos sucessivos*, ou descendente (*top-down*). De acordo com esta técnica, definem-se, inicialmente, as funções que deve ter um módulo, que é "implementado" então com a utilização de referências a módulos menores *ainda inexistentes*. Esses módulos menores são, por sua vez, submetidos ao mesmo processo, até que num certo nível, a implementação pode ser feita exclusivamente usando facilidades disponíveis diretamente na linguagem. Por exemplo, a implementação do sistema A da Fig. 5 é feita em termos dos blocos B e C, que, por sua vez, são implementados em termos de D, E, F e G, descendo pela árvore que descreve a estrutura de blocos do sistema.

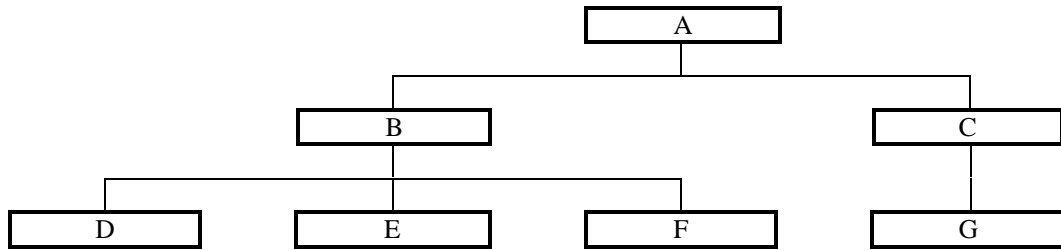


Fig. 5 — Decomposição de um sistema

Como mencionado acima, a decomposição facilita o processo de prova formal, uma vez que cada módulo pode ser provado correto tomando-se como hipótese a correção de todos os demais módulos.

Entretanto, fora dos livros-texto, raros são ainda os programas cuja correção foi provada formalmente. As técnicas normalmente usadas para validação, na prática, são ainda informais, baseadas em testes sistemáticos, ou ainda em técnicas de inspeção de código, apesar de todo o esforço de pesquisa no assunto.