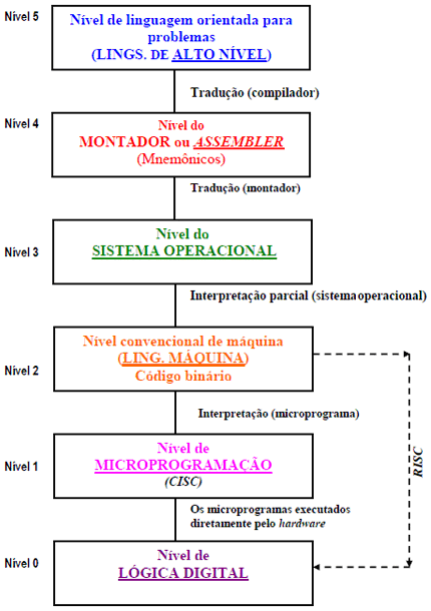
Níveis Conceituais

A maioria dos computadores modernos possui dois ou mais níveis, com o objetivo de facilitar a "conversa" entre o usuário e o computador.

No computador, cada linguagem usa sua antecessora como base. Sendo que na máquina multinível há hierarquias: a mais baixa é a mais simples e a mais alta a mais complexa.

Máquinas Multiníveis



Níveis Conceituais de uma Máquina Real

Segundo Tannenbaum, encontra-se nos níveis mais baixos a lógica digital, circuitos elétricos trabalhando com bits (0 e 1). Uma pessoa constrói um programa em **0** e **1**? Muito gasto de tempo. Essa lógica digital é passada para microprogramação e assim sucessivamente até chegar ao nível de linguagem em que se tem para programar: C, C++, Java, Pascal, ...

Nível dos Dispositivos

- Situado abaixo do nível 0
- Microeletrônica
- Características físicas
- Malha de transistores
- Tecnologias de fabricação de circuitos integrados

Nível 0 ou Nível da Lógica Digital

- É composto pelo hardware da máquina
- Portas Lógicas são os objetos de interesse dos projetistas de computadores nesse nível
- As portas lógicas (basicamente portas AND, OR e NOT) são os elementos primários de circuitos lógicos mais complexos.
- Combinação de portas lógicas:
  - Funções aritméticas;
  - Memórias (registradores);
  - Processadores.

Nível 1 ou Nível da Microprogramação

- Neste nível, inicia-se o conceito de programa como uma seqüência de instruções a serem executadas diretamente pelos circuitos eletrônicos.
- Poucas são as máquinas que têm mais de 20 instruções no nível do microprograma e a maior parte destas instruções envolve a movimentação de dados de uma parte da máquina para outros ou alguns testes simples.
- Utilizada especialmente (geralmente) em máquinas CISC - Complex Instruction Set Computer.

RISC x CISC

- **CISC - Complex Instruction Set Computer**
  - Arquitetura cujo processador é capaz de executar centenas de instruções complexas diferentes, sendo assim extremamente versátil.
  - Exemplos: 386, 486, Pentium e posteriores da Intel.
  - Muitas das instruções guardadas no próprio processador.
- **RISC - Reduced Instruction Set Computer**
  - Uma linha de arquitetura de computadores que favorece um conjunto simples e pequeno de instruções.
  - Exemplos: SPARC, MIPS, PowerPC, DEC Alpha, etc.
  - Considerado mais eficiente que as CISC
  - As instruções tendem a ser executadas em poucos (ou mesmo um único) ciclos de relógio.
- Tamanho do Código *vs* Desempenho
  - Geralmente, o desempenho de um RISC é melhor do que de um CISC;
  - Código gerado por um RISC tende a ser mais longo e complexo.
- Enxerga-se:
  - Um conjunto de 8 a 32 registradores
  - Um circuito chamado **ULA - Unidade Lógica e Aritmética**
- Os registradores e a ULA são conectados para formar o **Caminho de Dados (Data Path)**, estrutura sobre a qual os dados fluem.
- A operação básica do caminho de dados consiste na seleção de um ou de dois registradores para que a ULA opere sobre eles.
- Interpretador de um Microprograma
  - Busca, decodifica e executa as instruções, uma a uma, usando o caminho de dados para a realização de uma tarefa.
  - Pode ser controlado por hardware ou por software
  - Exemplo: Execução de uma instrução de SOMA (ADD)
    - A instrução deve ser executada na memória, seus operandos devem ser localizados e trazidos para os registradores, a soma deve ser calculada na ULA, e o resultado deve ser encaminhado para o lugar apropriado.

Nível 2 ou Nível ISA

- Nível ISA – Instruction Set Architecture: Arquitetura do Conjunto de Instruções;
- Nível Convencional de Máquina;
- Conjunto das Instruções Executáveis por uma máquina – processador;
- Cada máquina ou processador tem sua linguagem própria de Nível 2, a qual é documentada em manuais específicos de cada fabricante.
- Discute-se:
  - Tipo de Dados
  - Modelos de Memória e de Endereçamento
  - Formato e Tipos de Instruções
  - Fluxo de Controle

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Nível 3 ou Nível do Sistema Operacional (SOp)

- Esse nível suporta um conjunto de novas instruções, uma organização diferente da memória, a capacidade de rodar dois ou mais programas de forma simultânea, e outros.
- Fornece serviços básicos para os níveis acima. Tais serviços são: interface (gráfica ou linha de comando) com o usuário (Shell), gerenciamento de memória, escalonamento de processos, acionamento de dispositivos de entrada e saída de dados etc.
- Geralmente desenvolvido de forma híbrida, ou seja, parte em uma linguagem de alto nível, e parte diretamente em linguagem de máquina.
- **Níveis abaixo**: programadores de sistema, que são especialistas em projetar e implementar novas máquinas virtuais. Predominância de interpretação e linguagens freqüentemente numéricas, bom para as máquinas, mas ruim para as pessoas.
- **Níveis acima**: dirigidos aos programadores de aplicação com problemas a serem solucionados. Predominância de tradução, e as linguagens contêm palavras e abreviaturas significativas para as pessoas.

Nível 4 ou Nível de Linguagem de Montagem

- Uma forma simbólica de representação das linguagens dos níveis mais baixos.
- Provê um método para as pessoas escreverem programas para os níveis 1, 2, e 3 de uma maneira não tão desconfortável.
- Mnemônicos para as instruções de máquina.
- **Montador:** programa que executa a tradução ou interpretação dos programas em linguagem de montagem para uma linguagem do nível 1, 2 ou 3.

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

Nível 5 ou Nível de Linguagens de Alto Nível

- Linguagens projetadas para serem utilizadas por programadores de aplicação com problemas a serem resolvidos.
- Os programas escritos nessas linguagens são geralmente traduzidos para o nível 3 ou nível 4 por tradutores conhecidos como compiladores, embora às vezes sejam interpretados.

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Mais Níveis

- Os níveis 6 e superiores consistem em coleções de programas projetados para criar máquinas especialmente adequadas para certas aplicações, contendo grandes quantidades de informação acerca da aplicação.
- Máquinas virtuais voltadas a aplicações: Administração, Educação, Projeto de Computadores, Realidade Virtual, e outras aplicações.
- Dependendo do projeto da arquitetura, os níveis podem variar.

Compilador

A Teoria dos Compiladores tem como objetivo estudar e definir os parâmetros que permitem a construção de linguagens de programação e sua execução. Como um computador só compreende e executa instruções baseados em linguagem binária, mesmo os programas mais simples exigiriam um nível de abstração muito grande do programador, além de desprender muito tempo.

Um compilador (ou um conjunto de programas) que traduz um código fonte para uma linguagem de mais baixo nível (a linguagem alvo, que tem uma forma binária conhecida como código objeto). Normalmente, o código fonte é escrito em uma linguagem de programação de alto nível, com grande capacidade de abstração, e o código objeto é escrito em uma linguagem de baixo nível, como uma sequência de instruções a ser executada pela máquina X ou processador.

A compilação é um processo complexo muitas vezes consume mais tempo do que a execução do programa. Em qualquer fase da análise o compilador pode exibir mensagens dos erros detectados no programa fonte, gerando o cancelamento da compilação para que o usuário faça as correções.

Existem compiladores que permitem ao usuário ignorar ou reduzir a fase de otimização, diminuindo assim o tempo total de compilação.

- São programas que recebem como entrada arquivos texto contendo módulos escritos em linguagem de alto nível e geram como saída arquivos objeto para cada módulo.
- Se todas as bibliotecas ou módulos são apresentados como entrada, geram um programa executável diretamente.

O compilador e o interpretador é um dos tipos mais gerais de tradutores.

O processo de Compilação

O processo de compilação é composto de análise e síntese:

A análise tem por objetivo entender o código fonte, verifica erros, falhas e inconsistências, e representa em uma estrutura intermediária. Subdivide-se em análise: léxica, sintática e semântica.

A síntese constrói o código objeto a partir desta representação intermediária, e é mais variada, pode ser composta por etapas de: Geração de Código Intermediário, Otimização de Código e Geração de Código para Máquina X (final ou código de máquina), sendo somente esta última etapa obrigatória.

Pré-Processador

Muitos compiladores incluem um pré-processador. Um pré-processador normalmente é responsável por mudanças no código fonte destinada de acordo com decisões tomadas em tempo de compilação. Por exemplo, um programa em Linguagem C permite instruções condicionais para o pré-processador que podem incluir ou não parte do código caso uma assertiva lógica seja verdadeira ou falsa, ou simplesmente um termo esteja definido ou não.

Pré-processadores são mais simples que compiladores e são vistos pelos desenvolvedores como programas à parte, apesar dessa visão não ser necessariamente compartilhada pelo usuário.

Linker

Outra parte separada do compilador, cuja função é unir vários programas já compilados de uma forma independente e unificá-los em um programa executável.

Isso inclui colocar o programa final em um formato compatível com as necessidades do sistema operacional para carregá-lo em memória e colocá-lo em execução.

- São programas especiais que recebem como entrada arquivos objetos e geram como saída o programa final em linguagem de máquina.
- Gera um **programa executável** a partir de um ou mais arquivos objeto.

Montadores – Assemblers

- Montam um programa em linguagem de máquina a partir de sua versão em linguagem de montagem.
- Geram um **arquivo objeto.** Em geral, não pode ser executado diretamente pela máquina, por conter referências a sub-rotinas e dados especificados em outros arquivos.

Loaders

- Para executar um programa, um loader deve ser utilizado.
- O carregador é, em geral, parte do sistema operacional.

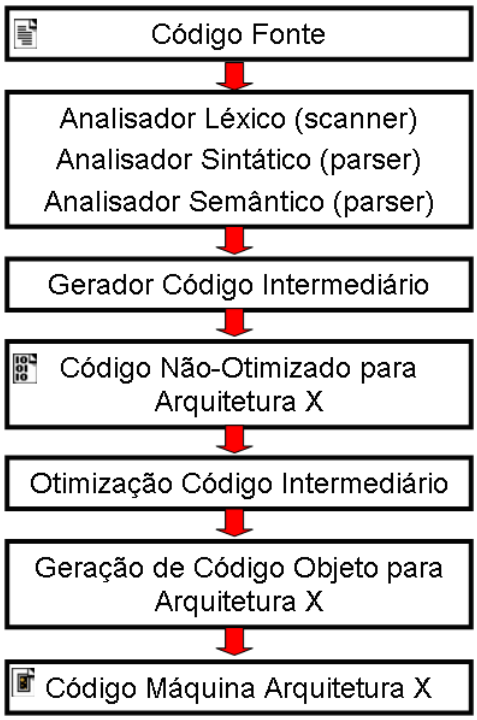
Componentes de um Compilador

A construção de um compilador é dividida em partes, cada uma com uma função especifica:

- **Analisador Léxico**: separa no programa fonte cada símbolo que tenha algum significado para a linguagem ou avisa quando um símbolo que não faz parte da linguagem é encontrado.

- **Análise Sintática**: é o responsável por verificar se a sequência de símbolos existentes no programa fonte forma um programa válido ou não. É construído sobre uma gramática composta de uma série de regras que descrevem quais são as construções válidas da linguagem.

- **Análise Semântica**: irá verificar se os aspectos semânticos estão corretos, ou seja, se não existem incoerências quanto ao significado das construções utilizadas pelo programador. A Análise Semântica depende de uma tabela de símbolos onde são armazenadas as informações de variáveis declaradas, funções ou métodos, entre outros.



Estrutura em Bloco da Compilação de um Programa Fonte

- **Geração de Código Intermediário:** após a verificação que não existem erros de ordem léxicos, sintáticos ou semânticos, o compilador realiza a sua tarefa de criar o programa objeto que reflete, mediante instruções de baixo nível, os comandos do programa fonte, para permitir a portabilidade da linguagem a outros computadores.

- **Otimização de Código**: Na fase de otimização do código intermediário que foi gerado anteriormente pela análise do programa global melhorou. Um exemplo de otimização seria encontrado no código de iniciação num ciclo constante, visto que essa inicialização iria ocorrer muitas vezes, como em um ciclo que é repetido. O otimizador puxaria esta constante fora de *loop* de inicialização para realizar uma única vez. Na otimização do código são aplicadas diversas técnicas para otimizar algumas características do programa objeto, como o seu tamanho ou sua velocidade.

- **Geração de Código Proposto**: Nesta fase, o código objeto final é gerado. Em alguns casos, esse código é diretamente executável, em outros requer algumas etapas pré-implantadas (montagem, de ligação e de carga). Para uma determinada linguagem de alto nível, é comum em todo o processo de geração de análise de código intermediário. Y é a geração do código de objeto que é particularizada por cada tipo de microprocessador.

Tipos de Compiladores:

- **Single-Pass**: Compilação em uma única leitura do programa fonte
- **Multi-Pass**: Compilação através de várias leituras do programa fonte
- **Load-and-Pass**: Compilação e a execução do programa fonte
- **Debugging**: Compilação permitindo a depuração do programa fonte
- **Optimizing**: Compilação e a otimização do programa alvo

Vantagens

- O código compilado é mais rápido de ser acessado
- Impossibilita ou pelo menos dificulta ser quebrado e visualizado o código-fonte original
- Permite otimização do código por parte do compilador
- Compila o código somente se estiver sem erro algum

Desvantagens

- Para ser utilizado o código precisa passar por muitos níveis de compilação
- Impossibilidade de não se ver o código-fonte pode dificultar observar falhas
- Processo de correção ou alteração do código requer que ele seja novamente recompilado

Interpretadores

Interpretadores são programas que lêem o código fonte de uma linguagem de programação convertendo em código executável. O funcionamento varia de acordo com a implementação.

Em muitos casos o interpretador lê linha-a-linha e converte em código objeto à medida que vai executando o programa. Linguagens interpretadas são mais dinâmicas por não precisarem **escrever-compilar-testar-corrigir-compilar-testar-distribuir**, e sim **escrever-testar-corrigir-escrever-testar-distribuir**. Mas existem também linguagens que funcionam tanto como interpretadores quanto como compiladores, como, por exemplo, Python (quando requerido).

A princípio, podem-se implementar compiladores e interpretadores para qualquer linguagem de programação. Mas para determinadas linguagens é mais fácil "fabricar" interpretadores, e para outras é mais prático um compilador.

- Recebem como entrada arquivos texto contendo programas em linguagem Assembly ou de alto nível, ou arquivos binários com instruções de máquina, e os executam diretamente.
- Interpretadores percorrem os programas, a partir de seu ponto de entrada, executando cada comando.
- Processadores são interpretadores implementados em hardware.

Vantagens:

- Correções e alterações são mais rápidas de serem realizadas
- Código não precisa ser compilado para ser executado
- Consomem menos memória

Desvantagens

- Execução do programa é mais lenta
- Necessita sempre ser lido o código original para ser executado

Comparativo: Compilador vs Interpretador

	Vantagem	**Desvantagem**
Compiladores	Execução mais rápida	Várias etapas de tradução
	Permite estruturas de programação mais complexas para a sua execução	Programação final é maior mais memória
	Permite a otimização do código fonte	Processo de correção de erros e depuração é mais demorado
Interpretadores	Depuração do programa é mais simples	Execução do programa é mais lenta
	Consome menos memória	Estruturas de dados demasiado simples
	Resultado imediato do programa ou rotina desenvolvida	Necessário fornecer o programa fonte ao utilizador

Bytecode

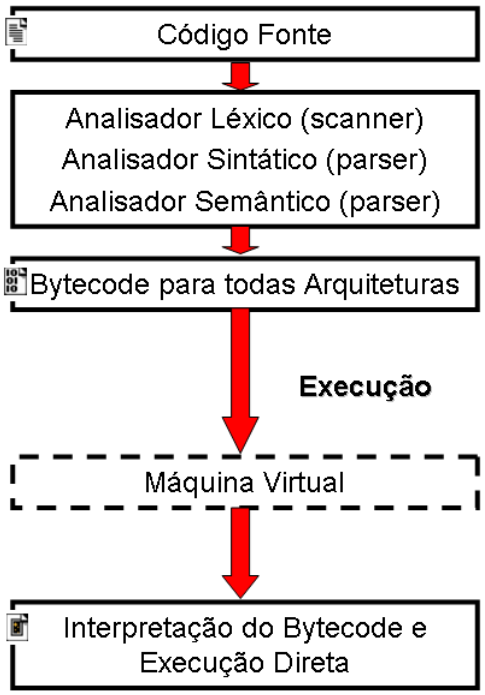
Um compilador traduz um programa de uma linguagem textual, acessível a um ser humano, para uma linguagem de máquina, específica para um processador e um sistema operacional.

Em linguagens de programação híbridas, o compilador tem o papel de converter o código fonte em um código chamado de Bytecode (código em bytes), que é um estágio intermediário entre o código-fonte (escrito numa linguagem de programação específica) e a aplicação final. Para executar o bytecode utiliza-se uma máquina virtual, que serve para interpretar e executar o bytecode.

As principais vantagens do bytecode são:

- **Portabilidade** (Independência de Sistema Operacional) — pois o bytecode roda na máquina virtual, e não no Sistema Operacional do equipamento.
- **Não-necessidade do pré-processamento**, típico dos compiladores.

O bytecode é encarado como um produto final, cuja validação da sintaxe e tipos de dados (entre outras funções dos compiladores) não será necessária. Como exemplos de linguagens que utilizam bytecode têm-se: Java, C# e Lua.



Estrutura em Bloco da Interpretação de um Programa Fonte