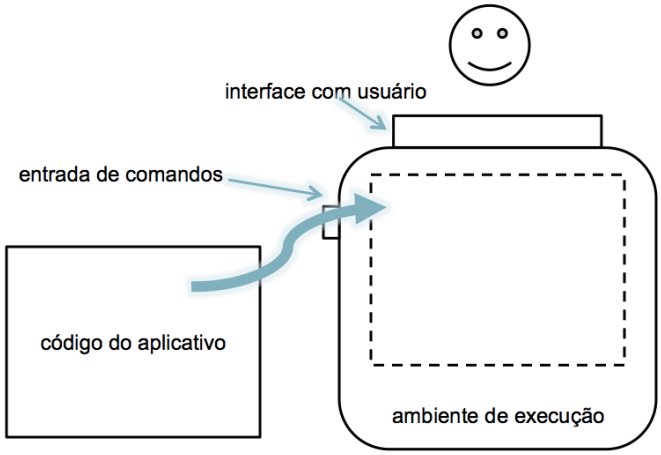
1. Aplicativos de software funcionam em um **ambiente de execução (*runtime environment*).**

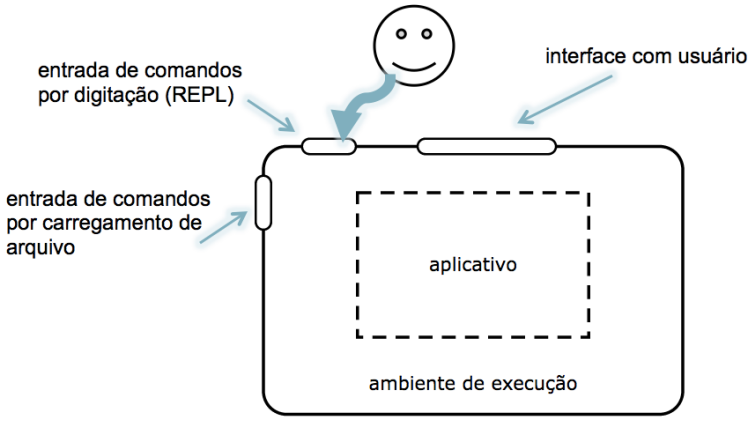


O ambiente de execução proporciona ao aplicativo acesso programático à interface com o usuário. Por exemplo, imprimir texto na tela.

Aplicativos realizam ações relativas à sua finalidade. Em um editor de texto, é possível digitar e formatar texto, num navegador de internet, é possível acessar sites, num editor gráfico, é possível criar figuras.

O ambiente de execução possui um mecanismo para carregar o aplicativo. O aplicativo consiste em instruções (programa) para o ambiente de execução que realizam a sua funcionalidade característica.

Alguns ambientes de execução permitem a digitação direta de comandos, como, por exemplo, através de um REPL (Read-Eval-Print Loop). A figura abaixo esquematiza esse caso.

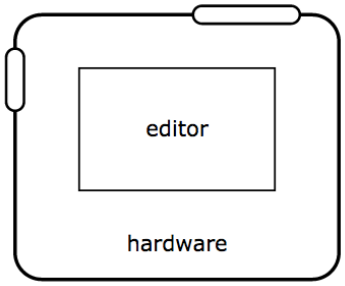


2. O ambiente de execução fundamental: **o hardware**

Para o hardware executar um aplicativo, este deve estar codificado em linguagem de máquina. Cada instrução dessa linguagem corresponde a um código binário. Abaixo, vê-se um trecho do arquivo do programa `ls` do Mac OS X com arquitetura Intel, na qual uma instrução pode abranger vários bytes.

```
0001d10 9a ff ff ff 9a ff ff ff 3a ff ff ff 55 48 89 e5
0001d20 48 8b 3f 0f b7 4f 58 31 c0 83 f9 07 74 44 48 8b
0001d30 36 0f b7 56 58 83 fa 07 74 38 0f b7 c9 83 f9 0a
0001d40 74 32 0f b7 d2 83 fa 0a 74 2a 39 d1 74 2c 66 83
0001d50 7f 56 00 75 25 f6 05 b6 37 00 00 01 75 1c b8 01
0001d60 00 00 00 83 f9 01 74 0a b8 ff ff ff ff 83 fa 01
0001d70 75 08 5d c3 5d e9 fe ef ff ff 48 8b 05 97 37 00
```

Na linguagem de máquina não há variáveis; dados na memória são acessados por meio de seus endereços. Há também os registradores, um tipo especial de memória, geralmente partes internas do processador. Cada operação possível tem um código próprio.



O sistema operacional é fundamental para esse tipo de programa. Ele funciona como uma biblioteca de funções. O acesso programático à interface com usuário, por exemplo, é proporcionado pelo sistema operacional.

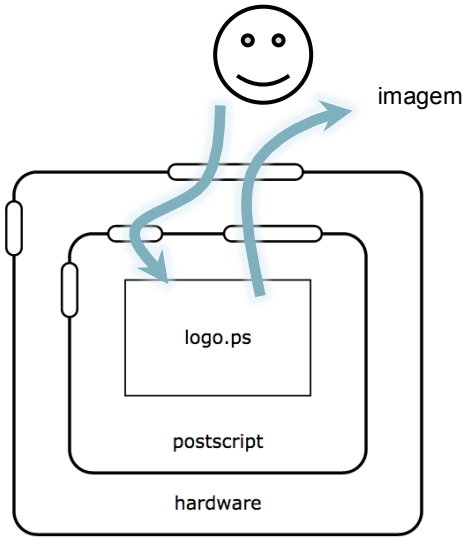
Não é prático programar em linguagem de máquina. Em geral, aplicativos em linguagem de máquina são gerados por ferramentas durante o processo de compilação. O mais próximo que se costuma chegar, em termos de programação, à linguagem de máquina são as linguagens de montagem.

Não há como enviar instruções diretamente ao hardware. O aplicativo deve ser carregado, pelo sistema operacional, para a memória, e o processador deve ser direcionado para começar a executar as instruções a partir dali.

3. Aplicativo e ambiente de execução ao mesmo tempo: **o interpretador**

Um interpretador é, em primeiro lugar um aplicativo. Isso quer dizer que ele funciona em um ambiente de execução, que tanto pode ser o hardware como outro interpretador.

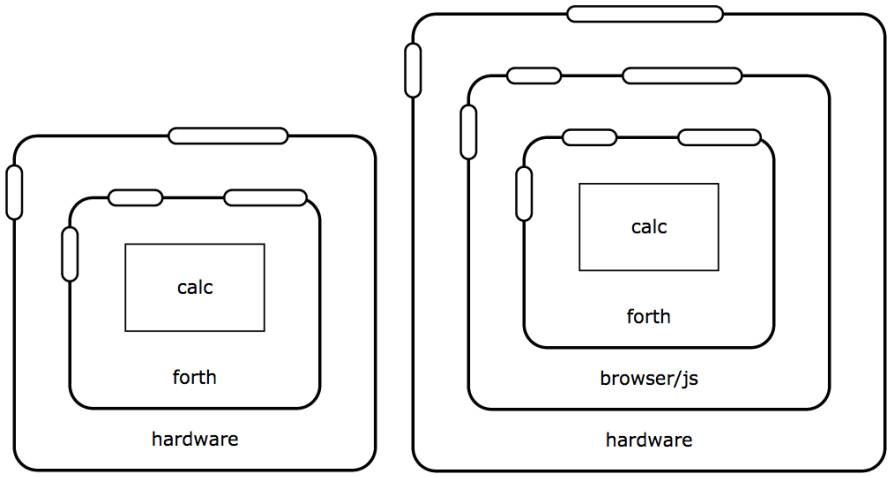
Assim como qualquer aplicativo, o interpretador executa ações coerentes com sua finalidade. O que distingue um interpretador de um aplicativo em geral é que o interpretador não recebe comandos através de ações na interface com usuário, mas através de um programa escrito em uma linguagem de programação.



A ilustração acima mostra a utilização de um interpretador da linguagem PostScript. O interpretador é carregado no hardware e passa a funcionar. Em seguida, o usuário começa a criar um logotipo. As instruções são digitadas uma a uma (não chegam a formar um arquivo nesse caso) e o interpretador exibe a imagem concomitantemente. O intepretador, portanto, funciona no modo REPL. Do ponto de vista do hardware, como ambiente de execução, a digitação do usuário é uma ação de interface. Do ponto de vista do interpretador PostScript, a digitação é entrada de comandos.

Uma grande vantagem dos interpretadores em relação ao hardware como ambientes de execução é a portabilidade. Desde que haja implementações do interpretador para as plataformas visadas, o mesmo

programa pode ser usado sem modificações em todas elas. Um programa em linguagem de máquina só funciona no hardware para a qual ele foi feito. As ilustrações abaixo mostram um aplicativo de calculadora feito na linguagem Forth. Na primeira, ele é executado por um interpretador Forth disponível na plataforma do hardware. Na segunda, ele é executado por um intepretador Forth feito em JavaScript, a linguagem de programação interpretada disponível em qualquer navegador web. O programa de calculadora é o mesmo. O interpretador Forth *esconde* tudo o que está sob ele (o hardware, o intepretador JavaScript, etc.).



O desempenho de um interpretador típico é inferior em relação ao desempenho do hardware. A razão disso é que uma instrução em uma linguagem de programação interpretada é uma string de texto de tamanho significativo, que ainda precisa passar por uma análise de formato antes da execução, enquanto uma instrução em linguagem de máquina é um código binário de pouquíssimos bytes.

Atualmente, muitos interpretadores são implementados com uma estratégia diferente, chamada *Just In Time Compiler* (JIT). O interpretador recebe o programa, e, internamente, compila-o para a linguagem de máquina (não se considera o caso de interpretadores que funcionam em outros interpretadores). A compilação JIT deve ocorrer durante o carregamento do programa. Como compilação é uma tarefa demorada, a compilação JIT pode retardar significativamente o início da execução do programa. Por isso, muitos compiladores JIT adotam a estratégia de não compilar o programa todo, mas apenas as partes mais usadas (encontradas pelo rastreamento da execução do programa e posterior análise estatística). Programas executados em interpretadores baseados em compilação JIT podem apresentar melhor desempenho do que programas em linguagem de máquina gerados por compiladores típicos, pois os interpretadores têm informações da execução do programa que possibilitam realizar otimizações (compiladores típicos não contam previamente com tais informações).

Qual é a imagem mental que se pode fazer a respeito do processo de interpretação? O interpretador funcionando diretamente no hardware converte as instruções da linguagem em código de máquina? Esse é o caso dos intepretadores baseados em compilação JIT. No caso de interpretadores típicos, o interpretador é um aplicativo feito em linguagem de máquina que muda o seu fluxo de execução de acordo com o programa escrito na linguagem intepretada. Há uma diferença conceitual.

Máquinas virtuais são interpretadores cujas linguagens de programação que executam imitam linguagens de máquina, isto é, são códigos binários altamente compactos que comandam uma espécie de máquina com registradores, pilha, etc. Máquinas virtuais são chamadas também de máquinas abstratas. A tabela abaixo resume as semelhanças e as diferenças dos ambientes de execução.

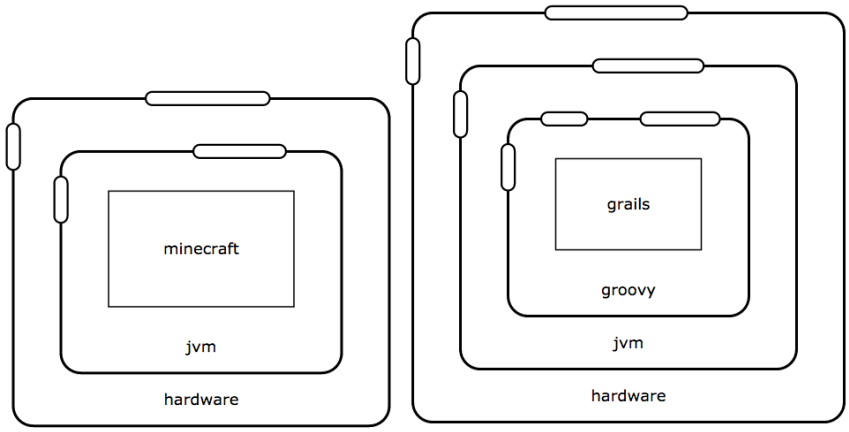
	Hardware	Interpretadores em geral	Máquinas virtuais
Tipo de implementação	Hardware	Software	Software
Tipo da linguagem executada	Binária	Textual	Binária

Programas em linguagens binárias geralmente são gerados por outros programas, como compiladores. Programas em linguagens textuais podem ser gerados tanto por programadores quanto outros programas. Javascript é uma linguagem textual que tem sido usada como alvo de compilação (muitas vezes, chamados de

transpilers). Já se discute a criação da linguagem Web Assembly como alvo mais conveniente para compilação.

A máquina virtual *P-code* fez história nos anos 1990 como mecanismo de *bootstrapping* da linguagem Pascal. A linguagem baseada em máquinas virtuais mais influente é Java. A Microsoft adotou o esquema de máquinas virtuais no seu conjunto de linguagens .NET. A máquina virtual java é conhecida como JVM (*Java Virtual Machine*) e a máquina virtual .NET é conhecida como CLI (*Common Language Infrastructure*).

As ilustrações abaixo trazem ambientes de execução envolvendo máquinas virtuais Java. Groovy é uma linguagem script que funciona na JVM.



Segurança é uma vantagem de linguagens executadas por máquinas virtuais. Quando o programa objeto é carregado na máquina virtual, diversas verificações são realizadas no código.

4. Roteiros de atividades sobre ambientes de execução

(4.1) Comparação entre aplicativos em geral e interpretadores. Instale os programas Inkscape e ghostscript. O primeiro é um aplicativo de edição gráfica e o segundo é um interpretador da linguagem gráfica PostScript. No programa Inkscape, desenhe um círculo e uma reta. Execute o interpretador ghostscript (em ambientes Unix, digite `gs` na linha de comando). O interpretador começa no modo REPL. Digite a sequência abaixo e observe o efeito. Como o Inkscape o o ghostscript se diferenciam.

```
newpath <enter>
100 <enter>
100 <enter>
moveto <enter>
200 <enter>
200 <enter>
lineto <enter>
stroke <enter>
newpath <enter>
300 <enter>
300 <enter>
150 <enter>
0 <enter>
360 <enter>
arc <enter>
stroke <enter>
showpage <enter>
```

(4.2) Intepretadores estão por todo lugar. Instale o Google Chrome. Procure o Console JavaScript. Digite comandos como:

```
2 + 2 <enter>
alert(“HELLO!!!”) <enter>
```

Há interpretadores para as mais diversas funcionalidades. Confira a tabela a seguir:

Tipo de aplicação	Exemplo de aplicativo	Exemplo de interpretador
Editor de texto	MS Word	TeX ou LaTeX
Editor gráfico	Inkscape	ghostscript
Programação		Linguagens script*, máquinas virtuais, Lisp, etc.

* Linguagens script: shell, perl, python, ruby, lua, javascript groovy, etc.

(4.3) Diferença entre o REPL e a interface com usuário como entrada e saída de dados. Os dois acessos que o usuário tem ao sistema (o REPL e a interface com usuário) costumam estar ligados a um mesmo teclado e a uma mesma tela. É preciso distinguir logicamente quando o usuário está enviando comandos para o ambiente de execução (via REPL) e quando o usuário está respondendo ao aplicativo (que são os próprios comandos digitados no REPL) através da interface com usuário. Instale um interpretador Python. Invoque o interpretador digitando `python` na linha de comando. Digite:

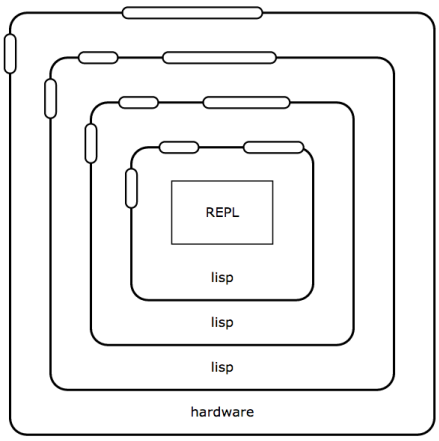
```
x = input()<enter>
34<enter>
x<enter>
print x<enter>
```

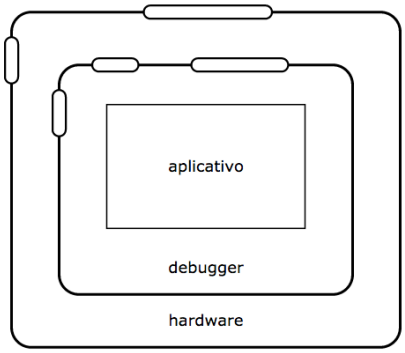
Copie as instruções acima e insira os resultados mostrados na tela. O resultado esperado tem 6 linhas. Classifique cada linha como relativa ao REPL ou relativa à interface com usuário.

O Console JavaScript do Google Chrome diferencia impressões de saída e resultados do REPL. Abra o Console JavaScript e digite as instruções a seguir, observando a diferenciação:

```
2<enter>
var x = 3<enter>
x<enter>
console.log(4)<enter>
{console.log(5); var y = 6; console.log(y); y + 1}<enter>
{console.log(8); console.log(9)}<enter>
```

(4.4) [opcional] Interpretadores feitos na mesma linguagem que processam. Livros sobre a linguagem Lisp costumam apresentar, em algum momento, um interpretador Lisp escrito em Lisp. É natural dadas as características da linguagem, e bem conciso. Charles Queinnec em *Lisp in small pieces* leva essa ideia ao extremo, propondo como exercício que o leitor carregue o interpretador várias vezes em si mesmo e meça o desempenho. Outro caso notável são os debuggers de linguagem de máquina. Eles, de certa forma, emulam o hardware, portanto podem ser vistos como interpretadores que executam linguagem de máquina. (Lembrando que, obviamente, eles também são implementados em linguagem de máquina.)





Procure um interpretador Lisp (sugestão: Dr. Racket) e o código de um intepretador Lisp escrito em Lisp e tente executar a proposta de Queinnec. Experimente um debugger de linguagem de máquina (sugestão: gdb no Unix).

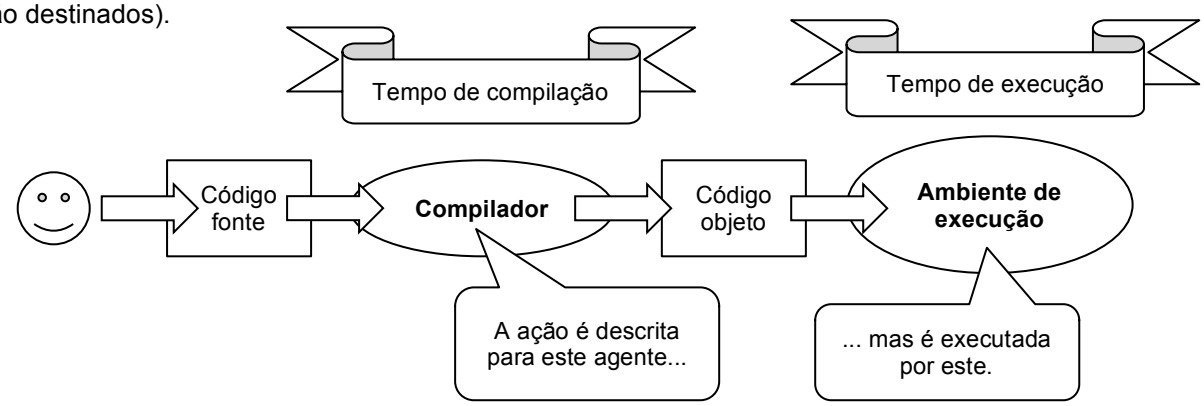
5. Questões abertas sobre ambientes de execução

(5.1) Vantagens das máquinas virtuais. A portabilidade é a vantagem evidente das máquinas virtuais, assim como qualquer linguagem interpretada. Mas não se fosse só isso, a Microsoft não adotaria o modelo de máquinas virtuais como fez no .Net, pois seus produtos funcionam predominantemente no mesmo hardware (Intel) e no mesmo sistema operacional (Windows). Pesquise sobre o ataque de buffer overrun em C e por que ele é impossível de acontecer em Java (exceção ArrayIndexOutOfBounds).

(5.2) Levando em consideração apenas a definição de uma linguagem, isto é, seus elementos e conceitos básicos, é possível determinar se ela é interpretada ou compilada?

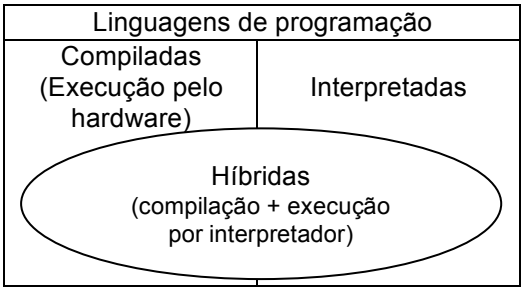
6. Transformando linguagens: **o compilador**

Um compilador não é um ambiente de execução. Ele *não* realiza as ações descritas no programa que processa, ele as traduz em instruções de uma outra linguagem. Portanto, não faz sentido falar em desempenho e portabilidade de um compilador, mas sim dos programas que ele gera (nos ambientes de execução a que são destinados).

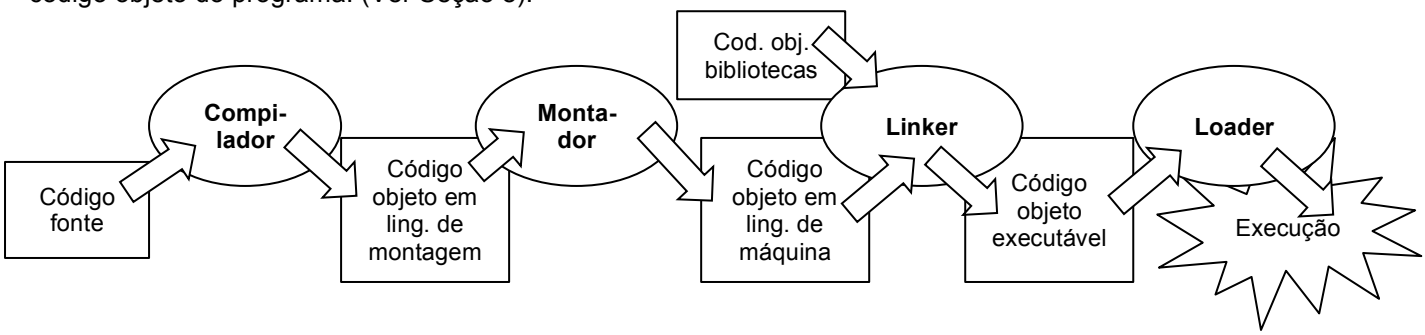


Os termos código fonte e código objeto, rigorosamente falando, só fazem sentido num contexto que envolve compilação. Importante também é a distinção que se cria entre tempo de compilação e tempo de execução. O programador descreve para o compilador a ação a ser executada pelo ambiente de execução em outro momento.

Geralmente, o ambiente de execução alvo dos compiladores é o hardware. Assim, há uma divisão natural das linguagens de programação: aquelas que são programadas em compiladores e executadas em hardware e aquelas que são executadas e programadas textualmente em intepretadores. Mais recentemente, com o advento das máquinas virtuais e demais interpretadores como ambiente de execução alvo de compiladores, essa dualidade foi quebrada. Surgiu, então, o termo linguagens de programação híbridas para designar o novo esquema: linguagens programadas em compiladores e executadas em interpretadores.



O processo de compilação propriamente dito geralmente tem como resultado um código objeto em linguagem de montagem. Portanto, ele é complementado pelo montador (*assembler*)[1] para se obter o código objeto em linguagem de máquina. Por fim, o linker compõe o código objeto das bibliotecas usadas no programa com o código objeto do programa. (Ver Seção 8).



O esquema acima é conhecido como compilação separada. O contrário dele seria a compilação com código fonte das bibliotecas.

Como já foi dito, sempre há um mecanismo de carregamento em um ambiente de execução. No caso do hardware, essa incumbência é do sistema operacional. A parte do sistema operacional que é responsável pela carga de programas é um programa chamado loader. Ele faz uma espécie de relocação de código assim como o linker, pois os endereços finais do código objeto na memória não são os mesmos que ele trazia no arquivo. Em sistemas operacionais da família Unix, geralmente o linker e o loader são o mesmo programa (chamado `ld`).

Uma vantagem do uso dos compiladores é a antecipação de erros. Como o compilador processa todo o programa para fazer a tradução, ele encontra alguns tipos de erros como nomes de variáveis e funções escritos indevidamente, tipos incompatíveis (no caso de linguagens com tipos), etc.

Compiladores podem ser vistos em certas condições como interpretadores, pois o processo de compilação pode ser programado em certa medida. Ou seja, o compilador continua não executando as ações essenciais que estão definidas no programa, mas ele pode executar ações que controlam a compilação. Por exemplo, quem avalia a expressão aritmética na instrução abaixo: o ambiente de execução ou o compilador?

```
i = 38 * 23 - 45;
```

Obviamente o compilador! Ele tem todas as informações em tempo de compilação para fazer a conta. Ele possui um avaliador aritmético. É como se o programador dissesse: “Compilador, faça a conta 38 * 23 – 45 para mim, e traduza para o ambiente de execução que é para atribuir esse valor à variável”.

O compilador executa em tempo de compilação ações de controle da compilação, ele é um interpretador com respeito às ações de compilação. Se muitos programadores entendessem isso mais profundamente, eles poderiam tirar maior proveito dos recursos oferecidos pelo compilador. Um exemplo claro disso são as diretivas do pré-processador C. Por exemplo, o programa

```
#define MAX 100
int main() {
      int i;
      for(i=0; i<MAX; i++) {
            printf(“%d”,i);
      }
}
```

[1] O montador é um compilador. Não se costuma chamá-lo de compilador por duas razões: a) como ele é usado juntamente com um compilador de outra linguagem, evita-se confusão de termos; b) a transformação que ele faz, da linguagem de montagem para a linguagem de máquina, é comparativamente mais simples do que o que fazem os compiladores de linguagens de alto nível de abstração. As linguagens de montagem não possuem estruturas aninhadas como laços e condicionais. O montador praticamente faz uma tradução linha a linha.

pode ser entendido a partir como um diálogo em que o programador diz: “Compilador, vamos combinar entre nós o valor MAX como valendo 100. Sempre que eu falar MAX, você coloca 100, mas isso fica entre nós. Então, traduza aí para o ambiente de execução: crie uma variável inteira, execute um laço com ela, iniciando em 0, incrementando de um em um até chegar a MAX, imprimindo o valor da variável em cada iteração”. O que chega no programa compilado para o ambiente de execução é o valor 100, não existe vestígio de MAX no programa compilado.

É possível, inclusive, fazer compilação condicional com diretivas do pré-compilador. O pré-processaodor C possui uma série de diretivas para esse fim, como `#if`, `#elif`, `#endif`, além de `#ifdef`, `#ifndef`, `#undef`. Qualquer programa em C de médio porte está repleto de diretivas como essa, no mínimo como forma de evitar múltiplos *includes*.

O mecanismo mais versátil de controle sobre a compilação é a macro. Há linguagens de programação em que macros são amplamente usadas como forma de alteração da própria linguagem, como é o caso do Lisp (na verdade, uma família de linguagens de programação). O pré-processador C possui a noção de macros, embora de forma limitada em comparação com Lisp.

7. Roteiros de atividades sobre compiladores

(7.1) Etapas da compilação. Digite o seguinte programa no arquivo `clp_lab.c` :

```
int main() {
      int i = 15;
}
```

Execute o compilador com a chave `-S`. Isso fará o compilador gerar apenas o código objeto em linguagem de montagem. Examine o arquivo gerado em um editor de texto.

```
gcc -S clp_lab.c
```

Execute o compilador com a chave `-c`. Isso fará o compilador invocar o montador para obter o código objeto em código de máquina. Examine o arquivo gerado com um visualizador de dados binários.

```
gcc -c clp_lab.c
hexdump -C clp_lab.o
```

(7.2) Experimentando com o pré-processador. Digite o seguinte programa no arquivo `clp_pre.c` :

```
#define TEST
int main () {
      int i;
#ifdef TEST
      i = 10;
#elif
      i = 100;
#endif
#ifdef PRO
      int j = 5;
#endif
}
```

Execute o pré-compilador apenas, com a chave `-E`. O resultado é mostrado na tela. Execute também definindo a variável `PRO` na linha de comando, com a chave `-DPRO`. Para definir uma variável `abc` na linha de comando, basta usar a chave `-Dabc`. Para atribuir o valor 123 a ela, use `-Dabc=123`.

```
gcc -E clp_pre.c
gcc –E -DPRO clp_pre.c
```

Estude sobre macros e diretivas do pré-processador C, inclusive os operadores de continuidade, #, ##, `defined()`, `#line`, `#error`, `#pragma`, `__DATE__`, `__TIME__`, `__LINE__`, `__FILE__`, `assert()`. Confira o livro de Luís Damas, **Linguagem C**, 10.ed, LTC, 2013, Capítulo 13.

(7.3) Falta de verificação de erros em interpretadores. Abra o Google Chrome e digite o seguinte programa em JavaScript no Console:

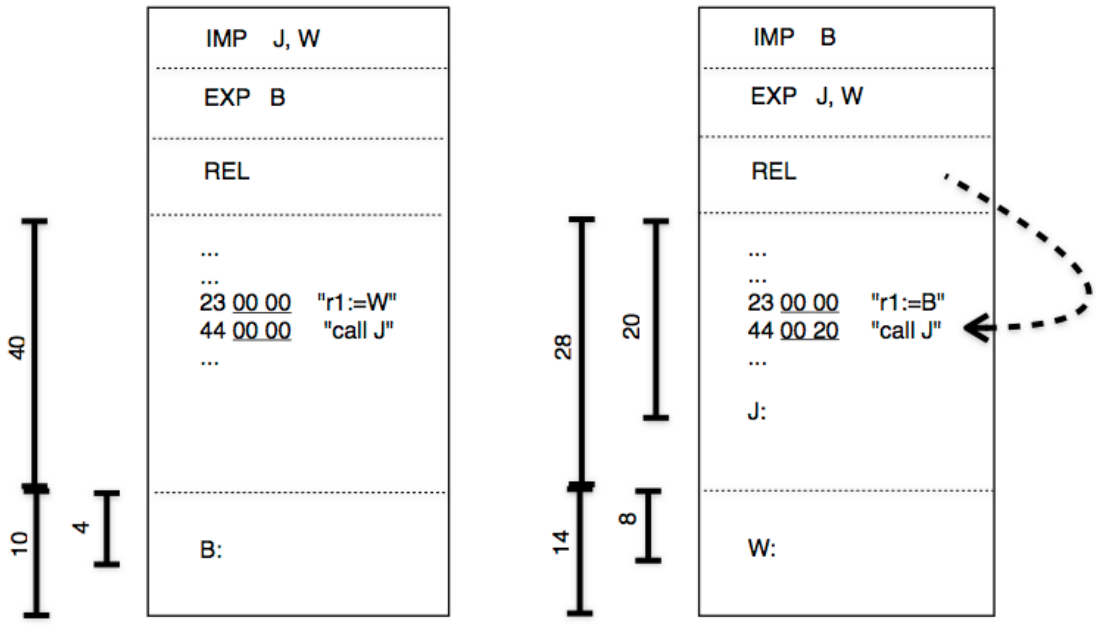
```
if (true) {
        alert("Deu certo")
} else {
        alrt ("<- Erro de propósito")
};
```

Repita trocando `true` por `false`. Observe que o erro de digitação só é percebido quando a instrução é executada. Uma instrução que é muito pouco usada no programa tem chances de só mostrar seus erros (alguns tipos de erro mais evidentes como erros de sintaxe) nas mãos do usuário. Isso não acontece em linguagens compiladas ou híbridas.

8. O linker

Há quem traduza a palavra linker como link-editor. Assim, a etapa realizada pelo linker é a link-edição. Um linker é um programa especializado em fazer relocação de código em linguagem de máquina. Enquanto estão em arquivos separados, os programas em código objeto começam todos do mesmo endereço zero. Quando eles são combinados em um só arquivo, cada um ocupa uma posição, começando onde o outro termina. Apenas o primeiro pode manter seus endereços relativos a zero. Os demais precisam de relocação. Algumas instruções de linguagem de máquina contêm em si endereços. Essas instruções precisam ser modificadas para refletir os endereços do programa no novo arquivo de código objeto gerado pelo linker.

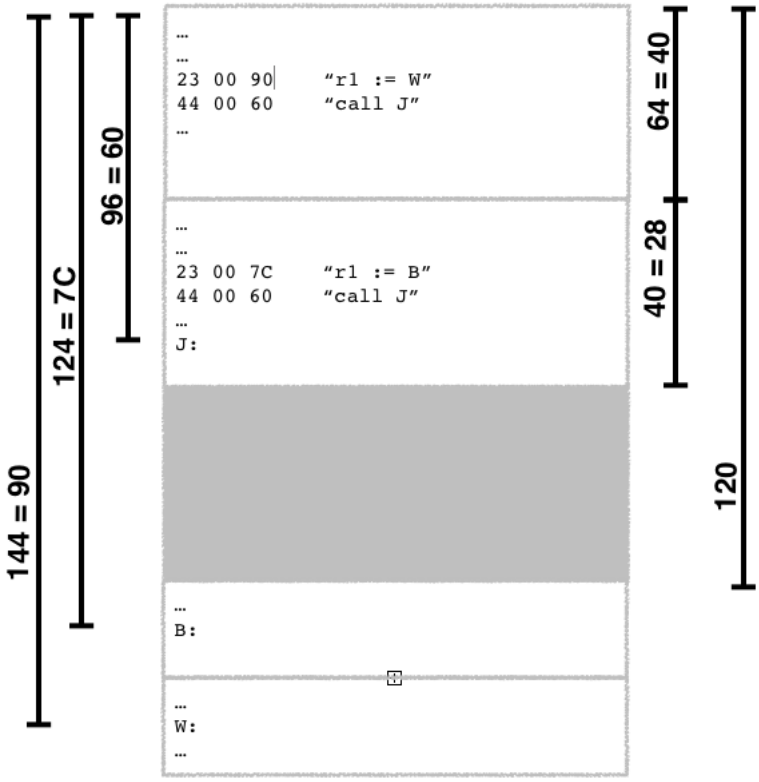
O diagrama abaixo ilustra dois arquivos no formato objeto. *Pede-se um diagrama do arquivo resultante da link-edição entre os dois.* Considera-se que as páginas nesse sistema medem 60 bytes.



A ilustração acima refere-se a dois arquivos no formato de código objeto. O começo de um arquivo objeto contém a tabela de símbolos. Na ilustração, ela está dividida em três partes na ordem: símbolos utilizados no arquivo que são definidos em outro programa objeto (símbolos importados, IMP), símbolos definidos no arquivo e à disposição de outros programas objeto (símbolos exportados EXP), e posições no arquivo que contêm endereços em e necessitam ser modificadas caso haja relocação (REL). Depois da tabela de símbolos com suas três divisões, fica o segmento de programa, com código em linguagem de máquina, e, por fim, o segmento de dados, com variáveis, arrays, estruturas de dados.

Assim, o diagrama da esquerda na ilustração mostra um arquivo objeto que importa os símbolos J e W, exporta B, (as relocações não são mostradas), possui um segmento de código que mede 64 bytes (40 em base 16), um segmento de dados que mede 16 bytes (10 hexa), e uma variável B que fica no endereço 4 do segmento de dados. Há outros dados e instruções, mas eles não são mostrados. A instrução 23 00 00 em linguagem de máquina é decodificada como o carregamento do valor de W no registrador r1. Isso é indicado em pseudocódigo como “r1 := W”. Essas expressões entre aspas não fazem parte do arquivo, são indicações da ilustração. 44 00 00 é a invocação de uma sub-rotina. O endereço dela corresponde aos bytes 00 00 (hexa) dentro da instrução. Essas posições deveriam ser preenchidas pelo endereço da sub-rotina J, mas como ela é externa, registra-se 00 00. A tabela de símbolos contém todos os endereços em que o endereço de J deveria aparecer (isto não está mostrado na ilustração), de forma que o linker saberá onde inserir os endereços de J quando combinar este programa objeto com algum programa que forneça o código de J. (Esse mecanismo é idêntico para W.)

O diagrama da direita mostra um arquivo objeto que importa o símbolo B, exporta J e W e possui uma relocação exibida na ilustração, que será explicada em breve. Além disso, possui um segmento de código de 40 bytes (28 hexa), no qual está contida uma sub-rotina J, que se inicia no endereço 32 (20 hexa), entre outros que não são mostrados. Quanto ao segmento de dados, ele possui 20 bytes (14 hexa) e a sua única variável mostrada no diagrama é W, na posição 8. A instrução 44 00 20 é a chamada a uma sub-rotina do próprio programa. Ela contém o endereço da sub-rotina (20 hexa), que precisará ser modificado caso haja relocação de código.



A ilustração mostra o arquivo objeto executável obtido da combinação dos arquivos objeto anteriormente descritos. Inicialmente o linker reúne os segmentos de código. No exemplo, o resultado combinado mede 104 bytes, e cabe, portanto, nas duas primeiras páginas. Lembrando que cada página mede 60 bytes. Em seguida, o linker prossegue com a parte dos segmentos de dados. Eles ficam na terceira página (a partir do endereço 120). Essa é a disposição final no arquivo. Agora, pode-se calcular os endereços finais das rotinas e variáveis. J era a posição 20 hexa do segmento que começa no endereço 40 hexa. Portanto, ficou no endereço 60 hexa. B fica na posição 4 do segmento de dados que começa em 120, portanto 124 (7C hexa). E W fica na posição 8 de um segmento que começa em 136 (120 + 16) [16 é o tamanho do segmento de dados que vem antes], portanto na posição 136 + 8 = 144 (90 hexa). Por fim, esses endereços são inseridos nas posições internas do código em linguagem de máquina.