Criação de Web Sites II

3

SSI
Cookies
Programação

Conteúdo

10. Server-side includes

SERVER-SIDE INCLUDES (SSI) são instruções para o servidor embutidas em uma página HTML. Normalmente, um servidor não analisa o conteúdo de uma página HTML, mas simplesmente a recupera e envia para o cliente após uma requisição. Se o suporte a SSI estiver instalado e a página for identificada como uma página que o servidor deve analisar (*server-parsed HTML*), ele irá tratar a página como se fosse um programa. Tudo o que for HTML puro será copiado para a saída padrão (como já era feito antes) mas se o servidor encontrar instruções especiais dirigidas a ele, da forma:

```
<!--#include file="texto_extra.txt" -->
```

ele irá tentar executá-las. Elas parecem comentários HTML, mas jamais devem chegar ao browser. O servidor deverá substitui-las pelo resultado da sua execução. As instruções sempre causam a inclusão de *texto* que o servidor copiará para a saída padrão. No final, o browser deverá receber uma página cujo conteúdo foi alterado dinamicamente, e cujo código difere daquele contido no arquivo HTML armazenado no disco do servidor.

Caso, por algum motivo, o servidor não abra o arquivo para interpretar os comandos SSI, eles chegarão ao browser mas não serão exibidos na página, pois estarão entre comentários HTML.

10.1. Como habilitar SSI no servidor

As páginas que podem sofrer transformações precisam ser identificadas pelo servidor. Normalmente, quando não está configurado o suporte a SSI, o servidor simplesmente copia o conteúdo de um arquivo solicitado na saída padrão, e não interpreta seu código.

Como o código só pode ser incluído em páginas HTML (e não em imagens, programas, etc.), deve-se restringir a procura pelos comandos SSI apenas a esses arquivos, identificados pela extensão do nome do arquivo. O servidor então tentará processar todos os comandos encontrados nessas páginas antes de enviar a nova página gerada dinamicamente ao browser. O desempenho do servidor, porém, deverá cair, pois o ele terá que analisar todas as páginas devolvidas à procura de comandos SSI, mesmo que elas não tenham tais comandos.

Uma solução para limitar a análise e interpretação apenas aos arquivos que realmente contêm instruções SSI seria identificá-los. Uma forma de identificação poderá ser através de localização, como fizemos para que o servidor identificasse programas CGI. Cria-se um diretório e determina-se que qualquer página localizada naquele diretório será analisada em busca de instruções SSI. Mas arquivos que contém SSI são geralmente documentos, que fazem parte de coleções de documentos. Separá-los das outras páginas seria inconveniente.

Uma outra solução, que é a mais usada, é identificar os arquivos que contém instruções SSI através de uma extensão de nome de arquivo diferente. Pode ser qualquer uma, mas convencionalmente utiliza-se a extensão .shtml.

O suporte a SSI varia entre servidores Web. Praticamente todos suportam as instruções <!--#include -->. Outros suportam instruções proprietárias e poderão suportar um pequeno grupo de instruções populares nos servidores Unix. A configuração de SSI em alguns servidores é desnecessária. Basta mudar a extensão do arquivo para .shtml que ele passará a abrir as páginas para ler os comandos. Em outros servidores é preciso ativar o servi-

ço, pois alguns comandos poderão oferecer riscos de segurança. Na seção seguinte mostraremos como configurar o suporte a SSI nos servidores mais populares.

10.2. SSI em servidores Apache

Há duas configurações necessárias no Apache para que ele passe a suportar SSI. Uma deve ser realizada pelo administrador do servidor, que tem acesso aos arquivos *.conf. A outra, poderá ser realizada em arquivos .htaccess.

Instalação do suporte a SSI

É preciso configurar o servidor para que o ele trate de forma diferente arquivos .shtml (que ele os analise e não devolva simplesmente). É preciso também dizer ao servidor que o arquivo .shtml deverá ser devolvido ao browser como text/html. Se isto não for feito, o browser poderá receber o código-fonte da página.

Para dizer ao servidor que ele deve analisar um certo arquivo, é preciso defini-lo como "server-parsed" (analisado pelo servidor). Essa alteração é realizada no arquivo srm.conf (ou httpd.conf). O trecho abaixo diz ao servidor que arquivos com extensão `.shtml` devem ser analisados:

```
AddHandler server-parsed .shtml
```

É preciso agora acrescentar, ao tipo de dados text/html, a extensão de arquivo .shtml, para que o servidor possa gerar corretamente o Content-type no cabeçalho de resposta:

```
AddType text/html .shtml
```

Ainda falta configurar a segurança. Normalmente, SSI está desligado em servidores Apache. É preciso permitir a interpretação dos comandos.

Controle da segurança

Esta parte pode ser configurada em dois lugares: no `access.conf` (ou `httpd.conf`), acessível pelo administrador do servidor, ou no `.htaccess`, sob o controle do administrador do site. A alteração é realizada na diretiva `Options` para cada diretório afetado (se for aplicado na raiz de documentos afetará todo o site.

Os servidores *Apache* permitem dois níveis de segurança para SSI. O mais baixo permite a execução de todos os comandos suportados pelo servidor. O outro suporta todos os comandos com exceção do comando `<!--#EXEC -->` que inclui texto na página mediante a execução de um programa no servidor. A seguir, um trecho do `access.conf` (ou `httpd.conf`) correspondente ao diretório raiz de documentos (`DocumentRoot`), permitindo a execução de todos os SSI inclusive o `<!--#EXEC -->`.

```
<Directory /usr/apache/htdocs>
   Options Indexes FollowSymLinks Includes ExecCGI
   AllowOverride None
   order allow,deny
   allow from all
</Directory>
```

Para não permitir a execução de `<!--#EXEC -->` basta omitir a opção `ExecCGI`. Se `Includes` não estiver presente, nenhum tipo de SSI funcionará.

Para configurar o suporte a SSI no .htaccess, basta acrescentar as opções na linha Options. Um arquivo `.htaccess` mínimo que permite qualquer tipo de SSI no seu diretório contém:

```
Options Includes ExecCGI
```

Para que páginas SSI funcionem, basta agora terem a extensão .shtml e conterem comandos SSI.

10.3. SSI em servidores Microsoft

Esta edição provisória não mostra como configurar SSI em servidores Microsoft. Aguarde a publicação de um anexo no site do curso.

10.4. Instruções mais comuns

Os comandos SSI geralmente são dependentes do fabricante do servidor, ou dos *plug-ins* suportados pelo servidor. Há, porém, uma coleção de instruções disponíveis na maioria dos servidores modernos. Teste os comandos abaixo no seu servidor. Não se esqueça de verificar antes se o serviço está habilitado e se a página possui a extensão de nome de arquivo correta.

<!--#ECHO *atributo*-->

Inclui o conteúdo de uma variável de ambiente CGI no documento.

Atributos:

- `VAR="variável-de-ambiente"` armazena o nome da variável de ambiente a retornar. *Exemplo:*

  ```
  <!--#ECHO VAR="LAST_MODIFIED"-->
  ```

  inclui na página a data da última modificação do documento. Há várias variáveis de ambiente que só são visíveis via SSI.

<!--#INCLUDE *atributo*-->

Adiciona todo o conteúdo de um arquivo ao documento, antes de enviá-lo ao cliente. Suporta um atributo, que pode ser `FILE` ou `VIRTUAL`. Este é o comendo SSI de maior suporte. Funciona na maioria dos servidores Windows e Unix.

Atributos:

- `FILE="arquivo"`. Informa a localização de arquivo no sistema de arquivos do sistema operacional local (caminho relativo ou absoluto).

  ```
  <!--#INCLUDE FILE="c:\paginas\htdocs\docs\extra.txt" -->
  ```

- `VIRTUAL="arquivo"`. Informa a localização de arquivo no sistema de arquivos do servidor (caminho relativo ou absoluto). Exemplo:

  ```
  <!--#INCLUDE VIRTUAL="/docs/extra.txt" -->
  ```

<!--#EXEC *atributo*-->

Inclui no documento os valores retornados pela execução de um programa no servidor (provoca a execução desse programa). Suporta um atributo, que pode ser `CMD` ou `CGI`.

Atributos:

- `CMD="comando"`. Informa o comando do sistema a ser executado.
- `CGI="/cgi-bin/programa.pl"`. Informa, a partir do diretório raiz do servidor, o caminho (diretório CGI) e o nome de um programa CGI a ser executado. *Exemplo:*

  ```
  <!--#EXEC CGI="/cgi-bin/counter.pl?doc=includes.html"-->
  ```

  aciona o programa `counter.pl` que conta o número de vezes que o documento foi acessado:

<!--#FSIZE *atributo***-->**

Inclui a informação sobre o tamanho de um determinado arquivo no documento. Suporta um atributo, que pode ser `FILE` ou `VIRTUAL`.

Atributos:

- `FILE="arquivo"`. Informa a localização de arquivo no sistema de arquivos do sistema operacional local (caminho relativo ou absoluto).

```
<!--#FSIZE FILE="c:\paginas\htdocs\img\imagem.gif" -->
```

- `VIRTUAL="arquivo"`. Informa a localização de arquivo no sistema de arquivos do servidor (caminho relativo ou absoluto). Exemplo:

```
<!--#FSIZE VIRTUAL="/img/imagem.gif" -->
```

<!--#FLASTMOD *atributo***-->**

Inclui a informação sobre a data da última modificação de um determinado arquivo no documento. Suporta um atributo, que pode ser `FILE` ou `VIRTUAL`.

Atributos:

- `FILE="arquivo"`. Informa a localização de arquivo no sistema de arquivos do sistema operacional local (caminho relativo ou absoluto).
- `VIRTUAL="arquivo"`. Informa a localização de arquivo no sistema de arquivos do servidor (caminho relativo ou absoluto).

<!--#CONFIG *atributo* **-->**

Modifica vários aspectos dos dados importados por SSI.

Atributos

- `errmsg="mensagem de erro".` Muda o formato da mensagem de erro padrão.
- `sizefmt="bytes | abbrev"`. Muda o formato de exibição dos tamanhos de arquivo. Pode ser abreviado (MB, kB, etc.) ou em bytes.
- `timefmt="código de data/hora"`. Muda o formato de exibição de datas.

10.5. Exemplos

Nas seções a seguir, apresentamos alguns exemplos e aplicações de SSI em documentos HTML. A seção a seguir contém uma página de testes que você poderá usar para saber quais os comandos SSI que são suportados pelo seu servidor.

Página de testes

O programa abaixo (disponível no disquete que acompanha esta apostila) testa o suporte dos comandos SSI listados acima no seu servidor. Ele tentará incluir arquivos, executar programas e imprimir diversas variáveis de ambiente.

```
<HTML>
<HEAD>
<TITLE>Teste de Server-Side Includes</TITLE>
</HEAD>

<body bgcolor="#FFFFFF">
```

```
<H1>Server-Side Includes</H1>

<P>Variáveis de configuração SSI (observe que comentários podem ser usados
dentro dos descritores):
     <PRE>&lt;!--#config timefmt="%c" formato simples de data/hora --&gt;
     <!--#config timefmt="%c" formato simples de data/hora -->
     &lt;!--#config sizefmt="%d bytes"--&gt;
     <!--#config sizefmt="%d bytes"-->
     &lt;!--#config errmsg="##ERRO!##"--&gt;
     <!--#config errmsg="##ERRO!##"--></PRE>

<p>As diretivas SSI abaixo estão executadas ao lado. Para ver o arquivo original
     é necessário visualizá-lo no seu diretório e não através do servidor.
     <p><b>Configurações necessárias:</b>
     1) Para testar a diretiva EXEC é necessário informar o nome um arquivo CGI
        que retorne texto e colocá-lo no lugar de "progcgi.exe". 2) Para testar
        as diretivas INCLUDE, FSIZE e FLASTMOD é preciso substituir ssitext.txt
        e ssifile.ext por arquivos existentes no mesmo diretório onde está a
        página SHTML.

<PRE><B>DIRETIVA SSI                                          RESULTADO</B>
<HR>
&lt;!--#exec cgi="/cgi-bin/progcgi.exe"--&gt;  <!--#exec cgi="/cgi-bin/ssicgi.exe"-->
&lt;!--#include file="ssitext.txt"--&gt;   <br><!--#include file="ssitext.txt"-->
&lt;!--#fsize file="ssifile.txt"--&gt;         <!--#fsize file="ssifile.txt"-->
&lt;!--#flastmod file="ssifile.txt"--&gt;      <!--#flastmod file="ssifile.txt"-->
&lt;!--#echo var="DOCUMENT_NAME"--&gt;         <!--#echo var="DOCUMENT_NAME"-->
&lt;!--#echo var="DOCUMENT_URI"--&gt;          <!--#echo var="DOCUMENT_URI"-->
&lt;!--#echo var="LAST_MODIFIED"--&gt;         <!--#echo var="LAST_MODIFIED"-->
&lt;!--#echo var="QUERY_STRING"--&gt;          <!--#echo var="QUERY_STRING"-->
&lt;!--#echo var="QUERY_STRING_UNESCAPED"--&gt;<!--#echo var="QUERY_STRING_UNESCAPED"-->
&lt;!--#echo var="DATE_LOCAL"--&gt;            <!--#echo var="DATE_LOCAL"--&gt;
&lt;!--#echo var="DATE_GMT"--&gt;              <!--#echo var="DATE_GMT"-->
&lt;!--#echo var="SERVER_SOFTWARE"--&gt;       <!--#echo var="SERVER_SOFTWARE"-->
&lt;!--#echo var="SERVER_NAME"--&gt;           <!--#echo var="SERVER_NAME"-->
&lt;!--#echo var="SERVER_PROTOCOL"--&gt;       <!--#echo var="SERVER_PROTOCOL"-->
&lt;!--#echo var="REQUEST_METHOD"--&gt;        <!--#echo var="REQUEST_METHOD"-->
&lt;!--#echo var="REMOTE_HOST"--&gt;           <!--#echo var="REMOTE_HOST"-->
&lt;!--#echo var="HTTP_ACCEPT"--&gt;           <!--#echo var="HTTP_ACCEPT"-->
&lt;!--#echo var="HTTP_USER_AGENT"--&gt;       <!--#echo var="HTTP_USER_AGENT"-->
&lt;!--#echo var="REFERER"--&gt;               <!--#echo var="REFERER"-->
&lt;!--#echo var="BOGUS"--&gt;                 <!--#echo var="BOGUS"--></PRE>
<HR>
</BODY>
</HTML>
```

Data de modificação

Uma das aplicações mais comuns de SSI, nos servidores que suportem o comando ECHO é a impressão da data de modificação do arquivo:

```
<p>Este arquivo foi modificado em <!--#echo var="LAST_MODIFIED"-->
```

O usuário que recebe a página HTML não tem como saber se houve no lugar onde aparece a data um comando SSI, pois o código ostra o texto resultante da transformação:

> Este arquivo foi modificado em Friday, Aug 13, 1999 23:59:45 GMT-03

Repositórios de imagens ou outros arquivos

SSI é útil para criar páginas que oferecem um repositório de imagens e programas para download. Você pode informar a data de alteração e o tamanho do arquivo automaticamente usando os comandos FSIZE e FLASTMOD:

```
<p><a href="/imagens/grande.jpg">Foto 1</a>
(<!--#fsize virtual="/imagens/grande.jpg "-->).
Modificada em <!--#flastmod virtual="/imagens/grande.jpg"-->
```

No browser, o texto aparecerá como:

> Foto 1 (25.5 kB). Modificada em Friday, Aug 6, 1999 12:20:45 GMT-03

Usando o comando CONFIG é possível mudar o formato das datas, para que possam ser usadas internacionalmente.

Cabeçalhos e rodapés

Talvez a maior utilidade de SSI seja para criar *templates*, ou seja, estruturas que são adicionados a todas as páginas. Isto pode ser uma barra de navegação, um rodapé padrão, um logotipo, etc. Considere um arquivo de textos contendo o seguinte código HTML, que é um menu de navegação que deve aparecer em cima e embaixo de todos os arquivos:

```
<table width=100% cellspacing=0 cellpadding=0 border=0>
<tr>
<td align=left><a href="javascript: viaMail()">
   <img src="email.gif" border="0"></a></td>
<td align=center><a href="javascript: marcador()">
   <img src="marcar.gif" border="0"></a></td>
<td align=center><A HREF="notas.html">
   <img src="notas.gif" border="0"></A></td>
<td align=right><A HREF="indice.html">
   <img src="indice.gif" border="0"></A></td>
</tr></table><p></P>
```

Ele pode ser incluído em todos os arquivos do site sem que o código precise ser repetido. Isto pode ser feito com um simples `<!--#include -->`:

```
(...) <BODY><TABLE WIDTH=450><TR><TD>
<!--#include virtual="menu.txt" -->

<H1>Capítulo X (...)
```

O resultado agora é que todas as páginas que possuírem o INCLUDE terão o mesmo menu no local onde antes estava o comando. Se esse menu precisar ser alterado, ele será alterado em um único lugar, e automaticamente modificará todas as páginas.

10.6. Exercícios

1. Escreva uma página HTML que inclua um rodapé padrão contendo informações sobre autoria e a última modificação do arquivo usando SSI.
2. Escreva uma página HTML que inclua o número gerado em arquivo pelo programa contador counter.pl (é necessário configurar suporte ao descritor exec) em uma área da página. O arquivo é um CGI e está disponível no disquete que acompanha esta apostila. Poderá não ser necessário instalá-lo. Verifique com seu instrutor se o programa está disponível em alguma área de CGI do servidor.
3. Escreva uma página HTML com uma tabela vazia que seja capaz de incluir linhas adicionais através de um arquivo de texto carregado via SSI. O arquivo de textos deve conter linhas de tabela da forma:

```
<tr><td>... texto ...</td><td>... texto... </td></tr>
<tr><td>... texto ...</td><td>... texto... </td></tr>
...
```

11. Cookies

A TECNOLOGIA CONHECIDA COMO HTTP Cookies, surgiu em 1995 como um recurso proprietário do browser Netscape, que permitia que programas CGI gravassem informações em um arquivo de textos controlado pelo browser na máquina do cliente. Por oferecer uma solução simples para resolver uma das maiores limitações do HTTP – a incapacidade de preservar o estado das propriedades dos documentos em uma mesma sessão – os cookies logo passaram a ser suportados em outros browsers e por linguagens e tecnologias de suporte a operações no cliente e servidor. Cookies são um padrão da Internet e sua especificação está publicada em um RFC.

Um cookie não é um programa de computador, portanto não pode conter um vírus executável ou qualquer outro tipo de conteúdo ativo. Pode ocupar no máximo 4 kB de espaço no computador do cliente. Um servidor pode definir no máximo 20 cookies por domínio (endereço de rede) e o browser pode armazenar no máximo 300 cookies. Estas restrições referem-se ao browser *Netscape* e podem ser diferentes em outros browsers.

Há várias formas de manipular cookies:

- Através de CGI ou outra tecnologia de servidor, como LiveWire, ASP, JSP, PHP ou Servlets, pode-se criar ou recuperar cookies.
- Através de JavaScript também pode-se criar ou recuperar cookies.
- Através do descritor <META> em HTML, pode-se criar novos cookies ou redefinir cookies existentes, mas não recuperá-los.

Um cookie é enviado para um cliente no cabeçalho HTTP de uma resposta do servidor. Além da informação útil do cookie, que consiste de um par nome/valor, o servidor também inclui um informações sobre o domínio e caminho do sistema de arquivos onde o cookie é válido, o seu tempo de validade e indicações se ele deve ou não ser criptografado.

11.1. Criação de cookies via cabeçalhos HTTP

Cookies podem ser criados através de um cabeçalho HTTP usando CGI. Como vimos nos capítulos anteriores, parte ou todo o bloco de cabeçalhos pode ser gerado por um programa CGI ou equivalente. Quando um programa CGI gera um cabeçalho não precisa se restringir ao `Content-type`. Pode também incluir campos de informação sobre a página que o servidor não inclui por *default*. Para criar cookies, por exemplo, o programa deverá formar o servidor a definir um ou mais cabeçalhos `Set-Cookie`, que irão instruir ao browser para que guarde a informação passada em cookies. Suponha que um servidor gere o seguinte cabeçalho:

```
HTTP/1.0 200 OK
Date: Friday, June 13, 1997
Server: Apache 1.02
Set-Cookie: cliente=jan0017
Set-Cookie: nomeclt=Marie
Content-type: text/html
```

```
<HTML><HEAD>
<TITLE> Capitulo 11</TITLE>
(...)
```

Quando receber a resposta do servidor e interpretar os cabeçalhos acima, o browser irá gravar dois novos cookies na memória, contendo as informações `cliente=jan0017` e `nomeclt=Marie`. Essas informações poderão ser recuperadas em qualquer página que tenha origem no servidor que definiu os cookies enquanto a presente sessão do browser ainda estiver aberta.

Um cookie tem um escopo de validade que é limitado pelo tempo, pelo domínio (nome da máquina ou rede) que gerou o cabeçalho para criá-lo, pelo caminho do sistema de arquivos onde reside o programa que o criou e pelo browser que realizou a requisição. Em outro browser, em outro caminho ou em outro domínio o cookie não existe. Ele também não existe se o seu período de vida chegar ao fim. No cookie acima, o tempo de vida é limitado à sessão do browser, mas isto pode ser alterado.

11.2. Alteração do tempo de vida

Um cookie criado da forma mostrada na seção anterior pode ser lido por outros programas ou páginas enquanto o browser não for fechado. Se o usuário fechar o browser e entrar numa página que tentar ler o cookie, ele não existirá mais. Mas é possível fazer com que ele persista, criando cookies persistentes. Se após o par nome/valor, o cabeçalho Set-Cookie tiver um parâmetro `expires` seguida por uma data no futuro, as informações do cookie serão gravadas em arquivo (e não apenas na memória RAM) e poderão persistir além da sessão atual do browser:

```
Set-Cookie: nomeclt=Marie; expires=Monday, 15-Jan-99 13:02:55 GMT
```

11.3. Espaço de nomes de um Cookie

Várias páginas de um site podem definir cookies. O espaço de nomes de um cookie é determinado através de seu domínio e caminho. Em um mesmo espaço de nomes, só pode haver um cookie com um determinado nome. A definição de um cookie de mesmo nome que um cookie já existente no mesmo espaço, sobrepõe o cookie antigo.

Por *default*, o espaço de nomes de um cookie é o domínio e caminho atual onde foi criado. Para fazer o cookie valer em outro domínio, mais restritivo ou pertencente à mesma rede, é preciso definir o campo `domain`. Por exemplo, se o domínio de um cookie é `chocolate.biscoitos.com`, ele só pode ser lido na máquina `chocolate.biscoitos.com` mas não em `agua.biscoitos.com` e. Para ampliar seu alcance à máquina `agua.biscoitos.com` e a outras máquinas do domínio, o campo `domain` deve ser especificado da forma:

```
Set-Cookie: nome=valor; domain=.biscoitos.com
```

Somente máquinas dentro do domínio `.biscoitos.com` podem redefinir o domínio. Eles não podem ser alterados para abranger domínios externos.

O caminho dentro do domínio onde o cookie é válido é o mesmo caminho onde foi criado. O caminho pode ser alterado de forma que tenha um valor mais amplo definindo o campo `path`. Por exemplo, se um cookie foi criado em um diretório `/bolachas/`, ele só será lido por páginas ou programas que estejam armazenados em `/bolachas/`. Programas ou páginas em / ou em /cgi-bin/, por exemplo, não conseguirão enxergá-lo. Para que seu escopo seja ampliado para /, o que o tornaria acessível em todo o site, é necessário que um caminho novo seja especificado da forma:

```
Set-Cookie: nome=valor; path=/
```

Um cookie chamado `bis` definido em `/` antes não colidia com um cookie também chamado `bis` definido em `/bolachas/`. Agora sim e o resultado pode ser imprevisível. É, portanto, preciso ter cuidado ao ampliar o escopo de cookies. Não há como isso afetar outros usuários, mas o mesmo usuário, se for visitar outra parte do site, poderá ter problemas.

Um cookie pode ser apagado se for definido um novo cookie com o mesmo nome e caminho/domínio e com data de vencimento (campo `expires`) no passado.

11.4. Parâmetros

É possível combinar todos os parâmetros. A ordem não importa. Todos são separados por ponto e vírgula e têm a forma propriedade=valor. A sintaxe completa do cabeçalho `Set-Cookie` está mostrada abaixo. Os campos são separados por ponto-e-vírgula. Todos, exceto o primeiro campo que define o nome do cookie, são opcionais.

```
Set-Cookie: nome_do_cookie=valor_do_cookie;
            expires=data no formato GMT (RFC 850);
            domain=domínio onde o cookie é válido;
            path=caminho dentro do domínio onde o cookie é válido;
            secure
```

Os parâmetros do cabeçalho `Set-Cookie` são usados na definição de cookies tanto em programas CGI quanto em roteiros JavaScript, rodando no cliente. O significado de cada parâmetro está explicado na tabela abaixo:

Parâmetro	Descrição
nome=valor	*Este campo é obrigatório.* Seqüência de caracteres que não incluem acentos, ponto-e-vírgula, percentagem, vírgula ou espaço em branco. Para incluir esses caracteres é preciso usar um formato de codificação estilo URL. Em *JavaScript*, a função `escape()` codifica informações nesse formato e a função `unescape()` as decodifica.

`expires=`*data*	*Opcional.* Se presente, define uma data com o período de validade do cookie. Após esta data, o cookie deixará de existir. Se este campo não estiver presente, o cookie só existe enquanto durar a sessão do browser. A data deve estar no seguinte formato:
	`DiaDaSemana, dd-mes-aaaa hh:mm:ss GMT`
	Por exemplo:
	`Monday, 15-Jan-1999 13:02:55 GMT`

`domain=`*domínio*	*Opcional.* Se presente, define um *domínio* onde o cookie atual é válido. Se este campo não existir, o cookie será válido somente no domínio onde o cookie foi criado.
`path=`*caminho*	*Opcional.* Se presente, define o *caminho* onde um cookie é válido em um domínio. Se este campo não existir, será usado o caminho do documento que criou o cookie.
`secure`	*Opcional.* Se presente, impede que o cookie seja transmitido a menos que a transmissão seja segura (baseada em SSL ou SHTTP).

11.5. Criação de cookies via HTML

Um cookie pode ser criado através de HTML usando o descritor <META> e seu atributo HTTP-EQUIV. O atributo HTTP-EQUIV deve conter um cabeçalho HTTP. O valor do cabeçalho deve estar presente

no seu atributo CONTENT. A presença do um descritor <META> dentro de um bloco <HEAD> de uma página HTML, criará um cookie no cliente quando este for interpretar a página.

```
<HEAD>
<META HTTP-EQUIV="Set-Cookie"
      CONTENT="nomeclt=Marie; expires=Monday, 15-Jan-1999 13:02:55 GMT">
(...)
</HEAD>
```

11.6. Recuperação de cookies

Toda requisição de um browser ao servidor consiste de uma linha que contém o método de requisição, URL destino e protocolo, seguida de várias linhas de cabeçalho. É através de cabeçalhos que o cliente passa informações ao servidor, como, por exemplo, o nome do browser que enviou o pedido. Uma requisição HTTP típica (capítulo 3) tem a forma:

```
GET /index.html HTTP/1.0
User-Agent: Mozilla/4.5 (WinNT; I) [en]
Host: www.alnitak.org.br
Accept: image/gif, image/jpeg, */*
```

Quando um cookie é recuperado pelo browser, ele é enviado em todas as requisições à URLs que fazem parte do seu espaço de nomes, através do cabeçalho do cliente `Cookie`. Apenas o par nome/valor é armazenado no cabeçalho. As informações dos campos `expires`, `path`, e `domain` não aparecem:

```
Cookie: cliente=jan0017; nomeclt=Marie
```

O servidor pode recuperar as informações do cookie através do cabeçalho ou através da variável de ambiente `HTTP_COOKIE`, definida quando o servidor recebe uma requisição com o cabeçalho Cookie. Para fazer uso dos dados de `HTTP_COOKIE`, é preciso tratar a string que a variável contém, separando cada cookie pelo ponto-e-vírgula e identificando nome e valor através do sinal de igualdade.

Cookies são armazenados em ASCII e contém vários caracteres reservados, entre eles o ponto-e-vírgula e o sinal de igualdade. Para evitar problemas, o cookie deve ser armazenado com esses caracteres convertidos em formatos codificados (x-www-form-encoding, por exemplo). Na recuperação dos dados, será necessário fazer uma decodificação. Várias linguagens oferecem ferramentas para esse tipo de codificação. Exemplos são Perl, C e JavaScript.

11.7. Programas CGI que criam cookies

Para manipular com cookies em Perl pode-se usar o módulo CGI:: ou bibliotecas semelhantes à `cgi-lib.pl`. Você pode usar a biblioteca `cookie.pl` (disponível no site do curso) para criar e apagar cookies em programas em Perl. Transfira-a para o seu diretório `cgi-bin` e carregue-a como foi feito com a biblioteca cgi-lib.pl:

```
require "cookie.pl";
require "cgi-lib.pl";
```

Agora você pode usar as funções `&MakeCookieHeader` para fazer o cabeçalho que cria cookies (antes de terminar o cabeçalho com uma linha em branco) e `&ReadCookies`, que funciona como o `&ReadParse` do `cgi-lib.pl` para ler os cookies criados.

`&MakeCookieHeader` cria cabeçalho de cookie (e conseqüentemente, cria um cookie no browser). Pode ser chamada várias vezes (para criar vários cookies) mas sempre *antes* de imprimir o fim do cabeçalho (antes do `'print &PrintHeader'` da biblioteca `cgi-lib.pl` ou antes de *qualquer* linha em branco ("`xxx xxx \n\n`"). Deve ser chamada como o `&PrintHeader`, precedida de print.

A função deve ter no mínimo dois argumentos. A sintaxe geral é:

```
print &MakeCookieHeader("nome", "valor", vida_em_dias,
                        "caminho", "dominio", seguranca);
```

Os dois primeiros campos (obrigatórios) contém o nome e valor do cookie (que podem conter espaços, etc. Todos os outros campos são opcionais.

O tempo de vida é um número de ponto-flutuante que representa o número de dias que o cookie vai durar. Só o número for negativo, será criado um anti-cookie (que dizimará qualquer outro cookie do mesmo nome que ele encontrar). É um campo opcional. Se não estiver presente, o cookie durará apenas até o fim da sessão.

O quarto campo informa o caminho do diretório abaixo do qual o cookie é válido. Se ausente, o cookie valerá apenas no diretório atual. Este campo é obrigatório para cookies persistentes ou não que seja criados em um lugar (por exemplo, o diretório `cgi-bin/`) e lidos em outro lugar (por exemplo, através de JavaScript em um diretório de documentos qualquer como o "`/`"). Se o cookie não for persistente, deve-se usar `'undef'` no lugar do número de dias.

O penúltimo campo informa o domínio abaixo do qual o cookie é válido. Deve ter no mínimo dois pontos (pode começar por ponto, por exemplo: .abc.com). Os campos anteriores, se não definidos, deverão conter o valor `'undef'`. Não é feita verificação de domínio (ou caminho), portanto, tenha cuidado em informar um domínio válido.

O último campo, se presente, deve conter um número positivo, e indica que o cookie só deve ser criado ou enviado em uma conexão segura. Se a conexão não for segura, ele não é criado ou enviado para o servidor pelo browser (mesmo que o domínio e caminho coincidam).

Veja alguns exemplos de criação de cookies. Cookie temporario:

```
print &MakeCookieHeader("usuario", "josé");
```

Cookie persistente que durará 15 dias:

```
print &MakeCookieHeader("usuario", "joao", 15);
```

Cookie com duração de 15 dias, no domínio abc.com.br, no caminho / e seguro:

```
print &MakeCookieHeader("usuario", "joao", 15, "/",
                        ".abc.com.br", 1);
```

Parâmetros não utilizados devem ser preenchidos com `undef`:

```
print &MakeCookieHeader("usuario", "joao", undef,
                         undef, ".abc.com.br");

print &MakeCookieHeader("usuario", "joao", undef, "/");
```

Lembre-se que todos os comandos acima devem ocorrer antes de qualquer linha em branco (pois ela terminará o cabeçalho).

11.8. Programas CGI que recuperam cookies

Para recuperar os cookies, eles devem estar no mesmo caminho e domínio e o browser que os lê também deve ser o mesmo. A biblioteca cookie.pl oferece a função `&ReadCookies` que funciona da mesma forma que

`&ReadParse` do `cgi-lib`: `&ReadCookies` lê todos os cookies e os coloca em vetor associativo escolhido pelo programador:

```
&ReadCookies(\%crackers);
```

Agora é só extrair o cookie, informando o seu nome no vetor associativo recém criado:

```
$cookie1 = $crackers{'nome1'}; # obtem o valor, informando-se o nome
$cookie2 = $crackers{'nome2'};
```

11.9. Exercícios

1. Crie um programa CGI (assacookie.pl) que ofereça um formulário HTML que pede o nome do usuário. Grave o nome do usuário como um cookie. Em outra página (comecookie.pl), leia a variável de ambiente e imprima na página o nome do usuário com uma mensagem de boas vindas.
2. Crie uma página HTML, com três botões de rádio (<INPUT TYPE=RADIO>). A página deve enviar seus dados para o programa computa.pl que primeiro tenta ler um cookie chamado "num". Se "num" não existir, ele deverá gravar um "cookie" num com o valor 1. Se "num" existe, ele deve somar 1 ao valor de "num" e gravar um novo cookie com a soma. Depois, deve gravar outro cookie "opcao" com a opção escolhida. O programa computa.pl deverá gravar os cookies e depois imprimir uma linha de cabeçalho "Location: http://URL" e terminar o bloco de cabeçalho com uma linha em branco (não deverá ter "Content-type"). Location redireciona a página. A URL deverá ser o da página que contém os botões. Na quinta vez em que o usuário clicar os botões, o programa não devera redirecioná-lo para outra página, mas terminar o cabeçalho com "Content-type" e gerar uma página HTML informando as opções que o usuário fez.
3. Faça um contador de visitas para que um usuário saiba quantas vezes ele já visitou sua página. O cookie deve ser persistente para que não desapareça quando o usuário fechar o browser. Faça-o durar 5 dias.

12. Princípios de programação

O objetivo deste capítulo é oferecer uma introdução aos princípios de programação de computadores que será necessária para o módulo seguinte.

Para programar é preciso saber uma linguagem de programação. Mas para aprender tal linguagem, é preciso não só praticar, mas também conhecer como funciona um programa e como o código de um programa é estruturado. A maior parte das linguagens de programação usadas hoje em dia tem estruturas semelhantes, que são usadas para controlar *quando* e *como* as instruções serão executadas. Os conceitos aqui apresentados têm por finalidade ajudar àqueles que nunca programaram a identificar as principais partes de um programa e ganhar alguma experiência na análise de programas, para que sejam capazes de aprender JavaScript – apresentada no módulo seguinte – e alterar ou criar programas em Perl (se desejar) para utilizar em seu site.

Perl não é objeto central deste curso e está apresentada nesta apostila em um apêndice.

12.1. Tipos de linguagens

Linguagens de computador podem ser classificadas de várias formas. Em um capítulo anterior as classificamos em gerações, que descreviam o quanto tais linguagens se distanciavam da linguagem da máquina e se aproximavam da linguagem humana.

Todas as linguagens contém *instruções*, que podem ser simples ou compostas por várias outras instruções mais simples. Nas linguagens de *alto-nível* (mais distantes da linguagem de máquina) essas instruções são geralmente expressas em linguagem humana (ou algo parecido). Uma instrução geralmente sugere *ação*, e *verbos* formam a maior parte das instruções das linguagens que são usadas para construir *procedimentos*. Por exemplo, um programa que lê um texto e o traduz para outra língua provavelmente terá instruções como "leia", "compare", "altere", expressas no vocabulário específico da linguagem. Existem também linguagens que não contém verbos, mas *substantivos*. HTML é um exemplo. Nesse caso, o interpretador já sabe quais ações irá tomar, mas depende da linguagem para saber *como* deverá proceder.

Linguagens procedurais

As linguagens que contém instruções capazes de expressar ações possuem a capacidade de construir procedimentos ou programas. Linguagens que são totalmente dependentes das instruções de ação são chamadas de *linguagens procedurais*. Exemplos são os programas em Shell que construímos em capítulos anteriores, que consistiam de uma lista de comandos Unix. Outras linguagens procedurais são C, Pascal, FORTRAN, BASIC, COBOL e Perl.

Linguagens declarativas

Linguagens que apenas declaram *como* um interpretador deve agir são *linguagens declarativas*. Não é possível construir programas apenas usando linguagens declarativas. Com HTML você pode descrever totalmente um texto. Com XML você pode descrever qualquer tipo de dados ou informação. Com CSS você pode descrever a cor e o tamanho de um parágrafo, mas quem faz o texto mudar de cor é o interpretador embutido no navegador. Não é possível com essas linguagens criar novos comportamentos que o navegador não tenha previsto.

Linguagens baseadas ou orientadas a objetos

As linguagens modernas raramente são 100% procedurais. Ela geralmente combinam estruturas com procedimentos, tratando as estruturas declaradas como *objetos*, e oferecendo instruções de ação que permitem manipular não só valores individuais como números e palavras, mas estruturas inteiras, que podem consistir de coleções de números e palavras. JavaScript, por exemplo, quando executado por um navegador, comporta-se como uma *linguagem baseada em objetos*. É possível, através de instruções da linguagem, ligar o evento de se mover um mouse sobre uma imagem à mudança da cor do texto de um parágrafo. CSS define a cor do parágrafo, que é formado de texto identificado por HTML. O browser, ao ler HTML e CSS, define sua cor e tamanho iniciais e depois mantém a página estática. Instruções de JavaScript, portanto, introduzem um novo comportamento (não previsto no browser), que podem identificar o objeto (pela estrutura do HTML) e alterar seus atributos (declarados em CSS).

Há várias outras linguagens baseadas ou orientadas a objetos. A maioria têm sua própria sintaxe e seus próprios mecanismos para identificar, manipular e operar com objetos. Exemplos são C++, Delphi, Java, Visual Basic e Perl 5.

12.2. Componentes de um programa

É comum usar um programa para realizar um determinado cálculo e, a partir dos resultados obtidos, realizar uma ou outra operação. Outra tarefa freqüente é a repetição. Pode-se criar um programa para repetir um certo procedimento cem vezes ou até que certa condição tenha sido atingida. Nas linguagens antigas, cada instrução ocorria em uma linha numerada. Para repetir uma seqüência de comandos, precisava-se saber qual a linha onde a seqüência começava e escrever uma instrução que pulasse àquela linha.

Nas linguagens modernas, a realização de operações é feita de forma mais ou menos padronizada, e não depende mais de linhas numeradas. Todas as linguagens modernas (chamadas de estruturadas) possuem estruturas usadas para controlar o fluxo de controle do programa. Uma seqüência de instruções pode ser agrupadas em um *bloco* e chamado como um roteiro externo ou *subrotina*. Valores podem ser armazenados em *variáveis*. Coleções de valores podem ser armazenadas em *estruturas de dados* e repetições e decisões podem ser realizadas usando *estruturas de controle de fluxo*. Todos esses termos, presentes em linguagens procedurais e linguagens baseadas em objetos), serão definidos abaixo.

Algoritmo: fluxo de controle

Qualquer procedimento segue uma seqüência de instruções que tem início e fim. Essa seqüência é chamada de algoritmo. Entre o início e o fim pode haver instruções que forcem a repetição de trechos do procedimento, que façam cálculos, que tomem diferentes decisões baseadas nesses cálculos ou até que forcem a saída do procedimento antes do seu término.

Um algoritmo pode ser tão simples a ponto de apenas ter instruções que são executadas em seqüência, com um resultado final previsível; ou ser complicado a ponto de ter milhares de resultados possíveis, dependendo de condições internas e externas, e apresentar resultados difíceis de se prever. A complexidade de um algoritmo às vezes é desnecessária. Algoritmos complexos não só são difíceis de analisar (e consertar, quando há erros), mas

também afetam o desempenho do computador e, devido à sua imprevisibilidade, podem oferecer riscos à segurança do programa.

Variáveis

Variáveis são abstrações da memória de um computador e permitem o armazenamento de valores temporários usados para realizar cálculos, transformações em textos, etc. Dependendo do que se armazena em uma variável, o espaço em memória (em bits) ocupado poderá ser maior ou menor. A forma como os dados armazenados são interpretados também são importantes. Um número armazenado pode ser um número inteiro ou apenas a parte do número que segue depois da vírgula. Pode também ser apenas um código de cor, ou de caractere, ou ainda ter dígitos que identificam seu sinal.

Por causa das diferenças em tamanho e forma, toda variável precisa não só identificar o valor que contém, mas também o seu *tipo*. Através do tipo, o programa sabe quanto espaço a variável requer e como as informações armazenadas devem ser interpretadas. Tipos de dados comuns são *número* (inteiro de vários tamanhos ou de ponto flutuante), *caractere* (número sem sinal que representa uma letra ou símbolo) ou *estado booleano* (indicador de estado verdadeiro ou falso).

Esses tipos são também chamados de *tipos primitivos*, pois são fundamentais em uma linguagem e não podem ser divididos em partes menores. Outros tipos de dados como datas, por exemplo, são *complexos* (uma data pode ser dividida em três números, representando dia, mês e ano). Tipos complexos, em linguagens orientadas a objetos, são definidos através de estruturas de dados chamadas de *objetos*.

Em um programa, variáveis precisam receber seu valor através de uma operação de *atribuição*. A forma de fazer isto depende de cada linguagem. Veja alguns exemplos:

```
a = 5;        (Java, JavaScript e C)
a := 5;       (Delphi, Pascal)
$a = 5;       (Perl)
```

O nome da variável acima é "a". Esse nome é chamado de *identificador*. Em Perl os identificadores de variáveis precisam começar com um $. Nas outras linguagens acima pode ser um nome que não comece com número. Todas as instruções acima atribuem (armazenam) o valor 5 na variável "a".

Algumas linguagens, a partir da atribuição, automaticamente adivinham o tipo de dados armazenado da variável (por exemplo, se é número, se é texto, se é booleano). Outras exigem que, antes de qualquer atribuição, o tipo de dados da variável seja *declarado*. Isto ocorre em linguagens como Java ou C, mas não em JavaScript ou Perl:

```
int a;        (declara que "a" armazena valores numéricos inteiros)
a = 5;
```

O trecho acima é uma declaração Java ou C.

Subrotinas

Um programa consiste de uma seqüência de instruções. Freqüentemente, várias seqüências precisam ser repetidas mais de uma vez. Elas são completas e funcionam como pequenos programas. Por exemplo, considere um programa que calcula taxas de juros. Ele terá que várias vezes repetir uma seqüência de operações de soma e multiplicação. A seqüência de operações que realiza o somatório poderia ser considerado um programa em si. A seqüência pode então receber um nome e ser definida como *subrotina*, podendo então ser chamada (pelo nome) várias vezes dentro do programa principal.

Subrotinas são usadas em todas as principais linguagens de programação. As linguagens estruturadas as chamam de *funções*. As linguagens orientadas a objetos as chamam de *métodos*. Os termos não significam exatamen-

te a mesma coisa mas todos referem-se a seqüências de instruções, localizadas fora do programa principal, e que podem ser identificadas pelo nome.

Algumas subrotinas são simplesmente seqüências de instruções independentes de qualquer valor inicial e que não retornam qualquer valor como resultado. Algumas exigem parâmetros de entrada e podem devolver um valor como resultado. Um exemplo é uma subrotina que realiza uma soma. Ela precisa receber como entrada dois números, e no final devolve o resultado que é a soma deles. As instruções em seu interior nada interessam ao programa principal, apenas a sua entrada e saída.

Assim como as variáveis, cada linguagem tem a sua própria forma de representar subrotinas, funções ou métodos. Em Perl, as subrotinas são representadas por uma seqüência de instruções entre chaves, precedidas pelo seu nome (o identificador da subrotina):

```
sub imprime {
   print "Linha 1: Olá! \n";
   print "Linha 2: Tchau! \n";
}
```

A subrotina acima se chama imprime, e pode ser chamada várias vezes de um programa em Perl usando a notação:

```
&imprime;
```

Um & é usado em Perl para indicar que o nome que segue é identificador de uma subrotina que deve ser chamada. Um ponto-e-vírgula termina todas as instruções em Perl.

Subrotinas que retornam valores podem ter esse valor atribuído diretamente a uma variável:

```
$resultado = &soma(5, 6);
```

Os parênteses que seguem o identificador da subrotina são os seus parâmetros. A maior parte das linguagens usa essa notação ou uma notação semelhante para indicar os parâmetros. Em Perl, a chamada acima poderia ser usada para invocar a subrotina:

```
sub soma {
    ($a, $b) = @_;          (obtém os parâmetros passados e os copia em $a e $b)
    return $a + $b;
}
```

A instrução return devolve o valor resultante da expressão $a + $b.

Em JavaScript, uma subrotina (ou função) é definida da forma:

```
function soma(a, b) {
    return a + b;
}
```

e pode ser chamada da forma:

```
resultado = soma(5, 6);
```

Neste caso, 5 é copiado em a e 6 em b.

Em Java, C, Delphi, VB a sintaxe é outra, mas as formas de chamar, passar parâmetros e obter resultados retornados é parecida. Quando você aprende uma linguagem estruturada, o aprendizado de uma segunda ou terceira é bem mais fácil.

Exercício

1. Abra um programa (código-fonte) escrito em JavaScript ou em Perl. Procure e tente identificar as variáveis, chamadas e definições de funções e subrotinas.

Estruturas de dados e objetos

A maior parte das linguagens permite que se manipule não só com valores primitivos como números e texto, mas também com estruturas complexas. Nas linguagens baseadas em objetos ou orientadas a objetos, geralmente não é necessário criar tais estruturas, pois grande parte delas já está pronta.

Com estruturas, pode-se ter *tipos de dados abstratos* definidos como coleções de tipos primitivos. Essas coleções podem ser manipuladas através de uma única variável através da qual se possa extrair as propriedades do objeto.

Uma das estruturas mais simples é o *vetor*, que consiste de uma coleção ordenada de valores, geralmente (mas não sempre) do mesmo tipo. O vetor permite que seus elementos sejam recuperados através de um índice que indique a sua posição. Um vetor em Perl pode ser definido da seguinte maneira:

```
@meses = ('janeiro', 'fevereiro', 'março', 'abril', 'maio', 'junho');
```

@meses é o identificador. Identificadores que começam com @ em Perl são vetores. Mas @ é usado somente na definição. Para obter um dos elementos do vetor acima, é preciso saber sua posição. A contagem começa de 0:

```
$mes = $meses[3];
```

O índice é passado entre colchetes logo após o identificador do vetor, desta vez precedido por um $ (pois um elemento do vetor é um valor primitivo). O índice 3 corresponde ao valor "abril".

Em JavaScript, vetores são criados da seguinte maneira:

```
meses = new Array('janeiro', 'fevereiro', 'março', 'abril', 'maio', 'junho');
```

A forma de manipulação é a mesma (só não tem o @ nem o $):

```
mes = meses[3];
```

Outra estrutura comum é o *hash*, também chamado de *vetor associativo* ou *hashtable*. O hash consiste em coleções formadas por pares de valores. Um dos valores serve de chave para se obter o segundo. Por exemplo, pode-se ter um hash para armazenar cores HTML onde utilizando a palavra "azul" como chave, se obtenha o número 0000ff. Em Perl, Um hash pode ser definido através de um vetor atribuído a um identificador precedido de %:

```
%cores = ('vermelho', 'ff0000', 'verde', '00ff00', 'azul', '0000ff');
```

Na definição, os pares são ordenados mas isto não influi na forma de obter os resultados. A cor correspondente a verde pode ser obtida fazendo:

```
$codigo = $cores{'verde'};
```

Observe que o "%" só aparece quando o hash é definido. Quando é usado aparece com um "$" antes. Uma nova cor pode ser colocada no hash fazendo simplesmente a atribuição:

```
$cores{'magenta'} = 'ff00ff';
```

Em uma linguagem baseada em objetos como JavaScript, as estruturas definidas em uma página HTML podem ser manipuladas pelo programa como *objetos*. Objetos funcionam como coleções de valores que podem ser atribuídos a uma variável (como vetores e hashes), mas podem também conter coleções de instruções (métodos).

Em um objeto, os valores que armazena são suas propriedades. Em JavaScript, propriedades são identificadas (ou criadas) com identificador próprio, ligado ao identificador do objeto através do ponto "." O objeto cores abaixo está definindo três propriedades:

```
cores.verde = "00ff00";
cores.azul = "0000ff";
cores.vermelho = "ff0000";
```

Agora uma das propriedades foi copiada para uma variável:

```
cor = cores.verde;
```

A variável cor contém o valor "00ff00".

Em Perl 5, que é uma linguagem orientada a objetos, as propriedades de um objetos são acessadas através do "->" que funciona de forma idêntica ao ponto:

```
cores->azul = "0000ff";
```

Algumas propriedades de objetos realizam ações. São os métodos. Em JavaScript eles são fáceis de identificar pois sempre têm um par de parênteses após o identificador (como as funções), mesmo que não tenha argumentos:

```
document.write("<p>Linha 2");
hoje.toGMTString();
```

Nos exemplos acima, document e hoje são objetos. O primeiro é um objeto criado pelo browser e representa a página HTML.

Exercício

2. Abra programas em Perl e páginas HTML com JavaScript e tente identificar vetores, objetos, hashes, propriedades e métodos.

Operadores e expressões

Linguagens têm uma coleção de *operadores* com os quais se pode formar expressões e, com elas, instruções. Já vimos o importante operador de atribuição, que permite copiar um valor para uma variável. Todos os programas também possuem os operadores básicos de aritmética: soma (+), subtração (-), multiplicação (*) e divisão (/). Outros operadores como resto, expoentes, raiz quadrada, operadores de comparação, operadores booleanos, etc. podem estar presentes mas apresentar uma sintaxe diferente. Às vezes um operador não está presente na linguagem como símbolo, mas somente através de uma função (subrotina) que é chamada externamente.

Combinando valores, operadores e variáveis obtém-se *expressões*. Toda expressão possui um *tipo de dados* (geralmente o tipo de dados da variável à qual o valor final é atribuído). Expressões podem também ser combinadas, organizadas em estruturas de controle de fluxo (seção seguinte) de forma a elaborar algoritmos complexos.

As diferentes sintaxes entre linguagens às vezes podem confundir. Por exemplo, o símbolo "." (ponto), usado em JavaScript para referenciar propriedades de objetos é usado em Perl para concatenar texto:

```
$dia = "sexta";
$frase = "Hoje é" . $dia . "-feira!";
```

Em JavaScript, a concatenação é feita usando o mesmo operador usado na adição:

```
dia = "sexta";
frase = "Hoje é" + dia + "-feira!";
```

Ambas as linguagens usam o operador "=" exclusivamente para atribuição. Para comparar um valor com outro é utilizado o operador "==":

```
if (dia == "sexta") { ... }
```

Exercícios

3. Abra um programa em Perl (ou JavaScript) e identifique suas expressões e tente compreender o funcionamento de algumas de suas instruções.

Estruturas de controle de fluxo

Para controlar a forma como uma seqüência de instruções é executada, a maior parte das linguagens estruturadas conta com estruturas de controle de fluxo. Elas permitem a realização de repetições e tomada de decisões. Sempre afetam um bloco de instruções. As principais estruturas são as estruturas IF (se alguma condição for verdadeira execute o bloco) e WHILE (enquanto alguma condição for verdadeira execute o bloco), presentes em todas as linguagens.

A estrutura IF geralmente opera sobre uma expressão booleana (um teste) que resulta em um valor verdadeiro ou falso. Ela precede um bloco de instruções que só será executado se a expressão for verdadeira. Tanto em Perl como em JavaScript, o bloco é delimitado por chaves { e }, e o IF escrito em letras minúsculas, antes de uma condição entre parênteses:

```
if (indice > 16) {
  ... instruções ...
}
```

Nem todas as linguagens usam chaves para iniciar e fechar blocos. Umas usam as palavras BEGIN e END, outras usam o nome da estrutura ao contrário (IF e FI, WHILE e ELIHW).

O IF, portanto, serve para que um programa possa tomar decisões. Às vezes ele deseja fazer algo quando a expressão do IF não resultar verdadeira. Ele pode controlar isto usando os operadores da linguagem para negar a expressão ou usar uma estrutura ELSE (caso-contrário). ELSE complementa o IF:

```
if (indice > 16) {
  ... instruções que serão executadas se indice for maior que 16
} else {
  ... instruções executadas se indice for maior ou igual a 16.
}
```

O IF pode ser combinado com o ELSE para criar estruturas de decisão mais complexas usando IF-ELSE IF-ELSE. A estrutura abaixo é válida em Perl:

```
if ($indice > 16) {
  ... instruções que serão executadas se indice for maior que 16
} elsif ($indice <= 20) {
  ... instruções executadas se indice for menor ou igual a 20.
} else {
  ... executadas se indice for maior ou igual a 16 e menor que 20.
}
```

A estrutura WHILE é usada para realizar repetições. A cada repetição o programa testa sua condição. Se a condição continuar verdadeira, uma nova rodada é executada. Se for falsa, o programa deixa o bloco WHILE. É

preciso que a condição mude de alguma maneira ou a repetição jamais chegará ao fim. A sintaxe de WHILE é a mesma em Perl e JavaScript:

```
x = 10;
while (x >= 0) {
  ... instruções ...
  x = x - 2;
}
```

As instruções acima serão repetidas cinco vezes pois a cada repetição, a variável testada (inicializada com o valor 10) diminui (na expressão x = x – 2), até que chegará o momento em que a expressão do WHILE será falsa, e o controle deixará o bloco.

A expressão x = x – 2 é uma operação de decremento. O seu oposto é uma operação de incremento. Tais operações são muito comuns em programas e são essenciais para que possa existir controle de blocos de repetições. A combinação inicialização – teste – incremento ou decremento é tão comum que há uma estrutura que já embute as três operações em sua declaração para realizar repetições: a estrutura FOR.

A sintaxe de FOR em JavaScript e Perl é a mesma. As três operações aparecem separadas por pontos e vírgulas:

```
for (x = 10; x >= 0; x = x - 2) {
  ... instruções ...
}
```

O bloco acima faz o mesmo que o WHILE mostrado antes. A inicialização é feita apenas uma vez, o teste é executado cada vez que o bloco chega ao fim e o incremento ou decremento é executado logo após a última instrução do bloco.

As linguagens geralmente possuem outras estruturas além das mostradas, mas com estas três é possível escrever qualquer algoritmo. Conhecendo e sabendo identificar essas estruturas no código-fonte de um programa permitirá analisar e saber acompanhar qualquer programa.

Exercícios

4. Identifique as estruturas WHILE, FOR e IF em programas Perl e páginas HTML com JavaScript. Tente, através delas, prever o funcionamento do programa.

12.3. Como escrever um programa

Um programa geralmente pode ser escrito usando um editor de textos comum. Ao escrever programas em Perl ou JavaScript, use sempre um editor de textos que tenha a capacidade de salvar em formato texto (não use editores como o Word, por exemplo).

O código-fonte

O código-fonte é o programa propriamente dito, escrito numa linguagem de programação compreendida pelo programador, que tem as estruturas, expressões e componentes que descrevemos na seção anterior. Dependendo de como o programa é executado, o código-fonte pode ou não ser visível pelo usuário do programa.

O programa-objeto

O programa-objeto é o arquivo que contém o programa pronto para a execução. Se o programa foi escrito em uma linguagem interpretada, o programa-objeto e o programa-fonte são a mesma coisa. Será preciso que o

interpretador leia o código e o interprete durante a execução. Se o programa foi escrito em uma linguagem que exige uma etapa de compilação, o programa-objeto provavelmente contém linguagem de máquina, e poderá ser executado diretamente pelo sistema operacional sem precisar de interpretador. A maior parte dos softwares são distribuídos como programas-objeto compilados (você não tem acesso ao seu código-fonte).

A execução

A execução do programa, seja ele compilado ou interpretado, faz com que ele seja transferido para a memória temporária do computador e geralmente causa a criação de um processo no sistema operacional. É somente neste estado que o programa pode realizar as tarefas programadas.