São Paulo Perl Mongers: Introdução ao Perl (Perl 101)

A visão deste curso é dar base para o programador que deseja ter uma visão geral sobre a linguagem Perl. Mostrando as estruturas que a compõe, bem como sua história.

1. Introdução

Flexível, portável, versátil, e disponível para todos, Perl cresceu de um substituto simples para Shell Scripts, a uma linguagem completa e de propósito geral.

Esta popular, rica em recursos está ganhando ainda mais terreno enquanto adquire mais "features" a cada nova versão ou extensão. Assim como fez o Moose [1], que trouxe uma grande melhora em Orientação a Objetos para o Perl 5, inspirado no Perl 6.

Uma característica particular, porem muito poderosa, do Perl é a sua implementação de bibliotecas com módulos, o qual a fez uma linguagem genuinamente extensível

Com a sua clareza, estrutura lógica e abordagem prática, este curso é ideal para guiar e acompanhar o leitor ao mundo Perl.

[1] http://www.iinteractive.com/moose/

1.1 Conceitos de Programação

Para podermos entender quais as bases do Perl, devemos primeiro relembrar alguns conceitos fundamentais de programação:

1.1.1 A Linguagem dinâmica

É uma classe de linguagens, com um alto nível de abstração [1], no qual em tempo de execução realizam tarefas que outras linguagens (chamadas "*estáticas*") fariam em tempo de *compilação*. Neste comportamento dinâmico, o software poderia incluir/importar novas partes, estender objetos e definições, tudo isso é provido de maneira direta e simples.

[1] Níveis de abstração é uma classificação, relativa ao desenvolvimento de software e a uma linguagem de programação;

Porem, é necessário atenção ao termo "linguagem dinâmica" é diferente de "tipagem dinâmica". O primeiro refere-se ao dinamismo em tempo de execução; o segundo, refere-se a validação dos tipos.

Leia mais:
http://en.wikipedia.org/wiki/Dynamic_language

1.1.2 Tipagem dinâmica

Uma linguagem de programação tem tipos dinâmicos (por exemplo: int, char, scalar, array, hash), quando a maioria das verificações de tipos é feita em tempo de execução. Ou seja, os tipos, propriamente ditos, são associados aos valores e não as variáveis.

Exemplos de linguagens com tipos dinâmicos: *Groovy, JavaScript, Lisp, Lua, Objective-C, PHP, Perl, Prolog, Python, Ruby, Smalltalk, Tcl*.

Tipos dinâmicos vão acusar seus erros em tempo de execução; por exemplo, um valor pode ser um tipo inesperado ou inexistente, ou uma operação errônea (sem sentido) pode ser atribuída a um tipo que não a suporta. Deve ficar evidente que este tipo de situação é *sempre* causada pelo programador, e em um cenário assim, pode ser um pouco mais trabalhoso encontrar o causador. É uma situação esperada.

Se compararmos as linguagens de Tipagem Dinâmica com as de Tipagem Estática, elas fazem muito menos verificações nos fontes em tempo de compilação, apenas é certificado de que o programa está sintaticamente correto. Verificações em tempo de execução podem potencialmente ser mais sofisticados, porque são utilizadas as informações geradas pela execução do software, em contra-partida, estas verificações são executadas todas as vezes que o software for chamado.

O desenvolvimento em linguagens de Tipos Dinâmicos tem grande auxílio por boas práticas de programação, como por exemplo Testes Unitários (Unit Testing). "Testar" é a chave em desenvolvimento profissional de software, e é particularmente importante nestes casos. Na prática, os testes asseguram a operação correta do seu software de forma muito mais ampla se comparado às estáticas (ou seja, através da verificação de tipos feita pelo compilador).

Saiba mais:
http://en.wikipedia.org/wiki/Dynamic_typing#Dynamic_typing

1.1.3 Sobre os "tempos"

Nos tópicos acima nós citamos os tempos de "Compilação" e de "Execução", é necessário relembrar o que é feito em cada um deles:

- **Tempo de Compilação (Compile-Time) [1]:** Refere-se as operações feitas por um compilador, as necessidades de uma linguagem de programação que devem ser equivalentes as descritas no código fonte para que este processo seja executado com sucesso. As operações executadas em "Compile-Time" geralmente incluem análise sintática, vários tipos de análises semânticas, etc;

- **Tempo de Execução (Run-Time) [2]:** Refere-se ao período enquanto um software está sendo executado, do começo ao seu fim. Também pode indicar a duração deste período. O termo é geralmente empregado para contraste de outras fases do desenvolvimento e uso de um software.

Estes são os dois mais importantes, porem nós ainda temos o "Link Time" [3] que consiste em:

Refere-se as operações feitas por uma ferramenta que fará a união entre os elementos que são necessários para que um software seja entregue com sucesso após o Tempo de

Compilação. As operações performadas em "Link Time" incluem a reunião de endereços externos ao software, vários tipo de validações para referências cruzadas entre módulos, entre outros.

[1] http://en.wikipedia.org/wiki/Compile_time
[2] http://en.wikipedia.org/wiki/Run_time_(computing)
[3] http://en.wikipedia.org/wiki/Link_time

1.2 Para Acompanhar este Curso

Este curso é, antes de mais nada, uma introdução prática ao mundo de desenvolvimento Perl, portanto, os seguintes itens são indispensáveis:

- Editor de Texto competente [1];
- Browser;
- Interpretador Perl instalado;
- Acesso a internet;
- Acesso a Console;

Também é desejável, ter noções de inglês para leitura técnica. Você vai perceber ao longo deste material que os melhores documentos estão em inglês, e que em muitos casos a tradução para "pt_BR", muitas vezes, deixa a desejar.

2. História da Linguagem

2.1 Perl 4 e 5

Perl é uma **linguagem em evolução**, frequentemente é atualizada com suporte para novas features. Apesar disso, ela ainda é uma linguagem fácil de aprender e não perdeu suas bases concisas, evoluindo de uma simples ferramenta de construir scripts para uma completa, no sentido exato da palavra, construtora de aplicações orientada a objetos [1].

Perl evoluiu de forma paralela, **próxima a Internet**. Ganhou popularidade nos seus primeiros dias como uma linguagem para escrever rapidamente scripts utilitários. Isso era graças ao poderoso processador de textos e familiaridade para programadores que faziam uso do *Sed* [2] e *Awk* [3], nestas duas o Perl era parcialmente inspirado. Perl estava obviamente relacionada ao *C*, mas isso também era uma característica derivada de *Lisp*. O resultado ganhou popularidade como uma linguagem para escrever *CGIs* (server-side) scripts para Web Servers, novamente, porque as habilidades de lidar com texto e porque era facilmente manipulável e expressiva. Tudo isso, durante o que foi chamado de *Perl 4*.

A *quinta* versão da linguagem a levou para um novo patamar, principalmente através dos recursos de **Orientação a Objetos** [1]. Seguindo a sua filosofia, Perl preocupou-se primeiramente em ter o recurso funcionando, ao invés de usar este tempo discutindo a parte ideológica, pois na época haviam grupos de usuários que se colocavam contra a mudança. A *versão 5* marcou a história do Perl, transformando-a em uma linguagem para escrever aplicações grandes e sérias, ao invés de scripts simples.

A *versão 5.005* começou a trazer suporte a threads [2], porem, somente dentro do interpretador. Isso trouxe a linguagem para o Windows e outros sistemas operacionais que não tinham suporte a processos filhos [3], através de uma emulação de uma

chamada "fork" (system-call) [4]. Uma mudança importante para suportar estas plataformas.

Na versão 5.6, Perl revisou a sua política de versionamento, para ser mais específica. Em particular, ela adotou o mesmo sistema utilizado pelo GNU/Linux [5] para versionar. Também nesta mesma versão, Perl introduziu melhorias, sendo as principais: um suporte melhor aos idiomas (comandos, sintaxe) Unix sobre o Windows [6] e suporte inicial ao Unicode [7].

A partir da 5.6, suporte experimental às threads ao nível do usuário, ou seja, para escrever Perl com suporte a esta tecnologia.

Perl 5.8 trouxe melhorias para a implementação de threads do interpretador. Também trouxe suporte completo para Unicode, suporte para PerlIO [8] e camadas para "filehandle", hashes restritos como substituto para os pseudo-hashes [10], melhor gerenciamento dos sinais [11]; mais tarde, melhorou o suporte para Windows e uma suite de testes com suporte a regressão.

[1] http://en.wikipedia.org/wiki/Object-oriented_programming
[2] http://en.wikipedia.org/wiki/Thread_(computer_science)
[3] http://en.wikipedia.org/wiki/Fork_%28operating_system%29
[4] http://en.wikipedia.org/wiki/System_call
[5] http://en.wikipedia.org/wiki/Linux_kernel#Version_numbering
[6] http://win32.perl.org/wiki/index.php?title=Main_Page
[7] http://en.wikipedia.org/wiki/Unicode
[8] http://perldoc.perl.org/PerlIO.html
[9] http://perldoc.perl.org/FileHandle.html
[10] http://perldesignpatterns.com/?PseudoHash
[11] http://en.wikipedia.org/wiki/Signal_%28computing%29

2.2 Futuro: Perl 6

O futuro é o Perl 6 [1]. O interpretador foi inteiro reescrito, totalmente Orientado a Objetos, desde suas raízes, bem como a adição da máquina virtual Parrot [2], local onde o Perl será executado na versão 6. Enquanto o Perl 6 é radicalmente diferente em seu interior, ele será compatível com a maioria dos fontes escritos para a versão 5.

Um destaque é o projeto Ponie [3], no qual estão transferindo o suporte ao Perl 5 para a Parrot, o planejamento é para que seja compatível com o Perl 5.10, ao menos.

[1] http://dev.perl.org/Perl6.
[2] http://www.parrotcode.org
[3] http://www.poniecode.org

2.3 Larry Wall

Perl foi originalmente desenvolvida por Larry Wall [1], um linguista [2] trabalhando como Sysadmin [3] na NASA [4]. E, no ano de 1987, criou uma linguagem -- *de propósito geral, com foco em scripts Unix* -- que transformou o processamento de texto em uma tarefa **muito** mais fácil. Desde então, ele continua a trabalhar no seu projeto e tornou-se um programador mundialmente conhecido. Atualmente, ele está trabalhando no Perl 6.

[1] http://en.wikipedia.org/wiki/Larry_Wall
[2] http://en.wikipedia.org/wiki/Linguistics

[3] http://en.wikipedia.org/wiki/Systems_administrator
[4] http://en.wikipedia.org/wiki/NASA

3. Quem usa Perl?

A resposta não pode ser mais simples: praticamente todos nós, direta ou indiretamente.

Perl foi fundamental para o crescimento e estruturação da internet como nós conhecemos hoje, visto que naquela época eram escassas as ferramentas para automatizar tarefas com a qualidade e a versatilidade de Perl. Outro fator importante é que ela nasceu em ambientes unix-like e nunca perdeu esta intimidade, e por suas qualidades, tornou-se a primeira escolha de quase todos os SysAdmins.

Se você imaginar neste momento, qual o caminho que os pacotes percorrem da sua estação até ter acesso a internet, eu posso afirmar -- com certeza -- de que a sua conexão foi gerenciada ou auxiliada de alguma forma por uma ferramenta escrita em Perl. Em todos os sistemas operacionais unix-like existem scripts Perl para as mais diversas tarefas dentro do sistema operacional.

Com as informações acima, não é difícil imaginar todas as grande corporações, provedores, hostings, startups, que fazem uso do Perl.

Assim como Hassan Schroeder, o primeiro webmaster da Sun afirmou: "Perl is the duct tape of the Internet.".

Para saber mais:
http://www.oreillynet.com/pub/a/oreilly/perl/news/importance_0498.html

4. A Cultura Perl

Neste capítulo gostaria de deixar o Sr. **Larry Wall** dar as suas próprias opiniões a respeito:

Seriously though, if there iss a germ of an important idea in Perl Culture, it's this: that too much control is just as deadly as too little control. We need control and we need chaos. We need order, and disorder. Simplicity, and complexity. Carefulness, and recklessness. Poise, and panic. Science, and art. ***(Larry Wall)***

We like to see that "There's more than one way to do it," because it breaks us out of our cultural preconceptions. ***(Larry Wall)***

But the right subset of Perl does survive. The dinosaurs will die off, and the monkeys will end up writing the poetry. Some of that poetry is even written in Perl, a language that can definitely be classified as "quirky". ***(Larry Wall)***

We don't expect a native German speaker to use the same subset of English as a native Mandarin speaker. Similarly, we don't look down on people for using subsets of Perl. ***(Larry Wall)***

There are certainly enough of them. You can write Perl programs that resemble Sed, or Awk, or C, or Lisp, or Python. This is Officially Okay in Perl culture. By way of contrast, try writing in the C subset of C++ and they'll make a laughingstock of you. (***Larry Wall)***
But seriously, many computer scientists have fallen into the trap of trying to define languages like George Orwell's Newspeak, in which it is impossible to think bad thoughts. What they end up doing is killing the creativity of programming. (***Larry Wall)***

Além de mostrar as frases do célebre autor, quero ressaltar a importância da cultura em si. O intuito é ajudar as pessoas a entenderem o motivo daquele "pseudo-dogma" para que torne-se cultural, para que as pessoas realmente acreditem nestas preposições, é uma forma muito mais elegante e eficiente de influência um uso correto da linguagem, sem sacrificar a criatividade para isso.

É essencial pensar nestas colocações antes de continuar.

Para saber mais:
http://www.wall.org/~larry/keynote/keynote.html

4.1 Comunidade e Encontros Sociais

Depois desta dose "cultural" é natural concluir que os programadores Perl sejam unidos, afinal, eles dividem as mesmas crenças. Estes são os lugares:

4.1.1 Perl Mongers

O programador envolvido com a comunidade Perl e que acredita na cultura da linguagem é normalmente chamado de "*Perl Monger*".

Em 1998 o primeiro grupo de programadores Perl foi criado por Brian D Foy [1] em Nova York, durante a primeira O'Reilly Perl Conference. A primeira idéia para o nome deste grupo era: `/New York Perl M((olu)ngerslaniacs)*/`. Porem o "Monger" acabou ficando mais popular.

Repare que a palavra "*Monger*" (ou Monge, em português) refere-se a fortes crenças na cultura da linguagem.

Para saber mais:
http://en.wikipedia.org/wiki/Perl_Monger

[1] http://en.wikipedia.org/wiki/Brian_D_Foy

4.1.2 Sites:

Muitas são as referências internacionais:

- http://perl.org: O principal endereço, com links para download da linguagem, documentação, etc;

- http://www.perlmonks.com: O principal forum da comunidade, onde você consegue encontrar ajuda para qualquer problema relacionado a Perl. Tem um excelente histórico, poucas perguntas ainda não foram respondidas. Também tem exemplos de código, dicas de programação, perguntas e repostas, bibliotecas, entre outros;

- http://planet.perl.org, http://ironman.enlightenedperl.org, http://perlsphere.net: Junção dos principais blogs sobre Perl;

- http://use.perl.org: Novidades relativas a communidade, seu funcionamento é inspirado em uma revista ou jornal;

4.1.3 Comunidades Locais

Na cultura da linguagem também nasceu o costume de reuniões em grupos menores e unidos pela localização geográfica, geralmente. O registro dos grupos pode ser encontrado no http://www.pm.org/. O grupo responsável por este documento é o São Paulo Perl Mongers (http://sao-paulo.pm.org/), formado em 2000, com mais de duzentos membros ativos.

4.1.4 Lista de discussão e IRCs

Outro costume da comunidade é manter-se em contato através de listas de discussão [1] e também dos canais IRC [2]. Oferecendo oportunidades para os participantes interagirem em problemas relacionados ao desenvolvimento Perl, boas práticas e troca de experiências.

Aproveitamos este material para convidar a todos para participar de nossa lista e canal IRC [2]:

- **Lista de discussão:** http://mail.pm.org/mailman/listinfo/saopaulo-pm
- **IRC:** irc.perl.org, canal "#sao-paulo.pm"

[1] http://en.wikipedia.org/wiki/Electronic_mailing_list
[2] http://en.wikipedia.org/wiki/IRC

4.1.5 Encontros Sociais

Este é outro item da nossa cultura. Encontros periódicos, para um *happy-hour* no qual o assunto principal é Perl, extremamente divertido e produtivo. Você saberá quando acontecerá o próximo acompanhando a nossa lista de discussão (São Paulo Perl Mongers).

5. Perl: Plataformas Suportadas

Agora que você já tem as bases sobre a cultura Perl e todo o embasamento teórico necessário, vamos nos focar a utilização desta linguagem. O primeiro passo é assegurar se a plataforma de sua preferência terá suporte.

- **Unix-like:** AIX, BSD/OS, Darwin, dgux, DYNIX/ptx, **FreeBSD**, Haiku, **Linux** (ARM), **Linux** (i386), **Linux** (i586), **Linux** (PPC), HP-UX, IRIX, **Mac OS X**, MachTen PPC, NeXT 3, NeXT 4, **OpenBSD**, OSF1, reliantunix-n, SCO_SV, SINIX-N, sn4609, sn6521, sn9617, **SunOS** (sun4-solaris) , **SunOS** (i86pc-solaris) e SunOS4;

- **MSDOS-like:** MS-DOS, PC-DOS, OS/2, Windows 3.1, Windows 95, Windows 98, Windows ME, Windows NT (Alpha), Windows NT (PPC), **Windows 2000**, **Windows XP**, **Windows 2003**, Windows CE, Windows Vista, **Windows 7**, **Windows 2008** e Cygwin;

- **Outras:** Amiga DOS, BeOS e MPE/iX;

Lembrando que o código escrito em Perl é **portável** dentre as plataformas, portanto, com alguns cuidados é plenamente possível escrever códigos que funcionam nas 45 plataformas citadas aqui.

É importante lembrar que a expansão do Perl em outras plataformas é constante, então, a nossa lista só tende a crescer.

Para saber mais:
http://perldoc.perl.org/perlport.html#PLATFORMS

6. Instalação do Perl

Para continuar com este curso, nós agora devemos saber instalar o Perl. Aqui cobriremos os sistemas operacionais mais comumente utilizado, tanto no meio acadêmico quanto corporativo.

6.1 Unix-like

6.1.1 FreeBSD

Ao fazer isso no FreeBSD nós temos duas opções:

- **Ports:** Cada “port” contêm todas as instruções necessárias para fazer os fontes originais do projeto compilarem e serem executados no FreeBSD. Ao instalar um "Port" o Makefile [1] busca automaticamente os fontes da aplicação desejada, tanto em disco local quanto remoto (HTTP/FTP), em seguida, os fontes são des-compactados em um diretório. O próximo passo é aplicar os patches necessários (a maioria deles visa compatibilidade com o FreeBSD), após, os fontes já estão prontos para compilar. A seguir, esta apostila vai demonstrar os poucos comandos necessários para cumprir todos os procedimentos listados acima -- portanto, é presumido que o leitor esteja com acesso ao terminal (local, virtual ou remoto). Segue:

```
# uname -a
FreeBSD bsd.domain.com.br 8.0-RELEASE-p2 FreeBSD 8.0-RELEASE-p2
#0:
Tue Jan  5 21:11:58 UTC 2010 root@amd64-builder.daemonology.net:/
usr/obj/usr/src/sys/GENERIC amd64

# cd /usr/ports/lang/perl5.10 && make install clean
```

Nota: O primeiro comando é apenas uma demonstração de qual a versão empregada do sistema; o segundo, é praticamente todo o trabalho necessário, com exceção de responder algumas simples perguntas durante o processo.

Para saber mais:

http://www.freebsd.org/doc/en/books/developers-handbook/tools-make.html

- **Packages:** Os packages consistem em instalar um pacote préviamente compilados. Alguns cluster são empregados pelo time de desenvolvimento do FreeBSD para compilar o máximo de pacotes possíveis estas plataformas: alpha, amd64, ARM, i386, ia64, MIPS, PC98, PPC, Sparc64, Sun4v e Xbox. Para isso, basta:

```
# pkg_add -rv perl5.10
```

Consiste apenas em dois comandos, sendo um deles o nome do pacote que nós desejamos. Esta forma de instalação cobre também todas as dependências necessárias, ou seja, para instalar o Perl você vai precisar instalar suas dependências, e o FreeBSD Package System fará isso por você.

6.1.2 GNU/Linux

O projeto GNU/Linux está disposto de forma diferente, se compararmo-os ao FreeBSD. Pois o Linux é representado enquanto sistema operacional porque grupos de usuários fazem versões customizadas e mais completas, estas são chamadas distribuições. Abaixo, mostraremos exemplos nos dois principais tipos de distribuições atualmente.

Para Linux, não será demostrado uma forma de compilar os fontes, vamos nos focar em usar os mecanismos da forma como a distribuição recomenda.

6.1.2.1 Debian's Like

É o padrão da distribuição ter o interpretador Perl instalado, sendo que inúmeros de seus scripts básicos são feitos nesta linguagem.

A forma como o Debian mantêm os seus pacotes é de forma binária, a exemplo da maioria das outras distribuições Linux. Isso denota o fato de nós instalarmos arquivos pré-compilados no seu SO. Estes são feitos com auxílio de inúmeros colaboradores do projeto, um trabalho meticuloso para deixar muitos softwares a disposição de seus usuários.

Se houver alguma customização do SO, ou qualquer outra situação em que o Perl não esteja instalado, poremos seguir os procedimentos para reverter esta situação:

```
# apt-get update && apt-get install perl
```

Novamente, o exemplo de instalação assume que você está com a sua base de pacotes atualizada, vide o primeiro comando. O segundo toma conta de tudo mais o que é necessário, download, desempacotar, instalar e registrar perante a base do SO.

Em um futuro próximo o projeto Debian vai usar mais Perl. O substituto do APT é o projeto denominado Cupt [1], o qual é escrito em Perl.

[1] http://wiki.debian.org/Cupt

6.1.2.2 Red Hat's Like

O projeto Red Hat é a distribuição mais antiga de Linux, nela o recomendado é fazer uso da ferramenta RPM (Red Hat Package Manager) [1]. Da mesma forma como no Debian, nós vamos fazer uso de uma versão previamente customizada (em alguns casos), compilada e empacotada pelo time de desenvolvimento da Red Hat, com auxílio de inúmeros colaboradores do projeto.

Nesta tipo de distribuição o normal é que o interpretador Perl venha instalado por padrão, já que esta faz parte da sua construção. Se em casos específicos o interpretador Perl não esteja presente, execute os comandos abaixo:

```
# yum update && yum install perl
```

O primeiro comando é apenas um gancho para recomendar que a base de pacotes esteja sempre atualizada. O segundo dá inicio a cadeia de ações necessárias para baixar, interpretar e fazer a instalação dos componentes, bem como suas dependências, na instalação local.

[1] http://en.wikipedia.org/wiki/RPM_Package_Manager

6.2 Windows

Este curso está focado em ambientes unix-like, acima foi demostrando a instalação da linguagem nos seus "sabores" mais comuns. Porem, não vamos deixar o leitor e usuário dos ambientes baseados em Microsoft Windows desamparados, seguem as referências técnicas para o orientar:

- http://win32.perl.org/wiki/index.php?title=Main_Page;
- http://www.activestate.com/activeperl/;
- http://www.perl.com/download.csp#win32;

7. Documentação do Perl

O projeto em torno da linguagem Perl nunca esqueceu de manter e aumentar a sua documentação. Hoje, podemos afimar que a documentação do Perl em si, e seus módulos são extremamente completos, uma base sólida para o estudente, usuário da linguagem. Para uma linguagem ser produtiva, o acesso a documentação deve ser rápido e eficiente, afinal, nós programadores precisamos consultar constantemente as propriedades de um módulo, exemplos de utilização, entre outros.

Enquanto falamos de Perl, o projeto que cuida da sua documentação é o Perldoc, este não passa de uma ferramenta para procurar e interpretar os arquivos tipo "pod".

- **Pod** ("*Plain Old Documentation*") [1]: É uma linguagem de marcação ("Markup Language" [2]) simples e fácil de usar, com o propósito de documentar softwares escritos em Perl e seus módulos;

Esta é a forma para ler e buscar informações na documentação, para isso existem duas formas principais, explicadas a seguir.

[1] http://perldoc.perl.org/perlpod.html

[2] http://en.wikipedia.org/wiki/Markup_language

7.1 Perldoc no Site

O site do Perldoc (http://perldoc.perl.org/) é relativamente novo, com grandes melhorias na navegação, busca, indexes, sintaxe, entre outros. No site você também consegue fazer download das páginas em *PDF* ou *HTML*. Para o aluno, iniciante, o mais indicado é começar a fazer a leitura da documentação através do site, visto que este é uma interface mais intuitiva e amigável.

7.2 Perldoc no Terminal

Outra forma de acessar o conteúdo do Perldoc é através da linha de comando. Exemplo:

```
$ perldoc perldoc
```

Observe, nós estamos passando um argumento para o comando, este indica qual documentação nós queremos ler. Normalmente, o Perldoc organiza a documentação que faz parte do "core" da linguagem com o sufixo "perl".

Exemplos: *perl, perl5004delta, perl5005delta, perl561delta, perl56delta, perl570delta, perl571delta, perl572delta, perl573delta, perl581delta, perl582delta, perl583delta, perl584delta, perl585delta, perl586delta, perl587delta, perl588delta, perl589delta, perl58delta, perlaix, perlamiga, perlapi, perlapio, perlapollo, perlartistic, perlbeos, perlbook, perlboot, perlbot, perlbs2000, perlcall, perlce, perlcheat, perlclib, perlcn, perlcommunity, perlcompile, perlcygwin, perldata, perldbmfilter, perldebguts, perldebtut, perldebug, perldelta, perldgux, perldiag, perldoc, perldos, perldsc, perlebcdic, perlembed, perlepoc, perlfaq, perlfaq1, perlfaq2, perlfaq3, perlfaq4, perlfaq5, perlfaq6, perlfaq7, perlfaq8, perlfaq9, perlfilter, perlfork, perlform, perlfreebsd, perlfunc, perlglossary, perlgpl, perlguts, perlhack, perlhist, perlhpux, perlhurd, perlintern, perlintro, perliol, perlipc, perlirix, perljp, perlko, perllexwarn, perllinux, perllocale, perllol, perlmachten, perlmacos, perlmacosx, perlmint, perlmod, perlmodinstall, perlmodlib, perlmodstyle, perlmpeix, perlnetware, perlnewmod, perlnumber, perlobj, perlop, perlopenbsd, perlopentut, perlos2, perlos390, perlos400, perlothrtut, perlpacktut, perlplan9, perlpod, perlpodspec, perlport, perlqnx, perlre, perlrebackslash, perlrecharclass, perlref, perlreftut, perlreguts, perlrequick, perlreref, perlretut, perlriscos, perlrun, perlsec, perlsolaris, perlstyle, perlsub, perlsyn, perlthrtut, perltie, perltoc, perltodo, perltooc, perltoot, perltrap, perltru64, perltw, perlunicode, perlunifaq, perluniintro, perlunitut, perlutil, perluts, perlvar, perlvmesa, perlvms, perlvos, perlwin32, perlxs e perlxstut;*

A quantidade de exemplos não tem o intuito de deixar o leitor iniciante confuso, apenas de mostrar a enorme quantidade de documentação que a linguagem tem. Este é um pequeno exemplo, a saber.

7.3 Livros

A quantidade de livros sobre a linguagem é quase tão grande quanto o Perldoc. Para o leitor que prefere uma interação com um livro tradicional, não vão faltar opções: http://books.perl.org/books, somente nesta página são 266 livros. Para o iniciante, seguem os títulos mais recomendados:

- Beginning Perl Web Development: From Novice to Professional (Suehring);
- Beginning Perl, Second Edition (James Lee);
- Higher-Order Perl (Dominus);
- Learning Perl Objects, References & Modules (Schwartz, Phoenix);

- Learning Perl on Win32 Systems (Olsen, Schwartz, Christiansen);
- Learning Perl, 4th ed. (Schwartz, Phoenix, Brian D Foy);
- Mastering Perl (Brian D Foy);
- Mastering Regular Expressions, Second Edition (Friedl);
- Modern Perl Programming [1];
- Network Programming with Perl (Stein);
- Object Oriented Perl (Conway);
- Perl Best Practices (Conway);
- Perl Cookbook, 2nd ed. (Torkington, Christiansen);
- Writing Perl Modules for CPAN (Tregar);

[1] http://www.modernperlbooks.com/mt/index.html

8. Executando os Programas em Perl

Agora nós estamos partindo para a parte mais voltada a prática, deste material. O primeiro passo é saber como os programas em Perl serão executados. Basicamente, existem duas formas para isso:

8.1 Interpretador

A forma mais simples, é chamar o interpretador diretamente via linha de comando:

```
$ perl hello_world.pl
```

Veja que o parâmetro para o interpretador é o caminho de um arquivo contendo código válido Perl, neste caso acima, o arquivo está no mesmo diretório.

Outra forma de usar o interpretador diretamente, é:

```
$ perl -e 'print "Hello Perl World!";'
```

Esta segunda forma nos dá muita praticidade para o dia-a-dia, pois pode funcionar como ferramenta para linha de comando, com todo o poder de Perl, exemplo:

```
$ cat /etc/passwd                                          \
     | perl -ne 'print $1, "\n" if ( /^(\w+):.*?$/ );'    \
     | head
root
toor
daemon
operator
bin
tty
kmem
games
news
man
```

8.2 Script Executável

O outro jeito de executar um software escrito em Perl é, gravar todo ele em um arquivo e o transformar em um executável. Veja os exemplos:

```
$ echo '#!/usr/bin/env perl' > /var/tmp/test.pl
$ echo 'print "Hello, Perl World!";' >> /var/tmp/test.pl
```

Repare que na primeira linha do script é uma indicação de quem vai interpretar os dados que estão nas linhas seguintes, o nome desta entrada é Shebang [1]. Nós poderíamos informar o binário do Perl diretamente, por exemplo: "/usr/bin/perl" ou "/usr/local/bin/perl" (dependendo de onde ele foi instalado), no entanto, com o auxílio do "env" (/usr/bin) [2] nós podemos simplificar.

Agora que nós já vemos um script válido, vamos dar permissão de execução:

```
$ chmod +x /var/tmp/test.pl
```

E testar:

```
$ /var/tmp/test.pl
Hello, Perl World!
```

[1] http://en.wikipedia.org/wiki/Shebang_%28Unix%29
[2] http://linux.die.net/man/1/env

9. Sintaxe

Um script Perl consiste em uma ou mais declarações. Estas são escritas simplesmente em um script de uma forma muito direta, não existe necessidade de ter um método “main” ou qualquer coisa neste sentido.

- Todas as declarações são terminadas por um ponto-e-virgula (“;”);

```
print “Hello, Perl World!\n” if ( $perl );
```

- Comentários começam com o simbolo “#”, e toda a linha será tratada como um comentário;
- Espaços em branco não são relevantes;

```
print
     “Hello, Perl World!\n”
     if ( $perl );
```

- Espaços só influem no software quando estão dentro de strings, ou seja entre aspas;
- Parênteses só são necessários para melhorar expressividade do software, ou seja, quando misturamos contextos, precisamos deles para nos ajudar;
- Variáveis são indicas por um símbolo antes do seu nome (“$”, “%”, “@”);

- Funções são chamadas através de um nome seguido de “()” ou começando com o “&”, esta segunda é a forma antiga de se chamar um método, porem, ainda tem utilização em contextos particulares;
- Referências são feitas com a variável, ou método, precedido de “\”;

9.1 Operadores

A seguir, falaremos sobre os principais operadores da linguagem, ou seja, os tipos básicos para lidar com as rotinas, em Perl.

9.1.1 Operadores Numéricos

Para expressar um número inteiro real em Perl você não deve utilizar aspas, pois com elas o conteúdo reservado será considerado uma “string”, portanto, operadores numéricos poderão ter ações diferentes do que lhe é esperado.

- Aritméticos:

```
my $sum = 1 + 2; # somatória
my $sub = 2 - 1; # subtração
my $mul = 2 * 2; # multiplicação
my $div = 2 / 1; # divisão
```

- Comparação entre números:

```
print “equals” if ( 1 == 1 );
print “inequality” if ( 1 != 2 );
print “greater than” if ( 2 > 1 );
print “less than” if ( 1 < 2 );
print “greater or equal” if ( 2 >= 2 );
print “less or equal” if ( 2 <= 2 );
```

- Acumuladores:

```
my $i += 10; # increase 10
my $i -= 10; # remove 10
```

9.1.2 Operadores para Strings

Perl é uma linguagem extremamente completa quando o assunto é tratamento de string, vamos analisar alguns deles:

- Comparação entre strings;

```
print “equals” if ( “test” eq “test );
print “inequality” if ( “test” ne “fail” );
print “less than” if ( “test” lt “lots of chars” );
print “greater than” if ( “test” gt “1” );
print “greater or equal” if ( “test” ge “test” );
```

```
print “less or equal” if ( “test” le “tset” );
```

- Concatenando strings:

```
print “test”, “ “, “\n”;
print “test” . “ “ .  “\n”;
```

- Acumulando uma string em um scalar;

```
my $text = “I’m a Perl ”;
$test .= “Programmer\n”;
```

9.1.3 Operadores Lógicos

Uma parte importante de uma linguagem de programação, são os seus operadores que aplicam a lógica Booleana [1]. Observe os exemplos abaixo:

```
my $test = 1;
my $other = 1;
print “and” if ( $test and $other ); # true
print “and” if ( $test && $other ); # true

my $test = undef;
my $other = 1;
print “or” if ( $test or $other); # true
print “or” if ( $test || $other); # true

my $not = undef;
print “not” if ( !$not );
print “not” if ( not $not );
```

[1] http://en.wikipedia.org/wiki/Boolean_logic

10. Estrutura da Linguagem

Após fazer toda a apresentação sobre o que envolve a linguagem, incluindo sua história, documentação e conceitos básicos de programação, chegou a hora de colocarmos os nossos conhecimentos em prática com Perl.

Este capítulo é dedicado a apresentar as estruturas da linguagem, e dar exemplos para o seu uso no dia-a-dia.

10.1 Scalar

Um scalar é o dado mais simples que o Perl manipula. Podendo ser tanto um número como uma string de caracteres. No entanto, você pode imaginar um número e uma string como estruturas muito diferentes, porem, o Perl as usa de forma muito parecida, portanto são descritas juntas.

Exemplos:

```
my $test = q{Hello, Perl World!};    # Hello, Perl World!
my $test = "I'm a String!";          # I'm a String!
my $test = 123456;                   # 123456
my $test = "123456";                 # 123456
my $test = "123" . "456";            # 123456
my $test = 123 + 456;                # 579
```

Um valor scalar também pode atuar sobre operadores, como mostrado nos exemplos acima, repare que ao final existe um comentário com o valor esperado na variável "`$test`". Também, um scalar pode ser guardado dentro do outro, e podem ser guardados dentro de devices [1] ou arquivos texto.

Saiba mais:
http://perldoc.perl.org/functions/scalar.html

[1] http://en.wikipedia.org/wiki/Device_file

10.1.1 Referências

Ao mesmo temo que o scalar é o tipo mais simples, ele pode guardar referências para estruturas mais complexas, no Perl nós chamamos isso de referência.

Exemplos:

```
my $reference = \%complex_hash;      # HASH(0x600e78)
my $reference = \@complex_array;     # ARRAY(0x600f58)
```

Agora um exemplo um pouco mais avançado, mostrando que até métodos podem ser passados como referencia:

```
#!/usr/bin/env perl

use strict;
use warnings;

sub method { print @_, "\n"; }

my $reference = \&method;

&{$reference}("string");

__END__
```

O exemplo acima também mostra como Perl é uma linguagem dinâmica no seu interior, e demostra também a forma como o interpretador usa para chamar determinadas rotinas.

10.2 Arrays

Array é simplesmente uma lista ordenada de dados, ou seja, um tipo espacial de variável que suporta uma lista no Perl. Cada elemento do array é um scalar em separado, com um valor scalar.

Arrays em Perl podem ter qualquer número de elementos. O menor array é o que tem zero elementos, enquanto o maior, pode preencher toda a memória disponível. Mantendo uma filosofia do Perl de não manter limites desnecessários.

Exemplos:

```
my @week = ( "Mon", "Tue", "Web", "Thu", "Fry" );
my @week = ( 1,     2,     3,     4,     5 );
my @week = ( [ 1, 2, 3 ], [ 4, 5, 6 ] );
```

10.2.1 Indexes nos Arrays

Cada elemento em um array Perl tem uma posição, no qual, a primeira é sempre zero. Também possuem tamanho dinâmico, sendo possível incluir um elemento ao final, a qualquer momento.

Para saber o tamanho de um array, basta incluir um "#" logo após a representação scalar de um array:

```
my @week = ( "Mon", "Tue", "Web", "Thu", "Fry" );
print $#week;    # 4
```

O aluno, neste momento pode estar um pouco confuso a respeito da representação scalar de um array. Esta funciona como uma referência, explicada há poucos capítulos atrás, para o array.

Uma outra forma de receber a representação scalar, é incluir o método scalar [1]:

```
my @week = ( "Mon", "Tue", "Web", "Thu", "Fry" );
print scalar @week;    # 5
```

Porem, neste segundo exemplo existe uma diferença entre os resultados. O causador desta discrepância é que neste último, nós estamos trazendo o array para um contexto scalar e diferentemente do anterior, nós não estamos pedindo o número de elementos de uma lista. No contexto scalar, elementos não tem indexes zero.

Agora que citamos a palavra "index" é necessário explicar como eles funcionam em um array, ou seja, todo elemento tem um número correspondente. Podemos demonstrar isso com mais um exemplo:

```
my @week = ( "Mon", "Tue", "Web", "Thu", "Fry" );
print $week[4];        # Fry
```

[1] http://perldoc.perl.org/functions/scalar.html

10.2.2 Funções para Manipular Listas/Arrays

Por padrão o Perl já nos deixa métodos para manipular as listas, neste capítulo vamos citar os mais importantes e comuns:

- **push [1]:** Este método serve para incluir um elemento em um array, porem, sempre ao final dele, por este motivo, o "push" sempre requer um ou mais argumentos. Exemplo:

```
my @week = ( "Mon", "Tue", "Web", "Thu", "Fry" );
push @week, "Sat";     # Mon, Tue, Web, Thu, Fry, Sat
```

- **pop [2]:** Este método faz exatamente ao inverso do "push", ele retira um elemento do final de um array. Exemplo:

```
my @week = ( "Mon", "Tue", "Web", "Thu", "Fry" );
pop @week;             # Mon, Tue, Web, Thu
```

- **shift [3]:** Basicamente, este método faz o mesmo que o "pop", porem, ele não age ao final do array, e sim no seu início. Exemplo:

```
my @week = ( "Mon", "Tue", "Web", "Thu", "Fry" );
shift @week;           # Tue, Web, Thu, Fry
```

Os métodos apresentados aqui são para funções básicas, no entanto, tenha a certeza de que o leitor irá encontrar muitos módulos para as mais diversas funções com arrays dentro da CPAN.

[1] http://perldoc.perl.org/functions/push.html
[2] http://perldoc.perl.org/functions/pop.html
[3] http://perldoc.perl.org/functions/shift.html

10.3 Hashes

Um Hash (também conhecido como Array Associativo [1]) é uma estrutura parecida com um Array, o qual foi apresentado a pouco, uma coleção de variáveis tipo Scalar, com elementos individuais servindo como índice para esta estrutura. Ao contrário de um Array, os índices não são elementos pequenos, com valores inteiros não-negativos, são valores tipo scalar arbitrariamente escolhidos, estes são chamados de "keys" (ou chaves, em português), e são utilizados para obter o dado de uma determinada parte do Hash.

A forma mais fácil de entender um Hash, é olhar para os exemplos:

```
my %hash = {
     chave  => "valor",
     numero => 1,
     objeto => \$referencia,
};
```

Agora, a forma de obter os dados desta estrutura será assim:

```
$hash{chave} == "valor";
```

O método mais utilizado, é o "keys" [2]. Veja um exemplo:

```
foreach my $k ( keys %hash ) {
     print "chave: $k -> valor: $hash{$k}\n";
}
```

O método "keys" é utilizado para obter somente as chaves de um Hash, fazendo seu retorno em forma de um array.

Porem, existem outros métodos:

- **each [3]:** Quando invocado em um contexto de lista, ele retorna dois elementos consecutivos, no qual o primeiro é a chave e o segundo é o valor, e assim sucessivamente para o Hash. Portanto você pode iterar [7] sobre a estrutura, veja:

```
while ( my ( $key, $value ) = each %hash ) {
     print "chave: $key -> valor: $value\n";
}
```

- **delete [4]:** Se você informar como parâmetro um elemento de um Hash, ou Array, ele vai deletar o elemento especificado, e, como retorno deste método é utilizado o valor antigo da parte que está sendo removida, veja o exemplo:

```
my $scalar = delete $hash{numero}; # 1
```

- **exists [5]:** Informando como parâmetro um elemento de um Hash ou Array, este método nos retornará "true" [8] (verdadeiro). Veja:

```
print "Exists\n" if exists $hash{chave};
```

- **values [6]:** Retorna uma lista somente com os valores que um Hash contêm, este método faz o inverso do "keys":

```
foreach my $value ( values %hash ) {
     print "value: $value\n";
}
```

Observe atentamente o funcionamento do Hash em Perl, porque ele é largamente utilizado para os elementos estruturais da linguagem, por ser extremamente flexível. Outro fator de atenção é que estruturas no padrão Hash são, também, muito utilizadas em outros sistemas computacionais, como caches, bancos de dados não-relacionais, etc.

Para saber mais:
http://www.cs.mcgill.ca/~abatko/computers/programming/perl/howto/hash/
http://www.perl.com/pub/a/2006/11/02/all-about-hashes.html

[1] http://en.wikipedia.org/wiki/Associative_array
[2] http://perldoc.perl.org/functions/keys.html
[3] http://perldoc.perl.org/functions/each.html

[4] http://perldoc.perl.org/functions/delete.html
[5] http://perldoc.perl.org/functions/exists.html
[6] http://perldoc.perl.org/functions/values.html
[7] http://en.wikipedia.org/wiki/Iteration#Computing
[8] http://en.wikipedia.org/wiki/Boolean_algebra_%28logic%29

10.4 Estruturas Complexas

Neste ponto, nós já sabemos quais os principais tipos de dados no Perl, como eles são utilizados e até presenciamos alguns exemplos introdutórios. Porem, o leitor que já tem mais experiência em desenvolvimento de softwares nos pergunta se é possível misturar os vários tipos de estrutura, e a reposta é Sim. Sim, é possível fazer uso de todos os tipos de estruturas nativas do perl de forma misturada, assim como mostramos no exemplo a seguir:

```
my $mix = [
     {    a_hash        => "yes",
          another_value => 1,
     },
     [ 1, 2, 3, 4, 5 ],
];
```

Na estrutura acima temos um scalar recebendo uma referência de um array, e dentro deste um hash e outro array.

Com estas opções Perl é o suficiente para lidar com qualquer tipo de dado ou estrutura que um software pode necessitar.

Para saber mais:
- http://perldoc.perl.org/perldsc.html
- http://docstore.mik.ua/orelly/perl/prog3/ch09_03.htm
- http://www.cbs.dtu.dk/courses/27619/advanced1.html

11. Variáveis Especiais

Na sua concepção Perl incluiu algumas variáveis especiais dentro do interpretador que podem ser manipuladas durante a execução de um software, para alterar o seu comportamento, ou, acrescentar mais produtividade, abaixo as principais variáveis serão mostradas.

11.1 Contexto "local"

Algumas das variáveis abaixo não devem ser utilizadas sem prévio cuidado, afim de não alterar o funcionamento padrão de um software, sem a precaução e conhecimento necessários. Portanto, antes de alterar o valor de uma variável destas certifique-se que será apenas no contexto "local". Exemplo:

```
open my $fh, '<', '/etc/passwd'
     or die $!;
local $/ = " ";
```

```
foreach my $line (<$fh>) {
     print $line, "\n";
}
close($fh);
```

Para saber mais:
http://perldoc.perl.org/functions/local.html

11.2 Relativas ao Filehandle

- $| Se o seu valor estiver diferente de zero, ele forçará o software a fazer o flush do STDOUT (saída padrão) após cada "write" ou "print";
- $= Mostra qual o tamanho de um arquivo, ou seja a quantidade de linhas horizontais;
- $- Número de linhas que resta no arquivo, ou seja, após imprimir algumas delas o Perl mostra quantas ainda faltam;

11.3 Variáveis Globais Especiais

- $_ Esta é a variável especial mais utilizada no Perl. Sua função é receber os elementos de uma estrutura sequencial (leia-se: arrays, hashes, splits, filehandles) e servir como entrada padrão para "pattern-searching". Exemplos:

```
my @array = ( 1, 2, 3, 4, 5 );
foreach ( @array ) {
     print;
}
```

- $. O número da linha que foi acabou de ser extraída (ou lida) de um filehandle.

- $/ Separador padrão utilizado em um filehandle, o normal é indicar uma quebra de linha;

- $$ O Número de processo, relativo ao sistema operacional, que o script atua está fazendo uso;

- $! Utilizada para repassar um número de erro ao software.

```
open my $fh, '<', '/var/tmp/perl_101.txt'
     or die "Ocorreu um erro: $!";
```

11.4 Variáveis Especiais: Para Saber Mais

Neste capítulo foi demostrado as principais variáveis especiais do Perl, porem, sem muita profundidade no assunto, visto que este material é voltado para o iniciante na linguagem, portando ao leitor que deseja saber mais, favor consultar as seguintes referências:

http://perldoc.perl.org/perlvar.html
http://www.perl.com/pub/a/2004/06/18/variables.html
http://www.kichwa.com/quik_ref/spec_variables.html

12. Estruturas da Linguagem

Este é o capítulo onde nós explicamos quais os principais itens do Perl para controlar o fluxo do sofware.

12.1 Estruturas Condicionais

Perl tem estruturas condicionais com excessão do Case/Switch [1], porém se você realmente quiser usá-los, pode-se recorre ao módulo Switch [2] na CPAN [3]. Em Perl as condições podem ser expressões (explicados com mais detalhes nos próximos capítulos). Operadores lógicos e booleanos são bastante empregados para esta tarefa.

- **if:** Análogo ao “se” (obviamente condicional) em português, indica que “se” uma condição for statisfeita, o bloco reservado por esta declaração será executado.

```
if ( condition ) {
     # coding block I
} elsif ( other condition ) {
     # coding block II
} else {
     # coding block III
}
```

No entanto o **if** tem uma negativa, esta chama-se **unless**, veja o seu funcionamento a partir do exemplo abaixo:

```
my $test = 1;
unless ( $test ) {
     # Não vai executar este bloco
}

$test = 0;
unless ( $test ) {
     # Com “test” falso (zero), o bloco é executado
}
```

[1] http://en.wikipedia.org/wiki/Switch_case
[2] http://search.cpan.org/~rgarcia/Switch-2.16/Switch.pm
[3] http://search.cpan.org

12.2 Estruturas de Repetição

Sua utilidade é manter um fluxo contido dentro de um derminado bloco, afim de ter todas as iterações feitas ou esperar por alguma ação do software, esta é a explicação simples do que uma Estrutura de Repetição representa, porem, tudo fica mais fácil de compreender com os exemplos a seguir.

- **while:**

```
my $test = 1;
while ( $test ) {
     # Este é um loop infinito, porque a expressão
     # não muda, não é alterado o valor de “$test”
     print “Loop\n”;
}

my $i = 1;
while ( $i <= 10 ) {
     # Este loop não é infinito, o contador “i” está
     # sendo incrementado
     print $i, “\n” and $i++;
}
```

Ou seja, quando alguma expressão suprir o que o **while** espera, o loop é interrompido.

Porem, neste mesmo exemplo nós temos uma negativa para o while, chamado “**until**”, veja o mesmo exemplo acima, utilizando a lógica desta negativa:

```
my $i = 1;
until ( $i == 10 ) {
     print $i, "\n" and $i++;
}
```

Observe que tanto o “while” como o “until” o fluxo de controle espera um fator ou situação para que o “loop” tenha um fim.

- **for:** Aqui nós temos um controle diferente para o fluxo, análogo a frase “para cada elemento faça”, esta é a idéia central do “for”, veja:

```
my $max = 10;
for ( my $i = 0; $i <= $max; $i++ ) {
     print $i, “\n”;
}
```

Este exemplo é igual ao que nós usariamos em linguagem C, com excessão a forma como declaramos variáveis.

- **foreach:** Tem basicamente a mesma função do “for” porem, com mais expressividade:

```
my @array = ( 1, 2, 3, 4, 5 );
foreach my $n ( @array ) {
     print $n, “\n”;
}
```

Aqui a frase que expressa o conceito acima é “para cada elemento na estrutura”.

13. Tratamento de Arquivos

Um dos pontos mais fortes do Perl é o tratamento de texto em si, de forma simples e poderosa nós podemos estender estas vantagens para os arquivos que estão no sistema de arquivos, afinal, em sua grande maioria eles também são texto. O método mais utilizado para esta tarefa é o "**open**" [1].

O método "**open**" [1] é composto de três parâmetros, sendo o primeiro o *filehandler* [2], o segundo a forma como ele será utilizado e o terceiro, o path para o arquivo.

```
open(
     my $fh,              # parametro 1
     ‘<‘,                 # parametro 2
     “/var/tmp/text.txt”  # parametro 3
) or die $!;
```

- **Parâmetro 1:** Consiste na variável que o Perl irá utilizar para "navegar" pelo arquivo. Ou seja, uma referência lógica para o arquivo no sistema de arquivos. Neste ponto nós podemos usar qualquer nome, é apenas uma variável local;
- **Parâmetro 2:** Este é o modo como nós vamos lidar com este arquivo, os modos podem ser:
  - **"<":** leitura, sendo que o fluxo dos dados é do arquivo para o *filehandler;*
  - **">":** escrita, o fluxo dos dados é do *filehandler* para o arquivo;
  - **">>":** escrita, porem, o arquivo é aberto com o modo "**append**" [3], no qual o conteúdo do arquivo é respeitado, e o novo conteúdo vai ser escrito ao final;
  - **"+>":** leitura e escrita, indica que nós vamos abrir o arquivo e percorre-lo nos dois sentidos tanto para leitura como para escrita, através do método "seek" [4];
- **Parâmetro 3:** *Path* para o arquivo, ou variável do tipo Scalar.

O método "open" [1], por ser muito simples e poderoso pode ser utilizado de várias formas diferentes, incluindo a leitura do *STDIN* [5] para uma variável local. Segue um exemplo de como interagir com um arquivo, para leitura e para escrita:

```
open my $ro, '<', '/etc/passwd'             or die $!;
open my $rw, '>', '/var/tmp/new_passwd.txt' or die $!;

while ( my $line = <$ro> ) {
     chomp $line;
     print $rw $line, "\n";
}

close($ro);
close($rw);
```

Observe que no primeiro "open" nós estamos abrindo um arquivo para leitura e no segundo um arquivo para escrita. Outro ponto de destaque é a forma como nós lidamos com o *filehandler*, no qual existem os símbolos "<" e ">" envolta, isso faz com que o

tratamento desta variável seja análoga a um Array, e por este motivo nós podemos jogar o conteúdo de um arquivo de forma muito simples para o Array. Veja:

```
my @array = <$ro>;
```

[1] http://perldoc.perl.org/functions/open.html
[2] http://en.wikipedia.org/wiki/Filehandle
[3] http://en.wikipedia.org/wiki/Open_%28system_call%29
[4] http://perldoc.perl.org/functions/seek.html
[5] http://en.wikipedia.org/wiki/Stdin

14. Expressões Regulares

Em ciências da computação, expressões regulares também são conhecidas como *regex* ou *regexp*, é uma forma concisa e flexível para fazer "*match*" [1] em strings (padrão texto), a procura de caracteres em particular, palavras ou padrões de caracteres. Uma *regex* é escrita de uma forma particular, e será interpretada pelo *engine* (motor) de expressões regulares do Perl (também conhecido como PCRE). Um dos mais famosos e utilizados motores para este tipo de tarefa. Também um dos grandes destaques do Perl.

Em expressões regulares nós temos caracteres especiais para construir as nossas expressões, abaixo seguem exemplos:

Limitadores de caracteres ou da posição deles na string:

- `\` Interpreta o caractere de forma literal, ao invés de um dialeto para a expressão;
- `^` Começo da linha;
- `.` Qualquer caractere, exceto uma quebra de linha;
- `$` Fim da linha;
- `|` Alternado, entre opções na expressão regular;
- `()` Agrupamento de expressão que foi encontrado no texto, depois ele pode ser recuperado através de variáveis especiais no Perl ($1, $2, ..., $9);
- `[]` Classe de caracteres;

Quantidade de vezes que um caractere está presente na string, ou sub-expressão:

- `*` Zero ou mais vezes;
- `+` Uma ou mais vezes;
- `?` Uma vez ou nenhuma;
- `{n}` Exatamente, "n" vezes;
- `{n,}` Ao menos "n" vezes;
- `{n,m}` Ao menos "n" vezes, porem, não mais do que "m";

Estes símbolos podem ser combinados assim com se fosse uma nova linguagem de programação, existem softwares que usam *regex* de forma tão abrangente que quase todo o software tem bases nele:

- `\w` Padrão para uma palavra, contendo caracteres alpha-numéricos, incluindo o "_";
- `\W` Exatamente ao contrário do "\w";

- `\s` Padrão para um caractere em branco;
- `\S` Exatamente ao contrário do “\s”, representa um caractere não em branco;
- `\d` Padrão para caracteres numéricos;
- `\D` Exatamente ao contrário, um caractere não numérico;

Existe muitos outros símbolos para utilizarmos com *regex*, estes são apenas os mais básicos.

Exemplo de *regex*:

```
my $string = “50 Thrid Street, San Francisco, CA”;

if ( $string =~ /^(\d+)\s+(.*?),\s+(.*?),\s+(\S+)$/ ) {
     print “Number: $1\n”;
     print “Street Name: $2\n”;
     print “City Name: $3\n”;
     print “State Name: $4\n”;
}
```

Existem vários sites para ajudá-lo a construir uma *regex*, dentre eles o destaque vai para o simples e conciso **Rubular**: http://www.rubular.com

[1] http://en.wikipedia.org/wiki/Pattern_matching

15. Módulos

Um módulo é um componente contendo código-fonte Perl para uso específico. Este é o mecanismo onde o Perl reúne fontes em um mesmo "namespace" [1], no qual um módulo pode conter outros. Esta forma de organização pode ser comparado com uma classe, enquanto programação Orientada a Objetos é aplicada, porem, estes dois conceitos são diferentes, isso deve estar claro ao leitor.

Uma coleção de módulos, acompanhando documentação, sistema de compilação/ instalação e, geralmente, uma suite de testes, compõem uma distribuição do Perl. Um exemplo disso é a CPAN, porem, com grandes proporções.

Como nós já citamos, Perl é uma linguagem que suporta estilos diferentes de programação, portanto, você pode encontrar módulos escritos de forma procedural [2] (por exemplo Test::Simple [3]), Orientado a Objetos (por exemplo o XML::Parser [4]), os dois considerados igualmente válidos, seguindo todas as atribuições que um módulo deve ter. Módulos também pode ser usados de forma mista, a exemplo do DBIx::Class [5], ou ainda, ser uma prática (pragma) como o "strict.pm" [6], o qual toma medidas perante o código, logo após ser invocado. Um módulo em Perl pode até afetar a sintaxe da linguagem introduzindo uma DSL [7], assim como o Moose [8] o faz.

[1] http://en.wikipedia.org/wiki/Namespace
[2] http://en.wikipedia.org/wiki/Procedural_programming
[3] http://search.cpan.org/~mschwern/Test-Simple-0.94/lib/Test/Simple.pm
[4] http://search.cpan.org/~msergeant/XML-Parser-2.36/Parser.pm
[5] http://search.cpan.org/~frew/DBIx-Class-0.08115/lib/DBIx/Class.pm

[6] http://search.cpan.org/~dapm/perl-5.10.1/lib/strict.pm
[7] http://en.wikipedia.org/wiki/Domain-specific_language
[8] http://search.cpan.org/~drolsky/Moose-0.94/lib/Moose.pm

15.1 CPAN

Trabalhamos com Ciências da Computação, somos em parte cientistas, logicamente. Sabemos que a melhor forma de aprender algo é através do seu significado, e neste assunto em particular, estamos tratando da CPAN.

-- CPAN: Comprehensive Perl Archive Network --

Sendo este o maior repositório de bibliotecas existente. Possivelmente, um dos maiores monumentos da comunidade open-source: atualmente com 7932 colaboradores, 77874 módulos, 19458 distribuições e 61037 uploads [1].

Portanto ao procurar soluções em Perl, um dos primeiros lugares de procura será a CPAN (http://cpan.org ou http://search.cpan.org).

Além de centralizar, ela nos provê acesso para:
- Buscas por nomes, conteúdo, descrição;
- Instalação do conteúdo, biblioteca, software;
- Visualização do conteúdo (fontes, documentação);
- Manutenção, principalmente upgrades ou downgrades;

Perl também é conhecido como o canivete suíço das linguagens de programação, definitivamente a CPAN amplifica o poder de uma ferramenta tão útil e versátil como esta.

A CPAN também é uma das formas mais fáceis de se obter o Perl, propriamente dito. Lá é mantida uma cópia muito atualizada do projeto que envolve o seu interpretador.

[1] `$ perl -e 'print time();' nos retorna "1263868440";`

15.1.1 CPAN: Exemplo Prático

Sabemos sua a utilidade e queremos vê-la em funcionamento:

1. Já temos o nosso interpretador Perl instalado no sistema operacional, e a instalação padrão demonstrada, inclui um binário chamado "cpan". Observe:

```
$ cpan

cpan shell -- CPAN exploration and modules installation (v1.9402)
Enter 'h' for help.

cpan[1]>
```

Ele nos apresenta um shell interativo para manipulação. De agora em diante, é aqui que nós vamos executar os próximos comandos.

2. Continuando o exemplo, vamos instalar um módulo simples:

```
[1]> install Test::More
CPAN: Storable loaded ok (v2.19)
Going to read /home/otaviof/.cpan/Metadata
Database was generated on Wed, 11 Nov 2009 15:28:14 GMT
CPAN: LWP::UserAgent loaded ok (v5.834)
CPAN: Time::HiRes loaded ok (v1.9719)
Fetching with LWP:
     http://www.perl.org/CPAN/authors/01mailrc.txt.gz
CPAN: YAML loaded ok (v0.70)
Going to read /home/otaviof/.cpan/sources/authors/01mailrc.txt.gz
CPAN: Compress::Zlib loaded ok (v2.015)
..............................................................
..........DONE
Fetching with LWP:
     http://www.perl.org/CPAN/modules/02packages.details.txt.gz
Going to read /home/otaviof/.cpan/sources/modules/
02packages.details.txt.gz
     Database was generated on Tue, 19 Jan 2010 12:39:07 GMT
...............
New CPAN.pm version (v1.9402) available.
[Currently running version is v1.9301]
You might want to try
     install CPAN
     reload cpan
to both upgrade CPAN.pm and run the new version without leaving
the current session.


.............................................................DONE
Fetching with LWP:
     http://www.perl.org/CPAN/modules/03modlist.data.gz
Going to read /home/otaviof/.cpan/sources/modules/
03modlist.data.gz
..............................................................
..........DONE
Going to write /home/otaviof/.cpan/Metadata
```

Observe que os primeiros passos da instalação são dedicados a acessar o repositório e obter as informações para realizar esta operação.

3. Para verificar quais as extensões/módulos estão instalados no seu computador, basta utilizar o comando "r", veja o exemplo:

```
cpan[3]> r
```

```
Package namespace    installed    latest  in CPAN file
Attribute::Handlers    0.78_03      0.87  SMUELLER/Attribute-
Handlers-0.87.tar.gz
AutoLoader                5.67      5.70  SMUELLER/
AutoLoader-5.70.tar.gz
(...)
Net::SNMP               v5.2.0    v6.0.0  DTOWN/Net-SNMP-
v6.0.0.tar.gz
Pod::Man                  1.37      2.23  RRA/
podlators-2.3.0.tar.gz
Term::ANSIColor           1.12      2.02  RRA/
ANSIColor-2.02.tar.gz
Text::Tabs           2007.1117 2009.0305  MUIR/modules/Text-Tabs
+Wrap-2009.0305.tar.gz
226 installed modules have no parseable version number
```

Outra opção, porem, esta deve ser chamada a partir do shell, é:

```
$ perl -MCPAN -e 'CPAN::Shell->install(CPAN::Shell->r)'
```

4. A própria CPAN está disponível para ser instalada via CPAN:

```
> install Bundle::CPAN
```

16. Boas Práticas

As boas práticas são as bases para implementações de sucesso, muitas vezes mais importantes do que a linguagem em si. Portanto, mesmo este sendo um curso dedicado a ensinar sobre Perl, é imprescindível tocarmos nesta temática.

16.1 Agile e Metodologias de Trabalho

Desenvolvimento de software utilizando os conceitos de "Agile" [1] refere-se às metodologias baseadas em interações, onde requisitos e soluções são desenvolvidos de forma colaborativa, entre times auto-organizados e multi-funcionais. O termo "Agile" (Ágil, em português) foi cunhado através do Manifesto Ágil [2], de 2001.

Ao citar o manifesto, devemos deixar claro seus valores:

- Interações e indivíduos são mais importantes do que processos e ferramentas;
- Software funcional, livre de bugs, é mais importante do que uma boa documentação;
- Colaboração com o cliente é mais importante do que um contrato;
- Responder a mudança é mais importante do que seguir o plano inicial;

Como pode ser visto pelos valores acima, nós temos um fator cultural muito forte, com o intuito de ser mais eficiente para o que nós vivemos atualmente, pois é reflexo da experiência e experimentação.

Os métodos do Agile, geralmente exigem um gerenciamento de projetos disciplinado, que encoraja freqüentemente a inspeção e a adaptação, uma filosofia de liderança que encoraja o trabalho em equipe, a auto-organização, e uma série de boas práticas de engenharia. Tudo isso para levar ao objetivo principal: uma entrega rápida de um software de altíssima qualidade, com proximidade a abordagem do empreendimento beneficiado, alinhando o desenvolvimento do software, necessidades do cliente e os objetivos da corporação.

As bases conceituais para esta metodologia, são:
- Lean (http://en.wikipedia.org/wiki/Lean_manufacturing);
- Soft Systems Methodology (http://en.wikipedia.org/wiki/Soft_systems_methodology);
- Speech act (http://en.wikipedia.org/wiki/Speech_act);
- Six Sigma (http://en.wikipedia.org/wiki/Six_Sigma);

[1] http://en.wikipedia.org/wiki/Agile_software_development
[2] http://en.wikipedia.org/wiki/Agile_Manifesto

16.2 Version Control System (VCS)

Ou traduzindo o termo: "Sistema de Controle de Versão". É o sistema para gerenciar as mudanças em documentos, programas ou qualquer outro tipo de informação guardada em arquivos de um computador. O seu uso mais comum está relacionado ao desenvolvimento de software, onde os programadores, guardam suas mudanças. Estas são geralmente identificadas com um número, denominado versão.

Ter um sistema destes enquanto existe desenvolvimento de um software, nos possibilita:

- Ter um histórico detalhado sobre todo o desenvolvimento do software;
- Trabalho colaborativo, entre grupos distribuídos;
- Junção de partes diferentes em um mesmo código-fonte, conhecido como "merge";
- Centralização do trabalho feito;
- Voltar a um estado anterior para corrigir um bug, ou fazer uma mudança;

Para saber mais:
http://en.wikipedia.org/wiki/Git_%28software%29
http://en.wikipedia.org/wiki/Version_control_system
http://git-scm.com/
http://www.kernel.org/pub/software/scm/git/docs/gittutorial.html

17.3 Test Driven Development (TDD)

Em uma tradução livre, TDD [1] significa: Desenvolvimento de Software Orientado a Testes. Consistem em uma técnica para desenvolvimento de software que apoia-se na repetição de um ciclo bastante curto: primeiro o programador desenvolve um teste (obviamente automatizado) no qual ele ira retornar falha, quando executado, então, o programador produz código para que simplesmente passe no teste, o último passo é refatorar, para que os fontes mantenham padrões aceitáveis de qualidade.

Existem muitos benefícios em usar TDD:

- TDD implica em escrever mais testes, e em um estudo realizado em 2005 [2], mostra que programadores que escrevem mais testes, são mais produtivos;

- Ter mais testes, aumenta a qualidade do código;

- Programadores usando TDD em projetos novos, afirmam que existe uma menor utilização do "debug". Porque, em conjunto com o controle de versão [3], quando os testes falham inesperadamente, pode-se simplesmente voltar a versão onde os testes não falham, e refazer o código problemático;

- Dar passos menores, durante o desenvolvimento, permite ao programador ter mais foco na primeira tarefa: fazer o primeiro teste passar. Gerenciar exceções e erros não são considerados iniciais. Testes para implementar situações desta natureza são feitos de forma separada, e todo o código é coberto por testes, aumentando a sua confiabilidade;

- Enquanto é verdade de que mais código é necessário para escrever softwares com TDD, afinal, existe o montante necessário para os testes em si, na maioria dos casos o montante final de código escrito é menor [4]. Os testes são essenciais para detectar os problemas na primeiras fases do código, e consequentemente, prevenindo que estes se transformem em uma epidemia em fases mais adiantadas, e que o programador não escreve mais código para a correção, ao invés de evitar o problema no início;
- O TDD direciona o desenvolvimento para ser mais flexível, extensível e modularizado. Este efeito vem porque a metologia exige que o programador pense no software como peças pequenas que devem ser escritas e testadas de forma independente, para depois serem integrados como um conjunto;

[1] http://en.wikipedia.org/wiki/Test-driven_development
[2] http://www.nrc-cnrc.gc.ca/eng/ibp/iit.html
[3] http://en.wikipedia.org/wiki/Version_control_system
[4] http://www.ipd.uka.de/mitarbeiter/muellerm/publications/edser03.pdf

16.4 Continuous Integration (CI)

A tradução para o nosso idioma é "Integração Contínua". Imagine-se trabalhando em um grande projeto de software, você é um programador em uma equipe. Cada um de vocês está escrevendo uma parte do projeto, trabalhando em recursos diferentes. Porem, quando você faz uma mudança no seu fonte, deve considerar as mudanças que já foram submetidas para o repositório principal, porque elas podem influenciar muito no seu trabalho. Este procedimento é chamado "Integração" (*Integration*). O que pode complicar todo este processo, é quando o desenvolvedor fica muito tempo sem fazer isso, então, nós temos um sério problema para resolver.

Uma boa prática para toda esta situação é automatizar este trabalho, no qual um software reúne os fontes escritos pelos participantes do projeto e executa a suite de testes.

Para saber mais:
http://wiki.hudson-ci.org/display/HUDSON/Meet+Hudson
http://hudson-ci.org/
http://en.wikipedia.org/wiki/Continuous_integration

16.5 DRY

Esta sigla é em inglês e pode ter duas formas de representar o seu conceito:

- **Don't Repeat Yourself (DRY):** o que nós podemos traduzir para "Não se repeita"; e
- **Duplication is Evil (DIE):** Duplicações são Malignas, em português;

As duas frases acima devem ser lembradas constantemente por um programador. Afinal, os códigos-fonte para nós nada mais são do que texto, e por este motivo nós podemos ficar tentados a tratá-los simplesmente como tal e usar as ferramentas de copiar e colar. Neste ponto nós passamos a ter um problema.

O princípio desta boa prática esta ligado ao conceito de que cada peça de conhecimento deve ter uma única representação em um sistema. Este princípio foi formulado por Andy Hunt [1] e Dave Thomas [2] no livro **"The Pragmatic Programmer"** [3]. Neste livro os autores mostram como aplicar este princípio em várias partes de um projeto, inclindo *Schemas* [4] para *Databases* [5], planos de testes, compilação (building) [6] e até documentação.

Quando o princípio **DRY** [7] é aplicado com sucesso, a modificação de um único elemento não modifica outro elemento, não relacionado. Adicionalmente, elementos que são relacionados de forma lógica, todas as suas mudanças são previsíveis e uniformes. Existem muitas técnicas para executar o **DRY** [7].

Leia mais:
http://en.wikipedia.org/wiki/DRY

[1] http://en.wikipedia.org/wiki/Andy_Hunt_%28author%29
[2] http://en.wikipedia.org/wiki/Dave_Thomas_%28programmer%29
[3] http://pragprog.com/the-pragmatic-programmer
[4] http://en.wikipedia.org/wiki/Database_schema
[5] http://en.wikipedia.org/wiki/Database_system
[6] http://en.wikipedia.org/wiki/Software_build
[7] http://www.artima.com/intv/dry.html

16.6 Testes de Aceitação

Consistem em automatizar a forma como uma aplicação está respondendo através de sua interface final para o cliente ou quem vai consumir este serviço. Para isso nós devemos ter em mãos uma ferramenta para simular estas interações, que pode ser, por exemplo, o navegador (browser). Afim de automatizar, temos ferramentas como *WWW::Mechanize* [1] (no Perl) ou *SeleniumHQ* [2]. O foco dos dois é simular, de forma seqüencial e prevista, a interação de um usuário comum na interface. A idéia neste tipo de teste é cobrir todas as funcionalidade da ferramenta.

[1] http://search.cpan.org/~petdance/WWW-Mechanize-1.60/lib/WWW/Mechanize.pm
[2] http://seleniumhq.org/

16.7 Testes de Integração

Quando estamos escrevendo testes, durante o desenvolvimento no código-fonte, nós fazemos testes unitários, ou seja, cada unidade escrita terá uma rotina para assegurar o seu funcionamento. Porem, os Testes de Integração tem o propósito de juntar o sistema, ou tudo o que foi feito até o momento, e testá-lo em conjunto. Para isso, é feito um plano para fazer estes testes, o qual costuma ser relativamente complexo, todo este procedimento deve crescer conforme o software é entregue, aumentar a cada ciclo previsto de trabalho. Pode ser feita pela própria equipe, programador individualmente, ou por um grupo especialista, dedicado somente a testar.

Leia mais:

http://en.wikipedia.org/wiki/Integration_testing

16.8 Conclusão: Automação

Uma conclusão para este capítulo é a respeito de uma prática que nós repetimos em, quase todas, as boas condutas citadas anteriormente: automatizar. Esta é a chave para o nosso controle e histórico. Se uma operação precisa ser feita duas vezes, conclui-se que nós devemos deixá-la guardada em forma de script, ou, descrição automatizada dos procedimentos. Perl é uma linguagem adequada e competente para esta tarefa.

Também fica outra observação: ao versionar um projeto, ele deve ser feito por completo, tudo o que é necessário para ele funcionar deve estar sendo citado (por sua documentação) ou, preferencialmente, presente no domínio do projeto.

Apesar de haver muitos procedimentos para um bom projeto, não é nossa intenção de apresentar práticas de execução complexa e trabalhosa, nossa intenção é deixar claro a importância da simplicidade, quanto mais simples melhor, menos é mais. **KISS** (Keep It Simple Stupid) [1].

[1] http://en.wikipedia.org/wiki/KISS_principle

Apendice I: Editores de Texto

Atenção: O editor é a arma mais potente no seu arsenal!

No mundo de linguagens dinâmicas a ferramenta mais comum, para o dia-a-dia, é um bom editor de textos.

Na afirmação acima, fica sendo a explicação a parte mais interessante: linguagens dinâmicas -- *como o próprio nome já diz* -- traz as diretivas de como o software vai se comportar para o tempo de execução, assim como já foi explicado, em outras palavras, o comportamento do software é mais difícil de prever.

A palavra "*prever*" é uma das pedras fundamentais para o trabalho de uma **IDE**, pois, ela ajuda o programador a entender e interagir com o *comportamento* do software.

Por exemplo: *Suponhamos que você criou uma nova variável: "*`my $test;`*", agora, vamos nos lembrar quais tipos, ou seja, quais valores eu posso atribuir a esta variável. Ao falar de Perl, nós já temos a resposta: qualquer um! Agora, voltando ao trabalho das IDEs, você deve estar de acordo comigo: este é um comportamento _muito_ mais complexo, se o compararmos a uma linguagem estática.*

Porem, vamos voltar aos editores de texto, agora. Na contra-mão das IDEs, temos muitas vantagens:

- rápido para inicializar (exige poucos recursos computacionais);
- foco para a interação com o texto;
- recursos de edição avançados;
- extensibilidade;
- macros;
- snippets;
- extremamente personalizável;

Quando trocamos o que está relacionado a "prever" e nos voltamos para o "*planejar*". Ou seja, uma interação entre homem e texto.

Editores recomendados:

- Vim (GVim, Vim, MacVim) [1];
- EMacs [2];
- TextMate [3];
- JEdit [4];

No entanto, ao leitor interessado pelo uso de uma IDE, temos um projeto que está crescendo bastante, ganhando adeptos e certamente merece a nossa atenção: Padre IDE ("Padre, the Perl IDE") [5].

Para saber mais:

http://www.moolenaar.net/habits.html
http://oreilly.com/catalog/9780596519544
http://pragprog.com/the-pragmatic-programmer

[1] http://www.vim.org/
[2] http://www.gnu.org/software/emacs/
[3] http://macromates.com/
[4] http://www.jedit.org/
[5] http://padre.perlide.org/