Programando em várias linguagens

Software Poliglota

Paul Fris: www.sxc.hu

O Linux é um fenômeno indiscutivelmente internacional. Foi iniciado por um finlandês de linhagem sueca e ajudado por um tenente britânico cuja língua-mãe é o galês. Hoje a versão estável do kernel é mantida por um brasileiro e desenvolvida por uma "fauna" de hackers oriundos dos quatro cantos da Terra e dos sete mares. Se é assim, por que todos os softwares que usamos são escritos em inglês? Este mês, Steve Goodwin joga alguma luz no desenvolvimento em múltiplos idiomas, bem como no pacote *gettext*. POR STEVEN GOODWIN

A língua inglesa é, hoje, tão ou mais poderosa e influente em todas as sociedades do mundo quanto foi o Latim há centenas de anos atrás. Não é a mais expressiva, muito menos a mais popular. Com toda a certeza, não é a mais fácil de aprender. Entretanto, é o idioma mais conhecido do planeta. Com os escombros do velho Império Britânico ainda presentes nos ermos mais longínquos, aliados ao crescimento contínuo e monstruoso dos Estados Unidos da América, mais e mais as pessoas são compelidas a aprender o idioma de Shakespeare e usá-lo para ganhar competitividade no mercado.

Os computadores e a Internet contribuíram para aumentar essa bizarra entropia lingüística. Existem mais sites em inglês do que em qualquer outro idioma. A maioria esmagadora das linguagens de programação usa palavras inglesas como *if* e *while*, independente da nacionalidade do projetista, bem como quase todos os *prompts* e mensagens de erro dos softwares que usamos.

Entretanto, como o Linux está no comando de muitos sistemas ao redor do globo, pareceria xenofóbico de nossa parte se continuássemos a escrever programas que só "falam" inglês. Adicionar a capacidade de alterar o idioma do programa (o famoso *locale*) não é nem um pouco difícil e mostra o comprometimento do desenvolvedor para com o bem comum e a comunidade do Software Livre. Mesmo que não seja você mesmo quem traduza os textos para as diversas línguas, tornar isso fácil para que outros possam fazê-lo é, no mínimo, simpático. Este artigo mostra como.

Turning japanese

O GNU/Linux usa a tecnologia de *locales* para determinar muitas coisas: a tradução apropriada para um determinado texto, o conjunto de caracteres do alfabeto e os detalhes culturais específicos como a forma de expressar números,

Quadro 1: Categorias de Locale

Uma categoria define um conjunto de dados; cada idioma aceito possui seu próprio conjunto. Uma categoria pode definir a maneira de exibir a informação ao usuário. Por exemplo, os milhares devem ser grafados com vírgulas (como em inglês) ou com pontos (como em português)? Outro exemplo: as datas serão escritas no formato mês-dia-ano ou dia-mês-ano? Essas informações não estão relacionadas com o idioma em si, mas com aspectos culturais. Por isso o termo 'locale' é usado, pois engloba tanto a língua quanto essas informações.

Há funções padronizadas para formatar as definições de locale. Por exemplo, *strfmon* e *strftime* modificam o formato de moeda e de hora, respectivamente.

Categoria	Significado
LC_COLLATE	Ordem da concatenação dos dados
LC_CTYPE	Como definir o modo de exibição de caracteres. Seu nome vem do arquivo ctype.h do código fonte do Unix, pois também está a cargo de conversões para maiúsculas e minúsculas.
LC_MESSAGES	O texto traduzido. Este é o objetivo do present e artigo.
LC_MONETARY	Formato e símbolos de moeda.
LC_NUMERIC	Formato e símbolos para números em geral.
LC_TIME	Formato e símbolos para hora e data.

valores monetários e datas. Cada uma dessas áreas está explicada no quadro 1, *Categorias de Locale*, embora o objetivo deste artigo seja apenas a tradução das mensagens de texto.

Portanto, vamos começar com o programa mais mequetrefe que conhecemos, o famigerado *Hello World*. Faremos nossos programas sempre em C, mas algumas das técnicas podem ser aplicadas independentemente de linguagem. Como exemplo, o equivalente em PHP pode ser visto no quadro 2: "E no PHP, como fica?"

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  printf("Hello World!\n");
  return 0;
}
```

À primeira vista, o lugar onde deveremos colocar os textos traduzidos parece bem óbvio. Entretanto, durante a compilação não sabemos qual o formato e tamanho do texto traduzido, ou quais os idiomas que deverão ser incluídos. Isso impede que se inclua qualquer tradução no próprio programa diretamente. Em vez disso, definiremos alguns *dicionários* para cada palavra e frase usada no programa e empregaremos o pacote *gettext* para agir como se fosse um intérprete. O intérprete substituirá as palavras em inglês pela versão apropriada do dicionário no momento em que o programa estiver sendo executado ("em tempo de execução"). A palavra correta será determinada pelo locale específico do usuário. Precisamos, portanto, fazer duas coisas:

1. Colocar marcas no código, que dizem ao gettext: "me vê aí a tradução correta para a frase XYZ";
2. Construir um dicionário de tradução para cada linguagem que desejamos incluir em nosso programa.

Marcar o código é tarefa simples. Nós, os programadores, temos que ir de linha em linha do nosso código e indicar em quais delas o texto precisa ser traduzido. Fazemos isso com uma função em C chamada (adivinhe!) de *gettext*. Essa função consultará os dicionários e converterá o texto para a língua de destino.

```
printf(gettext("Hello World!\n"));
```

Essa função encontra-se na biblioteca *libintl*, portanto precisamos incluir seu cabeçalho em nosso programa:

```
#include <libintl.h>
```

Sob o GNU/Linux, o processo de compilação não vai precisar de outra biblioteca além da *libintl*. A palavra GNU é essencial aqui: a internacionalização não seria possível, não fossem os recursos incluídos diretamente na biblioteca *glibc*. Os usuários de outros sistemas Unix não têm tanta sorte. Entretanto, sem um dicionário de idiomas, nada será traduzido. Em nosso exemplo, por enquanto, isso não importa, pois o texto original em inglês será usado em todos os casos em que o texto em outra língua não for encontrado. Os programadores certamente notarão que esse método não é eficaz em todas as situações: afinal, há mais de uma maneira de declarar uma linha de texto. Entretanto, só aprendemos uma das muitas maneiras de se marcar uma frase para tradução. Precisaremos, portanto, usar outro método para lidar com os casos em que uma chamada à função *gettext* resultaria em erro de sintaxe. Por exemplo:

```
char *pHello = "Hello, World!\n";
```

Para contornar o problema, precisaremos criar uma macro que inclua o marcador mas que não cause nenhum efeito adverso na sintaxe do comando:

```
#define gettext_noop(String) String
...
char *pHello = gettext_noop("Hello, World!\n");
```

Depois, precisamos invocar o módulo de tradução da maneira usual, antes de jogar o texto na saída padrão. Veja um exemplo:

```
printf (gettext (pHello));
```

Esses marcadores não apenas fazem a tradução enquanto o programa está sendo executado, como também indicam qual texto precisa ser traduzido. Mais adiante veremos uma ferramenta que usa esses marcadores para construir o dicionário para as traduções. Se fôssemos construir o dicionário manualmente, o marcador *gettext_noop* não seria necessário.

Alguns programadores substituem esse marcador de nove caracteres por um único caractere de sublinhado (*underscore*). Com isso, evitamos um problema comum, pois recomenda-se que as linhas de código em C tenham no máximo 80 caracteres, para melhor legibilidade. A palavra *gettext* ocupa um espaço precioso. Vejam como é simples:

```
#define _(str) gettext (str)
#define N_(str) gettext_noop (str)
```

Quadro 2: E no PHP, como fica?

Escrever software internacionalizável em PHP é muito semelhante à sua contrapartida em C. Até as funções têm os mesmos nomes! Entretanto, quando o script for parte de uma página web, é melhor definir o locale de outra forma em vez de especificá-lo explicitamente como mostrado. Algumas idéias são variáveis de sessão e *cookies* na máquina do usuário.

```
<?php
setlocale(LC_ALL, "fr_FR");
textdomain("lm");
echo gettext("Hello World!\n");
?>
```

O efeito do *setlocale* também pode ser alcançado com a função *putenv*, para exportar uma variável de ambiente:

```
putenv ("LANG=fr");
```

Quadro 3: Linguagens Suportadas pelo *xgettext*

C, C++, ObjectiveC
PO
Python
Lisp, EmacsLisp
librep
Java
awk
YCP
Tcl
RST
Glade

Tabela 1: Símbolos de hash

Símbolo	Tipo de comentário	Observação
. (ponto final)	Automático	Não deve nunca ser molestado!
: (dois pontos)	Referências	O arquivo e número da linha onde a mensagem se encontra.
, (vírgula)	Indicador	Informa se a tradução está confusa ('fuzzy'), por exemplo.
(espaço em branco)	Tradutor	Digitado por um humano.

A norma GNU manda colocar um espaço entre o nome da função e o primeiro parêntese, embora ele possa ser omitido sem problemas.

Pronto! Nosso programa está internacionalizado! Falta agora o dicionário. Já resolveremos isso.

Vienna Calling

Construir um arquivo que contenha todas as mensagens de texto de um programa não é tão demorado quanto parece. Naturalmente, é um procedimento bastante comum e pode ser facilitado com o uso de ferramentas especiais. A mais conhecida é o *xgettext*. Esta é uma das poucas ocorrências em que o 'x' não significa um programa gráfico. Em vez disso, é uma maneira abreviada de dizer "eXtração"'. Esse programa varrerá o código fonte em busca de quaisquer porções de texto que estejam sendo usadas com a função *gettext* (ou *gettext_noop*). Sempre que ele encontra alguma mensagem, guarda-a num arquivo de dicionário com extensão .PO prontinho para ser traduzido. O programa entende o bastante sobre a linguagem C e outras linguagens (ver quadro 3: xgettext: Linguagens Suportadas) para discernir entre comandos, funções, comentários, variáveis e, mais importante, o texto a ser traduzido.

```
$ xgettext -d lm helloworld.c
$ tail -n 3 lm.po
#: helloworld.c:5
msgid "Hello World!\n"
msgstr ""
```

Como você pode ver, cada mensagem a ser traduzida possui um identificador de marcador e uma mensagem equivalente, pronta para ser traduzida. Cada mensagem pode ser traduzida para apenas um idioma, portanto esse arquivo tornou-se um *modelo*. Cada tradutor faz uma cópia desse arquivo e traduz as mensagens contidas nele para sua própria língua. Às vezes, o arquivo PO é renomeado para POT para deixar clara a diferença entre os arquivos específicos de cada idioma e o arquivo modelo.

Observe que o *xgettext* fará uma busca no código para encontrar a função *gettext*. Ele não conhece tanto assim sobre a sintaxe da linguagem C (ou qualquer outra) para entender técnicas como a do *#define _(str)*, mostrada anteriormente. Não se deve deixar de usar tais truques, entretanto. Há duas soluções bem populares. Uma delas é especificar o "_" como uma palavra chave adicional que substituirá a função *gettext*:

```
$ xgettext -d lm -k_ helloworld.c
```

Alternativamente, pode-se fazer um pré-processamento do arquivo em C (provocando a expansão da macro) antes de rodar o *xgettext*.

```
$ xgettext -C -d lm <(gcc -E ➥
helloworld.c)
```

Neste exemplo especificamos o parâmetro -C. Isso indica que o resultado obtido é um arquivo fonte. Os usuários do automake têm uma vida mais tranqüila, pois o Makefile irá gerar esses arquivos automaticamente.

Um detalhe salta aos olhos: os comentários do arquivo contém índices de exibição ("hashes"). Os comentários vêm em quatro "sabores" e são determinados pelo caractere que precede o hash, conforme vemos na Tabela 1.

O programa *xgettext* pode também inserir comentários no arquivo PO quando, por exemplo, acreditar que a mensagem será usada com formatação especial. O arquivo PO também contém cabeçalhos para indicar a data de revisão do arquivo, bem como o tradutor que o editou por último.

Tendo em mãos esse arquivo como modelo, precisamos criar agora um dicionário para uma língua estrangeira. Francês, por exemplo.

Tour De France

Começamos fazendo uma cópia do arquivo modelo. Depois, adicionamos a ele as mensagens traduzidas para o francês dentro das aspas de cada declaração *msgstr*.

```
msgid "Hello World!\n"
msgstr "Bonjour, le Monde"
```

Para editar as mensagens, podemos usar um editor de textos comum ou, preferencialmente, uma das muitas ferramentas para edição de arquivos PO. Os usuários do Emacs podem tentar o modo PO desse aplicativo faz-tudo.

Listagem 1: Caçando erros

```
$ msgfmt  lm.po
msgfmt: lm.po: warning: Charset "CHARSET" is not a portable encoding name.

Message conversion to user's charset might not work.
lm.po:19: `msgid' and `msgstr' entries do not both end with '\n'
msgfmt: found 1 fatal error
```

Quadro 4: ISO 8859

ISO	Conjunto de caracteres
ISO 8859-1	Ocidente ou Europa Ocidental
ISO 8859-2	Europa Central e Leste Europeu
ISO 8859-3	Europa Meridional e Malta (além de Esperanto)
ISO 8859-4	Europa Meridional
ISO 8859-5	Leste Europeu, caracteres cirílicos como o russo
ISO 8859-6	Arábico
ISO 8859-7	Grego
ISO 8859-8	Hebreu
ISO 8859-9	Turco
ISO 8859-10	Nórdico (Sámi, Inuit, Islandês)
ISO 8859-11	Thai
ISO 8859-12	(era Celta, mas está obsoleto)
ISO 8859-13	Báltico
ISO 8859-14	Celta
ISO 8859-15	Transeuropeu (inclui símbolo do Euro)
ISO 8859-16	Sudeste Europeu (inclui símbolo do Euro)

Há várias ferramentas para quem não dispensa uma interface gráfica. O programa *poeditor* é uma boa pedida.

Para que possa ser usado por nosso *Hello World*, o arquivo PO precisa ser convertido num formato binário, legível para a máquina. O programa apto para a façanha chama-se *msgfmt*; quando executado, cria um arquivo (com extensão *.mo* em vez de *.po*) otimizado para melhor acesso às mensagens. Não é lá muito simples de usar, mas possui um belo depurador que, de cara, detectou o erro que introduzimos intencionalmente no código ali atrás. Matou na hora? Não? Bem, veja a Listagem 1.

O primeiro aviso simplesmente nos lembra de que não alteramos, ainda, as informações no cabeçalho. Isso é facilmente corrigível; só precisamos "remendar" a linha com a codificação de caracteres apropriada.

```
"Content-Type: text/plain; ➥
 charset=ISO-8859-1\n"
```

Para determinar a codificação apropriada para seu idioma, veja a tabela ISO 8859 ou consulte [1] para uma análise mais detalhada. Essa informação é mais útil para os tradutores do que para os programadores. Bem como a funcionalidade estendida oferecida por [2].

O segundo erro pode ser facilmente corrigido, embora seja mais difícil de ser encontrado por humanos em programas grandes. O msgfmt pode ainda fazer uma verificação de cadeias de caracteres para conferência do número correto de argumentos, bem como de tipos. Para isso, usamos a opção *-c*. Estamos prontos! Que tal um teste?

Quadro 5: Unicode

Todos os exemplos mostrados neste artigo usam caracteres ASCII. Com isso, abrangemos a maioria das línguas ocidentais mas negligenciamos os conjuntos de caracteres que requerem dois bytes, como o chinês. Para que sejam inteiramente reconhecidos, precisamos trabalhar em Unicode, o que envolve uma quantidade razoavelmente maior de trabalho, já que o tipo *char* não pode ser usado e é substituído por *wchar_t*. Além disso, funções populares como o *sprintf* precisam ser adaptadas para usar suas versões *wide*, como o *swprintf*.

Norwegian Wood

Para convencer nosso programa a usar o dicionário no idioma apropriado, precisamos adicionar mais algumas linhas em seu código para informar que estamos felizes em poder usar os locales. Esse punhado de linhas não é complicado de incluir e é relativamente comum a qualquer programa.

```
#include <locale.h>
...
char *pPackage = "lm";
char *pDirectory = "locale";
...
setlocale (LC_ALL, "");
bindtextdomain (pPackage, ➥
pDirectory);
textdomain (pPackage);
```

A função *bindtextdomain* indica o diretório principal onde os arquivos de dicionário residem, enquanto *textdomain* requer o nome de nosso *pacote*, ou seja, do programa. Nosso pacote chama-se 'lm', uma vez que criamos um dicionário chamado *lm.mo*. Observe que, se um caminho relativo for especificado no diretório *locale*, não se pode mudar de diretório (com o comando *cd*), pois as coisas ficariam fora de alcance para nosso programa.

Em nosso diretório principal, precisamos criar um diretório *locale* e copiar o arquivo *lm.mo* para o lugar apropriado na árvore. Esse lugar seria:

```
$ mkdir -p locale/fr/LC_MESSAGES
$ cp lm.mo locale/fr/LC_MESSAGES
```

Como o *pacote* se chama 'lm.mo' qualquer que seja o idioma, usaremos o nome do diretório para distinguir entre o *lm.mo* francês e o *lm.mo* tedesco. Esse nome é determinado pelos códigos internacionais de idioma, que podem ser encontrados em [3]. O diretório chamado LC_MESSAGES é necessário devido à grande variedade de informações diferentes sobre locales que possam existir. Há diretórios para indicar o formato de data e de como representar números e valores monetários (veja o quadro 1: *Categorias de Locale* para mais informações).

Vamos rodar nosso programa (sem ter que recompilá-lo!) usando o locale francês. Olhem só que beleza!

```
$ LANG=fr_FR ./hello
Bonjour, le monde
```

O comando *LANG=fr_FR ./hello* alterou a variável LANG apenas para essa instância do programa *hello*. Para alterar o locale de forma permanente, devemos exportar a variável de ambiente LANG da maneira usual:

```
$ export LANG=fr_FR
$ ./hello
Bonjour, le Monde
```

Isso talvez não funcione se seu sistema estiver exclusivamente em inglês, já que não há um locale francês instalado (outros potenciais problemas são discutidos em [4]). O arquivo */etc/locale.gen* indica quais locales foram gerados em sua máquina, enquanto o arquivo */usr/share/i18n/SUPPORTED* informa quais locales *podem* ser instalados (bem como os conjuntos de caracteres ISO-8859). Para gerar um locale francês, basta:

```
$ su
# Você deve ser root para essas ações
Password:
# echo "fr_FR ISO-8859-1" >> ➥
/etc/locale.gen
# locale-gen
Generating locales...
  fr_FR.ISO-8859-1... done
Generation complete.
```

Usuários do Debian devem usar o comando *dpkg-reconfigure locales*.

Podemos testar o novo locale com nosso programa "Hello World". Se ainda não funcionar, o bug pode estar nele. Para tirar a prova, use um dos programas multilingüe do pacotão GNU como, por exemplo, o *rm*.

```
$ LANG=fr_FR rm arquivo_inexistente
rm: Ne peut enlever 'arquivo_➥
inexistente': Aucun fichier ou ➥
répertoire de ce type
```

Para tornar seu dicionário disponível para os outros usuários, deve-se copiá-lo para o repositório global de arquivos *.mo* em */usr/share/locale/* (se sua distribuição usa um diretório diferente, veja o conteúdo da variável de ambiente

TEXTDOMAINDIR). Esse diretório usa a mesma árvore hierárquica vista antes. Copiar seus arquivos aqui (o que, obviamente, requer poderes de superusuário) significa que seu código não precisa mas especificar um diretório na função *bindtextdomain*. Substitua o nome do diretório pela palavra NULL.

Agora que já entendemos o processo técnico por trás da internacionalização de programas, vamos lançar um olhar mais atento aos detalhes que devemos considerar quando desenvolvemos um programa poliglota.

Spanish Eyes

Cada desenvolvedor tem seu próprio método de lidar com mensagens de texto em programas. Alguns, por exemplo, usam sempre a mesma biblioteca. Há, também, diversos métodos de montar mensagens dinamicamente, seja para conjugar verbos, adicionar plurais, decidir o gênero – e fugir de soluções incômodas como "Sr(a)." – ou construir sentenças grandes a partir de componentes menores (como aquela voz sexy que anuncia o próximo vôo no aeroporto). Tentaremos, no restante deste artigo, abordar alguns desses métodos e jogar alguma luz sobre os principais problemas envolvidos.

```
printf("Deleting %d file%s", ➥
iNum, iNum==1?"":"s");
```

O exemplo acima é uma das formas mais comuns de se criar um plural consistente. Se o número de arquivos (indicado pela variável iNum) for 1, o *printf* mostrará o substantivo *file* no singular. Qualquer outro número força o uso do plural, *files*. Mas isso é inglês! Nem todas as línguas seguem esse padrão. Em francês, por exemplo, não se pode falar 'zero arquivos', o plural é proibido para a quantidade zero. Algumas línguas bálticas precisam de flexões separadas para zero, um e mais de um. Para essas compensações, uma função separada, chamada *ngettext*, está disponível. Ela possui dois identificadores de mensagem (*string ID*, um para o singular e outro para o plural) e um número. O número é usado para determinar qual versão da mensagem deve ser usada na tradução. Veja o exemplo:

```
printf( ngettext("Deleting %d ➥
file", "Deleting %d files", ➥
iNum), iNum);
```

Ao deparar com o marcador *ngettext*, o programa *xgettext* gerará duas *String IDs* no arquivo *.PO*, prontinhas para o tradutor, bem como um comentário especial do tipo *c-format*, que veremos mais adiante.

```
#: helloworld.c:32
#, c-format
msgid "Deleting %d file"
msgid_plural "Deleting %d files"
msgstr[0] ""
msgstr[1] ""
```

Nem todos os problemas podem ser resolvidos pelo *ngettext*. Em algum ponto do programa o desenvolvedor pode tropeçar em casos – bem comuns, por sinal – em que usamos dois ou mais argumentos em um *printf*, pois a ordem das palavras é obrigatória. Mesmo em um programa simples em inglês, tipos %d e %s no lugar errado causam falha de segmentação (*core dumped*) no *printf*. Depois de traduzir uma frase simples, como "There are %d files named %s", não é nem um pouco difícil que o texto traduzido seja "Com o nome %s há %d arquivos". O que é pior: nós, os programadores, não podemos prever todas as milhares de possibilidades que cada um dentre as centenas de idiomas do mundo oferecem aos tradutores. Problemas mais sutis podem ocorrer em frases como "Copying file from %s to %s" ("Copiando arquivos de %s para %s").

Há dois métodos de resolver o problema da ordem correta das palavras. O primeiro método requer que o tradutor modifique a disposição das idéias na frase de forma a não ofender a ordem dos argumentos. O comando *msgfmt* pode ser invocado com a opção *-c*, de forma a fazer verificação de erros no arquivo *.PO*. A chave *-c*, na realidade, faz três testes distintos, sejam eles: *format* (que nos salvará a vida por ora), *header* (verifica a presença e conteúdo do cabeçalho) e *domain* (verifica problemas com as diretivas de domínio).

A segunda solução põe o fardo nas costas do programador, e é a mais acertada. O formato da mensagem tem de ser alterado para descrever a ordem dos parâmetros. Portanto, usando nosso exemplo de cópia de arquivos visto aí atrás, teríamos:

```
printf( gettext ("Copying file ➥
from %1$s to %2$s"), ➥
pSrc, pDest);
```

Os especificadores de formato especiais *%1$s* e *%2$s* são manipulados apenas pela função *printf* presente na biblioteca *glibc*. Versões não-GNU da *libc* podem não ser tão ricas em recursos.

Tendo resolvido o problema da ordem das palavras, temos ainda a questão de montar as mensagens durante a execução do programa – ou melhor, evitar fazer isso! É uma má idéia. As soluções que temos só funcionam se a frase inteira estiver disponível para o tradutor. Dividir o texto em seções e usar *strcat* deve ser, a qualquer custo, evitado: o tradutor não tem como entender a ordem ou o significado da sentença, pois não a possui inteira. Cada sentença contida no arquivo de dicionário deve estar completa e fazer sentido sozinha.

```
/* Por favor, não faça isso!! */
strcpy("Copying file from ");
strcat(pSrc);
strcat(" to ");
strcat(pDest);
```

Uma das palavras mais difíceis de traduzir é 'the'. O inglês só possui um artigo definido: é 'the' e pronto! Entretanto, o francês, o alemão, o espanhol e o português (entre muitos outros) não seguem a mesma filosofia. Dependendo da língua, há traduções diferentes para masculino, feminino, neutro e plural. O mesmo vale para o artigo indefinido 'a'. Normalmente essas palavras estão no meio de uma sentença. Como já estamos cansados de saber, construir as mensagens dinamicamente é uma péssima idéia. Em alguns casos pode ser tentador picotar a mensagem para acomodar todas as traduções, como mostrado na Listagem 2.

Devemos modificar o texto acima para que a mensagem diga '*a* directory' ('um' diretório) e '*a* file' ('um' arquivo), assim a versão traduzida será correta a despeito do gênero. Entretanto, se tivéssemos uma porção do programa que produzisse uma versão abreviada

da lista de arquivos, estaríamos duplicando o trabalho do tradutor! Por exemplo, veja o que acontece na Listagem 3.

É verdade! Estamos dobrando o trabalho do tradutor! Entretanto, o trabalho a mais é mínimo. Especialmente se comparado à aporrinhação do programador que seria necessária, ou levando-se em conta a confusão que a versão errada do 'the' poderia causar em algumas culturas e países.

China Girl

O último problema de implementação que veremos envolve a estética: a disposição dos elementos na tela, as caixas de diálogo, os menus e a alternância entre elementos com a tecla [Tab], por exemplo. Embora seu programa pareça bem bonito em inglês, no mesmo instante em que as palavras mudam seu *layout* bonitinho "vai pro brejo". Palavras em alemão, por exemplo, são em média 50% mais longas que suas equivalentes em inglês. Você tem duas opções: ignorar o problema ou contorná-lo com código inteligente.

Muitos (se não todos) os programas em modo texto não dão a mínima para formatação especial. Pelo contrário, seu resultado tem que ser simples para que possa ser jogado na entrada padrão de outro comando ou script. Softwares com interface gráfica são completamente diferentes. Por exemplo, o programador pode escolher colocar o texto em duas colunas, por exemplo X1 e X2, para tornar a coisa visualmente agradável ao usuário comum. Veja bem, não há nada de errado com isso! Desgraçadamente, entretanto, o texto da esquerda pode "vazar" para a coluna da direita se estivermos usando um locale diferente.

Para que isso não ocorra, o desenvolvedor vai ter que escrever mais código. Isso pode envolver o ajuste da posição da coluna da direita, talvez até com uma estimativa da largura da coluna da esquerda. Outra solução seria hifenizar tudo. Ou ainda rolar o texto quando ele for maior do que a janela visível (um exemplo bem conhecido é o XMMS). Os mais radicais simplesmente mutilarão a frase, descartando o pedaço que sobra e solicitando ao tradutor que seja mais conciso. A solução empregada irá variar dependendo da quantidade de trabalho que tanto os desenvolvedores quanto os tradutores querem ter. Na maioria das vezes, apenas os aplicativos em que a imagem é tudo (como os jogos, por exemplo) têm urgência em resolver esse tipo de problema.

Vienna

À medida que o software amadurece, novas mensagens vão sendo adicionadas. Retraduzir completamente o software é tarefa ingrata e, obviamente, perda de tempo. Em vez disso, usamos a ferramenta *msgmerge*, que constrói um novo arquivo .PO a partir do arquivo modelo (o .PO original, normalmente renomeado para .POT) sem qualquer tradução e da versão antiga do arquivo .PO já traduzido. O novo arquivo conterá todas as traduções já feitas e mais as novas mensagens ainda a traduzir.

```
$ msgmerge po_antigo.po ➥
po_atualizado.pot >➥
po_novo_e_atualizado.po
```

Metropolis

Com o pacote *gettext*, podemos criar programas poliglotas mesmo que não falemos nenhuma das línguas em questão. Usando catálogos separados, cada língua pode ter uma versão do software sem que seja preciso compilá-lo. Com isso, o trabalho de internacionalização é distribuído e voltado para os resultados, não para o código.

Com esse pensamento nos despedimos. Até mais! Farewell! Au revoir! Auf Wiedersehen! Adiós! Arrivederci! ■

Listagem 2: Menos tradução, mais problemas

```
if ( mygetfiletype(szFilename) == DIRECTORY)
      pFiletype = gettext ("directory");
if ( mygetfiletype(szFilename) == FILE)
      pFiletype = gettext ("file");
printf (gettext ("%1$s is a %2$s"), szFilename, pFiletype);
```

Listagem 3: Mais trabalho para o tradutor

```
if ( mygetfiletype(szFilename) == DIRECTORY)
      pFiletype = gettext ("directory");     /* mensagem já usada
antes - menos trabalho? */
if ( mygetfiletype(szFilename) == FILE)
      pFiletype = gettext ("file");
printf("%s : %s", szFilename, pFiletype);     /* nada a traduzir aqui
*/
```

Informações

[1] Sopa de letrinhas ISO8859: *http://wwwwbs.cs.tu-belin.de/user/czyborra/charsets/*

[2] Dados sobre idiomas: *http://www.eki.ee/letter/*

[3] Códigos internacionais de idioma: *http://www.loc.gov/standards/iso639-2/langcodes.html*

[4] FAQ do GNU gettext: *http://www.haible.de/bruno/gettext-FAQ.html#integrating_noop*

[5] Unicode: *http://www.unicode.org*

[6] FAQ do Unicode para Unix e Linux: *http://www.cl.cam.ac.uk/~mgk25/unicode.html*

[7] Tutorial de Unicode no Python: *http://www.reportlab.com/i18n/python_unicode_tutorial.html*

[8] FAQ sobre Perl, Unicode e i18n: *http://rf.net/~james/perli18n.html*

SOBRE O AUTOR

Quando operários vão a um bar, eles falam sobre futebol. Portanto, presume-se que, quando jogadores de futebol vão a um bar, eles falem sobre os operários! Mas quando Steven Goodwin vai a um bar não fala nem sobre futebol nem sobre operários: invariavelmente, fala sobre computadores...