FMA 215

Aula 16: Automação de Procedimentos

Arnaldo G. de Oliveira F.
e
Jorge L. deLyra

16 de Setembro de 2002

1 Alguns Conceitos Relevantes

Vimos anteriormente o uso de programação de linha de comando e de shell scripts para a execução de tarefas repetitivas. Esses recursos são muito flexíveis porque há no Linux um grande número de programas que executam pequenas tarefas e que, combinados de forma adequada, podem executar sequências extensas de operações. Há situações, por outro lado, em que não só queremos executar certos conjuntos de tarefas, como também que sejam verificadas certas condições prévias para a sua realização. Por exemplo, para que seja criado um arquivo binário executável é necessário que exista antes um arquivo texto com o código fonte escrito em alguma linguagem de programação, de forma que este possa ser processado pelo compilador adequado para gerar o executável. Assim, a condição para a criação do executável é a existência de um arquivo com o código fonte.

A ferramenta apropriada para lidar com esse tipo de situação é o comando `make`. Esse comando lê um arquivo chamado `Makefile`, escrito com uma sintaxe simples, que contém as instruções de como executar as tarefas desejadas. A estrutura do `make` é baseada em três elementos básicos: o **alvo** (**"target"**); um conjunto de **dependências**; um conjunto de **regras** (**"rules"**) de como produzir o alvo a partir das dependências. O alvo é, em geral, um arquivo a ser construído; no caso do exemplo acima, que será nosso exemplo nas tarefas, o alvo é o arquivo binário executável. As dependências são arquivos que devem existir previamente para que se possa gerar o alvo. As regras são as tarefas ou operações específicas sobre as dependências que geram o alvo. Essas regras podem ser operações de linha de comando, scripts ou programas, ou seja, todo o conjunto de elementos que já vimos antes.

1.1 Estrutura Básica de uma `Makefile`

Um exemplo da estrutura básica da tríade alvo – dependência – regra de uma Makefile é dado a seguir. Note que vários destes blocos básicos podem ser encadeados para codificar relações mais complexas de dependência entre muitos arquivos diferentes. Basta que uma das dependências de uma regra seja, além disso, o alvo de uma outra regra. O `make` percorre recursivamente todas as relações de dependência contidas na `Makefile`, até poder decidir que arquivos devem ou não ser refeitos.

```
# Estrutura central básica de uma Makefile.
# Comentários são feitos como em shell scripts.

alvo: dependencia1 dependencia2
	regra1
	regra2
```

O que está antes do sinal "`:`" é o alvo, o que vem depois é uma lista de dependências separadas por espaços. As linhas de regra, contendo uma regra cada uma, vêm a seguir. Note o espaço em branco no início de cada uma destas linhas. Este é um importante elemento da sintaxe da `Makefile`: trata-se na realidade de um caracter "`TAB`". Linhas contendo regras devem sempre ser iniciadas por um `TAB`, que *não* pode ser substituído por um conjunto de 8 espaços em branco.

Cada linha de uma regra na `Makefile` será executada por uma shell, sendo que por default é usada a `bash`. Note que cada linha é executada por um processo de shell *independente* dos outros, de forma que o conjunto das regras que seguem uma linha de alvo e dependências *não* forma um shell script, pois neste todas as linhas são executadas pelo *mesmo* processo de shell. Se quisermos que um script longo e complexo seja executado como um bloco em uma regra, a opção mais apropriada é escrever o script em um arquivo executável separado e executá-lo como uma das linhas de regra.

Só podemos incluir diretamente um script na `Makefile` se o escrevermos na forma de uma única linha, o que em geral não é muito conveniente.

A verificação da relação entre o alvo e as dependências é mais complexa do que aparenta até aqui, pois não é só a existência ou não dos arquivos que conta. Além disso o comando `make` verifica quais as *datas* em que foram criados ou modificados pela última vez os diversos arquivos envolvidos para decidir se realiza ou não as operações que recriam o alvo a partir das dependências. Assim, se a data da última modificação de algum dos arquivos que constam nas dependências for mais recente do que a data correspondente do alvo, o comando `make` repete as regras para a confecção deste. Se a data das dependências for anterior à do alvo, o comando `make` nada faz.

O comando `make` contém muitos elementos adicionais de sintaxe, que o tornam extremamente poderoso e flexível. Não pretendemos esgotar aqui todos estes elementos, mas vale a pena mencionar alguns dos mais importantes e mais utilizados. Um destes elementos é o uso de variáveis dentro da `Makefile`, as quais são também chamados de **macros**. Trata-se, na realidade, de variáveis de environment. O programa `make` herda da sessão que o inicia todo o conjunto de variáveis de environment, que ele pode utilizar internamente. Além disso, podemos acrescentar a este conjunto de variáveis herdadas novas variáveis que são definidas dentro da `Makefile`, sempre em letras maiúsculas, como é próprio para variáveis de environment. Estas variáveis são úteis para evitar repetições desnecessárias de conjuntos de nomes de arquivos ou comandos, à maneira do uso destas variáveis em shell scripts. Um exemplo simples da estrutura global de uma `Makefile` é dado abaixo, com os elementos mais comuns.

```
# Estrutura global de uma Makefile.

# Definicao de variaveis ou macros.
VARIAVEL1 = nome1 nome2 nome3
VARIAVEL2 = nome4 nome5 nome6

# Primeiro bloco de alvos, dependências e regras.
# Note a sintaxe específica para se fazer referência
# _ ao valor de uma variável, com os parentesis.
alvo1: dependencia1 dependencia2 $(VARIAVEL1)
        regra1
```

```
        regra2

# Segundo bloco de alvos, dependências e regras.
# Note que o primeiro alvo é uma das dependências, o que
# _ faz com que, neste caso, a inclusão da VARIAVEL1 nas
# _ dependências seja redundante.
alvo2: dependencia3 alvo1 $(VARIAVEL1) $(VARIAVEL2)
        regra3
        regra4
```

Inicialmente, definimos as variáveis auxiliares `VARIAVEL1` e `VARIAVEL2`. A seguir, são definidos os alvos `alvo1` e `alvo2`, separados por dois pontos (“`:`”) de suas respectivas dependências; algumas dessas dependências são definidas pelas variáveis previamente definidas. Observe que, nesse exemplo, tanto o `alvo1` quanto o `alvo2` dependem do conteúdo definido pela `VARIAVEL1`. Na linha seguinte a cada alvo, são definidas as regras; é importante lembrar que o espaço entre o começo da linha e cada `regra` é dado por um caracter de tabulação, isto é, deve-se sempre digitar um `TAB` no início da linha de definição de cada regra.

Outro conceito muito útil é o de **variáveis automáticas**. Estas são variáveis especiais da `Makefile`, com sua sintaxe própria, que permitem que o comando `make` reconheça certos padrões gerais de nomes que estão envolvidos numa determinada regra como alvos ou dependências. São úteis em particular a variável `$@`, que representa o alvo envolvido na regra, a variável `$<` que representa o primeiro elemento da lista de dependências, bem como a variável `$+` que representa o conjunto das dependências. Estas estruturas tornam mais sintética a descrição das regras, como é ilustrado abaixo num caso um tanto trivial.

```
# Uma forma um tanto complicada de se copiar arquivos.

arquivo2: arquivo1
        cp -pf $< $@

arquivo4: arquivo3
        cp -pf $< $@
```

Além disso, há também algumas estruturas de controle que podem ser usadas dentro da `Makefile`, bem como uma sintaxe especial usando o símbolo `%` dentro dos alvos e das dependências, o que permite a criação de regras especiais chamadas **"pattern rules"**. Estas regras especiais baseadas no reconhecimento de "patterns" têm uma aplicação muito geral, ampla e flexível. Elas tornam a escrita da `Makefile` muito mais simples e sintética, pois codificam em uma única regra um grande conjunto de regras que satisfazem a um "pattern" definido, como veremos nas tarefas. Elas são de particular utilidade se combinadas com as variáveis especiais que mencionamos acima. Há inclusive uma variável especial de grande utilidade que é específica para este tipo de regra, a variável `$*` que representa a parte do "pattern" comum entre o alvo e as dependências.

Observe que, apesar de que o comando `make` foi criado no Unix para auxiliar no desenvolvimento e manutenção de projetos grandes e complexos de software, geralmente utilizando a linguagem C, a sua utilidade não se limita, em absoluto, a este tipo de uso. Só para dar alguns outros exemplos de nossa experiência imediata: estas apostilas são desenvolvidas e mantidas através do uso do `make`, inclusive para manter atualizada a versão em HTML que se encontra na home-page da disciplina; toda a pesquisa de nosso pequeno grupo de pesquisa é baseada em uma coleção de programas, escritos primordialmente em Fortran, que são mantidos e desenvolvidos através do `make`; o livro-texto sobre Teoria Quântica de Campos na Rede, que pode ser encontrado na home-page de nosso grupo, é escrito e mantido através de uma estrutura bastante complexa de `make`.

Pode-se entender uma `Makefile` como uma forma de registro para a solução de um problema. Quando tentamos resolver o problema de como fazer uma determinada coisa no sistema, vamos descobrindo por tentativa e erro as relações relevantes e as operações necessárias para resolver cada parte do problema. A `Makefile` funciona como um registro ativo das relações de dependência e da sequência de operações que resolve o problema. Mesmo muito tempo depois da sua formulação, a leitura de uma `Makefile` nos recorda rapidamente e com precisão da solução que havíamos encontrado anteriormente para resolver o problema de como fazer uma determinada coisa no sistema.

2 Tarefas

Nestas tarefas vamos comparar a utilidade do `make` com a de um shell script convencional, cuja estrutura já conhecemos. Construiremos uma `Makefile` para a compilação de um programa escrito em Fortran, começando com uma de estrutura bem simples e incluindo nela, aos poucos, várias das estruturas mais importantes suportadas pelo comando `make`.

1. Vamos inicialmente criar um "shell script" capaz de compilar um programa escrito em várias subrotinas, separadas em arquivos diferentes. Para isso, use um browser para copiar o arquivo

   `http://latt.if.usp.br/fma215/materiais/aula-16/decmat.tgz`

   dentro de um diretório de trabalho de temporário com o nome, digamos, `makeaula`. Entre no diretório `makeaula` e dê o comando

   `tar xzvf decmat.tgz`

   Dê um `ls -l` e verifique que foi criado o seguinte conjunto de arquivos:

   ```
   -rw-rw-r--  1 <nome> <nome> 7739 Oct 30 21:47 Decompmat.f
   -rw-rw-r--  1 <nome> <nome>  439 Oct 30 21:47 cb_gen.f
   -rw-rw-r--  1 <nome> <nome> 1525 Oct 30 21:47 checkmats.f
   -rw-rw-r--  1 <nome> <nome>  278 Oct 31 15:05 fp_def.f
   -rw-rw-r--  1 <nome> <nome> 1427 Oct 30 21:47 getseed.f
   -rw-rw-r--  1 <nome> <nome> 2179 Oct 30 21:47 ranr.f
   -rw-rw-r--  1 <nome> <nome> 1082 Oct 30 21:47 writemat.f
   ```

2. Crie o script `comp.tcsh`, dado abaixo. Esse script verifica que existem os arquivos Fortran, compila cada arquivo `*.f` gerando seu correspondente objeto binário `*.o` e, então, cria o executável.

   ```
   #!/bin/tcsh -f
   #
   # Script para compilar um programa em Fortran.
   #
   # Define algumas variaveis de shell.
   set progfiles = ( Decompmat checkmats getseed \
   ```

```
                 ranr writemat writemats     )
set files = ( $progfiles cb_gen fp_def )
#
# Verifica a existencia dos arquivos necessarios.
#
# Loop sobre os arquivos.
foreach file ( $files )
    if ( ! -f "$file.f" ) then
        echo "Erro: arquivo $file.f não encontrado"
        exit 1
    endif
end
#
# Gera os objetos binarios e cria uma lista deles.
#
# Inicializa uma variável vetorial.
set ofiles
#
# Loop sobre os arquivos.
foreach progfile ( $progfiles )
    #
    # Gera o objeto binario.
    echo -n "Compilando $progfile.f... "
    g77 -O -c -o $progfile.o $progfile.f
    #
    # Acrescenta o objeto a lista.
    set ofiles = ( $ofiles $progfile.o )
    echo " feito."
end
#
# Cria o executável efetuando a linkagem dos objetos.
echo -n "Gerando executável... "
g77 -o decmat $ofiles
echo " feito."
```

Dê um `ls -l`, verifique que o script pode ser executado, altere as autorizações do arquivo com `chmod` se necessário e rode-o. Dê o comando `file *` para identificar cada arquivo criado e verificar que são o que esperamos.

Observe que *todas* as operações são efetuadas *toda vez* que o script for executado, independentemente de existir ou não o executável, o que poderia tomar muito tempo caso o programa contivesse muitos módulos. Para eliminar essa redundância de operações, seria necessário acrescentar-se ao script uma série de "`ifs`" condicionais que verificassem quais módulos já foram compilados e quais deveriam ser refeitos, o que aumentaria muito a complexidade do script.

3. Vamos criar agora uma `Makefile` bastante simples, com as regras necessárias apenas para a compilação de nosso programa, como no script acima. Crie o arquivo abaixo (sem autorização de execução pois não se trata de um executável) com o nome `Makefile`.

```
# Makefile para compilacao de programa em Fortran.

# Primeiro alvo (default target).
decmat: Decompmat.o checkmats.o getseed.o \
ranr.o writemat.o writemats.o
        g77 -o decmat Decompmat.o checkmats.o getseed.o \
            ranr.o writemat.o writemats.o

# Demais alvos (targets).
Decompmat.o: Decompmat.f fp_def.f cb_gen.f
        g77 -O -c -o Decompmat.o Decompmat.f

checkmats.o: checkmats.f fp_def.f cb_gen.f
        g77 -O -c -o checkmats.o checkmats.f

writemats.o: writemats.f fp_def.f cb_gen.f
        g77 -O -c -o writemats.o writemats.f
```

Para efetuar a compilação, simplesmente execute o comando `make`. Esse comando procurará automaticamente um arquivo `Makefile` no diretório corrente. Se não houver tal arquivo, o comando terminará com uma mensagem de erro. Se o arquivo tiver outro nome, rode o comando

```
make -f <nome_da_makefile>
```

Como padrão, o `make` busca o primeiro alvo ("target"), determina as suas dependências e tenta efetuar as suas regras. No caso acima, há uma série de arquivos que devem existir antes de se executar a compilação final, que faz a linkagem. A barra invertida (\), no final da linha, indica que a linha seguinte é uma continuação. Se os arquivos `*.o` não existem, o comando `make` entende que deve procurar regras para criá-los, ou seja, procurará alvos cujos nomes sejam essas dependências. Assim, as regras do segundo, terceiro e quarto "targets" serão efetuadas antes do primeiro. Se isso não for possível por qualquer motivo, `make` interromperá seu procedimento e apresentará uma mensagem de erro.

Agora, rode

```
./decmat
```

para verificar que o programa gerado funciona.

4. Vamos agora observar como o comando `make` se comporta com a alteração de algum arquivo. Edite, por exemplo, o arquivo `fp_def.f` e altere o valor do parâmetro `ndim` para 2. Rode novamente o comando `make`. Quais compilações foram efetuadas?

5. Mova o arquivo `checkmats.f` para `checkmats.f.OLD`. Edite novamente o arquivo `fp_def.f`, recoloque o valor anterior do parâmetro modificado e rode `make`. Houve compilação? Qual? Agora, mova `checkmats.f.OLD` de volta para `checkmats.f` e rode

   ```
   make checkmats.o
   ```

   o que deve gerar o objeto `checkmats.o`. Como se vê, é possível ordenar ao `make` que execute qualquer um dos alvos existentes na `Makefile`, o primeiro é apenas o default. Repita o comando acima. Por que não houve compilação? Agora, rode `make` sem argumentos novamente e analise as mensagens apresentadas. Quais arquivos foram compilados e por quê? Mais uma vez, rode o programa `decmat` criado para confirmar que ele funciona como antes.

6. Até aqui, procuramos entender qual a estrutura básica de uma `Makefile` e como funciona o processo de execução das regras segundo alterações nas dependências. O passo seguinte é sofisticar um pouco a `Makefile`, usando variáveis. Edite a `Makefile` e modifique-a para que fique como segue:

```
# Makefile para compilacao de programa em Fortran.

# Definicao dos objetos binarios.
OBJ =           \
Decompmat.o     \
checkmats.o     \
writemat.o      \
getseed.o       \
ranr.o

# Primeiro alvo. Observe que trata-se de um alvo
# _ virtual, que não corresponde a um arquivo.
all: decmat

# A regra de linkagem.
decmat: $(OBJ)
        g77 -o decmat $(OBJ)

# Demais dependencias.
Decompmat.o: Decompmat.f fp_def.f cb_gen.f
checkmats.o: checkmats.f fp_def.f cb_gen.f
writemats.o: writemats.f fp_def.f cb_gen.f

# Regra geral de compilacao de objetos.
%.o: %.f
        g77 -O -c -o $@ $<
```

Dê um

```
touch Decompmat.f
```

e rode novamente `make` para verificar que tudo funciona perfeitamente. Você sabe dizer porque foi necessário o `touch`?

Analisemos a nova forma da `Makefile`. Primeiramente, definimos uma variável `OBJ` cujo valor é o conjunto de objetos a serem produzidos na compilação. A seguir, criamos um alvo "default", com o nome `all`; neste caso, esse "target" não possui nenhuma regra para ser executada, pois ele apenas depende de outro alvo, o `decmat`. Assim, `all` funciona apenas como um apelido ou "alias" para a compilação. Depois, informamos na `Makefile` como gerar o executável a partir dos objetos definidos por `OBJ`. Finalmente, dizemos quais objetos dependem de que arquivos, sem especificar as regras para produzi-los.

Nesse ponto, lançamos mão de "pattern rules", isto é, regras do tipo "pattern", que são regras comuns a vários alvos. Estas, por sua vez, fazem uso de variáveis automáticas, definidas por símbolos especiais como `%` e `@`.

O símbolo de percentagem (`%`) num "target" indica o "pattern" a ser considerado como nome. A repetição de `%` na dependência indica que o mesmo nome deve ser considerado na dependência, como um curinga repetido. Logo,

```
%.o:  %.f
```

diz que todo arquivo cujo nome termina com `.o` depende de um arquivo com a mesma parte inicial do nome-base mas cujo nome termina com `.f`. A regra de nossa `Makefile` para efetuar a compilação apresenta duas variáveis:

`$@` (cifrão, arroba) – representa o nome completo do "target"; em nosso caso, podem ser os objetos `Decompmat.o`, `checkmats.o`, `writemat.o`, `getseed.o` ou `ranr.o`;

`$<` (cifrão, menor) – representa a primeira dependência apenas, o que é apropriado em nosso caso pois as outras são arquivos incluídos na primeira.

7. Há vários outros tipos de variáveis automáticas compreendidas pelo comando `make`. Como exercício, execute o comando

   ```
   info make
   ```

e procure por “automatic variables”. Encontre uma segunda variável que substitui todo o conjunto de dependências. Qual variável substitui apenas o nome do diretório dado no alvo? E o nome do arquivo?

8. Já vimos toda a estrutura básica da `Makefile` incluindo o uso de algumas variáveis. No entanto, essa ferramenta é bastante robusta, apresentando ainda muitos recursos. Para dar um exemplo de como podem ser as relações entre os vários alvos, criaremos uma estrutura de compilação um pouco mais sofisticada, com maior nível de organização e, consequentemente, mais complexa.

   Na nova estrutura, a compilação será feita dentro de sub-diretórios, e tentaremos automatizar a `Makefile` ao máximo. Doravante os arquivos fonte deverão permanecer dentro de um diretório `src` e os objetos, num diretório `lib`. Se os diretórios não existem, deverão ser criados; logo, temos dois alvos com suas respectivas regras:

   ```
   # Variaveis: nomes dos diretorios.
   S = src
   L = lib

   # Regras para criar os sub-diretorios.
   $(S):
           mkdir $@

   $(L):
           mkdir $@
   ```

   Tudo foi feito com variáveis para evitar repetições, pois passaremos a nos referir aos objetos e às fontes sempre acompanhados de seus respectivos diretórios.

9. Com isso, a estrutura de dependências e as regras para compilação mudam. A nova `Makefile` deve ter o aspecto que é dado a seguir.

```
# Makefile para compilacao de programa em Fortran.

# Variaveis: nomes dos diretorios.
S = src
L = lib

# Definicao dos objetos.
OBJ =              \
$(L)/Decompmat.o  \
$(L)/checkmats.o  \
$(L)/writemat.o   \
$(L)/getseed.o    \
$(L)/ranr.o

# Primeiro alvo.
all: decmat

# Regra de linkagem.
decmat: $(OBJ)
        g77 -o decmat $(OBJ)

# Demais dependencias.
$(L)/Decompmat.o: $(S)/Decompmat.f \
    $(S)/fp_def.f $(S)/cb_gen.f
$(L)/checkmats.o: $(S)/checkmats.f \
    $(S)/fp_def.f $(S)/cb_gen.f
$(L)/writemats.o: $(S)/writemats.f \
    $(S)/fp_def.f $(S)/cb_gen.f

# Regra geral de compilacao de objetos.
$(L)/%.o: $(S)/%.f $(L)
        g77 -O -c -o $@ $<

# Regras para criar os sub-diretorios.
$(S):
        mkdir $@

$(L):
```

```
        mkdir $@
```

Observe que a própria regra de compilação exige a existência do diretório `lib` para que possa ser executada.

Dê um `touch` em algum arquivo fonte e rode `make`.

Por que não é efetuada a compilação? A mensagem de erro deve dizer algo como que não encontra uma regra para criar um "target" `src/Decompmat.f`, exigido por `lib/Decompmat.o`. Isso ocorre porque informamos na `Makefile` que os novos arquivos fonte estão dentro de um diretório `src`, mas não definimos uma regra para colocá-los ali. Assim, devemos introduzir uma regra que diz como extrair arquivos `*.f` de `decmat.tgz` dentro do diretório `src`:

```
# Regra para extrair arquivos de um tar.
$(S)/%.f: decmat.tgz $(S)
        tar xmzvf $< -C $(S) $*.f
```

Nessa estrutura, todo arquivo `*.f` dentro de `src` depende da existência de `decmat.tgz` e do próprio diretório `src`. A variável `$*` carrega o valor da parte do "pattern" definido por "`%`" que é comum ao alvo e às dependências. A opção `m` no comando `tar` não preserva as datas originais dos arquivos; isso é feito para se evitar problemas de comparação das datas dos arquivos pelo `make` em caso de termos uma `Makefile` mais complexa, em que são efetuadas alterações recorrentes dos arquivos extraídos.

Rode `make` e, então, `decmat`. Dê `ls -l *` para certificar-se de que toda a estrutura está correta.

10. Agora você pode remover os arquivos `*.f`, `*.o` e `decmat` do diretório corrente. Para esse tipo de tarefa, é usual e conveniente criar-se uma regra que "limpa" a área de trabalho, eliminando todos os arquivos desnecessários e, eventualmente, até os "targets" da `Makefile`. Acrescente à sua `Makefile` a seguinte estrutura:

```
# Regra para limpar a area.
clean:
        rm -f *~ core $(S)/*~ $(L)/*.o

# Regra para limpar mais a area.
veryclean: clean
        rm -f decmat
        rm -fr $(L) $(S)

# Regra para _realmente_ limpar a area.
verycleanindeed: veryclean
        rm -f decmat.tgz
```

Observe que o alvo `verycleanindeed` depende de `veryclean`, que por sua vez depende de `clean`; por outro lado, este último é efetuado sempre que se roda o comando

`make clean`

pois `clean` não tem dependências. Logo, todos são sempre efetuados sempre que se roda `make verycleanindeed`; tome bastante cuidado com estruturas desse tipo, pois, apesar de úteis, podem, também, apagar os dados inadvertidamente em certas circunstâncias.

Execute o alvo `clean` e rode `make` novamente. O que acontece? Como você modificaria os alvos acima para que `make clean` não force a recompilação de `decmat`?

11. Vale aqui uma observação. Depois de um `veryclean` seguido de `make`, os alvos são feitos e `decmat` é gerado. Repare na última linha das mensagens; deve haver um comando para remover uma série de arquivos `*.f`. Como isso é possível se não demos tal ordem?

    Quando colocamos regras gerais ("pattern rules") na `Makefile`, ela interpreta que certos arquivos, gerados em passos intermediários por essas regras e que não foram explicitados, são dispensáveis. Assim, elimina-os depois que termina suas tarefas. Esse é o motivo pelo qual alguns passos são repetidos se executamos `make` seguidamente, ou seja, mesmo que os alvos já estejam atualizados. Para evitarmos esse tipo de conduta ou se quisermos preservar alguns dos arquivos gerados nos passos intermediários, devemos classificar esses alvos segundo uma classe especial, que corresponde a um grupo especial de nomes reservados para casos como esse, os "Special Built-in Target Names" (execute `info make` e procure esse ítem). Isso é feito com o alvo "`.PRECIOUS`", colocando-se na `Makefile` a linha

```
.PRECIOUS: arquivo1 arquivo2 etc.
```

Introduza uma variável `SRC` em sua `Makefile`, cujo valor são todos os arquivos `src/*.f` (escreva-os explicitamente). Depois, defina `$(SRC)` como "`.PRECIOUS`".

12. Vale citar outro caso de "Special Built-in Target Names". O alvo `clean` e seus semelhantes não correspondem a arquivos, bem como o alvo `all`; são apenas nomes de alvos para indicar certas regras. É possível que existam arquivos com esses mesmos nomes, por coincidência ou incidentalmente, o que interferiria na execução dos alvos. Ou, ainda, é possível que queiramos, por algum motivo excêntrico, manter arquivos com o mesmo nome do alvo, mas que sejam independentes deste. Para desvincular nomes de alvos dos respectivos arquivos, devemos caracterizar esses alvos, segundo a classe especial citada acima, como "phony targets". Isso é feito introduzindo na `Makefile` a linha

   ```
   .PHONY: all clean veryclean verycleanindeed
   ```

13. Suponhamos que você estivesse satisfeito e executado

   ```
   make verycleanindeed
   ```

   o que apaga tudo exceto a `Makefile`. No entanto, você verifica que faltaram alguns testes com o programa `decmat` e você precisa baixá-lo novamente. Para evitar esse transtorno toda vez, vamos adicionar a estrutura abaixo:

```
# Regra para obter arquivo tar pela rede.
decmat.tgz:
        wget -qN \
    http://latt.if.usp.br/fma215/materiais/aula-16/decmat.tgz
```

O comando `wget` puxa o arquivo pela rede, segundo o protocolo `http`. A opção `-q` corresponde a "quiet", isto é, sem avisos. A opção `-N` faz com que o arquivo só seja puxado pelo `wget` caso não exista ou seja mais antigo do que o arquivo que está nas páginas da disciplina, bem no espírito do `make`. Assim, nossa `Makefile` está completa e bastante automatizada. Execute o alvo `verycleanindeed`, dê um `ls -l` para verificar que a área está vazia (exceto pela `Makefile`) e, então, `make`. Finalmente, execute novamente `ls -lR` e rode `decmat` para confirmar que a operação foi bem sucedida.

3 Problemas e dicas

1. É comum querermos gerar um arquivo PS a partir de um arquivo LaTeX. Para tanto, é necessário que exista o arquivo fonte, que o compilemos para gerar um arquivo DVI e, então, façamos a conversão para deste para PS com o comando

   ```
   dvips arquivo.dvi -o arquivo.ps
   ```

   Crie uma `Makefile` capaz de realizar essas tarefas. Não se esqueça de incluir nas dependências eventuais arquivos que sejam incluídos por meio do comando `input`. Sugestão: puxe do servidor o arquivo fonte desta apostila e crie sua `Makefile`.

2. Consulte a apostila sobre arquivos "tar e backups" e desenvolva uma `Makefile` capaz de gerar tais tipos de arquivos num certo tipo de mídia (floppies ou zips), verificando antes a existência prévia desses arquivos. Use a própria capacidade de comparação de datas da `Makefile` para que ela considere ou não necessária a atualização do arquivamento. A `Makefile` deve ser capaz de montar a mídia num diretório e executar o comando `tar` convenientemente.

3. A `Makefile` que produzimos nas tarefas apresenta todos os comandos que executa. Imagine que você estivesse preparando essa `Makefile` para impressionar seus amigos e não quisesse que nenhum deles soubesse o que se passa. Como silenciar o comando `make`? Dica: consulte

"Summary of Options" em `info make`. Descubra qual é a variável especial definida na `Makefile` que cumpre essa exigência. Inspecione as outras opções e verifique qual delas permite que a `Makefile` execute tarefas paralelamente.

4. Suponhamos que você não esteja conseguindo resolver algum problema no processo de "debugging" de sua `Makefile`. Dê o comando

   `make -d >& log.txt`

   respire fundo e leia o arquivo `log.txt` para saber como `make` se comporta com sua `Makefile`. Trata-se de um *longo* arquivo.

5. Descubra como alterar a shell default da `Makefile` de `bash` para `tcsh` ou alguma outra. Dica: há uma variável na seção "Quick reference" de `info make` que trata disso.

6. *Desafio.* No ítem 4, efetuamos a modificação do arquivo `fp_def.f` para que o parâmetro lá definido fosse alterado. Complete a `Makefile` das tarefas para que ela possa alterar automaticamente esse arquivo ao lhe darmos como argumento um número. Observe que deverá haver um alvo padrão ("default") caso não forneçamos nenhum argumento. Dicas: crie um "target" extremamente genérico que dependa do arquivo `fp_def.f` e que saiba como editá-lo (provavelmente você terá de fazer uso do editor `sed`); chame a própria `Makefile` recursivamente para construir o executável. Sugestões: construa um "fake target" com o nome `Makefile` para evitar recursão infinita ou "crashes" (por quê?); dê um `touch` num arquivo de controle para evitar compilações redundantes (isso é necessário?).

4 Uma Lista de Comandos Relevantes

`info <nome-de-comando>` : mostra uma página de informação sobre o comando; para sair, basta digitar a letra `q` (de "quit").

`make` : é um programa que permite a execução automática de programas que atuam em arquivos, segundo configurações colocadas em um arquivo chamado `Makefile` no diretório de trabalho.

`wget <URL>` : puxa um arquivo pela Internet através da WWW; tanto o site da WWW quanto o arquivo são dados na `<URL>`.