Laboratórios de Informática 2016–2017
DETI - Universidade de Aveiro

Créditos Estes guiões constituem um trabalho continuado de refinamento que contou com a colaboração de João Paulo Barraca, Diogo Gomes, André Zúquete, João Manuel Rodrigues, António Adrego da Rocha, Tomás Oliveira e Silva, Sílvia Rodrigues e Óscar Pereira.

TEMA 6

Ferramentas de Automação

Objetivos:

- Automação em projetos de software
- GNU make, automake e autoconf

6.1 Automação em Projetos

Durante o desenvolvimento de um projeto de software, ou mesmo de um documento técnico em LaTeX, é frequente ter de realizar operações de forma repetitiva a cada nova versão. Um exemplo é a necessidade de compilar um programa sempre que é modificado, gerar documentação a partir de ficheiros `.tex` ou transferir novas versões de páginas HMTL para publicação num servidor web.

Considere o caso comum no desenvolvimento de uma aplicação simples em *Java*. De uma forma geral, e para evitar problemas, é necessário apagar todas as classes locais.

```
rm -f *.class
```

De seguida é necessária a compilação de todos os ficheiros de código fonte.

```
javac *.java
```

Se se pretender gerar documentação no formato *JavaDoc* será necessário:

- Criar um diretório para armazenar a documentação: **`mkdir doc`**
- Caso exista o directório, apagar o seu conteúdo: **`rm -rf doc/*`**
- Gerar a documentação: **`javadoc -d doc classname`**

Voltar a gerar a documentação exige a repetição do processo.

No caso de se utilizar LaTeX com bibliografia e acrónimos, torna-se necessário executar uma sequência tal como:

- Compilar o documento e criar a lista de índices, referências e acrónimos: **`pdlatex doc.tex`**
- Colocar as referências no texto e a lista no final: **`bibtex doc`**
- Compor o documento final com acrónimos e referências: **`pdflatex doc.tex`**
- Pode ser preciso repetir o passo pois podem ter sido introduzidas referências novas: **`pdflatex doc.tex`**

Estes passos têm de ser repetidos sempre que se altera algum ficheiro **`.tex`** ou a lista de referências ou a lista de acrónimos.

Outro problema frequente refere-se ao processo de distribuição de código fonte não compilado. Muitos programas não são autocontidos; dependem de bibliotecas, módulos ou classes externas às tradicionalmente incluídas na linguagem. Ora, como garantir que num dado sistema existem todas as dependências necessárias para compilar um certo programa? E se o programa requer um módulo M para o qual existem várias versões de fornecedores diferentes? Recusa-se a compilar se não encontrar o módulo M do fornecedor X, mesmo que o sistema já possua o módulo M do fornecedor Y?

Estas questões também se colocam com aplicações distribuídas na forma compilada, mas que usam bibliotecas partilhadas carregadas dinamicamente. É por essa razão que existem sistemas de gestão de pacotes como o **`apt-get`**, que já usou.[1]

De forma a agilizar o processo de desenvolvimento, distribuição e adaptação a sistemas heterogéneos, existem ferramentas que auxiliam o programador ou escritor técnico,

[1] O problema não se colocaria com aplicações monolíticas, mas seria um desaproveitamento de recursos. Quando quer beber vinho, tem de comprar um pacote com a garrafa, um copo e um saca-rolhas?

automatizando processos ou permitindo a deteção dos módulos existentes e a adaptação dinâmica.

Neste guião irão ser abordadas 2 funcionalidades principais:

- Automatização de tarefas.
- Deteção do ambiente para compilação e instalação.

6.2 GNU make

A ferramenta *make*[1] permite determinar que partes de um software necessitam de ser compiladas ou recompiladas após alguma alteração. É particularmente útil quando o programa é grande, porque permite reaproveitar ao máximo o trabalho que já se teve ao compilar outras partes não afetadas pela alteração. Repare que alguns programas podem demorar largos minutos ou mesmo horas a compilar todo o código fonte.

Embora o processo de compilação seja a principal aplicação do *make*, esta ferramenta pode ser aplicada a qualquer processo em que a produção de um ficheiro alvo depende de executar certas operações sobre outros ficheiros, que podem ser dados ou, por sua vez, também eles serem dependentes de outros.

Para o *make* saber quem depende de quem e o que tem de fazer para construir ou atualizar os ficheiros alvo, tem de ler essa informação de uma *makefile*. Uma *makefile* é um ficheiro de texto com uma sintaxe simples, que geralmente tem o nome **`Makefile`** e é colocado no diretório base de um projeto. Nesse caso, para compilar o projeto bastará invocar o comando **`make`** e assistir à magia da resolução de dependências.

Se não existir um ficheiro **`Makefile`**, o *make* procura outro chamado **`makefile`** e, se esse não existir, apresenta um erro.

Exercício 6.1

Crie um directório denominado **`tema6`** e desloque-se para lá.

Execute o comando **`make`** e verifique o seu resultado.

Crie um ficheiro vazio chamado **`Makefile`** e volte a executar **`make`**. Compare o resultado com a execução anterior.

Também é possível usar uma *makefile* com outro nome qualquer, executado **`make -f nome-da-makefile`**. Isto pode ser usado para ter *makefiles* diferentes para ambientes

diferentes, *Linux* vs *OS X*, por exemplo.

6.2.1 Formato de uma *makefile*

Uma makefile é composta, essencialmente, por um conjunto de *regras*, com o formato e o significado que veremos na Subseção 6.2.2. Além de regras, também pode conter:

- Definições de *variáveis*.

```
nome-da-variavel = valor
```

- Comentários, iniciados por um símbolo **#** e que se prolongam até ao fim da linha.

```
# comentario ignorado pelo make
```

- Diretivas de inclusão de outra makefile. Podem usar-se para organizar as regras de forma modular.

```
include outra-makefile
```

6.2.2 Regras

As regras são a parte fundamental de uma makefile. Cada regra tem o seguinte formato:

```
alvo ... : pré-requisito ...
        receita-para-criar-alvo
        ...
```

O *alvo* é um ficheiro a ser gerado ou atualizado. Também se podem definir alvos falsos (*phony targets*), que se usam para invocar ações e que não vão corresponder a ficheiros. A seguir ao símbolo **:** vem uma lista de zero ou mais *pré-requisitos* ou dependências. Estes são os ficheiros de que o alvo depende. A *receita* é uma sequência de comandos de shell que permitirão gerar o alvo a partir dos pré-requisitos. Note que as linhas da receita têm de começar com um caráter TAB.

Quando o **make** é executado, procura a primeira regra da makefile e verifica-a: se algum pré-requisito é mais recente do que o alvo (ou algum dos alvos), então executa a receita. Porém, antes disso verifica se é preciso atualizar cada um dos pré-requisitos, procurando e verificando regras em que sejam alvos. Aplicado recursivamente, este processo garante que todos os alvos que dependam direta ou indiretamente de uma alteração serão atualizados.

Por exemplo, uma makefile para compilar um pequeno programa *Java*, poderia conter apenas uma regra:

```
Programa.class: Programa.java
	javac Programa.java
```

Esta makefile declara que o alvo **`Programa.class`** depende do pré-requisito **`Programa.java`** e que é possível criar (ou atualizar) o primeiro executando a receita **`javac Programa.java`**. Assim, bastará correr **`make`** (ou **`make Programa.class`**) para produzir o programa executável. Correr o *make* de novo não vai re-executar a receita, porque o alvo é mais recente que o pré-requisito. Vai simplesmente produzir a mensagem abaixo.

```
make: 'Programa.class' is up to date.
```

Só depois introduzir alguma alteração ao ficheiro **`Programa.java`** é que o comando **`make`** voltará a ter efeito.

Escrever receitas de compilação é simples, no entanto têm de ser respeitadas regras e boas práticas.

- A receita tem de ser indentada com TAB. Convém usar um editor de texto que preserve esses carateres.
- A receita pode estar vazia. Nesse caso, o *make* atualiza os pré-requisitos da regra e nada mais. (Isto é frequente em *phony targets*.)
- Se uma linha for demasiado longa, pode ser dividida utilizando o caráter `\`.
- Podem ser utilizadas variáveis nas receitas através da sintaxe **`$(VARNAME)`**.
- Se uma instrução da receita começar com o caráter **`@`**, é executada normalmente, mas não é apresentada pelo *make*. Por exemplo **`@echo LABI`** irá apresentar a mensagem **`LABI`**, mas a instrução em si não é apresentada.
- As receitas são executadas de forma sequencial. Se uma instrução falhar, a execução da receita é abortada. Pode evitar-se isso se a instrução for precedida de um símbolo `-`; mesmo que falhe, as restantes instruções serão tentadas.
- É possível executar receitas de que invocam outras receitas, mesmo que estas estejam noutros directórios.

O exemplo seguinte ilustra alguns dos aspetos descritos.

```
regra-vazia: Programa.class
```

```
Programa.class: Programa.java
        @echo \
          "Compilar ficheiro"
        -rm -f *.class        # avança, mesmo que falhe
        javac Programa.java
```

6.2.3 Alvos falsos

Uma regra pode ter um alvo que não corresponde a um ficheiro. Diz-se que é um *alvo falso* ou *phony target*. É meramente um nome que pode ser usado para invocar explicitamente uma certa receita. Como a receita não vai criar um ficheiro com esse nome, será executada sempre que o alvo for invocado. Para garantir que o aparecimento casual de um ficheiro com o mesmo nome do alvo não impede que esse alvo seja invocado, pode declarar-se explicitamente que se trata de um alvo falso usando a sintaxe:

```
.PHONY: alvo-falso
```

É muito usual as makefiles terem pelo menos dois alvos falsos, que se tornaram standards de facto:

`all` Executa um processo completo. Por exemplo a compilação de todos os ficheiros da aplicação e a geração do ficheiro executável final.

`clean` Limpa todos os ficheiros temporários ou auxiliares, deixando apenas ficheiros de código fonte.

Por exemplo, poderíamos reescrever a makefile como se segue.

```
.PHONY: all clean

all: Programa.class

Programa.class: Programa.java clean
        @echo "Compilar ficheiro"
        javac Programa.java

clean:
        rm -f *.class
```

Repare que o alvo **`all`** depende de todos executáveis do projeto (neste caso apenas um) e nem precisa de receita. Foi colocado como primeira regra, para ser o alvo por defeito.

O alvo **clean** limpa todos os ficheiros **.class** e neste caso também foi colocado como pré-requisito da regra anterior para forçar uma limpeza antes da compilação.

Exercício 6.2

Dentro do directório **tema6**, crie um sub-directório chamado **doc-latex**.

Neste directório coloque um ficheiro LaTeX muito simples chamado **doc.tex**.

Crie uma **Makefile** com alvos para compilar tudo e para limpar os ficheiros temporários.

Verifique que consegue invocar os alvos e confira os resultados.

Exercício 6.3

Altere a **Makefile** que criou anteriormente de forma a aproveitar o mecanismo de resolução de dependências. Neste caso, a compilação principal deverá depender de um alvo que cria os índices (**pdflatex nome.tex**) e de outro que compila a bibliografia (usando **bibtex**). Insira uma pequena bibliografia para testar.

Crie uma dependência na compilação para a limpeza dos ficheiros auxiliares e verifique que funciona.

Exercício 6.4

Crie uma regra chamada **publish** que envie (utilizando **scp**) o documento produzido para a sua área no servidor *xcoa.av.it.pt*.

Tenha em atenção que esta regra só deverá ser executada caso a compilação tenha sido previamente executada.

Podemos aproveitar a capacidade do *make* detetar a modificação de ficheiros para invocar

a compilação apenas se algum ficheiro com código fonte for alterado.

Exercício 6.5

Sabendo que o ficheiro **.bib** é convertido para um ficheiro **.bbl** depois de um ficheiro **.aux** ser criado, que o documento final depende do ficheiro **.tex**, dos índices (ex. **.toc**) e da bibliografia (**.bbl**), re-escreva as regras da makefile de forma a utilizarem o nome dos ficheiros e assim usufruir da detecção de alterações.

Verifique que necessita de alterar o ficheiro **.tex** para que a compilação seja repetida.

6.2.4 Uso de variáveis

É possível definir e usar variáveis numa makefile. Pode usar-se esta facilidade para evitar repetição de listas em múltiplas regras. O exemplo que se segue atribui uma lista de alvos à variável **objects**, usa o seu valor como lista de dependências do alvo **all** e como lista de ficheiros a remover na receita de **clean**.

```
objects = A.class B.class C.class

all: $(objects)

clean:
        rm -f $(objects)
#...
```

Também existem variáveis especiais cujo valor é atribuído automaticamente em cada regra avaliada pelo *make*. Chamam-se *variáveis automáticas* e as mais importantes são:

$@ representa o alvo da regra;

$< representa o primeiro pré-requisito;

$^ representa a lista de todos os pré-requisitos;

$? representa a lista dos pré-requisitos mais recentes que o alvo;

$* representa a string que coincidiu com o padrão **%** numa regra implícita (ver abaixo).

Por exemplo, a regra seguinte cria um ficheiro **abc** por concatenação de três ficheiros **one**, **two**, **three**, mas evita repetir os nomes dos ficheiros.

```
abc: one two three
	cat $^ > $@
```

6.2.5 Regras implícitas

É possível definir regras gerais, que se aplicam a ficheiros cujos nomes satisfaçam certos padrões. Chamam-se regras implícitas de padrão e são caracterizadas por conterem um símbolo **%** no alvo. Por exemplo, a regra

```
%.class: %.java
	javac $<	# $< representa o pré-requisito
```

especifica como se pode obter um ficheiro objeto de java (**.class**) por compilação do respetivo ficheiro fonte (**.java**). Com esta makefile, se invocarmos **make A.class** e existir um ficheiro **A.java**, o *make* irá ser executar o comando **javac A.java**.

6.2.6 Sub-Directórios

Frequentemente os código fonte de um programa encontra-se dividido por módulos, ou pacotes, colocados em directórios diferentes. Ora, para compilar todo o programa é necessário compilar todos os seus módulos pela ordem correta. O sistema *make* suporta este processo através de um mecanismo que permite invocar uma makefile a partir de outra makefile.

Exercício 6.6

No directório **tema6** crie um directório **src** e coloque algum código java que possua de outra disciplina.

Crie uma **Makefile** que permita compilar este código através do alvo *all*.

Não se esqueça de criar também o alvo *clean*.

Dada a estrutura criada, deverá ter um directório **tema6** com dois sub-directórios **doc** e **src**. Seria interessante que uma **Makefile** no directório pai pudesse invocar a compilação do código fonte e a geração da documentação. As linhas seguintes demonstram como tal pode ser feito. Neste caso, é necessário utilizar a palavra reservada **.PHONY** para garantir que os alvos são processados apesar de já existirem diretórios com esses nomes.

```
SUBDIRS = doc src

.PHONY: subdirs $(SUBDIRS)

subdirs: $(SUBDIRS)

$(SUBDIRS):
	$(MAKE) -C $@
```

Exercício 6.7

Adapte a estrutura demonstrada anteriormente de forma a que possa propagar a compilação e a limpeza (*clean*) para o código e para a documentação.

6.3 GNU Automake e Autoconf

O *automake* [2] é uma ferramenta que, baseando-se no *make* possibilita a gestão da compilação para programas mais complexos, e combinado com a ferramenta *autoconf* [3] facilita a distribuição dos programas na sua forma de código fonte. Em particular, possibilita que os ficheiros **`Makefile`** sejam construídos de forma dinâmica, adaptando-se às condições existentes (ex. localização de ficheiros ou programas) no sistema alvo. É ainda útil para que durante esta adaptação ao sistema alvo, seja possível verificar a existência de dependências sem os quais o programa não poderia ser compilado. Por exemplo, para os programas desenvolvidos nas disciplinas iniciais da Universidade de Aveiro é necessário que exista o *Java* na sua versão 1.6. Para os relatórios de Laboratórios de Informática é necessário que exista o LaTeX com suporte para **`pdflatex`** e todos os packages declarados no documento.

De uma forma muito simples pode-se considerar que o sistema necessita de dois tipos de ficheiros: Templates para ficheiros **`Makefile`** com nome **`Makefile.am`** e uma configuração de configuração com nome **`configure.ac`**. O primeiro é utilizado para facilitar a escrita de ficheiros **`Makefile`** complexa e será abordado na Subseção 6.3.2, enquanto o segundo é utilizado para validar a integridade do sistema e existência de todas as dependências e será abordado na Subseção 6.3.1.

6.3.1 GNU Autoconf

O ficheiro **`configure.ac`** contém uma série de macros que indicam que validações executar e como agir face a esse resultado. Pode considerar que os macros irão efetuar validações no sistema (ex. existência de *Java*) ou determinar parâmetros operacionais (ex. localização do compilador **`javac`** e sua versão). Da perspectiva de um aluno isto é

extremamente importante pois permite enviar código (p.ex. para o docente) junto com a validação de que o ambiente de avaliação tem os componentes necessário à correta execução.

No início deste ficheiro, terá de ser declarado qual o nome do pacote de software, a sua versão e qual o ficheiro principal de código fonte. O exemplo seguinte demonstra o que seria esperado para um qualquer programa *Java* a realizar como parte do pacote *tema6*. Também é utilizada a macro `AC_PROG_GCJ` pois vamos utilizar uma aplicação *Java*.

```
AC_INIT([tema6], [0.1])
AM_INIT_AUTOMAKE
AC_CONFIG_SRCDIR([src/Foo.java])
AM_PROG_GCJ
```

Após a criação deste ficheiro é necessário inicializar o sistema *autoconf*. Para isso é necessário executar os commandos:

- **`aclocal`**: Cria uma base de dados local para utilização no projeto. Esta base de dados é construída com base nas macros que se encontrem definidas no ficheiro **`configure.ac`**.

- **`automake -a`**: Processa os ficheiros **`Makefile.am`** e gera ficheiros adicionais necessários para a instalação.

- **`autoconf`**: Gera o *script* **`configure`**.

Exercício 6.8

Na raíz da sua área de trabalho crie uma pasta chamada **tema6-auto**. Lá dentro crie dois directórios: **src** e **doc**. No directório **src** coloque um qualquer programa em *Java*[a].

No directório **tema6-auto** crie um ficheiro chamado **configure.ac** com as macros apresentadas anteriormente.

Execute **aclocal**, **automake -a** e **autoconf**. Verifique que existe um ficheiro chamado **configure**.

Execute o ficheiro **configure**.

[a]Caso não tenha um programa disponível, escreva um que imprima uma mensagem fixa tal como "Laboratórios de Informatica"

Até agora o script **configure** apenas irá realizar algumas verificações muito básicas e incompletas para qualquer projeto. É necessário adicionar macros de forma a configurar o projeto para que este possa ser produzido no sistema. A sequência de macros a utilizar varia para cada projeto de acordo com as suas necessidades. Neste caso em concreto, será necessário realizar validações em relação à existência de compiladores *Java* na versão correta e do sistema de produção de documentos LaTeX.

A macro **AC_CHECK_PROG** permite verificar a existência de programas específicos no sistema e segue a sintaxe seguinte:

```
AC_CHECK_PROG (variável, programa-a-verificar, se-encontrado,
      se-não-encontrado, caminho, ignorar)
```

Em que:

- `variável`: o nome de uma variável onde se armazena o resultado do teste.
- `programa-a-verificar`: o nome do programa a verificar (ex. **javac**).
- `se-encontrado`: que valor deverá a variável ter caso o programa seja encontrado.
- `se-não-encontrado`: que valor deverá a variável ter caso o programa não seja encontrado (opcional).
- `caminho`: directórios onde procurar (opcional).

- `ignorar`: localizações que se ignoram. Serve para excluir versões antigas ou conhecidas por não funcionarem como necessário.

A variável pode depois ser utilizada numa condição para agir de forma adequada. Aplicado ao caso do **`javac`** podemos definir:

```
AC_CHECK_PROG(EXISTE_JAVAC,javac,yes)

if test "$EXISTE_JAVAC"; then
        AC_MSG_NOTICE([[Compilador de Java encontrado.])
else
        AC_MSG_ERROR([[Compilador de Java em falta.])
fi
```

Repare como a variável `$EXISTE_JAVAC` foi utilizada numa condição para informar o utilizador que o compilador foi encontrado (`AC_MSG_NOTICE`), ou para abortar a configuração com uma mensagem (`AC_MSG_ERROR`) de erro. Caso o programa **`javac`** não exista, o resultado deverá ser o seguinte:

```
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking for javac... no
configure: error: Compilador de Java em falta.
```

Exercício 6.9

Utilizando a metodologia anterior escreva no seu ficheiro **`configure.ac`** todas as verificações que acha necessárias para construir devidamente a documentação e o código fonte *Java*.

Depois terá de voltar a executar os comandos **`aclocal`**, **`autoconf`** e **`automake`**.

Verifique a execução do novo **`configure`** gerado.

Além de se saber que existem as aplicações necessárias, também é necessário saber se as aplicações possuem a versão correta, ou são capazes de suportar as funcionalidades pretendidas. Até agora, verificou-se que existe LaTeX e *Java* mas não é sabido se o sistema consegue realmente compilar documentos, ou se a versão de *Java* é a correta. Determinar a versão de uma aplicação implica utilizar funcionalidades da **bash**, nomeadamente executar comandos e filtrar o seu resultado.

```
AC_MSG_CHECKING([Verificando versao do Java])
JAVA_VERSION=$(java -version 2>&1 |grep "java version" |cut -d '.' -f 2)
AC_MSG_RESULT($JAVA_VERSION)

if test $JAVA_VERSION -lt 6; then
        AC_MSG_ERROR([Necessario Java versao 6])
fi
```

No exemplo anterior o comando **grep** filtra as linhas da execução do comando **java -version** de forma a obter apenas a linha com informação de versão. O comando **cut -d '.' -f 2** separa esta linha pelo caráter "**.**" e devolve o segundo item. Repare que na comparação não é utilizado o operador "**<**" como está habituado a utilizar, mas sim o operador "**-lt**" que possui o mesmo significado: *less than*. Isto é uma particularidade da **bash** que utiliza "**-lt**" quando os operandos são inteiros e "**==**" quando os operadores são sequências de caráters (Strings).

Exercício 6.10

Execute o comando **java -version** e analise o que é impresso no ecrã e reconstrua o comando do exemplo.

Adicione ao seu **configure.ac** as verificações necessárias para verificar que possui *Java* versão 6 e **pdflatex** versão 2.4.

Também é possível verificar pela existência de bibliotecas específicas, ou no caso de se utilizar LaTeX a existência de *packages*. No entanto, para o caso de LaTeX ou *Java* não existem macros que realizem estas funções automaticamente, pelo que se torna necessário utilizar comandos externos.

Para verificar se existe um *package* de *Java* pode-se utilizar o comando **kpsewhich nome-da-package.sty** e observar o seu resultado. Para isto recorre-se à macro

AC_MESSAGE_CHECKING para indicar uma mensagem de um teste personalizado.

```
AC_MSG_CHECKING(Verificando existencia do package biblatex)
HAVE_BIBLATEX=$(kpsewhich biblatex.sty)
AC_MSG_RESULT($HAVE_BIBLATEX)
if test "x$HAVE_BIBLATEX" == "x"; then
        AC_MSG_ERROR(Falta package biblatex)
fi
```

Esta fórmula para escrever testes pode ser utilizada para qualquer validação que se pretenda. Por exemplo, para verificar se existem ficheiros em localizações específicas.

Exercício 6.11

Adicione testes ao seu **configure.ac** de forma a validar a existência das *packages* que utiliza.

Também pode ser importante validar se o sistema consegue realmente produzir um produto final pretendido. Neste caso será a documentação e a aplicação. Para isso é possível definir testes que efectivamente compilam pequenos programas de forma a validar o ambiente de compilação. Estes testes podem validar a existência de *packages* ou de *classes*. O exemplo seguinte valida se é possível compilar aplicações *Java* e se é possível utilizar a o método **exit** da classe *java.lang.System*.

```
# Preparar teste
AC_MSG_CHECKING(Verificando possibilidade de compilar programas Java)
    cat <<__EOF__ >conftest.java [
import static java.lang.System.*;

public class conftest {
  public static void main(String[] args) {
    exit(0);
  }
}]
__EOF__

# Compilar
javac conftest.java

# Testar resultado
if test $? = 0; then
```

```
        AC_MSG_RESULT([ok])
else
        AC_MSG_ERROR([ERRO])
fi

# Limpar ficheiros produzidos.
rm -f conftest.java conftest.class
```

Exercício 6.12

Construa testes de forma a verificar se o sistema é capaz de gerar documentos com os *packages* necessários e compilar aplicações *Java*.

Por fim, caso todos os testes tenha completado corretamente, é necessário gerar os ficheiros **`Makefile`** para compilar o programa. Embora não tenha sido abordado neste guião, é possível adaptar os **`Makefile`** de acordo com o que foi sendo detetado na execução do *script* **`configure`**.

```
AC_CONFIG_FILES([
 Makefile
 src/Makefile
 doc/Makefile
])
AC_OUTPUT
```

Exercício 6.13

Adicione ao seu ficheiro **`configure.ac`** a indicação para se gerarem os ficheiros **`Makefile`** necessários.

Verifique que ele o tenta fazer. Tenha em atenção que ainda não deu indicações de como construir estes ficheiros.

6.3.2 GNU Automake

A ferramenta *automake* encaixa na *autoconf* permitindo definir ficheiros **`Makefile`** de uma forma mais simplificada e adaptável ao sistema alvo. Neste caso são utilizados ficheiros **`Makefile.am`**, que a ferramenta converte para ficheiros **`Makefile.in`**. O *script* **`configure`** irá utilizar estes ficheiros para gerar os ficheiros **`Makefile`** finais.

Para o exemplo seguido é necessária a criação de um ficheiro **Makefile.am** na raíz do projeto indicando que sub-diretórios existem, seguido de ficheiros **Makefile.am** em cada directório. O conteúdo do ficheiro **Makefile.am** na raíz do projeto seria:

```
SUBDIRS = src doc
```

De realçar que a ferramenta *automake* também necessita que existam vários ficheiros de texto na raíz do projeto. Estes ficheiros não são necessários para os testes ou a compilação mas são obrigatórios de existir a quando da utilização desta ferramenta.

Exercício 6.14

Crie um ficheiro **Makefile.am** na raíz do seu projeto e outros em cada um dos directórios. Estes podem ser vazios.

Execute o comando **automake** e crie os ficheiros em falta. O conteúdo não é relevante. No final deverá obter vários ficheiros **Makefile.in** e o **automake** não deverá mostrar qualquer erro.

Verifique que o *script* **configure** gera ficheiros **Makefile** e inspecione o seu conteúdo.

Depois de existirem ficheiros **Makefile.am** o *script* **configure** irá gerar ficheiros **Makefile** com bastante informação e uma grande panóplia de alvos. Para além dos típicos *all* e *clean*, outros como *distclean* ou *install* são igualmente criados.

Exercício 6.15

Verifique que alvos são criados. Verifique por exemplo qual a utilidade do alvo *distclean*.

Exercício 6.16

Em cada um dos ficheiros **Makefile.am** adicione regras para os alvos *all* e *clean*, de forma a compilar e gerar documentação.

Na raíz do projeto execute **make**. Que observou?

Uma funcionalidde de um dos alvos é a de criar um pacote *.tar.gz* para distribuição imediata, contendo todo o programa e documentação[2]. Para isto é apenas necessário que se declarem os ficheiros a incluir neste pacote. Neste exemplo, para incluir o código fonte será necessário adicionar a seguinte informação ao ficheiro **`src/Makefile.am`**:

```
bin_PROGRAMS = foo
foo_SOURCES = Foo.java

%.class: %.java
        javac $*.java

foo : Foo.class

all: foo
```

Exercício 6.17

Corrija os seus ficheiros **`Makefile.am`** de forma a que ao executar **`make dist`** seja incluído todo o projeto num ficheiro *.tar.gz*.

Para incluir a documentação, outro directório ou ficheiro qualquer, é necessário que seja definida a variável `EXTRA_DIST` no ficheiro **`Makefile.am`** principal. Neste caso:

```
EXRA_DIST = doc/doc.tex
```

Exercício 6.18

Adicione o directório de documentação à lista de directórios a incluir no ficheiro *.tar.gz*.

Invoque **`make dist`** e verifique que funciona.

[2]Para verificar o conteúdo execute: **`tar -ztf nome-do-ficheiro.tar.gz`**

Referências

[1] GNU Project (FSF), *GNU Make*, `http://www.gnu.org/software/make/`, [Online; acedido em 31 de Outubro de 2016], 2013.

[2] ——, *GNU Automake*, `http://www.gnu.org/software/automake/`, [Online; acedido em 31 de Outubro de 2016], 2013.

[3] ——, *GNU Autoconf*, `http://www.gnu.org/software/autoconf/`, [Online; acedido em 31 de Outubro de 2016], 2013.