Das Programmierwerkzeug make

Einführung in die Rechnernutzung am GPI

Thomas Forbriger

GPI, Karlsruhe & BFO, Schiltach

Juli 2009

Die klassische Aufgabe

Ein kleines Projekt

Zutaten eines kleines Softwareprojekts:

```
thof@lizzy:~> ls -l polynom_discriminant.c polynom_roots.c polynom_iscomplex.c poly
-rw-r--r-- 1  2015 poly.c
-rw-r--r-- 1  2015 polynom.h
-rw-r--r-- 1  2015 polynom_discriminant.c
-rw-r--r-- 1  2015 polynom_iscomplex.c
-rw-r--r-- 1  2015 polynom_printpolynomial.c
-rw-r--r-- 1  2015 polynom_roots.c
```

Erzeugung des Programms `poly`:

```
thof@lizzy:~> gcc -Wall -c -o polynom_roots.o polynom_roots.c
thof@lizzy:~> gcc -Wall -c -o polynom_iscomplex.o polynom_iscomplex.c
thof@lizzy:~> gcc -Wall -c -o polynom_printpolynomial.o polynom_printpolynomial.c
thof@lizzy:~> gcc -Wall -c -o polynom_discriminant.o polynom_discriminant.c
thof@lizzy:~> gcc -Wall -c -o poly.o poly.c
thof@lizzy:~> gcc -o poly poly.o polynom_*.o -lm
```

Die klassische Aufgabe

Ein kleines Projekt

Wird `polynom_iscomplex.c` verändert, so muss diese Datei neu kompiliert und anschließend `poly` neu gelinkt werden.

Wird `polynom.h` verändert, so müssen möglicherweise alle Dateien kompiliert und anschließend wieder gelinkt werden.

Die klassische Aufgabe

Ein großes Projekt

Große Projekte können aus über hundert Quelldateien in unterschiedlichen Verzeichnissen bestehen. Darunter sind viele Header-Dateien, die von unterschiedlichen Quelldateien eingebunden werden und sich auch gegenseitig einbinden. Wird eine der Dateien geändert, ist es möglicherweise unüberschaubar, welche Teile des Codes neu übersetzt und gelinkt werden müssen.

1. Lösung: Einfach immer alles übersetzen und linken (zeitraubend und deshalb ineffizient).
2. Lösung: Eine Software verwenden, die entscheiden kann, welche Befehle nach einer Änderung in welcher Reihenfolge ausgeführt werden müssen.

IDE, CASE und `make`

Große Programmpakete mit integriertem Editor und interaktivem Quellcode-Browser sind bekannt unter den Bezeichnungen

IDE Integrated development environment (z.B. *Eclipse*)

CASE Computer aided software engineering

Unter UNIX/Linux steht das sehr viel mächtigere, aber auf die Kommandozeile orientierte Werkzeug `make` zur Verfügung. Wir geben einfach ein:

```
thof@lizzy:~> make poly
gcc -ansi -Wall -o poly poly.o polynom_roots.o polynom_iscomplex.o polynom_discrim
```

Woher weiß `make` was es tun muss?

Beim Aufruf des Programms `make` liest dieses automatisch die Datei `Makefile`. In dieser stehen alle relevanten Angaben:

`Makefile` (lines 40 to 48):

```
[...]
# the polynomial example in C
polynom_roots.o: polynom.h polynom_roots.c
polynom_printpolynomial.o: polynom.h polynom_printpolynomial.c
polynom_discriminant.o: polynom.h polynom_discriminant.c
polynom_iscomplex.o: polynom.h polynom_iscomplex.c
poly.o: polynom.h poly.c
poly: poly.o polynom_roots.o polynom_iscomplex.o polynom_discriminant.o \
  polynom_printpolynomial.o
        $(CC) $(CFLAGS) -o $@ $^ -lm
[...]
```

Im Folgenden werden wir ein Makefile Stück für Stück zusammenstellen.

`make` und `gmake`

Die GNU Version von `make` bietet wesentlich mehr Funktionalität als das Standard `make`. Um klarzustellen, dass für ein `Makefile` GNU `make` benötigt wird, wird GNU make häufig mit `gmake` bezeichnet. In dieser Vorlesung wird mit `make` immer GNU `make` gemeint.

Die wichtigsten Konzepte für ein `Makefile`

Rules

Folgende elementare Sequenz wird als *Rule* bezeichnet:

```
target: prerequisite prerequiste...
<tab>command
```

Target zu erzeugende Datei

Prerequisite Datei aus der ein *Target* generiert wird

Command Befehl, der ausgeführt werden muss, um ein *Target* zu erzeugen; beachte das <tab> Zeichen am Beginn der Zeile

Rule gibt die Abhängigkeiten (*Dependencies*) zwischen *Target* und *Prerequisites* an; das Kommando ist optional; es kann mehrere *Rules* zu einem *Target* geben, aber nur eine darf ein *Command* enthalten

Die wichtigsten Konzepte für ein `Makefile`

Ein einfaches Beispiel

`Makefile` (lines 61 to 67):

```
[...]
# the polynomial example in Fortran
polyf.o: polyf.f
        gfortran -c -o polyf.o polyf.f
polynomf.o: polynomf.f
        gfortran -c -o polynomf.o polynomf.f
polyf: polyf.o polynomf.o
        gfortran -o polyf polyf.o polynomf.o
[...]
```

Falls seit der letzten Übersetzung `polyf.f` verändert wurde, erkennt `make`, dass `polyf.o` neu erzeugt werden muss. Da `polyf.o` dann neuer ist als `polyf`, erzeugt `make` anschließend automatisch das Binary durch den entsprechenden Befehl.

Die wichtigsten Konzepte für ein `Makefile`

Ein einfaches Beispiel

Der Beweis:

```
thof@lizzy:~> make polyf
make[1]: 'polyf' is up to date.
thof@lizzy:~> touch polyf.f
thof@lizzy:~> make polyf
gfortran -c -o polyf.o polyf.f
gfortran -o polyf polyf.o polynomf.o
```

Die wichtigsten Konzepte für ein `Makefile`

Die Verwendung von *Pattern Rules*

Da das *Command* zum Übersetzen einer Quellcodedatei immer gleich aussieht und anhand der Dateiendung die Programmiersprache erkannt werden kann, bietet `make` die Möglichkeit *Rules* mit gleichem *Command* einmal allgemein als *Pattern Rule* zu definieren. `Makefile` (lines 69 to 76):

```
[...]
# the polynomial example in C++
polyxx.o polynomxx.o: polynomxx.h
polyxx.o: polyxx.cc
polynomxx.o: polynomxx.cc
%.o: %.cc
        g++ $(CXXFLAGS) -c -o $@ $<
polyxx: polyxx.o polynomxx.o
        $(CXX) $(CXXFLAGS) -o $@ $^ -lm
[...]
```

Die wichtigsten Konzepte für ein `Makefile`

Die Verwendung von *Pattern Rules*

Der Beweis:

```
thof@lizzy:~> make polyxx
make[1]: 'polyxx' is up to date.
thof@lizzy:~> touch polynomxx.h
thof@lizzy:~> make polyxx
g++ -ansi -Wall -c -o polyxx.o polyxx.cc
g++ -ansi -Wall -c -o polynomxx.o polynomxx.cc
g++ -ansi -Wall -o polyxx polyxx.o polynomxx.o -lm
```

Die wichtigsten Konzepte für ein `Makefile`

Die Verwendung von *built-in implicit Rules*

Man muss `make` nicht mitteilen, mit welchem *Command* eine Object-Datei aus einer C Quellcodedatei erzeugt wird. Die wichtigsten *Rules* sind bereits in `make` fest eingebaut und können mit

```
make -p
```

abgefragt werden.

Variablen, bedingte Ausführung, etc.

In `Makefile`s können Variablen verwendet werden. Deren Inhalt kann beim Aufruf von `make` auf der Kommandozeile übergeben werden.

Es gibt Konstrukte wie `if-then-else` zur Programmierung bedingter Anweisungen.

Ein `Makefile` kann ein anderes einbinden (per `include`).

Und vieles mehr...

Variablen

Im `Makefile` können Variablen gesetzt und verwendet werden:

`Makefile` (lines 17 to 24):

```
[...]
# define standard compiler flags
CFLAGS=-ansi -Wall
CXXFLAGS=-ansi -Wall
FFLAGS=-Wall
# define standard compilers
CC=gcc
CXX=g++
FC=gfortran
[...]
```

Diese können beim Aufruf von `make` mit einem anderen Inhalt belegt werden:

```
thof@lizzy:~> make poly CFLAGS=-pedantic
gcc -pedantic -o poly poly.o polynom_roots.o polynom_iscomplex.o polynom_discriminant.o polynom_printpolynomia
```

Automatisches Erzeugen der *Rules*

Der GNU C Compiler `gcc` ist in der Lage die *Rules* für C Quellcode automatisch zu generieren. Das gleiche gilt für den C++ Compiler.

```
thof@lizzy:~> gcc -M poly.c
poly.o: poly.c /usr/include/stdc-predef.h polynom.h /usr/include/stdio.h \
 /usr/include/features.h /usr/include/sys/cdefs.h \
 /usr/include/bits/wordsize.h /usr/include/gnu/stubs.h \
 /usr/include/gnu/stubs-64.h \
 /usr/lib64/gcc/x86_64-suse-linux/4.8/include/stddef.h \
 /usr/include/bits/types.h /usr/include/bits/typesizes.h \
 /usr/include/libio.h /usr/include/_G_config.h /usr/include/wchar.h \
 /usr/lib64/gcc/x86_64-suse-linux/4.8/include/stdarg.h \
 /usr/include/bits/stdio_lim.h /usr/include/bits/sys_errlist.h \
 /usr/include/stdlib.h /usr/include/bits/waitflags.h \
 /usr/include/bits/waitstatus.h /usr/include/endian.h \
 /usr/include/bits/endian.h /usr/include/bits/byteswap.h \
 /usr/include/bits/byteswap-16.h /usr/include/sys/types.h \
 /usr/include/time.h /usr/include/sys/select.h /usr/include/bits/select.h \
 /usr/include/bits/sigset.h /usr/include/bits/time.h \
 /usr/include/sys/sysmacros.h /usr/include/bits/pthreadtypes.h \
 /usr/include/alloca.h /usr/include/bits/stdlib-float.h
```

Automatisches Erzeugen der *Rules*

Auch diese Aufgabe kann `make` übernehmen. Eine entsprechende *Rule* mit *Command* sieht wie folgt aus:

`Makefile` (lines 51 to 54):

```
[...]
%.d: %.c
        @$(SHELL) -ec '$(CC) -M $(CPPFLAGS) $< \
        | sed '\''s,\($*\)\.o[ :]*,\1.o $@ : ,g'\'' > $@; \
        [ -s $@ ] || rm -f $@'
[...]
```

Automatisches Erzeugen der *Rules*

Damit erzeugt `make` eine *Rule*, die auch beinhaltet, von welchen Dateien die *Rule*-Datei selber abhängt:

```
thof@lizzy:~> cat poly.d
poly.o poly.d : poly.c /usr/include/stdc-predef.h polynom.h /usr/include/stdio.h \
 /usr/include/features.h /usr/include/sys/cdefs.h \
 /usr/include/bits/wordsize.h /usr/include/gnu/stubs.h \
 /usr/include/gnu/stubs-64.h \
 /usr/lib64/gcc/x86_64-suse-linux/4.8/include/stddef.h \
 /usr/include/bits/types.h /usr/include/bits/typesizes.h \
 /usr/include/libio.h /usr/include/_G_config.h /usr/include/wchar.h \
 /usr/lib64/gcc/x86_64-suse-linux/4.8/include/stdarg.h \
 /usr/include/bits/stdio_lim.h /usr/include/bits/sys_errlist.h \
 /usr/include/stdlib.h /usr/include/bits/waitflags.h \
 /usr/include/bits/waitstatus.h /usr/include/endian.h \
 /usr/include/bits/endian.h /usr/include/bits/byteswap.h \
 /usr/include/bits/byteswap-16.h /usr/include/sys/types.h \
 /usr/include/time.h /usr/include/sys/select.h /usr/include/bits/select.h \
 /usr/include/bits/sigset.h /usr/include/bits/time.h \
 /usr/include/sys/sysmacros.h /usr/include/bits/pthreadtypes.h \
 /usr/include/alloca.h /usr/include/bits/stdlib-float.h
```

Automatisches Erzeugen der *Rules*

In der Regel wird dann noch ein Kommando eingefügt, welches alle relevanten *Rule*-Dateien automatisch generiert und einliest.

`Makefile` (lines 56 to 58):

```
[...]
POLYSRC=polynom_roots.c polynom_printpolynomial.c \
  polynom_discriminant.c  polynom_iscomplex.c poly.c
-include $(patsubst %.c,%.d,$(POLYSRC))
[...]
```

Phony Targets

In manchen Fällen ist es hilfreich *Targets* zu definieren, die keine Datei erzeugen. Diese *Targets* werden als *phony* bezeichnet. In der Regel wird ein *Target* `clean` definiert, um Zwischenergebnisse (durch `make clean`) wieder zu löschen:

`Makefile` (lines 34 to 38):

```
[...]
.PHONY: clean
clean: ;
        -find . -name \*.bak | xargs --no-run-if-empty /bin/rm -v
        -/bin/rm -vf flist a.out *.s *.ps *.hex *.dasm *.o *.d
        -/bin/rm -vf poly polyxx polyf polyf2c polyf.c polynomf.c
[...]
```

Dokumentation

Auf den Linux-Rechnern ist eine ausführliche Dokumenation zu `make` abrufbar:

- `info make`
- `pinfo make`
- `tkinfo make`
- `man make`

Weitere Dokumentation ist auf dem Internet verfügbar (siehe Web-Seite zur Vorlesung).

`make`: nicht nur beim Programmieren

`make` kann überall eingesetzt werden, wo Dateien nach einem standardisierten Schema aus dem Inhalt anderer Dateien erzeugt werden. Das kann zum Beispiel die Verarbeitung seismologischer Daten zu sein. Ein großer Vorteil von `make` besteht darin, dass mit dem Erstellen des `Makefile`s gleichzeitig die Datenverarbeitung dokumentiert wird. Es ist dann ausreichend, die Rohdaten und das `Makefile` aufzubewahren. Damit kann die Datenanalyse jederzeit reproduziert werden.

Im folgenden zeige ich Beispiele aus der täglichen Arbeit.

Datenverarbeitung

Filtern seismischer Daten

Die Seismogramme in der Datei *name*`.sff` werden mit dem Programm `stufi` gemäß der Filtereinstellungen in `brb.fil` gefiltert und nach *name*`.brb.sff` geschrieben.

```
# filter rules
%.brb.sff: %.sff brb.fil
        stufi $(word 2,$^) -v -o $<
        /bin/mv -fv $<.sfi $@
%.lp.sff: %.sff lp.fil
        stufi $(word 2,$^) -v -o $<
        /bin/mv -fv $<.sfi $@
%.mag.sff: %.sff mag.fil
        stufi $(word 2,$^) -v -o $<
        /bin/mv -fv $<.sfi $@
```

Datenverarbeitung

Fortgeschritten: Automatische Erzeugung der *Rules*

```
# define stufi filters to be used
FILTERS=brb lp mag
# how to create a filter rule
%.rule: Makefile
      ( echo "%.$(patsubst %.rule,%,$@).sff:" \
             " %.sff $(patsubst %.rule,%,$@).fil"; \
        echo -e '\tstufi $$(word 2,$$^) -v -o $$<' ; \
        echo -e '\t/bin/mv -fv $$<.sfi $$@' ) > $@
# create and include filter rules
include $(addsuffix .rule,$(FILTERS))
```

Datenverarbeitung

Plotten seismischer Daten

Mehrere Wellenformen, die in Dateien mit unterschiedlichen Namen gespeichert sind, aber am gemeinsamen Mittelteil der Dateinamen als zusammengehörig erkennbar sind, werden in einem Plot dargestellt:

```
# plots
%.raw.ps: edd%.sff pa%.sff
      stuplo -d $@/ps -t -a -s x -i -c daIcsfT -V -Y 0.7 \
        -l 3,2,1 -h 1.4,1.4,1.4,1.4 -X "Zeit (UT)" $^
%.scaled.ps: BRB.%.sff LP.%.sff MAG.%.sff Pa.%.sff Pacc.%.sff  \
  Pacc.rsa.%.sff LP.%.le2.tre.sff MAG.%.lpsim.sff
      stuplo -d $@/ps -t -a -s x -i -c daIcsfT -V -Y 0.7 \
        -l 3,2,1 -h 1.4,1.4,1.4,1.4 -X "Zeit (UT)" $^
```

Bildbearbeitung

Rules, um Postscript-Dateien aus tif- und jpeg-Dateien zu erzeugen und diese nach Encapsulated Postscript und PDF zu konvertieren:

```
tmp/%.ps: tmp/%.jpg
        djpeg -pnm -verbose $< | pnmtops -nocenter -noturn > $@
tmp/%.ps: tmp/%.tif
        tifftopnm $< | pnmtops -noturn > $@
tmp/%.eps: tmp/%.ps
        ps2epsi $< $@
tmp/%.pdf: tmp/%.eps
        cd tmp; epstopdf $(patsubst tmp/%,%,$<)
```

LaTeX

Übersetzen eines LaTeX Textes in eine `dvi`-Datei und weiter in eine Postscript Datei. Der Befehl `make para.psp` erzeugt die Datei `para.ps` aus `para.tex` und den dazugehörigen Plot-Dateien, zeigt sie im Previewer an und löscht sie anschließend wieder.

```
# figures
PSPLOTS=plot1.ps plot2.ps plot3.ps plot4.ps wplot1.ps wplot2.ps
PDFPLOTS=$(patsubst %.ps,%.pdf,$(PSPLOTS))
$(PDFPLOTS): %.pdf: %.ps; ps2pdf -dPDFSETTINGS=/printer $<
# go for the manuscript
para.dvi: para.tex $(PSPLOTS)
para.pdf: para.dvi $(PDFPLOTS)
%.dvi: %.tex; latex $(patsubst %.tex,%,$<)
.PHONY: rebib
rebib: para.tex; latex para; bibtex para; latex para; latex para
%.ps: %.dvi; dvips -ta4 $(patsubst %.dvi,%,$<)
%.psp: %.ps; gv $<; /bin/rm -fv $<
%.pdf: %.dvi; pdflatex $(patsubst %.dvi,%,$<)
%.pdp: %.pdf; acroread $<; /bin/rm -fv $<
%.x: %.dvi; xdvi -paper a4 $<
```

Pretty printing

Ausdruck von ASCII-Dateien mit Listen, Notizen und Texten zum Feldpraktikum mit ansprechender Formatierung und Überschrift:

```
YEAR=$(shell date +'%Y')
TITLE=Feldpraktikum $(YEAR)
DATE=$(shell date +'%d/%m/%Y')
packliste.xxx: ; echo "Packliste Geophysikalisches Feldpraktikum" > $@
gruppen$(YEAR).xxx: ; echo "Gruppeneinteilung Feldpraktikum $(YEAR)" > $@
countdown.xxx: ; echo "Countdown $(shell date +'%Y')" > $@
adressen.xxx: ; echo "Adressen $(TITLE)" > $@
%.ps: %.txt %.xxx
      a2ps -Eplain -1 -o $@ --center-title="$(shell cat $(word 2,$^))" \
        --left-title="$(TITLE)" --margin=36 --sides=duplex \
        --left-footer=" " --right-footer="#{user.name}, GPI, $(DATE)" \
        --header=" " --borders=no $<
```

File directory

Mit `make edit` wird eine Liste `flist` aller Dateien erzeugt, die als Quellen für ein Projekt dienen. Diese Liste wird in den Editor geladen. Die Dateinamen dienen als Sprungziele, zur Auswahl der Datei im Editor:

```
WWWWORKDIR=$(HOME)/work/txt/html
WWWPAGESRC=$(WWWWORKDIR)/txt/programming_online.html

flist: Makefile $(wildcard *.tex *.txt *.sty *.sh *.awk) \
        $(wildcard code/Makefile code/*.c code/*.cc code/*.h code/*.f) \
        $(WWWPAGESRC)
      echo $^ | tr ' ' '\n' | sort > $@

.PHONY: edit
edit: flist; vim $<
```

Übungsaufgabe

Aufgabestellung

Erstellen Sie ein `Makefile` zur Erzeugung des Programms `poly` aus den C Quellcode Dateien in den Programmbeispielen. Steigern Sie langsam die verwendeten Konzepte. Benutzen Sie schließlich *Pattern Rules* und *built-in implicit Rules*. Ergänzen Sie das Makefile um einen Eintrag, mit dem sich ein Postscript-Ausdruck der Quellcode-Dateien erzeugen lässt und der `a2ps` verwendet, um die Syntax-Elemente hervorzuheben und Zeilennummern vor den Text zu drucken.

Danksagung

Diese Präsentation wurde mit der LaTeX Beamer Klasse von Till Tantau erstellt:

`http://latex-beamer.sourceforge.net`