Programmiermethoden in der Mathematik
Wintersemester 05/06

TU Berlin, Institut für Mathematik
Thomas Slawig

Makefiles

Mit dem Unix-Kommando `make` kann man auf komfortable Weise ausführbare Programme erstellen, besonders wenn diese aus vielen Quelldateien erzeugt werden müssen. Vorteile sind:

- Es werden automatisch alle Quelldateien übersetzt, die seit dem letzten Erzeugen des ausführbaren Programms verändert wurden.
- Es werden aber auch *nur diese* neu übersetzt, d.h. Objektdateien, die zu unveränderten Quelldateien gehören, werden nicht neu erzeugt.

Dazu werden in einem **Makefile** *Ziele, Abhängigkeiten* und *Regeln* beschrieben:

- **Ziele** sind zum einen die ausführbare Datei, zum anderen alle Objektdateien, die für sie verwendet werden.
- Die **Abhängigkeiten** eines Zieles sind alle Dateien, deren Änderung sich auf die Generierung des Zieles auswirkt.

  So hängt eine Objektdatei von der entsprechenden Quelldatei und den in diese eingebundenen, selbstgeschriebenen Header-Dateien ab. Systemheader-Dateien muss man nicht aufführen, da sich diese ja nicht ändern.

  Die Abhängigkeiten werden im Makefile in einer Zeile angegeben. Dabei steht zuerst die Zieldatei und dann hinter einem Doppelpunkt die Liste der Dateien, von denen sie abhängt.

  Ein Beispiel:

  ```
  datei.o: datei.cc includedatei.h
  ```

  Ein ausführbares Programm als Zieldatei hängt von allen Objektdateien ab, die zusammengelinkt werden sollen:

  ```
  main: main.o f1.o f2.o
  ```

  `make` arbeitet rekursiv, d.h. die Abhängigkeiten der Objektfiles von den Quell- und Headerdateien müssen in dem letzten Beispiel nicht wiederholt werden.
- Ist nun eine der abhängigen Dateien verändert worden, so wird das Ziel neu erzeugt. Wie dies geschieht, muss in einer **Regel** festgelegt sein. Die Regel steht in einer Zeile unter der entsprechenden Abhängigkeit. Sie beginnt mit dem Tabulatorzeichen (→ |, wichtig!!!), gefolgt von dem Kommando, das zum Erzeugen der Zieldatei ausgeführt werden muss. Für eine Objektdatei ist das der Aufruf des Compilers, für ein ausführbares Programm der Aufruf des Linkers.

Es ergeben sich dann die folgenden Zeilen im Makefile:

(1) Für eine Objektdatei:

```
datei.o: datei.cc includedatei.h
→ |   g++ -c datei.cc
```

(2) Für ein ausführbares Programm:

```
main: main.o f1.o f2.o
→ |   g++ main.o f1.o f2.o -o main
```

Bemerkungen:

- Eine Abhängigkeit oder ein Kommando endet jeweils am Ende der Zeile. Mit \ am Ende einer Zeile wird diese mit der folgenden verbunden.

- Kommentare im Makefile beginnen mit `#` und gehen bis zum Ende der Zeile.
- Heißt das Makefile auch `Makefile` oder `makefile`, so wird es mit dem Kommando `make Ziel` aufgerufen und die entsprechende Regel ausgeführt. Sonst mit `make -f Dateiname`.
- `make` (ohne Zielangabe) führt automatisch die erste Regel aus.

Flexibel wird `make` durch die Möglichkeit, Makros und Variablen zu definieren, die dann mit in Regeln benutzt werden können.
Es gibt bereits einige vordefinierte Variablen, u.a.:

```
CXX = g++                                  # C++ Compiler
COMPILE.cc = $(CXX) $(CXXFLAGS) -c         # Regel zum Compilieren von C++ Dateien
LINK.cc = $(CXX) $(CXXFLAGS) $(LDFLAGS)    # Regel zum Linken von C++ Dateien
$@                                         # Name des Ziels mit Dateiendung
$^                                         # Namen aller abh"angigen Dateien
                                           # (ohne Mehrfachnennung)
$*                                         # Name des Ziels ohne Dateiendung
$<                                         # Name der ersten abh"angigen Datei
$?                                         # Namen aller abh"angigen Dateien,
                                           # die neuer als das Ziel sind
$+                                         # Namen aller abh"angigen Dateien
```

Vordefinierte Variablen können überschrieben werden. Die Variablen `CXXFLAGS` und `LDFLAGS` sind für Optionen (engl.: *flags*) bestimmt und *nicht* vordefiniert. Eine nichtdefinierte Variable hat bei ihrer Benutzung keinen Wert.

Für die wichtigsten Programmiersprachen gibt es bereits vordefinierte (implizite) Regeln, z.B.

```
%.o: %.cc                  # % ist ein Platzhalter f"ur jeden beliebigen Namen
     $(COMPILE.cc) $<      # so macht man aus einer C++ Quelldatei eine Objektdatei
```

Daraus ergibt sich:

- Für die Erzeugung der Objekt-Dateien in (1) reicht es, nur die Abhängigkeiten von Include-Dateien anzugeben. Gibt es keine Include-Dateien, so kann diese Zeile also wegfallen.
- Ein Beispiel für eine Regel zum Erzeugen eines ausführbaren Programms lautet

```
main: main.o f1.o f2.o
      $(LINK.cc) $^ -o $@
```

Ein etwas komplexeres Beispiel mit einer statischen Bibliothek:

```
CXX     = g++                       # C++ Compiler
CXXFLAGS =                          # C++ Compileroptionen
LIBS = lib.a                        # statische Bibliotheken
OBJS = main.o f1.o f2.o             # alle Objektdateien

main: $(OBJS)
      $(LINK.cc) $(OBJS) $(LIBS) -o $@
clean:
      rm -f $(OBJS) main            # aufr"aumen
```

Hier wird ebenfalls die implizite Regel zum Compilieren ausgenutzt. Die Option `-f` (force) des `rm`-Kommandos ohne Rückfragen und unterdrückt Fehlermeldungen, z.B. wenn Dateien nicht existieren.

Hinweis: Alle Angaben hier beziehen sich auf das GNU `make`.