C Makefile

Der Code eines Projekts sei z.B. in den Dateien

```
run_cg.c quickdirty.c geom_pbc.c cg.c dotprod.c laplace.c
global.h quickdirty.h geom_pbc.h cg.h dotprod.h laplace.h
```

enthalten, die alle in einem Verzeichnis liegen.
Nun möchte man *bedingt compilieren*, d.h. ein Quellprogramm soll nur neu übersetzt werden, wenn es nötig ist (Änderung am Programm oder an einer Include–Datei). Dafür gibt es ein Werkzeug namens **make**, dem das folgende Konzept zu Grunde liegt:
Ein File (*target*) muss neu erzeugt werden, wenn es noch nicht existiert oder wenn eine seiner "Zutaten" (*prerequisites*) inzwischen verändert wurde. Zur Erzeugung des Targets formuliert man einen Befehl (*command*), wie man ihn auf der Kommandozeile eingeben würde. So entsteht eine *Regel* der Form

```
target : prerequisites
<TAB>command
```

Der `command`–Teil kann auch mehrere Shell–Befehle umfassen – jeder **muss** mit einem **Tabulator** eingerückt werden!
Fortsetzungszeilen werden durch \ am Zeilenende angekündigt.
Hier nun als direkte Anwendung das **Makefile** für das genannte Projekt:

```
-------------------------------------------------------------------
run_cg.x : run_cg.o quickdirty.o geom_pbc.o cg.o \
           dotprod.o laplace.o
      gcc -O -o run_cg.x run_cg.o quickdirty.o geom_pbc.o cg.o \
           dotprod.o laplace.o -lm

run_cg.o : run_cg.c global.h quickdirty.h geom_pbc.h cg.h \
           dotprod.h laplace.h
      gcc -O -c run_cg.c

quickdirty.o : quickdirty.c
      gcc -O -c quickdirty.c

geom_pbc.o : geom_pbc.c global.h
      gcc -O -c geom_pbc.c
```

```
cg.o : cg.c dotprod.h
      gcc -O -c cg.c

dotprod.o : dotprod.c
      gcc -O -c dotprod.c

laplace.o : laplace.c global.h
      gcc -O -c laplace.c
--------------------------------------------------------------------
```

Es gibt eine kaum überschaubare Menge von Makefile–Tricks, insbesondere in großen Projekten grenzt das oft an Hexerei. Hier nun eine Version unseres Makefiles mit moderaten Verbesserungen.

```
--------------------------------------------------------------------
CC = gcc
CFLAGS = -O
LDFLAGS = -lm

headers = global.h quickdirty.h geom_pbc.h cg.h dotprod.h laplace.h
objects = run_cg.o quickdirty.o geom_pbc.o cg.o dotprod.o laplace.o

run_cg.x : $(objects)
      $(CC) $(CFLAGS) -o $@ $(objects) $(LDFLAGS)

run_cg.o : run_cg.c $(headers)
      $(CC) $(CFLAGS) -c $<

quickdirty.o : quickdirty.c
      $(CC) $(CFLAGS) -c $<

geom_pbc.o : geom_pbc.c global.h
      $(CC) $(CFLAGS) -c $<

cg.o : cg.c dotprod.h
      $(CC) $(CFLAGS) -c $<

dotprod.o : dotprod.c
      $(CC) $(CFLAGS) -c $<

laplace.o : laplace.c global.h
```

```
	$(CC) $(CFLAGS) -c $<

clean :
	rm -f *.o *.x
-------------------------------------------------------------------
```

Es sind (oberhalb der Regeln) einige Abkürzungen eingeführt worden, die Schreibarbeit sparen und die Anwendung flexibler machen, siehe unten. Auch innerhalb der Regeln gibt es nützliche Abkürzungen:

```
$@    target
$<    first prerequisite
```

Damit sieht die Befehlszeile in allen Compilier–Regeln gleich aus.
Wenn man das `Makefile` genau unter diesem Namen abgespeichert hat, dann wird der gesamte Compile/Link–Prozess einfach durch den Befehl

```
make
```

ausgelöst, denn es wird, wenn nichts anderes angegeben ist, das *erste* Target erzeugt/aktualisiert, und das ist das Binärfile `run_cg.x`.
Man kann auch die im Makefile vordefinierten Parameter auf der Kommandozeile umsetzen: so wird

```
make CC=g++
```

mit `g++` statt `gcc` compilieren, und

```
make CFLAGS=-Wall
```

schaltet die Optimierung aus und stattdessen die Warnungen ein etc.
Schließlich ist noch das Target `clean` zum Aufräumen dazu gekommen.
`make clean` ist nötig, wenn man das Makefile selber editiert hat oder wenn man `make` mit neuen Parametern aufrufen will:

```
make
...
make clean
make CC=icc
```