1 Writing Makefiles

1.1 Targets, Rules, Dependencies

Ein Makefile dient dazu, dem Programm *make* mitzuteilen

- was es tun soll: *Targets*
- wie es das tun soll: *Rules*

Werden für ein *Target* andere Dateien benötigt, sogenannte *Dependencies*, müssen diese auch angegeben werden.

Bemerkung *Dependencies* sind für make auch *Targets*, die an anderer Stelle erstellt werden müssen.

1.1.1 Beispiel

Kompilierung eines einfachen C-Programms names prog aus den Dateien prog.c und prog.h.

```
1  prog: prog.c prog.h
       gcc -o prog prog.c
```

Listing 1: Einfaches Makefile mit Target "prog".

1.2 Das Defaulttarget

Beim Aufruf von *make* kann das zu erzeugende Target direkt angegeben werden. Wenn nicht, wird das *Defaulttarget* erzeugt. Dies ist das erste Target im Makefile.

1.2.1 Beispiel

Kompilierung eines einfachen C-Programms mit anschließendem Aufruf von *make*.

```
   # Simple Makefile for c-compiling.
2  main: prog.o main.o
       gcc -o main prog.o main.o
4
   prog.o: prog.c
6      gcc -c prog.c

8  main.o: main.c
       gcc -c main.c
10
   clean:
12     rm -rf *.o main
```

Listing 2: Ein einfaches Makefile. Bemerke, dass in diesem Fall das Defaulttarget durch "main" gegeben ist.

```
$ make main.o
$ make
```

Listing 3: Aufruf von make. In der ersten Zeile wird nur das Target main.o erzeugt. Im zweiten dann das Defaulttarget main.

1.3 Pattern in Regeln

Es besteht die Möglichkeit Regeln durch eine Art *Wildcard-Pattern* zu defenieren. Wichtige vordefinierte Variablen sind:

- `$<` die erste Abhängigkeit
- `$@` Name des Targets
- `$+` eine Liste aller Abhängigkeiten
- `$^` eine Liste aller Abhängigkeiten, ohne doppelte Einträge

1.3.1 Beispiel

```
%.o: %.c
    gcc -Wall -g -c $<
main: prog.o main.o
    gcc -o main $^
```

Listing 4: Nutzung von Patterns zur Erzeugung von Objektdateien aus ihren jeweiligen c-Dateien.

1.4 Variablen in Makefiles

Es ist auch möglich in Makefiles *Variablen* zu definieren. Dies wird üblicherweise in Großbuchstaben gemacht. Beispielsweise:

`CC` Compiler

`CFLAGS` Compiler-Flags

`LDFLAGS` Linker-Optionen

Der Zugriff auf die jeweilige Variable erfolgt durch `$( )`.

1.4.1 Beispiel

```
CC = gcc

main: prog.o main.o
    $(CC) -o main $^
```

Listing 5: Verwendung der Variable `CC` zur Beschreibung des Compilers.

1.5 Kommentare in Makefiles

Kommentare in Makefiles können durch das Voranstellen einer Raute `#` eingefügt werden.

1.6 Phony Targets

Im Allgemeinen prüft *make*, ob ein Target aktueller ist als alle Dependencies von den es abhängt. Ist dies nicht der Fall, wird es neu erzeugt.
Manche Targets sollen aber unabhängig von ihren Dependencies immer erzeugt werden. Solche Targets nennt man *Phony*s.

1.6.1 Beispiel

Ein klassisches Beispiel für ein *Phony* ist das Target `clean`. Dieses Target ist in der Regel keine Datei und sollte deshalb im Allgemeinen immer erzeugt werden. Problematisch wird dies allerdings, wenn eine Datei mit dem selben Namen `clean` existiert, dass von keinen anderen Dateien abhängt. In diesem Fall ist das Target `clean` immer aktueller als seine Dependencies und wird also nie erzeugt. Deshalb deklariert man das Target `clean` häufig als Phony.

```
.PHONY: clean
clean:
	rm -rf $(BIN) $(OBJ)
```

Listing 6: Das Target `clean` wird häufig als Phony deklariert.

1.7 Pattern Substitution

Wie bei Rules kann man auch Variablen mit Hilfe von *Pattern* erzeugen.

1.7.1 Beispiel

```
OBJ= datei1.o datei2.o datei3.o
SRC= $(OBJ:%.o=%.c) datei4.c
SRC=/file/$(SRC)
```

Listing 7: Einsatz von Pattern zur Deklaration von Variablen. Hier werden die Objektdateien genutzt um die Source-Dateien zu deklarieren.

1.8 Abhängigkeiten als Target (make dep)

Werden Objekt-Dateien durch Pattern erzeugt, entfallen die Abhängigkeiten von Header-Dateien. Um dieses Problem zu umgehen, definiert man meist ein Target names `dep`.

1.8.1 Beispiel

```
1  SRC = datei1.c datei2.c datei3.c
   DEPENDFILE = .depend
3
   dep: $(SRC)
5      gcc -MM $(SRC) > $(DEPENDFILE)

7  -include $(DEPENDFILE)
```

Listing 8: Hier wird das Target `dep` definiert, um Header-Dateien bei der Kompilierung mit einzubeziehen.

1.9 Rekursives Make

Für große Projekte sind die Source-Dateien oft auf verschiedene Verzeichnisse verteilt. Es besteht die Möglichkeit für jedes dieser Unterverzeichnisse ein separates Makefile zu schreiben, das von einem zentralen Makefile im Root-Ordner aufgerufen wird.