4 Automatisieren der Compilierung mit `make`

4.1 Überblick über `make`

`make`:

- erlaubt die **Automatisierung** der vielen Schritte,
  die zur **Generierung eines Programms** aus vielen Quelldateien notwendig sind
- **Verwaltung der Abhängigkeiten** dieser Schritte,
  was bei **Programmänderungen** wichtig wird

Beispiele:

- **Kompilierung eines C-Programms** aus vielen Quell-Dateien.
- **Kompilierung der LaTeX-Quellen** dieser Skriptnotizen:
  Folien werden **separat** kompiliert, dann in einzelne Postscript-Seiten zerlegt, diese dann in die Skriptnotizen eingebunden und zusammen kompiliert. Weiterhin werden die vielen **Beispielprogramme** aus ihren Quelldateien eingebunden.

Wenn sich etwas ändert, wird nur das Notwendige neu kompiliert, was viel Zeit spart. Dazu schaut sich `make` das **Veränderungsdatum** der Dateien an. Die **Korrektheit** der Abhängigkeitsprüfung wird garantiert (wenn die Regeln einmal richtig aufgestellt wurden).

Die **Abhängigkeiten** werden in einem **„Makefile“** aufgeschrieben. **Viel Wissen** um Standard-Abhängigkeiten ist in `make` sogar **bereits eingebaut**. Dies funktioniert über **Datei-Endungen**.

4.2 Grundlagen von Makefiles

4.2.1 Einfaches Beispiel: Editor übersetzen

Wir nehmen an, daß wir ein **ausführbares Programm namens** `edit` erzeugen wollen, für das wir **acht C-Quellen** und **drei Header-Files** haben:
`editor-simple/Makefile`:

```
# Makefile for "edit".

edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
```

Kommentare werden mit dem Musikkreuz „`#`“ eingeleitet.

Das **ausführbare Programm** ist `edit`, es wird aus acht Objektdateien zusammengebaut.

Eine **Regel besteht** aus einem **Ziel,** einer **Liste von Vorbedingungen** und einer **Liste von Anweisungen**.

Ein **Ziel** ist eine Datei, die **erzeugt** werden muß. Es kann auch etwas sein, was **nur getan** werden muß, ohne daß eine Datei entsteht.

Eine **Vorbedingung** ist eine Datei, die für die Erzeugung eines Ziels notwendig ist. Ein Ziel hat häufig viele Vorbedungungen.

Eine **Anweisung** ist etwas, was `make` ausführt. Eine Regel kann mehr als eine Anweisung haben, jede auf einer Zeile.

ACHTUNG: Vor jeder Anweisungs-Zeile muß ein **Tabulator-Zeichen** stehen! Acht Blanks stattdessen sind nicht erlaubt! Das ist eine **häufige Falle**. Z.B. der Editor `vim` hilft, indem er solche Syntaxfehler farblich hervorhebt. ►► Ausprobieren.

Eine **Regel erklärt**, wie eine Ziel-Datei neu aus ihren Vorbedingungs-Dateien erzeugt werden kann. Eine Regel kann auch nur beschreiben, wie eine bestimmte Aktion auszuführen ist.

Ein Makefile kann noch mehr als Regeln enthalten, aber dies ist das **Grundschema**.

Im Beispiel wird jede der acht **Objektdateien aus** der zugehörigen **C-Quelle** erzeugt.

Außerdem **hängen** die Objektdateien noch **von** den **Header-Dateien** ab, die in der C-Quelle mit eingeschlossen wurden.

▶▶ `kbd.c`

▶▶ `defs.h`

`defs.h` wird von allen C-Quellen eingeschlossen. `command.h` wird nur von den Editier-Kommandos eingeschlossen, und `buffer.h` wird nur von den Low-Level-Files eingeschlossen, die den Editor-Puffer manipulieren.

Lange Zeilen können mit **Backslash-Newline** aufgeteilt werden.

Oft wollen wir auch **alle generierten Dateien** wieder **löschen**. Hierfür können wir eine **Aktion** definieren: `clean`

Eine Aktion **erzeugt keine Datei**. Oft hat sie auch keine Vorbedingung.

Eine **Anweisung** muß mit einem **Tab** beginnen und enthält ein **beliebiges Shell-Kommando**. Da `make` nichts über die Semantik dieser Kommandos weiß, sollte man die Vorbedingungen und Ziele tunlichst **richtig aufschreiben**. Es gibt dafür aber Hilfen, dazu später mehr.

4.2.2 Grundalgorithmus von `make`

Um die **Übersetzung** zu **starten**, tippt man

▶▶ `make`

`make` fängt mit dem **ersten Ziel** an, dem **Default-Ziel**. Hier ist das `edit`, das wir deswegen nach vorne gestellt haben.

`make` schaut sich an, **wie** das **Ziel erzeugt** wird, und es schaut sich **an,** wovon es **abhängt**.

`make` kann `edit` nicht sofort erzeugen, da **auch** dessen **Vorbedungungen**, die Objektdateien, **Regeln** haben. Es geht **rekursiv** durch die Abhängigkeiten, bis es Dateien findet, die keine Regeln haben. Aus den Abhängigkeiten stellt es eine Reihenfolge von Anweisungen zusammen, die alle notwendigen Schritte in der **richtigen Reihenfolge** enthält.

Falls sich eine **Vorbedingungs-Datei ändert**

▶▶ `touch search.c`

▶▶ `make`

dann werden nur die notwendigen Dateien neu erzeugt. Auch bei **komplizierteren Abhängigkeiten**:

▶▶ `touch buffer.h`

▶▶ `make`

Wenn **nichts zu tun** ist, sagt `make` uns das auch:

▶▶ `make`

Falls eine Regel **nicht** zur Erzeugung des gewünschten Ziels **benötigt** wird, wird sie **nicht benutzt**.

Man kann auf der Kommandozeile auch ein **Ziel angeben**:

▶▶ `make clean`

Dann wird das **Default-Ziel nicht** benutzt.

4.2.3 Grundlagen von Variablen

Im Beispiel haben wir die Objektdateien in der Regel von `edit` **zweimal aufgeführt**. Das ist **fehlerträchtig**. Mit Variablen kann man das vermeiden:
`editor-simple/Makefile-var`:

```
# Makefile for "edit".

objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

Es ist üblich, in jedem Makefile eine Variable **namens** `objects`, `OBJECTS`, `objs`, `OBJS`, `obj` oder `OBJ` zu haben.

Um Variablen zu **benutzen**, schreibt man ein **Dollar**-Zeichen und **runde Klammern**.

Man beachte, daß auch die **Regel für `clean`** viel **robuster** gegenüber Änderungen wird.

Einschub: Der Name eines Makefiles ist normalerweise `Makefile` oder `makefile`. Man kann aber auch einen anderen Namen auf der Kommandozeile angeben:

►► `make -f Makefile-var`

►► `make -f Makefile-var clean`

4.2.4 Grundlagen impliziter Regeln

Die **Regeln** für die Erzeugung von Objekt-Dateien aus C-Quellen sind eigentlich **immer analog**.

`make` enthält daher bereits viele eingebaute **implizite Regeln**.

`make` orientiert sich dabei an den **Dateiendungen**. Um z.B. von einer `.c`-Datei zu einer `.o`-Datei zu kommen, braucht man immer ein „`cc -c`"-Kommando.

Wenn eine `.o`-Datei erzeugt werden soll, und wenn es eine zugehörige `.c`-Datei **bereits gibt**, dann wird die implizite **Regel automatisch angewendet**, und die `.c`-Datei wird automatisch in deren **Liste der Vorbedingungen aufgenommen**.

So **könnte** unser Makefile in der **Praxis aussehen**:
`editor-simple/Makefile-impl`:

```
# Makefile for "edit".

objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        -rm edit $(objects)
```

Implizite Regeln werden **häufig angewendet**.

▶▶ `make -f Makefile-impl`

▶▶ `make -f Makefile-impl clean`

Auf das obige „`.PHONY`“ für `clean` kommen wir noch zurück, ebenso auf das **Minus** vor `rm`. Das erstere verhindert, daß aus Versehen eine Datei namens `clean` erzeugt wird, das letztere bewirkt, daß Fehler wegen nicht-exisitierender Dateien beim Löschen ignoriert werden.

4.3 Aufbau eines Makefiles

4.3.1 Struktur eines Makefiles

Ein Makefile kann enthalten:

- explizite Regel
- implizite Regel (selbstdefiniert)
- Variablendefinition
- Direktive
- Kommentarzeile mit `#`

Eine **Direktive** macht etwas **besonderes,** wenn das **Makefile eingelesen** wird:

- anderes Makefile **einlesen**
- **bedingtes** Benutzen/Ignorieren von Teilen des Makefiles
- definieren einer **zusammengefaßten Anweisungssequenz**

Ein **Kommentar** geht vom `#`-Zeichen bis zum **Zeilenende. Ausnahme**: ein **Backslash** am Ende ohne einen zweiten Backslash davor **setzt** die Kommentarzeile **fort**.

Kommentare dürfen **fast überall** stehen. Eine Zeile nur mit einem Kommentar erscheint als **leer** und wird daher **ignoriert**.

4.3.2 Name eines Makefiles

Als **Default** wird `makefile` oder `Makefile` genommen, in dieser **Reihenfolge**. **Empfohlen** wird das **zweite**, weil es sich im Verzeichnis besser heraushebt.

Gnu-`make` sucht **außerdem** vorher noch nach `GNUmakefile`. Das sollte man aber nur benutzen, wenn man verschiedene Makefiles für verschiedene `make`s haben will.

Wenn man **mehrere Makefiles** im selben Verzeichnis haben will, muß man mit der Kommandozeilenoption `-f` arbeiten. Dann benutzt man als **Dateierweiterung** oft `.mk` oder auch `.m` oder `.mak` .

4.3.3 Aufbau der Regeln

ziele : *vorbedingungen*
        anweisung
        ...

oder auch:

ziele : *vorbedingungen* ; *anweisung*
        anweisung
        ...

Anmerkung: Die **zweite Form** habe ich noch **nie** in der Praxis **gesehen**.

Die **Reihenfolge** der Regeln ist **unwichtig, außer** der **ersten** Regel. Die erste Regel ist die **Default-Regel**.

Ausnahmen: Ziele, die **mit** einem **Punkt beginnen**, zählen nicht. Und **Regel-Muster** (s.u.) auch nicht.

Deshalb ist die erste Regel meißt die, die das **gesamte Programm zusammensetzt**. Dieses Ziel wird **oft `all` genannt**.

Die ***ziele*** sind Dateinamen, von Leerzeichen getrennt. **Meist** hat man **nur ein** Ziel pro Regel.

Eine ***anweisung*** beginnt immer mit einem **Tab**. (Man kann es nicht oft genug sagen!)

Man kann **Zeilen** mit **Backslashes umbrechen**. Es gibt allerdings **kein Limit** für die Zeilenlänge.

Ein einzelnes **Dollar-Zeichen** erreicht man durch zwei davon, wg. der Expansion von Variablennamen.

Eine Regel sagt `make` zwei Dinge:

- Wann ist das Ziel veraltet
- Wie aktualisiert man das Ziel bei Bedarf

Die **Bedingungen** für **„veraltet“** werden von den Vorbedingungen angegeben.

Die ***Vorbedingungen*** sind ebenfalls **Dateinamen,** von Leerzeichen getrennt. Ein Ziel **ist veraltet**, wenn es **nicht existiert**, oder wenn es **älter als irgendeine** seiner **Vorbedingungs**-Dateien ist.

Die ***Anweisungen*** werden mit einer **Unix-Shell**, normalerweise `sh`, ausgeführt.

4.3.4 Unechte Ziele

Manchmal will man eine Aktion ausführen, die gar keine Datei erzeugt, z.B. das Aufräumen mittels `make clean`, siehe oben.

Was passiert, wenn man `make clean` **mehrfach aufruft?** Es wird **jedesmal wieder** ausgeführt, weil das Ziel nicht existiert. (Und zwar jeweils **einmal**.)

Was passiert, wenn jemand eine **Datei namens** `clean` aus Versehen **erzeugtt?** Die Regel wird **nie mehr ausgeführt!**

Lösung bei Gnu-`make`: Dieses Ziel als „unecht“ markieren:
`editor-simple/Makefile-impl`:

```
# Makefile for "edit".

objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        -rm edit $(objects)
```

`.PHONY` ist ein **besonderes**, fest **eingebautes Ziel**. Seine Vorbedingungen werden **nicht als Dateien interpretiert**, sondern als unechte Ziele.

Die Regeln für unechte Ziele werden, wenn verlangt, **immer ausgeführt, egal ob** eine **Datei** dieses Namens **existiert** oder nicht.

Weiterhin wird für unechte Ziele **nicht nach impliziten Regeln gesucht**, was die **Geschwindigkeit** von `make` etwas **steigert**.

Eine **andere Einsatzmöglichkeit** für unechte Ziele ist die **rekursive** Verwendung von `make` in **Unterverzeichnissen**. Man könnte schreiben:
`Makefile-subdirs`:

```
subdirs = foo bar baz

subdirs:
        for dir in $(subdirs); do \
          $(MAKE) -C $$dir; \
        done
```

Hier rufen wir `make` **rekursiv** für jedes Unterverzeichnis auf. **Näheres** dazu kommt **später**.

Anmerkung: Wir müssen die **Shell-Kommandos mit Backslashes verbinden**, damit sie für `make` **eine einzige Anweisung** werden. Sonst würden sie in drei getrennte Shells gesteckt und ergäben Syntaxfehler.

Nachteile dieser Methode? Fehler in Unter-Makefiles werden im Haupt-Makefile nicht erkannt. Mögliche **Nebenläufigkeit** kann nicht ausgenutzt werden, z.B. auf **Mehrprozessormaschinen**.

Lösung: Die Unterverzeichnisse als unechte Ziele deklarieren:
`Makefile-subdirs2`:

```
subdirs = foo bar baz

.PHONY: subdirs $(subdirs)

subdirs: $(subdirs)

$(subdirs):
        $(MAKE) -C $@

foo: baz
```

Hier haben wir übrigens **explizit deklariert**, daß das **Unterverzeichnis** `foo` **erst nach** dem Unterverzeichnis `baz` bearbeitet werden darf. Das ist besonders für **parallele Übersetzung** wichtig.

Der **Ausdruck** `$@` bezeichnet das aktuelle Ziel. Diese **automatischen Variablen** erläutern wir **später**.

Unechte Ziele **können Vorbedingungen** haben. Das ist praktisch, wenn man in einem Makefile mehrere ausführbare Programme beschreiben will. Dann definiert man ein unechtes Ziel namens `all`, das von den echten Zielen abhängt:
`Makefile-all`:

```
all : prog1 prog2 prog3
.PHONY : all

prog1: prog1.o utils.o
        cc -o prog1 prog1.o utils.o
prog2: prog2.o
        cc -o prog2 prog2.o
prog3: prog3.o sort.o utils.o
        cc -o prog3 prog3.o sort.o utils.o
```

Jetzt werden **mit einem Aufruf** `make` **alle drei** Programme gebaut.

Man kann **aber auch** nur `make prog1 prog3` sagen.

Wenn ein **unechtes Ziel** eine **Vorbedingung eines anderen** unechten Ziels ist, dient es als **Unterprogramm**:
`Makefile-cleanall`:

```
.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
        rm programm

cleanobj :
        rm *.o

cleandiff :
        rm *.diff
```

Leider gibt es **„unechte Ziele“ nur bei Gnu-**`make`, nicht bei allen `makes`. Will man Makefiles auch für diese `makes` schreiben helfen **Regeln ohne Vorbedingungen und ohne Anweisungen**:

`Makefile-cleanforce`:

```
clean: FORCE
        rm $(objects)

FORCE:
```

Wenn es das Ziel dieser Regel, hier `FORCE`, **nicht als Datei gibt**, dann wird angenommen, daß **alle Ziele**, die von diesem Ziel **abhängen**, **jedesmal neu bearbeitet** werden müssen.

Der **Name** eines solchen Ziels **ist egal**.

Die **Wirkung** ist die **gleiche** wie eben, aber es ist **nicht so gut lesbar**.

Problem: Wenn eine **Datei mit diesem Namen erzeugt** wird, geht die Regel schief.

4.3.5 Mehrere Regeln für ein Ziel

Eine Datei kann **Ziel von mehreren Regeln** sein.

Die **Vorbedingungen** aller Regeln werden dann **vereinigt**.

Nur **höchstens eine** Regel darf **Anweisungen** haben.

Es ist oft **praktisch**, ein paar **zusätzliche Abhängigkeiten** mit **zusätzlichen Regeln** zu schreiben, die dann keine Anweisungen haben:

`Makefile-multrule`:

```
objects = foo.o bar.o

foo.o : defs.h
bar.o : defs.h test.h

$(objects) : config.h
```

Hier wird die **zusätzliche Abhängigkeit** aller Objekt-Dateien von `config.h` **in** einer

separaten Zeile ausgedrückt. Man könnte das dazutun oder löschen, ohne die echten Regeln anzufassen. Durch die Variable `$(objects)` geht das so flexibel.

Falls **gar keine Regel** angegeben ist, wird eine **implizite** Regel gesucht.

4.3.6 Mehrere Ziele in einer Regel

Man kann auch **mehrere Ziele** in einer Regel haben. Das macht in zwei Fällen Sinn:

- man gibt **nur Vorbedingungen** an und keine Anweisungen
- für alle Ziele gibt es **ähnliche Anweisungen**

Das **erstere** setzt voraus, daß man **auch mehrere Regeln** für die Ziele hat.

Das **letztere** benötigt normalerweise **automatische Variablen**, um den aktuellen Dateinamen in die Regel einzusetzen. Beispiel:
`Makefile-multtarget`:

```
bigoutput littleoutput : text.g
        generate text.g -$(subst output,,$@) > $@
```

ist äquivalent zu:
`Makefile-multtarget2`:

```
bigoutput : text.g
        generate text.g -big > bigoutput
littleoutput : text.g
        generate text.g -little > littleoutput
```

Die automatische Variable `$@` und die verwendete Textersetzungsfunktion `subst` wird später erläutert.

4.3.7 Statische-Muster-Regeln

Eine **flexiblere Variante** von mehreren Zielen in einer Regel sind Statische-Muster-Regeln. Hier müssen die **Vorbedingungen nicht** mehr **identisch** sein, **sondern** nur noch **analog**.

Die Syntax ist:

ziele : *ziel-muster* : *vorbedingungen*
        anweisung
        ...

Das Neue ist hier das **Ziel-Muster**. Es enthält ein **Prozent-Zeichen %** als **Wildcard**. Mithilfe dieses Wildcards **muß** das Ziel-Muster **auf jedes** der **Ziele passen**.

Der **Vorteil** ist, daß man das **Prozent-Zeichen auch in** den **Vorbedingungen** verwenden darf, damit kann man die **Liste der Vorbedingungen an** das jeweilige **Ziel**

anpassen. Beispiel:
`Makefile-statpat`:

```
objects = foo.o bar.o

all: $(objects)

$(objects): %.o: %.c defs.h
        cc -c -o $@ $<
```

Hier hängt foo.o ab von **foo.c** und **defs.h**, und es hängt bar.o ab von bar.c und defs.h.

In das Muster `%.c` wird also jeweils der **„Stamm“** des Ziels **eingesetzt**. Der Stamm ist der **Teil des Ziels**, der auf das **Prozent-Zeichen matcht**.

Es ist auch erlaubt, daß das **Prozent-Zeichen** gar **nicht** in einer Vorbedingung **vorkommt**. Was passiert dann? Dann ist diese Vorbedingung **für alle Ziele gleich**.

Wir sehen wieder die Verwendung von **automatischen Variablen**, um auch die Anweisung an das Ziel und die Vorbedingungen anzupassen. `$@` wird durch das **aktuelle Ziel** ersetzt, `$<` durch die **aktuelle erste Vorbedingung**. Später dazu mehr.

Ein weiteres Beispiel:
`Makefile-statpat2`:

```
bigoutput littleoutput : %output : text.g
        generate text.g -$* > $@
```

Hier sind **zwar** die **Vorbedingungen konstant**, **aber** wir haben eine **elegante** Möglichkeit, die **Art der Generierung** zu steuern. Die automatische Variable `$*` enthält den **aktuellen Stamm** des Ziels.

Was ist, wenn ein **Prozent-Zeichen im Dateinamen** vorkommt? Mit **Backslash quoten**. Was ist, wenn ein **Backslash im Dateinamen** vorkommt? **MSDOS**-Dateinamen!

Daher etwas **ungewöhnliche Quote-Regeln**:

- ein **Backslash vor** einem **Prozent-Zeichen** quotet dieses
- **zwei Backslashes vor** einem Prozent-Zeichen ergeben einen Backslash und ein normales Wildcard
- ein **Backslash an anderer Stelle** bleibt einfach erhalten

Beispiel:

`C:\Prog\\100\%\In\\%.c`
ergibt
`C:\Prog\\100%\In\`Stamm`.c`

Anmerkung: MSDOS und Windows erlauben wohl doch kein „%“, wohl aber Unix!

Eine Statische-Muster-Regel hat **Ähnlichkeit mit** einer **impliziten Regel**. **Beide** haben ein **Muster** für das Ziel und für die Vorbedingungen. (Zu impliziten Regeln später mehr.)

Der **Unterschied** ist, ***wann*** die Regel **angewandt** wird. Eine **implizite** Regel wird angewandt, **wenn** es **keine explizite** Regel gibt. Eine **Statische-Muster**-Regel wird explizit **genau auf** die **genannten Ziele** angewandt.

Gibt es **zwei Statische-Muster-Regeln** für ein Ziel, ist das ein **Fehler**. Passen **mehrere implizite Regeln**, entscheidet die **Reihenfolge** der Definition, welche davon angewandt wird.

Wenn man **nicht** so **genau weiß, welche Dateien** es im Verzeichnis gibt, ist eine explizite **Statische-Muster-Regel sicherer**. Z.B. kann eine `.o`-Datei sowohl aus einer `.c`-Datei als auch aus einer `.cpp`-Datei erzeugt werden.

Und man kann mit einer Statischen-Muster-Regel **gut Sonderfälle behandeln**, für die die impliziten Regeln nicht passen.

4.4 Anweisungen in Regeln

Eine Anweisung muß **mit** einem **Tab beginnen**.

Leere Zeilen und Zeilen nur mit **Kommentaren** dürfen **zwischen Anweisungszeilen** stehen, sie werden **ignoriert**.

Zeilen mit nur einem Tab sind ***keine*** **leeren Zeilen**, sondern **leere Anweisungen**. Beachte, daß von allen Regeln für ein Ziel nur eine davon Anweisungen haben darf!

4.4.1 Anweisungs-Ausführung

Anweisungen werden **immer von `/bin/sh` ausgeführt**, selbst wenn die Login-Shell oder die Environment-Variable `$SHELL` auf etwas anderes gesetzt ist.

Jede Anweisung wird in einer eigenen Shell ausgeführt:
`Makefile-shell`:

```
# vermeide Probleme mit symb. Links im Pfad:
cd=cd -P

test1:
	pwd
# FALSCH:
	$(cd) editor-simple
	pwd

test2:
	pwd
# RICHTIG:
	$(cd) editor-simple ; pwd
```

▶▶ `make -f Makefile-shell test1`

▶▶ `make -f Makefile-shell test2`

4.4.2 Anweisungs-Echo

Normalerweise **druckt** `make` **jede Anweisung** vor ihrer Ausführung.

Das kann man mit `@` vor einer Anweisung verhindern:
`Makefile-print`:

```
all:
	# .......
	@echo "**** Kompilation erfolgreich beendet ****"
```

▶▶ `make -f Makefile-print`

Wenn die **Anweisung** ohnehin **nur druckt**, will man sie nicht sehen.

Anmerkung: Die **Kommentarzeile *wurde* gedruckt**, weil sie mit **Tab beginnt** und deshalb eine **(leere) Anweisung** ist. Die Shell ignoriert alles ab dem Kommentarzeichen, aber **Drucken und Shell-Aufruf finden statt**.

Wenn man `make` mit dem **Parameter** `-n` aufruft, dann werden die Kommandos **nur gedruckt**, nicht ausgeführt. In diesem Falle werden **auch** `@`**-Zeilen** gedruckt.

▶▶ `make -f Makefile-print -n`

4.4.3 Parallele Ausführung

Normalerweise führt `make` **eine Anweisung zur Zeit** aus. Mit der **Option** `-j` nutzt es **Nebenläufigkeit** in den Abhängigkeiten aus, um **mehrere Anweisungen gleichzeitig** auszuführen. Gut auf **Mehrprozessormaschinen**.
`Makefile-par`:

```
targets = prog1 prog2 prog3

all: $(targets)

$(targets) : prog% :
	@echo "Starting to generate $@..."
	@sleep $*
	@echo "...continueing to generate $@"
	@sleep 1
	@echo "Completed $@."
```

▶▶ `make -f Makefile-par`

▶▶ `make -f Makefile-par -j`

Gibt man hinter `-j` eine **Zahl** an, so ist das die **max. Anzahl** der **Jobs**.

▶▶ `make -f Makefile-par -j 2`

Ein **Problem** bei paralleler Bearbeitung: **Nur ein Job** kann **Stdin** haben. Die **anderen** Jobs bekommen eine **ungültige Eingabe**. Potentiell **interaktive Kommandos** kann man also **nicht parallelisieren**. **Welcher** Job **das eine Stdin** bekommt, ist ziemlich **zufällig**.

Mit der **Option** `-l` kann man die **max. Systemlast** angeben, bei der noch parallelisiert werden soll:

▶▶ `make -f Makefile-par -j -l 2.5`

▶▶ `make -f Makefile-par -j -l 0.1`

4.5 Fehlerbehandlung

4.5.1 Returncodes

Nach Ende **jeder Anweisung prüft** `make` den **Return-Code**. Falls ein **Fehler** aufgetreten ist, **bricht** `make` die **aktuelle Regel ab**, und normalerweise **auch** die **gesamte Bearbeitung**.

`Makefile-error`:

```
targets = prog1 prog2 prog3

all: $(targets)

$(targets) : prog% :
	@echo "Starting to generate $@..."
	@sleep $*
	@echo "...hitting some error for $@" ; exit 42
	@echo "Completed $@."
```

▶▶ `make -f Makefile-error`

Wenn man seine **Kompilation testen** will, dann kann man `make` sagen, daß es **so weit wie möglich weitermachen** soll, mit Hilfe der **Option** `-k` bzw. `--keep-going`:

▶▶ `make -f Makefile-error -k`

4.5.2 Ignorieren von Fehlern

Manchmal ist ein **Scheitern nicht schlimm**, oder Programme geben **falsche Return-Codes** zurück. Dann kann man `make` anweisen, den Return-Code zu **ignorieren**, indem man ein **Minus** `-` der Anweisung **voranstellt**:

`Makefile-error2`:

```
prog = foo

.PHONY: all clean

all:

clean:
	-rm $(prog)
	@echo "Wir machen trotzdem weiter..."
```

▶▶ `make -f Makefile-error2 clean`

4.5.3 Inkonsistente Zieldateien bei Fehlern oder Unterbrechungen

Wenn eine **Anweisung** mit einem Fehler **abbricht**, dann kann die **Zieldatei in** einem **inkonsistenten Zustand** sein:

`Makefile-error3`:

```
target = generated_web_page.html

all: $(target)

$(target):
	@echo "Starting to generate $@..."
	@echo "<html><body>" > $@
	@echo "<p>bla bla" >> $@
	@sleep 2
	@echo "...hitting some error for $@" ; exit 42
	@echo "</body></html>" >> $@
	@echo "Completed $@."

clean:
	-rm $(target)
```

▶▶ `make -f Makefile-error3`

▶▶ `cat generated_web_page.html`

Ein **neuer Aufruf** von `make` **korrigiert** den Fehler ***nicht***, weil die **Datei** nun **existiert:**

▶▶ `make -f Makefile-error3`

Gibt man das **spezielle Ziel** `.DELETE_ON_ERROR` an, dann werden solche Dateien **gelöscht**:

`Makefile-error4`:

```
target = generated_web_page.html

.DELETE_ON_ERROR:

all: $(target)

$(target):
	@echo "Starting to generate $@..."
	@echo "<html><body>" > $@
	@echo "<p>bla bla" >> $@
	@sleep 2
	@echo "...hitting some error for $@" ; exit 42
	@echo "</body></html>" >> $@
	@echo "Completed $@."

clean:
	-rm $(target)
```

▶▶ `make -f Makefile-error3 clean`

▶▶ `make -f Makefile-error4`

Aus **historischen Gründen** ist das **nicht** das **Default**.

Wenn man `make` **abbricht**, z.B. durch **Ctrl-C**, dann kann es ebenfalls eine **inkonsistente Zieldatei** geben.

Hier ist das **Löschen** der aktuellen Zieldatei bereits **Default**.

▶▶ `make -f Makefile-error3` und Drücken von Ctrl-C

In beiden Fällen wird **nur gelöscht**, falls die Datei **seit Beginn der Bearbeitung** der Regel bereits **modifiziert** wurde. **Anderenfalls** wird die konsistente, aber veraltete Datei **einfach belassen**.

Manchmal ist eine inkonsistene Datei immer noch **besser als gar keine**. Dann verhindert das **spezielle Ziel** `.PRECIOUS`, daß alle Dateien, von denen es abhängt, gelöscht werden: `Makefile-error5`:

```
target = generated_web_page.html

.PRECIOUS: $(target)

all: $(target)

$(target):
	@echo "Starting to generate $@..."
	@echo "<html><body>" > $@
	@echo "<p>bla bla" >> $@
	@sleep 2
	@echo "...hitting some error for $@" ; exit 42
	@echo "</body></html>" >> $@
	@echo "Completed $@."

clean:
	-rm $(target)
```

▶▶ `make -f Makefile-error5`

▶▶ `cat generated_web_page.html`

▶▶ `make -f Makefile-error5 clean`

4.6 Rekursive Verwendung von `make`

Bei größeren Projekten hat man **oft mehrere Unterverzeichnisse**, in denen jeweils übersetzt werden soll. Dann soll das `make` im oberen Verzeichnis rekursiv `make`s in den Unterverzeichnisssen aufrufen. Beispiel:

▶▶ `cd subdirs`

▶▶ `ls`

`Makefile`:

```
subdirs = doc src test
clean_subdirs = $(patsubst %, clean_%, $(subdirs))

.PHONY: all clean $(subdirs) $(clean_subdirs)

all: $(subdirs)

$(subdirs):
	cd $@ && $(MAKE)

test: src

clean: $(clean_subdirs)

$(clean_subdirs): clean_% :
	cd $* && $(MAKE) clean
```

Hier haben wir **drei Unterverzeichnisse**. Für das **Default-Ziel** `all` sollen **alle drei rekursiv** mit `make` bearbeitet werden. (Dabei soll `test` **erst nach** `src` bearbeitet werden.)

Wir machen das, indem wir per `cd` **in** das **Unterverzeichnis** wechseln und dort nochmal `make` **aufrufen**.

Die **Variable** `$(MAKE)` enthält dabei den Namen des laufenden `make`-Programms. Das ist **besser als direkt** `make` zu sagen, weil man so auch ein anderes `make`-Programm nehmen kann.

Man **beachte** die `&&` nach dem `cd`. Das zweite Shell-Kommando wird **nur ausgeführt**, wenn das erste **keinen Fehler** hatte. (Shell-Feature)

Wieder die **automatische Variable** `$@` benutzt.

Wir haben hier **drei getrennte Regeln** für die Unterverzeichnisse, um ggf. **Parallelität** ausnutzen zu können.

Für das **Aufräumen tricksen** wir hier etwas: Mittels der Funktion `patsubst` erzeugen wir drei Ziele `clean_doc`, `clean_src` und `clean_test`, für die wir dann eine **Statische-Muster-Regel** definieren. In dieser verwenden wir dann die **automatische Variable** `$*`, um auf den **Stamm** und damit auf den eigentlichen Verzeichnisnamen zugreifen zu können. Danach geht es wie gehabt **rekursiv weiter**.

In den **Unterverzeichnissen** wird dann entweder die **eigentliche Arbeit** getan:
`doc/Makefile`:

```
doc = doc.txt

.PHONY: all clean

all: $(doc)

$(doc):
	cat /dev/null > $@

clean:
	-rm $(doc)
```

`test/Makefile`:

```
.PHONY: all clean

all:
	@echo "Performing tests..."

clean:
	@echo "Cleaning up tests..."
```

Oder es geht **noch eine Rekursionsstufe** weiter:

`src/Makefile`:

```
subdirs = prog1 prog2
clean_subdirs = $(patsubst %, clean_%, $(subdirs))

.PHONY: all clean $(subdirs) $(clean_subdirs)

all: $(subdirs)

$(subdirs):
	cd $@ && $(MAKE)

clean: $(clean_subdirs)

$(clean_subdirs): clean_% :
	cd $* && $(MAKE) clean
```

`src/prog1/Makefile`:

```
prog = prog1

.PHONY: all clean

all: $(prog)

$(prog): defs1.h
	cat /dev/null > $@

clean:
	-rm $(prog)
```

`src/prog2/Makefile`:

```
prog = prog2

.PHONY: all clean

all: $(prog)

$(prog): defs2.h
	cat /dev/null > $@

clean:
	-rm $(prog)
```

▶▶ `make`

▶▶ `make`

▶▶ `make clean`

4.6.1 Details der Variablen `MAKE`

Was passiert mit der **Rekursion, wenn** man `make -n` aufruft, um einen **Probelauf** durchzuführen?

▶▶ `make -n`

Die **Rekursion** wird **trotzdem** ausgeführt! Warum?

Zeilen, die `$(MAKE)` oder `${MAKE}` **enthalten**, werden trotz `-n` ausgeführt.

Das gleiche gilt für die **Option** `-t` bzw. `--touch`. Bei dieser Option werden die Anweisungen **ebenfalls nicht ausgeführt**, aber die **Daten der Ziele** ggf. trotzdem per `touch`-Befehl **aktualisiert**:

▶▶ `make`

▶▶ `touch src/prog1/defs1.h`

▶▶ `make -t`

Was ist, wenn man **für** die **Rekursion** unbedingt **Anweisungen** ausführen lassen muß, diese aber **nicht** `$(MAKE)` oder `${MAKE}` **enthalten**? **Zeilen,** die **mit + beginnen,** werden genauso behandelt:

`Makefile2`:

```
my_very_special_make = make

subdirs = doc src test
clean_subdirs = $(patsubst %, clean_%, $(subdirs))

.PHONY: all clean $(subdirs) $(clean_subdirs)

all: $(subdirs)

$(subdirs):
	echo "Nun geht's ab ins Unterverzeichnis..."
	+cd $@ && $(my_very_special_make)

test: src

clean: $(clean_subdirs)

$(clean_subdirs): clean_% :
	cd $* && $(MAKE) clean
```

▶▶ `make -f Makefile2 -n`

Man **beachte,** daß die `echo`**-Anweisung** vor der `+`-Zeile ***nicht*** **ausgeführt** wird, weil sie kein Plus trägt.

4.6.2 Optionen und Variablen an Sub-`make`s übergeben

Woher weiß das **Sub-**`make` , daß das oberste `make` mit der **Option** `-n` oder `-t` aufgerufen wurde? Die `make`-Variable `MAKEFLAGS` wird ins Environment exportiert und vom Sub-`make` gelesen:

▶▶ `cd ..`

`Makefile-flags1`:

```
.PHONY: all

all:
	+echo "Die Make-Flags auf Ebene $(MAKELEVEL) sind >$(MAKEFLAGS)<"
	$(MAKE) -f Makefile-flags2
```

Über die **Environment-Variable** `MAKELEVEL` wird den Sub-`make`s mitgeteilt, die **wievielte Rekursionsebene** gerade aktuell ist. Auf der **obersten Ebene** ist sie **Null**. Das ist **nützlich**, um zu testen, **ob** das Makefile **rekursiv oder direkt** aufgerufen wurde.

`Makefile-flags2`:

```
CC=gcc -c
LD=gcc

.PHONY: all

all:
	+echo "Die Make-Flags auf Ebene $(MAKELEVEL) sind >$(MAKEFLAGS)<"
	+echo "Weitere Variablen sind CC: >$(CC)< und LD: >$(LD)<"
```

▶▶ `make -f Makfile-flags1`

Die **Option** `-w` bzw. `--print-directory` sagt, daß `make` am Anfang und Ende ausgeben soll, **in welchem Directory** es arbeitet. Sie wird **bei** der **Rekursion automatisch** dazugefügt.

▶▶ `make -f Makfile-flags1 -n`

Mehrere Optionen werden **in Kurzform** überreicht:

▶▶ `make -f Makfile-flags1 --keep-going --silent`

Die **Option** `-s` bzw. `--silent` **unterdrückt die Ausgabe der** aktuell ausgeführten **Anweisung**.

Man kann auch **weitere** Werte **über Environment-Variablen** per `export` weiterreichen. Aber im allg. **besser** ist der **folgende**, explizite **Weg**:

▶▶ `make -f Makfile-flags1 --keep-going --silent 'CC=jcc' 'LD=jld'`

Diese Definitionen werden also per `MAKEFLAGS` weitergereicht.

Hier haben wir einen **anderen Compiler und Linker** auf der Kommandozeile spezifiziert.

Variablendefinitionen auf der Kommandozeile gehen vor Definitionen im Makefile! Dies gilt **allgemein** und ist oft sehr nützlich.

Die Variable `MAKEFLAGS` gibt es bei Gnu-`make`, aber nicht bei allen `make`s. Sonst muß man die traditionelle Methode verwenden:
`Makefile-flags1b`:

```
.PHONY: all

all:
	+echo "Die Make-Flags auf Ebene $(MAKELEVEL) sind >$(MAKEFLAGS)<"
	$(MAKE) -f Makefile-flags2 $(MFLAGS)
```

Die **Variable** `$(MFLAGS)` ist die traditionelle Art, Optionen weiterzugeben und geht auch bei Gnu-`make`:

▶▶ `make -f Makfile-flags1b --keep-going --silent`

Nachteil: **Variablendefinitionen** auf der Kommandozeile werden bei diesen anderen `makes` **nicht rekursiv** übergeben.

Die **Option** `-j` (Grad der **Parallelität**) ist ein **Sonderfall**: Die verschiedenen `makes` **sprechen sich direkt ab**, wieviele Anweisungen gerade ausgeführt werden. **Sonst** wäre die Anzahl **schwer zu begrenzen**. Dabei **zählen** die `makes` selbst **nicht mit**. Sonst würden evtl. alle Slots nur für die `makes` aufgebraucht.

4.7 Muster-Regeln

Außer normalen expliziten Regeln und Statischen-Muster-Regeln gibt es auch **Muster-Regeln**. Sie sieht aus **wie** eine **Statische-Muster-Regel**, aber der **erste Teil** mit der expliziten Liste der Ziele **fehlt**:
`Makefile-patrule`:

```
prog: foo.o bar.o
        cc -o prog foo.o bar.o

%.o : %.c
        cc -c $< -o $@
```

Wenn hier **gesucht** wird, **wie ein** `foo.o` **erzeugt** werden kann, dann **paßt** diese Regel, **vorausgesetzt,** es steht ein `foo.c` **zur Verfügung**.

Das **Prozent-Zeichen** paßt wie bei den Statischen-Muster-Regeln auf den **Stamm** des Dateinamens, beide Prozent-Zeichen werden durch den **gleichen Text** ersetzt.

Auf der **rechten Seite** dürfen auch **mehrere Vorbedingungen** stehen.

Falls eine Vorbedingung **kein Prozent-Zeichen** enthält, dann ist sie für **alle** Ziele **gleich**. Dies ist nur manchmal sinnvoll, da die Anwendung nicht so genau eingeschränkt ist wie bei Statischen-Muster-Regeln.

Die **Reihenfolge** der Muster-Regeln ist **wichtig**. Es wird im wesentlichen die **erste Regel** genommen, die paßt.

Wenn eine Muster-Regel **mehrere Ziele** hat, wird das **anders** behandelt **als bei** einer **Statischen-Muster-Regel**:
`Makefile-patrule2`:

```
# Quellen sind foo.y und foo.l

foo: foo.tab.o foo.o

%.tab.c %.tab.h : %.y
        # Option --defines schreibt auch *.tab.h:
        bison --defines $<

%.c : %.l %.tab.h
        lex -t $< > $@
```

Hier erzeugt der Parser-Generator **Bison** außer der Datei `foo.tab.c` **durch** die **Option** `--defines` auch eine **Datei** `foo.tab.h` mit Definitionen, die **für** den Scanner-Generator **Lex benötigt** werden.

`make` **erkennt**, daß die eine Anweisung gleich **zwei Ziele** erzeugt, und berücksichtigt das bei der Ausführung: Der Parser-Generator **Bison** wird **nur einmal** aufgerufen.

4.8 Altmodische Suffix-Regeln

In vielen Makefiles sieht man sog. **Suffix-Regeln**. Man braucht sie, weil **andere** `makes` und **alte** `makes` Muster-Regeln **nicht unterstützen**. Muster-Regeln sind **flexibler**.
`Makefile-suffix`:

```
prog: foo.o bar.o
        cc -o prog foo.o bar.o

.c.o:
        cc -c $< -o $@
```

Dies ist **äquivalent** zur schon oben gezeigten Muster-Regel:
`Makefile-patrule`:

```
prog: foo.o bar.o
        cc -o prog foo.o bar.o

%.o : %.c
        cc -c $< -o $@
```

Man beachte, daß die **Reihenfolge** der **Suffixe andersherum** ist als in der Muster-Regel.

Es ist auch das **leere Suffix** als Ziel erlaubt, dann hat die Regel **nur ein Suffix**:
`Makefile-suffix2`:

```
foo: foo.o

.o:
# % : %.o
	cc -o $@ $<

.c.o:
# %.o : %.c
	cc -c $< -o $@
```

Suffix-Regeln dürfen **keine Vorbedingungen** haben. Versucht man **trotzdem**, eine anzugeben, wird es als **normale Regel** mit einem **merkwürdigen Ziel-Namen** interpretiert: `Makefile-suffix3`:

```
prog: foo.o bar.o
	cc -o prog foo.o bar.o

.c.o: defs.h    # Falsch!
	cc -c $< -o $@
```

Suffix-Regeln funktionieren mit einer **Liste bekannter Suffixe**. Die bekannten Suffixe sind die **Vorbedingungen** des **besonderen Ziels** `.SUFFIXES`: `Makefile-suffix4`:

```
.SUFFIXES: .csp

prog: foo.o bar.o
	cc -o prog foo.o bar.o

.csp.o:
	csp-compile -c $< -o $@
```

Gibt man Vorbedingungen hierfür an, werden sie **zur Standardliste dazugefügt**. **Sonderbehandlung:** Eine Regel für `.SUFFIXES` **ohne Vorbedingungen löscht** die aktuelle Liste. **Danach** kann man **neue** Vorbedingungen definieren.

4.9 Automatische Variablen

Ein paar der automatischen Variablen sind schon vorgestellt worden. Hier nun eine **komplette Liste**:

Automatische Variablen

`$@` **Ziel** der Regel.
Falls Archivmitglied, dann Name des Archivs.
Falls mehrere Ziele, dann regelauslösendes Ziel.

`$%` Vollständiges **Ziel, falls Archivmitglied**.
Falls Ziel kein Archivmitglied, dann leer.

`$<` **Erste Vorbedingung** dieser Regel.

`$?` Alle **neueren Vorbedingungen**.
Falls eine Vorbedingung Archivmitglied ist, dann nur Mitgliedsname.

`$^` **Alle Vorbedingungen**.
Falls eine Vorbedingung Archivmitglied ist, dann nur Mitgliedsname.
Doppelnennungen werden eliminiert.

`$+` **Alle Vorbedingungen**, wie `$^`, aber **Doppelnennungen werden nicht eliminiert**.

`$*` **Stamm** des Namens in der Regel.

`$(@D)`, `$(%D)`, `$(<D)`, `$(?D)`, `$(^D)`, `$(+D)`, `$(*D)`: **Directory-Teil** des Namens

`$(@F)`, `$(%F)`, `$(<F)`, `$(?F)`, `$(^F)`, `$(+F)`, `$(*F)`: **Datei-im-Directory-Teil** des Namens

Kurs Unix-Tools, WS 01/02, Jan Bredereke 25

Einschub: Archivmitglieder als Ziel.

```
foolib(hack.o) : hack.o
        ar cr foolib hack.o
```

Das **fügt** hack.o **in** das **Archiv** foolib ein. (**„r“** schreibt, **„c“** stellt Warnungen ab, wenn noch nicht drin.) Es **gibt schon** eine **implizite Regel**, die genau dies tut.

Man kann diese Archiv-Notation **in Zielen und in Vorbedingungen** anwenden, aber **nicht in Anweisungen**. Dort helfen aber die automatischen Variablen.

Man darf bei der Archiv-Notation ggf. auch **mehrere Mitgliedsnamen** in die runden Klammern schreiben, getrennt durch Blanks.

Bei `$@` können **mehrere Ziele** vorkommen, wenn wir eine **Muster-Regel** haben.

`$%` ist **Gnu-`make`-spezifisch**.

`$+` ist nützlich, wenn es auf die Reihenfolge von Bibliotheken beim Linken ankommt.

Bei `$*` hängt die Definition des „Stamms“ von der Art der Regel ab:

Statische-Muster-Regel: Genau das, was auf `%` **matcht**.

Muster-Regel: Das, was auf `%` **matcht**, ggf. mit unverändertem **Directory-Pfad davor**.
Aus dir/a.foo.b und dem Muster a.%.b wird dir/foo .

Explizite Regel: Sonderbehandlung: Hier gibt es keinen Stamm. Daher **Ziel minus bekannte Suffixe**. Nur aus Kompatibilitätsgründen, man sollte es **vermeiden**.

Die **D/F-Formen** sind **Gnu-`make`-spezifisch**.

Beispiele:

```
$@ = dir/foo.o
$(@D) = dir
$(@F) = foo.o

$@ = bar
$(@D) = .
```

Es ist **kein Slash am Ende** des Directory-Teils.

Bei `$%` bezieht sich der Directory-Teil auf ***innerhalb* des Archivs**.