Kapitel 5

GNU `make`

5.1 Einführung

Bei der Erstellung von Programmen durchläuft man (auf der praktischen Seite) immer wieder denselben Vorgang:

1. Programm-Sourcecode erstellen/ändern
2. Einzelne Sourcefiles zu Objectcode-Files kompilieren
3. Die Objektcode-Files zum Programm zusammenlinken

Bei größeren Projekten kann man eine ganze Reihe von Sourcefiles haben. Bei jeder Änderung nun alle Sourcefiles neu zu kompilieren und die entstandenen Objectcode-Files zu linken ist damit eine ziemlich teure Angelegenheit[1]. Deshalb möchte man nach einer Änderung am Sourcecode nur die Files neu kompilieren, die von dieser Änderung betroffen sind. Als Beispiel sind in Abbildung 5.1 die Abhängigkeiten eines winzigen Projektes dargestellt. Dabei bedeutet z.B. der Pfeil von `foo.c` zu `foo.o`, daß das Objectcode-File `foo.o` vom Sourcefile `foo.c` abhängt, also bei einer Änderung von `foo.c` das File `foo.o` neu erzeugt werden muß.
Man sieht hier auch, daß eine Änderung in einem File, etwa `bar.h`, die Neugenerierung von mehreren Files, `foo.o` und `baz.o`, nötig macht.
Obwohl Abhängigkeiten in Form von Sourcecode–Objectcode Beziehungen die am häufigsten auftretenden sind, gibt es auch andere Formen von Abhängigkeiten und Generierung von Files aus anderen (Source-)Files. Z.B. steht dieses Script

[1]Auf einem 133MHz Pentium Rechner dauert z.B. die komplette Neugenerierung eines Betriebssystem-Kernels für NetBSD eine gute halbe Stunde. Dabei ist dieses Projekt mit 459 resultierendes Objectcode-Files noch relativ klein.

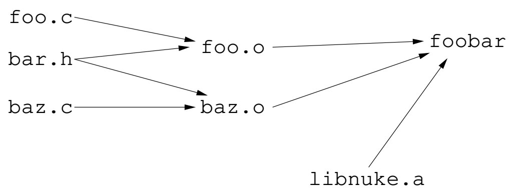

Abbildung 5.1: Abhängigkeiten in einem Projekt

```
foobar:  libnuke.a foo.o baz.o
       gcc -o foobar foo.o baz.o -lnuke

foo.o:  foo.c bar.h
       gcc -o foo.o foo.c -I../include -c

baz.o:  baz.c bar.h
       gcc -o baz.o baz.c -I../include -c
```

Abbildung 5.2: `Makefile` Beispiel

als Postscript-Datei im Netz zur Verfügung. Diese Postscript-Datei wurde mittels `dvips` aus einem `DVI`[2]-File erzeugt. Dieses wiederum ist die Ausgabe des `latex` Tools, das eine Datei mit Text und Formatierungsbefehlen, `Script.tex` als Eingabe nimmt.
GNU `make` ist ein Tool, mit welchem man die Abhängigkeiten zwischen Files und Kommandos zur Generierung von Files aus anderen Files angeben kann. Darüberhinaus bietet es auch noch die Möglichkeit, eigene *Variablen* zu definieren, die in Kommandos und Abhängigkeiten ausgewertet werden, sowie eine Reihe eingebauter *Funktionen*, mit deren Hilfe man etwa Filenamen umformen kann. Als Eingabe nimmt `make` ein einzelnes File mit dem Namen `Makefile`, in welchem alle Abhängigkeiten und Kommandos abgelegt sind.
In Abbildung 5.2 sieht man das zu den Abhängigkeiten in Abbildung 5.1 korrespondierende `Makefile`.
Neben dem GNU `make` gibt es eine ganze Reihe weiterer `make`-Varianten. Diese unterscheiden sich deutlich im Umfang ihrer features und sind nicht kompatibel zu GNU `make`. Da dieses aber den (m.E.) größten Funktionsumfang bietet, wird es hier ausschließlich vorgestellt. Man sei gewarnt, daß alles außer einfachen Regeln und Kommandos in anderen Versionen von `make` nicht so funktioniert, wie in GNU `make`!

[2] Das Ausgabeformat des LaTeX-Systems, siehe Kapitel 6

Im weiteren wird nun zunächst der Aufbau und Inhalt eines `Makefiles` dargestellt, bevor in Abschnitt 5.3 die *Regeln* genauer vorgestellt werden, mit denen man die Abhängigkeiten zwischen Files beschreibt. Im darauf folgenden Abschnitt werden die *Variablen* so wie sie von `make` bereitgestellt werden, behandelt. *Kommandos*, die für die Neugenerierung von Files verntwortlich sind, werden in Abschnitt 5.3.1 behandelt. Danach folgen noch Abschnitte über die Direktiven in einem `Makefile` und die Aufrufparameter des `make`-Tools. Danach werden die in `make` vordefinierten impliziten Regeln behandelt. Den Abschluß bilden ein Abschnitt über die `autoconf` und `automake` Tools, welche bei der Erstellung von protierbaren `makefiles` helfen, sowie Abschnitte mit Beispielen, Warnungen und etwas Literatur.

5.2 Makefiles

Die Abhängigkeiten zwischen Files und die Kommandos, um Files aus seinen Abhängigkeiten neu zu bauen, werden `make` in einem speziellen Eingabefile mitgeteilt. Gibt man `make` nicht mittels der `-f` *`File`* Option explizit den Namen dieses Files an, so wird nach einem File gesucht, welches den Namen `Makefile` oder `makefile` oder aber `GNUMakefile` trägt. Ein Makefile kann folgende Bestandteile enthalten:

- *Regeln* geben Abhängigkeiten und Kommandos an
- *Variablenzuweisungen* weisen `make`-Variablen Werte zu
- *Direktiven* steuern z.B. die konditionale Ausführung von Makefiles
- *Kommentare* werden eingeleitet durch # und reichen bis zum Ende der Zeilen, in welchen sie stehen.

Ein etwas ausführlicheres Beispiel eines `Makefiles`:

```
# Makefile f"ur Unix-Kurs Script
# Spezielles Ziel
.PHONY: all ps clean

# variablenzuweisungen
FIGSS=Abb.Libs.fig Abb.Link.fig Abb.Dyn.fig \
       Abb.Sed.Work.fig Abb.Make.fig1.fig
FIGSO=$(patsubst %.fig,%.eps,$(FIGSS))

MAG_DEFAULT=0.5
```

```
SRCS=Script.tex $(FIGSO)

# Regeln
all: Script.dvi
ps: Script.ps
# Regeln mit Kommandos
clean:
        rm -f $(OBJS) Script.ps *.dvi *.aux *.bbl \
                      *.blg *.log *.toc

Script.ps: Script.dvi
        dvips -o $@ $<

Script.dvi: $(SRCS)
        latex $<
        bibtex Script
        checklatex $<

# Implizite Regel
%.eps: %.fig
        fig2dev -L ps -p dummy -m $(word 1,$(MAG_$*) \
                $(MAG_DEFAULT)) $< >$@
```

Die Komponenten können in einem `Makefile` in beliebiger Reihenfolge auftreten. In den nun folgenden Abschnitten werden wir uns der Reihen nach den einzelnen Komponenten ausführlicher widmen.

5.3 Regeln

Regeln bestehen aus zwei Teilen

- Abhängigkeiten, welche die Abhängigkeit zwischen bestimmten Files angeben
- Kommandos, die zur Regenerierung abhängender Teile bei Änderungen dienen

Dabei dürfen die Kommandos auch weggelassen werden, wenn man nur die bloße Tatsache einer Abhängigkeit zwischen Files ausdrücken will[3]
Abhängigkeiten werden dabei durch zwei Komponenten beschrieben. Das *Ziel* ist der abhängige Teil. Die *Quellen* sind die Teile, von denen das Ziel abhängt. Aus

[3] I.A. wird es natürlich genau eine Regel geben, die Kommandos zum Bauen der Abhängigkeit enthält.

denen es also bei einer Änderung neu generiert wird. Z.B. ist in Abbildung 5.1 `baz.o` ein Ziel und `baz.c` und `bar.h` seine Quellen. Dies wird im zugehörigen `Makefile` in Abbildung 5.2 durch die Abhängigkeit

```
baz.o:  baz.c bar.h
```

ausgedrückt.

Ein Ziel kann also durchaus mehrere Quellen haben. D.h. wenn sich eine dieser Quellen ändert, muß auch das Ziel neu gebaut werden. Ebenso kann man aber in einer Abhängigkeit auch mehrere Ziele angeben. Dies dient der Abkürzung und so steht etwa die Abhängigkeit

```
baz.o foobar.o:  bar.h
```

für zwei Abhängigkeiten `baz.o: bar.h` und `foobar.o: bar.h`

Man kann in Anhängigkeiten auch Wildcards benutzen, wie z.B. in der tcsh. Diese werden dann beim Lesen des `Makefiles` gegen die Files im akt. Directory expandiert, etwa sagt `foo.o: *.h`, daß `foo.o` von allen Header-Files abhängt. Man beachte aber, daß die Expansion beim Lesen der Regeln passiert, eine Abhängigkeit der Form `foo: *.o` also nicht zum gewünschten Ergebnis (`foo` hängt von allen Objectcode-Files ab) führt, wenn die Objectcode-Files selbst noch generiert werden müssen und also beim Lesen des `Makefiles` noch nicht existieren.

Ziele und Quellen entsprechen meist realen Files, die und aus denen neu gebaut wird. Es ist aber auch möglich, Ziele zu definieren (und dann natürlich auch diese als Quellen zu benutzen), welche keinem File entsprechen. Solche Pseudo-Ziele stehen also einfach als eine Art Abkürzung für irgendwelche Tätigkeiten, die ausgeführt werden sollen. `make` ist es egal, ob die Kommandos einer Regel wirklich ein File erzeugen, das den Namen des Ziels trägt. Solange die Kommandos ohne Fehler ablaufen, geht `make` davon aus, daß die Regel das Ziel korrekt neu gebaut hat. Pseudo-Ziele benutzt man z.B. um eine Distribution „aufzuräumen", also etwa Objectcode-Files wieder zu entfernen. Dazu könnte man eine Regel der Form

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

benutzen. Ein Pseudo-Ziel, das oft benutzt wird ist das Ziel `all`, welches die Compilierung des gesamten Projektes anstösst.

Probleme kann man bekommen, wenn aus Versehen wirklich mal ein File mit dem Namen `all` erzeugt wurde. Dann führt ein `make all` nicht zur Neugenerierung des Projektes, da ein File mit dem Namen ja bereits existiert[4]

Deshalb kann man `make` mitteilen, daß bestimmte Ziele Pseudo-Ziele sind. Dazu gibt man eine Abhängigkeit für das spezielle `.PHONY` an:

[4]Wenn man voraussetzt, daß alle Abhängigkeiten jÜnger als dieses File sind.

```
.PHONY: all
```

Dies teilt `make` mit, daß es jedes File `all` beim Bauen des Zieles `all` ignorieren und die Kommandos der Regel für `all` auf jeden Fall ausführen soll.
Ein anderes spezielles Ziel, `.SUFFIXES` ist in Verbindung mit impliziten Regeln wichtig, die in einem späteren Abschnitt besprochen werden.

5.3.1 Kommandos

Kommandos folgen auf die Angabe der Abhängigkeiten in einer neuen Zeile. Genauer gesagt fasst `make` jede Zeile, die auf eine Abhängigkeitsangabe folgt und *mit einem TAB beginnt* als eine Kommandozeile auf. Ein oft gemachter Fehler ist es, Kommandozeilen nicht mit einem TAB zu beginnen, was `make` dann zu manchmal nicht offensichtlichen Fehlermeldungen veranlasst (da es die Zeile dann als Abhängigkeit auffasst). Es kann mehrere Kommandozeilen in einer Regel geben. Die Kommandos werden dann in der Reihenfolgen, in der sie im File auftauchen, ausgeführt. Jede Kommandozeile wird[5] an eine eigene Subshell übergeben, wobei als Vorgabe `/bin/sh` eingestellt ist. Da jede Zeile an eine eigene Shell zur Ausführung übergeben wird, sollte man sich bewußt sein, daß Kommandos, wie etwa `cd`, die den lokalen Status einer Shell ändern, in nachfolgenden Kommandozeilen keine Auswirkungen hinterlassen. Natürlich kann man in einer Kommandozeile einen beliebigen, von der Shell interpretierbaren Ausdruck hinschreiben, etwa

cd src; make; cd ../doc; make

Dieser wird dann von *einer* Shell abgearbeitet und zeigt auch die gewünschten Effekte.
Manchmal ist es nötig, längere Kommandos anzugeben. Der Lesbarkeit wegen kann man solche Kommandos über mehrere Zeilen verteilen, wenn man am Ende jeder Zeile einen Rückstrich '\' angibt (natürlich, bis auf die letzte Zeile). Trotzdem wird dies dann von `make` intern wieder zu einer Zeile zusammengesetzt. In Kommandos sind neben den im `Makefile` definierten Variablen noch einige spezielle Variablen verfügbar, die die Namen der Abhängigkeiten u.ä. angeben. Diese werden dann vor jeder Ausführung der Kommandos automatisch belegt und expandiert, siehe den nächsten Abschnitt.
Tritt bei der Ausführung eines Kommandos ein Fehler auf, angezeigt durch einen Rückgabewert des Kommandos $\neq 0$, so geht `make` davon aus, daß der Neubau des zugehörigen Zieles fehlgeschlagen ist[6]. Als Folge davon wird der komplette `make`-Prozess abgebrochen.

[5] Nach Variablenexpansion, etc.
[6] Etwa, weil die Compilierung eines Files wegen Syntaxfehlern in der Eingabe misslang.

Es kann manchmal sein, daß das Fehlschlagen eines Kommandos nicht unbedingt bedeutet, daß der Generierungsprozess fehlschlug, z.B. wenn in einem Kommando ein Directory (etwa `/usr/local/man`) kreiert werden soll, welches bereits existiert. Man kann daher solchen Kommandos ein '`-`' voranstellen, was zur Folge hat, daß ein Fehlschlag dieses Kommandos ignoriert wird, etwa '`-mkdir /usr/local/man`'-
`make` gibt in der Standardeinstellung jedes Kommando (nach der Expansion von Variablen) auf `stdout` aus, bevor es an die Shell zur Ausführung übergeben wird. Dies kann manchmal unerwünscht sein, weshalb man durch Vorstellen eines '`@`' vor das Kommando dieses Verhalten unterdrücken kann, etwa '`@echo "Installing..."`'

5.4 Variablen

In `make` gibt es fünf Arten von Variablen:

- Auf der Kommandozeile können Variablen gesetzt werden, etwa durch `make CC=gcc`. Dies setzt die Variableb `CC` auf den Wert `gcc`. Soclhermassen gesetzte Variablen können nicht mehr durch normale Zuweisungen im `Makefile` verändert werden. Will man die verändern, muß man die `override` Direktive benutzen, s.u.

- Für jede (globale) Umgebungsvariable, die beim Aufruf von `make` im Environment der Shell definiert war, wird eine `make`-Variable gleichen Namens und Inhaltes angelegt. Davon ausgenommen sind nur die folgenden Variablen:

  SHELL Wird immer auf `/bin/sh` gesetzt

  MAKE Wird auf die Aufrufsequenz von `make` gesetzt. Nach einem Aufruf der Form `gnumake CC=gcc all` wird `MAKE` also auf den Wert `gmake` gesetzt.

  MAKEFLAGS Enthält die Optionen und Variablen, die auf der Kommandozeile angegeben wurden. Im vorigen Beispiel würde `MAKEFLAGS` also den Wert `CC=gcc` enthalten.

- Variablen, die im `Makefile` *lazy* gesetzt werden, z.B. durch eine Zeile der Form `CFLAGS=$(FLAGS) -Wall`. Dies setzt die Variable `CFLAGS` auf den Inhalt der Variablen `FLAGS`, gefolgt von `-Wall`. Die Variablen heissen lazy, weil der Inhalt der Variablen `FLAGS` erst dann eingesetzt wird, wenn diese Variable schließlich expandiert wird. Es wird also der bei jeder Expansion gerade aktuelle Wert in `FLAGS` eingesetzt. Dies bedeutet aber auch, daß Konstrukte der Form `CFLAGS=$(CFLAGS) -Wall` verboten sind[7], da sie bei der Expansion zu unendlicher Rekursion beim Einsetzen von Variablenwerten führen.

- Variablen, die in einem `makefile` *eager* gesetzt werden, etwa wie in `CFLAGS:=$(CFLAGS) -Wall`. Hier wird der Variablen `CFLAGS` der gerade aktuelle Wert der Variablen `CFLAGS` zugewiesen, gefolgt von `-Wall`. Dadurch unterscheiden sich Variablen diesen Typs von *lazy* Variablen: Lazy Variablen expandieren ihren Wert bei jeder Expansion der Variablen, während eager Variablen ihre rechte Seite nur einmal, bei der Definiton der Variablen expandieren und dann bei jeder Expansion der Variablen diesen expandierten Wert einsetzen.
- Automatische Variablen treten nur in Kommandos auf. Sie werden bei der Anwendung einer Regel, wenn also ein Ziel aus seinen Quellen neu gebaut werden muss, automatisch auf bestimmte Werte gesetzt, etwa den Namen des Ziels (`$@`), etc. Eine komplette Liste aller automatischen Variablen folgt weiter unten.

Variablenreferenzierung wird geschrieben, indem man ein Dollarzeichen, gefolgt von dem Namen der Variablen, der im übrigen nur Buchstaben, gefolgt von Buchstaben, Unterstrichen oder Ziffern, entahlten darf, in runden Klammern, etwa `$(CFLAGS)`.

Variablen können in einem `Makefile` in rechten Seiten von Variablendefinitonen, Zielen oder Quellen in Abhängigkeiten, Kommandos und Direktiven auftreten.

`make` expandiert Variablen rekursiv, d.h. nach jeder Expansion einer Variablen wird die komplette Zeile wieder neu expandiert. Dies hat zur Folge, daß man als Namen einer Variablen wieder eine Variablenreferenz schreiben kann, etwa ist folgendes vollkommen legal und expandiert bei der Expansion der Variablen `C` schliesslich zu dem Text '`y`':

```
A=B
B=y
C=$($(A))
```

Wird `C` irgendwo benutzt, also `$(C)` expandiert, expandiert dies zunächst zu `$(B)`, da die Referenz zur Variablen `A` zuerst expandiert wird. Dies expandiert dann im zweiten Schritt zu `y`.

Es gibt einige Variablen, die eine spezielle Bedeutung haben:

MAKE

[7] Und auch in `make` zu einer Fehlermeldung führen.

MAKEFLAGS Wurden schon weiter oben erwähnt

VPATH Enthält eine Liste von Directories, durch Doppelpunkte getrennt, in denen nach Files gesucht werden soll. Hat man z.B. alle Header-Dateien in einem eigenen Verzeichnis `include` untergebracht müsste man eigentlich bei jeder Regel etwas der Form `foo: foo.o include/bar.h` schreiben. `make` sucht nun aber auch in allen Verzeichnissen, welche in `VPATH` angegeben sind, nach Quellen und Zielen, so daß nach der Zuweisung `VPATH=include` die obige Anhängigkeit einfach als `foo: foo.o bar.h` geschrieben werden kann. Neben dieser Möglichkeit bietet `make` auch noch die Option, nur nach Files mit bestimmter Endung in bestimmten Directories zu suchen. Siehe dazu die Beschreibung der `vpath` Direktive in Abschnitt 5.5.

Da man an eine lazy Variable nicht durch eine einfache Zuweisung einen Wert anhängen kann (s.o.), stellt `make` noch einen Zuweisungsoperator bereit, der Werte an die Definiton einer Variablen anhängt: '+='. So liefert etwa folgende Zuweisung den gewünschten Effekt, `-Wall` an die akt. Definition von `CFLAGS` anzuhängen:

```
CFLAGS+=-Wall
```

Möchte man einer Variablen nur dann einen Wert zuweisen, wenn sie nicht schon vorher (etwa über die Kommandozeile) definiert wurde, dann benutzt man den '`?=`' Zuweisungoperator:

```
CFLAGS ?= -Wall -O2 -g
```

Dies weißt `CFLAGS` nur dann einem Wert zu, wenn die Variable bis dahin unbekannt (nicht definiert) war.

5.4.1 Automatische Variablen

Automatische Variablen können nur in Kommandos auftauchen und erhalten ihren Wert durch die Abhängigkeiten der Regel, zu der sie gehören. So gibt es etwa eine automatische Variable `$@`, der der Wert des Zieles der Regel zugewiesen wird. `make` kennt die folgenden automatischen Variablen:

`$@` Der Name des Ziels der Regel.

`$<` Der Name der *ersten* Quelle in der Abhängigkeit.

`$?` Alle Quellen, die jünger als das Ziel sind.

`$^` Alle Quellen, getrennt durch ein Space. Aus dieser Liste werden doppelte Einträge entfernt.

`$+` Wie `$^`, nur das doppelte Einträge nicht entfernt werden.

`$*` Der Stamm einer impliziten Regel, siehe Abschnitt 5.7.

Beispiel:

```
foo: foo.o bar.o
      gcc -o $@ $<
# Resultiert in 'gcc -o foo foo.o'

bar: foo.o bar.o bar.o
      gcc -o $@ $^
# Resultiert in 'gcc -o bar foo.o bar.o'

baz: baz.o foo.o baz.o
      gcc -o $@ $+
# Resultiert in 'gcc -o baz baz.o foo.o baz.o'
```

5.4.2 Funktionen

`make` bietet eine ganze Reihe von Funktionen, die bei der Expansion von Text ähnlich wie Variablen ausgewertet werden.
Die Syntax eines Funktionsaufrufes hat dann die Form

`$(`*`Funktions-Name Argumente...`*`)`

Funktionen dienen dazu,

- Text umzuformen
- Patternmatching zu machen
- Schleifen zu realisieren
- Kommandos in einer externen Shell aufzurufen

Textmanipulation

Text wird in `make` manchmal als eine Folge von *Worten* aufgefasst (z.B. Filenamen). Worte werden dabei durch Leerzeichen oder TABs voneinander getrennt.
Zur Manipulation von Text stehen folgende Funktionen zur Verfügung:

- `$(subst `*`from`*`,`*`to`*`,`*`text`*`)` Ersetzt jedes Vorkommen von *from* in dem Text *text* durch *to*.

- `$(patsubst pattern,repl,text)` Ersetzt in jedem Wort in *text*, welches auf das Pattern *pattern* passt, dieses Pattern durch *to*. In dem Pattern kann ein '`%`' auftauchen, das eine beliebige Folge von Zeichen beschreibt. Taucht `%` in *to* auf, so wird es durch den Match ersetzt, der durch das `%` im *Pattern* vorgenommen wurde.

Als Beispiel dieses Fragment eines `Makefiles`:

```
FILES=a.c b.c c.c
OFILES=$(patsubst %.c,%.o,$(FILES))

AFILES=yacc.tab.c yacc.tab.h
ACFILES=$(subst .tab,,$(AFILES))
```

Hier expandiert `$(OFILES)` später zu `a.o b.o c.o`.
Die Variable `$(ACFILES)` expandiert zu `yacc.c yacc.h`.

- `$(strip text)` Diese Funktion entfernt allen Whitespace[8] vom Anfang und Ende des Textes.

- `$(findstring find,text)` Liefert *find*, wenn *text find* enthält, den leeren Text, sonst.

Als erläuterndes Beispiel, folgendes `Makefile`:

```
SOPTS=$(strip $(CFLAGS))
ifneq ($(SOPTS),)
 ISWALL=$(findstring -Wall,$(SOPTS))
ifneq ($(ISWALL),)
 # -Wall enabled
endif
endif
```

Hier wird zuerst die Variable `$(SOPTS)` als der Wert von `$(CFLAGS)` definiert, mit allem Whitespace am Anfang und Ende entfernt. In der zweiten Zeile wird dann geprüft, ob der Wert dieser Variablen der leere Text ist. Der Grund für die Entfernung des Whitespaces ist der, daß Text, welcher z.B. nur aus Lerrzeichen besteht verschieden ist von dem leeren Text (der einfach keine Zeichen enthält). Da es in Variablendefinitionen durchaus passieren kann, daß sich Whitespace an den Anfang oder das Ende einer Definition verirrt, muß man also vor der Prüfung auf den leeren Text erst den Whitespace entfernen.

[8] Spaces, TABs

In dem Beispiel wird dann in der dritten Zeile der Variablen `$(ISWALL)` der Wert `-Wall` zugewiesen, wenn dieser Text in `$(SOPTS)` autaucht, der leere Text sonst. Mann kann dann einfach testen, ob diese Option in den Flags gegeben wurde (4. Zeile).

- `$(filter pattern...,text)` Filtert Worte aus *text*, die auf keins der Pattern *pattern...* passen.
- `$(filter-out pattern...,text)` Filtert Worte aus *text*, die auf eins der *pattern...* passen.
- `$(sort text)` Sortiert alle Worte in *text*.

Beispiel:

```
FILES=a.c foo.h b.c baz.y
CFILES=$(filter %.c,$(FILES))
NONCHFILES=$(filter-out %.c %.h,$(FILES))
SFILES=$(sort $(FILES))
```

Hier wird `$(CFILES)` der Wert `a.c b.c` zugewiesen, da diese Worte auf das Pattern `%.c` passen. Die Variable `$(NONCHFILES)` erhält den Wert `baz.y`, da nur dieses Wort auf keins der Pattern `%.c` oder `%.h` passt. Die Variable `$(SFILES)` expandiert zu `a.c b.c baz.y foo.h`, der sortierten Wortliste.

Filenamen-Manipulation

- `$(dir Namen...)` Liefert den Directory-Teil für jeden Namen
- `$(notdir Namen...)` Liefert den Filenamen-Teil für jeden Namen

Ein Beispiel:

```
FILES=../src/foo.c bar.c nix/
DFILES=$(dir $(FILES))

FFILES=$(notdir $(FILES))
```

Hier expandiert `$(DFILES)` zu `../src ./ nix/`. Da das zweite Wort von `$(FILES)` keinen Directory-Teil hat, wird statt des leeren Textes ein `./` eingefügt.
Der Variablen `$(FFILES)` wird der Wert `foo.c bar.c` zugewiesen. Da das letzte Wort von `$(FILES)` keinen Filenamen-Teil hat, wird es durch das leere Wort ersetzt.

- `$(suffix Namen...)` Liefert den Suffix jeden Namens
- `$(basename Namen...)` Liefert den Namen ohne Suffix

Z.B.:

```
FILES=../src/foo.c bar.c nix/

SFILES=$(suffix $(FILES))
BFILES=$(basename $(FILES))
```

Hier wird `$(SFILES)` zu `.c .c`, da das letzte Wort von `$(FILES)` keinen Filenamen-Teil hat, wird seine Endung auch zum leeren Wort. Der Variablen `$(BFILES)` wird der Wert `../src/foo bar nix/` zugewiesen. Anders, als bei dem Shell-Kommando `basename` wird also der Directory-Teil nicht abgeschnitten.

- `$(addsuffix Suffix,Namen...)` Fügt *Suffix* an jeden Namen in *Namen* an.
- `$(addprefix Prefix,Namen...)` Fügt *Prefix* vor jeden Namen an.
- `$(join` *Liste*$_1$, *Liste*$_2$`)` Fügt Worte aus den Listen aneinander. Dabei wird dem ersten Wort der ersten Liste das erste Wort der zweiten Liste angehängt (ohne Whitespace). Dem zweiten Wort der ersten Liste das zweite Wort der zweiten Liste, etc. Diese neu gebildeten Wort bilden das Ergebnis der Funktion.

Beispiel:

```
FILES=a b c

CFILES=$(addsuffix .c,$(FILES))

XFILES=$(addprefix foo_,$(CFILES))
# Liefert foo_a.c foo_b.c foo_c.c

YFILES=$(join $(FILES),.c .h .c)
# Liefert a.c b.h c.c
```

Der Variablen `$(CFILES)` wird der Wert `a.c b.c c.c` zugewiesen, die Variable expandiert zu `foo_a.c foo_b.c foo_c.c`. Schließlich wird der Variablen `$(YFILES)` der Wert `a.c b.h c.c` zugewiesen.

- `$(word N,Text)` Liefert das N-te Wort aus *Text*.
- `$(wordlist S,E,Text)` Liefert das S-te bis E-te Wort.
- `$(words Text)` Anzahl Worte in *Text*.

Beispiel:

```
FILES=a b c d

NFILES=$(words $(FILES))
# Liefert 4

SECOND=$(word 2,$(FILES))
# Liefert b

SECOND_TO_LAST=$(wordlist 2,4,$(FILES))
# Liefert b c d
```

- `$(wildcard Pattern)` Expandiert das *Pattern* im aktuellen Directory, z.B. `$(wildcard *.c)`

Schleifen

- `$(foreach Var,Liste,Body)` Bindet die Variable *Var* nacheinander an die Worte aus *Liste* und fügt die Auswertung von *Body* an das Ergebnis an.

Beispiel:

```
FILES=a b c d
XFILES=$(foreach x, $(FILES), $(x).c)
# Liefert a.c b.c c.c d.c
```

Shell & Sonstiges

- `$(origin var)` Liefert eine Zeichenkette, die den Typ der Variablen *var* angibt. Folgende Typen können auftreten:

  undefined Die Variable wurde nie definiert.

  default Die Variable hat einen voreingestellten Wert.

  environment Die Variable kommt aus dem Environment.

  file Die Variable wurde in einem `Makefile` definiert.

command line Die Variable wurde auf der Kommandozeile definiert.

override Der Variablen wurde mittels der `override` Direktive ein Wert gegeben, siehe Abschnitt 5.5.

automatic Die Variable ist eine automatische Variable.

- `$(shell` *`Args`*`)` Führt das Kommando *Args* in einer Subshell aus und liefert dessen Ausgabe als Ergebnis.

Beispiel:

```
ifneq ($(origin PATH),environment)
PATH=$(shell echo $$PATH)
endif
```

5.5 Direktiven

Direktiven dienen dazu

- Nur Teile eines `Makefiles` einzulesen, abhängig von bestimmten Bedingungen.
- Andere `Makefiles` einzubinden
- Variablenwerte zu überschreiben oder deren Gültigkeitsbereich zu ändern
- Suchpfade für Quellen zu setzen

Direktiven stehen an den Stellen, an denen auch Variablenzuweisungen oder Regeln stehen dürfen.

5.5.1 Conditionale

Conditionale haben die folgende generelle Form:
`if`*Bedingung Argumente*
True-Teil
`else`
False-Teil
`endif`
Falls die *Bedingung* zutrifft, wird der *True-Teil* verwendet (gelesen), ansonsten der *False-Teil*. Hierbei kann das `else` und der *False-Teil* auch fehlen. Natürlich kann man Conditionale auch schachteln.
Es sei angemerkt, daß `make` die Auswertung beim Einlesen des `Makefiles` vornimmt, nicht zur Auswertungszeit der Regeln.
Es gibt folgende Bedingungen für die Conditionale:

ifdef ***Variable*** Diese Bedingung ergibt 'wahr', falls die Variable *Variable* definiert ist.

ifndef ***Variable*** Diese Bedingung ergibt 'wahr', falls keine Variable *Variable* definiert ist.

ifeq (*A*,*B*) Ergibt 'wahr', falls die Texte *A* und *B* gleich sind.

ifneq (*A*,*B*) Ergibt 'wahr', falls die Texte *A* und *B* ungleich sind.

Bei den letzten beiden Conditionalen kann man statt `(A,B)` auch noch die folgenden Formen benutzen, die äquivalent sind:

- `"A" "B"`
- `'A' 'B'`

Beispiel:

```
ifdef CC
 # Variable CC definiert
 ifeq ($(CFLAGS),)
    # CFLAGS ist leer
  CFLAGS=-Wall -O3 -g
 else
    # CFLAGS nicht leer
  CFLAGS+=-Wall
 endif
endif
```

5.5.2 Direktiven für Variablen

Variablen, die auf der Kommandozeile von `make` gesetzt wurden, können normalerweise nicht mehr durch Variablenzuweisungen überschrieben werden, d.h. Zuweisungen an sie haben keinen Effekt. Möchte man jetzt aber sichergehen, daß eine Variable einen bestimmten Wert zugewiesen bekommt, kann man die `override` Direktive benutzen. Sie kann in drei Formen auftreten:

override ***Var=Value***

override ***Var:=Value***

override ***Var+=Value***

Diese Varianten entsprechen den gleichnamigen Zuweisungsoperatoren auf Abschnitt 5.4. Der Effekt ist der, daß die Variable auf jeden Fall den Wert auf der rechten Seite der Zuweisung erhält, auch wenn sie vorher auf der Kommandozeile definiert wurde. Da dieses Feature die Möglichkeiten des Benutzers, Variablen auf fixe Werte zu zwingen (etwa mittels `CC=gcc` den Compiler auf den GCC festzulegen) torpediert, sollte es nur vorsichtig eingesetzt werden.
Ruft man in einem `Makefile` `make` wieder via eines Kommandos auf, dann werden die Werte der Variablen des aktuellen `make`-Prozesses nicht an das Sub-`make` weitergegeben. Manchmal ist dies aber wünschenswert, wenn nämlich per Variablen wichtige Konfigurationsparameter festgelegt werden. Traditionell behilft man sich, indem man dem Aufruf von `make` (im `Makefile`) die Variablen explizit per Zuweisung als Argumente zum Aufruf mitgibt. `make` bietet aber auch die Möglichkeit, Variablen zu exportieren. Dies entspricht dem Export von Variablen in der Bourne-Shell (und benutzt auch die gleiche Syntax):

export *Var*

export *Var=Value*

export *Var:=Value*

export *Var+=Value*

Dies hat zur Folge, daß die Variable *Var* auch in Sub-`make`s mit ihren aktuellen Werten definiert sind. Wie in der Bourne-Shell kann man das `export` auch vor eine Zuweisung setzen, um eine Zeile zu sparen.
Will man eine Variable wieder lokal machen, also ihren Export verhindern, benutzt man die folgende Direktive:

unexport *Var*

5.5.3 Sonstige Direktiven

Um den Text eines anderen Files in ein `Makefile` einzubinden, benutzt man die folgende Direktive:

include *File*

Diese Direktive wird ebenfalls nur beim Einlesen des `Makefiles` ausgewertet. Der Inhalt des angegebenen Files wird dann eingelesen und so behandelt, als hätte er an der Stelle dieser Direktive gestanden.
Diese Direktive ist besonders bei großen Projekten extrem nützlich, wenn Standardeinstellungen in einem File gehalten werden können, das dann von verschiedenen anderen `Makefiles` eingebunden werden kann.

In Abschnitt 5.4 wurde schon die spezielle Variable `VPATH` vorgestellt, in welcher Pfade angegeben werden können, in denen nach Quellen gesucht wird. Allerdings hat diese Variable den Nachteil, daß sie für alle Quellen benutzt wird, egal ob es sich bei den Quellen um **C**-Files oder Header-Files handelt.
`make` stellt die `vpath` Direktive bereit, mit der für spezifische Quellen Suchpfade gesetzt werden können:

vpath *Pattern Path* Setzt den Suchpfad für alle Quellen, die auf das Pattern *Pattern* passen. In dem Pattern steht `%` für eine beliebige Folge von Zeichen. Z.B.: `vpath %.h ../include:/usr/include` setzt den Suchpfad für Header-Dateien

vpath *Pattern* Löscht den Suchpfad für alle Quellen, die auf *Pattern* passen.

vpath Löscht alle Suchpfade

5.6 Aufrufparameter

`make` kennt eine ganze Reihe von Kommandozeilen-Parametern:

`make` *Optionen Zuweisungen Ziele*

Neben der Definition von Variablen auf der Kommandozeile (*Zuweisungen*), die schon in Abschnitt 5.4 besprochen wurden und der Angabe von Zielen (*Ziele*), die neugebaut werden sollen, gibt es auch noch eine Reihe von Optionen:

-f *File* Liest das angegebene *File* als Makefile (statt der Defaults)

-n Führt keine Kommandos aus, sondern gibt nur auf dem Bildschirm aus, welche Kommandos ausgeführt werden würden um die Ziele neu zu bauen.

-k Bricht nach einem Fehler in einem Kommando nicht ab, sondern versucht noch andere Sub-Ziele neu zu bauen die nicht von dem fehlgeschlagenen Ziel abhängen.

-C *Verzeichnis* Wechselt vor dem Lesen des `Makefiles` in das angegebene *Verzeichnis*. Nützlich für `make`-Aufrufe in Unterverzeichnissen.

-i Ignoriert alle Fehlermeldungen von Kommandos und nimmt an, daß alle Kommandos immer ohne Fehler ausgeführt wurden.

-q Versucht nicht, Ziele neu zu bauen, sondern gibt den Exit-Status 0 zurück, wenn alle Ziele jünger als ihre Quellen sind. Müssten Kommandos ausgeführt werden, um Ziele zu aktualisieren, wird ein Wert $\neq 0$ zurückgegeben.

-s Verhindet, daß die Kommandos vor der Ausführung noch auf dem Bildschirm ausgegeben werden.

5.7 Implizite Regeln

Bisher wurde immer angenommen, daß man alle Abhängigkeiten einer Regel namentlich aufführt. Es kann aber auch nützlich sein, eine Regel für alle Ziele/Quellen zu haben, die auf ein bestimmtes Pattern passen, etwa eine Regel um aus C-Files (*.c) Objectcode-Files (*.o) zu machen. `make` bietet zu diesem Zweck implizite Regeln. Hierbei wird das Ziel durch ein Pattern beschrieben, wobei das '`%`' für eine beliebige Folge von Zeichen steht. In den Quellen kann dann auch ein '`%`' auftauchen, das dann durch den Match des Patterns im Ziel ersetzt wird. Zum Beispiel würde folgende Regel zur Compilierung von C-Files zu Objectcode-Files verwendet werden können:

```
%.o: %.c
	$(CC) -o $@ -c $<
```

Wenn nun z.B. ein Ziel namens `foo.o` neu gebaut werden muß, dann wird `%` auf `foo` gematcht und nach einer Quelle `foo.c` gesucht.
Da die Namen der Ziele und Quellen in impliziten Regeln nicht mehr direkt angesprochen werden können, muß man in ihnen automatische Variablen verwenden, die an bestimmte Teile von Ziel und/oder Quellen gebunden werden, siehe Abschnitt 5.4. Es gibt eine automatische Variable, die nur in impliziten Regeln definiert ist: `$*` Sie wird gerade an den Stamm, also den Teil, welcher das `%` matcht, gebunden. In obigem Beispiel würde `$*` an `foo` gebunden werden.
Natürlich kann man bei impliziten Regeln auch Quellen angeben, die kein Pattern enthalten. Dies ist ein einfacher Weg, alle Ziele eines bestimmten Typs von einem File (etwa einem generellen Header-File) abhängig zu machen.
Zu erwähnen ist noch, daß `make` implizite Regeln immer nur anwendet, wenn es keine expliziten Regeln[9] für das Ziel gibt. So kann man in Spezialfällen (für bestimmte Files) die implizite Regel durch Angabe einer konkreten überstimmen. Es ist aber möglich, explizite Regeln zu geben, die keine Kommandos enthalten, also nur Abhängigkeiten hinzufügen. Diese ergänzen die implizite Regel, überschreiben sie aber nicht.
`make` hat bereits eine ganze Reihe von impliziten Regeln vordefiniert, etwa zur Compilierung von C-Files zu Objectcode. Eine komplette Liste findet man in [SM98].

[9] Mit Kommandos

5.8 makedepend/autoconf

5.8.1 Abhängigkeiten generieren

Bei großen Projekten steigt die Anzahl der Abhängigkeiten zwischen Files meist rapide an. Vergißt man, eine Abhängigkeit in das `Makefile` einzutragen, kann dies zur Folge haben, daß ein Ziel nach einer Ändrung an einer seiner Quellen nicht neu gebaut wird, was zu extrem schwer zu findenden Fehlern führt[10]. Deshalb wäre es wünschenswert, wenn man Abhängigkeiten automatisch generieren lassen könnte.
Für **C**-Files gibt es ein Programm, `makedepend`, welches für eine Sammlung von Source-Files die Header-Files in Abhängigkeiten der Objectcode-Files ausgibt.
Da Header-Files wiederum anderer Header-Files einbinden können, ist diese Aufgabe nicht vollkommen trivial. Nehmen wir die Abhängigkeiten aus Abbildung 5.1; nehmen wir weiter an, daß das File `bar.h` noch ein File `../boo.h` inkludiert.
Ein Aufruf von `makedepend foo.c baz.c` würde dann folgende Abhängigkeiten generieren:

```
foo.o: foo.c bar.h ../boo.h
baz.o: baz.c bar.h ../boo.h
```

Per Default hängt `makedepend` diese Ausgaben an ein bestehendes `Makefile` im akt. Verzeichnis an. Genauer gesagt wird ein vorhandenes `Makefile` gelesen und nach einer Zeile der Form

```
# DO NOT DELETE THIS LINE -- make  depend  depends  on it.
```

gesucht. Alles nach dieser Zeile wird gelöscht und durch die neuen Abhängigkeiten ersetzt. So muß man nicht vor jedem Aufruf von `makedepend` das `Makefile` neu generieren.
Man kann aber mittels der Option `-f`*`File`* auch ein alternatives *File* angeben, in welches die Abhängigkeiten geschrieben werden. Dieses kÖnnte man dann etwa mittels der `include` Direktive in das eigentliche `Makefile` einbinden.

5.8.2 Configurationsfiles erstellen

Wenn man portable Software schreiben will (muß), dann hat man zwei Probleme

- Feststellung der Fähigkeiten eines Zielsystems.

[10]Was der Author bei der Entwicklung eines Device-Drivers schmerzhaft erfahren musste.

- Bereitstellung fehlender Funktionalität für ein bestimmtes System.

Der erste Punkt ist oft schwieriger, als man denkt. So gibt es derzeit so viele Varianten verschiedenster UNIX und andersartiger Systeme, daß es kaum möglich ist, statisch alle Fähigkeiten eines Systems zu kennen. Auch soll die Software ja auch auf Systemen laufen, die zwar bestimmte Fähigkeiten[11] besitzen, die aber zum Zeitpunkt der Programmierung des Systems dem Entwickler nicht bekannt waren.
Während man bei dem zweiten Punkt auf sich selbst gestellt bleibt[12], gibt es für den ersten Punkt ein Packet, das einen unterstützt.
Das `autoconf` Packet unterstützt nun diese Art der Portabilität. Als Voraussetzung dafür muß die Software schon portabel angelegt sein. Etwa muß für bestimmte Features des Betriebssystems eine Abfrage im Code eingebaut sein, die im Falle des Fehlens dieses Features einen Workaround anwendet.
Dies geschieht (wenn man in **C** programmiert) meistens, indem man bestimmte Präprozessorsymbole einführt, die im Falle des Vorhandenseins eines Features definiert sind, sonst undefiniert[13]. Für bestimmte Werte im `Makefile` selbst, etwa das Verzeichnis, in welches man die Software installieren will, oder bestimmte zu benutzende Programme, kann man dort `make`-Variablen einführen, etwa `$(CC)` für den zu verwendenden C-Compiler.
Um diese Informationen zu erhalten, läßt man auf der Zielmaschine zur Installation ein Configurations-Script laufen:

- Dieses führt Tests auf die in Frage kommenden Eigenschaften aus
- erzeugt `Makefiles` aus Schablonen `Makefile.in`
- erzeugt ein Header-File `config.h` aus `config.h.in`

In dem erzeugten `Makefile` werden bestimmte Platzhalter durch gefundene Werte, etwa den Pfad zu dem zu benutzenden C-Compiler, ersetzt. Das Header-File enthält dann Definitionen für C-Präprozessor Symbole, abhängig von den angetroffenen Features.
Ein Beispiel für den Lauf eines solchen `configure` Scriptes:

```
> ./configure --prefix=$HOME --enable-shared
...
creating Makfile
creating src/Makefile
creating config.h
```

[11] Etwa einen bestimmten Systemaufruf oder Libraryroutine.

[12] D.h. wenn das Zielsystem keine `vsprintf` Routine kennt, muß man halt selbst eine (abgespeckte) Version davon bereitstellen.

[13] Im Sinne der `#ifdef` Präprozessor-Direktiven.

Hier wurden zwei `Makefiles` erzeugt: Das eine im aktuellen Verzeichnis, das andere im Verzeichnis `src`.
Beispiel für ein `Makefile.in`

```
srcdir = @srcdir@
VPATH = @srcdir@

CC = @CC@

INSTALL = @INSTALL@
INSTALL_PROGRAM = @INSTALL_PROGRAM@
INSTALL_DATA = @INSTALL_DATA@

install: all
        ${INSTALL_PROGRAM} indent ${bindir}/gindent
```

In dieser Schablone werden alle Platzhalter der Form `@`*Name*`@` durch Werte ersetzt, die durch das Configurations-Script bestimmt werden. So wird etwa `@CC@` durch den Namen des C-Compilers ersetzt, der benutzt werden soll; `@srcdir@` durch den Namen des Verzeichnisses, in welchem der Source-Code des Projektes steht. `@INSTALL@` wiederum wird durch den Namen eines BSD-`install` kompatiblen Installationsprogrammes ersetzt, etc.
Beispiel für ein `config.h.in`:

```
/* config.h.in. Generated automatically from
   configure.in by autoheader.*/

/* Define if on AIX 3.
   System headers sometimes define this.
   We just want to avoid a redefinition error message.  */
#ifndef _ALL_SOURCE
#undef _ALL_SOURCE
#endif

/* Define if using alloca.c.  */
#undef C_ALLOCA

/* Define to empty if the keyword does not work.  */
#undef const
```

In dieser Schablone für das Header-File kann man neben normalen Direktiven (hier dem `_ALL_SOURCE` Teil) Zeilen der Form

`#undef` *`Symbol`*

einfügen. Diese werden dann von dem Configurations-Script durch Zeilen der Form `#define` *Symbol* ersetzt, wenn das entsprechende Feature vorhanden ist. Man braucht dieses File meistens nicht selbst zu erzeugen, da das Programm `autoheader` es automatisch aus den Test in dem `configure.in` File ableiten kann, s.u.
Das Configurations-Script selbst ist ein normales Bourne-Shell Script, welches man allerdings nicht selbst schreiben muß, sondern das man aus einer Beschreibung der durchzuführenden Test mittels des `autoconf` Programms erzeugt. Diese Beschreibung steht i.A. in einem File namens `configure.in`. Dieses File enthält neben normalen Shell-Befehlen sogenannte *Tests*. `autoconf` ersetzt diese Tests durch Shell-Befehle, welche die von dem Test gewünschte Prüfung durchführen.
Beispiel für ein `configure.in`:

```
dnl Process this file with autoconf to produce a
    configure script.
AC_INIT(indent.c)
AC_PROG_CC
AC_PROG_CPP
AC_PROG_INSTALL
AC_AIX
AC_ISC_POSIX
AC_REPLACE_FUNCS(memcpy)
AC_UNISTD_H
AC_DIR_HEADER
if test -z "$dirheader"; then
  AC_DEFINE(NODIR)
fi
AC_XENIX_DIR
AC_OUTPUT(Makefile, src/Makefile)
```

Hier dient die Zeile `AC_INIT(indent.c)` zur Initialisierung, Kommentare werden durch `dnl` eingeleitet. Beispielhaft prÜft der Test `AC_PROG_INSTALL` auf ein BSD-kompatibles `install` Programm (s.o.) und ersetzt den Platzhalter `@INSTALL@` durch dieses Programm.
Der Test `AC_UNISTD_H` prüft, ob das Header-File `unistd.h` vorhanden ist und setzt dann das Symbol `HAVE_UNISTD_H` in `config.h`.
Die letzte Zeile startet die Ersetzung von den Schablonen in die Files `Makefile` und `src/Makefile`.
`autoconf` bietet unzählige Tests:

- Test auf Vorhandensein & Ort verschiedenster Utilities
- Test auf Libraries

- Test auf bestimmte Funktionen in Libs
- Systemaufruftest
- Größe von Datentypen
- Header-Files
- typedefs, Definitionen in Headern
- ...

Für eine ausführliche Beschreibung der Tests sei auf die Dokumentation [ME98] des `autoconf` Packetes verwiesen.

5.9 Beispiele

Bei der Generierung von Dokumenten mit LaTeX (siehe Kapitel 6) bindet man oft Abbildungen ein, die mittels `xfig` erzeugt wurden. Diese Abbildungen werden dann mit dem `fig2dev` Programm vom xfig-Format zu Encapsulated-Postscript konvertiert. Dabei benutzt man einen Aufruf der Form

`fig2dev -L ps -p dummy -m` *`Mag XFIG-File`* `>`*`EPS-File`*

Hierbei steht *Mag* für den Skalierungsfaktor der Abbildung. Dieser kann für verschiedene Abbildungen variieren, da alle Abbildungen in `xfig` meistens in der gleichen Größe entworfen werden, im Text aber verschieden groß erscheinen sollen.

Folgende Regeln und Variablendefinitionen erlauben es nun, innerhalb eines `Makefiles` eine Standardskalierung anzugeben, die aber für bestimmte Abbildungen überschrieben werden kann.

```
FIGSRCS=abb1.fig abb2.fig
FIGEPS=$(patsubst %.fig,%.eps,$(FIGSRCS))

# Default Skalierung
MAG_DEFAULT=0.5

# Spezielle Sklalierung fuer  abb1
MAG_abb1=0.8

# fig -> eps
%.eps: %.fig
	fig2dev -L ps -p dummy \
	  -m $(word 1,$(MAG_$*) $(MAG_DEFAULT)) $< >$@
```

Alle XFIG-Files werden in der Variablen `$(FIGSRCS)` angegeben, daraus wird mittels einfacher Ersetzung die Liste aller EPS-Files gemacht (die dann in den Quellen des Dokumentzieles als Abhängigkeiten angegeben werden können). Fur das File `abb1.eps` wird eine abweichende Vergröserung angegeben. In der impliziten Regel für die Konvertierung von XFIG-Files zu EPS-Files wird nun ein Trick angewandt: Mittels der `word` Funktion wird das erste Wort der Liste `$(MAG_$*) $(MAG_DEFAULT)` genommen. Der erste Teil enthält dabei den Namen des Stammes (z.B. `abb1`) der impliziten Regel. D.h. das erste Wort expandiert zum Inhalt der Variablen `$(MAG_abb1)`, wenn man `abb1.eps` neu bauen soll. Diese Variable ist aber nur dann definiert, wenn man eine spezielle Skalierung für die Abbildung definiert hat, expandiert im Standardfall also zum leeren Text. Damit ist das erste Wort der Liste die Standardskalierung, wenn man keine spezielle Skalierung für die fragliche Abbildung definiert hat. Ist eine solche definiert, ist sie das erste Wort der Liste und wird von der `word` Funktion ausgewählt.

5.10 Falltüren

Eine beliebte Falltür im Zusammenhang mit Variablen ist, daß Variablen lazy ausgewertet werden, wenn sie mittels des = Operators zugewiesen wurden. Das Anhängen von Werten an Variablen funktioniert also nicht durch `CFLAGS=$(CFLAGS) -Wall`, sondern man muß den speziellen += Operator verwenden: `CFLAGS+=-Wall`.
Ebenso vergißt man leicht, daß Variablen, die auf der Kommandozeile gesetzt wurden, nicht mehr überschrieben werden können (außer, man benutzt die `override` Direktive).
Auch werden Variablen nicht an Sub-`make` Prozesse weitergereicht, es sei denn man exportiert sie explizit.

5.11 Literatur

Die ultimative Referenz für GNU `make` ist natürlich die Dokumentation [SM98], welche mit dem Programm als Info-Seiten installiert sein sollte (`info make`). Die meisten Bücher über `make`-ähnliche Programme gehen nicht auf GNU `make` ein, können aber lesenswert sein, um die allgemeinen Prinzipien zu verstehen. Ich habe leider noch keins gefunden, das empfehlenswert wäre.
Auskunft über das `makedepend` Programm gibt dessen Manual-Seite. Auf jedem System, auf dem X-Windows installiert ist, sollte es auch eine Version des `makedepend` Programmes geben, da es mit der X-Windows Distribution instal-

liert wird. Die Manual-Seite sollte sich also zusammen mit denen des X-Windows System finden (meistens `/usr/X11R6/share/man`).
Eine Referenz über die Prinzipien und verfügbaren Tests des `autoconf` Packetes bietet dessen Dokumentation [ME98], die auch als Info-Seiten installiert sein sollten.