4.10 Ein größeres Beispiel: Makefile zu genFamMem

Nach der Weihnachtspause ein **größeres Beispiel zur Auffrischung**: Das Makefile des Programms **genFamMem** von Jan Bredereke.

genFamMem ist ein Generator für Familien von formalen Anforderungen in CSP-OZ. Homepage: `http://www.tzi.de/~brederek/genFamMem/`

Struktur der Quellen von genFamMem

genFamMem/					
INSTALL	*bin/*	*dist/*	src/	sty/	test/
	genFamMem	*genFamMem.tgz*	Makefile	*.sty	*
	*.*o*		*.c		*.good
	*.*d*		*.l		*.*log*
	*.*tab.c*		*.y		
	*.*tab.h*				
	*.*tab.o*				
	*.*tab.d*				

Legende:

normal	handgeschriebene Datei
kursiv	generierte Datei / Verzeichnis

Kurs Unix-Tools, WS 01/02, Jan Bredereke 26

`Makefile-genFamMem`:

```
# ------------------------------------------------------------------------
#
# project: genFamMem - maintaining families of requirements
#
# author: Jan Bredereke
# copyright: Jan Bredereke, 2000
# $Id: Makefile,v 1.51 2001/02/22 12:39:36 brederek Exp $
#
# file: Makefile - main Makefile of the generator program
#
# You will need GNU's gmake for this.
#
# ------------------------------------------------------------------------

# definitions:

# directories relative to the current dir:
BINDIR=../bin
TESTDIR=../test
STYDIR=../sty
DISTDIR=../dist
DISTNAME=genFamMem
```

```
DISTROOT=../..
# directory and file relative to $(DISTROOT)
DISTDIR_ROOT=$(DISTNAME)/dist
TARBALL=$(DISTDIR)/genFamMem.tgz
TARBALL_ROOT=$(DISTDIR_ROOT)/genFamMem.tgz
DISTCONTENTS=$(DISTNAME)/INSTALL $(DISTNAME)/src $(STYLEFILES_ROOT)
# the rest is relative to the current dir, again:
EXECUTABLE=$(BINDIR)/genFamMem
CSOURCES=main.c relation.c identifier.c genidentifier.c linerange.c calc.c \
        filterspec.c debug.c printuseshier.c
YSOURCES=firstpass.y
LSOURCES= firstpasstex.l secondpasstex.l firstpasslex.l
TSOURCES= z1.tex cspoz1.tex case-study.tex case-study-uh-fm.tex \
        case-study-uh-fm-of.tex case-study-uh-f-usf.tex cspoz2.tex
TDIRSOURCES=$(TSOURCES:%=$(TESTDIR)/%)
TDIRLOGS=$(TDIRSOURCES:%=%.log)
COBJECTS=$(CSOURCES:%.c=$(BINDIR)/%.o)
YOBJECTS=$(YSOURCES:%.y=$(BINDIR)/%.tab.o)
YGENCS=$(YSOURCES:%.y=$(BINDIR)/%.tab.c)
YGENHS=$(YSOURCES:%.y=$(BINDIR)/%.tab.h)
YGENOUTPUTS=$(YSOURCES:%.y=$(BINDIR)/%.output)
LOBJECTS=$(LSOURCES:%.l=$(BINDIR)/%.o)
LGENCS=$(LSOURCES:%.l=$(BINDIR)/%.c)
OBJECTS=$(COBJECTS) $(LOBJECTS) $(YOBJECTS)
GENCS=$(LGENCS) $(YGENCS) $(YGENHS) $(YGENOUTPUTS)
CDEPENDS=$(CSOURCES:%.c=$(BINDIR)/%.d)
YDEPENDS=$(YSOURCES:%.y=$(BINDIR)/%.tab.d)
LDEPENDS1=$(LSOURCES:%.l=$(BINDIR)/%.d)
LDEPENDS=$(filter-out $(BINDIR)/firstpasslex.d,$(LDEPENDS1))
DEPENDS=$(CDEPENDS) $(YDEPENDS) $(LDEPENDS)
TESTLOGS=$(TDIRSOURCES:%=%.log)
TESTGOODS=$(TDIRSOURCES:%=%.good)
FAMSTYLEFILES=$(STYDIR)/oz-fam.sty $(STYDIR)/csp-oz-fam.sty
EXTERNALSTYLEFILES=$(STYDIR)/csp-oz.sty $(STYDIR)/oz.sty
EXTERNALSTYLEFILESSOURCES=/home/brederek/doc/sty/csp-oz.sty \
        /home/brederek/doc/sty/oz.sty
STYLEFILES=$(FAMSTYLEFILES) $(EXTERNALSTYLEFILES)
STYLEFILES_ROOT=$(STYLEFILES:$(STYDIR)/%=$(DISTNAME)/sty/%)

CPPFLAGS=-I$(BINDIR)
CFLAGS=-g -DYYDEBUG
YFLAGS=--verbose
YACC=bison
LEX=flex
# Gnu tar:
TAR=tar

# ----------------------------------------------------------------------------

# targets that might be used manually:
```

```
all: $(EXECUTABLE)

maintainer-all: all $(EXTERNALSTYLEFILES)

depend:
	rm -f $(DEPENDS)
	$(MAKE) $(DEPENDS)

clean:
	rm -f $(GENCS) $(OBJECTS) $(EXECUTABLE) $(TESTLOGS)

dist-clean:
	$(MAKE) clean
	rm -f $(DEPENDS)
	-rmdir $(BINDIR)

maintainer-clean:
	$(MAKE) dist-clean
	rm -f $(TARBALL)
	-rmdir $(DISTDIR)
	rm -f $(EXTERNALSTYLEFILES)

tests: $(TESTLOGS)

# (Currently, the tarball distribution contains the sources only, but not
# the (voluminous) test data. Ask me if you really need to do regression
# testing. J.B.)
dist: $(DISTDIR) $(EXTERNALSTYLEFILES)
	cd $(DISTROOT) ; \
	    $(TAR) --create --verbose --gzip --owner=0 --group=0 \
	        --file=$(TARBALL_ROOT) \
	        --exclude=CVS --exclude='*.swp' \
	        $(DISTCONTENTS)

# ---------------------------------------------------------------------------

# some general issues:

# read the source dependences (which are auto-generated)
include $(DEPENDS)

# what are not real targets:
.PHONY: all maintainer-all depend clean dist-clean maintainer-clean tests dist

# in case of an error, generally delete a half-generated target:
.DELETE_ON_ERROR:

# ---------------------------------------------------------------------------

# the actual rules for file generation:
```

```
# the directory for binaries:
$(BINDIR):
	mkdir $(BINDIR)

# the directory for the distribution tar ball:
$(DISTDIR):
	mkdir $(DISTDIR)

# the executable:
$(EXECUTABLE): $(OBJECTS)
	$(CC) $(LDFLAGS) -o $(EXECUTABLE) $(OBJECTS) $(LDLIBS)

# the C objects:
$(COBJECTS): $(BINDIR)/%.o: %.c
	$(CC) -c $(CPPFLAGS) $(CFLAGS) -o $@ $<

# the bison and flex objects:
$(YOBJECTS) $(LOBJECTS): %.o: %.c
	$(CC) -c $(CPPFLAGS) $(CFLAGS) -o $@ $<

# the depend files:
$(CDEPENDS): $(BINDIR)/%.d: $(BINDIR) %.c
$(YDEPENDS): $(BINDIR)/%.tab.d: $(BINDIR) %.y
$(LDEPENDS): $(BINDIR)/%.d: $(BINDIR) %.l

# how to make an automatically generated dependences file for a source file:
$(BINDIR)/%.d: %.c
	$(CC) $(CFLAGS) -MM $< | sed -e '1,1s,^,'"$(BINDIR)/," > $@
$(BINDIR)/%.d: $(BINDIR)/%.c
	$(CC) $(CFLAGS) -MM $< | sed -e '1,1s,^,'"$(BINDIR)/," > $@
$(BINDIR)/%.tab.d: %.y
	$(CC) $(CFLAGS) -MM $< | \
	    sed -e '1,1s,^,'"$(BINDIR)/,"';1,1s/\.o: /.tab.c: /' > $@
$(BINDIR)/%.tab.d: %.l
	$(CC) $(CFLAGS) -MM $< | \
	    sed -e '1,1s,^,'"$(BINDIR)/,"';1,1s/\.o: /.c: /' > $@

# drop implicit rule for *.l:
%.c: %.l

$(BINDIR)/firstpass.tab.c: firstpass.y
	$(YACC) $(YFLAGS) --defines --name-prefix=fplyy --output-file=$@ $<

$(BINDIR)/firstpasslex.c: firstpasslex.l global.h
	$(LEX) $(LFLAGS) -Pfplyy -o$@ $<

$(BINDIR)/firstpasslex.o: $(BINDIR)/firstpass.tab.c

$(BINDIR)/firstpasstex.c: firstpasstex.l
	$(LEX) $(LFLAGS) -Pfpmtyy -o$@ $<
```

```
$(BINDIR)/firstpasstex.o: $(BINDIR)/firstpasstex.c

$(BINDIR)/secondpasstex.c: secondpasstex.l
	$(LEX) $(LFLAGS) -Psptyy -o$@ $<

$(BINDIR)/secondpasstex.o: $(BINDIR)/secondpasstex.c

$(EXTERNALSTYLEFILES): $(EXTERNALSTYLEFILESSOURCES)
	cp $^ $(STYDIR)

# ------------------------------------------------------------------------

# tests

# all tests depend on the source file, on the regression comparison log, and
# on the executable:
$(TDIRLOGS): %.log: % %.good $(EXECUTABLE)

# how to run the individual tests:

$(TESTDIR)/z1.tex.log:
	$(EXECUTABLE) -M x $< > $@
	diff -q $<.good $@

$(TESTDIR)/cspoz1.tex.log:
	$(EXECUTABLE) --debug Member1 $< > $@
	diff -q $<.good $@

$(TESTDIR)/case-study.tex.log:
	$(EXECUTABLE) --debug SpecificationC $< > $@
	diff -q $<.good $@

$(TESTDIR)/case-study-uh-fm.tex.log:
	$(EXECUTABLE) --debug --uhier-familymember SpecificationC $< > $@
	diff -q $<.good $@

$(TESTDIR)/case-study-uh-fm-of.tex.log:
	$(EXECUTABLE) --uhier-familymember SpecificationC \
	    --only-features $< > $@
	diff -q $<.good $@

$(TESTDIR)/case-study-uh-f-usf.tex.log:
	$(EXECUTABLE) --debug --uhier-feature UserSpace $< > $@
	diff -q $<.good $@

$(TESTDIR)/cspoz2.tex.log:
	$(EXECUTABLE) --debug Member1 $< > $@
	diff -q $<.good $@
```

Es geht los bei **all:**.

Außerdem gibt es **clean:**, **tests:** und **dist:**.

maintainer-all:, **dist-clean:** und **maintainer-clean:** sind Varianten.

depend: wird später erläutert.

Alle sind als **.PHONY:** deklariert.

Regel für **$(EXECUTABLE)**.

$(OBJECTS) zusammengesetzt aus drei Teilen.

$(COBJECTS) erzeugt aus $(CSOURCES) mit Hilfe von Textersetzung. (Wird später im Detail erläutert.)

$(CSOURCES) ist Liste der C-Quellen.

Analog für **$(YSOURCES)** und **$(LSOURCES)**.

$(TSOURCES) sind die Test-Eingaben. **tests:** hängt von **$(TESTLOGS)** ab, den aktuellen Testausgaben. **$(TESTGOODS)** sind die Vergleichsdateien dazu. Am Ende des Makefiles ist die genaue **Durchführung der Tests** spezifiziert, mit allen Parametern.

Die **$(COBJECTS)** werden mit einer Muster-Regel erzeugt. Dabei ist zu beachten, daß die Objekt-Datei in einem **anderen Verzeichnis** abgelegt wird.

Die Übersetzung der **Lex- und Yacc-Quellen** erfolgt in **zwei Schritten**: Erst nach C, dann in eine Objekt-Datei.

Bei **Yacc** ist es **noch** etwas **komplexer**: Aus der **.y-Datei** werden **zwei Dateien** erzeugt, eine **.tab.c-Datei** und eine **.tab.h-Datei**, die für Lex benötigt wird.

Daher benötigen wir eine **explizite Abhängigkeit** von **firstpasslex.o**, der Lex-Objekt-Datei, von firstpass.tab.h (bzw. von firstpass.tab.c, die gleichzeitig erzeugt wird).

Dieses Programm hat **einen Yacc-Parser** und **drei Lex-Scanner**. Mit der **Lex-Option -P** werden sie durch ein Präfix auseinandergehalten. Daher brauchen wir jeweils separate, **explizte Regeln**.

Alle **generierten Dateien**, auch *.c und *.h, werden im **bin/-Verzeichnis** abgelegt.

Die **Abhängigkeiten durch include-Anweisungen** usw. in den Quellen werden **automatisch extrahiert** und im Makefile berücksichtigt. Deswegen zu Anfang **include $(DEPENDS)**

$(DEPENDS) wird wieder aus **drei Teilen** zusammengesetzt. Diese werden wiederum **aus den Quelldateien** per **Textersetzung** generiert.

Zu jeder Quelldatei gibt es eine **.d-Datei** mit den Abhängigkeiten:
`calc.d`:

```
../bin/calc.o: calc.c global.h relation.h linerange.h identifier.h \
 genidentifier.h calc.h
```

Diese Dateien werden **mit expliziten Regeln** aus den .c- .y- und .l-Dateien **erzeugt**.

Die **Option -MM** von gcc gibt eine solche Datei aus. Mit **sed** bearbeiten wir sie nach, damit die nach bin/ verschobene Objektdatei richtig angesprochen wird.

Details zu generierten Makefiles folgen **später**.

4.11 Implizite Regeln

Implizite Regeln sind **entweder Muster-Regeln oder altmodische Suffix-Regeln**. Implizite Regeln werden **angewandt, wenn** zwar eine **Abhängigkeit** spezifiziert ist, **aber keine Anweisungen** dafür gegeben sind.

Wir hatten implizite Regeln **bereits** kurz **zu Anfang kennengelernt**:
`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)
```

Hier wurde der **C-Compiler implizit aufgerufen**.

Es sind eine ganze **Reihe von Regeln bereits eingebaut**, **weitere** Regeln kann man **auf** die schon **bekannte Weise definieren**.

Mit einer Reihe von **Variablen in** den **eingebauten Regeln** kann man deren **Verhalten anpassen**.

Einige der Regeln:

Einige Implizite Regeln

```
# C kompilieren:
.c.o:
	$(CC) $(CPPFLAGS) $(CFLAGS) $(TARGET_ARCH) -c -o $@ $<
# C++ kompilieren:
.cc.o:
	$(CXX) $(CPPFLAGS) $(CXXFLAGS) $(TARGET_ARCH) -c -o $@ $<
.C.o:
	$(CXX) $(CPPFLAGS) $(CXXFLAGS) $(TARGET_ARCH) -c -o $@ $<
# Pascal kompilieren:
.p.o:
	$(PC) $(PFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c -o $@ $<
# einzelne Objekt-Datei linken:
.o:
	$(CC) $(LDFLAGS) $(TARGET_ARCH) $^ $(LOADLIBES) $(LDLIBS) -o $@
# Yacc nach C kompilieren:
.y.c:
	$(YACC) $(YFLAGS) $< && mv -f y.tab.c $@
# Lex nach C kompilieren:
.l.c:
	$(LEX) $(LFLAGS) -t $< > $@
```

Kurs Unix-Tools, WS 01/02, Jan Bredereke 27

Es gibt noch **viel mehr Regeln**. Die **komplette Liste** kann man sich mit `make -p -f /dev/null` ausgeben lassen.

In Wirklichkeit sind diese Regeln **komplizierter** aufgebaut, z.B. gibt es eine Variable `$COMPILE.c`, die die gesamte Zeile für die C-Kompilation enthält. Ebenso gibt es noch eine Variable, mit der man die Ausgabe-Option `-o` global umdefinieren kann.

Interessant sind aber die **gezeigten Variablen**.

Die **Compiler-Variablen** sind geeignet **vorbesetzt**, z.B. mit `cc`, `g++`, `pc`, `yacc` und `lex`. Man **kann** sie ggf. **umdefinieren**.

Die **Variablen für zusätzliche Argumente** sind normalerweise **leer**, aber man kann sie **ebenfalls umdefinieren**.

`CPPFLAGS` sind die zusätzlichen **Optionen für** den **C-Präprozessor**, `CFLAGS` die für den eigentlichen **C-Compiler**.

4.12 Verkettung impliziter Regeln

Manchmal kann eine **Datei durch** eine **Verkettung impliziter Regeln erzeugt** werden. Beispiel:

►► `cd chain`

Makefile:

```
# Vorhandene Datei: foo.l
# Eingebaute implizite Regeln: .l.c, .c.o und .o
foo:
```

Dieses Makefile ist reichlich **minimalistisch**!

►► `make`

Was passiert? `make` weiß, daß die Datei `foo` **erzeugt werden muß**, und es weiß **durch Nachschauen**, daß die Datei `foo.l` **existiert**. In der Sammlung impliziter Regeln **findet es** einen **Weg**, das Ziel aus der vorhandenen Datei **zu erzeugen**.

Die Dateien `foo.c` und `foo.o` sind **Zwischendateien**.

Zwischendateien sind Dateien, die im Makefile **nicht erwähnt** werden.

Sie **sind anders** in zwei Punkten:

- Sie werden **nur erzeugt**, wenn sie wirklich **benötigt** werden.
- Sie werden **am Ende** des `make`-Laufes **gelöscht.** (Siehe `rm`.)

Man kann auch eine **im Makefile erwähnte Datei zu** einer **Zwischendatei** machen. Dafür muß man sie als Voraussetzung zum **besonderen Ziel** `.INTERMEDIATE` angeben.

Man kann auch ein **Mittelding** haben: Nur erzeugt, **wenn nötig**, aber **nicht automatisch gelöscht**: Dafür muß man die Datei als Voraussetzung zum **besonderen Ziel** `.SECONDARY` angeben.

`Makefile2:`

```
# Vorhandene Datei: foo.l
# Eingebaute implizite Regeln: .l.c, .c.o und .o
.INTERMEDIATE: foo.o
.SECONDARY: foo.c
foo:
```

▶▶ `rm foo`

▶▶ `make -f Makefile2`

Keine implizite Regel darf in einer Kette **mehr als einmal angewendet** werden. `make` wird also nicht einmal erwägen, `foo` aus `foo.o.o` zu erzeugen. Dies **vermeidet auch Totschleifen** bei der Regelsuche.

4.13 Details von Variablen

4.13.1 Syntax

Erlaubte Zeichen in Variablennamen: **Fast alle**, aber man sollte für normale Variablen **nur Buchstaben, Zahlen und Underscores** verwenden. Streng **verboten** sind **nur „:“, „#“, „=“ und Whitespace**.

Groß-/Kleinschreibung ist relevant.

Konventionen: Oft ganz **groß**geschrieben. **Gnu empfieht: Kleinschreibung** für normale, lokale Variablen, **Großschreibung** nur für Parameter impliziter Regeln oder Variablen, die evtl. auf der Kommandozeile umdefiniert werden.

Verwendung von Variablen: **Dollar**zeichen und Name in **Klammern**, entweder **runde oder geschweifte** Klammern.

Wenn **ohne Klammern**, dann muß der Name **einbuchstabig** sein. Empfohlen **nur für** die **automatischen** Variablen.

Whitespace ist ggf. relevant:
`Makefile-whitespace`:

```
dir = /home/brederek    # Dies ist FALSCH!
file = foo.txt
pathfile = $(dir)/$(file)

all:
	@echo "Die Datei ist $(pathfile)."
```

▶▶ `make -f Makefile-whitespace`

Whitespace **vor und nach dem Gleichheitszeichen** wird **ignoriert**, aber Whitespace **vor dem Zeilenende** wird **erhalten**. Dies kann zu Überraschungen führen.

Es gibt **kein Limit für** die **Länge** eines Variablenwertes, außer dem Swapspace.

Wenn ein Wert **undefiniert** ist, wird der **leere String** eingesetzt. Das passiert z.B. oft bei Variablen aus den impliziten Regeln.

4.13.2 Variablen-Expansion

Wenn **in einer Variablen-Definition** eine andere Variable vorkommt, dann wird sie **zunächst *nicht* expandiert**.

Die Variablen-**Expansion** findet **erst** statt, wenn die **äußere Variable verwendet** wird. Dies passiert dann **rekursiv**:
`Makefile-var-rekursion`:

```
foo:
	cc -o foo $(CFLAGS) foo.c

#

CFLAGS = $(include_dirs) -O
include_dirs = -Ibar -Ibaz
```

Vorteil: Die **Reihenfolge** der Definition ist **egal**.

Daher ist **folgendes nicht erlaubt**:
`Makefile-var-rekursion2`:

```
foo:
	cc -o foo $(CFLAGS) foo.c

#

CFLAGS = $(include_dirs) -O
include_dirs = -Ibar -Ibaz

CFLAGS = $(CFLAGS) -Wall
```

►► `make -f Makefile-var-rekursion2`

`make` **erkennt** die **Rekursion** und meldet einen **Fehler**.

4.13.3 Anhängen an Variablen

Man kann diesen Effekt aber **anders erreichen**:
`Makefile-var-append`:

```
foo:
	cc -o foo $(CFLAGS) foo.c

#

CFLAGS = $(include_dirs) -O
include_dirs = -Ibar -Ibaz

CFLAGS += -Wall
CFLAGS += -pg
```

►► `make -f Makefile-var-append -n`

Dies wird **häufig benutzt**. Gerade für die Variablen aus **impliziten Regeln** ist es sinnvoll.

Die **Reihenfolge** des Anhängens ist die Reihenfolge in der Datei.

Ansonsten wird bei dieser Operation der Wert der Variablen **noch nicht expandiert**, dies passiert erst bei der Verwendung. Es wird an den un-expandierten Text angehängt.

4.13.4 Zwei Arten von Variablen

Als **Gnu-Erweiterung** gibt es **zwei Arten** von Variablen:

- normale, rekursiv expandierte Variablen
- einfach expandierte Variablen

In den meisten **Fällen sind** die **normalen**, rekursiv expandierten Variablen **das richtige**.

Aber **Nachteile**: Expansion **bei jeder Verwendung**. Schlecht z.B., wenn ein **Shell-Aufruf** darin.

Einfach expandierte Variablen werden zum **Definitionszeitpunkt** ein einziges Mal expandiert:
`Makefile-var-simple-expand`:

```
x := foo
y := $(x) bar
x := later

all:
	@echo "x = >$(x)<, y = >$(y)<"
```

▶▶ `make -f Makefile-var-simple-expand`

Kennzeichen ist eine Zuweisung mit „`:=`“ statt „`=`“.

Diese Variablen **funktionieren wie in Programmiersprachen**, mit allen **Problemen**, die das bringt. Manche Leute **mögen** es aber **lieber**.

Man kann jetzt auch **mit Variablen „rechnen“**:
`Makefile-var-simple-expand2`:

```
include_dirs := -Ibar -Ibaz
CFLAGS := $(include_dirs) -O
CFLAGS := $(CFLAGS) -Wall

foo:
	cc -o foo $(CFLAGS) foo.c
```

▶▶ `make -f Makefile-var-simple-expand2 -n`

Aber zum Anhängen ist das **nicht notwendig**, s.o.

Vorsicht mit der **Reihenfolge** der Zeilen! (Ich mußte **alles umstellen**.)

Anmerkung: Der **+=-Operator** geht natürlich **auch** für einfach expandierte Variablen.

4.13.5 override

Man kann **Variablen auf der Kommandozeile** setzen, z.B. CFLAGS. **Was ist**, wenn man im Makefile **trotzdem** immer noch eine Option **anhängen** will?
`Makefile-var-append`:

```
foo:
	cc -o foo $(CFLAGS) foo.c

#

CFLAGS = $(include_dirs) -O
include_dirs = -Ibar -Ibaz

CFLAGS += -Wall
CFLAGS += -pg
```

▶▶ `make -f Makefile-var-append 'CFLAGS=-Ibar -Ibaz' -n`

(Das ist **ohne das** `-O` .)

Das **geht schief**, die angehängten Sachen gehen verloren.

Dies geht mit override:
`Makefile-var-append2`:

```
foo:
	cc -o foo $(CFLAGS) foo.c

#

CFLAGS = $(include_dirs) -O
include_dirs = -Ibar -Ibaz

override CFLAGS += -Wall
override CFLAGS += -pg
```

▶▶ `make -f Makefile-var-append2 'CFLAGS=-Ibar -Ibaz' -n`

So **klappt** es.

Anmerkung: Man kann override **auch für einfache Zuweisungen** benutzen, aber **sinnvoll** ist es meist **nur für** das **Anhängen**.

4.13.6 Bedingte Zuweisung

Wenn man das Makefile aus verschiedenen Teilen zusammensetzt, ist manchmal die bedingte Zuweisung sinnvoll:
`Makefile-var-cond`:

```
myflags ?= -frob

foo:
	bar $(myflags) $@

########################################################################

myflags = -frob2
```

▶▶ `make -f Makefile-var-cond -n`

▶▶ `make -f Makefile-var-cond -n` ohne die letzte Zeile

Man braucht das aber **eher selten**. Denn **Kommandozeilen-Zuweisungen gehen** ohnehin **vor**!

Sinnvoll ist es auch, wenn man erwartet, daß Make-Variablen durch **Variablen aus dem Environment** gesetzt werden. Aber das **ist selbst** ziemlich **gefährlich**.

Ansonsten funktioniert es **gleich für beide Arten** von Variablen.

4.13.7 Ersetzungen

Ersetzungen sind schon kurz im obigen **genFamMem-Beispiel** angesprochen worden.

Eine Ersetzung ist eine **Variablenreferenz**, bei der auf den Namen ein **Doppelpunkt** folgt. Danach kommt die **zu ersetzende** Endung, ein **Gleichheitszeichen,** und die **neue** Endung:

`Makefile-subst`:

```
LSOURCES= firstpasstex.l secondpasstex.l firstpasslex.l
LOBJECTS=$(LSOURCES:.l=.o)

foo:
	@echo $(LOBJECTS)
```

▶▶ `make -f Makefile-subst`

Es wird **jedes Wort** in der Variablen **getrennt bearbeitet**. Worte sind durch **Whitespace** abgetrennt.

Es werden **nur** die **Wort-*Enden*** bearbeitet.

Bei **Gnu-**`make` gibt es eine **erweiterte** Form: Mit **Prozentzeichen**:

`Makefile-subst2`:

```
LSOURCES= firstpasstex.l secondpasstex.l firstpasslex.l
LOBJECTS=$(LSOURCES:%.l=../bin/%.o)

foo:
	@echo $(LOBJECTS)
```

▶▶ `make -f Makefile-subst2`

Jetzt kann man z.B. **auch** einen **Präfix** davorhängen oder entfernen. (Wie bei genFamMem gemacht.)

4.14 Details des Aufrufs von `make`

4.14.1 Angabe von Zielen

Das **Default-Ziel**, das von `make` erzeugt wird, ist das **erste** Ziel.

Man kann aber auch **Ziele auf der Kommandozeile explizit** angeben:
`Makefile-goal`:

```
all: foo bar

foo:
	@echo "Working on foo..."
bar:
	@echo "Working on bar..."

foobar.tgz:
	@echo "Creating a distribution of foo and bar..."

clean:
	@echo "Cleaning away generated files of foo and bar..."
.PHONY: clean
```

▶▶ `make -f Makefile-goal foo`

Erzeugt nur foo, aber nicht bar

Manche Ziele werden **überhaubt nur erzeugt, wenn** sie **explizit** benannt werden:

▶▶ `make -f Makefile-goal foobar.tgz`

Und natürlich gibt es die **unechten Ziele** wie **clean**, die oft auf der Kommandozeile angegeben werden.

4.14.2 Kommandozeilenoptionen von `make`

Kommandozeilenoptionen von `make`

`-f` *file*, `--file=`*file*	nimm *file* als Makefile
`-n`, `--dry-run`	nichts wirklich tun, nur Kommandos drucken
`-t`, `--touch`	nichts tun, aber Datum aktualisieren
`-q`, `--question`	nichts tun und nichts drucken, nur Returncode setzen
var=value	Variablenzuweisung, geht vor Zuweisungen im Makefile
`-j` [*jobs*], `--jobs`[=*jobs*]	parallele Ausführung, ggf. Obergrenze für Zahl der Jobs
`-l` [*load*], `--max-load`[=*load*]	Lastgrenze für parallele Ausführung
`-k`, `--keep-going`	auch bei einem Fehler noch so viele Regeln wie möglich versuchen
`-W` *file*, `--what-if=`*file*, `--assume-new=`*file*	so tun, als ob *file* ganz neu sei (gut mit -n)
`-o` *file*, `--assume-old=`*file*	so tun, als ob *file* ganz alt sei (gut mit `--touch` danach)

Kurs Unix-Tools, WS 01/02, Jan Bredereke 28

Kommandozeilenoptionen von `make` (2)

`-h`, `--help`	drucke Hilfe
`-v`, `--version`	drucke `make`-Version
`-s`, `--silent`	drucke keine Kommandos
`-w`, `--print-directory`	drucke Verzeichnis am Anfang und Ende
`--no-print-directory`	hebt `--print-directory` auf, z.B. in Rekursion
`-C` *dir*, `--directory=`*dir*	wechsle erst ins Verzeichnis *dir*
`-d`, `--debug`[=*options*]	drucke Debug-Info
`-e`, `--environmen-overrides`	gib Variablen aus Environment Vorrang vor Makefile
`-i`, `--ignore-errors`	ignoriere alle Fehler
`-I` *dir*, `--include-dir=`*dir*	suche auch im Verzeichnis *dir* nach include-Makefiles
`-p`, `--print-data-base`	drucke alle Regeln und Variablen
`-r`, `--no-builtin-rules`	lösche alle eingebauten impliziten Regeln
`-R`, `--no-builtin-variables`	lösche alle eingebauten impliziten Regeln und die Variablen dazu
`-S`, `--no-keep-going`	hebe Wirkung von `--keep-going` auf, z.B. in Rekursion
`--warn-undefined-variables`	warnt bei Verwendung undefinierter Variablen

Kurs Unix-Tools, WS 01/02, Jan Bredereke 29

Als-Ob-Ausführung, Option **-n**:
Haben wir hier schon **oft benutzt**.

Vermeiden unnötiger Neu-Kompilation, Option **-t**:
Haben wir schon im Zusammenhang mit der **Variablen $(MAKEFLAGS)** diskutiert.

Kompilation eines Programms testen, **-k**:
Haben wir schon u.a. im Zusammenhang mit den **Returncodes** diskutiert.

Variablen überschreiben:
Haben wir im Zusammenhang mit **override** schon diskutiert.

4.15 Konventionen für Makefiles

Man sollte **nicht beliebige Systemkommandos** im Makefile benutzen, damit es **portabel** bleibt. Die folgenden sind unkritisch:

Erlaubte System-Hilfsprogramme in Makefiles

```
cat           mkdir
cmp           mv
cp            pwd
diff          rm
echo          rmdir
egrep         sed
expr          sleep
false         sort
grep          tar
install-info  test
ln            touch
ls            true
```

Nicht erlaubt: `mkdir -p`

Problematisch: Symbolische Links

System-Hilfsprogramme mit veränderlichen Namen

```
AR=ar
BISON=bison
CC=cc
FLEX=flex
INSTALL=install
LD=ld
LDCONFIG=ldconfig
LEX=lex
MAKE=make
MAKEINFO=makeinfo
RANLIB=ranlib
TEXI2DVI=texi2dvi
YACC=yacc
CHGRP=chrgrp
CHMOD=chmod
CHOWN=chown
...
```

Benutzung als `$(AR), $(BISON), $(CC), ...`

ranlib ist nicht überall vorhanden und muß ggf. ignoriert werden.

Besonders für die **Compiler-Programme** sollten Variablen vorgesehen werden, um die **Aufruf-Flags zu variieren**. Ein Beispiel war **CFLAGS** oben für CC.

Standard-Variablen für Installationsverzeichnisse

```
prefix=/usr/local
exec_prefix=$(prefix)
bindir=$(exec_prefix)/bin
libdir=$(exec_prefix)/lib
infodir=$(prefix)/info
includedir=$(prefix)/include
mandir=$(prefix)/man
man1dir=$(prefix)/man1
man2dir=$(prefix)/man2
manext=.1
man1ext=.1
man1ext=.2
...
srcdir=(bei Konfiguration gesetzt)
```

Kurs Unix-Tools, WS 01/02, Jan Bredereke 32

Vorteil dieser Variablen: Man kann **alle Gnu-Pakete** auf dieselbe Weise **an** die **aktuelle Installation anpassen**. Beispiel: **Installationsverzeichnis**.

Standard-Ziele bei Gnu-`make`

`all`	alles Übersetzen (Default-Ziel)
`install`	alles Übersetzen und installieren
`uninstall`	lösche alle installierten Dateien
`clean`	lösche alle durch `all` übersetzten Dateien im Build-Verzeichnis
`distclean`	dito, lösche zusätzlich Konfigurationsdateien
`maintainerclean`	lösche alles, was mit diesem Makefile wiederhergestellt werden kann
`info`	erzeuge `.info`-Dateien
`dvi`	erzeuge `.dvi`-Dateien
`dist`	erzeuge eine Distributions-tar-Datei
`check`	führe ggf. Selbst-Tests durch
...	...

Kurs Unix-Tools, WS 01/02, Jan Bredereke 33

4.16 Fortgeschrittene Makefile-Strukturen

4.16.1 Bedingte Teile eines Makefiles

Man kann **Teile** des Makefiles nur **unter Bedingungen** abarbeiten lassen:
`Makefile-cond`:

```
libs_for_gcc = -lgnu
normal_libs =

foo: $(objects)
ifeq ($(CC),gcc)
        $(CC) -o foo $(objects) $(libs_for_gcc)
else
        $(CC) -o foo $(objects) $(normal_libs)
endif
```

Bedingte Ausdrücke

```
ifeq (arg1,arg2)
ifneq (arg1,arg2)
ifdef variable-name
ifndef variable-name
```

Varianten für die Syntax:

```
ifeq (arg1,arg2)
ifeq 'arg1' 'arg2'
ifeq "arg1" "arg2"
ifeq 'arg1' "arg2"
ifeq "arg1" 'arg2'
```

Kurs Unix-Tools, WS 01/02, Jan Bredereke 34

Diese Syntax ist **Gnu-spezifisch**. Andere `make`s kennen bedingte Teile oft auch, aber mit leicht **variierender Syntax**.

4.16.2 Andere Makefiles einschließen

Mit

```
include datei1 datei2 ...
```

kann man **andere Makefiles einlesen**.

Wenn eine Datei **nicht existiert**, ist das zunächst noch **kein Fehler**. Wenn `make` den Rest eingelesen hat, versucht es anschließend, dieses **Makefile nach gegebenen Regeln zu erzeugen**. Erst wenn das nicht gelingt, gibt es einen Fehler.

4.16.3 Dynamisch generierte Abhängigkeiten

Es ist **sehr mühsam**, alle **Abhängigkeiten** der Quellen **von Hand** korrekt einzutragen und aktuell zu halten.

Die meisten modernen Compiler können solche `make`-Abhängigkeiten **automatisch generieren**. Man kann sie dann per `include` einschließen.

Beispiel: Siehe **genFamMem-Beispiel** von oben:.
`Makefile-genFamMem`

Man beachte auch, wie wir **explizit angegeben** haben, **wovon** die **.d-Dateien** jeweils **abhängen**. Wenn z.B. eine .c-Datei geändert wird, wird die **.d-Datei automatisch neu erzeugt**.

Wenn die **includeten Makefiles nicht existieren** oder veraltet sind, **geht `make` so vor**: In einer **ersten Runde** erzeugt `make` **nur** die **includeten Makefiles neu**. Danach **fängt es noch einmal ganz von vorne** an, mit jetzt aktuellen Makefiles.

4.17 Eingebaute besondere Ziele im Detail

Eingebaute besondere Ziele

`.PHONY:` ziel1 ziel2 . . .	Liste der unechten Ziele
`.DELETE_ON_ERROR:`	bei Fehler wird das aktuelle Ziel gelöscht (gilt für alle)
`.PRECIOUS:` ziel1 ziel2 . . .	Zieldateien bei Fehler und Unterbrechung niemals gelöscht
`.INTERMEDIATE:` ziel1 ziel2 . . .	nur bei Bedarf erzeugt, nach Gebrauch gleich wieder gelöscht
`.SECONDARY:` ziel1 ziel2 . . .	nur bei Bedarf erzeugt, aber nicht wieder gelöscht
`.SUFFIXES:` .c .o . . .	Liste der Dateinamen-Suffixe für Suffix-Regeln
`.DEFAULT: ;` anweisung . . .	Anweisungen für Vorbedingungen, die kein Ziel sind
`.NOTPARALLEL:`	keine Parallelausführung, selbst bei Option `-j`
`.IGNORE:` ziel1 ziel2 . . .	ignoriere Fehler für diese Ziele (Wenn leer: Für alle)
`.SILENT:` ziel1 ziel2 . . .	drucke keine Kommandos für diese Ziele (Wenn leer: Für alle)
`.EXPORT_ALL_VARIABLES:`	exportiere alle Make-Variablen an Kindprozesse

Kurs Unix-Tools, WS 01/02, Jan Bredereke 35