Standard– und Programmiertools für Unix

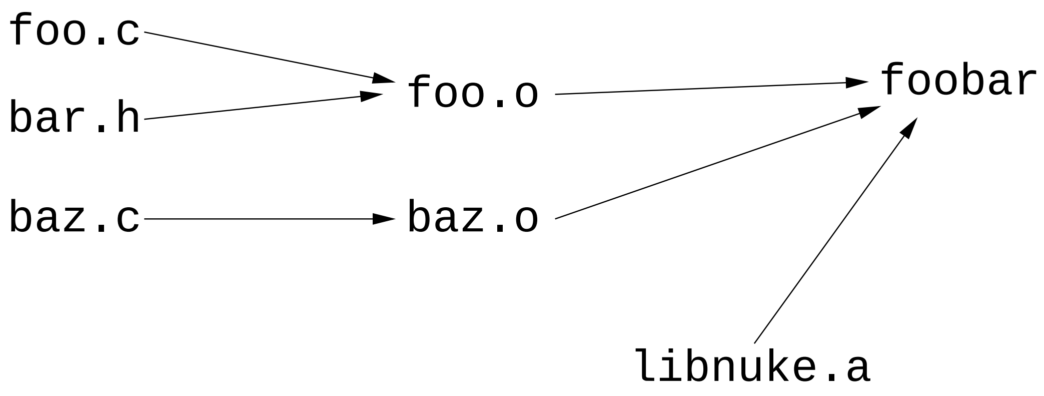
Stephan Thesing

4. GNU `make`: Verwaltung von Projekten

`make` bietet:

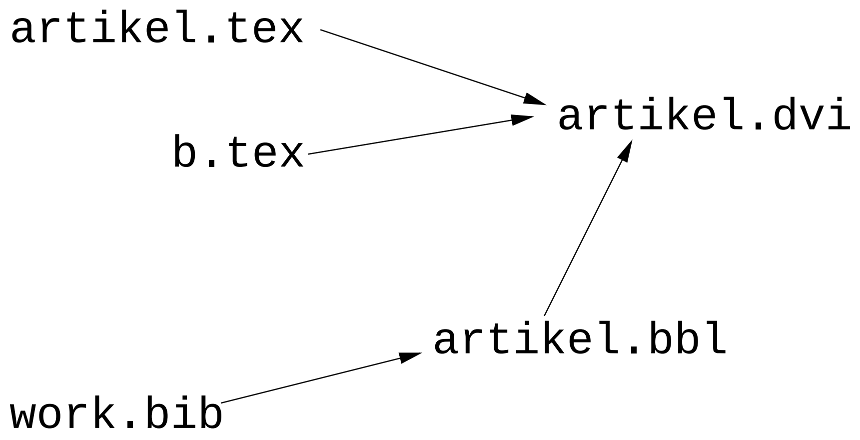
- Verwalten der Abhängigkeiten in Projekten
- Automatische Neugenerierung von Teilen nach Änderungen
- Möglichkeit der Parametrisierung nach Plattformen

Abhängigkeiten zwischen Files, z.B. Source-Code, Object-Code, Programme:



Man will bei Änderungen nur die Teile neu generieren, die von den Änderungen abhängen.

Dies funktioniert auch bei anderen Anwendungen als Compilierung:



`make` dient dazu

- die Abhängigkeiten anzugeben
- automatisch nach einer Änderung *nur* die betroffenen Teile neu zu generieren

Dazu gibt man in einem `Makefile`

- Regeln und
- Kommandos an

Beispiel eines `Makefiles`:

```
all: nexpr

%.o: %.c nexprparse.tab.h
        gcc -c $< -o $@ -O6 -Wall $(CFLAGS)

nexpr: nexpr.o nexprparse.tab.o nexprlex.o
        gcc -o nexpr nexpr.o nexprparse.tab.o nexprlex.o -ly -lfl

nexprparse.tab.h nexprparse.tab.c: nexprparse.y
        bison -d nexprparse.y

nexprlex.c: nexprlex.l
        flex -t >nexprlex.c nexprlex.l
```

Ein Aufruf von

`make`

sorgt dann dafür, daß

- das Ziel **`all`** gebaut wird, welches von **`nexpr`** abhängt
- **`nexpr`** hängt von drei Object-Code Files ab, die überprüft werden
- Alle Ziele, die von jüngeren Quellen abhängen, werden neu gebaut

Optionen:

- **`-n`** : Check
- **`-k`** : Fehler ignorieren

4.1. Regeln

Dienen zur Angabe der Abhängigkeiten zwischen Files.

Haben die Form

Ziel: *Quelle*

oder

Ziel:: *Quelle*

Es kann mehrere Ziele/Quellen in jeder Regel geben.

Ein Ziel, eine Quelle muß nicht unbedingt ein File sein (Pseudoziele):

```
all: nexpr
```

Man kann Wildcards (Shell) in Regeln verwenden:

```
foo: *.o
        gcc -o foo $^
```

Problem: Pattern wird beim Auflösen der Abhängigkeit expandiert. Meist nicht das, was man möchte.

Auch von anderen make-Programmen bekannt sind statische Pattern Regeln:

```
foo.o bar.o baz.o:%.o: %.c
        gcc -o $@ -c $<
```

Schwache Form von Pattern, eingeschränkt auf die angegebenen Files.

Mächtigere Form: Pattern-Regeln

```
%.o: %.c header.h
        gcc -c -o $@ $<
```

Man gibt Pattern für Ziel und Quelle an.

Diese Regeln werden nur benutzt, wenn es keine spezielleren für ein Ziel gibt (und die Quellen existieren/gebaut werden können).

Mehrere Ziele möglich, aber:

Alle Ziele müssen durch *einen* Aufruf der Kommandos der Regel kreiert werden.

4.2. Kommandos

Kommandos dienen dazu, Ziele zu erzeugen, deren Quellen jünger sind, als das Ziel.

Werden nach der Regel angegeben, durch ein TAB am Zeilenanfang eingerückt (wichtig!!)

Mehrere Kommandos möglich (neue Zeile)

Jedes Kommando wird in einer eigenen Subshell (normalerweise `/bin/sh`) ausgeführt.

Schlägt ein Kommando fehl, wird der Neubau abgebrochen (aber: man kann dies ignorieren)

Man kann nicht mehrere Kommandos in verschiedenen Regeln für dasselbe Ziel haben.

Ausnahme: Regeln der Form `foo::foo.o`

Man kann aber mehrere Regeln ohne Kommandos haben:

Abhängigkeiten werden dann einfach hinzugefügt.

```
foo.o: foo.c
        gcc -c foo.c
foo.o: foo.h
```

`make` kennt eine ganze Reihe bereits eingebauter Regeln.

Z.B. zum Erzeugen von Object-Code aus **C** Sourcen, etc.

Diese Regeln können z.B. auch in RCS eingecheckte Files hervorholen.

Es reicht z.B. folgendes Makefile um `foo` zu erzeugen:

```
foo: a.o b.o c.o
        gcc -o foo $^
```

Die Files `a.o`, `b.o` und `c.o` werden automatisch aus den Sourcefiles erzeugt.

Die Erzeugung wird dabei anhand der Endung gesteuert (.c $\rightarrow$ .o)

Will man die eingebauten Regeln nur für bestimmte Endungen:

```
.SUFFIXES: .c .o
```

4.3. Variablen

Man kann Variablen in Makefiles definieren.

`make` ersetzt Variablen immer textuell und interpretiert dann das Ergebnis.

Bsp:

```
CC=gcc
CFLAGS=-O6 -Wall $(INCLUDES)
INCLUDES=-I/usr/local/include -I/home/thesing/include

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

Es gibt drei Arten von Variablen:

- Rekursiv aufgelöste Variablen
- Statisch aufgelöste Variablen
- Automatische Variablen

```
CFLAGS=-O6 -Wall $(INCLUDES)
INCLUDES=...
FLAGS:=$(FLAGS) -Werror
foo.o: foo.c
        $(CC) $(CFLAGS) -c $< -o $@
```

Variablen werden verwendet, indem man ein `$`, gefolgt von dem Namen der Variablen in runden Klammern, angibt:

```
$(CFLAGS)
$(OFLAGS)$(CFLAGS)
FILES=a.c b.c c.c
$(FILES:.c=.o)
```

Speziellere Zuweisungen:

```
CFLAGS ?=-Wall
CFLAGS+=-Werror
```

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

Definiert `C` als `$(B)`, was zu `y` expandiert.

Nützlich für Flags, z.B. in xfig → EPS:

```
FIGS=fig1.fig fig2.fig
EPS=fig1.eps fig2.eps

MAG_fig1=0.5
MAG_fig2=0.7

%.eps: %.fig
	fig2dev $(FIGFLAGS) -m$(MAG_$*) $< >$@
```

Automatische Variablen (verfügbar in Kommandos)

`$@` Zielname

`$<` Die erste Quelle

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

`$^` Alle Quellen

`$*` Der Stamm einer Pattern-Regel

```
foo.o: foo.c
        $(CC) -c $< -o $@ $(CFLAGS)
print: a.ps b.ps c.ps
        lpr $? ; touch print
rm: $(FILES)
        rm -f $^
```

Spezielle Ziele

```
.PHONY: all clean
.SUFFIXES: .c .o .y .l
```

Zwischenfiles werden von `make` gelöscht.

Entstehen durch Verkettung mehrerer impliziter Regeln.

Z.B. `foo.o` aus `foo.c` erzeugt, dieses aus `RCS/foo.c,v` gemacht.

`foo.c` ist ein temporäres File und wird von `make` am Schluß gelöscht.