

## 7. Übersetzungssteuerung mit make

### Inhalte der Vorlesung

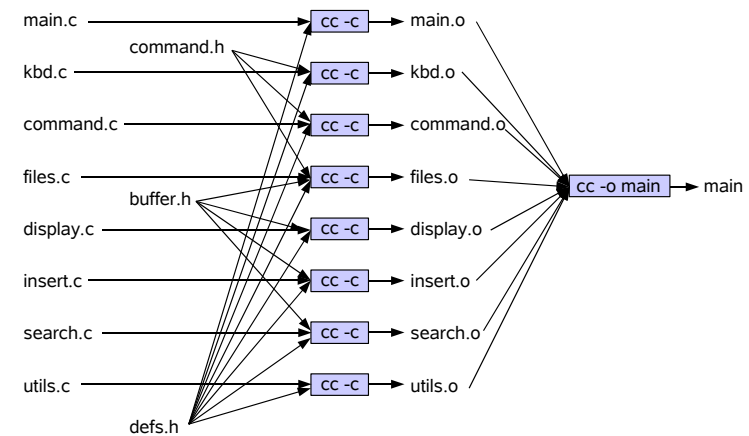
1. Einführung
2. Lexikalische Analyse
3. Der Textstrom-Editor sed
4. Der Scanner-Generator lex
5. Syntaxanalyse und der Parser-Generator yacc
6. Syntaxgesteuerte Übersetzung
- 7. Übersetzungssteuerung mit make

## 7. Übersetzungssteuerung mit make

- 7.1 Grundlagen von make
- 7.2 Arbeiten mit make
- 7.3 Ein wenig Fortgeschrittenes

### Motivation

#### Beispiel: Übersetzung eines kleinen Editors





## Motivation (2)

- einige Probleme größerer SW-Projekte:
  - viele einzelne, verschiedene Schritte bei Übersetzung
  - Reihenfolge wichtig (Abhängigkeiten!)
  - gesamte Übersetzung dauert lange
    - nach Änderung nur Teile übersetzen: schwierig
      - Abhängigkeiten!



## Motivation (3)

- Lösung: make
  - verwaltet Übersetzungsanweisungen
  - verwaltet Abhängigkeiten
  - prüft Notwendigkeit von Teil-Neuübersetzungen
    - über Veränderungsdatum der Dateien
    - Korrektheit automatisch garantiert
      - falls vollständig spezifiziert
  - viele Standard-Abhängigkeiten bereits eingebaut
    - über Datei-Endungen
  - „Makefile“: enthält alle verwalteten Informationen



## Grundlagen von Makefiles

- 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
```



## Grundlagen von Makefiles (2)

- ausführbares Programm: `edit`
- Quellen: 8 C-Quelldateien, 3 C-Header-Dateien
- „#“: Kommentar



## Regel

- Ziel
  - zu erzeugende Datei
    - oder nur Name einer Aktion
- Liste von Vorbedingungen
  - Dateien, notwendig zur Erzeugung des Ziels
- Liste von Anweisungen
  - was make tun soll
  - mehrere Anweisungen möglich
    - eine Anweisung pro Zeile



## Grundlagen von Makefiles (3)

- zeigen:
  - kbd.c
  - defs.h
- lange Zeilen: Backslash-Newline
- Beispiel für Aktion: clean
  - keine Datei erzeugt
  - oft keine Vorbedingung



## Syntax-Falle

- vor jeder Anweisungszeile: Tabulator-Zeichen!
  - 8 Blanks *nicht* erlaubt
- z.B. Editor vim hilft: solche Fehler in rot
  - ausprobieren



## Algorithmus von make

- Aufruf: make
- Vorgehen von make:
  - Default-Ziel: erstes Ziel
    - hier: edit
  - Analyse der Abhängigkeiten
    - edit hängt von 8 Objekt-Dateien ab
    - rekursiv weitere Abhängigkeiten
    - schließlich existierende Dateien ohne Regeln
  - Sortieren der Abhängigkeiten
  - Ausführen



## Algorithmus von make (2)

- Zeigen:
  - make
    - alles übersetzen
  - touch search.c
  - make
    - nur Änderungen übersetzen
  - touch buffer.h
  - make
    - kompliziertere Abhängigkeiten
  - ..... (b.w.)



## Algorithmus von make (2)

- Zeigen (2):
  - make
    - nichts zu tun, make sagt es
  - make clean
    - Aktion als explizites Ziel
    - Default-Ziel nicht benutzt



## Variablen

- Fehlerträchtig:
  - Objektdaten dreimal aufgeführt
    - Regel für edit: Vorbedingung
    - Regel für edit: Anweisung
    - Regel für clean: Anweisung
- Lösung: Variablen
  - editor-simple/Makefile-var



## Variablen (2)

- 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)
```



## Variablen (3)

- Verwendung:
  - \$ und runde Klammern
    - „\$“ in Anweisung benötigt (z.B. Shell-Variable): „\$\$“ schreiben
- Aufruf von Makefile mit Nicht-Standard-Namen
  - zeigen:
    - make -f Makefile-var
    - make -f Makefile-var clean



## Implizite Regeln

- Regeln für „C-Quelle → Objekt-Datei“: alle ähnlich
- → viele eingebaute implizite Regeln
  - orientiert an Dateiendungen
    - Beispiel: von „.c“ nach „.o“ immer „cc -c“
  - automatische Anwendung:
    - benötigt: „xy.o“
    - vorhanden: „xy.c“
      - nicht im Makefile erwähnt
      - einfach als Datei vorgefunden
  - kann Makefile viel einfacher machen
    - Beispiel: editor-simple/Makefile-impl
      - z.B. so: Makefile für die Praxis



## Implizite Regeln (2)

- 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 (3)

- zeigen:
  - make -f Makefile-impl
  - make -f Makefile-impl clean
- später genauer erklärt:
  - „.PHONY: clean“
    - niemals Datei „clean“ erzeugen
  - Minus vor „rm“
    - Fehler ignorieren, falls gar keine Dateien da

## 7. Übersetzungssteuerung mit make

7.1 Grundlagen von make

→ 7.2 Arbeiten mit make

7.3 Ein wenig Fortgeschrittenes

## Struktur eines Makefiles

- Makefile kann enthalten:
  - explizite Regel
  - implizite Regel (selbstdefiniert)
  - Variablendefinition
  - Pragmat
    - z.B. „include“
  - Kommentarzeile mit „#“

## Name für Makefile

- make sucht in dieser Reihenfolge:
  - makefile
  - Makefile
    - empfohlen, weil auffälliger in Verzeichnis
- mehrere Makefiles in einem Verzeichnis
  - make -f xy.mk
    - Datei-Endung: .mk, .m oder .mak

## Default-Regel

- ist erste Regel
  - heißt oft „all“
  - baut meist Gesamtprogramm
- Ausnahmen:
  - Ziele mit Punkt am Anfang zählen nicht
    - Beispiel: „. PHONY :“
    - haben Sonderbedeutung
  - Regel-Muster (s.u.) zählen nicht
- nicht benutzt,  
wenn Ziel(e) auf Kommandozeile angegeben

## Vorbedingungen

- Liste von Dateinamen, von Blanks getrennt
- Ziel ist veraltet, wenn:
  - Ziel älter als eine der Vorbedingungen, oder
  - Ziel existiert nicht

## Anweisungen

- ausgeführt von Unix-Shell `sh`
  - unter Windows: analoger Ersatz

## Unechte Ziele

- manchmal: Aktion, die keine Datei erzeugt
  - Beispiel: „make clean“
  - mehrfacher Aufruf: jedesmal ausgeführt
    - weil Datei „clean“ nicht existiert
- Problem: Datei „clean“ aus Versehen erzeugt
  - Regel nie mehr ausgeführt!
- Lösung bei Gnu-make:  
Ziel als „unecht“ markieren
  - make ignoriert, ob Datei „clean“ existiert
  - „.PHONY“: eingebautes Ziel mit Sonderbedeutung
    - Vorbedingungen sind unechte Ziele

Demo



## Unechte Ziele (2)

- 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)
```

## Projekt mit mehreren ausführbaren Programmen

- Problem: nur eine Default-Regel
- Lösung:
  - unechtes Ziel „all“ in Default-Regel
  - ausführbare Programme als Vorbedingungen von „all“

Demo



## Projekt mit mehreren ausführbaren Programmen (2)

- 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
```



## Projekt mit mehreren ausführbaren Programmen (3)

- „make“ baut alle drei Programme
  - trotzdem auch möglich: „make prog1 prog3“
    - baut nur prog1 und prog3
- unechtes Ziel als Vorbedingung eines unechten Ziels
  - wie Unterprogramm
  - siehe Makefile-cleanall



## Projekt mit mehreren ausführbaren Programmen (4)

- Makefile-cleanall

```
.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
          rm programm

cleanobj :
          rm *.o

cleandiff :
          rm *.diff
```



## Mehrere Regeln für ein Ziel

- Vorbedingungen vereinigt
- höchstens eine Regel mit Anweisungen erlaubt
  - falls keine: implizite Regel
- praktisch für zusätzliche Abhängigkeiten
  - besonders zusammen mit Variablen

Demo



## Mehrere Regeln für ein Ziel (2)

- Makefile-multirule

```
objects = foo.o bar.o

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

$(objects) : config.h
```
- auch mehrere Ziele in einer Regel erlaubt
  - z.B. wenn nur Vorbedingungen und keine Anweisungen

## Statische-Muster-Regeln

- für feste Liste von Zielen mit analogen Regeln
- Syntax:  
*ziele* : **ziel-muster** : *vorbedingungen*  
*anweisung*  
...
  - Ziel-Muster: enthält „%“ als Joker-Zeichen
    - Ziel-Muster *muß* auf jedes Ziel passen
  - „%“ darf auch in Vorbedingungen vorkommen
    - dadurch Vorbedingung an Ziel angepaßt

Demo



## Statische-Muster-Regeln (2)

- Makefile-statpat

```
objects = foo.o bar.o

all: $(objects)

$(objects): %.o: %.c defs.h
cc -c -o $@ $<
```
- foo.o hängt ab von foo.c und defs.h
- „Stamm“ des Ziels:
  - der Teil, der auf „%“ matcht
- „\$@“, „\$<“: automatische Variablen
  - „\$@“: aktuelles Ziel; „\$<“: aktuelle erste Vorbedingung
    - später mehr dazu

## Ausführung von Anweisungen

- **neue** Shell für jede Zeile
  - „cd xyz“ hat nicht die erwartete Wirkung

Demo



## Ausführung von Anweisungen (2)

- 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
```
- beide Ziele vorführen

## Feinheiten bei Anweisungen

- Zeile mit Anweisung **muß** mit **Tabulator**-Zeichen beginnen
  - Vorsicht mit Tabulator / Blanks!
    - Blanks vor Anweisung: Syntaxfehler
    - leere Zeile mit Tabulator vorne: Anweisungszeile ohne Anweisung
      - Achtung: zwei Regeln mit Anweisungen, ohne daß man es sieht
- leere Zeilen und Zeilen nur mit Kommentar:
  - zwischen Anweisungszeilen (und sonst auch) erlaubt
  - ignoriert

## Echo ausgeführter Anweisungen

- normal: make druckt Anweisung vor Ausführung
- „@“ vor Anweisung verhindert das

Demo



## Echo ausgeführter Anweisungen (2)

- Makefile-print

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

- Anmerkung:

- vor Kommentar ist Tabulator-Zeichen
- daher Anweisung, daher von make gedruckt
- ist *Shell*-Kommentar

## Fehlererkennung

- Anweisung kann zu Fehler führen
  - Shell liefert Return-Code
    - 0: OK, >0: Fehler
- make prüft Return-Code
  - bei Fehler:
    - Regel abgebrochen
    - (normalerweise) alles abgebrochen

Demo



## Fehlererkennung (2)

- Makefile-error

```
targets = prog1 prog2 prog3

all: $(targets)

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



## Ignorieren von Fehlern

- wenn:
  - Scheitern nicht schlimm ist
    - Beispiel: Löschen nicht vorhandener Dateien
  - Programm falschen Return-Code liefert
- „–“ vor Anweisung ignoriert Return-Code



## Ignorieren von Fehlern (2)

- Makefile-error2

```
prog = foo

.PHONY: all clean

all:

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

## Inkonsistente Zieldateien bei Fehlern oder Unterbrechungen

- bei Abbruch: Zieldatei evtl. nur halb geschrieben
  - siehe Makefile-error3
- Problem:
  - neuer Aufruf korrigiert das *nicht*
    - Datei-Datum ist ja aktuell
- Lösung:
  - spezielles Ziel „.DELETE\_ON\_ERROR:“
    - löscht Ziel bei Abbruch, falls bereits verändert
    - aus historischen Gründen leider nicht Default

Demo



## Inkonsistente Zieldateien bei Fehlern oder Unterbrechungen (2)

- 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)
```



## Inkonsistente Zieldateien bei Fehlern oder Unterbrechungen (3)

- 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)
```

## 7. Übersetzungssteuerung mit make

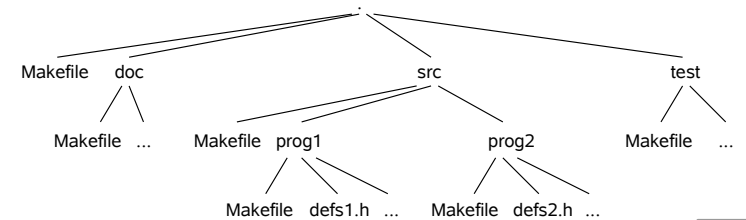
7.1 Grundlagen von make

7.2 Arbeiten mit make

→ 7.3 Ein wenig Fortgeschrittenes

## Rekursive Verwendung von make

- mehrere Unterverzeichnisse in größerem Projekt
  - Makefile in jedem Unterverzeichnis
  - Makefile im obersten Verzeichnis koordiniert Unter-Makefiles
  - Beispiel:



Demo



## Rekursive Verwendung von make (2)

- subdirs/Makefile

```
.PHONY: all clean doc src test

all: doc src test

doc src test:
    cd $@ && $(MAKE)

test: src

clean:
    cd doc && $(MAKE) clean
    cd src && $(MAKE) clean
    cd test && $(MAKE) clean
```



## Rekursive Verwendung von make (3)

- Variable „\$(MAKE)“:
  - Name des laufenden make-Programms
    - Sonderbedeutung bei „als-ob“-Ausführung
- Verkettung von Shell-Anweisungen mit „&&“:
  - zweite nur ausgeführt, falls erste erfolgreich
- zeigen:
  - doc/Makefile
  - test/Makefile
  - src/Makefile
    - noch eine Rekursionsstufe weiter



## Rekursive Verwendung von make (4)

- subdirs/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
```



## Rekursive Verwendung von make (5)

- gleiche Aufgabe wie eine Ebene höher
- hier allgemeiner gelöst, mit ein paar Tricks
  - Liste der Unterverzeichnisse in Variable
  - „all“ analog wie eben, aber generisch
  - „clean“ generisch, mit Variablenersetzung
    - Variable „\$\*“: der Stamm des Musters
    - Funktion „\$(patsubst ...)“: mustergesteuerte Ersetzung für jedes Wort in Liste
      - hier: Voranhängen von „clean\_“
- ausführen (auf oberster Ebene):
  - make
  - make clean

## Muster-Regeln

- ähnlich wie Statische-Muster-Regeln
  - aber: explizite Ziele fehlen
  - gilt implizit immer, wenn sie paßt
- Syntax:
 

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

  - Ziel-Muster: enthält „%“ als Joker-Zeichen
  - „%“ darf auch in Vorbedingungen vorkommen
- Reihenfolge von Muster-Regeln wichtig
  - erste genommen, die paßt
  - ist anders als bei Statische-Muster-Regeln

Demo



## Muster-Regeln (2)

- Makefile-patrulle
 

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

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

## Altmodische Suffix-Regeln

- gleicher Zweck wie Muster-Regeln
  - weniger flexibel
  - von allen makes unterstützt
    - Muster-Regeln Gnu-make-spezifisch

Demo



## Altmodische Suffix-Regeln (2)

- Makefile-suffix

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

.c.o:
    cc -c $< -o $@
```
- Achtung:
  - Reihenfolge der Suffixe andersherum als bei Muster-Regel
  - Vorbedingung nicht erlaubt
    - sonst als normale Regel interpretiert
  - Suffix muß in Liste bekannter Suffixe sein
    - sind die Vorbedingungen für Ziel „SUFFIXES:“

## Automatische Variablen

Variable	Bedeutung
<code>\$@</code>	<b>Ziel</b> der Regel.
<code>\$&lt;</code>	<b>Erste Vorbedingung</b> der Regel.
<code>^</code>	<b>Alle Vorbedingungen</b> der Regel. Doppelnennungen werden eliminiert.
<code>*</code>	<b>Stamm</b> des Namens in der Regel.
<code>\$(@D)</code> , <code>\$(&lt;D)</code> , ...	<b>Directory-Teil</b> des Namens.*)
<code>\$(@F)</code> , <code>\$(&lt;F)</code> , ...	<b>Datei-Teil</b> des Namens.*)
...	...

\*) Gnu-spezifisch

## Variablen in impliziten Regeln

- anpassen der vordefinierten Regeln
- einige implizite Regeln:
  - `.c.o:`

```
$(CC) $(CPPFLAGS) $(CFLAGS) $(TARGET_ARCH) -c -o $@ $<
```

    - CC = cc, andere Variablen undefiniert
  - `.o:`

```
$(CC) $(LDFLAGS) $(TARGET_ARCH) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

    - Objekt-Dateien linken (Executable ist ohne Suffix)
  - `.cc.o:`

```
$(CXX) $(CPPFLAGS) $(CXXFLAGS) $(TARGET_ARCH) -c -o $@ $<
```

    - CXX = g++, andere Variablen undefiniert
  - `.y.c:`

```
$(YACC) $(YFLAGS) $< && mv -f y.tab.c $@
```
  - `.l.c:`

```
$(LEX) $(LFLAGS) -t $< > $@
```

## Verkettung impliziter Regeln

- Beispiel:
  - Ziel: `foo`
  - vorhandene Datei: `foo.l`
  - eingebaute implizite Regeln: `.l.c`, `.c.o` und `.o`
  - `make` findet automatisch den Weg  
`foo.l` → `foo.c` → `foo.o` → `foo`
- „`make foo`“ oft ohne Makefile möglich
  - nur eingebaute implizite Regeln
- Zwischendateien automatisch wieder gelöscht

## Variablenexpansion

- erst bei Verwendung
  - Reihenfolge der Definition egal
  - Mehrfachzuweisung nicht erlaubt
    - Alternative: Anhäng-Operator
      - Beispiel:  
`CFLAGS += -Wall`  
`CFLAGS += -pg`
        - Reihenfolge: wie in Datei
      - gut für implizite Regeln

## Ersetzungen in Variablen

- Beispiel:
  - Liste von C-Quellen → Liste von Objekt-Dateien
  - `CSOURCES = input.c work.c output.c`  
`OBJECTS = $(CSOURCES:.c=.o)`
  - nur Wort-Enden bearbeitet
  - Gnu-make: auch allgemeinere Form mit „`%`“
- es gibt noch *viel* mächtigere Funktionen
  - `$(patsubst ...)`
  - ...

## Automatische Generierung von Abhängigkeiten mit Gnu-Make

- Problem:
  - Parallele Pflege der Abhängigkeiten durch „`#include`“ im C-Quellcode und im Makefile
  - fehlerträchtig
- Lösung:
  - C-Compiler generiert Liste der `#include`-Dateien
  - Daraus automatisch Makefile-Teil generieren
  - Standard-Make: „`make depend`“
  - Gnu-Make:
    - Eine `*.d`-Datei pro `*.c`-Datei, automatisch erzeugt
    - per „`include`“ in Makefile eingeschlossen



## Automatische Generierung von Abhängigkeiten mit Gnu-Make (2)

- editor-simple/Makefile-dep

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

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

# Lese die include-Dateien mit den Abhängigkeiten:
include $(objects:.o=.d)

.PHONY : clean
clean :
    -rm edit $(objects)
    -rm $(objects:.o=.d)

# Erzeuge die include-Dateien mit den Abhängigkeiten:
%.d: %.c
    $(CC) -MM $(CPPFLAGS) $< > $@.$$$$: \
    sed 's,\(($*\)\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@: \
    rm -f $@.$$$$
```

## Aufrufoptionen von make

Langform	kurz	Bedeutung
--file=file	-f file	nimm file als Makefile.
--dry-run	-n	nichts wirklich tun, nur Anweisungen drucken.
--touch	-t	nichts tun, aber Datum aktualisieren.
var=value		Variablenzuweisung, geht vor Zuweisung in Makefile.
--print-data-base	-p	druckt alle Regeln und Variablen.
--help	-h	Gib die möglichen Aufrufoptionen aus.
...	...	...

## Gnu-make und andere makes

- geht nicht mit allen makes:
    - Ersetzungen in Variablenreferenzen
    - Muster-Regeln mit %
    - Anhängen an Variablendefinitionen mit +=
    - unechte Ziele mit .PHONY
    - Statische-Muster-Regeln
    - ...
- ... also alles, was Spaß macht. ☹

## autoconf/automake - ein Überblick

- autoconf
  - erzeugt Shell-Script configure
    - konfiguriert SW-Paket automatisch
    - keine Benutzer-Interaktion
  - nur Paket-Autor muß autoconf haben
  - alle benötigten Features einzeln getestet: hybrid konfigurierte Systeme möglich
  - Skripte zur Feature-Erkennung von allen Paketen geteilt
- automake
  - erzeugt Makefile-Template für autoconf
  - alle Gnu-Standard-make-Ziele automatisch generiert

## Inhalte der Vorlesung

1. Einführung
2. Lexikalische Analyse
3. Der Textstrom-Editor sed
4. Der Scanner-Generator lex
5. Syntaxanalyse und der Parser-Generator yacc
6. Syntaxgesteuerte Übersetzung
- 7. Übersetzungssteuerung mit make