Wissenschaftliches Programmieren

Bálint Aradi
http://www.bccms.uni-bremen.de/cms/people/b-aradi

7. Abhängigkeiten, Makefiles

Quellen:

• Info-Seiten für GNU Make

http://www.gnu.org/software/make/manual/make.html

Übergabe allokierbarer Arrays

- Allokierbare Felder müssen nur dann mit dem allocatable Attribut übernommen werden, wenn ihr Allokierungsstatus im Unterprogramm verändert werden soll:

```
program main
  implicit none

  integer, allocatable :: array(:)

  allocate(array(10))
  call sub1(array)
  call sub2(array)

end program
```

Allokierungstatus **unverändert**

```
subroutine sub1(aa)
  integer, intent(inout) ::  aa

  aa = 1

end sub1
```

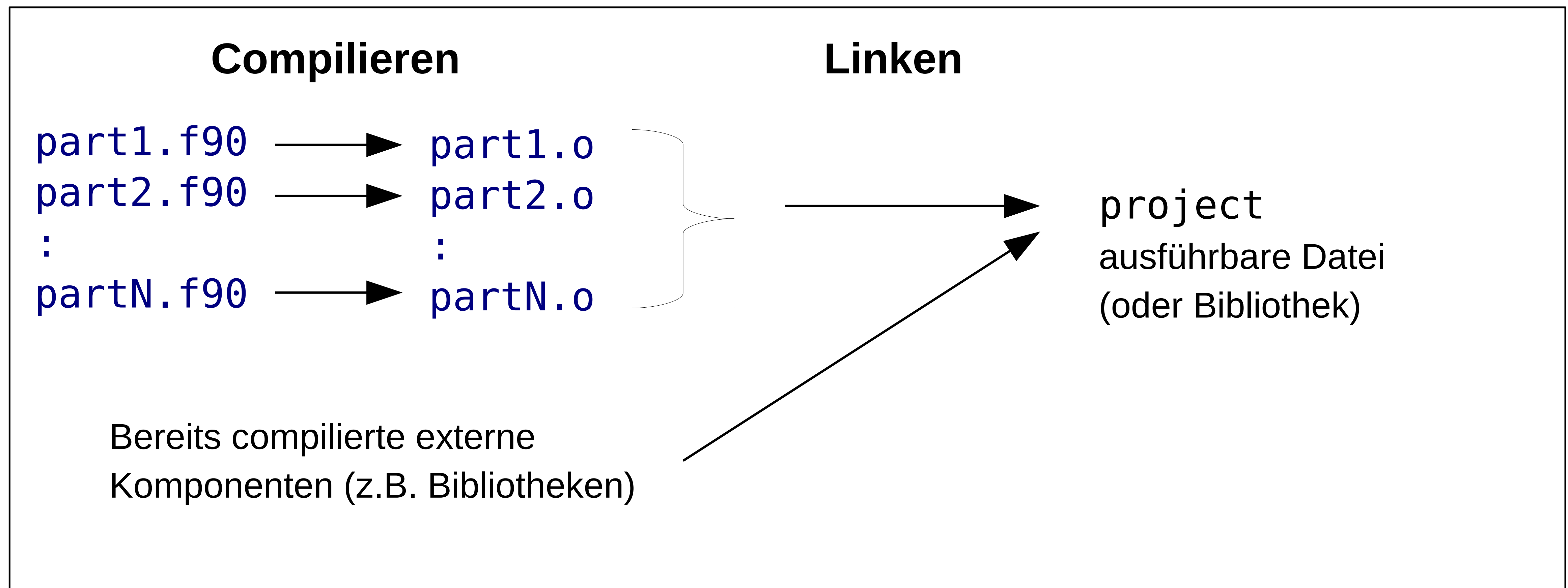
Allokierungstatus **verändert**

```
subroutine sub2(aa)
  integer, intent(inout), &
      &allocatable :: aa

  deallocate(aa)
  allocate(aa(20))
  aa = 2

end sub2
```

Compilieren/Linken

Erstellung einer ausführbaren Datei besteht grundsätzlich aus zwei Schritten:

- Compilieren der Quelldateien (Übersetzung in Maschinenformat)
- Zusammenfügen (Linken) der nötigen Komponenten



- Werden keine extra Optionen gesetzt, versucht der Compiler nach Compilieren alle angegebenen Komponenten, die entstandenen Objektdateien zu linken.

```
gfortran a.f90 b.f90
```

Compiliert a.f90, b.f90 und linkt das Ergebnis.

Compilieren

- Komponenten werden immer separat Compiliert, auch wenn sie alle als Kommandozeilenargument dem Compilier übergeben werden!

```
gfortran -o myprog a.f90 b.f90 c.f90
```

äquiv.

```
gfortran -c a.f90
gfortran -c b.f90
gfortran -c c.f90
gfortran -o myprog a.o b.o c.o
```

- Compilieren ohne Linken wird bei den meisten Compilern mit der Option „-c“ erreicht:

```
gfortran -c a.f90
```

Erzeugt Objektdatei a.o, Linken findet nicht statt.

- Beim Linken müssen alle Objektdateien, die zum Projekt gehören aufgezählt werden.
- Wird kein Name für das Ergebnis (ausführbare Datei oder Bibliothek) gesetzt (Option „-o“), wird der Name `a.out` verwendet.
- In den meisten Fällen sollte zum Linken der Compiler (und nicht `ln`) verwendet werden, da er automatisch auch die benötigten compilerinterne Bibliotheken anfügt.

```
gfortran -o test a.o b.o
```

äquiv.

```
ln -o test a.o b.o /usr/lib/gcc/i686/libgfortran.a ...
```

Compilieren

- Quelldateien werden immer einzeln compiliert.

Beim Compilieren einer Datei hat der Compiler nur beschränkte (oder gar keine) Information über die anderen Komponenten!

test.f90:

```
program test
  implicit none

  real :: i1

  i1 = 3.0
  call doubleValue(i1, 4)
  write (*,*) "i1 = ", i1

end program test
```

keine Überprüfung von Anzahl und Typ der Parameter!

sub.f90: Subroutine **außerhalb** eines Modules

```
subroutine doubleValue(val)
  implicit none
  integer, intent(inout) :: val

  val = 2 * val

end subroutine doubleValue
```

```
gfortran -c sub.f90
gfortran -c test.f90
gfortran -o test sub.o test.o
./test
```

Nicht deterministisch!

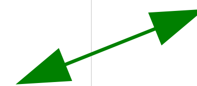
Module

- Wenn die Unterprogramme in Modulen definiert sind, ist die Überprüfung der Parameter gesichert

test.f90:

```
program test
  use sub
  implicit none

  real :: i1

  i1 = 3.0
  call doubleValue(i1, 4)
  write (*,*) "i1 = ", i1

end program test
```

Anzahl und Typ der Parameter wird überprüft!



sub.f90: Subroutine **innerhalb** eines Modules

```
module sub
  implicit none

contains

  subroutine doubleValue(val)
    integer, intent(inout) :: val

    val = 2 * val

  end subroutine doubleValue

end module sub
```

Unterprogramme sollten immer in Module gepackt werden.

```
gfortran -c sub.f90
gfortran -c test.f90
gfortran -o test sub.o test.o
```

Compiler error:

- Bad nr. of arguments
- Bad argument type

Compilieren von Modulen

- Wenn Module compiliert werden, wird extra Information über die Schnittstelle gespeichert
- Diese Information wird bei der Compilierung jeglicher Komponenten eingelesen, die das Modul via „use“ einbinden.

gausselim.f90:

```
program gausselim
  use accuracy
  use eqsolver
  implicit none
  ...
```

accuracy.f90:

```
module accuracy
  implicit none
  ...
```

eqsolver.f90:

```
module eqsolver
  use accuracy
  implicit none
  ...
```

```
gfortran -c accuracy.f90
gfortran -c eqsolver.f90
gfortran -c gausselim.f90
gfortran -o gausselim accuracy.o \
   eqsolver.o gausselim.o
```

- `gfortran -c accuracy.f90`: Erzeugt accuracy.o und accuracy**.mod**
- `gfortran -c eqsolver.f90`: Liest accuracy.mod, Erzeugt eqsolver.o und eqsolver**.mod**
- `gfortran -c gausselim.f90`: Liest accuracy.mod und eqsolver.mod, Erzeugt gausselim.o
- `gfortran -o gausselim ...`: Liest keine Schnittstelleninformationen

Schnittstellen-information (accuracy.mod, eqsolver.mod)

Compilieren von Modulen

- Reihenfolge bei Compilierung der Module wichtig:
  Quellcode eines Modules muss compiliert werden, bevor das Modul (via „use“) benutzt werden kann.

```
rm -f *.mod
gfortran -c eqsolver.f90
gfortran -c gausselim.f90
gfortran -c accuracy.f90
gfortran -o gausselim accuracy.o \
   eqsolver.o gausselim.o
```

Compilerfehler: Fehlende Modulinformation

- Der Compiler überprüft nicht, ob eine Schnittstellendatei zur aktuellen Version des Quellcodes gehört:

```
gfortran -c accuracy.f90
gfortran -c eqsolver.f90
gfortran -c gausselim.f90
gfortran -o gausselim accuracy.o eqsolver.o gausselim.o
emacs accuracy.f90
gfortran -c gausselim.f90
gfortran -o gausselim accuracy.o eqsolver.o \
  gausselim.o
```

accuracy.f90 wird geändert

Compiler liest altes accuracy.mod! (kein Fehler)

- Schnittstellendateien sind Architektur, Compiler- (und sogar Compilerversion-) abhängig!

Make

- Stellt sicher, dass Komponenten in der richtigen Reihenfolge compiliert werden bzw. bei größeren Projekten nur die nötigen Komponenten nach einer Veränderung neu compiliert werden.
- Die Information über die Abhängigkeiten/Voraussetzungen muss vom Benutzer definiert werden (eventuell mit entsprechenden Tools: z.B. automake, cmake, etc.)
- Die Konfigurationsdatei für Make ist standardisiert: *IEEE Standard 1003.2-1992*
- GNU Make benutzt zahlreiche Erweiterungen (sehr bequem, aber nicht transferabel)
- GNU Make kann praktisch für jede Architektur compiliert werden
- Möglicher Name der Konfigurationsdateien:
  - Makefile
  - makefile
  - GNUmakefile

Wird nur von GNU make eingelesen und von anderen Make Programmen ignoriert

Makefile

- Einfaches Makefile besteht aus **Regeln**:

```
Ziel(e): Voraussetzung1 Voraussetzung2 ...
[TAB]Befehl
```

Unix-Befehl, wie man *Ziel* erzeugt

Ziele, die vor dem aktuellen Ziel schon bearbeitet sein müssen

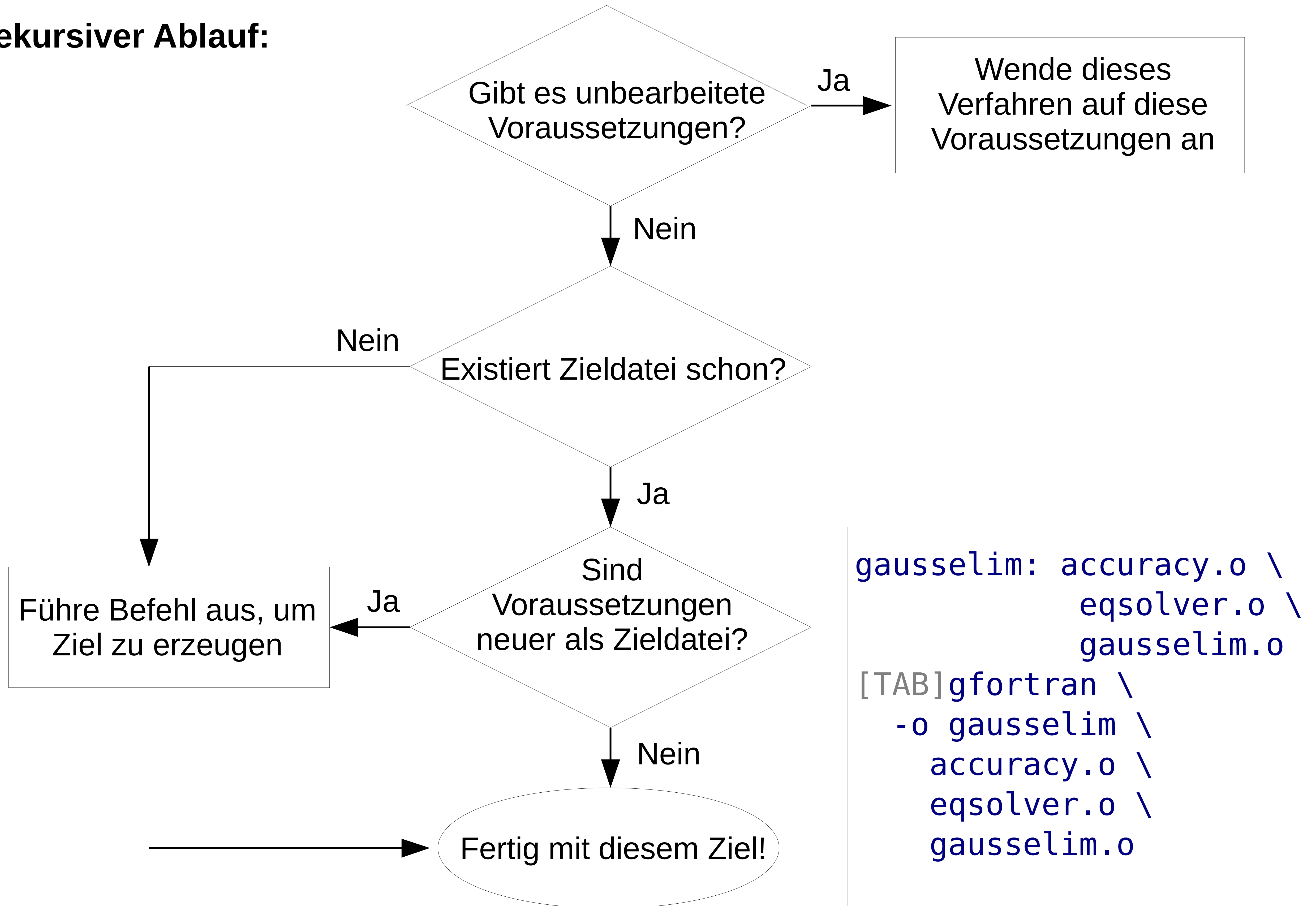
Ziel/Ergebnis der Regel

```
gausselim: accuracy.o eqsolver.o gausselim.o
[TAB]gfortran -o gausselim accuracy.o eqsolver.o gausselim.o
```

- Ziel ist meistens der Name der Datei, die erzeugt werden muss
- Voraussetzungen sind meistens Namen von Dateien, die vorhanden sein müssen, damit die Zieldatei erzeugt werden kann.
- Befehle sind Unix-Befehle, meistens Compiler- oder Linkeraufrufe, um Zieldatei zu erzeugen.

Makefile – Ablauf

Rekursiver Ablauf:



Makefile

- Regeln für das Gausselimination-Projekt:

Makefile:

```
gausselim: accuracy.o eqsolver.o gausselim.o io.o
[TAB]gfortran -o gausselim accuracy.o io.o eqsolver.o gausselim.o

accuracy.o: accuracy.f90
[TAB]gfortran -c accuracy.f90

eqsolver.o: eqsolver.f90 accuracy.o
[TAB]gfortran -c eqsolver.f90

gausselim.o: gausselim.f90 accuracy.o io.o eqsolver.o
[TAB]gfortran -c gausselim.f90

io.o: io.f90 accuracy.o
[TAB]gfortran -c io.f90
```

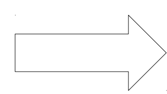
Wenn kein Ziel spezifiziert, fängt Make mit der ersten Regel an:

```
make
```

```
gfortran -c accuracy.f90
gfortran -c io.f90
gfortran -c eqsolver.f90
gfortran -c gausselim.f90
gfortran -o gausselim accuracy.o io.o
eqsolver.o gausselim.o
```

Makefile

- Make vergleicht den Zeitpunkt der letzten Veränderung der Voraussetzungen mit dem Zeitpunkt der Erstellung des Zieles (falls Ziel schon existiert)



Bei Veränderung werden nur jene Komponenten des Projektes neu compiliert, die von der veränderten Datei abhängen.

1. Projekt das erste Mal compiliert, es wird alles mitcompiliert

```
make
emacs io.f90
make
```

```
gfortran -c accuracy.f90
gfortran -c io.f90
gfortran -c eqsolver.f90
gfortran -c gausselim.f90
gfortran -o gausselim accuracy.o io.o
eqsolver.o gausselim.o
```

2. Eine Datei des Projektes wird bearbeitet

3. Objektdateien von vorangehenden Compilieren noch da, nur abhängige Teile werden neu compiliert.

```
gfortran  -c io.f90
gfortran  -c gausselim.f90
gfortran -o gausselim accuracy.o io.o
eqsolver.o gausselim.o
```

Variablen in Makefile

- Im Makefile können Variablen deklariert und verwendet werden
- Wertzuweisung mittels „=“, Verwendung: $(*Variablenname*)
- Kommentarzeilen können mit „#“ eingefügt werden

```
# Compiler, compiler options, linker, linker options
FC = gfortran
FCOPTS = -O2
LN = $(FC)
LNOPTS =

# Object files
OBJS = accuracy.o eqsolver.o gausselim.o io.o

gausselim: $(OBJS)
[TAB]$(LN) $(LNOPTS) -o gausselim $(OBJS)

accuracy.o: accuracy.f90
[TAB]$(FC) $(FCOPTS) -c accuracy.f90

eqsolver.o: eqsolver.f90 accuracy.o
[TAB]$(FC) $(FCOPTS) -c eqsolver.f90
...
```

Zwecks Wiederverwendung sollten Compilername und Compileroptionen bzw. Linkername und Linkeroptionen als Variablen deklariert werden!

Automatische Variablen

- Beispiele für automatische Variablen:

`$@`	Name des Zieles
`$<`	Name der ersten Voraussetzung
`$^`	Name aller Voraussetzungen getrennt durch Leerzeichen

- Die Wertzuweisung der automatischen Variablen erfolgt durch Make
- Die Wertzuweisung wird bei jeder Verwendung neu wiederholt:

```
gausselim: $(OBJS)
[TAB]$(LN) $(LNOPTS) -o $@ $^          <------    $@   gausselim
                                                  $^   $(OBJS)

accuracy.o: accuracy.f90
[TAB]$(FC) $(FCOPTS) -c $<             <------    $<   accuracy.f90

eqsolver.o: eqsolver.f90 accuracy.o
   $(FC) $(FCOPTS) -c $<               <------    $<   solver.f90

...
```

Zusätzliche Voraussetzungen

- Wird eine Regel ohne Befehl definiert, wird (werden) die angegebene(n) Voraussetzung(en)e als zusätzliche der Liste der Voraussetzungen angehängt.
- Wenn eine Regel mit Befehl ausgewertet wird (um ein Ziel zu erzeugen), werden alle Voraussetzungen berücksichtigt. Zuerst diejenige, die in der den Befehl enthaltenden Regel definiert sind, dann diejenige, die ohne Befehl spezifiziert wurden.
- Die zusätzliche Voraussetzungen werden auch dann berücksichtigt, wenn sie erst nach der Regel mit ausführbarem Befehl definiert sind!

```
accuracy.o: accuracy.f90
[TAB]$(FC) $(FCOPTS) -c $<

eqsolver.o: eqsolver.f90
[TAB]$(FC) $(FCOPTS) -c $<

accuracy.o:

eqsolver.o: accuracy.o
```

Bei der Auswertung wird die gesammte Voraussetzungsliste berücksichtigt: eqsolver.f90 und accuracy.o

Verändert die Voraussetzungsliste von accuracy.o nicht

Erweitert die Voraussetzungsliste von eqsolver.o mit accuracy.o

Allgemeine Regeln

- Regel nach dem selben Muster können zu einer allgemeinen Regel zusammangefasst werden. (z.B. wenn für Dateien mit gleichem Suffix immer die selbe Regel ausgeführt werden muss):

```
%.o: %.f90
[TAB]$(FC) $(FCOPTS) -c $<
```

```
accuracy.o:

eqsolver.o: accuracy.o

...
```

Die zusätzlichen (individuellen) Voraussetzungen müssen weiterhin explizit definiert werden

```
accuracy.o: accuracy.f90
[TAB]$(FC) $(FCOPTS) -c $<

eqsolver.o: eqsolver.f90
[TAB]$(FC) $(FCOPTS) -c $<

...
```

Jede .o Datei setzt die Existenz einer entsprechenden .f90 Datei voraus, und muss in der spezifizierten Weise compiliert werden

- Ziel wird via Musterabgleich gebildet, Wert für „%“ wird dann in die Liste der Voraussetzungen eingesetzt

- '%' ist **GNU-Erweiterung**, Standarformat:

```
.f90.o:
[TAB]$(FC) $(FCOPTS) -c $<
```

Unechte Ziele (*phony targets*)

Unechtes Ziel (*phony target*) = Ziel, für das bei der Ausführung des zugehörenden Befehls, keine Datei erzeugt werden sollte.

- Anwendung: Ausführung einer bestimmten Aktion (z.B. Aufräumen):

```
clean:
[TAB]rm -f *.o *.mod
```

- Wenn Make nicht mit der ersten Regel anfangen soll, muss das Ziel explizit als Kommandozeilenparameter spezifiziert werden:

```
make clean      ──────►   rm -f *.o *.mod
```

- Normale Regel werden nicht ausgeführt, wenn eine Datei mit dem entsprechenden (zufällig) existiert.

```
touch clean
make clean      ──────►   make: `clean' is up to date.
```

- **Unechte Regel** müssen deshalb mit **.PHONY**: gekennzeichnet werden, damit sie auch dann ausgeführt, wenn eine Datei mit dem Zielnamen schon existiert.

```
.PHONY: clean
```

Unechte Ziele (*phony targets*) #2

- Unechte Ziele können selber Voraussetzungen haben (meistens andere unechte Ziele):

```
.PHONY: clean realclean

clean:
[TAB]rm -f *.o

realclean: clean
[TAB]rm -f gausselim
```

```
make realclean  →  rm -f *.o
                   rm -f gausselim
```

Beenden von Make

Erfolgreiches Beenden von Make:

- Wenn alle Voraussetzungen für das ausgewählte Ziel vorhanden sind oder erzeugt werden konnten, und zum ausgewählten Ziel gehörender Befehl erfolgreich ausgeführt wurde.

Abbruch mit Fehlermeldung:

- Wenn eine der Voraussetzung nicht existiert und keine Regel zur Erzeugung definiert wurden.
- Wenn während der Ausführung des Befehlteils der bearbeiteten Regel ein Fehler aufgetreten ist (z.B. Compiler bricht die Compilierung mit Fehler ab)

Fortran 95 und Make

- Die Regeln im Makefiles basieren auf Dateinamen
- Dateiname für eine .mod Datei wird durch den Namen des Moduls bestimmt, das in der Datei definiert wird, und nicht durch den Namen der Datei, in der es definiert wird.

test.f90
```
module TestModul
:
```

```
gfortran -c test.f90
```

```
test.o
testmodul.mod
```

- Für Objektdateien aus F95-Quellcode müssen deswegen Voraussetzungen i.A. über .o und nicht über .mod Dateien spezifiziert werden!

```
irgendwas.o: testmodul.mod
```

Problematisch
Es läßt sich keine *einfache* Regel konstruieren, welche Datei compiliert werden muss, damit tesmodul.mod entsteht.

```
irgendwas.o: test.o
```

Besser
Wenn test.o erstellt wird (aus test.f90) entstehen .mod Dateien für alle Module, die im test.f90 definiert sind.

Alternativen zu Make

Schwächen von Make:

- Seltsames Format (TAB und Leerzeichen sehr leicht verwechselbar)
- Nicht 100% transferabel: Befehle funktionieren nur in Unix-Umgebungen

Alternativen zu Make:

- jam (http://www.perforce.com/jam/jam.html)
- Ant (http://ant.apache.org/)
- CMake (erzeugt makefiles für Make)
- SCons
- Waf
- Meson
- :

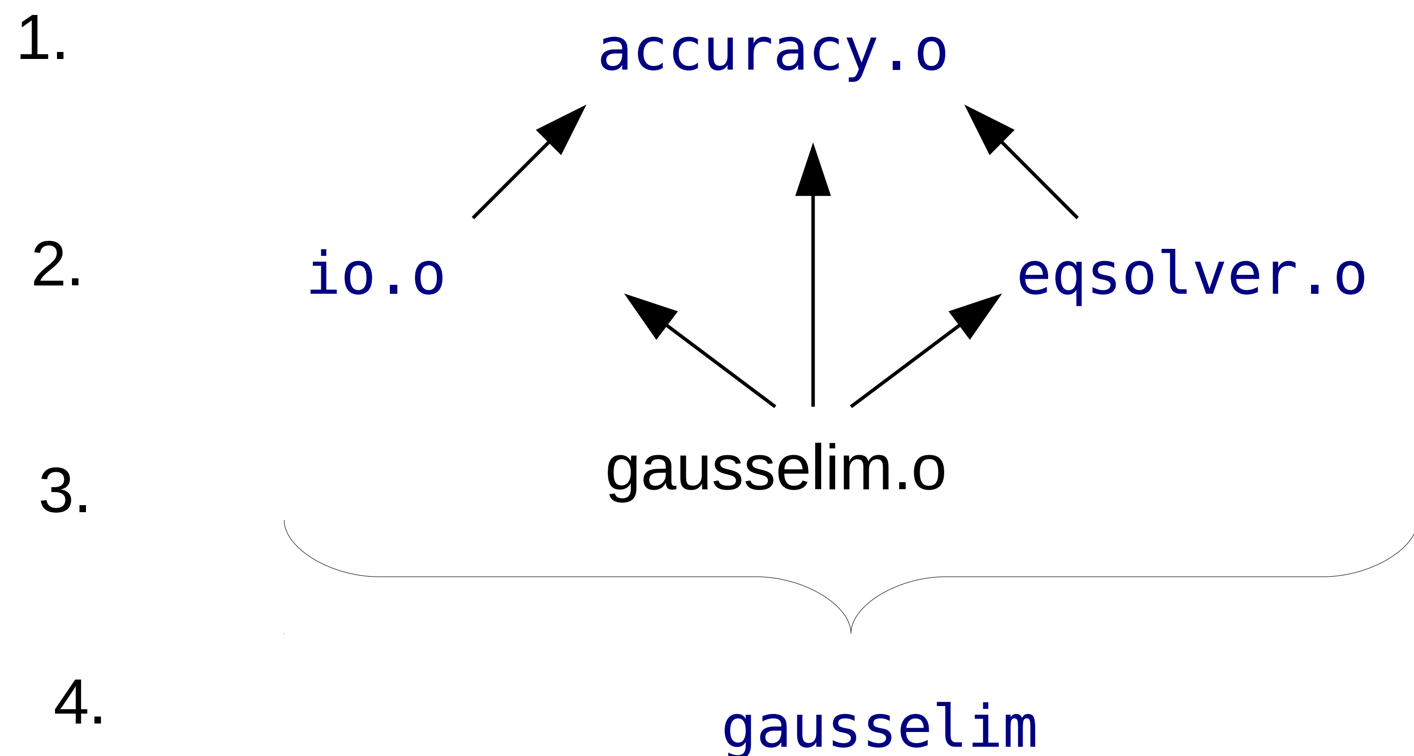
Bemerkungen zu Make

- Make wertet zuerst den gesammten Abhängigkeitsbaum aus, und fängt erst dann mit der Erstellung der einzelnen Komponenten in der richtigen Reihenfolge.
- Komponenten, die nicht voneinander abhängen können parallel bearbeitet werden. (Zeitersparnis beim Compilieren großer Projekte auf SMP-Maschinen)

`make -j2`

↑

Anzahl der parallelen Make-Threads



- Make kann für jedes Projekt verwendet werden, wo die Reihenfolge verschiedener Aktionen über Dateiabhängigkeiten sich ausdrücken lassen.

Aufgabe

- Schreiben Sie ein Makefile für das Gauss-Elimination-Projekt:
  - 'make' soll das Program compilieren
  - 'make clean' soll intermediäre Dateien (Objektfiles, mod-Dateien) löschen
  - 'make realclean' soll 'clean' ausführen und zusätzlich die ausführbare Datei löschen
  - Compiler, Compileroptionen, Linker, Linkeroptionen sollten in Variablen gespeichert werden.
- Stellen Sie sicher, dass die Abhängigkeiten richtig definiert worden sind. (Machen Sie Veränderungen in einzelnen Quellcodedateien, und vergewissern Sie sich, dass beim Neucompilieren nur die nötigen Komponenten neu compiliert werden.)
- Schreiben Sie in die README-Datei, wie man das Programm kompiliert.
- Checken Sie das Makefile in das Repository ein.