Universität Karlsruhe
Rechenzentrum

Zirkel 2
Postfach 6980
76128 Karlsruhe

Programmentwicklung unter UNIX mit make

—

eine Einführung

Release 1_0

Termin:	**Fr., 26. Januar 1996, 14-16 Uhr**
Ort:	**Raum 217, RZ-Gebäude**
Ansprechpartner:	**harald.meyer@rz.uni-karlsruhe.de**

Information zu dieser Dokumentation

Dieses Dokument wurde am 25. Januar 1996 mit LaTeX 2_ε formatiert. Sie finden in ihm die Kopien der Folien des Vortrags „Programmentwicklung unter UNIX mit make — eine Einführung" vom 26. Januar 1996 sowie die Vortragsnotizen dazu. Soweit es möglich war, wurden die Notizen zu einer Folie auf der gleichen Seite wie die Kopie der Folie untergebracht (linke Spalte: Folie, rechte Spalte Notizen).

Die Folien und die Notizen zu den Folien können Sie sich über das World Wide Web als komprimierte PostScript-Files beschaffen (URL siehe Folie „Dokumentationen zu make").

Für den Vortrag werden zwei Overhead-Projektoren benötigt (z.B. um auf einem Projektor eine Folie mit Syntax-Regeln und auf dem zweiten Projektor eine Folie mit Beispielen dazu zu zeigen).

$\boxed{R}$ ← *Abschnitte, die so eingeleitet werden und deren Text kursiv gedruckt wird, enthalten „Regieanweisungen" (z.B. Hinweise auf gleichzeitig anzuzeigende Folien).*


```
% RCS-Kennung der LaTeX-Quelle zu diesem Dokumentationsteil:
% $Source: /home/ws/rz65/Kurse/Make/LaTeX/RCS/mk_titel.tex,v $
% $Revision: 1.20 $ $Date: 1996/01/25 12:12:20 $ $Author: rz65 $
```

Gliederung

- **Problembeschreibung**
- **Literatur zu make**
- **Demos zu make**
- **Nomenklatur**
- **Optionen**
- **Syntaxregeln**
- **Library-Erstellung**
- **spezielle make-Anwendungen**

Problembeschreibung Grundregeln für das Schreiben von Programmen, Zerlegung in Teilprogramm, Beschreibung der Abhängigkeiten zwischen verschiedenenen Programmteilen durch einen Baum, detaillierte Beschreibung der Vorgänge bei der Aktualisierung der Files in den Zweigen des Baums, Automatisierbarkeit der Aktualisierung.

Literatur zu make in WWW: Folien und Notizen zu dieser Veranstaltung, make-Buch von GNU, man-Pages, Beispiele auf diversen FTP-Servers.

Demos zu make tar-Files zu dieser Veranstaltung in WWW

Nomenklatur Erläuterung der Begriffe „Makefile“, „Command“, „Rule“ ...

Optionen Optionen für den Aufruf von make

Syntaxregeln Wie schlagen sich die im Nomenklatur-Teil eingeführten Begriffe in der Syntax des Makefiles nieder?

Library-Erstellung Aufbau von Fortran-Unterprogrammbibliotheken

spezielle make-Anwendungen Beispiel: Makefile, mit dem diese Folien erstellt wurden

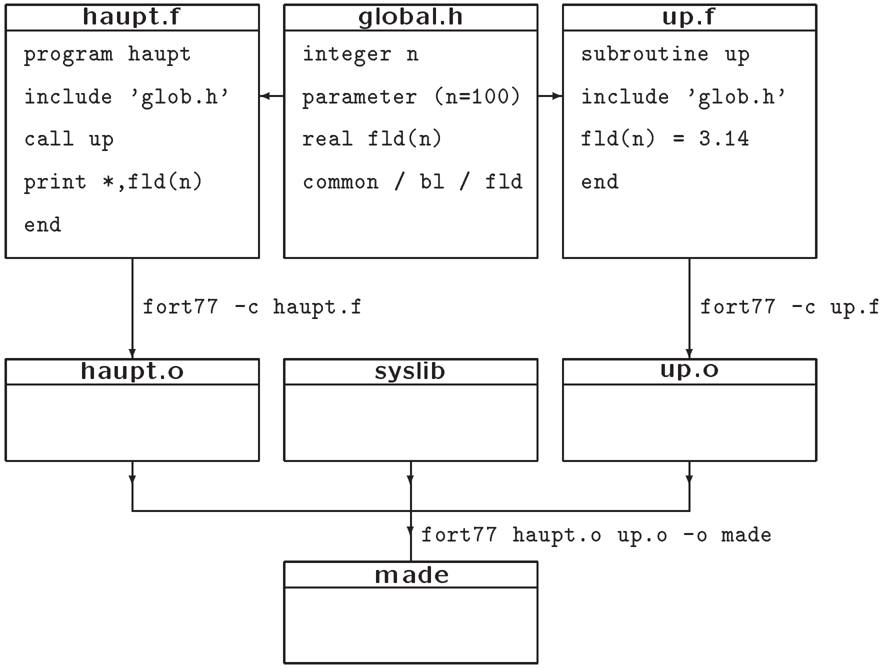
```
% RCS-Kennung der LaTeX-Quelle zu diesem Dokumentationsteil:
% $Source: /home/ws/rz65/Kurse/Make/LaTeX/RCS/mk_gliederung.tex,v $
% $Revision: 1.12 $ $Date: 1996/01/25 12:14:38 $ $Author: rz65 $
```

1. **Vermeidung überlanger Quelldateien („Spaghetti-Code“)**
2. **Zerlegung der Quelle in kleinere, übersichtliche Files**
3. **Herausziehen gemeinsamer Vereinbarungen in „Header“- oder „Include“-Files**
4. **Änderung einer Komponente erfordert den Neuaufbau bestimmter Teilzweige des Paketes**
5. **Vermeidung überflüssiger Übersetzungs- und Bindeaktionen**
6. **Automatisierung der Übersetzung, Installation in die Produktionsumgebung, Bereinigung usw.**

1. Große Dateien führen bei Editoren zu langen Wartezeiten für das Suchen, Laden und Speichern, umständlichem Blättern (Scrolling).
2. Kurze Lade-, Such- und Speicherzeiten bei Editoren, langes Blättern entfällt. Man handelt sich allerdings eine große Anzahl kleiner Files ein.
3. Falls der gemeinsame Vereinbarungsteil in jeder Teildatei physikalisch vorliegt, entstehen früher oder später Inkonsistenzen aufgrund nicht überall nachgezogener Änderungen (z.B. unterschiedliche Feldlängenangaben in Commonblöcken).
4. Welche Dateien nach einer Änderung neu übersetzt werden müssen, hängt davon ab, auf wieviele Dateien sich die Änderung auswirkt. Oft muß nur die lokale Programmeinheit übersetzt werden, in anderen Fällen (z.B. bei Header-Files oder bei tief verschachtelten Abhängigkeiten) sind zahlreiche Übersetzungen notwendig.
5. Die Holzhammermethode (Übersetzung aller Dateien) ist zwar sicher, kostet jedoch (besonders bei hohem Optimierungslevel des jeweiligen Compilers) sehr viel Zeit und ist daher sehr ineffizient.
6. Die Programmerstellung besteht nicht nur aus dem in der Entwicklungsumgebung ablaufenden eigentlichen Übersetzungs- und Bindevorgang, sondern auch aus Tests sowie aus der Installation des getesten Programms in der Produktionsumgebung, wofür u.U. umfangreiche Kopiermaßnahmen bzw. Zugriffsrechtseintragungen notwendig sind. Auch diese sollten automatisiert werden.

```
% RCS-Kennung der LaTeX-Quelle zu diesem Dokumentationsteil:
% $Source: /home/ws/rz65/Kurse/Make/LaTeX/RCS/mk_problembeschreibung.tex
% $Revision: 1.27 $ $Date: 1996/01/25 12:18:02 $ $Author: rz65 $
```

- **Hauptprogramm 'haupt.f'**
- **Unterprogramm 'up.f'**
- **Headerfile 'glob.h'**
- **Ziel: Erstellung des ausführbaren Programms 'made'**
- **WWW-Demo-File:** `demo1.tar.gz`



- Ein komprimiertes tar-File mit den Quellen und dem Makefile zu diesem Beispiel finden Sie auf der RZ-WWW-Seite zu make (siehe Folie „Dokumentationen zu make"), die Datei heißt `demo1.tar.gz`.
- Globale Vereinbarung (z.B. das Common-Feld `fld` der Länge `n`) sollten nicht lokal in jeder Programmeinheit eingetragen werden, sondern in einem Headerfile stehen, welches vom Compiler mittels der 'include'-Direktive zur Übersetzungszeit dynamisch eingebaut wird. Hierdurch werden Inkonsistenzen bei der Änderung globaler Objekte vermieden.
- Globale Feldlängen sollten niemals als Integer-Konstanten im Quelltext auftauchen, sondern als Parametervariablen (vgl. File 'glob.h', Parametervariable `n`). Auf diesem Wege muß man nicht an Dutzenden von Stellen (DO-Schleifen usw.), sondern nur an einer einzigen Stelle Änderungen vornehmen, falls ein Feld nachträglich eine andere Länge erhalten soll.
- Große Felder sollten in Fortran 77 stets in Commonblöcken abgelegt werden, da große 'lokale' Felder bei vielen Unix-Varianten Probleme mit dem Variablen-Stack heraufbeschwören. Dies ist keine Fortran-spezifische Besonderheit; auch bei anderen Programmiersprachen sollten Stack-Variablen nicht zu groß werden, z.B. sollten in C keine riesigen Objekte der Storage Class 'auto' angelegt werden (Abhilfe: große Felder mit `malloc` allocieren).
- Die einzelnen Fortranprogrammeinheiten werden getrennt mit der Option `-c` übersetzt, hierdurch entsteht aus einem Fortran-File `xxx.f` ein Object-File `xxx.o`. Die Option `-c` verhindert, daß unmittelbar im Anschluß an die Übersetzung gleich der Linker zum endgültigen Binden aufgerufen wird (dies geschieht erst nach der Erstellung aller Object-Files.
- Im letzten Schritt wird `fort77` nicht mit `.f`-Argumenten aufgerufen, sondern mit einer Reihe von `.o`-Argumenten (beachte: keine `-c` Option, d.h. es wird fertig gebunden). In diesem Falle versteckt sich hinter `fort77` der Linker `ld`. Er bindet die Fortran-Object-Files mit den notwendigen Routinen des Fortran-Laufzeitsystems (Syslib) zu einem ausführbaren Programm `made` zusammen. Die Option `-o` dient zur Vereinbarung des Namens des zu erstellenden Executables; ohne Angabe dieser Option lautet der Name bei Unix-Systemen `a.out`.

- Hinweis: Der direkte `ld`-Aufruf anstelle des `fort77`-Aufrufs führt i.A. nicht zum Erfolg, da der `ld` im Gegensatz zum `fort77` nicht weiß, welche Teile der Syslib noch hinzugebunden werden müssen. Das Gleiche gilt auch für C, auch dort muß zum endgültigen Binden anstelle des `ld` der `cc` bzw. `c89` aufgerufen werden.
- Frage: Warum wird bei der Erstellung des ausführbaren Programms nicht gleich alles in einem Aufwasch (z.B. durch ein Unix-Statement

  ```
  fort77 haupt.f up.f -o made
  ```

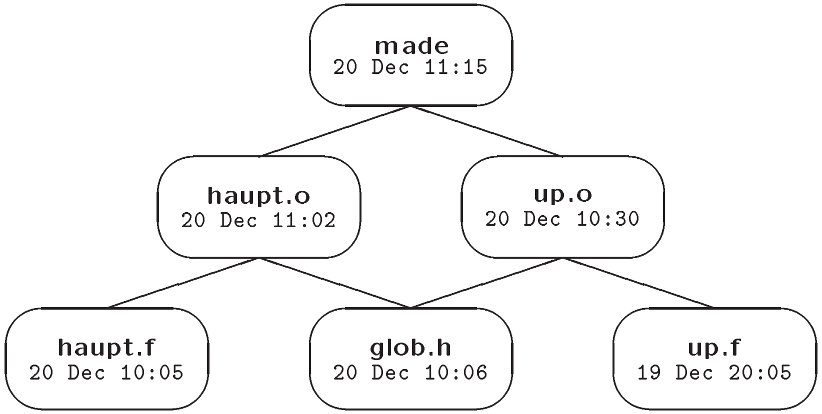
  o.ä.) erledigt ? Antwort: Hierdurch würden bei der Programmentwicklung jedesmal alle Programmeinheiten übersetzt werden, selbst wenn man nur an einigen wenigen Änderungen vorgenommen hätte. Dies ist bei umfangreichen Paketen sehr ineffizient. Dort ist es besser, nur die Aktionen zu starten, die aufgrund der Veränderungen notwendig wurden, das Gros der Files wird in der Regel nicht davon betroffen sein.

R *Folie liegen lassen, da die Nachfolgefolie auf sie Bezug nimmt.*

Seite wurde absichtlich frei gehalten

Abhängigkeitsbaum zu dem Problem

- **Knoten: Dateien**
- **Wurzel: zu erstellendes Programm**
- **Blätter: Quelldateien**
- **Aktualisierung längs der Äste in Richtung Wurzel (Übersetzung, Binden)**
- **Vgl. last modification dates der Knoten → Kriterium zur Aktualisierung**



Abhängigkeitsbaum zu dem Problem

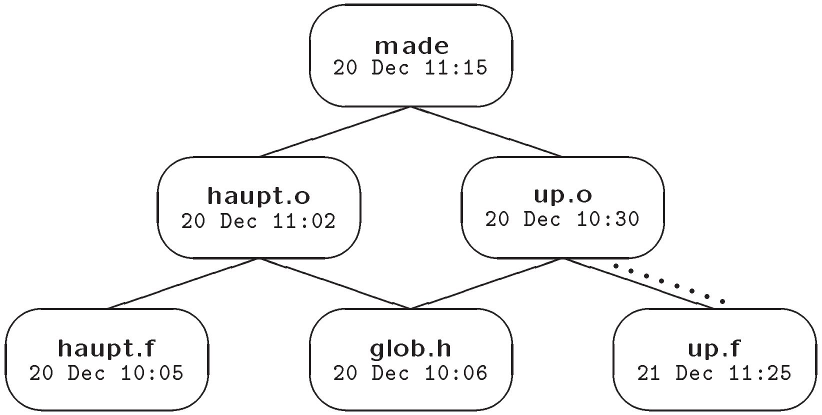
- Was bei der Veränderung an einer Datei an Aktionen notwendig wird, kann man sich anhand einer Baumstruktur veranschaulichen, bei der für jedes File aufgetragen wird, von welchen anderen Files es abgeleitet wird. Für jede Datei `xxx.y` erhält man auf diesem Wege einen Teilbaum, in dessen Knoten all diejenigen Files aufgeführt sind, bei deren Änderung auch `xxx.y` neu erstellt werden muß. Beispiele:
  - Die Datei `up.o` hängt von `up.f` und `glob.h` ab bzw. wird von diesen abgeleitet.
  - Das ausführbare Programm `made` wird von den beiden Dateien `up.o` und `haupt.o` abgeleitet (und streng genommen auch noch von der Fortran-Systembibliothek), jedoch nicht unmittelbar von `haupt.f`, `up.f` oder `glob.h`.
- An der Wurzel des Baums (oben) findet man das zu erstellende ausführbare Programm, die Blätter (unten) werden von den eigentlichen Quellfiles gebildet (oder bei Anwendung eines Versionsverwaltungssystems wie RCS oder SCCS von den dazugehörigen Archiv-Dateien), in den dazwischenliegenden Knoten findet man die Object-Files. Die Übergänge in Richtung Wurzel beschreiben die einzelnen Übersetzungs- und Bindevorgänge.
- Das Unix-Betriebssystem speichert für jede Datei Datum und Uhrzeit der letzten Änderung (last modification date (LMD), intern wird hierfür die Zahl der Sekunden zwischen dieser Änderung und dem 1.1.1970 UTC ausgewertet). Dies liefert einen Schlüssel für die Entscheidung, ob ein Knoten aus den Files, von denen er abgeleitet wird, neu erstellt werden muß (z.B. durch Übersetzen oder Binden): Ein Knoten muß aktualisiert werden, falls mindestens einer seiner Kinderknoten ein jüngeres LMD aufweist als er selbst. Trivialer Sonderfall: Falls der Knoten noch gar nicht existiert, muß er natürlich auch aus seinen Kinderknoten abgeleitet werden (erstmalige Übersetzung bzw. erstmaliges Binden).
- Eine Datei kann nicht noch einmal als Kind, Enkel usw. in seinem eigenen Abhängigkeitsteilbaum auftreten, da sonst das bekannte Huhn-Ei-Problem auftritt (wer war zu erst da ?). Falls bei der Beschreibung der Abhängigkeiten ein solcher Zyklus auftritt, wird dies von `make` als Fehler erkannt.

- Eine Datei kann in mehreren Teilbäumen als Knoten auftreten (beispielsweise die Datei `glob.h`, welche sowohl in dem Teilbaum zu `haupt.o` als auch im Teilbaum zu `up.o` auftritt).
- Falls an einem Kind Änderungen auftraten, muß sein nächsthöherer Knoten aktualisiert werden, dessen Änderung macht die Aktualisierung seines nächsthöheren Knotens notwendig usw. Auf diese Weise pflanzt sich eine Welle von Aktualisierungen in Richtung Wurzelknoten fort.
- Die verschiedenen Knotendateien müssen nicht unbedingt im gleichen Filesystem liegen, sondern können physikalisch auf unterschiedlichen Fileserver-Rechnern abgelegt sein. Zu Vermeidung von fehlerhaften LMD-Vergleichen müssen die Uhren der verschiedenen Fileserver möglichst genau miteinander synchronisiert werden, andernfalls entstehen schwer zu erkennende Fehler. Anm.: Das RZ synchronisiert die Uhren seiner Workstations mittels des Network-Time-Service, dessen Bezugsuhr wird von einem Langwellenempfänger gesteuert, welcher die Radiosignale des Normalzeitsenders der Physikalisch-Technischen Bundesanstalt empfängt. Auf diese Weise laufen die RZ-Workstations bis auf wenige Millisekunden synchron.
- Die verschiedenen Fileserver können sogar in verschiedenen Zeitzonen liegen. Hieraus ergeben sich (bei Synchronisation der Uhren) keine Probleme, da im LMD nicht die lokale Zeit angegeben wird, sondern die UTC-Zeit.

R *Folie liegen lassen, da die Nachfolgefolien auf sie Bezug nehmen. Die Vorgängerfolie (Quelltextbrocken) kann nun entfernt werden.*

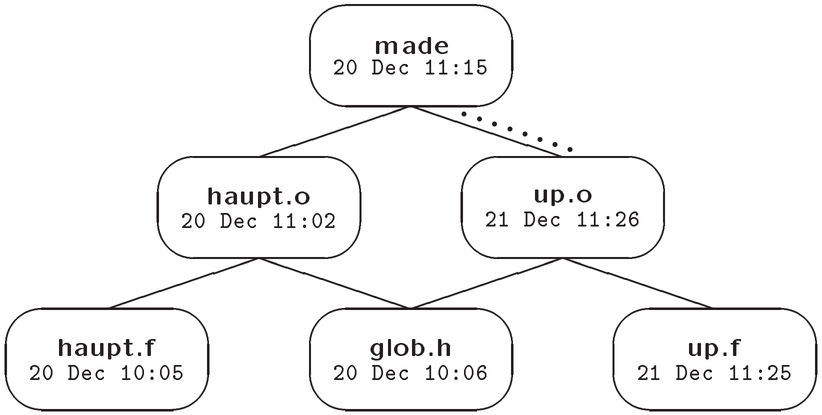
Seite wurde absichtlich frei gehalten

Auswirkung einer Änderung an 'up.f'



'up.f' muß neu übersetzt werden
→ 'up.o'-Aktualisierung

Auswirkung der 'up.o'-Aktualisierung



'made' muß neu gebunden werden.

- Bei der ersten Aktualisierung wird das LMD der Datei `up.f` verändert, z.B. dadurch, daß man an der Datei mit einem Editor Korrekturen angebracht hat. Das last modification date weist danach den Wert `21 Dec 11:25`.
- Vergleicht man nun das LMD der Datei `up.o` (`21 Dec 11:25`) mit dem LMD der Datei `up.f`, aus der `up.f` abgeleitet wurde, so ergibt sich, daß `up.o` neu aufgebaut werden muß.
- Durch den Neuaufbau von `up.o` wird dessen LMD geändert, und zwar auf `21 Dec 11:26`. Diese Änderung führt zu einer weiteren Änderung in Richtung Wurzel des Abhängigkeitsbaums, da die Datei `made`, die aus `haupt.o` und `up.o` abgeleitet wird, nun ebenfalls ein „abgelaufenes" LMD (`20 Dec 11:15`) aufweist.
- Durch Neuaufbau von `made` aus `haupt.o` und `up.o` wird das LMD von `made` verändert, sodaß es wieder jünger ist als die LMD-Werte der Files, von denen `made` abgeleitet wurde. Nun ist die gesamte Aktualisierung des Baumes abgeschlossen.

Automatisierte Aktualisierung: make

- **Gründe für die Automatisierung:**
  - **manuelle Bearbeitung: stumpfsinnig**
  - **Datumsvergleich sehr einfach**
  - **einfache Aktualisierungen-Schemata**
  - **rekursive Abarbeitung des Baums**
- **manuell zu errichten: Beschreibung der Abhängigkeiten durch ein Textfile in einem standardisierten Format.**
- **Einheitliches, einfaches Starten der Aktualisierung: Kommando** `make`
- **GNU-Version von make:** `gmake`

- Beim Aufbau verschiedener Programmpakete gibt es trotz der inhaltlichen Unterschiede viele Ähnlichkeiten, welche die Automatisierung des Aufbaus nahelegen:
  - Die manuelle Bearbeitung des Abhängigkeitsbaums ist sehr fehleranfällig, da einerseits die Abhängigkeiten der diversen Files exakt beachtet werden müssen (kein Ast darf vergessen werden), andererseits die an den verschiedenen Knoten durchzuführenden Aktionen (LMD-Vergleich, Compiler- und Linkeraufruf) aufgrund ihrer Monotonie schnell abstumpfend wirken.
  - Der Vergleich der Aktualisierungszeitpunkte als Kriterium für einen Compiler- bzw. Binderlauf kann leicht automatisiert werden, da das Betriebssystem diese Zeitpunkte stets bei einer Datei mitführt und auch aktualisiert.
  - Die Übersetzungs- oder Bindevorgänge laufen für die meisten Quell- bzw. Object-Files immer nach einem vorgegebenen relativ starren Schema ab, welches leicht durch bestimmte Textmuster beschrieben werden kann und damit automatisch ablaufen kann.
  - Die Abarbeitung des u.U. tief verschachtelten Abhängigkeitsbaums kann durch Anwendung von Rekursion in kleine handliche „Brocken“ zerlegt werden.
- Was bleibt zu tun: Die Beschreibung der Abhängigkeiten hängt natürlich von dem aktuellen Programmpaket ab, sie ist daher für jedes Paket gesondert zu erbringen. Allerdings gibt es zahlreiche Utilities, um gewisse Abhängigkeiten (z.B. `#include` von Headerfiles) automatisch zu erkennen und zu formulieren. Die Formulierung dieser Abhängigkeiten geschieht mittels eines einheitlichen Formats in einem Textfile, dem Makefile.
- Nachdem das Makefile einmal erstellt ist, kann der Neuaufbau des Paketes jederzeit durch ein kurzes Unix-Kommando eingeleitet werden. Dieses lautet im einfachsten (und häufigsten) Fall: `make`.
- Die GNU-Version von make weist einen umfangreicheren Satz eingebauter Regeln und Erweiterungen auf als die meisten herstellereigenen make-Varianten. Zur Vermeidung von Namenskonflikten mit dem herstellereigenen make installiert das RZ die GNU-Version unter dem Namen `gmake` (analog zu `gtar`, `gcc`).

RZ-WWW-make-Seite:

http://www.uni-karlsruhe.de/~Harald.Meyer/Make/

1. **Folien zur Veranstaltung:**
   WWW-make-Seite: slides.ps.gz
2. **Beispiel-tar-Files dazu:**
   WWW-make-Seite: demo1.tar.gz, demo2.tar.gz, ...
3. **Dokumentation zu gnu-make:**
   R. M. Stallman, R. McGrath:
   GNU Make – A Program for Directing Recompilation
   WWW-make-Seite: gmake.book.ps.gz
4. **man-Pages in der WWW-make-Seite:**

AIX:	make.aix.ps.gz
HP-UX:	make.hpux.ps.gz
IRIX:	make.irix.ps.gz
Linux:	make.gnu.ps.gz
UXP/M:	make.uxpm.ps.gz

5. **Makefile zu div. Paketen:**
   ftp ftp.uni-stuttgart.de
   ftp ftp.ira.uka.de

1. Die Folien wurden verkleinert (vier Folien pro A4-Blatt) und in einer komprimierten PostScript-Datei abgelegt. Zur Ausgabe auf der Xerox 4050 ist folgendes Unix-Kommando anzugeben:

   ```
   pm -dxerox slides.ps.gz
   ```

   Hinweis: Das File muß vor dem Absenden nicht dekomprimiert werden, `pm` versteht auch komprimierte Files.
2. Sie können sich die komprimierten tar-Files aus dem WWW holen und in Ihrem Arbeitsverzeichnis auspacken. Falls Sie über eine gnu-tar-Version verfügen, lautet das Auspack-Kommando:

   ```
   gtar -xvzf demo1.tar.gz
   ```

   Andernfalls folgendes Kommando verwenden:

   ```
   cat demo1.tar.gz | gzip -d | tar -xvf -
   ```
3. Die gnu-make-Dokumentation wurde als komprimierte PostScript-Datei (gnu-smallbook, 4 Seiten pro Blatt) abgelegt. Die darin angegebenen Beispiele beziehen sich zumeist auf C-Programme, nicht auf Fortran-Programme. Zur Ausgabe auf der Xerox 4050 ist folgendes Unix-Kommando anzugeben:

   ```
   pm -dxerox gmake.book.ps.gz
   ```
4. Die make-Varianten verschiedener Hersteller können sich geringfügig voneinander unterscheiden. Die make-Man-Pages diverser Plattformen können Sie nach dem Herunterladen aus WWW mittels `pm` ausdrucken, z.B.:

   ```
   pm -dxerox make.hpux.ps.gz
   ```
5. Beispiele zu Makefiles finden Sie in Hülle und Fülle auf diversen Anonymous-FTP-Servern, z.B. auf den beiden angegebenen Servern am Rechenzentrum der Uni Stuttgart bzw. bei der Informatik-Rechnerabteilung in Karlsruhe. Die Files liegen zumeist als komprimierte tar-Files (Endungen `.tar.Z` bzw. `.tar.gz`). Diese können durch folgende Unix-Kommandos ausgepackt werden:

   `cat abcdefg.tar.Z|uncompress|tar -xvf -` oder
   `gtar -xvZf abcdefg.tar.Z` bzw.
   `cat abcdefg.tar.gz|gzip -d|tar -xvf -` oder
   `gtar -xvzf abcdefg.tar.gz`

```
% RCS-Kennung der LaTeX-Quelle zu diesem Dokumentationsteil:
% $Source: /home/ws/rz65/Kurse/Make/LaTeX/RCS/mk_dokumentation.tex,v $
% $Revision: 1.16 $ $Date: 1996/01/15 19:49:46 $ $Author: rz65 $
```

Begriff	Erläuterung/Beispiel
Makefile	Textdatei mit Abhängigkeitsbaum, Regeln usw. `Makefile,makefile`
Target	Knoten; muß aktualisiert werden, wenn er veraltet ist. `up.o`
Dependency	Kinderknoten, von dem ein Target abgeleitet wird `up.f,glob.h`
Command	einzelne Unix-Anweisung `fort77 -c up.f` `@echo "Converting..."` `-rm *.o`
Implicit Rule	allgemeine Regel; gibt an, wie Targets eines bestimmten Typs aus Dependencies eines anderen Typs abgeleitet werden.
Explicit Rule	Regel; bestimmt die Ableitung eines konkreten Targets aus seinen Dependencies
Macro	Variable, z.B. `$(FC)`
Goal	eine Art „Haupt“-Target

R *Auf dem anderen Projektor sollte immer noch die Folie mit dem Abhängigkeitsbaum liegen, da auf sie verwiesen wird.*

Makefile Diese Textdatei beschreibt den Abhängigkeitsbaum und die Regeln, nach denen Knoten aktualisiert werden. Die Datei wird manuell mit einem Editor oder unter Zuhilfenahme bestimmter Utilities erstellt. make sucht nach einer solchen Textdatei namens `Makefile` oder `makefile`. Es können jedoch auch andere Namen angegeben werden (bei Verwendung der Option `-f`, s.u.).

Target Knoten, welcher im Rahmen einer Aktualisierung neu aufgebaut werden muß, da mindestens einer seiner Kinderknoten jünger ist als er selbst.

Dependency Kinderknoten zu einem bestimmten Target. Das last modification date einer jeden Dependency wird mit dem last modification date des Targets verglichen.

Command Unix-Kommando, welches bei der Aktualisierung eines Targets ausgeführt wird. make ruft hierzu eine der üblichen Command Shells (z.B. `/bin/sh`) auf, dabei sind die Syntaxregeln der Shell zu beachten (Metazeichen-Ersetzung usw.). make führt jede Command-Zeile in einem eigenen Shell-Aufruf aus; so daß z.B. lokale Directory-Wechsel in einer Zeile in der Nachfolgezeile nicht mehr gelten. Die folgenden beiden Command-Zeilen führen nicht dazu, daß in der 2. Zeile das Command im Directory `schnoed` ausgeführt wird:

```
cd␣schnoed
fort77␣-c␣prog.f
```

Abhilfe in solchen Fällen: Die beiden Kommandos unter Einschaltung des Trennzeichens ; in eine Zeile zusammenziehen:

```
cd␣schnoed;␣fort77␣-c␣prog.f
```

Normalerweise protokolliert make alle Kommandos, die es ausführt, auf der Standardausgabe mit. Dies kann durch Voranstellen eines `@` vor das Kommando verhindert werden (siehe `@echo`-Beispiel).

Tritt während der Ausführung eines Kommandos ein Fehler auf, wird der make-Lauf normalerweise beendet, dies kann durch Voranstellen eines Minuszeichens vor das Kommando verhindert werden (siehe `-rm`-Beispiel).

```
% RCS-Kennung der LaTeX-Quelle zu diesem Dokumentationsteil:
% $Source: /home/ws/rz65/Kurse/Make/LaTeX/RCS/mk_nomenklatur.tex,v $
% $Revision: 1.15 $ $Date: 1996/01/15 19:46:40 $ $Author: rz65 $
```

Implicit Rules Die Verwendung impliziter Regeln erspart einem bei umfangreichen Paketen sehr viel Schreibaufwand, da man hier nicht für jede (Target,Dependencies)-Kombination einzeln vorschreiben muß, was bei der Aktualisierung getan werden soll, sondern für alle Targets bzw. Dependencies eines bestimmten Typs festlegen kann, wie die Aktualisierung aussieht. So kann man z.B. eine allgemeine Regel angeben, nach der Fortran-Quelldateien in die entsprechenden Object-Files umgesetzt werden. Die Regel wird anhand des Extension-Paares `.f.o` make gegenüber mitgeteilt, welches daraufhin jede Fortran-Datei (Extension `.f`) dementsprechend in das dazugehörende Object-File (Extension `.o`) übersetzen kann. make kennt einen fest eingebauten Satz impliziter Regeln (built-in rules), dessen Umfang von der verwendeten make-Version abhängt. Im allgemeinen weist die GNU-Version einen besseren Regelsatz als die proprietären make-Versionen auf. Sollten die built-in rules nicht ausreichen, kann man im Makefile weitere implicit rules vereinbaren.

Explicit Rules Hierdurch wird für eine ganz bestimmte Kombination (Target,Dependencies) vereinbart, wie die Aktualisierung ablaufen soll. Diese Regel gilt dann nur für die genannte Kombination, alle anderen Kombinationen des gleichen Target- bzw. Dependency-Typs müssen durch andere (explizite oder implizite) Regeln verarbeitet werden. Wegen des Schreibaufwands geht man i.A. sparsam mit expliziten Regeln um, sie sind aber in manchen Fällen notwendig, z.B. wenn ein bestimmtes Fortran-File auf eine andere Art in das entsprechende Object-File übersetzt werden soll als die restlichen Fortran-Quellen. Hierbei wird man das Gros der Fortran-Files mit einer impliziten Regel verarbeiten, dem Ausreißer wird eine Extrawurst in Form der expliziten Regel gebraten. Merke: Gibt es für eine bestimmte (Target,Dependencies)-Kombination sowohl eine implizite als auch eine explizite Regel, so wird die explizite vorgezogen.

Macro Macros sind eine Art Variablen, welche innerhalb des Makefiles als Platzhalter dienen können. Beim Macronamen unterscheidet make zwischen Groß- und Kleinschreibung, so sind z.B. `SHELL` und `Shell` verschiedene Macros, die durchaus unterschiedliche Werte aufweisen können,
z.B. `SHELL=/bin/shell` und `Shell=/brent/spar`.

Wird in einem Makefile ein bestimmtes Macro referiert, so expandiert make dieses Macro, d.h. es verwendet den Macro-Wert. Zu Beginn des make-Laufs werden sämtliche Environment-Variablen als Macros geladen, wobei die Macro-Namen mit den entsprechenden Environment-Variablen-Namen übereinstimmen und die Macro-Werte aus den Werten der Environment-Variablen kopiert werden. Bestimmte Macros sind make-intern vordefiniert (built-in), z.B. die Macros, mit denen im Makefile Compiler angesprochen werden. Diese Macros können wie die aus dem Environment erhaltenen Macros auch durch Makefile-interne Direktiven überschrieben werden.

Macros werden an Stelle von Stringkonstanten verwendet, wenn solche Konstanten mehrfach im Makefile auftauchen. Auf diese Weise muß man bei Ersetzen solcher Konstanten durch neue Werte nur an einer einzigen Stelle (nämlich bei der Vereinbarung des Macros) Änderungen vornehmen. Macros nehmen daher im Makefile die Rolle ein, welche in Fortranquelltexten den globalen Parametervariablen zukommen bzw. in C den in Headerfiles mittels `#define` vereinbarten Präprozessormacros.

Goal Komplexe Makefiles können viele (Target,Dependency)--Beziehungen enthalten. Theoretisch kann man make aufrufen, um irgend einen ganz speziellen Knoten im Baum aufzubauen. In der Praxis interessieren einen aber die in den „mittleren“ Tiefen des Baums angesiedelten Äste überhaupt nicht, sondern man wird beim make-Aufruf nur die Endziele angeben. Solche „Haupt“-Targets bezeichnet man als Goals. Merke: Jedes Goal ist ein Target, aber nicht jedes Target ein Goal !

Optionen zu 'make'

Aufruf: `[g]make [options] [goals]`

Option	gnu-make-Langform	Anm.*
-d	--debug	agi
-e	--environment-overrides	aghiu
-f ...	--file=... --makefile=...	aghiu
-i	--ignore-errors	aghiu
-n	--just-print --dry-run	aghiu
-p	--print-data-base	aghiu
-r	--no-built-in-rules	aghiu
-s	--silent --quiet	aghiu
-t	--touch	aghiu
-v	--version	g
-w	--print-directory	g
	--warn-undefined-variable	g

*Anmerkungen zur Verfügbarkeit bei gmake bzw. den hersteller-eigenen make-Versionen: a=AIX, g=gmake, h=HP-UX, i=IRIX, u=UXP/M

Die oberste Zeile der Folie zeigt, wie make aufgerufen wird (die eckigen Klammern umgrenzen optionale Teile des Aufrufs). Die herstellereigene make-Version wird durch `make` aufgerufen, die GNU-Version durch `gmake`. Für make gibt es eine Reihe von Optionen, welche durch ein Minuszeichen eingeleitet werden; sie werden in der Tabelle auf der Folie aufgezählt und weiter unten erläutert. In der Regel wird man make jedoch ohne Optionen aufrufen. Auf die (meist leere) Optionsliste folgt die Liste der Goals, die beim aktuellen make-Lauf erstellt werden sollen. Auch diese Liste darf leer sein, in diesem Fall wird das oberste Goal des Makefiles ausgeführt. Somit lautet der einfachste (und gleichzeitig verbreitetste) make-Aufruf: `make`.

Soll der aktuelle make-Lauf die Targets `laurel` und `hardy` erstellen, so lautet der Aufruf `make laurel hardy`.

Bei den Optionen der proprietären makes folgt auf das einleitende Minuszeichen meist nur noch ein einziger Buchstabe. Die GNU-Version versteht neben diesen Kurzformen auch noch äquivalente Langformen, welche nicht durch ein einziges Minuszeichen eingeleitet werden, sondern durch zwei Minuszeichen.

`-d` druckt Debug-Informationen. Hierdurch wird make dazu veranlaßt, während der Aktualisierung des Baums zusätzliche (der genaue Ausgabeumfang hängt von der verwendeten make-Version ab) Informationen auszugeben (Angaben über aktualisierte Files, Datei-Änderungszeitpunkte, angewendete Regeln usw.). Diese Option ist gelegentlich ganz nützlich, um herauszubekommen, warum dieses verdammte make nicht das tut, was man eigentlich von ihm erwartet. Allerdings sollte man bei Verwendung dieser Option die make-Standard-Ausgabe in ein File umleiten, da die Ausgabe recht groß werden kann:
```
make -d laurel hardy > make.lst
```

```
% RCS-Kennung der LaTeX-Quelle zu diesem Dokumentationsteil:
% $Source: /home/ws/rz65/Kurse/Make/LaTeX/RCS/mk_optionen.tex,v $
% $Revision: 1.10 $ $Date: 1996/01/24 20:12:26 $ $Author: rz65 $
```

-e Zu Beginn des make-Laufs werden alle Environment-Variablen als gleichnamige make-Variablen übernommen. Makefile-interne Zuweisungen überschreiben normalerweise diese Startwerte der make-Variablen; diese Veränderbarkeit kann jedoch durch die Option -e abgeschaltet werden, d.h. damit werden die ursprünglichen Werte eingefroren.

-f Mit dieser Option kann man eine Alternative zu den Standard-Makefilenamen `Makefile` und `makefile` vereinbaren. Beispiel: `make -f mf_sonderlauf`

-i ignore: Normalerweise wird der make-Vorgang abgebrochen, wenn bei einem von make ausgeführten Kommando ein Fehler auftritt. Dieser Abbruch ist in den meisten Fällen auch sinnvoll, da ein fehlerhaft aufgebauter Knoten in der Regel zu einer ganzen Reihe weiterer vermurxter Knoten in Richtung Wurzel des Baums führt. Die -i-Option verhindert solche Abbrüche, d.h. make wurstelt nach Fehlern unbeeindruckt weiter. Hinweis: Die Option wirkt so, als stünde vor allen Commands ein -.

-n Die Aktualisierungskommando werden gar nicht durchgeführt, sondern nur auf der Standardausgabe angezeigt („was wäre, wenn. . . “). Sie können diese Option verwenden, wenn Sie größere Änderungen am Makefile vorgenommen haben und sich nicht sicher sind, ob die Abhängigkeiten richtig eingetragen wurden.

-p Diese Option läßt make Macrovariablen und eingebaute Regeln ausdrucken. Wie schon in einer Vorgängerfolie erwähnt wurde, kann man diese Liste auch ohne echtes Makefile erhalten, in dem man `/dev/null` mittels der Option -f als „Makefile“ angibt:

```
make -p -f /dev/null
```


Optionen zu 'make' (Forts.)

-r Alle vordefinierten Regeln werden verworfen. Man muß in diesem Falle alle Regeln explizit gestalten oder eigene implizite Regeln vereinbaren. Man kann diese Option verwenden, wenn man Files mit einer bestimmten Extension verarbeiten muß, die einen ganz anderen Dateityp kennzeichnet, als dies make in seinen built-in rules vorsieht.

-s Die auszuführenden Kommandos werden nicht ausgedruckt. Hinweis: Dieses Kommando wirkt so, als stünde vor allen Commands des Makefiles ein @.

-t Die Aktualisierungskommandos werden gar nicht durchgeführt, stattdessen wird das last modification date des Targets durch `touch` aktualisiert. Dies kann dann sinnvoll sein, wenn ein von vielen Files eingebautes Headerfile nur kosmetischen, aber keinen inhaltlichen Änderungen unterzogen wurde (z.B. wenn nur Kommentare geändert wurden) und man sich die daraus normalerweise folgenden zeitaufwendigen Übersetzungen ersparen will.

-v gibt die aktuelle Version des make-Programms aus.

-w Gibt an, in welchem Directory make gerade arbeitet. Dies ist bei rekursiven make-Aufrufen mit Directory-Wechseln zu Debug-Zwecken sinnvoll.

--warn-undefined-variables Sinnvoll, um Tippfehler bei Macrovariablennamen aufzufinden.

make-Syntax 1: Grundsätzliches

- **Makefile: Textfile, zeilenorientiert**
- **Fortsetzungszeilen:**

```
TEX_SOURCES=mk.tex␣mk_gliederung.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣mk_option.tex␣mk_syntax.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣mk_problembeschreibung.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣mk_macros.tex
```

- **Kommentare:**

```
#␣dies␣ist␣ein␣Kommentar
```

- **Include weiterer Makefiles:**

```
include␣Makefile_common_macro_defs
```

- **keine bedingten Anweisungen**
  keine Funktionen (Macros mit Argum.)
  aber: gnu-make-Erweiterungen dazu

zeilenorientiert Das Makefile wird aus einzelnen make-Statements aufgebaut, die durch das Zeilenende von einander getrennt werden.

Fortsetzungszeilen weisen in der letzten Spalte (unmittelbar vor dem Zeilenvorschub) einen Backslash \ auf (vgl. Fortsetzungszeilenankündigung bei diversen Shells). Häufiger Fehler: Zwischen dem Backslash und dem Zeilenende befinden sich noch Blanks oder Tabulatoren → make erkennt die Fortsetzungszeile nicht.

richtig:
```
OBJS=haupt.o␣\
␣␣␣␣␣up.o
```

falsch:
```
OBJS=haupt.o␣\␣
␣␣␣␣␣up.o
```

Solche Fehler kann man z.B. durch Ausgabe der Hex- oder Oktaldarstellung des Makefile-Inhalts finden.

Kommentare Kommentarzeilen werden durch ein Doppelkreuz # eingeleitet. make übergeht Kommentar- und Leerzeilen, jedoch sollten andere Anweisungen, welche sich über mehrere Zeilen (d.h. mit Fortsetzungszeilen) erstrecken, nicht durch Kommentar- oder Leerzeilen unterbrochen werden.

Include In manchen Fällen kann es sinnvoll sein, Makefiles aufzuteilen und bestimmte Teile (meist globale Pfad-Vereinbarungen o.ä.) in einem getrennte File abzulegen. Diese Teile werden von make dynamisch beim Lesen des Haupt-Makefiles eingebaut.

Bedingte Anweisungen Die GNU-make-Version erlaubt auch bedingte Anweisungen. Mit diesen kann man z.B. betriebssystemabhängig unterschiedliche Übersetzungsvorgänge ablaufen lassen.

```
% RCS-Kennung der LaTeX-Quelle zu diesem Dokumentationsteil:
% $Source: /home/ws/rz65/Kurse/Make/LaTeX/RCS/mk_syntax.tex,v $
% $Revision: 1.29 $ $Date: 1996/01/18 07:49:01 $ $Author: rz65 $
```

- **Macro-Vereinbarungen:**
  `OBJS=haupt.o␣up.o`
- **Macro-Referenzen:**
  `RMLIST=made␣$(OBJS)`
  `RMLIST=made␣${OBJS}`
- **keine Rekursion bei Macro-Zuweisungen! d.h. folgendes ist falsch:**
  `OBJS=haupt.o`
  `OBJS=$(OBJS)␣up.o`
  aber: gmake-Erweiterungen dazu
- **typische Macros (built-in/Environment):**

Name	Wert	Bedeutung
FC	f77	Fortran-Compiler
FFLAGS	-O	$(FC)-Optionen
CC	cc	C-Compiler
CFLAGS	-O	$(CC)-Optionen
LD	ld	Linker
LFLAGS	—	$(LD)-Optionen
HOME	var.	Home-Verzeichnis

Macro-Vereinbarungen Der Macro-Name steht links vom Gleichheitszeichen, rechts davon befindet sich der Wert, welcher dem Macro zugewiesen wird. Der Macro-Name darf Klein- und Großbuchstaben, Ziffern und den Unterstrich _ enthalten. Anm.: Meistens werden Macronamen großgeschrieben.

Macro-Referenzen Wird ein Macro ausgewertet (z.B. im rechten Teil der Vereinbarung eines weiteren Macros), so ist der Macroname in runden oder geschweiften Klammern (d.h. die beiden Vereinbarungen für `RMLIST` sind äquivalent) anzugeben und davor ein Dollarzeichen zu setzen. Ausnahme: Ist der Macroname nur ein Zeichen lang, so können die Klammern entfallen.

Rekursionen dürfen in normalen Macro-Zuweisungen nicht auftreten. Es gibt jedoch gmake-spezifische Erweiterungen, die Rekursion erlauben (z.B. die Ersetzung des Standard-Zuweisungsoperators = durch :=).

typische Macros — eine Auswahl:

`FC` , der Fortran-Compiler, heißt bei den meisten make-Versionen `f77`. Aber: Das muß nicht immer die beste Lösung sein. Beispiel (HP-UX): Dort gibt es eine Alternative `fort77`, die aufgrund einer abweichenden Liste von Compileroptionen dem built-in-Wert `f77` vorzuziehen ist, d.h. dort sollte man stets folgende Vereinbarung einbauen: `FC=fort77`

`FFLAGS` Optionen, mit denen der Fortran-Compiler (siehe Macro `$(FC)`) aufgerufen wird. `-O` besagt, daß der Compiler optimierten Code erzeugen soll.

`CC` , der C-Compiler, heißt bei den meisten make-Versionen `cc`. Auch hier kann eine Alternative sinnvoller sein: der ANSI-C-Compiler `c89`.

`CFLAGS` Optionen, mit denen der C-Compiler (siehe Macro `$(CC)`) aufgerufen wird. Auch hier besagt `-O`, daß der Compiler optimierten Code erzeugen soll.

`LD` , der Binder , heißt bei den meisten make-Versionen `ld`. Diesen Wert wird man fast nie ändern müssen.

`LFLAGS` Optionen, mit denen der Linker (siehe Macro `$(LD)`) aufgerufen wird. Die Voreinstellung (Leerstring) besagt, daß gar keine Default-Optionen vorliegen.

`HOME` enthält den Namen des Home-Verzeichnisses. Dessen Name hängt natürlich von der aktuellen Userid ab. Der Wert wird zu Beginn des make-Laufs aus dem Environment geladen.

make-Syntax 3: Explicit Rules

- **Rule – Format 1:**

  <target>:<dep1>␣<dep2>...;<cmd1>
  TAB<cmd2>
  TAB<cmd3>

- **Rule – Format 2:**

  <target>:<dep1>␣<dep2>...
  TAB<cmd1>
  TAB<cmd2>
  TAB<cmd3>

- **Mehrfache Targets:**

  <target1>␣<target2>:<dep1>␣<dep2>...

- **TAB: Tabulator-Zeichen:**
  Ordnungszahl: dez=9,hex=0x09,oct=011

make-Syntax 3: Explicit Rules

R *Die Folie mit dem Abhängigkeitsbaum muß nun entfernt werden, aber so, daß später wieder leicht auf sie zurückgegriffen werden kann.*

R *Linker Projektor: Syntax-Regeln, rechter Projektor: Beispiele*

Rule – Format 1 Das Target wird von der Dependency-Liste durch einen Doppelpunkt abgetrennt, die Dependency-Liste von dem (ersten) bei der Aktualisierung auszuführenden Kommando durch einen Strichpunkt. Die Dependency-Liste kann auch leer sein, in diesem Fall wird das Target immer von make aufgebaut. Falls mehr als ein Kommando zur Aktualisierung ausgeführt werden muß, werden die restlichen Kommandos in den folgenden Zeilen aufgeführt, wobei jede dieser Zeilen durch einen Tabulator TAB in Spalte 1 eingeleitet werden muß. Falls überhaupt keine Commands zur Aktualisierung angegeben werden, sucht make nach impliziten Regeln für den Aufbau des Targets. Man beachte die Asymmetrie der Anordnung zwischen dem ersten Kommando und den restlichen Kommandos ! Dies wirkt häufig unübersichtlich, außerdem rutscht das erste Kommando bei diesem Format oft weit nach rechts (und damit aus dem Scrollbereich des Editorfensters), so daß man besser dem Format 2 den Vorzug gibt.

Rule – Format 2 Im Gegensatz zum Format 1 wird hier das erste zur Aktualisierung auszuführende Kommando nicht in der Target-Zeile angegeben, sondern erst in der nächsten Zeile; daher entfällt der trennende Strichpunkt in der Target-Zeile. Auch in diesem Fall muß jede Kommando-Zeile durch ein TAB eingeleitet werden.

Mehrfache Targets Diese Syntaxform kann gewählt werden, um auszudrücken, daß mehrere Files vom gleichen Satz Dependencies abhängen. Ein Target kann sogar in verschiedenen Rules auftreten, make vereinigt in diesem Fall die Abhängigkeiten des entsprechenden Targets.

TAB Fehlende Tabulatorzeichen am Anfang von Commands sind die häufigste Ursache für Syntaxfehler in Makefiles. Beachte: TAB muß in Spalte 1 stehen, d.h. auch ␣TAB ist ungültig ! Zur Prüfung, ob in Spalte 1 ein TAB steht, fahre man mit dem Cursor an den linken Zeilenrand und versuche anschließend, den Cursor um ein Zeichen nach rechts zu setzen (z.B. durch die →Taste). Falls dies funktioniert, stand in Spalte 1 kein TAB, sondern ein normales Blank; andernfalls wäre der Cursor wesentlich weiter nach rechts gesprungen (häufig zu Spalte 8).

make-Syntax 3: Beispiele

- **Rule – Format 1:**

```
dok.ps:␣dok.tex␣t1.tex␣t2.tex;␣latex␣dok.tex
TAB latex␣dok.tex
TAB dvips␣dok.dvi␣-o␣dok.ps
```

- **Rule – Format 2:**

```
dok.ps:␣dok.tex␣t1.tex␣t2.tex
TAB latex␣dok.tex
TAB latex␣dok.tex
TAB dvips␣dok.dvi␣-o␣dok.ps
```

- **Mehrfache Targets:**

```
kop.tga␣kop.tif␣kop.ps:␣orig.gif
TAB giftopnm␣␣orig.gif␣>␣orig.ppm
TAB ppmtotga␣␣orig.ppm␣>␣kop.tga
TAB pnmtotiff␣orig.ppm␣>␣kop.tif
TAB pnmtops␣␣␣orig.ppm␣>␣kop.ps
TAB rm orig.ppm
```

- **WWW-Demo-File:** `demo2.tar.gz`

Rule – Format 1 Bei diesem Beispiel soll eine PostScript-Ausgabedatei `dok.ps` aus einem Satz von LaTeX-Files erstellt werden (auch dies ein Beispiel dafür, daß sich die make-Anwendung durchaus nicht auf den Bereich der Programmiersprachen im engeren Sinne beschränkt). Das PostScript-File wird aus der LaTeX-Hauptdatei `dok.tex` sowie den beiden Unterdateien `t1.tex` und `t2.tex` abgeleitet. Die Erstellung des Targets umfaßt drei Commands: Zunächst werden die Quellen zweimal (damit die LaTeX-internen Kreuzreferenzen stimmen) mit dem Programm `latex` formatiert, anschließend wird das von `latex` automatisch angelegte DVI-File `dok.dvi` durch den PostScript-DVI-Treiber `dvips` zur Erstellung des Targets `dok.ps` ausgewertet.

Rule – Format 2 Dieses Beispiel stimmt inhaltlich mit dem Beispiel zu Format 1 überein, es wurde lediglich die alternative (und übersichtlichere) Syntaxform gewählt.

Mehrfache Targets Bei diesem Beispiel geht es darum, aus einem Rasterimage `orig.gif`, welches im GIF-Format vorliegt, drei Kopien in unterschiedlichen Formaten (Targa, TIFF und PostScript) zu erstellen. Hierzu wird die GIF-Datei zunächsteinmal in das Austauschformat ppm (portable pix map → PIXUTIL-Dokumentation des RZ, beim Studentenwerk erhältlich) konvertiert und als Hilfsdatei `orig.ppm` abgelegt. Anschließend wird diese Hilfsdatei mittels der Formatkonversionsprogramme `ppmtotga` (Wandlung ins Targa-Format), `pnmtotiff` (Wandlung ins TIFF-Format) und `ppmtops` (Wandlung ins PostScript-Format) in die entsprechenden Targetfiles umgewandelt. Im letzten Schritt wird die Hilfsdatei `orig.ppm` wieder gelöscht.

Beispiel dazu Ein komprimiertes tar-File mit den Quellen und dem Makefile zu diesem Beispiel finden Sie auf der RZ-WWW-Seite zu make (siehe Folie „Dokumentationen zu make"), die Datei heißt `demo2.tar.gz`.

make-Syntax 4: Implicit Rules

- **Suffix-Vereinbarung (→ Implicit Rule, oft unnötig, da make viele built-in suffixes kennt):**
  `.SUFFIXES:␣.<e1>␣.<e2>␣.<e3>`
- **Implicit Rule:**
  ```
  .<e1>.<e2>:
  TAB <cmd1>
  TAB <cmd2>
  TAB <cmd3>
  .<e3>:
  TAB <cmd4>
  TAB <cmd5>
  TAB <cmd6>
  ```
- `<cmd1>` **usw. enthalten spezielle Macrovariablen** `$*`, `$@`, **. . . , da bei impliziten Regeln keine konkreten Target- bzw. Dependency-Namen angegeben werden, sondern Platzhalter.**
- **make versucht, den Baum durch bekannte built-in rules zu verlängern.**
  → Vereinbarungsaufwand sinkt
  → Vorsicht: Mehrdeutigkeiten !

R *voriges Folienpaar nun entfernen*

R *Linker Projektor: Syntax-Regeln, rechter Projektor: Beispiele*

Suffix-Vereinbarung Hierbei ist darauf zu achten, daß der Punkt am Anfang nicht vergessen wird und daß die Großschreibung eingehalten wird. Andernfalls erkennt make nämlich das Schlüsselwort `.SUFFIXES` nicht. Im „Dependency“-Teil wird eine Liste von Extensions (einschließlich des einleitenden Punkts) angegeben, die Listenelemente werden dabei durch Blanks von einander getrennt. Durch diese `.SUFFIXES`-Angabe wird make gegenüber mitgeteilt, mit welchen Dateitypen es bei impliziten Regeln „rechnen“ muß. Da make schon über eine umfangreiche Menge vordefinierter (built-in) Suffixe verfügt, ist es nur selten notwendig, diese eingebaute Liste durch eine explizite `.SUFFIXES`-Angabe zu erweitern: Die vordefinierte Liste umfaßt die Extensions zu den Quellfiles der üblichen Programmiersprachen, Object- und Achive-Files u.v.a.m.; für „exotischere“ Anwendungsfälle (z.B. für Bildformatkonversionen) muß man aber die Extensions der entsprechenden Dateien explizit angeben (s.Beispiel).

Implicit Rule Durch diese Regel wird vereinbart, wie Dateien eines bestimmten Typs (identifiziert durch ihre Extension) in Dateien eines anderen Typs (identifiziert durch eine entsprechende <u>zweite</u> Extension oder durch das <u>Fehlen</u> einer Extension) abgebildet werden. Implicit Rules werden eingeleitet durch eine Zeile, welche eines der beiden folgenden Formate aufweist:

`.<e1>.<e2>:`	double dot rule
`.<e3>:`	single dot rule

Die double-dot-Format `.<e1>.<e2>:` leitet dabei eine Regel für die Umwandlung von Dateien mit der Extension `.<e1>` in Dateien mit der Extension `.<e2>` ein (man beachte: Extension des Targets ist `.<e2>`, Extension der Dependency ist `.<e1>`):

`xxxxxx.<e1>` → `xxxxxx.<e2>`

Die single-dot-Format `.<e3:>` leitet dabei eine Regel für die Umwandlung von Dateien mit der Extension `.<e3>` in Dateien <u>ohne</u> ein (man beachte: Target ohne Extension, Extension der Dependency ist `.<e3>`):

`xxxxxx.<e3>` → `xxxxxx`

single dot rules werden häufig zur Erstellung von ausführbaren Programmen (Executables) verwendet, da diese in UNIX normalerweise keine Extension aufweisen (im Gegensatz zu DOS (Extension `.exe`, `.com`)).

Command-Teil der Implicit Rule Da die Implicit Rule für alle Datei-Paare einer bestimmten (Target,Dependeny)-Typ-Kombination gelten soll, kann man in deren Command-Teil natürlich keine konkreten (d.h. zu tatsächlichen Dateien gehörende) Namen erwarten; vielmehr findet man hier Platzhalter-Variablen, deren Namen nur aus und dem Macro-Indikator `$` und einem ganz bestimmten Sonderzeichen (z.B. `<`, `@`) bestehen, diese speziellen Variablen vertreten z.B. den Target-Namen oder den Namen des Targets nach Abstreifen der Extension. Solche Macros heißen bei make „Automatic Macros“ (siehe unten), da ihre Werte von make automatisch an die aktuellen Argumente angepaßt werden.

Automatische Verlängerung des Baums durch built-in rules Built-in rules legen nicht nur fest, wie ein Target (z.B. `xxx.o`) aus einer schon bestehenden Datei eines bestimmten Typs (z.B. `xxx.f`) abgeleitet wird, sondern auch, ob eine bestimmte Datei womöglich selbst als Target einer impliziten Regel in Frage kommt, deren Dependencies gar nicht explizit im Makefile definiert werden, sondern automatisch von make aufgespürt werden. Dies sind zwei ganz verschiedene Paar Stiefel: Im ersten Fall geht es für make darum, eine passende Aktion für den Übergang von einer Datei eines bestimmten Typs in Richtung Baumwurzel zu einer Datei eines anderen Typs zu finden; im zweiten Fall geht es darum, den Baum selbsttätig in in die andere Richtung (d.h. nach unten, weg von der Wurzel) zu verlängern. Dieser Unterschied soll an folgenden beiden Beispielen erläutert werden:

1. Falls in einer expliziten Regel festgelegt ist, daß `up.o` von `up.f` abhängt, aber in dieser Regel kein Kommando zur tatsächlichen Ableitung gegeben wurde, sucht make nach einer impliziten Regel dafür, d.h. nach einer Regel `.f.o`, und wendet sie auf die konkrete (Target,Dependency)-Kombination (sprich: `(up.o,up.f)` an.
2. Falls für eine Datei keine Dependencies angegeben sind, d.h. diese Datei ein Blatt des Baums darstellt, durchforstet make seine impliziten Regeln nach solchen, die zur Ableitung des Blatts aus anderen Dateien des aktuellen Directories angewendet werden können. Beispiel: Ist `up.o` als Blatt des Baums gegeben, so sucht make (da es eine Reihe impliziter Regeln zur Ableitung von Object-Files aus Fortran-, C- oder Pascal-Quellen kennt) nach Files wie `up.f`, `up.c` oder `up.p`. Falls eine solche Datei existiert und sie neuer ist als `up.o` (oder falls `up.o` noch gar nicht existiert), wird der entsprechende Compiler zur Ableitung aufgerufen.

Pferdefüße Wie man sieht, muß man mit der Aussage „Datei `xxx` ist ein Blatt des Baums.“ mit Vorsicht begegnen, da das Blatt sich u.U. als ganz schön dicker Ast herausstellt, aus dem noch einiges sprießt. Diese automatische Verlängerung des Baums kann zu sehr üblen Effekten führen, wenn sie aufgrund von Mehrdeutigkeiten bei den Dependencies zu unerwarteten Ableitungen führt. Beispiel: Zu einem Object-Files `up.o` soll keine explizite Abhängigkeit im Makefile aufgeführt sein, d.h. es soll von make automatisch aus dem entsprechenden Quellfile abgeleitet werden. Was passiert aber nun, wenn es im aktuellen Verzeichnis sowohl eine Datei `up.f` (d.h. eine Fortran-Datei, die vom Namen her „passen“ würde) als auch eine Datei `up.c` (eine ebenso „passende“ C-Datei) gibt ? Glaubt man Murphy (und Murphy hat bekanntlich immer recht), so wird natürlich die unpassende Alternative gewählt. Zur Verteidigung von make muß jedoch gesagt werden, daß Ihnen nicht zu helfen ist, wenn Sie in einem Directory solcherlei Mehrdeutigkeiten zulassen.

- **Suffix-Vereinbarung:**

```
.SUFFIXES:␣.gif␣.tif
```

- **Implicit Rule:**

```
.gif.tif:
TAB giftopnm␣$<␣|␣pnmtotiff␣>␣$@
.f.o:
TAB $(FC)␣-c␣$<
.f:
TAB $(FC)␣$<␣-o␣$@
```

- **built-in rules ausdrucken:**

```
gmake -p -f /dev/null
```

- **WWW-Demo-File:** `demo3.tar.gz`

Suffix-Vereinbarung Durch die angegebene Vereinbarung wird die Liste der dem Programm make bekannten Extensions um die beiden Extensions `.gif` und `.tif` erweitert (diese beiden Extensions kennzeichnen Rasterimages im GIF- bzw. TIFF-Rasterformat).

Implicit Rules Die erste implizite Regel legt fest, wie Dateien mit der Extension `.gif` in Dateien mit der Extension `.tif` konvertiert werden. Die Formulierung des entsprechenden Kommandos besagt, daß hierzu das Dependency-File (welches geändert wurde und durch das Automatic Macro `$<`(s.u.) vertreten wird) von dem Konvertierungsprogramm `giftopnm` (siehe PIXUTIL-Beschreibung des RZ) gelesen wird. Die Ausgabe von `giftopnm` wird über eine Pipe in ein zweites Konvertierungsprogramm `pnmtotiff` geleitet, welches das gewünschte Target im TIFF-Format (vertreten durch das Automatic Macro `$@`) erzeugt.

Die zweite implizite Regel gibt an, wie Fortran-Programmeinheiten (Extension `.f`) in Object-Files (Extension `.o`) überführt werden sollen. Hierzu soll nämlich der Fortran-Compiler (dessen Name üblicherweise durch das Macro mit dem Namen `FC` gegeben ist) mit der Option `-c` (d.h. es soll nur übersetzt, aber nicht gebunden werden) aufgerufen werden. `$<` vertritt die Dependency (d.h. die Fortran-Programm-Einheit). Ein weiteres Automatic Macro für das Target ist hier übrigens nicht notwendig, da der Fortran-Compiler selbsttätig den Object-File-Namen aus dem Quell-Namen ableitet:

`xxxxxx.f` $\longrightarrow$ `xxxxxx.o`

Bei der letzten impliziten Regel finden wir eine single dot rule: Sie legt fest, wie Fortran-Programmeinheiten (Extension `.f`) in Files ohne Extensions umgewandelt werden. Solche Files sind der üblichen Namenskonvention entsprechend ausführbare Programme. Die Regel besagt, daß hierzu der Compiler mit der Dependency (d.h. dem Fortran-Programm) als erstem Argument (vertreten durch `$<`) aufgerufen werden soll, wobei die `-o`-Option besagt, daß das aufgebaute Executable den Namen des Targets (vertreten durch `$@`) tragen soll – Sie erinnern sich: ohne diese Option `-o` würde das Executable auf den Namen `a.out` getauft, und das wäre ja bäbä !

Hinweis Bei der `.SUFFIXES`-Anweisung wurden nur die „exotischen“ Extensions `.gif` und `.tif` aufgeführt, nicht jedoch in den weiter unten genannten Beispielen verwendeten Extensions `.f` und `.o`; Extensions wie `.f`, `.o` braucht man nämlich nicht explizit in der `.SUFFIXES`-Anweisung angeben, da sie in make built-in sind.

Beispiel dazu Ein komprimiertes tar-File mit den Quellen und dem Makefile zu diesem Beispiel finden Sie auf der RZ-WWW-Seite zu make (siehe Folie „Dokumentationen zu make“), die Datei heißt `demo3.tar.gz`.

Seite wurde absichtlich frei gehalten

make-Syntax 5: Automatic Variables

$@ aktuelles Target

$% Target-Membername, falls das Target Member einer archive-Datei ist, sonst Leerstring.

$< (erste) Dependency

$? Liste aller Dependencies, die neuer als das Target sind.

$^ Liste aller Dependencies ohne Mehrfachnennungen

$+ Liste aller Dependencies mit Mehrfachnennungen

$* Targetname ohne Extension

make-Syntax 5: Automatic Variables

R *voriges Folienpaar nun entfernen*

R *Linker Projektor: Automatic Variables, rechter Projektor: Beispiele*

`$@` In diesem Macro ist der Name des aktuellen Targets abgespeichert. Falls in einer Abhängigkeitsdefinition mit Mehrfach-Targets gearbeitet wurde, erfolgt die Ausführung des Aktualisierungsaktion für jedes Target, wobei bei jeder Einzelaktion in `$@` der „richtige“ Dateiname vorzufinden ist.

`$%` Dieses Automatic Macro kann nur bei der Erstellung von archive-Libraries (solche Libs werden später beschrieben) sinnvoll verwendet werden. Es enthält in diesen Fällen den Namen des als Target dienenden archive-Members ohne den vorangestellten archive-Library-Namensteil (dieser wird in `$@` geliefert). Ist das Target kein archive-Member, so liefert `$%` nur einen Leerstring.

`$<` Hierin wird der Name der ersten Dependency aus der Liste der Dependencies eines Targets geliefert. Die weiter hinten in der Liste liegenden Dependencies kann man mit diesem Automatic Macro nicht erreichen.

`$?` In diesem Automatic Macro liefert make die Liste aller Dependencies, die neuer als das Target sind; die einzelnen Listenelemente sind durch Leerzeichen getrennt. Anders ausgedrückt: `$?` benennt alle „Übeltäter“, welche die Aktualisierung verursachten.

`$^` In diesem Automatic Macro liefert make die Liste aller Dependencies, gleichgültig, ob sie nun neuer oder älter als das Target sind. Falls eine Dependency mehrfach in der ursprünglichen Dependency-Liste vorkam, wird jedoch nur das erste Auftreten in `$^` berücksichtigt, d.h. die Liste enthält keine Mehrfachnennungen.

`$+` Ähnlich wie bei `$^` finden Sie hier die Liste aller Dependencies des aktuellen Targets, mit dem Unterschied, daß bei `$+` auch Mehrfachnennungen auftreten dürfen, d.h. die überzähligen Exemplare werden nicht verworfen. Dies findet bei Linkvorgängen Verwendung, wenn die Dependency-Liste aus Library-Namen besteht, da aufgrund der Hirnlosigkeit der Unix-Linker Libraries manchmal mehrfach in der Argumentliste des Linkers auftreten müssen; in solchen Fällen darf keine Reduktion der Mehrfachnennungen stattfinden.

`$*` make liefert in diesem Automatic Macro den Namen des aktuellen Targets ohne den Extension-Teil.

- **Phony Target: keine phys. Datei**

```
clean:
TAB rm␣-f␣*.o
```

- **typische Phony Targets (Konvention):**
  - all
  - clean
  - install
  - uninstall
- **explizite Phony-Vereinbarung bereinigt Namenskonflikte bei existierender gleichnamiger Datei:**

```
.PHONY:␣all␣clean␣install uninstall
```


Normalerweise verbirgt sich hinter einem Target auch eine Datei gleichen Namens, welche durch eine bestimmte Aktion erstellt bzw. aktualisiert wird. Es gibt jedoch auch Targets, bei denen gar keine gleichnamige Datei erstellt wird. Beispiel: Häufig findet man in Makefiles ein Target namens „`clean`“, mit dem man alle abgeleiteten Files löscht, um nur noch die ursprünglichen Quellfiles übrigzulassen, wie dies bei dem angegebenen Beispiel der Fall sein soll (Hinweis: Die `rm`-Option `-f` verhindert, daß eine Fehlermeldung ausgegeben wird, wenn gar kein Object-File mehr übrig ist, das man löschen könnte). Man beachte, daß zum einen keine Dependencies für `clean` angegeben werden, zum anderen aber auch keine Commands vorhanden sind, welche ein File namens `clean` erzeugen. Ein solches nicht mit einer tatsächlichen Datei verbundenes Target wird als „Phony Target“ bezeichnet. Phony Targets können zwar grundsätzlich beliebige Namen tragen, in der Praxis verwendet man jedoch in vielen Fällen Namen gemäß einer allgemein gepflegten Konvention:

`all` Durch dieses Target werden alle Executables des Pakets in der aktuellen Entwicklungsumgebung erstellt, es erfolgt jedoch noch eine Installation der erstellten Programme in der Produktionsumgebung. An den make-Lauf mit diesem Target schließen sich in der Regel Tests der erstellten Programme an. Erst wenn diese bestanden sind, sollte in die Produktionsumgebung installiert werden.

`clean` Die Auswahl dieses Target führt dazu, das alle abgeleiteten Files gelöscht werden, d.h. man erhält wieder den ursprünglichen Quellzustand vor dem Aufbau der Executables usw.; m.a.W. durch `clean` macht man reinen Tisch, was ja durch die Namensgebung auch nahegelegt wird.

`install` Durch dieses Target werden die erstellten Programme nach den (bestandenen) Tests von der Entwicklungsumgebung in die Produktionsumgebung kopiert. Neben den eigentlichen Kopiervorgängen sind hierzu auch Anpassungen an den Zugriffsrechten auf die installierten Programme notwendig; z.B. ist es häufig so, daß während der Entwicklung nur der Owner auf die Dateien zugreift, aber in der Produktionsphase auch ganz andere Leute auf die Dateien zugreifen sollen.

`uninstall` Mit diesem Target kann man die Produktionsumgebung wieder von den durch einen früheren Installationsvorgang in sie eingebauten Dateien des Pakets befreien. Dieses Target wird in der Regel sträflich vernachlässigt.

Bei der Definition der Phony Targets wurde beschrieben, daß make zu diesen Targets von sich aus gar keine gleichnamige Datei erstellt. Nun kann es aber vorkommen, daß z.B. bei dem Phony Target `clean` aus irgendwelchen Gründen im aktuellen Directory eine Datei namens „`clean`“ existiert. In diesem Fall würde make bei der `clean`-Regel feststellen, daß es kein File gibt, von dem `clean` abhängt und das neuer als `clean` ist (da die Dependency-Liste ja leer ist. Demzufolge würden die Commands zur „Aktualisierung“, d.h. das Löschen der Object-Files, nicht ausgeführt. Diese Konfliktsituation (Target, daß eigentlich phony sein sollte und eine „zufälligerweise“ koexistierende gleichnamige Datei, welche mit dem Paket aber gar nichts zu tun hat) kann jedoch bereinigt werden, in dem man das Phony Target als Dependeny eines zusätzlichen ganz speziellen Targets namens „`.PHONY`“ vereinbart. Man beachte, daß sich „`.PHONY`“ von normalen Targets durch den Punkt in Spalte 1 unterscheidet: Solche merkwürdigen Targetnamen sind für bestimmte Spezialzwecke reserviert (siehe nächste Folie).

Seite wurde absichtlich frei gehalten

- `.PHONY: <targ1> <targ2> ...`
- `.SUFFIXES: .<e1> .<e2> ...`
- `.DEFAULT:`
  TAB `<cmd1>`
  TAB `<cmd2>`
- `.PRECIOUS: <targ1> <targ2> ...`
- `.IGNORE: <targ1> <targ2> ...`
- `.SILENT: <targ1> <targ2> ...`

`.PHONY` Dieses spezielle Target wurde auf der letzten Folie schon angesprochen, es dient dazu, bestimmte Targets als Phony Targets kenntlich zu machen.

`.SUFFIXES` Auch dieses spezielle Target wurde schon angesprochen. In der Dependency-Liste dieses Targets werden bestimmte Extensions gegenüber make vereinbart.

`.DEFAULT` make versucht selbsttätig, den Abhängigkeitsbaum nach unten zu verlängern, d.h. es prüft, ob die angegebenen Blätter des Baum (diese dürfen ja nur als Dependencies auftreten, sonst wären sie keine Blätter) nicht doch noch aus irgendeiner impliziten Regel abgeleitet werden können. Falls es eine solche Ableitungsregel findet, dann erfolgt die angegebene Verlängerung des Baums. Falls jedoch keine solche Regel gefunden wird, dann kann man für solche Files den Abhängigkeitsbaum trotzdem noch verlängern, und zwar durch das spezielle Target `.DEFAULT`. Beispiel: Wenn man noch gar nicht alle Programmeinheiten des Paketes erstellt hat, aber schon das Makefile dafür, kann man mit

`.DEFAULT:` TAB `touch␣$@`

dafür sorgen, daß die fehlenden Dateien als leere Files (Dummies) angelegt werden.

`.PRECIOUS` Wenn ein make-Lauf abgebrochen wird (z.B. dadurch, daß man ihm mit `Cntl-C` ein entsprechendes Signal zusendet), kann es vorkommen, daß unvollständige Dateien erstellt werden. Beispiel: Der von make aufgerufene Compiler hat das zu erstellende Object-File bereits halb geschrieben, da ereilt ihn sein Schicksal — in einem solchen Fall liegt nach dem Abbruch ein bruchstückhaftes Object-File mit erneuertem last modification date vor. Würde man make erneut aufrufen, so würde es feststellen, daß das Object-File neueren Datums wäre und damit gar nicht neu erstellt werden müßte — ein scheußlicher Irrtum, denn spätestens bei der Erstellung des Executables würde das entstandene Wrack merkwürdige Fehler produzieren. Aus diesem Grunde löscht make normalerweise die während solcher Abbruchvorgänge entstandenen Files. Diese Vorsichtsmaßnahme kann unterbunden werden, indem man das entsprechende File als Dependency des speziellen Targets `.PRECIOUS` vereinbart.

`.IGNORE` Wie früher schon erwähnt wurde, wird der make-Lauf beendet, wenn bei der Ausführung einer Aktualisierung ein Fehler auftritt, es sei denn, make war mit der Option `-i` aufgerufen worden oder das Aktualisierungs-Kommando war im Makefile durch ein dem Kommando vorangehendes Minuszeichen speziell gekennzeichnet worden. Will man Fehler bei einigen Targets tolerieren, aber nicht bei allen, so verbietet sich die Option `-i`, da diese global auf alle Targets wirkt. Andererseits ist es auch mühsam, alle Commands zu den Targets, bei denen man Fehler tolerieren möchte, mit dem Minuszeichen zu versehen. Hier hilft das Spezialtarget `.IGNORE`: make ignoriert bei allen Targets Fehler, welche als Dependencies dieses speziellen Targets vereinbart werden.

`.SILENT` Mit diesem Spezialtarget hat es eine ähnliche Bewandtnis wie mit `.IGNORE`, jedoch bezieht sich `.SILENT` nicht auf den Abbruch bei Fehlern, sondern auf die Ausgabe des Aktualisierungskommandotexts. Auch hier gibt es eine Möglichkeit, das normale Verhalten global abzuschalten (make-Option `-s`) bzw. lokal abzuschalten (Voranschalten eines `@` vor den Kommandotext). Als Mittelding zwischen diesen beiden Extrema fungiert das Spezialtarget `.SILENT`: Die Ausgabe des Kommandotexts wird für all diejenigen Targets unterbunden, welche in der Liste der Dependencies zu `.SILENT` vereinbart wurden.

Seite wurde absichtlich frei gehalten

Ein Beispiel-Makefile

Makefile zu dem schon früher erwähnten Programm:

```
SHELL=/bin/sh
FC=fort77

.PHONY:␣all␣clean

OBJS=haupt.o␣up.o

all:␣made

made:␣$(OBJS)
TAB $(FC)␣$(OBJS)␣-o␣made

$(OBJS):␣glob.h

clean:
TAB rm␣-f␣$(OBJS)␣made
```

siehe auch WWW-Demo-File: `demo1.tar.gz`

R *voriges Folienpaar nun entfernen*

R *Linker Projektor: Folie mit dem Abhängigkeitsbaum wieder auflegen, rechter Projektor: aktuelle Folie*

`SHELL=/bin/sh` Diese Vereinbarung legt fest, daß die von make ermittelten Aktualisierungskommandos über die Standard-Shell ausgeführt werden. Dies ist eine Konvention.

`FC=fort77` Hiermit wird vereinbart, daß der Fortran-Compiler `fort77` verwendet werden soll. Diese Vereinbarung überschreibt den Defaultwert `f77`, da dieser Compiler z.B. bei HP-UX eine etwas andere Optionsliste aufweist. Man beachte, daß von dieser Umdefinition zahlreiche built-in rules betroffen sind.

`.PHONY:␣all␣clean` Hierdurch werden die Targets `all` und `clean` als Phony Targets vereinbart.

`OBJS=haupt.o␣up.o` Diese Macrovereinbarung legt den Satz Object-Files fest, welche bei der Erstellung auftreten.

`all:␣made` Das oberste Target (Default-Target) sollte der Konvention entsprechend immer `all` heißen, mit ihm sollte der Aufbau der Executables verbunden sein.

`made:␣$(OBJS)` Hierdurch wird vereinbart, daß `made` von den genannten beiden Object-Files abhängt. Die darunterstehende explizite Regel gibt an, wie `made` aus den Object-Files abgeleitet werden soll.

`$(OBJS):␣glob.h` Hier taucht ein Mehrfach-Target auf (`$(OBJS)` enthält ja zwei Files). Man beachte, daß die Abhängigkeit der Object-Files `haupt.o` und `up.o` von den dazugehörigen Quellen `haupt.f` und `up.f` gar nicht vereinbart werden muß, da diese Abhängigkeiten durch die make-interne Verlängerung des Baumes nach unten automatisch aufgefunden wird.

`clean:` Mit diesem Target werden alle abgeleiteten Files gelöscht, so daß nur noch die reinen Quellen übrigbleiben.

Beispiel-tar-File: Ein komprimiertes tar-File mit den Quellen und dem Makefile zu diesem Beispiel finden Sie auf der RZ-WWW-Seite zu make (siehe Folie „Dokumentationen zu make“), die Datei heißt `demo1.tar.gz`.

Protokoll eines Make-Laufs

Bei dem o.g. Beispiel sei `up.f` verändert worden. Auszug aus dem make-Listing bei Verwendung der Debug-Option `-d` für Trace. Aus Platzgründen wurden viele „uninteressante“ Zeilen übergangen.

```
GNU Make version 3.74, by Richard Stallman and Roland McGrath.
        :    :    :    :    :
Reading makefiles...
Reading makefile `Makefile'...
Updating makefiles....
 Considering target file `Makefile'.
        :    :    :    :    :
 No need to remake target `Makefile'.
Updating goal targets....
Considering target file `made'.
  Considering target file `haupt.o'.
   Looking for an implicit rule for `haupt.o'.
   Trying pattern rule with stem `haupt'.
        :    :    :    :    :
   Trying implicit dependency `haupt.f'.
   Found an implicit rule for `haupt.o'.
        :    :    :    :    :
   Finished dependencies of target file `haupt.o'.
   Dependency `haupt.f' is older than dependent `haupt.o'.
  No need to remake target `haupt.o'.
  Considering target file `up.o'.
   Looking for an implicit rule for `up.o'.
   Trying pattern rule with stem `up'.
        :    :    :    :    :
   Trying implicit dependency `up.f'.
   Found an implicit rule for `up.o'.
        :    :    :    :    :
   Finished dependencies of target file `up.o'.
   Dependency `up.f' is newer than dependent `up.o'.
  Must remake target `up.o'.
fort77   -c up.f
        :    :    :    :    :
  Successfully remade target file `up.o'.
 Finished dependencies of target file `made'.
 Dependency `haupt.o' is older than dependent `made'.
 Dependency `up.o' is newer than dependent `made'.
Must remake target `made'.
fort77 haupt.o up.o -o made
        :    :    :    :    :
Successfully remade target file `made'.
```


R *Linker Projektor: Folie mit dem Abhängigkeitsbaum liegen lassen, rechter Projektor: aktuelle Folie*

`Reading makefiles` Zu Beginn wird das Makefile gelesen und damit der Grundstock des Abhängigkeitsbaums erstellt.

`Updating makefiles` Makefiles können Änderungen an sich selbst vorsehen (dies ist eine GNU-Erweiterung). Die Aktualisierung des Makefiles hat Vorrang vor allen anderen Aktualisierungen.

`Updating goal targets` Nun kommen die eigentlichen Targets an die Reihe.

`Considering target file 'made'` Es wird untersucht, wovon `made` abhängt.

`Considering target file 'haupt.o'` Entsprechend den angegebenen Abhängigkeiten wird nun die Dependency `haupt.o` untersucht. Ohne daß explizit eine Abhängigkeit des Object-Files `haupt.o` von `haupt.f` angegeben worden ist, findet make trotzdem diese Abhängigkeit (automatische Verlängerung des Baumes nach unten), da es im aktuellen Directory das „passende“ File `haupt.f` vorfindet.

`Dependency 'haupt.f' is older than dependent 'haupt.o'`

`No need to remake target 'haupt.o'.` Das Object-File `haupt.o` muß demnach nicht aktualisiert werden.

`Considering target file 'up.o'` Nun wird der andere Zweig der angegebenen Abhängigkeiten von `made` durchlaufen. Auch hier erfolgt die automatische Verlängerung des Baums nach unten. d.h. auch hier wird automatisch das passende File `up.f` erfaßt.

`Dependency 'up.f' is newer than dependent 'up.o'`

`Must remake target 'up.o'.` Das Object-File `up.o` muß aktualisiert werden. Hierzu wird die built-in rule für die Ableitung des Object-Files aus dem Fortran-Quelltext ausgeführt.

`Successfully remade target file 'up.o'` Alles Paletti.

`Finished dependencies of target file 'made'.` Alle Dependencies von `made` sind auf dem neuesten Stand.

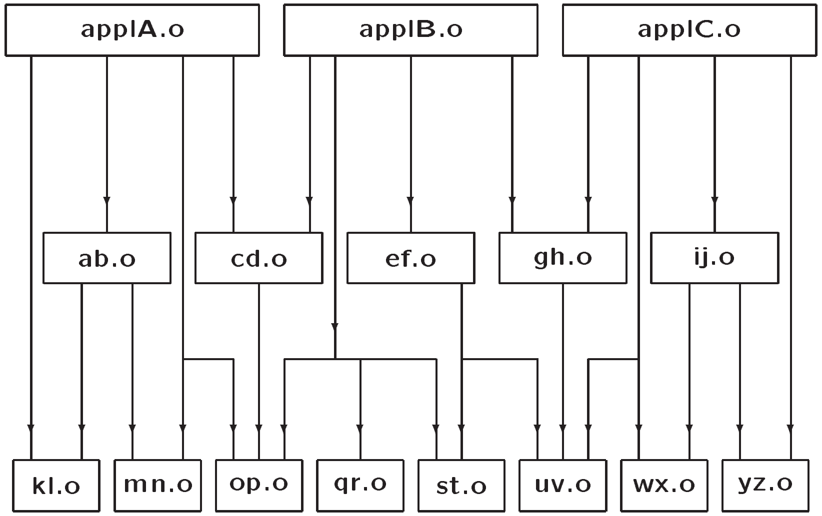
`Dependency 'haupt.o' is older than dependent 'made'.` Schön und gut. . .

`Dependency 'up.o' is newer than dependent 'made'.` Hoppla, das gibt Arbeit !

`Must remake target 'made'` Wer hätt's gedacht !

`Successfully remade target file 'made'.` Nun ist alles getan.

Aufrufstruktur:




```
# Macros fuer Object-Files:
AOBJS=applA.o ab.o cd.o kl.o mn.o op.o
BOBJS=applB.o cd.o ef.o gh.o op.o qr.o st.o uv.o
COBJS=applC.o gh.o ij.o uv.o wx.o yz.o

# Rules zum Binden:
applA: $(AOBJS)
TAB $(FC) $(AOBJS) -o applA
applB: $(BOBJS)
TAB $(FC) $(BOBJS) -o applB
applC: $(COBJS)
TAB $(FC) $(COBJS) -o applC
```


Sinn und Zweck von Libraries

Häufig werden bestimmte Unterprogramme (Beispiel auf der Folie: `cd`, `uv`) in ganz mehreren Anwendungen (Beispiel auf der Folie: `applA`, `applB`, `applC`) verwendet; wobei die Anwendungen miteinander gar nichts zu tun haben. Typische Beispiele für solche Routinen sind:

spezielle mathematische Funktionen: orthogonale Polynome (Tschebyschew, Laguerre, Legendre), Pseudozufallszahlengeneratoren, Lineare-Algebra-Routinen, numerische Integration & Differentiation, Lösung von Differentialgleichungen

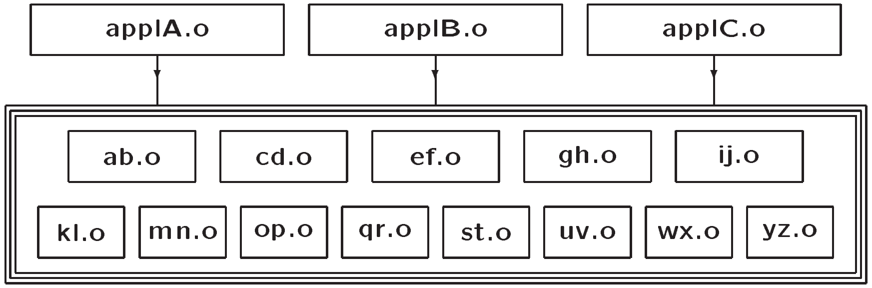
Datum und Uhrzeit: Berechnung von Wochen- und Feiertagen, Zeitzonenumrechnung, Fristenberechnungen

Graphik: Grundsoftware, Visualisierung von Feldern

Natürlich kann man sich die Quellen für solche Routinen leicht von einem FTP-Server beschaffen und physikalisch in seine Anwendung kopieren oder sie anderweitig referieren (z.B. über symbolische Links oder per Include). Der Nachteil solcher Lösungen besteht darin, daß auf diesem Weg das oft schon ausufernde Quelltext- und Aufrufgestrüpp der jeweiligen Applikation durch die Einbauten noch weiter aufgebläht wird. Ein andere, auch nicht viel bessere Alternative bestünde darin, sich einen Zoo vorübersetzter Object-Files zu halten, aus dem man die jeweils benötigten beim Binden der Anwendung als Argumente des Linkers angibt. Auch das ist nicht der Weisheit letzter Schluß: Man müßte sich genau merken, welche dieser „externen“ Routinen benötigt werden und welche von diesen wiederum sekundär usw. aufgerufen werden. Es ist daher eine Lösung vorzuziehen, welche den Aufwand für den Einbau dieser externen Routinen möglichst gering hält.

Hierfür faßt man die genannten Routinen in Unterprogrammbibliotheken zusammen, deren interne Struktur (Aufteilung der Problemlösung auf verschiedene Quelltextfiles, Verwendung interner Hilfsroutinen (Low-level-Routinen)) einem im Grunde genommen völlig wurscht ist — Hauptsache, die Routinen funktionieren so, wie man es von ihnen erwartet. Man muß dabei „lediglich“ darauf achten, daß die Aufrufformalitäten (korrekte Argumentliste, Voraussetzungen für den Aufruf, Seiteneffekte) genau eingehalten werden. Diese Unterprogrammbibliothek ist daher eine „black box“, bei der im Idealfall nur die Nahtstelle zur Applikation beachten werden muß.

```
% RCS-Kennung der LaTeX-Quelle zu diesem Dokumentationsteil:
% $Source: /home/ws/rz65/Kurse/Make/LaTeX/RCS/mk_library.tex,v $
% $Revision: 1.9 $ $Date: 1996/01/25 12:23:40 $ $Author: rz65 $
```



Bibliothek „maus“

```
#␣Macros␣zum Binden:
LIBFILE=libmaus.a
LFLAGS=-L.␣-lmaus

#␣Rules␣zum␣Binden:
applA:␣applA.o␣$(LIBFILE)
TAB $(FC)␣applA.o␣$(LFLAGS)␣-o␣applA
applB:␣applB.o␣$(LIBFILE)
TAB $(FC)␣applB.o␣$(LFLAGS)␣-o␣applB
applC:␣applC.o␣$(LIBFILE)
TAB $(FC)␣applC.o␣$(LFLAGS)␣-o␣applC
```

Nomenklatur:

umgangssprachl. Name: `maus`

Unix-Pfadname:
`libmaus.a`
`[libmaus.sl]`
`[libmaus.so]`

Linker-Option:
`-L.␣-lmaus`

Zugriff auf standardisierte Software: Das Rad muß nicht wieder neu erfunden werden, sondern man greift auf die Produkte anderer zu (die in vielen Fällen auch zu dem jeweiligen Gebiet wesentlich kompetenter sind).

Orthogonalität: Da der interne Aufbau der Bibliothek für die Anwendung uninteressant ist, kann man die Bibliothek leicht durch eine andere (neue Version, preiswerteres Konkurrenzprodukt) ersetzen, ohne daß dazu ein großer Umstellungsaufwand betrieben werden kann, es muß lediglich neu gebunden werden, die Bibliothek wird en bloc ausgetauscht.

Schutz geistigen Eigentums: Da Bibliotheken in der Regel in übersetzter Form verkauft werden, kann der Käufer oder Lizenznehmer einer Bibliothek keine Quellen „abkupfern“.

automatische Auflösung von Referenzen: Beim Einbinden der Bibliotheksroutinen muß man (im Gegensatz zum direkten Hinzubinden von Object-Files) die einzubauenden Object-Files <u>nicht</u> <u>explizit</u> aufführen, sondern der Linker fischt sich die notwendigen Teile <u>selbsttätig</u> aus der Library heraus (und erwischt dabei auch die intern aufgerufenden Unterprogramme), d.h. der Aufwand für die Ausformulierung des Bindesteps verringert sich drastisch.

Nomenklatur: Unterprogrammbibliotheken werden umgangssprachlich durch einen bestimmten Namen gekennzeichnet (im Folienbeispiel würde man von der „maus"-Bibliothek sprechen. Es gibt eine Reihe von Konventionen, welche festlegen, wie dieser Name in den Pfadnamen der Bibliothek (d.h. in den Dateinamen) und in die Argumentliste des Binders (`$(FC)`, `$(CC)`) eingeht:

- Der Pfadname beginnt immer mit dem Präfix `lib`, an das sich der eigentliche Bibliotheksname oder eine Kurzform desselben anschließt.
- Bibliotheksnamen weisen im Extensionteil die Kennung `.a` auf; das `.a` dient als Abkürzung für „archive".
- Beim Binden wird nur der innere Teil des Bibliotheksnamens mit vorangestellter Option `-l` angeben (Beispiel: `-lmaus`, wichtig: Kleinbuchstabe `l`), dieser Option geht meist noch eine Anweisung voran, durch die dem Binder mitgeteilt wird, wo er nach der Library suchen soll. Hierzu wird die Option `-L` angegeben (wichtig: Großbuchstabe `L`), auf welche unmittelbar der Name eine Directorys folgt, in dem der Linker nach der Bibliothek `libxxxx.a` suchen soll.
- Ein bestimmtes Dienstprogramm baut die Bibliothek aus Object-Files aufgebaut: der Archiver `ar`. Vorsicht: Diesen Archiver nicht mit dem „tape archiver" `tar` verwechseln oder mit einem Versionsverwaltungs-Archiver oder mit einem Archivierungsprogramm zum Ablegen von Daten auf einem Hintergrundmedium — der Begriff „Archiv" taucht an vielen Ecken und Enden auf, bedeutet aber häufig was ganz Anderes !

externe Symbole und Object-Files: Wenn man nicht mit Unterprogrammbibliotheken arbeitet, sondern mit explizit angegebenen Object-Files, so erfolgt das Zusammentragen der Files nicht anhand der Namen der von der Applikation referierten Unterprogramme (diese wird oft als „Liste der externen Referenzen" bezeichnet), sondern eben durch Angabe der Object-File-Namen. Diese können etwas mit den Unterprogrammnamen zu tun haben, müssen es jedoch nicht. Darin besteht ein ganz wesentlicher Unterschied zwischen den beiden Verfahren:

- Werden explizit Object-Files zusammengeführt, so müssen Sie selbst dafür Sorge tragen, daß in diesen Object-Files auch alle externen Referenzen intern abgesättigt werden können; m.a.W.: daß die Object-Files auch tatsächlich alle benötigten Unterprogramme enthalten. Eine automatische Suche ist auf diesem Wege nicht möglich.
- Wird mit Libraries gearbeitet, so sucht der Linker automatisch nach den entsprechenden externen Symbolen. Dagegen spielt es keine Rolle, in welchem der in der Library archivierten Object-File das entsprechende externe Symbol versteckt ist, d.h. der Object-Filename tritt nach außen hin gar nicht in Erscheinung. Dies wird dadurch ermöglicht, daß der Archiver bei Bibliotheksaufbau eine Symboltabelle anlegt, in der aufgeführt ist, welche externen Symbole in welchem in der Library eingebauten Object-File vorhanden sind (solch eine Symboltabelle wird in der weiter unten folgenden Folie „Library-Erstellung durch `ar`" gezeigt). Der Binder ermittelt anhand dieser Symboltabelle das aus der Archiv-Library einzubauende Object-File. Vorsicht: Wenn ein externes Symbol in mehreren der Object-Files auftritt, gibt's Probleme !

- **Einfaches Programm** `mampf.f`**:**

```
SUBROUTINE␣MAMPF
CALL␣APFEL
CALL␣BIRNE
END
```

- **Übersetzung (liefert** `mampf.o`**):**

```
fort77␣-c␣mampf.f
```

- **Ausdrucken der Symboltabelle:**

```
nm mampf.o
```

liefert:

```
Symbols from mampf.o:

Name        Value   Scope  Type    Subspace
mampf   |          0|extern|entry  |$CODE$
apfel   |           |undef |code   |
birne   |           |undef |code   |
```


Externe Symbole

Filename vs. Externes Symbol: Zwischen dem Namen einer Fortran-Datei und den in dieser Datei definierten Unterprogrammnamen muß keinerlei Zusammenhang bestehen. In manchen Fällen wird für jede Programmeinheit (Subroutine oder Function) eine eigene Fortran-Datei angelegt, in anderen Fällen werden verschiedene Programmeinheiten in einer Datei zusammengefaßt. Wichtig ist, daß man die in den verschiedenen Programmeinheiten definierten Namen externer Symbole nicht mit dem Filenamen verwechselt.

- Der Dateiname legt fest, wie das Quell- oder Object-File im Unix-Dateisystem angesprochen wird. Durch den Compiler wird eine recht starre Abbildung zwischen Quell- und Object-Name vorgeschrieben (`abc.c`→`abc.o`).
- die externen Symbole beschreiben, welche Entries (Unterprogrammeinsprünge) vorhanden sind. Diese Symbolnamen werden aus den Subroutine- oder Function-Namen abgeleitet, wobei je nach Compiler und Optionssatz bei dieser Ableitung noch folgende Operationen vorgenommen werden können:
  - Der Compiler kann den Symbolnamen unverändert aus dem Quelltext übernehmen (so daß `SUBROUTINE miau` und `SUBROUTINE Miau` verschiedene Routinen sind).
  - Der Compiler kann den Unterprogrammnamen in Kleinbuchstaben umwandeln und dies als externes Symbol verwenden:
    `SUBROUTINE␣Miau`→Symbol `miau`
  - Der Compiler kann den Unterprogrammnamen in Großbuchstaben umwandeln und dies als externes Symbol verwenden:
    `SUBROUTINE␣Miau`→Symbol `MIAU`
  - Der Compiler kann irgendwelche Sonderzeichen (z.B. `_`) an den (ggf. in Groß- oder Kleinbuchstaben gewandelten) Unterprogrammnamen anhängen:
    `SUBROUTINE␣Miau`→Symbol `miau_`

  Welcher Fall bei Ihrem Rechner vorliegt, können Sie der Man-Page des Compilers entnehmen oder mit dem Dienstprogramm `nm` ermitteln. Beim Beispiel auf der Folie (Standardeinstellung des HPUX-Fortran-Compilers) wird der Unterprogrammname in Kleinbuchstaben gewandelt, es werden jedoch keine speziellen Zeichen angehängt.

Utility `nm`: Rufen Sie dazu `nm` mit dem Namen eines Object-Files als Argument auf; `nm` liefert Ihnen dann die Liste der in diesem Object-File angesprochenen externen Symbole (in der Spalte `Name`). Dabei wird in der Spalte `Scope` folgendes angezeigt:

`Scope=extern` Es handelt sich um ein in dem Object-File selbst definiertes externes Symbol (Beispiel: `mampf`).

`Scope=undef` Das entsprechende Symbol wurde nur referiert, d.h. es handelt sich um ein externes Unterprogramm (Beispiel: `apfel,birne`).

Die in der Folie gezeigten Beispiele wurden unter HP-UX erzeugt; andere Betriebssysteme weisen `nm`-Varianten auf, welche sich in der Liste der `nm`-Optionen und im Format des `nm`-Listings unterscheiden können. Die man-Pages zu `nm` für die Betriebssysteme AIX, HP-UX, IRIX, Linux und UXP/M finden Sie in der RZ-WWW-Seite zu diesem make-Kurs. In den ebenfalls über diese WWW-Seite zu erhaltenden Demo-tar-Files `demo4.tar.gz` und `demo5.tar.gz` finden Sie Beispiele für den Aufruf von `nm`.

Seite wurde absichtlich frei gehalten

Library-Erstellung durch `ar`

- **Programm** `obst.f,gwuerz.f`:

```
SUBROUTINE APFEL        SUBROUTINE CHILI
END                     END
SUBROUTINE BIRNE        SUBROUTINE DILL
END                     END
```

- **Übersetzung und Library-Aufbau:**

```
fort77␣-c␣obst.f␣gwuerz.f
ar␣rsv␣libfressalien.a␣\
␣␣␣␣␣␣␣obst.o␣gwuerz.o
```

- **Ausdrucken der Symboltabelle:**

`nm libfressalien.o`

liefert:

```
Symbols from libfressalien.a[obst.o]:
Name        Value   Scope  Type     Subspace
apfel   |         0|extern|entry   |$CODE$
birne   |         8|extern|entry   |$CODE$
Symbols from libfressalien.a[gwuerz.o]:
Name        Value   Scope  Type     Subspace
chili   |         0|extern|entry   |$CODE$
dill    |         8|extern|entry   |$CODE$
```


Einbau der Object-Files in die Library Eine Library wird mit dem Programm `ar` aufgerufen, wobei man die Optionen `rsv` angibt. Auf diese Optionen folgt unmittelbar der Unix-Pfadname der Library-Datei, danach folgen die Object-Files, welche in die Library aufgenommen werden sollen. Falls schon ein entsprechendes Object-File zuvor in der Library enthalten war, wird es durch das neue ersetzt.

`ar`-Optionen `rsv` Die anzugebenden Optionen sind betriebssystemabhängig. Die Optionen müssen nicht durch ein „-“ eingeleitet werden, bei einigen Betriebsystemen kann man jedoch ein solches Zeichen angeben. `r`=replace, `s`=update symbol table, `v`=verbose output.

Begrenzung der Namenslänge der Object-Files Viele `ar`-Versionen erlauben nur eine begrenzte Länge des Namens des einzubauenden Object-Files. Häufig sind maximal fünfzehn Zeichen erlaubt, d.h. nach Abzug der beiden Zeichen „`.o`“ für die Extension verbleiben höchstens dreizehn Zeichen für den Teil des Filenames vor dem Punkt (und damit auch für den entsprechenden Teil des Quelldateinamens). Bei einigen Betriebssystemen finden Sie weitergehende Informationen zum `ar`-Fileformat in Kapitel 4 (File-Formate) der man-Pages z.B. über das Kommando `man 4 ar`.

Symboltabelle einer Library Ähnlich wie bei den Symbolen in einem einzelnen Object-File kann einem auch bei Libraries das Dienstprogramm `nm` Informationen über die Symboltabellen ausgeben. Die Aufrufsyntax ist ganz analog (siehe Beispiel auf der Folie). Die bei obigem `ar`-Aufruf angegebene Option `s` sorgt übrigens dafür, daß die Library-Symboltabelle nach dem Einbau eines Object-Files auf den neuesten Stand gebracht wird. Nicht alle Betriebssystem weisen diese `ar`-Option auf: Bei einigen muß die Symboltabelle über einen gesonderten Aufruf erzeugt werden; das dazu benötigte Dienstprogramm heißt meist `ranlib`. Anmerkung: In der Regel dürfen die Symboltabelleneinträge wesenlich größere Namenslängen aufweisen als die im Archiv eingetragenen Object-Files.

Beispiel dazu Komprimierte tar-Files mit den Quellen und dem Makefile dazu finden Sie auf der RZ-WWW-Seite zu make (siehe Folie „Dokumentationen zu make“), die Files heißen `demo4.tar.gz` und `demo5.tar.gz`.

Library-Erstellung in make

Nomenklatur:

```
<libname>(<objectname>)
```

libfressalien.a(obst.o) ← obst.o ← obst.f

Target — intermediär — Dependency

Zweistufiger Abhängigkeitsbaum:

```
libfressalien.a:␣libfressalien.a(obst.o)␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣libfressalien.a(gwuerz.o)

libfressalien.a(obst.o):␣obst.f
libfressalien.a(gwuerz.o):␣gwuerz.f
```

Implizite Regel:

```
.f.a:
TAB $(FC)␣-c␣$(FFLAGS)␣$<
TAB ar␣-rs␣$@␣$*.o
TAB rm␣-f␣$*.o
```


Nomenklatur: Wie schon mehrfach geschildert wurde, entsteht eine Library durch Einbau von Object-Files. Das Object-File innerhalb dieser Library wird in der genannten Form `<libname>(<objectname>)` angesprochen, z.B. `libfressalien(obst.o)`. Durch diesen „Pfadnamen" kann es von dem isolierten Object-File (z.B. `obst.o`) unterschieden werden. Charakteristisch für die Erstellung einer Library mit make ist, daß das isolierte Object-File nur „intermediär" in Erscheinung tritt: Es muß kurzfristig existieren, da der Compiler eben nur solche isolierten Object-Files ausspuckt; aber dieses isolierte Object-File wird sofort nach dem Einbau in die Library gemeuchelt.

Abhängigkeitsbaum: Die Library (gekennzeichnet durch ihren Pfadnamen, z.B. `libfressalien.a`) ist von allen in sie eingebauten Object-Files abhängig — nicht jedoch von den isolierten Object-Files. **Das isolierte Object-File tritt im Abhängigkeitsbaum nirgendwo auf.** Das jeweilige in der Library eingebaute Object-File `<libname>(<objectname>)` hängt von dem dazugehörigen Quelltext-File ab (und ggf. noch von Include-Files). Aus diesem Grund ergibt sich die genannte zweistufige Formulierung.

Implizite Regel: Zur Erstellung der Library wird eine Double-Dot-Rule angegeben, durch welche Quell-Files (Extension `.f`) in Archive-Libraries (Extension `.a`) überführt werden. Anhand des zweistufigen Abhängigkeitsbaums können dann die Unix-Kommandos für alle in Frage kommenden Files von make erstellt und ausgeführt werden:

- Übersetzung des Quell-Files (damit verbunden: erzeugung des isolierten intermediären Object-Files)
- Einbau des soeben erzeugten Object-Files in die Library und Aktualisierung der Symboltabelle der Library
- Löschung des intermediären Object-Files

Beispiel: Makefile zu diesen Folien

Makefile für die Erstellung dieser Folien

Goals:

`all` **:**
: **Erzeuge komprimierte PostScript-Files (einseitige Ausgabe, Vierseitenmodus, Ankündigungsposter)**

`clean` **:**
: **Bereinigung des Verzeichnisses**

`show,show4,showA` **:**
: **Preview der PostScript-Files in X11-Fenster**

`qad` **:**
: **schnelles Preview der Ergebnisse während der Entwicklung**

`simplex,duplex,quadro` **:**
: **Ausgabe auf dem Drucker**


```
##############################################################
#␣Makefile␣zur␣Erstellung␣der␣Materialien␣fuer␣den␣make-Kurs␣#
##############################################################
#
#␣$Id:␣Makefile,v␣1.33␣1996/01/25␣12:28:34␣rz65␣Exp␣$
#
SHELL=/bin/sh
NULL=
#
#␣LaTeX-Quell-Dateien
LATEX_SRCS␣=␣mk.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣mk_dokumentation.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣mk_example_mf1.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣mk_gliederung.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣mk_library.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣mk_macros.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣mk_nomenklatur.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣mk_optionen.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣mk_problembeschreibung.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣mk_spielwiese.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣mk_syntax.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣mk_titel.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣tmp_makefile.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣$(NULL)
LATEX_SRCS2␣=␣mk_A4_Ankuendigung.tex␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣$(NULL)␣
#
```

```
##############################################################
#␣Abhaengigkeiten␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#
##############################################################
#
.PHONY:␣all␣clean␣show␣show4␣showA␣qad␣simplex␣duplex␣quadro
#
#␣Goal␣'all':␣Es␣werden␣alle␣komprimierten␣PostScript-Files␣aufgebaut
all:␣mk.ps.gz␣mk4.ps.gz␣mk_A4_Ankuendigung.ps.gz
#
#␣Erzeugung␣der␣komprimierten␣PostScript-Ausgaben␣aus␣den␣DVI-Files
mk.ps.gz:␣mk.dvi␣
TAB dvips␣-f␣mk.dvi␣|␣gzip␣-9␣>␣mk.ps.gz
#
mk4.ps.gz:␣mk.ps.gz
TAB cat␣mk.ps.gz␣|␣gzip␣-d␣\
␣␣␣␣␣␣␣␣␣␣␣␣|␣pstops␣'2:0L@.67(20.5cm,0)+1L@.67(20.5cm,14.85cm)'␣\
␣␣␣␣␣␣␣␣␣␣␣␣|␣gzip␣-9␣>␣mk4.ps.gz
#
mk_A4_Ankuendigung.ps.gz:␣mk_A4_Ankuendigung.dvi␣
TAB dvips␣-f␣mk_A4_Ankuendigung.dvi␣|␣gzip␣-9␣>␣mk_A4_Ankuendigung.ps.
#
#␣Erzeugung␣der␣temporaeren␣LaTeX-Quellen␣(TAB's␣sichtbar␣machen)
tmp_makefile.tex:␣Makefile␣mk_filter.sh
TAB mk_filter.sh␣Makefile␣tmp_makefile.tex
#
#␣Erzeugung␣der␣DVI-Files␣aus␣den␣LaTeX-Quellen
mk.dvi:␣$(LATEX_SRCS)␣
TAB latex␣mk.tex
#
mk_A4_Ankuendigung.dvi:␣$(LATEX_SRCS2)␣
TAB latex␣mk_A4_Ankuendigung.tex
#
#␣Goal␣'clean':␣Bereinigung␣des␣Verzeichnisses
clean:
TAB rm␣-f␣*.dvi␣*.ps.gz␣*.aux␣*.lof␣*.log␣*.toc␣*.lot␣*~␣tmp_*.tex
```



```
##############################################################
#␣diverse␣Goals␣zur␣Ausgabe␣des␣Dokuments␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#
##############################################################
#
#␣Goals␣zum␣Anzeigen␣der␣Ergebnisse␣in␣einem␣X-11-Fenster
show:␣␣␣mk.ps.gz
TAB cat␣mk.ps.gz␣|␣gzip␣-d␣>␣/tmp/mk.show.ps␣
TAB gv␣/tmp/mk.show.ps
TAB rm␣/tmp/mk.show.ps
#
show4:␣␣mk4.ps.gz
TAB cat␣mk4.ps.gz␣|␣gzip␣-d␣>␣/tmp/mk4.show.ps␣
TAB gv␣-landscape␣/tmp/mk4.show.ps
TAB rm␣/tmp/mk4.show.ps
#
showA:␣␣mk_A4_Ankuendigung.ps.gz
TAB cat␣mk_A4_Ankuendigung.ps.gz|gzip␣-d>/tmp/mk_A4_Ankuendigung.show.
TAB gv␣/tmp/mk_A4_Ankuendigung.show.ps
TAB rm␣/tmp/mk_A4_Ankuendigung.show.ps␣␣␣␣␣␣␣␣
#
#␣Goal␣'qad'␣(Quick-and-dirty):␣Formatierung,␣Anzeige␣ohne␣PS-Erzeugung␣
qad:␣mk.dvi
TAB xdvi␣-geometry␣+0+0␣mk.dvi
#
#␣Goals␣zum␣Ausdrucken␣der␣Folien␣im␣Ein-,␣Zwei-␣und␣Vierseitenmodus
simplex:␣mk.ps.gz
TAB pu␣-dxerox␣-o␣plex=simplex␣mk.ps.gz
#
duplex:␣mk.ps.gz
TAB pu␣-dxerox␣-o␣plex=duplex␣mk.ps.gz
#
quadro:␣mk4.ps.gz
TAB pu␣-dxerox␣-o␣plex=duplex␣mk4.ps.gz
```