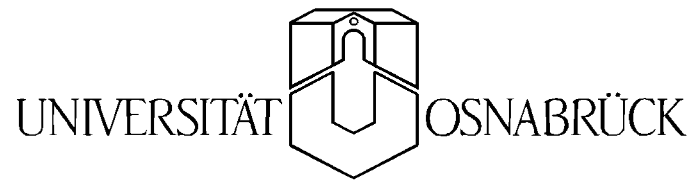
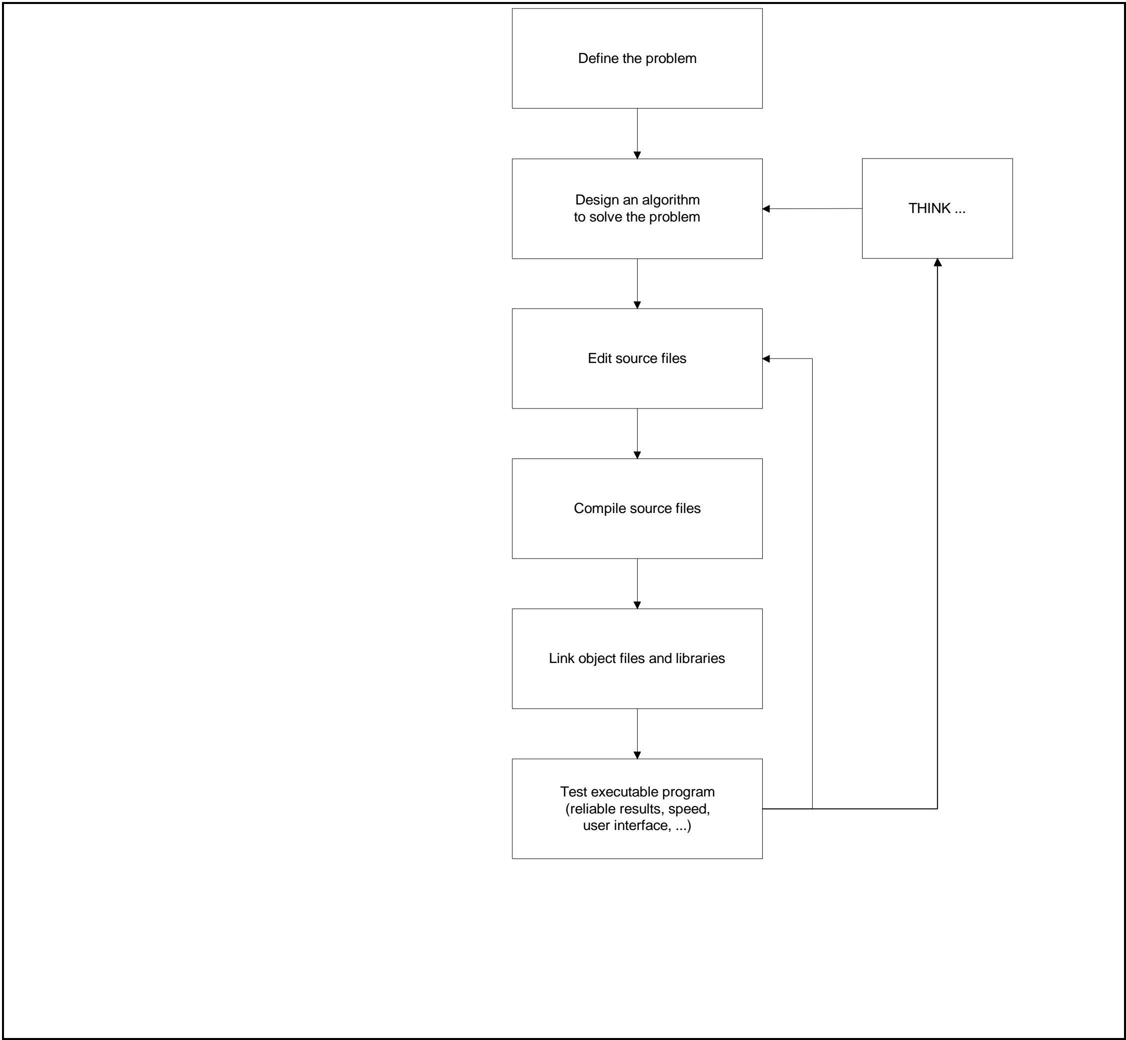


Rechenzentrum

Programmentwicklung unter UNIX


Define the problem
Design an algorithm
to solve the problem
THINK ...
Edit source files
Compile source files
Link object files and libraries
Test executable program
(reliable results, speed,
user interface, ...)

Programmentwicklung unter UNIX

Autor:	Frank Elsner
Adresse:	Universität Osnabrück - Rechenzentrum - Albrechtstraße 28 D-49076 Osnabrück E-Mail: F.Elsner@rz.uni-osnabrueck.de

Version:	1.1
Stand:	97/01/28
Dateiname	E:\DOC\UNIX\unixprog.doc
anonymous FTP	ftp.rz.uni-osnabrueck.de pub/reports/prog_unix-1.1.ps

Inhaltsverzeichnis

1. Einleitung ... 1
    Voraussetzungen, Zielsetzung und Gliederung ... 1
    Abhängigkeiten ... 1
    Verwendete typografische Konventionen ... 2
    Namenskonventionen für Dateien ... 2
    Weiterführende Literatur ... 4
2. Anmelden und Einrichten der Umgebung ... 5
    telnet, login - Anmelden im System ... 5
    bsh, csh, ksh, chsh - Auswählen eines Kommandointerpreters ... 5
    setup - Setzen von Umgebungsvariablen ... 6
    exit - Abmelden aus dem System ... 7
3. Werkzeuge zur Programmentwicklung ... 8
    Entwickeln von Programmen ... 8
    vi, xlc, xlf - Entwickeln von "kleinen" Programmen ... 8
    man, info - Anzeigen von Online-Informationen ... 11
    ar, dbx, ... - Entwickeln von Programmen (unabhängig von der Sprache) ... 11
    xlc, cflow, ... - Entwickeln von C Programmen ... 11
    xlf, ftnchek, ... - Entwickeln von Fortran Programmen ... 11
    awk, perl, ... - Lösen spezieller Probleme ... 12
    find, grep, sort - Sortieren und Suchen ... 12
    ftp, mail, ... - Versenden und Drucken von Dateien und E-Mail ... 12
4. info - Abrufen von Online-Informationen ... 13
    Aufrufen des InfoExplorer Systems ... 13
    Wechseln zwischen Menü- und Textmodus ... 13
    Navigieren im Textmodus ... 13
5. vi - Erfassen und Bearbeiten von Textdateien ... 15
    Starten, Speichern und Beenden ... 15
    Laden und Speichern von Dateien und Beenden der Sitzung ... 16
    Einfügen und Korrigieren von Text ... 16
    Zeilenweises Löschen, Verschieben und Kopieren von Text ... 16
    Bewegen im Text und Markieren von Bereichen ... 17
    Suchen und Ersetzen von Zeichenketten und regulären Ausdrücken ... 17
    Trennen und Zusammenfügen von Zeilen ... 17
    Ausführen von Shell Kommandos, Wiederholen und Rückgängigmachen ... 17
    Anzeigen und Ändern von Optionen ... 19
6. make - Regelbasiertes Generieren von Programmen ... 20
    Erstellen einer make Beschreibungsdatei ... 20
    Erzeugen des ausführbaren Programmes ... 23
7. xlc - Übersetzen und Binden von C Programmen ... 25
    Übersetzen, Binden und Starten ... 25
    Auflisten der Optionen und Argumente ... 25
    Erstellen einer make Beschreibungsdatei ... 26
8. xlf - Übersetzen und Binden von Fortran Programmen ... 27
    Übersetzen, Binden und Starten ... 27

Auflisten der Optionen und Argumente....27
Verwenden nicht-portabler xlf Erweiterungen....28
Erstellen einer make Beschreibungsdatei (Makefile)....28
9. ar - Verwalten von Bibliotheken....30
Erstellen einer Bibliothek....30
Auflisten wichtiger Optionen und Argumente....30
10. dbx, gprof - Suchen von Fehlern und Messen von Programmen....32
Durchführen einer Debug-Sitzung....32
Durchführen einer Zeitmessung....33
Durchführen einer Laufzeitanalyse....34
11. co - Verwalten von unterschiedlichen Programmversionen....35
Ein- und Auschecken....35
Anzeigen des Protokolls aller Änderungen....36
Verwenden einer make Beschreibungsdatei....38
12. Überblick über Standardbibliotheken....40
Verwenden von Standardbibliotheken....40
Zordnen der verfügbaren Dateien....40
13. Überblick über numerische Bibliotheken....41
Verwenden von numerischen Bibliotheken....41
Zordnen der verfügbaren Dateien....41
Verwenden von Computeralgebrasystemen....42
14. Überblick über Graphik-Bibliotheken und -Programme....44
Verwenden von Graphik-Bibliotheken und -Programmen....44
Drucken und Plotten....44
Anhang....46
make Beschreibungs- oder Steuerdatei (Makefile)....46

1. Einleitung

Dieses Handbuch wendet sich an Programmierer, die auf einem UNIX System vornehmlich mit den Programmiersprachen C, C++ und Fortran Programme entwickeln wollen.

Voraussetzungen, Zielsetzung und Gliederung

Grundlegende Kenntnisse über das An- und Abmelden bei einem UNIX System, das Aufrufen von Kommandos und das Erzeugen und Verwalten von Dateien und Verzeichnissen, wie sie im Rahmen eines Einführungskurses in UNIX erworben werden, werden vorausgesetzt. Ferner werden grundlegende Kenntnisse der Programmiersprachen C, C++ und/oder Fortran und der Vorgehensweise bei der Entwicklung von Programmen vorausgesetzt. (Dieses Handbuch ist **kein** Lehrbuch über Programmieren in C, C++ oder Fortran.)

In diesem Handbuch werden schwerpunktmäßig folgende Fragen behandelt:

- **Welche Werkzeuge stehen unter UNIX zur Programmentwicklung zur Verfügung?**
- **Welche Systembibliotheken und welche numerischen und graphischen Bibliotheken stehen unter UNIX zur Verfügung?**

Das Handbuch ist folgendermaßen gegliedert:

- Die ersten Kapitel beschreiben die Werkzeuge zur Entwicklung von Programmen, wobei der Schwerpunkt auf C und Fortran Programmierung liegt.
- Die folgenden Kapitel geben einen Überblick über Systembibliotheken, numerische Bibliotheken und Bibliotheken zur Visualisierung numerischer Ergebnisse.

Abhängigkeiten

Als Grundlage der Beschreibung dient eine Workstation **IBM RISC/6000** des Rechenzentrums (`titan.rz.Uni-Osnabrueck.DE`) unter dem Betriebssystem **AIX**, einem UNIX Derivat.

Die Beschreibung läßt sich, mit einigen Einschränkungen (z.B. Existenz und Pfadnamen von Kommandos und Bibliotheken, unterschiedliche Compiler und Compiler-Optionen), auch auf andere UNIX Systeme übertragen.

Erkundigen Sie sich ggf. bei einem lokalen Experten, wie Sie sich eine entsprechende Umgebung auf Ihrem UNIX System einrichten können.

Verwendete typografische Konventionen

Fettschrift	Fettschrift bezeichnet Definitionen, Kommandos oder wichtige Textpassagen. Beispiel: Das Kommando **`xlc`** ruft den IBM C Compiler auf.
Kursivschrift	Kursivschrift bezeichnet englische Fachausdrücke oder Variablen, die durch konkrete Werte zu ersetzen sind. Beispiel: Geben Sie eine beliebige Zahl *n* ein.
[...]	Eckige Klammern bezeichnen optionale Syntaxelemente. Beispiel: **`vi`** [*filename* ...]
`<...>`	Spitze Klammern bezeichnen Umschalttasten, die zusammen mit einer weiteren Taste (oder zwei weiteren Tasten) eingegeben werden. Beispiel: Geben Sie die Tastenkombination `<CTRL>+V` ein.
`Courier`	Schreibmaschinenschrift bezeichnet Kommandos, Dateinamen oder Programmbeispiele. Dateien sind zusätzlich mit einem Rahmen umgeben. Beispiel: Die mathematische Standardbibliothek heißt `/usr/lib/libm.a.`
`$ ...`	Das Zeichen `$` symbolisiert die Eingabeaufforderung des Kommandointerpreters, hinter dem Sie Kommandos eingeben. Die Benutzereingabe ist durch Fettschrift ausgezeichnet. Sie müssen Kommandos immer durch `<CR>` (*Enter*) bestätigen (ausführen).
`# ...`	Hinter dem Zeichen `#` folgt Kommentar, d.h. eine kurze Erläuterung: Beispiel: `$` **`vi sample5.c`** `# starts the vi editor.`

Namenskonventionen für Dateien

Wie unter UNIX üblich und (oftmals erforderlich !) werden in diesem Handbuch folgende Endungen für Dateinamen verwendet:

Endung	Bedeutung	Beispiel
`.a`	Bibliothek	`libm.a`
`.C`	C++ Quelltextdatei	`sample1.C`
`.c`	C Quelltextdatei	`sample1.c`
`.f`	Fortran 77 Quelltextdatei	`sample2.f`
`.f90`	Fortran 90 Quelltextdatei	`sample3.f90`
`.h`	C Deklarationsdatei (*C Header File*)	`stdio.h`
`.o`	Objektdatei	`sample1.o`
keine Endung	ausführbares Programm	`sample`

Weiterführende Literatur

In diesem Handbuch werden einige Problemfelder nur angeschnitten, zum anderen sind die Beschreibungen von Kommandos und Bibliotheken aus Platzmangel unvollständig. Abhängig von Ihren konkreten Aufgaben benötigen Sie weiterführende Literatur, z.B. die genaue C und/oder Fortran Sprachbeschreibung oder die Referenzdokumentation für die Compiler oder die Bibliotheksfunktionen.

Aufgrund des Umfangs der zur Verfügung stehenden Literatur[1] sei an dieser Stelle nur auf die Benutzerhandbücher zum IBM C bzw. Fortran Compiler (*IBM xlc C Compiler User's Guide* bzw. *IBM xlf Fortran Compiler User's Guide*) und einige Bücher aus der O'Reilly UNIX Nutshell Serie (*Learning the vi Editor, UNIX for Fortran Programmers, Using C on a UNIX System, Managing Projects with make*) verwiesen, die ihrerseits wieder umfangreiche Literaturhinweise enthalten.

Die genannten Handbücher und Bücher stehen Ihnen in der RZ Bibliothek zur Einsicht und Kurzentleihe zur Verfügung.

[1] Nehmen Sie sich bei Gelegenheit die Zeit und stöbern und sichten Sie in einschlägigen Buchhandlungen, welche Bücher zum Thema UNIX verfügbar sind.

2. Anmelden und Einrichten der Umgebung

In diesem Kapitel werden die Verfahren zum An- und Abmelden und die Möglichkeiten zur Konfiguration einer benutzerspezifischen Umgebung beschrieben. Auf lokale Besonderheiten der UNIX Workstation Systeme des RZ wird besonders hingewiesen.

telnet, login - Anmelden im System

Geben Sie zunächst das **telnet** Kommando ein, um sich bei einem UNIX System des RZ mit Internet-Namen ***hostname***.rz.Uni-Osnabrueck.DE anzumelden. Für ***hostname*** können Sie z.B. titan einsetzen:

```
$ telnet titan.rz.Uni-Osnabrueck.DE # starts terminal emulation.
```

Danach werden Sie durch das **login** Kommando automatisch aufgefordert, Benutzerkennung (*user identification*, *login name*) und Passwort (*password*) einzugeben, um sich gegenüber dem System zu authentisieren, hier z.B. mueller und geheim12.:

```
login: mueller
password: geheim12.
```

bsh, csh, ksh, chsh - Auswählen eines Kommandointerpreters

Die bekanntesten Kommando-Interpreter unter UNIX sind die Bourne Shell **bsh**, die Korn Shell **ksh** und die C Shell **csh**.

Der Systemadministrator hat in der Datei /etc/passwd einen Kommando-Interpreter voreingestellt. Diese Voreinstellung können Sie bei Bedarf mit dem **chsh** Kommando interaktiv verändern[2]:

```
$ chsh    # changes login shell.
```

Die folgenden Konfigurationsdateien beziehen sich auf den **Korn Shell** Kommando-Interpreter **ksh**, der auf den RISC/6000 Systemen des RZ voreingestellt ist.

[2] In diesem Skript wird ausschließlich die Korn Shell verwendet.

Zu Beginn jeder Sitzung wird automatisch die Systemdatei `/etc/profile` ausgeführt. Sie können in einer eigenen Konfigurationsdatei `$HOME/.profile` festlegen, welche weiteren Kommandos **automatisch** ausgeführt werden sollen, wenn Sie eine Sitzung beginnen. Darüberhinaus können Sie in einer eigenen **ksh** Konfigurationsdatei festlegen, welche Kommandos **bei jedem Aufruf** des **ksh** Kommando-Interpreters ausgeführt werden sollen:

Konfigurationsdatei	... wird ausgeführt	... veränderbar durch
/etc/profile	zu Beginn der Sitzung	Systemadministrator
$HOME/.profile	zu Beginn der Sitzung	Eigentümer
$HOME/.kshrc	bei jedem Aufruf von **ksh**	Eigentümer

setup - Setzen von Umgebungsvariablen

Das vom RZ entwickelte **setup** Kommando informiert in knapper Form über verfügbare Produkte und richtet Umgebungen für Produkte ein. Eine detaillierte Auflistung aller verfügbaren Produkte erhalten Sie durch:

$ **setup -al**[3]

Zweckmäßigerweise richten Sie die Umgebungen für benötigte Produkte sofort nach dem Anmelden ein. Das folgende Beispiel zeigt mögliche Einträge in der Konfigurationsdatei `$HOME/.profile`:

```
      export TERM=vt220                    # sets/exports TERM
variable.
      stty echoe ^H                        # enables backspace.
      set -o emacs                         # enables command-line
editing.
      setup elm vast90 f90                 # sets up environment for
                                           # elm mail and some f90
tools.
```

[3] Hinter setup verbirgt sich folgendes alias Kommando: ". /usr/misc/setup/setup.sh". Sie können also auch schreiben: . /usr/misc/setup/setup.sh -al

exit - Abmelden aus dem System

Beenden Sie Ihre Sitzung durch Eingabe des Kommandos **`exit`** bzw. durch Eingabe der Tastenkombination `<CTRL>+D`:

```
$ exit
```

3. Werkzeuge zur Programmentwicklung

In diesem Kapitel erhalten Sie eine Auflistung wichtiger Kommandos zur Programmentwicklung[4]. In den folgenden Kapiteln werden einige ausgewählte Kommandos genauer behandelt. Sie können sich mittels der Kommandos **`man`** oder **`info`** eine Online-Beschreibung aller verfügbaren Kommandos anzeigen lassen.

Entwickeln von Programmen

Sie entwickeln Programme zur Lösung von Problemen und führen diese Programme auf einem Computersystem aus. Hieraus folgt, daß Sie Kenntnisse aus verschiedenen Bereichen aktivieren und verknüpfen müssen.

Sie haben also die Aufgabe,

1. (**Fachwissen in einem Anwendungsgebiet**)
   ein bestehendes Problem durch einen Algorithmus zu lösen,

2. (**Kenntnis einer Programmiersprache**)
   den Algorithmus in eine Programmiersprache umzusetzen,

3. (**Kenntnis einer Entwicklungsumgebung**)
   die Quelldatei der gewählten Programmiersprache zu übersetzen, zu binden, auszuführen und die Ergebnisse kritisch zu überprüfen.

Dieses Handbuch befaßt sich vornehmlich mit dem 3. Punkt, d.h. mit der Frage, welche Werkzeuge zur Entwicklung von Programmen zur Verfügung stehen.

vi, xlc, xlf - Entwickeln von "kleinen" Programmen

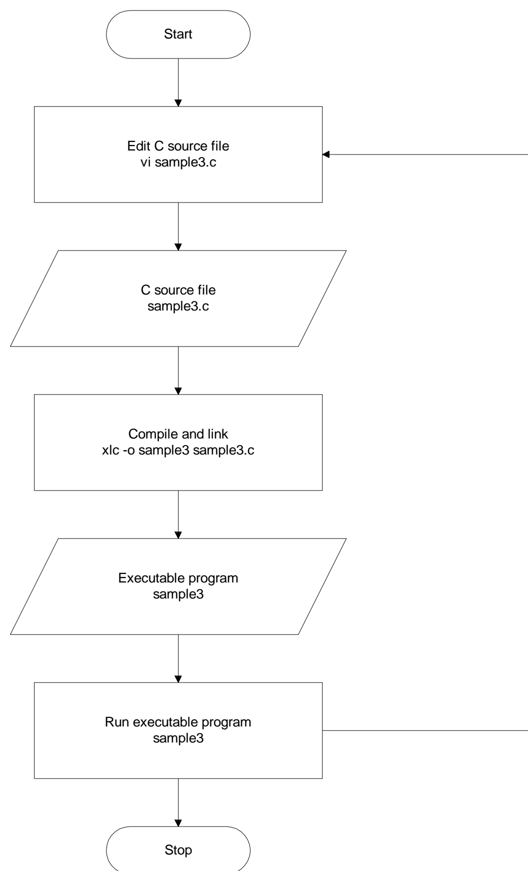
Die wesentlichen Werkzeuge zur Programmentwicklung sind **Editor**, **Compiler** und **Linker**. Die im folgenden behandelten Compiler **`xlf`** und **`xlf90`** (für Fortran 77 und Fortran 90) und **`xlc`** und **`xlC`** (für C und C++) rufen den Linker **ld** automatisch auf.

---

[4] Das UNIX Betriebssystem ist mit einer Vielzahl von "kleinen" nützlichen Kommandos ausgestattet, die nahezu beliebig kombiniert werden können. Hierzu werden Datenströme über Pipelines durch die Werkzeuge geleitet und in jedem Werkzeug einem Arbeitsschritt unterworfen (in Analogie zu einem am Fließband zusammengesetzten Auto). Diese Stärke von UNIX, komplexe Arbeiten in eine Vielzahl von kleinen Schritten zu erledigen, erschließt sich dem Neuling allerdings erst nach einer gewissen Eingewöhnungszeit.

Sprache/ Aufgabe	Fortran 77	Fortran 90	C	C++
Erfassen des Quelltextes	`vi sample1.f`	`vi sample2.f90`	`vi sample3.c`	`vi sample4.f`
Übersetzen und Binden	`xlf -o sample1 sample1.f`	`xlf90 -o sample2 sample2.f90`	`xlc -o sample3 sample3.c`	`xlC -o sample4 sample4.C`
Ausführen des Programms	`sample1`	`sample2`	`sample3`	`sample4`

In der Regel treten bei der Übersetzung (Schritt 2) oder bei der Ausführung (Schritt 3) Fehler auf, die sukzessive eliminiert werden. Das Entwickeln von Programmen stellt demnach einen iterativen Prozeß dar, der de facto nie beendet wird, weil ein Programm immer weiter entwickelt werden kann (durch Beseitigen von Fehlern und/oder Hinzufügen neuer Eigenschaften):



Neben diesen elementaren Werkzeugen zur Programmentwicklung gibt es weitere UNIX Werkzeuge, die in der Regel bei größeren Entwicklungsprojekten verwendet werden. Es ist allerdings prinzipiell (!) möglich, sich mit den oben genannten beiden Werkzeugen, nämlich Editor und Compiler, zu begnügen.

Im folgenden erhalten Sie einen Überblick über die verfügbaren Werkzeuge - Sie sollten sich, entsprechend Ihrem Arbeitsstil und/oder den Vorgaben des Projektes in Ihrer täglichen Arbeit konsequent auf einige ausgewählte Werkzeuge beschränken[5]. Die folgenden Abschnitte stellen wichtige Werkzeuge nach Aufgabengebieten zusammen.

Verwenden Sie ggf. das **`man`** Kommando, um sich die Syntax und weitere Erläuterungen direkt am Bildschirm anzeigen zu lassen, z.B. für den **`xlc`** Compiler durch:

---

[5] Der Autor arbeitet mit vi, xlf, xlc, xlf90, xlC, make, ar, rcs, cextract, dbx und gprof und teilweise mit integrierten Entwicklungsumgebungen unter Windows und UNIX (softbench).

```
$ man xlc
```

man, info - Anzeigen von Online-Informationen

info	ruft das IBM InfoExplorer Informationssystem auf.
man	liefert eine Online Beschreibung eines Kommandos.

ar, dbx, ... - Entwickeln von Programmen (unabhängig von der Sprache)

ar	verwaltet Objektbibliotheken.
dbx	**d**ient zum kontrollierten Ausführen von Programmen (*Debugging*).
ld	bindet Objektdateien und Bibliotheken zu ausführbaren Programmen.
nm	listet Symbole in Objektdateien und Bibliotheken auf.
make	automatisiert die Generierung von Programmen.
gprof	dient zur Laufzeitanalyse von Programmen.
sccs/rcs	verwaltet Versionen von Dateien eines Projektes.
size	zeigt die Größe der Segmente in einem Programm an.
strip	entfernt die Symboltabelle aus einem Programm.
vi, emacs	erzeugt und bearbeitet Textdateien.

xlc, cflow, ... - Entwickeln von C Programmen

cb	verbessert das Layout von C Quellen.
cextract	extrahiert Funktionsbeschreibungen aus C Quellen.
cflow	stellt den Kontrollfluß in einem C Programm dar.
cxref	erzeugt Kreuzreferenzen zwischen Funktionen.
lint	führt eine genaue syntaktische und semantische Analyse durch.
xlc	übersetzt und bindet C Programme.

xlf, ftnchek, ... - Entwickeln von Fortran Programmen

fsplit	trennt eine Fortran Quelldatei nach Programm-Einheiten auf.
ftnchek	führt eine genaue syntaktische und semantische Analyse durch.
xlf	übersetzt und bindet Fortran 77 Programme.
f90, xlf90	übersetzt und bindet Fortran 90 Programme.

`vast90`	konvertiert zwischen Fortran 77 und Fortran 90 Programmen.

awk, perl, ... - Lösen spezieller Probleme

`awk, perl`	dient zur Mustererkennung und -verarbeitung.
`lex`	erzeugt Scanner für lexikalische Analyse.
`sed`	bearbeitet Textdateien (*stream editor*).
`yacc`	erzeugt Parser für syntaktische Analyse.

find, grep, sort - Sortieren und Suchen

`find`	sucht Dateien unter vorgegebenen Bedingungen.
`grep`	sucht in Dateien nach regulären Ausdrücken.
`sort`	sortiert in Dateien nach vorgegebenen Feldern.

ftp, mail, ... - Versenden und Drucken von Dateien und E-Mail

`ftp`	sendet oder empfängt Dateien im Netzwerk.
`lpr`	druckt eine Datei auf einem ausgewählten Drucker/Plotter.
`mail, elm`	sendet und empfängt E-Mail.

Die Liste ist bei weitem nicht vollständig[6]. Sie enthält eine Auswahl von Standard UNIX Kommandos und weiteren kommerziell verfügbaren (z.B. `vast90`) und Public Domain Produkten (z.B. `rcs`).

[6] Konsultieren Sie z.B. das Verzeichnis /usr/bin und /usr/local/bin.

4. info - Abrufen von Online-Informationen

In diesem Kapitel wird in Kürze das IBM Informationssystem **InfoExplorer** vorgestellt, das u.a. Informationen zu allen Kommandos enthält (nur RISC/6000). Online Dokumentation zu jedem einzelnen Kommando in der UNIX üblichen Form (*Manual Page*) können Sie darüberhinaus über das `man` Kommando abrufen.

Aufrufen des InfoExplorer Systems

Sie rufen **InfoExplorer** über das `info` Kommando auf:

```
$ info
```

Die folgende Kurzbeschreibung bezieht sich auf die zeilenorientierte Benutzeroberfläche.

Wechseln zwischen Menü- und Textmodus

Im **InfoExplorer** befinden Sie sich entweder im Menü- oder im Textmodus. Sie gelangen durch Eingabe von `<CTRL>+o` in die Menüzeile. In der Menüzeile können Sie durch Drücken der Cursortasten zwischen verschiedenen Menüpunkten wechseln und durch Drücken der Eingabetaste einen Menüpunkt auswählen. Falls Sie einen Artikel ausgewählt haben, wechselt **InfoExplorer** automatisch in den Textmodus und zeigt den ausgewählten Artikel an.

Durch Auswahl von *Exit* beenden Sie die **InfoExplorer** Sitzung. Es steht Ihnen u.a. ein Menüpunkt *Help* zur Verfügung steht, in dem ein Überblick über **InfoExplorer** und seine Bedienung gegeben wird.

Navigieren im Textmodus

Im Textmodus stehen Ihnen folgende Kommandos zur Verfügung:

Tastenkombination	Funktion
`<CTRL>+o`	wechselt zwischen Menü- und Textmodus.
`<CTRL>+n`	blättert eine Seite vor (*next*).

<CTRL>+p	blättert eine Seite zurück (*previous*).
<CTRL>+f	springt zum nächsten Stichwort (f*orward*).
<CTRL>+b	springt zum vorhergehenden Stichwort (*backward*).
Cursortasten	bewegen den Cursor im Text.

5. vi - Erfassen und Bearbeiten von Textdateien

In diesem Kapitel wird der Editor **vi** (*visual editor*) behandelt. **vi** ist ein kommando-orientierter Editor, mit dem Textdateien bearbeitet werden können. Der **vi** ist grundsätzlich auf jedem UNIX System vorhanden und stellt deshalb den "kleinsten gemeinsamen Nenner" bei der Texterfassung -und bearbeitung unter UNIX dar.

Das Bearbeiten von Texten gehört zu den häufigsten Tätigkeiten, speziell bei der Programmentwicklung. Die in diesem Kapitel gegebene Referenz ist deshalb sehr ausführlich. Sie sollten sich möglichst frühzeitig und intensiv mit den Möglichkeiten des **vi** Editors (oder eines anderen Editors wie z.B. **emacs** oder **the**) auseinandersetzen.

Starten, Speichern und Beenden

Geben Sie folgendes Kommando zum Start des **vi** Editors ein, wobei im Beispiel die Datei `sample1.f` geladen bzw. neu erzeugt wird:

```
$ vi sample1.f
```

Die wichtigste Taste zur Bedienung des **vi** ist die `ESCAPE` Taste (im folgenden mit `<ESC>` bezeichnet), die jederzeit einen Wechsel in den **Kommando-Modus** bewirkt:

```
<ESC>       # always changes to command mode.
```

Ein Wechsel in den **Texteingabe-Modus** erfolgt u.a. durch Eingabe von `<ESC>` und eines der Kommandos `i`,`a` ,`I` und `A` und wird durch `<ESC>` wieder beendet. Sie können eine Texteingabe auch als ein »langes« Kommando der Form **<ESC>i*text*<ESC>** interpretieren, mit dem Sie den Text *text* eingeben:

```
<ESC> i text<ESC>       # inputs text into current file.
```

Ein Wechsel in den **Systemzeilen-Modus** erfolgt durch Eingabe von `<ESC>` und eines der Kommandos `:`, `!` und `?`. Der Cursor springt in die letzte Bildschirmzeile, und Sie können nun eines der Subkommandos eingeben. Ein Subkommando und die folgenden Argumente müssen mit der `<CR>` Taste (*Enter*) abgeschlossen werden.

Die folgende Beschreibung setzt den Kommandomodus voraus. Sie können jederzeit `<ESC>` (auch mehrmals hintereinander) eingeben, um diese Voraussetzung herzustellen

Laden und Speichern von Dateien und Beenden der Sitzung

Kommando	Bedeutung
`: n`	wechselt zur nächsten geladenen Datei.
`: e!` ***file***	lädt die Datei *file* als neue aktuelle Datei.
`: wq!`	sichert die bearbeiteten Dateien und beendet die vi Sitzung.
`: w! [`***file***`]`	sichert die aktuelle Datei [unter dem Namen *file*].
`: q!`	beendet die Sitzung, ohne die bearbeiteten Dateien zu sichern.
`: q`	beendet die Sitzung.
`<CTRL+A>`	wechselt zur vorhergehenden geladenen Datei.
`: args`	zeigt die geladenen Dateien an, die aktuelle hervorgehoben durch eckige Klammern.

Einfügen und Korrigieren von Text

`a` ***text*** `<ESC>`	fügt den Text *text* ein (Anhängen).
`i` ***text*** `<ESC>`	fügt den Text *text* ab Cursorposition ein.
`A` ***text*** `<ESC>`	fügt den Text *text* am Ende der Zeile ein (Anhängen).
`I` ***text*** `<ESC>`	fügt den Text *text* am Anfang der Zeile ein.
`cw` ***text*** `<ESC>`	ersetzt das nächste Wort durch *text*.
`c$` ***text*** `<ESC>`	ersetzt den Rest der Zeile durch *text*.
`<BACKSPACE>`	löscht Zeichen während des Einfügens.
`<CTRL+V>`	maskiert ein nicht-druckbares Zeichen.
`: r` ***file***	fügt die Datei *file* ab Cursorposition ein.

Zeilenweises Löschen, Verschieben und Kopieren von Text

`[`***n***`] dd`	löscht eine [oder *n*] Zeilen .
`d$`	löscht den Rest der Zeilen.
`[`***n***`] yy`	kopiert eine [oder *n*] Zeilen in einen Puffer.
`y$`	kopiert den Rest der Zeilen in einen Puffer.
`[`***n***`] P`	fügt Puffer vor Cursorposition ein.
`[`***n***`] p`	fügt Puffer ab Cursorposition ein.
`"`***x*** `[`***n***`] yy`	kopiert in den benannten Puffer *x* (*x*=a,...z)..
`"`***x p***	fügt Inhalt von Puffer *x* ab Cursorposition ein.

Bewegen im Text und Markieren von Bereichen

<CTRL> G	zeigt die aktuelle Zeilennummer an.
: set nu	zeigt am linken Rand permanent Zeilennummern an.
h (j, k, l)	bewegt Cursor nach links (oben, unten und rechts).
0	positioniert am Zeilenanfang.
$	positioniert am Zeilenende.
1G	positioniert am Dateianfang.
G	positioniert am Dateiende.
n G	positioniert an der Zeile mit der Nummer *n*.
<CTRL+ F>	rollt einen Bildschirm nach unten.
<CTRL+ B>	rollt einen Bildschirm nach oben.
m***c***	setzt eine Marke *c* an die Cursorposition (c=a, ..., z).
\`***c***	positioniert an Marke *c*

Suchen und Ersetzen von Zeichenketten und regulären Ausdrücken

/***text***	sucht Wörter, die mit *text* beginnen, in Richtung Dateiende.
?***text***	sucht Wörter, die mit *text* beginnen, in Richtung Dateianfang.
n	wiederholt letzte Suche.
N	wiederholt letzte Suche in umgekehrter Richtung.
%	sucht zur aktuellen Klammer die öffnende bzw. schließende.
:%s/***old***/***new***/gc	ersetzt überall *old* durch *new* mit Abfrage (y: Bestätigung, <CR>: nicht ersetzen).

Trennen und Zusammenfügen von Zeilen

i <CR>	trennt Zeile ab Cursorposition.
J	fügt folgende Zeile ans Ende der aktuellen Zeile an.
: set ts=***n***	setzt den Tabulatorabstand auf *n* Zeichen
: set ai	setzt die Option autoindent (Automatisches Einrücken).
<CTRL+D>	geht zur nächstkleineren Einrückung.

Ausführen von Shell Kommandos, Wiederholen und Rückgängigmachen

: ! [***cmd***]	verläßt vi temporär und führt das Kommando *cmd* aus (ggf. Rückkehr mit exit).
: r ! ***cmd***	fügt die Ausgabe von *cmd* ab Cursorposition ein.
!! ***cmd***	benutzt aktuelle Zeile als Eingabe für *cmd* und ersetzt durch dessen Ausgabe.
.	wiederholt letztes Kommando

`u`	macht letztes Kommando rückgängig.

Anzeigen und Ändern von Optionen

`: set all`	zeigt alle gesetzten Optionen an.
`: set` ***`opt`*** `=` ***`value`***	setzt die Option *opt* auf den Wert *value* (siehe auch vi Manual Page zu `$HOME/.exrc`).

6. make - Regelbasiertes Generieren von Programmen

In diesem Kapitel wird das **make** Kommando behandelt. **make** ist zusammen mit **sccs** bzw. **rcs** das bekannteste und am häufigsten verwendete Werkzeug zur Verwaltung umfangreicher Software-Projekte. Ausgehend von einer Beschreibung der Abhängigkeiten der Dateien ist **make** in der Lage, automatisch alle Schritte durchzuführen, die zum Übersetzen und Binden eines ausführbaren Programmes notwendig sind.

Erstellen einer make Beschreibungsdatei

Ein typisches Projekt besteht aus mehreren Quelldateien mit der Endung `.f` (bzw. `.f90`, `.c` oder `.C`) und ggf. weiteren Dateien wie Deklarationsdateien mit der Endung `.h`, sowie Parameterdateien, die spezielle Aspekte des Programmes "parametrisieren", und Eingabedateien mit numerischen oder alphanumerischen Werten, z.B. Meßdaten..

Sie können z.B. ein C Programm automatisiert vom Kommando **make** erzeugen lassen. Hierzu benötigen Sie eine **make** Beschreibungsdatei (*Makefile*), auch als Steuerdatei bezeichnet. Die **make** Beschreibungsdatei benennt alle in einem Projekt verwendeten Dateien sowie die Regeln, um aus ihnen eine Zieldatei (in der Regel ein ausführbares Programm) bzw. mehrere Zieldateien zu erzeugen.

Im folgenden wird ein einfaches Beispiel für eine **make** Beschreibungsdatei für 3 Quelldateien `pgm.c, func1.c, func2.c` beschrieben, die Sie als Vorlage für eigene Projekte verwenden und entsprechend erweitern können.

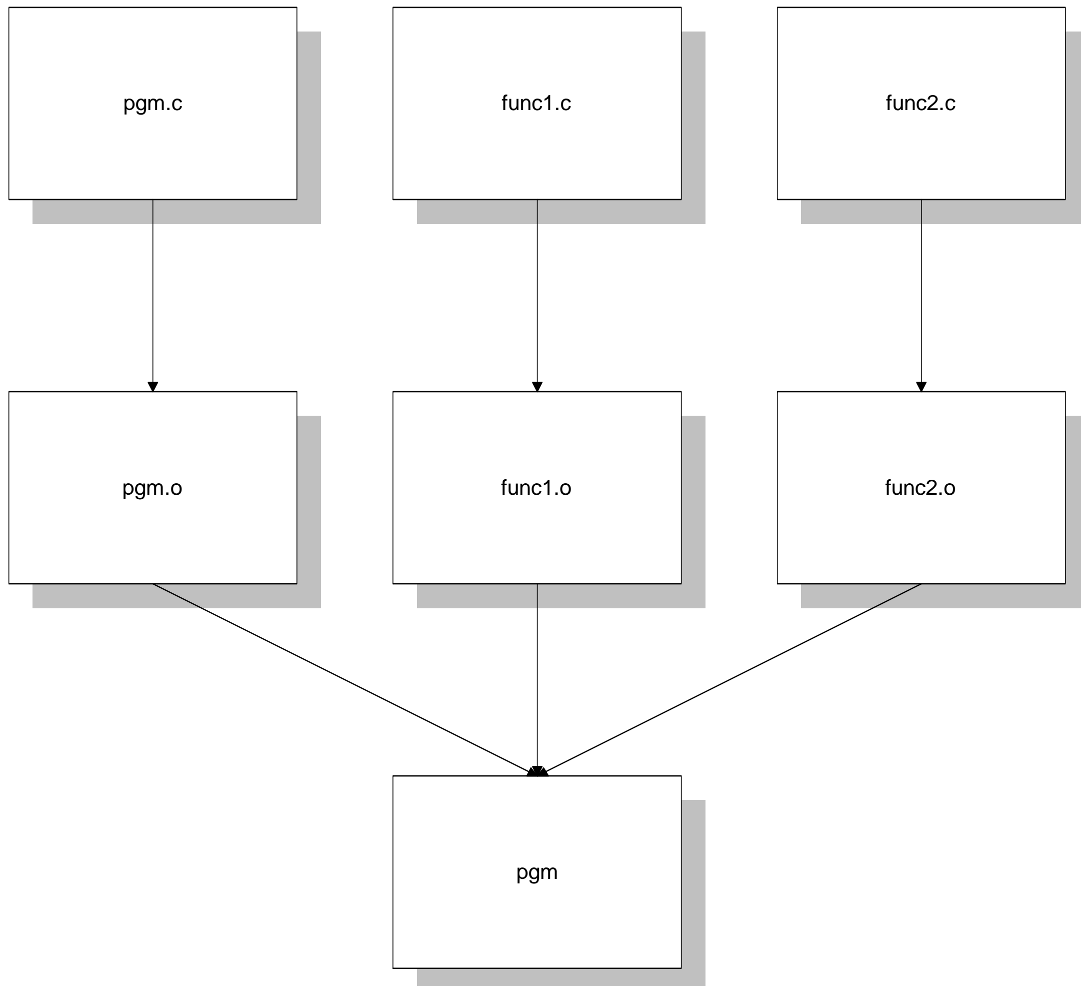
```
$ cat pgm.c
```

```
/* pgm.c */

... main(...)                 /* Program starts here ... */
{
      ... func1(...);         /* and calls func1() ...    */
      ... func2(...);         /* and then func2() ...     */
}
```

Die Zieldatei `pgm` setzt sich aus drei Objektdateien zusammen, die ihrerseits auf drei C Quelldateien beruhen. Die Quelldatei `pgm.c` enthält das Hauptprogramm *main()* und die Quelldateien `func1.c` und `func2.c` enthalten die Funktionen *func1()* und *func2()*,

die im Hauptprogramm *main()* aufgerufen werden. Die Abhängigkeiten zwischen den Quelldateien und den Zieldateien[7] lassen sich folgendermaßen graphisch darstellen:



Sie müssen das Projekt mit den Abhängigkeiten zwischen den Datein in einer *Makefile* Datei[8] beschreiben. Aus der Abbildung ist ersichtlich, daß insgesamt 4 Transformationen notwendig sind, um das ausführbare Programm zu generieren. Ein Eintrag für eine Transformation hat folgende allgemeine Form[9]:

target file: *source file*
`<TAB>`*command to transform source file into target file*

[7] Der Begriff Quelldatei wird in diesem Kontext allgemein für eine Datei verwendet, die über eine Transformatione in eine Zieldatei verwandelt wird. In diesem Sinne ist z.B. func1.o eine der 3 Quelldateien für pgm.

[8] make verwendet per Voreinstellung sogenannte eingebaute Regeln oder Suffixregeln, die die oben genannten Konventionen für Dateinamen voraussetzen, d.h. z.B. daß C Quelldateien die Endung .c besitzen.

[9] Der Tabulator in der 2. Zeile ist zwingend notwendig!

```
# Makefile
# calling sequence: make -f Makefile (or simply make)

# macro(s)

CC=xlc
CFLAGS=-O

# rule(s)

pgm: pgm.o func1.o func2.o
<TAB> $(CC) -o pgm pgm.o func1.o func2.o

func1.o: func1.c
<TAB> $(CC) $(CFLAGS) -c func1.c

func2.o: func2.c
<TAB> $(CC) $(CFLAGS) -c func2.c

pgm.: pgm.c
<TAB> $(CC) $(CFLAGS) -c pgm.c
```

Erzeugen des ausführbaren Programmes

make führt auf Grundlage der **make** Beschreibungsdatei alle notwendigen Schritte in der richtigen Reihenfolge durch.

Starten Sie **make**, wobei es per Voreinstellung die Datei `Makefile` im aktuellen Verzeichnis verwendet und hierin die erste Zieldatei, hier das ausführbare Programm **pgm**, generiert:

```
$ make   # creates target pgm using the file Makefile as default.

xlc -O func1.c
xlc -O func2.c
xlc -O pgm.c
xlc -o pgm pgm.o func1.o func2.o

$ pgm   # runs the executable program.
```

Der Vorteil von **make** besteht insbesondere darin, daß es die Zeitstempel von Dateien auswertet und deshalb bereits durchgeführte Transformationsschritte nicht unnötig ein zweites Mal ausführt.

Verändern Sie zum Beispiel nur die Quelldatei `func2.c` und beobachten Sie den Effekt:

```
$ vi func2.c
$ make

xlc -O func2.c                              # re-compilation.
xlc -o pgm pgm.o func1.o func2.o            # linking.
```

In diesem Fall hat **`make`** nur die geänderte (!) Quelldatei neu übersetzt und danach das Programm neu gebunden, d.h. es hat automatisch nur die unbedingt notwendige Aktionen ausgeführt.

7. xlc - Übersetzen und Binden von C Programmen

In diesem Kapitel wird der IBM **`xlc`** C Compiler beschrieben.

Übersetzen, Binden und Starten

Sie können eine C Quelldatei `sample1.c` folgendermaßen am Bildschirm auflisten, übersetzen, binden und starten:

```
$ cat sample1.c
```

```
#include <stdio.h>
int main(void)
{
      printf("Hello World!\n");
      exit(0);
}
```

```
$ xlc -o sample1 sample1.c          # compiles and links C source.

$ sample1                           # runs program.

Hello World!
```

Auflisten der Optionen und Argumente

Sie können **`xlc`** folgendermaßen aufrufen, um am Bildschirm eine Kurzbeschreibung der vollständigen Syntax angezeigt zu bekommen:

```
$ xlc
$ man xlc
```

Sie erhalten als Ausgabe dieser Kommandos eine Syntaxbeschreibung, die im folgenden beschränkt auf wichtige Optionen und Argumente wiedergegeben wird:

$ xlc [***options***] ***fn1*** [***fn2*** ...]

Optionen:

`-c`	übersetzt, ohne zu binden
`-g`	erzeugt Debug-Information

`-I`***`incdir`***	definiert Suchpfad *incdir* für Header Files (`#include "..."`)
`-o `***`exefile`***	gibt ausführbarem Programm den Namen *exefile*
`-L`***`libdir`***	definiert Suchpfad ***`libdir`*** für Bibliotheken
`-l`***`x`***	durchsucht Bibliothek ***`libx.a`*** nach externen Referenzen

Argumente:

`fn1``, `***`fn2`***`, ...`	definiert Objektdateien *fn1*, *fn2*, ... (mit Endung `.o`) bzw. definiert Quelldateien *fn1*, *fn2*, ... (mit Endung `.c`)

Erstellen einer make Beschreibungsdatei

Sie können ein C Programm automatisiert vom Kommando **`make`** erzeugen lassen. Hierzu benötigen Sie eine **`make`** Beschreibungsdatei (*Makefile*), die alle notwendigen Informationen für **`make`** enthalten muß.

Im folgenden wird ein Beispiel für eine **`make`** Beschreibungsdatei für 3 Quelldateien `pgm.c, func1.c, func2.c` gezeigt, die Sie als Vorlage für eigene Projekte verwenden können.

```
$ cat Makefile
```

```
      # Makefile

      # macro(s)
      CC=xlc
      CFLAGS=-O
      PGM=pgm
      OBJS=func1.o func2.o

      # rule(s)
      .c.o:
            $(CC) $(CFLAGS) -c $<         # creates object from
source.

      # target(s), first target is default target
      $(PGM): $(PGM).o $(OBJS)
            $(CC) -o $(PGM) $(PGM).o $(OBJS)    # creates executable.
```

Erzeugen Sie nun die gewünschte Zieldatei (*target*), in diesem Fall das ausführbare Programm **`pgm`**, durch Aufruf von **`make`**:

```
$ make   # creates target pgm using dependencies from Makefile.
$ pgm    # runs the executable program.
```

8. xlf - Übersetzen und Binden von Fortran Programmen

In diesem Kapitel wird der IBM **`xlf`** Fortran Compiler beschrieben.

Übersetzen, Binden und Starten

Sie können die Quelldatei `sample2.f` folgendermaßen auflisten, übersetzen, binden und starten:

$ **cat sample2.f**

```
      program main
      print *, "Hello World!"
      end
```

$ **xlf -o sample2 sample2.f**

$ **sample2**

```
      Hello World!
```

Auflisten der Optionen und Argumente

Sie können **`xlf`** folgendermaßen aufrufen, um am Bildschirm eine Kurzbeschreibung der vollständigen Syntax angezeigt zu bekommen:

$ **xlf**
$ **man xlf**

Sie erhalten eine Syntaxbeschreibung, die im folgenden beschränkt auf wichtige Optionen und Argumente wiedergegeben wird:

$ **xlf** [***options***] ***fn1*** [***fn2*** ...]

Optionen:

`-c`	übersetzt, ohne zu binden
`-g`	erzeugt Debug-Information für
`-L`***libdir***	definiert Suchpfad *libdir* für Bibliotheken
`-l`***x***	durchsucht Bibliothek *libx.a* nach externen Referenzen
`-O[`***n***`]`	führt Optimierung der Stufe *n* durch

`-o` ***`exefile`***	gibt ausführbarem Programm den Namen *exefile*
`-pg`	erzeugt *Profiling* Information

Argumente:

`fn1``, ..` definiert Quelldatei(en) *fn1*, ... (mit Endung `.f`)
bzw. definiert Objektdateien(en) *fn1*, ... (mit Endung `.o`)

Verwenden nicht-portabler xlf Erweiterungen

Der **`xlf`** Compiler unterstützt u.a. folgende Erweiterungen des genormten Fortran 77 Sprachumfangs, während der **`xlf90`** Compiler den kompletten Fortran 90 Sprachumfang abdeckt:

- Quelltext kann im freien Eingabeformat eingegeben werden (Option -`k`).
- In Zeichenketten sind analog zu C folgende Symbole erlaubt:
  `\t, \b, \f, \0, \', \", \\, \x`
- Die Zeichen '_' und '`$`' sind in Namen erlaubt.
- Namen dürfen maximal 250 Zeichen lang sein.
- In der Anweisung ... ! *comment* leitet das Ausrufezeichen ! einen *inline*-Kommentar ein, der bis zum Zeilenende reicht.
- Variablen können bereits bei der Definition mit einem Anfangswert versehen werden.

Erstellen einer make Beschreibungsdatei (Makefile)

Sie können ein Fortran Programm automatisiert vom Kommando **`make`** erzeugen lassen, um Schreibarbeit zu vermeiden. Hierzu benötigen Sie eine **`make`** Beschreibungsdatei (*Makefile*), die alle notwendigen Informationen für **`make`** enthalten muß. Im folgenden wird ein Beispiel für eine **`make`** Beschreibungsdatei gezeigt (siehe auch den entsprechenden Abschnitt bei **`xlc`**):

```
# Makefile
# command: make -f Makefile (or simply make)

# macro(s)
F77=xlf
FFLAGS=-O
```

```
# rule(s)
.f.o:
      $(F77) $(FFLAGS) -c $<

# target(s)
$(PGM): $(PGM).o $(OBJS)
      $(F77) -o $(PGM) $(OBJS) $(PGM).o
```

9. ar - Verwalten von Bibliotheken

In diesem Kapitel wird der Bibliotheksverwalter **ar** vorgestellt, mit dem Sie Objektdateien in einer Bibliothek zusammenfassen können. Insbesondere bei größeren Projekten ist es ratsam, inhaltlich zusammengehörige Funktionen in einer Bibliothek zu konzentrieren.

Erstellen einer Bibliothek

Der Bibliotheksverwalter **ar** faßt mehrere Objektdateien eines Projektes oder eines Anwendungsgebietes (z.B. Graphik, Numerik, Systemfunktionen) zu einer Objektbibliothek (oder kurz Bibliothek) zusammen.

Geben Sie folgendes Kommando ein, um eine neue Objektbibliothek `librng.a` zu erzeugen, die die Objektdateien `random1.o, random2.o` und `random3.o` enthalten soll:

```
$ ar vq librng.a random1.o random2.o random3.o

ar: Creating an archive file librng.a.
q - random1.o
q - random2.o
q - random3.o
```

Die folgenden Kommandos sind nun äquivalent, da die Bibliothek `librng.a` die Objektdateien `random1.o` bis `random3.o` enthält:

```
# Using 3 object files ...
$ xlc -o sample1 sample1.c random1.o random2.o random3.o

# Using 1 library file ...
$ xlc -o sample1 sample1.c librng.a
```

Auflisten wichtiger Optionen und Argumente

Sie können die Arbeitsweise von **ar** u.a. über folgende Optionen und Argumente steuern:

```
ar [options] library objs
```

```
$ ar -tv libfile.a                   # lists table of contents.
$ ar -uv libfile.a file(s).o         # updates file(s) in lib.
$ ar -dv libfile.a file(s).o         # deletes file(s) in lib.
```

10. dbx, gprof - Suchen von Fehlern und Messen von Programmen

In diesem Kapitel werden der symbolische Debugger **dbx**, das Meßprogramm **timex** und das Analyseprogramm **gprof** vorgestellt.

Weitere Werkzeuge zur Analyse von Programmen sind z.B. **ftnchek, vast90** und **lint** (syntaktische und semantische Analyse) sowie die Funktionen *gettimer* (für C) und *MCLOCK* (für Fortran) zur Zeitmessung.

Durchführen einer Debug-Sitzung

Übersetzen Sie zunächst alle Quelldateien mit der Option -g. Der Compiler fügt aufgrund dieser Option in allen Objektdateien zusätzliche Informationen hinzu, die für die spätere Ausführung unter Kontrolle des symbolischen Debuggers **dbx** benötigt werden[10] (Änderungen im Makefile: CFLAGS=-g, FFLAGS=-g):

```
cat div_by_zero.f
```

```
      program DivisionByZero
      double precision x
      x=0.0D0
      print *, "Sinus(x)/x fuer x=0: ", sin(x)/x
      print *, "1/0:                  ", 1.0/x
      end
```

```
$ xlf -g -qflttrap -o div_by_zero div_by_zero.f
```

Sie können nun das Programm unter Kontrolle von **dbx** ablaufen lassen. Die Eingabeaufforderung innerhalb einer **dbx** Sitzung ist (dbx):

```
$ dbx div_by_zero
(dbx) help        # lists available dbx commands.
```

[10] Dieses "Aufblähen" mit zusätzlichen Informationen verlangsamt die Programmausführung. Ihre endgültige Version (release version) sollte deshalb mit der Option -O übersetzt werden.

```
run                      - begin execution of the program
print <exp>              - print the value of the expression
where                    - print currently active procedures
stop at <line>           - suspend execution at the line
stop in <proc>           - suspend execution when <proc> is called
cont                     - continue execution
step                     - single step one line
next                     - step to next line (skip over calls)
trace <line#>            - trace execution of the line
trace <proc>             - trace calls to the procedure
trace <var>              - trace changes to the variable
trace <exp> at <line#>   - print <exp> when <line> is reached
status                   - print trace/stop's in effect
delete <number>          - remove trace or stop of given number
screen                   - switch dbx to another virtual terminal
call <proc>              - call a procedure in program
whatis <name>            - print the declaration of the name
list <line>, <line>      - list source lines
registers                - display register set
quit                     - exit dbx
```

```
(dbx) run          # starts program execution.
(dbx) ...
(dbx) quit         # terminates dbx session.
```

Die primäre Aufgabe von **`dbx`** ist die Unterstützung bei der Fehlersuche, z.B. bei der Suche nach einer *Floating Point Exception* wie Division durch Null oder nach einem Zeiger-Fehler, die jeweils einen Programmabsturz (*core dump*) verursachen.

Durchführen einer Zeitmessung

Starten Sie das Programm `sample1` unter Kontrolle von **`timex`**:

```
$ timex sample1 2> sample1.timing > sample1.stdout
```

Nach der Ausführung enthält die Datei `sample1.timing` folgende Zeitmessungen:

```
real    0.06        # turn-around time (time to wait)
                    # = time for loading, executing and exiting
user    0.01        # time spent in user code
sys     0.02        # time spent for system calls
                    # real - user -sys = other pgms running
```

Durchführen einer Laufzeitanalyse

Übersetzen Sie alle Quelldateien mit der Option `-pg`, um Informationen für **`gprof`** zu erzeugen (`CFLAGS=-pg, FFLAGS=-pg`):

```
$ xlc -pg -o sample1 sample1.c
```

Führen Sie nun das Programm `sample1` aus, wobei automatisch eine zusätzliche Datei `gmon.out` erzeugt wird:

```
$ sample1                     # creates profiling info in gmon.out.
```

Das Programm erzeugt automatisch im aktuellen Verzeichnis eine Datei `gmon.out` mit Laufzeitinformationen, die Sie im Anschluß mit **`gprof`** aufbereiten können[11]:
elm

```
$ gprof > sample1.gprof       # creates well-documented profile
                              # file, named: sample1.gprof
$ vi sample1.gprof
```

Die von **`gprof`** erzeugte Datei `sample1.gprof` enhält in verschiedenen Abschnitten Informationen über das Laufzeitverhalten von `sample1`. Für den ersten Überblick ist das flache Profil (*flat profile*) nützlich, in dem die vom Programm verwendeten Funktionen nach Häufigkeit sortiert aufgelistet werden. Andere Abschnitte gehen detailliert auf die Eltern-Kind-Beziehung von Funktionen ein, d.h. sie erläutern, welche Funktion (*caller* oder *parent*) andere Funktionen (*child* oder *callee*) mit einer bestimmten Häufigkeit aufgerufen hat.

Auf Grundlage dieser Informationen können Sie die Kernfunktionen Ihres Programmes (*Hot Spots*) herausfinden und ggf. an diesen Stellen Optimierungen vornehmen.

[11] Beachten Sie beim Profiling immer den Unterschied zwischen der Häufigkeit (Wie häufig wurde eine Funktion aufgerufen?) und der Verweilzeit (Wie lange verweilte das Programm innerhalb einer Funktion?). Die heißen Stellen eines Programmes (Hot Spots) sind Funktionen mit einer hohen relativen Verweilzeit.

11.co - Verwalten von unterschiedlichen Programmversionen

In diesem Kapitel wird das **Revision Control System**, kurz **RCS**, behandelt. **RCS** ist ein System zur Versionskontrolle von Textquellen aller Art.

Ein- und Auschecken

Die wesentlichen RCS Kommandos sind `ci` und `co`, zum *Check In* bzw. *Check Out* einer Arbeitsdatei:

1. Das `ci` Kommando speichert einen Text `xyz` mit allen durchgeführten Änderungen und zusätzlichen Informationen (z.B. Datum jeder Änderung, Autor, Grund der Änderung) in einer neuen bzw. aktualisierten RCS Datei `xyz,v`.

2. Auf Grundlage der RCS Datei kann das zu `ci` komplementäre Kommando `co` z.B. jede beliebige Version des Textes oder ein Protokoll aller durchgeführten Änderungen aus `xyz,v` erzeugen. Dies ist vor allem bei Quelltextdateien hilfreich, wenn eine neue Version nicht stabil ist und deshalb die Vorgängerversion rekonstruiert werden soll.

Als Beispiel dient folgendes (einfache und zudem fehlerhafte) C-Programm `hellow.c`:

```
/* $Header$ */
/* $Log$ */

int main (void)
{
 static char chRCS_Id[]="$Id$"[12];
 printf("Hello World!\n");
 exit(0);
}
```

Übergeben Sie die Quelltextdatei in die Kontrolle des Revision Control Systems:

```
$ setup rcs      # sets environment
$ ci hellow.c    # checks in hellow.c into RCS file hellow.c,v.
```

[12] Beachten Sie die fett ausgezeichnete Zeile, in der das RCS Schlüsselwort `$Id$` enthalten ist.

Das **ci** Kommando fragt zunächst interaktiv nach einer Kurzbeschreibung der Datei hellow.c (hier z.B.: This is a simple "Hello World" Program.) und erzeugt dann eine RCS Datei mit Endung ,v, hier: hellow.c,v, mit allen notwendigen Informationen. Als erste Versionsnummer wird von **ci** automatisch die Nummer 1.1 (*Initial Revision*) vergeben.

Nun können Sie bei Bedarf mit dem Kommando **co** eine **Arbeitsdatei** der Version 1.1 zur weiteren Bearbeitung "auschecken", wobei die Option -l die RCS Datei gegen erneutes "Auschecken" schützt:

```
$ co -l hellow.c # checks out latest version for editing (with lock)
```

Das **co** Kommando sucht die RCS Datei hellow.c,v, und extrahiert die letzte Version als Arbeitsdatei hellow.c. Dabei expandiert **co** die Schlüsselwörter $Header$ (oder kürzer Id) und Log automatisch mit den aktuellen Werten, hier z.B. der Eintrag für Id:

```
$Id: hellow.c,v 1.1 1994/01/18 12:40:34 elsner Exp $
     a)       b)    c)                  d)     e)
```

Die Zeile beinhaltet (a) den Namen der RCS Datei, (b) die Versionsnummer, (c) Datum und Zeit der letzten Änderung, (d)den Autor und (e) den Status, Exp für Experimental.

Fügen Sie zusätzlich eine Zeile der Form /* Log */ und eine Zeile mit der #include <stdio.h> Direktive hinzu und übergeben Sie die modifizierte Arbeitsdatei wieder an RCS:

```
$ ci hellow.c   # checks in for the second time.
```

Das **ci** Kommando fragt nun interaktiv nach einer Kurzbeschreibung der vorgenommenen Änderungen (hier z.B.: Added #include <stdio.h>.) und fügt dann die Änderungen in die Datei hellow.c,v ein. Die Versionsnummer wird von **ci** automatisch erhöht, lautet nun also 1.2.

Anzeigen des Protokolls aller Änderungen

Das **`rlog`** Kommando gibt einen zusammenfassenden Überblick über die RCS Datei `hellow.c, v` und alle vorgenommenen Änderungen (*Log Messages*):

```
RCS file: hellow.c,v
...
description:
This is a simple "Hello World" program.
----------------------------
```

```
revision 1.2
date:1994/01/18 12:41:52;
author: elsner;  state: Exp;  lines: +2 -2
Added #include <stdio.h>.
----------------------------
revision 1.1
date: 1994/01/18 12:40:34;
author: elsner;  state: Exp;
Initial revision
```

Das **ident** Kommando sucht in jeder beliebigen Datei, also auch in einer Datei vom Typ *Executable*, nach RCS Schlüsselwörtern der Form `$Id:<any text>$`. Mit Hilfe von **ident** können Sie also jederzeit feststellen, aus welchen Versionen von Programmquellen ein Programm zusammengesetzt ist:

```
$ ident hellow          # checks hellow for RCS keywords.
```

```
$Id: hellow.c,v 1.1 1994/01/18 12:40:34 elsner Exp $
```

Verwenden einer make Beschreibungsdatei

Ein kleiner Wermutstropfen ist die mangelhafte Zusammenarbeit von **make** unter AIX mit **RCS**. **make** kennt keine vordefinierten Suffixregeln für RCS Dateien.

Ergänzen Sie folgendermaßen Regeln für C und Fortran Programme in **make** Beschreibungsdateien (*Makefiles*):

```
CO=co
F77=xlf
RM=rm

.SUFFIXES: .f,v .c,v

.f,v:
<TAB> (${CO} $*.f; (F77) $(FFLAGS) -o $* $*.f; $(RM) -f $*.f)

.f,v.o:
<TAB> (${CO} $*.f; $(F77) $(FFLAGS) -c $*.f; $(RM) -f $*.f)

...

# <Your project description>
...
```

12.Überblick über Standardbibliotheken

In diesem Kapitel werden die C und Fortran Laufzeitbibliothek vorgestellt und es erfolgen Hinweise auf weitere Standardbliotheken.

Verwenden von Standardbibliotheken

Viele Probleme lassen sich in Teilprobleme zerlegen, für die bereits Funktionen in Standardbibliotheken zur Verfügung stehen. Falls nicht besondere Gründe für die Entwicklung eigener Funktionen sprechen, sollten Funktionen aus Standardbibliotheken verwendet werden.

Folgende Standardbibliotheken stehen u.a. zur Verfügung[13]:

- **`libc.a`** - Standard Bibliothek für C
- **`libxlf.a`** - Standard Bibliothek für Fortran
- **`libm.a`** - Standard Mathematik Bibliothek für C und Fortran

Zordnen der verfügbaren Dateien

Die folgende Tabelle enthält Bibliotheken, ihre Dateinamen und, falls vorhanden, die zugehörigen C Deklarationsdateien (*C header files*):

Bibliothek	**Datei**	**Zugehörige Deklarationen**

Standard Bibliothek für C	`/usr/lib/libc.a`	`/usr/include/stdio.h`
		`...`
		`(siehe: /usr/include)`

Standard Bibliothek für Fortran	`/usr/lib/libxlf.a`	-
Standard Mathematik Bibliothek für C und Fortran	`/usr/lib/libm.a`	`/usr/include/math.h` `/usr/include/errno.h`

[13] Die Aufzählung gilt speziell für das RS/6000 Cluster des Rechenzentrums. Erkundigen Sie sich ggf. bei Ihrem Systemverwalter, welche Bibliotheken installiert sind oder konsultieren Sie das Verzeichnis /usr/lib und die Handbücher Ihres UNIX Systems.

13.Überblick über numerische Bibliotheken

In diesem Kapitel werden einige wichtige numerische Bibliotheken vorgestellt.

Verwenden von numerischen Bibliotheken

Viele numerische Probleme lassen sich auf elementare numerische Verfahren zurückführen. Falls nicht besondere Gründe für die Entwicklung eigener Prozeduren sprechen, sollten Prozeduren aus kommerziellen Bibliotheken verwendet werden.

Folgende numerischen Bibliotheken stehen u.a. zur Verfügung[14]:

- `libm.a` - Standard Mathematik Bibliothek für C und Fortran
- `libblas.a` - Basic Linear Subroutine Library (BLAS)
- `libessl.a` - IBM Engineering and Scientific Subroutine Library
- `libnagf.a` - NAG Fortran Library
- `libosl.a` - IBM Optimization Subroutine Library

Zordnen der verfügbaren Dateien

Die folgende Tabelle enthält für einige Bibliotheken die Bezeichnungn, den Dateinamen und, falls vorhanden, die zugehörigen C Deklarationsdateien (*C header files*) auf:

[14] Die Aufzählung gilt speziell für das RS/6000 Cluster des Rechenzentrums. Erkundigen Sie sich ggf. bei Ihrem Systemverwalter, welche numerischen Bibliotheken installiert sind.

Bibliothek	Datei	Zugehörige Deklarationen
Standard Mathematik Bibliothek für C und Fortran	/usr/lib/libm.a	/usr/include/math.h /usr/include/errno.h
Basic Linear Subroutine Library	/usr/lib/libblas.a	-
IBM Engineering and Scientific Subroutine Library	/usr/lib/libessl.a	/usr/include/essl.h
NAG Fortran Library	/usr/local/lib/libnagf.a	/usr/local/include/nagf.h

Das Einbinden von C Deklarationsdateien erfolgt in der Übersetzungsphase, wobei eine entsprechende `#include` Direktive in der Quelldatei notwendig ist. Das Einbinden von Bibliotheken erfolgt sowohl für C als auch für Fortran während der Bindephase, also typischerweise in der folgenden Form[15]:

```
$ xlc -o pgm pgm.o /usr/lib/libm.a
  # links pgm.o using external functions from libm.a.
```

Für Bibliotheken, deren Dateinamen die Form `libXXX.a` haben, gibt es zur Vermeidung von Schreibarbeit eine Compiler-Option `-lXXX`:

```
$ xlc -o pgm pgm.o -L/usr/lib -lm
  # links pgm.o using external functions from libm.a.
```

Verwenden von Computeralgebrasystemen

Ferner stehen folgende Computeralgebrasysteme zur Verfügung, mit denen sich auch numerische Probleme bearbeiten lassen:

- **Maple**
- **Mathematica**

[15] Die meisten Compiler kennen die Optionen -Llibpath und -lXXX. Die Option -l setzt voraus, daß die Bibliothek den Namen libXXX.a besitzt.

Interessant sind hierbei insbesondere folgende Möglichkeiten im Bereich der Numerik:

- Durchführen von numerischen Berechnungen mit beliebiger Genauigkeit
- Erzeugen von optimierten C und Fortran Code
- Einfaches Zugreifen auf umfangreiche Bibliotheken math. Funktionen und Verfahren

14.Überblick über Graphik-Bibliotheken und -Programme

In diesem Kapitel wird ein kurzer Überblick über Graphik-Bibliotheken und -Programme gegeben, mit denen numerische Ergebnisse visualisiert werden können. Ferner werden die möglichen Ausgabegeräte aufgelistet.

Verwenden von Graphik-Bibliotheken und -Programmen

Folgende Graphik-Bibliotheken und -Programme stehen zur Verfügung:

Produkt	Kurzbeschreibung	Aufruf
GNU gnuplot	`setup gnuplot` `man gnuplot`	`setup gnuplot` `gnuplot`
IBM Data Explorer	`setup data_explorer` `man dx`	`setup data_explorer` `dx # needs X`
NAG Graphics Library	`setup nag_help` `naghelp`	`setup nag_graphics` `xlf -L/usr/local/lib -lnagg -o pgm ...`
Maple (enthält integrierte Graphik)	`setup maple` `man maple`	`setup maple` `[x]maple`
Mathematica (enthält integrierte Graphik)	`setup mathematica` `man math`	`setup mathematica` `math`
POV-RAY Raytracer	`setup raytracer` `povray`	`setup raytracer` `povray` ***`options`***
XFIG	`setup xfig` `setup -i xfig`	`setup xfig` `xfig`

Drucken und Plotten

Sie können mit dem **`lpr`** Kommando eine Datei mit Graphikinformationen auf einem Drucker bzw. Plotter ausgeben:

```
$ lpr -Pqn fn           # prints file fn to printer queue qn
```

Zum Beispiel wird eine PostScript Datei `sample.ps` mit folgendem Kommando auf dem HP Laserjet III SI (PostScript) des RZ gedruckt:

```
$ lpr -Pps2rz sample.ps
```

Folgende Drucker und Plotter stehen zur Verfügung[16]:

Name (queue)	Typbezeichnung	Format
lprz	IBM 6262	ASCII
pcl2rz	HP Laserjet III Si	PCL 5, DIN A4
pcl1rz	HP DeskJet 500 C	PCL, DIN A4
pl1rz	IBM 6186 DIN A0 Plotter	HPGL
ps1rz	Apple Laserwriter II NTX	PostScript, DIN A4
ps2rz	HP Laserjet III Si PostScript	PostScript, DIN A4
ps2rzdp	HP Laserjet III Si PostScript, doppelseitiger Druck	PostScript, DIN A4

Eine Ausnahme bildet der elektrostatische Plotter VERSATEC CE3425E. Hier ist eine Kombination der Kommandos **vsplot** und **lpr** vorgesehen. Erzeugen Sie zunächst eine Plotdatei *fn* im PPM[17] Format (siehe z.B. **man** xv). Konvertieren Sie die Datei mit dem Kommando **vsplot** in das Versatec Format und plotten Sie im Anschluß mit **lpr**:

```
$ vsplot fn | lpr -Ppunrz
```

Ausgaben können im AVZ, Untergeschoß, Raum B20, abgeholt werden.

[16] Stand zum Zeitpunkt der Drucklegung.

[17] Die Portable Bitmap Tools (PBM) bestehen aus Werkzeugen, um u.a. diverse Graphikformate wie z.B. GIF, TIFF, Targa über ein universelles Austauschformat PPM ineinander umzuwandeln.

Anhang

make Beschreibungs- oder Steuerdatei (Makefile)

```
#--------------------------------------------------------------------#
# Makefile template for C and Fortran programming
# Calling sequence: make -f Makefile or simply make
#--------------------------------------------------------------------#

#--------------------------------------------------------------------#
# Macros:
#--------------------------------------------------------------------#

# Linker flags
LFLAGS=-L/usr/local/lib -lm -lxlf -lblas -lnagf
# Fortran Compiler
F77=xlf
# Full profiling and debugging
FDEBUG=-pg
# Full optimization
FOPTIMIZE=-O3
# Compiler Flags, either FDEBUG or FOPTIMIZE
FFLAGS=$(FDEBUG)
# C Compiler
CC=xlc
# Full debugging and profiling
CDEBUG=-pg
# Full optimization
COPTIMIZE=-O
# Compiler Flags, either CDEBUG or COPTIMIZE
CFLAGS=$(CDEBUG)

#--------------------------------------------------------------------#
# Suffix rules:
#--------------------------------------------------------------------#

.f.o:
	$(F77) $(FFLAGS) -c $<

.f:
	$(F77) -o $@ $(FFLAGS) $(LFLAGS) $<

.c.o:
	$(CC) $(CFLAGS) -c $<

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

#--------------------------------------------------------------------#
# Project specific rules:
# Substitute appropriate values for your project!
#--------------------------------------------------------------------#

PGM=sample1
OBJS=func1.o func2.o
LIBS= ./libll.a ./libhelp.a

$(PGM): $(PGM).o $(LIBS) $OBJS) Makefile
	$(CC) -o $(PGM) $(LDFLAGS) $(PGM).o $(OBJS) $(LIBS)
```