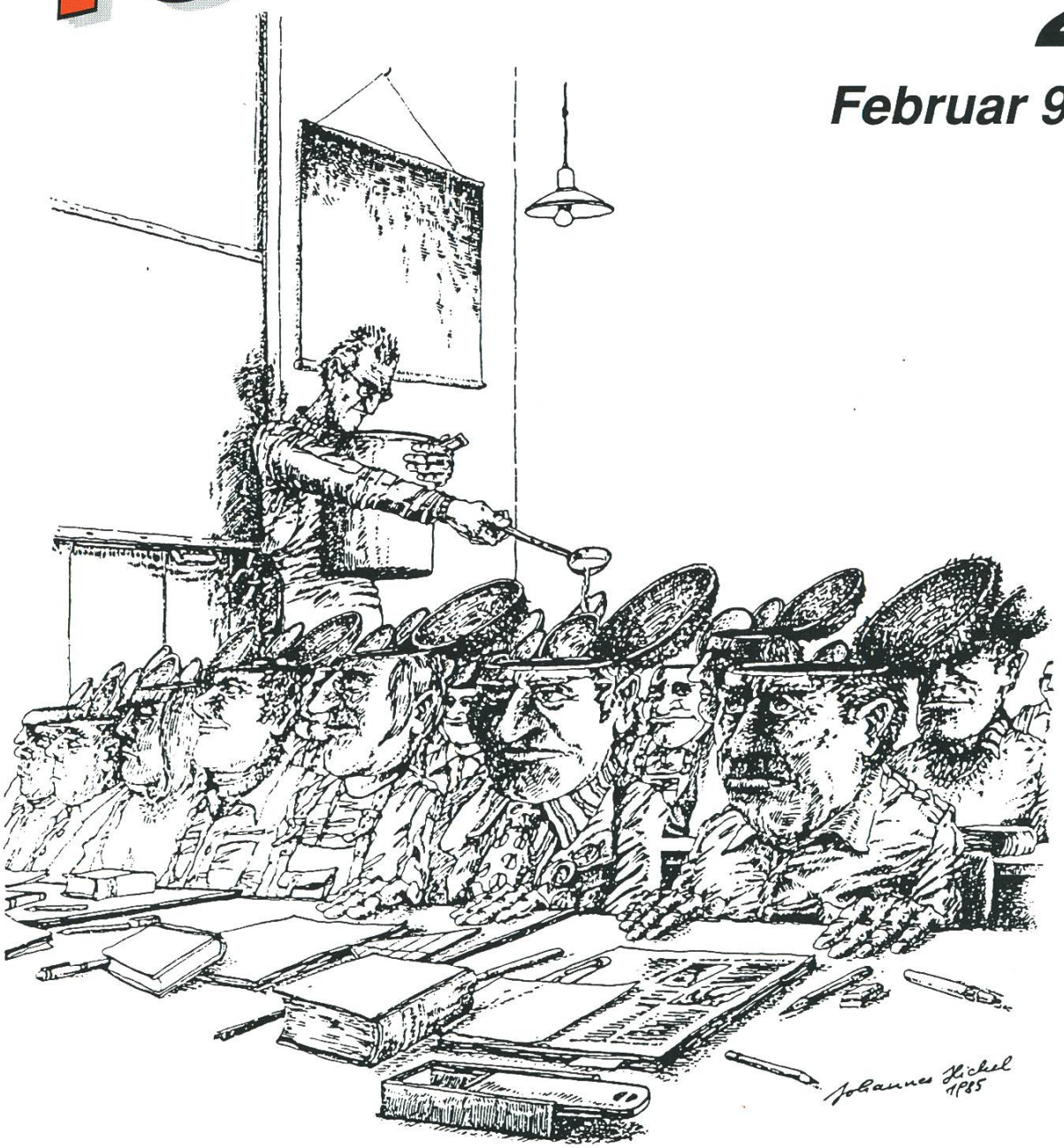
Visionen

2

Februar 94



C-Kurs, zweiter Teil
Die römische Göttin
Swansea-Aufgaben

Vorlesung: UNIX/ C

Visionen Herausgegeben vom Verein der Informatikstudierenden an der ETH Zürich

Adressen

Aktuar: Stefan Rohmer
Keltenstrasse 6, 8044 Zürich
Tel. 01 / 251 34 51
e-mail: stefan@vis.inf.ethz.ch

Exkursionen: Boris Nordenström
Hardstrasse 324, 8005 Zürich
Tel. 01 / 273 24 80
e-mail: banorden@iiic.ethz.ch

Feste & Kultur: Frank Möhle
Dielsdorferstrasse 7, 8155 Niederhasli
Tel. 01 / 851 03 21
e-mail: fmoehle@iiic.ethz.ch

Präsidentin: Grete Danielsen
Dohlenweg 26, 8050 Zürich
Tel 01 / 302 48 97
e-mail: gcdaniel@iiic.ethz.ch

Prüfungen und Unterricht:
Leonhard Jaschke
Südstrasse 67, 8008 Zürich
Tel. 01 / 383 60 55
e-mail: ljaschke@iiic.ethz.ch

Quästor: Daniel Kluge
Irringersteig 3, 8006 Zürich
Tel. 01 / 252 04 14
e-mail: dankluge@iiic.ethz.ch

Redaktor: Patrick Leoni
Sandstr. 6, 8610 Uster
Tel. 01 / 940 05 14
e-mail: pleoni@iiic.ethz.ch

Verleger: Hans Domjan
Kapfhalde 3, 6020 Emmenbrücke
Tel. 041 / 53 68 83
e-mail: hdomjan@vis.inf.ethz.ch

Visinfo(Infosystem): Michel Müller
Rheinländerstr. 15, 4056 Basel
Tel. 061 / 321 81 23
e-mail: mimuelle@iiic.ethz.ch

Impressum

Herausgeber:
Verein der Informatikstudierenden an der ETH Zürich.

Verleger: Hans Domjan
Redaktor: Patrick Leoni

Adresse Verlag & Redaktion:
VIS
Verein der Informatikstudierenden
Haldeneggsteig 4, IFW B29
ETH Zentrum
8092 Zürich

Tel: 01 632 72 12 (Mo-Fr, 12.15-13.00)
Fax: 01 262 39 73
e-mail: vis@iiic.ethz.ch
Postkonto 80-32779-3
Präsenzzeit: Mo-Fr: 12.15-13.00

Auflage: 1400
Inseratepreise:

1 Seite	s/w	SFr. 500.–
1 Seite	Farbe	SFr. 750.–
1/2 Seite	s/w	SFr. 250.–
1/4 Seite	s/w	SFr. 150.–

Redaktions- und Anzeigeschluss für die nächste Ausgabe:
Freitag, 18. Februar 1994

Hei Folkens!

Das Wintersemester ist nun bald zu Ende, was eine gute Gelegenheit ist, es revue passieren zu lassen:

ACM-Programming Contest
Vom VIS schon in den Sommerferien vorbereitet, wurde der Regionalwettbewerb in Zürich zu einem durchschlagenden Erfolg. Die zwei Gewinnerteams schickten wir nach Swansea (England), wo wir den fünften und sechzehnten Platz belegten. Alles in aAllem war dies wohl der bedeutenste Event in diesem Semester.

Die Feten
Das Erstsemestrigenfest kam auch gut an (wenn sich auch nur wenige daran erinnern können), an das Fondue brauche ich wohl niemanden zu erinnern, denn offensichtlich waren ja alle von Euch dabei. Das Jahr beschlossen wir dann mit dem Rocky-X-Mas. In dieser Beziehung also waren unsere Veranstaltungen ein voller Erfolg.

Samichlaus
Unsere Untervereinigung der Weihnachtsmänner und Nikoläuse hatte am 6. Dezember ein kleines, aber bemerkenswertes Auftreten.

VIS-Präsenz in den Vorlesungen
Damit die Schnarchnasen unter Euch auch wissen, wie wir ohne Bart aussehen, haben wir uns entschieden, ab und zu während des Semesters in den Vorlesungen vorbeizuschauen und zu erzählen, was denn momentan gerade los ist. Den Anfang machte Frank (Fetenkaiser), als er den kommenden 10. Geburtstag des VIS ankündigte (mehr dazu in der letzten Ausgabe, S.7).

Kontaktparty
Natürlich sind auch unsere Beziehungen zur Industrie nicht vernachlässigt worden. Am 24. Januar fand für die oberen Semester die Informatik Kontaktparty statt, die auch dieses Jahr ein voller Erfolg war.

Alles in allem also war es ein erfolgreiches Semester. Dies war natürlich nicht ohne die aktive Mithilfe von vielen unter Euch möglich. Unsere aktivsten HelferInnen laden wir zum Dank daher wie immer zum Mitarbeiteressen ein.

Nun zurück in die Gegenwart.
Erstens möchte ich alle am Mittwoch dem 16.Februar zum ordentlichen VIS MV-Apero des WS 93/94 einladen. Wir bitten um rechtzeitiges Erscheinen, da Ihr sonst nichts mehr von den kalten Platten abbekommt. Ausserdem brauchen wir ein neues Vorstandsmitglied, da Patrick (Redaktor) Leoni den Vorstand zwecks Diplomimplementation leider verlassen wird. Wenn Du Lust hast, im Vorstand mitzuarbeiten, dann komme auf jeden Fall zum Apero.

Für die kommende Prüfungssession

organisiert der VIS wieder Lerngruppen. Doch das Echo war nur für das 1.Vordiplom gut - und auch nur auf Seiten der Lernwilligen. Wir brauchen daher noch dringend Tutoren, die bereit sind ihr Wissen an andere weiterzuvermitteln. Für diejenigen unter Euch, die anhand alter Vordiplome lernen (getreu nach Survival Guide), eine gute Nachricht: Ab sofort sind die Musterlösungen im VIS-Büro erhältlich.

In den Ferien ist das VIS-Büro jeden Mittwoch von 17-19 Uhr besetzt, wo ihr Vordiplome etc. beziehen könnt, wenn Ihr im Semester nicht mehr dazu gekommen seid.

Ich wünsche allen schöne Semesterferien - jenen, die am Lernen, im Praktikum oder im Militär sind wünsche ich noch viel Erfolg.

Ha en fin ferie, og lykke til med prøvene!

Grete

"Elephants"

President Clinton, as part of his goal to increase technical awareness and interest in the sciences, asked the various major computer companies to cooperate in a large Multimedia publishing project. The general theme was "Elephants".

The piece from Apple was titled: "User Friendly Elephants and Their Friend, the Mouse".

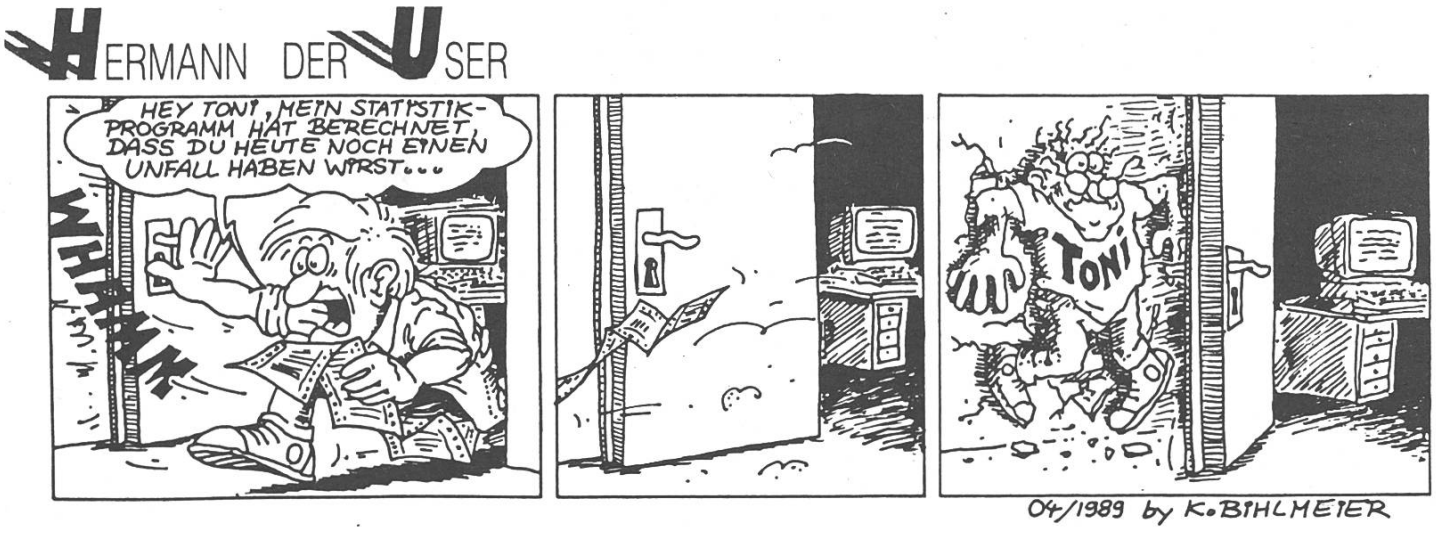
IBM's: "How to Sell an Elephant to Someone Who Wants a Racehorse".

Novell's: "Connecting Elephants".

Borland's: "All Elephants Should Cost $99".

NeXT's: "Painting an Elephant Black".

Microsoft's: "Why You Should Buy Microsoft Windows".



C-Kurs Teil 2 – Further on up the road

Von Leonhard Jaschke und Patrick Leoni

Willkommen, liebe Leserin, lieber Leser, zur zweiten Folge unseres visionären C-Kurses. Die vor Monatsfrist erschienene erste Etappe wurde, so kam uns zu Ohren, gelesen, gelobt und kritisiert. Vor allem aber wurden wir auf Fehler aufmerksam gemacht. Da kann man nur sagen: "Sorry!", und ansonsten auf eine Visionenausgabe von vor ein paar Jahren verweisen, wo die folgende profunde Weisheit zu lesen war:

1. Ein Programm hat immer mindestens einen Fehler.
2. Ist dieser gefunden und beseitigt, so ist es immer noch ein Programm.

Dies gilt, das ist empirisch bewiesen, nicht nur für Programme, es ist auch auf C-Kurse anwendbar. Da ausserdem der Debugger erst dieses Mal erwähnt wird, konnten wir ihn noch nicht über unseren Text drüberlassen.

Schreiten wir also zur Korrigenda:
1. Auf Seite 28 stand ein *Nono*, wir fielen genau in jene Falle, vor der wir Euch LeserInnen so warnten: Das "=/=="-Syndrom hat uns doch noch gepackt. In der if-Anweisung

```
if ((strcmp(Person,"Mann") =
TRUE)) SchenkeRosen;
```

die wir, nebenbei, eh als falsch bezeichneten, bräuchte es natürlich für ein regelrechtes, gemeintes, nämlich *falsches* Verhalten den Vergleichsoperator == statt des einfachen =. Sonst reklamiert schon der Compiler, und zwar mit dem vergleichsweise nichtssagenden

```
fehler.c:42: invalid lvalue
in assignment,
```

denn ein = führt in C ja eine Zuweisung aus. Was ein *lvalue* ist? Lest in K&R nach, und staunt selbst...

Themen der Dezember-Ausgabe

- C: Einleitung und Prinzip
- Elementare Datentypen
- Variablendefinition
- Arrays
- Konstanten
- Zuweisungen, Ausdruecke
- Kontrollstrukturen
- Bloecke
- Konsolenausgabe – printf
- Das Hauptprogramm
- Compilieren

Themen dieser Ausgabe

- Der Debugger gdb
- if genauer erklaert
- break und continue
- Funktionen und Prozeduren
- Funktionsparameter
- Rueckgabe von Werten
- Der Praeprozessor: include und define
- Konsoleneingabe – scanf
- Zeichenketten (Strings) und Stringfunktionen
- Strings als Parameter
- Files

Im Übrigen hat uns ein findiger Kopf den folgenden Tip übermittelt:

In Vergleichen Variable-Konstante sollte man die Konstante zuerst schreiben und erst nach dem == die Variable. Bedingungen sehen dann aus wie folgt:

```
if (132 == IQ) BrightBoy;
```

Dann ist man gefeit vor der =/== –Trapdoor. Falls man nämlich wieder einmal versehentlich nur *ein* Gleichheitszeichen hinschreibt, beschwert sich der Compiler auf jeden Fall. Denn eine Konstante ist garantiert kein l-Wert.

2. Zwei Seiten vorher wird die Sprachschöpfung aus dem RZ H-Stock als "Operon" bezeichnet. Gemeint ist natürlich "Oberon". Ein Freudscher: Die Oberon-Projekte dürfen fürwahr als "Werke" (lat. "Opera") bezeichnet werden!

Debugger

Nun geht's zur Sache: Beginnen wir mit einer kurzen Besprechung des Debuggers *gdb*. Das heisst soviel wie *GNU-Debugger* und der ist, genauso wie gcc, auf unseren Sparcs verfügbar. So komfortabel wie derjenige etwa einer Borlandsprache auf dem PC arbeitet er keineswegs, sein Erscheinungsbild ist eher als nüchtern einzustufen. Nichts von "mit-der-Maus-sich-Records-entlanghangeln". Dafür kann gdb sonst sehr viel, und wer ihn zu bedienen weiss, kann mit ihm ein Problem in seinen (oder fremden) Programmen rasant einkreisen und festnageln.

Der Debugger verfügt über ziemlich viele Kommandi, mittels derer man ein Programm laden, starten, schritt- und abschnittweise abarbeiten, scheitern lassen, beeinflussen, disassemblieren und anhalten kann. Seine Variablen lassen sich einsehen, verändern und als Bedingungen gebrauchen, seine Typen untersuchen und seine Funktionen in ihrem Zusammenspiel beobachten. Ob der Fülle von Möglichkeiten übersieht man leicht, dass die Kenntnis von vielleicht 10 Kommandi ausreicht, um 90 % der Funktionalität von gdb nutzen zu können (Die bekannte 80-20-Regel trifft auch hier zu, sogar in erhöhtem Ausmass). Die Bedienung von gdb ist sehr einfach, die eingebaute help-Funktion hilft (dem Anfänger jedenfalls) immer. Was wir hier (vorläufig?) *nicht* besprechen ist das Zusammenspiel von gdb und dem Editor emacs. Damit wäre ein integriertes Arbeiten mit Editor, Compiler und Debugger möglich.

gdb

gdb ist kommandozeilengesteuert. Breakpoints[1] werden etwa mittels *break 69* gesetzt. Diese Kommandi können abgekürzt werden, "in br" heisst gleichviel wie "info breakpoints". Die Regel fürs Abkürzen ist einfach die, dass die Angaben eindeutig sein müssen, und darum hat sich jeder selbst zu kümmern.

Um ein Programm so zu compilieren, dass es von gdb bearbeitet werden kann, muss man es mit der Option *-g* übersetzen. Der Compileraufruf fürs Programm vom letzten Mal lautet also:

```
cc -g sort.c -o sort .
```

---

1 Ich behandle das Wort "Breakpoint" fortan so, als ob es deutsch wäre. Ein ähnlich prägnanter deutscher Ausdruck (der auch noch so verbreitet ist wie "Breakpoint") ist mir noch nicht untergekommen. *Andere* Puristen der deutschen Sprache mögen mir verzeihen, *ich* werde mir nie verzeihen... pal

gdb kann mittels

```
gdb sort
```

auf das erzeugte Programm angesetzt werden. Nun ist alles zur zünftigen debugging-session parat.

Die allerwichtigsten Kommandi, auf die gdb hört, seien hier in aller Kürze aufgelistet:

a) Programm ablaufen lassen

show args	Zeigt die ans Programm übergebenen Kommandozeilenargumente
set args	Setzt neue Argumente, die dem Programm beim Start übergeben werden
kill, ki	Hält die Ausführung der gerade bearbeiteten Programmes an
run, r	Startet das Programm
continue, c	Startet das Programm nach einem Unterbruch (Breakpoint o.ä.) neu
jump, j	Setzt die Ausführung des Programmes an der angegebenen Zeilennummer fort
until, u	Arbeitet das Programm ab, bis es eine grössere als die aktuelle Zeilennummer erreicht
step, s	Startet das Programm, und stoppt beim Erreichen einer anderen als der aktuellen Zeile Quellcode
next, n	Wie step, aber Funktionsaufrufe werden als eine einzige Anweisung behandelt
finish, fin	Beendet die aktuelle Funktion und hält dann die Ausführung an

b) Stack kontrollieren

Hier geht es um lokale Variablen und verschachtelte Funktionsaufrufe.

backtrace, bt	Zeigt alle Stackrahmen an, bis zu main. (Versteht ihr nicht? Mussen si gerade nich weinen, ausprobieren!)
return, ret	Beendet die aktuelle Funktion, ohne (!) den noch übrigbleibenden Code abzuarbeiten

c) Daten inspizieren

whatis, what	Zeigt den Datentyp des übergebenen Ausdrucks an
print, p	Zeigt den Wert des übergebenen Ausdrucks an
set	Setzt eine Variable auf einen neuen Wert
display, disp	Zeigt jedesmal, wenn das Programm anhält, den Wert eines Ausdrucks an
undisplay	Löscht diese Ausgabe
disable display;	
enable display	suspendiert / reaktiviert die Evaluation und Ausgabe eines Ausdrucks

d) Breakpoints

watch, wa	Setzt "Watchpoint"; das Programm stoppt dann, sobald der angegebene Ausdruck seinen Wert ändert
break, b	Setzt einen Breakpoint an die angegebene Zeile oder an den Start der angegebenen Funktion
clear, cl	Löscht den Breakpoint an der angegebenen Zeile
delete, del	Löscht den Breakpoint mit der angegebenen Nummer

condition, cond Ändert die Funktionsweise des angegebenen Breakpoints (Nummer), indem er den Ablauf nur noch bei Zutreffen der angegebenen Bedingung stoppt

ignore, ign Lässt den Breakpoint erst nach der angegebenen Anzahl Passagen aktiv werden

e) "Miscellaneous"

list, l Zeigt den Quelltext des gerade bearbeiteten Programms an, portionenweise. Die Zeilen sind versehen mit Zeilennummern (zu gebrauchen für break)

info, in Zeigt diverse Informationen über das bearbeitete Programm an. Dieses Kommando ist eines der wichtigsten von gdb. Falls Ihr vergessen habt, an welchen Orten Ihr Breakpoints gesetzt, welche Variablen-Displays ihr eingestellt und in welchen Stackrahmen ihr 'rumgepfuscht habt, hilft Euch genau dieses Kommando weiter. Nützliche Informationen liefern unter anderem die folgenden Kommandi:

info display Zeigt alle displays an

info break Zeigt alle Breakpoints an

info variables Zeigt eine Liste *aller* Variablen an

quit Beendet gdb

Nicht erschrecken. Wir lernen hier C, nicht gdb. Aber die Rosinen könnt Ihr schon aus obiger Liste pflücken...

Was nun kommt, wäre die Anwendung des Gelernten. Wenn ihr Lust habt, könnt Ihr versuchen, unser Sortierprogramm vom letzten Mal zu "debuggen" (vielleicht findet Ihr einen "Bug" – es hat sicher noch einen...). Ihr merkt dabei, dass die Bedienung von gdb schnell in die Finger geht. Am Ende dieser Folge werden wir ein Programm vorstellen, das ihr dann durch gdb bearbeiten lassen könnt.

Kontrollstrukturen – genauer betrachtet.

In diesem Abschnitt wollen wir die Schleifen etwas genauer betrachten. In der Übersicht vom letzten Mal fehlte die Möglichkeit, Schleifen vorzeitig zu verlassen (in Oberon gibt es dafür das EXIT), sowie die, ohne grosses Gefackel an den Schluss eines Schleifenrumpfs zu springen (in Oberon *muss* man dann eben fakkeln). Zunächst aber noch einmal zum IF bzw. if.

if vs. IF

In Pascal besteht ja die Möglichkeit, nach dem THEN eine einzige Anweisung anzugeben, auf die dann mit ein Semikolon folgt. Dieses Semikolon trennt dann nachfolgende Anweisungen nicht nur von der abhängigen Anweisung sondern vom ganzen IF-Statement. Ein Beispiel:

(Pascal:)

```
IF sem = 0 THEN INC(sem);[2]
Race := TRUE;
```

Die Variable Race wird hier auf TRUE gesetzt, egal ob die Bedingung wahr war oder nicht. Das IF-Statement ist nach ...(sem); abgeschlossen.

---

2 Das ist nur ein Beispiel; kommt ja nicht auf die Idee, Semaphoren *so* zu implementieren...

(Oberon:)

```
IF Sem = 0 THEN INC(Sem) END;
Race := TRUE;
```

In Oberon ist das END für die gewünschte Verhaltensweise lebenswichtig, denn das THEN leitet ja hier einen neuen Block ein. Diese Tatsache ist Quelle einiger Umgewöhnungsprobleme von Pascal nach Modula und Oberon, aber man lernt sie schätzen – spätestens dann, wenn sie einen vor einer Falltür bewahrt hat.

Falls in Pascal *mehrere* Anweisungen von einer Bedingung abhängen, muss man sie mittels BEGIN und END einkapseln.

(Pascal:)

```
IF Health = Flu THEN BEGIN
   Drug := HotTea;
   Clothes := Pyjama
END;
```

In Oberon lässt man das BEGIN weg, da THEN bereits einen neuen Block einleitet.

C ist in dieser Hinsicht wie Pascal. Die Blockbegrenzer { und } sind nur nötig, falls *mehrere* Anweisungen vom if abhängen, sonst können sie weggelassen werden. Die obigen Anweisungsfolgen lauten also in C:

```
if (sem == 0) sem++;
race= 1;
```

bzw.

```
if (health == flu) {
   drug = hot_tea;
   clothes = pyjama;
}
```

Die geschweiften Klammern im ersten Fall wegzulassen ist ein Akt der Bequemlichkeit, und jeder muss selbst wissen, ob er das tun will. Wir raten dazu, *in aller Regel* diese Klammerung vorzunehmen. Zum Problem wird diese Sache nämlich beim gefürchteten else-Konflikt. In

```
if (n == 0)
   if (m == 0) printf ("n and
                  m zero\n");
else printf("n nonzero\n");
```

gehört das else nicht, wie ich es durch den String und die Einrückung angedeutet hatte, zum ersten sondern zum zweiten if. Falls also "n nonzero" ausgedruckt wird, so ist das falsch, denn dies heisst nur, dass *m* != 0 ist.

Umgehen kann man das, indem man doch wieder Klammern setzt:

```
if (n == 0) {
   if (m == 0) printf ("n and
                  m zero\n");
}
else printf("n nonzero\n");
```

Die Einrückung allein lässt den Compiler natürlich eiskalt. Die Regel hierzu lautet: Innerhalb eines Blockes gehört ein else immer zum letzten if, zu dem noch kein else existiert.

Da übrigens Bedingungen normale arithmetische Ausdrücke sind, kann ein Vergleich wie

```
if (n == 0) dummy++;
```

abgekürzt werden zu

```
if (!n) dummy++;
```

Besonders guter Stil ist das allerdings nicht, meist sagt die epischere erste Form genauer aus, was der oder die Programmierende wollte.

break und continue

Kommen wir nun zu *break* – das übrigens nicht mit der gleichnamigen Debuggeranweisung verwechselt werden sollte. Diese Anweisung ermöglicht es, eine Schleife sofort zu verlassen. In

```
for (i=0; i<20; i++) {
   image = before;
   break;
   image = after;
}
```

wird die Schleife beim break verlassen, und *image* bleibt *before*. Die Variable i enthält eine *Null*.

Die Anweisung *continue* springt ebenfalls ans Ende der Schleife, verlässt sie aber nicht. Sie wird normal weiter ausgeführt. Ersetzen wir im obigen Programmfragment "break" durch "continue", so ist zwar in *image* nachher auch *before* gespeichert, aber i beträgt immerhin 20.

break und continue funktionieren bei den Kontrollstrukturen *while*, *for* und *do–while*, break darüber hinaus noch bei *switch*. Letzteres hat eine besondere Bedeutung, wie ich es das letzte Mal erwähnte. Jede Alternative eines switch-Statements muss mit einem break-Statement abgeschlossen sein. Die Alternativen werden sonst einfach linear durchlaufen, da die case-Marken nur Textmarkierungen sind. Brauchbar ist dieses Verhalten eigentlich nur bei für mehrere Alternativen gültigen Anweisungsfolgen. Hier zeigt Oberon, wie man's richtig macht.

Eine Kontrollstruktur haben wir noch "vergessen" – es handelt sich um das verpönte *goto*. In C existiert diese Sprunganweisung durchaus, führt aber ein zu Recht karges Dasein. K&R meint dazu:

> C verfügt über eine beliebig zu missbrauchende goto-Anweisung. Formal ist goto nie notwendig...

Das meinen wir auch. Wer will, kann selbst nachlesen, wie goto funktioniert. Für Spaghetticode übernimmt der VIS dann aber keine Verantwortung...

Funktionen und Prozeduren

Wie in Oberon (und Freunde) gibt es auch in C Prozeduren und Funktionen, im Gegensatz dazu ist allerdings die Funktionsdefinition die Norm und die Prozedurdefinition die Ausnahme. Genauer gesagt werden in C Prozeduren als Funktionen ohne Rückgabewert deklariert. Auch die Darstellungsform ist natürlich grundlegend anders. Genug der Worte, lassen wir ihnen Daten folgen: Eine Funktionsdefinition hat in C folgenden Aufbau:

```
[static] <Typ> <F_name>
   ( <Parameterliste> )
{
   Statementsequence;
}
```

Das Schlüsselwort *static* teilt dem Compiler mit, dass die Funktion NICHT exportiert werden soll. Im Gegensatz zu den wirthschen Sprachen, in denen per Default alle Funktionen und Prozeduren lokal definiert sind, ist der Normalfall in C, dass Funktionen exportiert werden. Ich rate dringendst dazu, alle lokalen Funktionen mit *static* zu bezeichnen: Es kommt oft genug vor, dass nicht dokumentierte Funktionen, die fälschlicherweise exportiert wurden, in anderen Programmen (wegen Namensgleichheiten) grosse Verwirrung stiften.

Die Typbezeichnung sagt uns, was für einen Wert die Funktion zurückliefert. Sie kann also *char, int, long* ... sein (Beispiele kommen gleich weiter unten), das Schlüsselwort für eine Funktion ohne Rückgabewert (= Prozedur) ist *void*. Zur Beachtung: Die Typbezeichnung kann auch weggelassen werden, dann ist der Rückgabewert automatisch vom Typ integer. Viele C-Programmierer machen das auch bei Prozeduren. Da man dort den Rückgabewert sowieso nicht beachtet, ist es ja egal. DAS IST HUNDSMISERABLER STIL! Am besten lässt man sich überhaupt nicht darauf ein, eine Funktion bzw. Prozedur ohne Typbezeichnung zu deklarieren --- das macht ein C-Programm schon viel lesbarer, da der Leser bzw. die Leserin dann nie im Unklaren darüber gelassen wird ob die Funktion einen Rückgabewert hat oder nicht.

Bevor ich noch auf die Parameterliste zu sprechen komme, möchte ich zur Illustration einige (fiktive) Beispiele für (parameterlose) Funktionsköpfe geben:

- `long time()`
  eine Funktion, die uns die aktuelle Zeit zurückgibt.

- `float pi()`
  eine Funktion, die pi auf eine bestimmte Anzahl Stellen berechnet.

- `static void initialize()`
  eine (interne!) parameterlose Prozedur, die globale Variablen initialisiert.

- `void ClearScreen()`
  eine (exportierte!) parameterlose Prozedur, die den Bildschirm löscht.

Die Eingabeparameter der Parameterliste sehen aus wie normale Typendeklarationen, ohne Semikolon, durch Kommata getrennt.

Beispiele:

```
· float sin(float x)
· void output(int x, int y, float
  pi)
```

Bei den Ausgabeparametern muss ich auf ein Thema vorgreifen, das erst in der nächsten Folge behandelt werden wird: Pointer. In C gibt es keine VAR-Parameter! Sie werden mit Hilfe von Pointervariablen (wie es im Hintergrund der Oberon-Compiler macht) ausprogrammiert: Wir geben statt der Variablen ihre Adresse an, die Prozedur (bzw. Funktion) kann in diese Adresse ihr Resultat schreiben und über sie gelangt dieser Wert zur aufrufenden Prozedur zurück. Die aufgerufene Funktion muss sich ständig im Klaren darüber sein, dass diese Variable eine Pointervariable ist (und dementsprechend anders behandelt wird). Pointervariablen werden in der Parameterliste mit einem Stern gekennzeichnet:

```
void input(int *a)
```

In dieser Deklaration wird das a durch den Stern als Pointer deklariert. Die Prozedur liefert die Ganzzahl a zurück. Innerhalb der Prozedur spricht man den Inhalt von a (also dessen Wert) mit **a*) an. Mit *a* erhält man die Adresse (diese ist allerdings nicht vom Typ int sondern in dieser Folge noch vom geheimnisvollen Typ *address*). Der Stern ist der Inhaltsoperator auf einen Pointer, vergleichbar mit dem Referenzierungsoperator(^) in Oberon. Aufgrund der Terminologie erwarten wir einen Adressoperator auf eine *normale* Variable: Und hier ist er: `&` liefert die Adresse einer gewöhnlichen Variable. (Parallele in Oberon: SYSTEM .ADR)

Ein Beispiel zur Illustration:

```
int normal;
int *pointer;

pointer= &normal; /* (1) */
*pointer= normal; /* (2) */
```

In (1) wir der Pointer *pointer* auf die Variable *normal* gesetzt. (Das heisst er zeigt dorthin). In (2) wird der Inhalt von *normal* an die Adresse von *pointer* kopiert.

Als kleines Beispiel möchte ich hier ein Programm implementieren, das eine Funktion aufruft, die einen Eingabeparameter x quadriert und das Resultat im Ausgabeparameter y speichert:

```
#include <stdio.h>

void quadr(int x, int *y)
{
   *y= x*x;
}

void main(int argc, char
*arv[])
{
   int x,y;

   x= 4;
   quadr(x,&y);
   printf("%d\n",y);
}
```

Anhand dieses Beispiels hoffe ich, die Vorgangsweise mit den Pointern als Ausgabeparametern einer Prozedur verständlich gemacht zu haben.

Bei den Parametern kann nach der ANSI-Norm (welche ihrerseits seit ein paar Wochen ISO-Norm genannt wird...) ein *const* vorangestellt werden, wie z. B. hier:

```
void strcpy(char *a, const
char *b)
```

Damit teilt man dem Compiler mit, dass das Character-Array(!) b nicht verändert werden darf, während das Character-Array a beliebigen Veränderungen unterworfen sein darf.

Der Wert einer Funktion wird mit *return* zurückgegeben. Wenn also jemand Lust hat, die Funktion

```
float sin(float x)
```

neu zu schreiben, dann wird er sicherlich eine Variable *result* (oder so ähnlich) benötigen. Am Ende des Anweisungsblocks muss er sie dann mit *return(result)* zurückgeben – genau wie in Oberon. Achtung: Weder Compiler noch Laufzeitsystem überprüfen die Rückgabe eines Wertes. Wer das *return* vergisst, wird eventuell lange nach diesem Fehler suchen.

Der Präprozessor – #define #define again

Wir kommen nun zur Behandlung des Präprozessors[3]. Der ganze dem Compiler übergebene Quelltext wird vor der eigentlichen Übersetzung durch ihn hindurchgeschleust. Während dieser Vorverarbeitung werden gewisse Konstrukte aufgelöst, die nicht zur eigentlichen Sprache gehören, damit der Compiler danach einen sauberen Code vorfindet. Beispiele solcher Konstrukte haben wir bereits gesehen: #define und #include sind Präprozessorbefehle.

Jede Präprozessoranweisung beginnt mit einem # (Gartenhag, Fis, Nummernzeichen, ...). Ich bespreche hier nur die wichtigsten Anwendungen der haupt-

[3] K&R sagt dazu "Preprozessor", aber das wäre ja wie "Breakpunkt" oder "Brechpoint" – ein grauslicher Sprachhybrid.

sächlichen Kommandi (viele sind es sowieso nicht):

#define und #undef

Die Anweisung #define dient dazu, Konstanten zu vereinbaren, wie wir es schon in der letzten Folge gesehen haben. Nach einem

```
#define LEFT "Nice"
```

wird jedes Vorkommen von LEFT im Programm wird dann durch den konstanten String "Nice" (inkl. Anführungszeichen) ersetzt. Dies funktioniert auch mit Zahlen:

```
#define LUCKY 13.
```

Dieses Merkmal ist noch für andere Sachen gut: Nach

```
#define FOREVER for(;;)
```

kann eine Endlosschleife durch einfaches Angeben von "FOREVER" eingeleitet werden. So eine Schleife ist manchmal sehr nützlich, wenn das Abbruchkriterium nicht so einfach in der Form einer einzelnen Bedingung angegeben werden kann. Es entspricht dann durchaus brauchbarem Stil (und ist auch Usus), die Schleife durch ein break zu verlassen.

Mittels

```
#undef LEFT
```

lässt sich die obige Definition wieder löschen und ein Vorkommen von "LEFT" im Programmtext wird vom Präprozessor nicht mehr beeinflusst.

Genauso wie sich in Programmen Anweisungen abhängig von gewissen Bedingungen ausführen lassen (oder eben nicht) – Stichwort if – lassen sich gewisse Teile des Quelltextes in Abhängigkeit von gewissen Bedingungen zur Compilation zulassen oder davon ausschliessen. Das Zauberwort hierfür lautet #ifdef. Falls der Name hinter #ifdef definiert ist, so wird der nachfolgende Quelltext übersetzt, ansonsten wird er ignoriert. Die dazugehörige #else- oder #endif-Anweisung wird gesucht.

Ein Beispiel:

```
#define AUTO 1
#ifdef  AUTO
printf ("Auto ist definiert
worden");
#else
printf("Auto ist
undefiniert");
#endif
#undef  AUTO
#ifndef AUTO
printf ("Auto ist nimmer
definiert");
#endif
```

Die Bedingung wird nicht erst zur Laufzeit des Programmes geprüft. Der Präprozessor sorgt dafür, dass der Compiler wirklich nur die beiden Zeilen

```
printf ("Auto ist definiert
worden");
printf ("Auto ist nimmer
definiert");
```

zur Übersetzung zugespielt bekommt. En passant eingeführt wurde hiermit die Anweisung #ifndef, die überprüft, ob ein gewisser Name *nicht* definiert ist.

⇨ Seite 28

Diplomarbeiten, Semesterarbeiten und bezahlte Mitarbeit in der Fachgruppe Datenbanken

Im folgenden möchten wir Euch ein paar Projekte vorstellen, die bei uns in der Fachgruppe Datenbanken bei Professor Schek durchgeführt werden, und für welche immer wieder interessante Diplom- oder Semesterarbeiten und bezahlte Mitarbeit angeboten werden.

Abbildung eines Objektmodels auf ein paralleles Relationales Datenbanksystem

Im Rahmen des Nationalfonds-Projektes "Hybride Wissensbanksysteme" wird das Objektmodell COCOON auf ein paralleles Relationales Datenbanksystem als Speichersystem abgebildet und untersucht, wie weit Parallelisierung die effizientere Ausführung von Anfragen und Änderungen ermöglicht. Andere Aspekte des Projekts werden von Dr. Reimer von der Informatikforschungsabteilung der Rentenanstalt und Professor Marti aus dem Institut für Informationssysteme durchgeführt.

Ausgehend von der Annahme, dass normalerweise viel mehr Anfragen als Änderungsaufträge gemacht werden, wählten wir den folgenden Ansatz: Um Anfragen effizient – das heisst mit möglichst wenigen Joins – durchzuführen, werden die Daten stark repliziert gespeichert. Dies bedeutet natürlich einen höheren Aufwand bei Änderungsoperationen, welche auf Grund der möglichen Sperrkonflikte die Anfrage-Performance wiederum negativ beeinflussen können. Daher werden die Änderungsoperationen mittels des Mehrschichten- Transaktionsansatzes parallelisiert, um einerseits die Daten bei Änderungen effizient nachzuführen und konsistent zu halten und andererseits Sperren möglichst kurz zu halten.

Das Ziel ist es, einen Prototypen zu entwickeln, der COCOON wie oben beschrieben abbildet. Im Laufe dieser Arbeit bieten sich eine Fülle von interessanten Aufgaben auf den Gebieten der Implementation und Performance- Messungen an.

Falls Ihr Euch für eine Form der Mitarbeit an diesem Projekt interessiert, möchte ich Euch bitten, sich mit Michael Rys (IFW C41.2, rys@inf.ethz.ch) in Verbindung zu setzen.

The CONCERT Abstract-Object Kernel

CONCERT is a prototype database system being designed and developed within our group. CONCERT is a kernel database system tailored to sup-

porting highly application-area specific classes of data, in heterogeneous application development environments. The objects managed by the kernel are termed abstract, since they are managed directly in terms of their foreign, application-area defined representations. In addition, the kernel is open to the exploitation of foreign services in the management of such abstract objects.

Since objects are abstract and their representations are unknown to the kernel, an alternative approach to their modelling within the kernel must be adopted. The CONCERT approach is to model abstract objects in terms of small number of potentially-foreign operations implementing abstractions which are known to the kernel. For example, a B-tree can be applied to any objects for which an ordering abstraction exists, or a grid-file to any objects for which a (say, two-dimensional) spatial abstraction exists. These abstractions can be defined through the provision of a small number of operations, and these operations may be implemented by either an application system itself, or by some foreign service provider. While managing foreign objects as abstract data types, as described above, has been proposed before, the CONCERT approach is novel for a kernel storage manager since it directly supports data independence.

In order to facilitate the application of foreign code against stored abstract-objects, the CONCERT kernel must provide a mechanism whereby access to abstract objects can be shared, and whereby clients operate against a uniform view of such objects that is, the view required by foreign operations. Since abstract objects are potentially-large and of varying sizes, a traditional approach to buffer management is inapplicable. Hence an approach to buffer management has been developed which is well-suited to the application of foreign operations against buffered abstract objects. The approach is based on operating system-supported memory mapping, and provides clients with uniform addressability of individual buffered abstract objects.

Please check with Stephen Blott (IFW C43.1), Lukas Relly (IFW C41.1) or Helmut Kaufmann (IFW C41.1) for details.

Integration von Datenbanken und CIM-Subsystemen

Das Projekt "Integration von Datenbanken und CIM-Subsystemen" untersucht, wie sich verschiedenartige Informationssysteme integrieren lassen. Als zu untersuchendes Beispiel dienen Applikationen aus dem Bereich Computer Integrated Manufacturing (CIM), also CAD-, CAM-, PPS-Applikationen. Das Projekt wird vom KWF und von der Industrie (ABB, Sulzer) gefördert. Weiter beteiligt ist das Institut für Konstruktion

und Bauweisen der ETH, die CIM-Gruppe von Prof. Flemming.

Unser Institut interessiert sich vor allem für den Aspekt, wie konsistent und systemübergreifend die Daten verwaltet werden können. Wir gehen dabei davon aus, dass die einzelnen Subsysteme jeweils bereits eine eigene Datenverwaltung besitzen, die teilweise auf Datenbanktechnologie basieren, in manchen Fällen aber auch völlig anders aussehen können. Es gibt zum Beispiel CAD-Systeme, bei denen die Konstruktionsdaten in einfachen Dateien (Files) verwaltet werden, ohne Zugriffskontrolle und ohne Schutz vor Datenverlust oder Manipulation.

In einem ersten Schritt haben wir einige der global (also für verschiedene Systeme) relevanten Daten modelliert. Wir wollen nun Datenbanktechnologie dafür einsetzen, diese Daten lokal zu verwalten und dennoch globale systemübergreifende Integritätsbedingungen sicherzustellen. Wir bauen für Beispielapplikationen einen Prototyp, an dem wir unsere Konzepte demonstrieren können. Bereits sind einige Arbeiten im Bereich föderierte Transaktionsverwaltung gelaufen, wo es darum geht, globale Transaktionen mit Subtransaktionen auf den jeweiligen CIM-Systemen zu realisieren.
Als Beispiel für in diesem Projekt ausgegebene Diplomarbeiten sind die Arbeiten "Föderierte Transaktionsverwaltung aufgesetzt auf relationalen Datenbanksystemen" von Otto Mayer und Hanspeter Waldegger zu erwähnen. Otto hat einen Lock-Manager entworfen und gebaut, der zwischen lokalen und globalen Transaktionen unterscheiden und diese passend unterstützen kann. Zugeschnitten auf die verwendete Sprache SQL werden Sperren alloziert und gezielt freigegeben. Hanspeter hat einen Query-Analysator (einen Scanner und Parser) für SQL gebaut, welcher die für den Lock- und für den Recovery-Manager benötigte Informationen aus den Queries extrahiert. Beide Arbeiten wurden übrigens mit dem ersten Preis der NCR-Stiftung 1993 ausgezeichnet. Momentan laufen Diplomarbeiten im Bereich Performance-Messung und die Entwicklung eines Gateways für ein CAD-System.

Geplant sind unter anderem weitere Arbeiten im Bereich *Transaktionsverwaltung* und *Kontrolle und Verwaltung systemübergreifender Konsistenzregeln.* Wer sich für dieses Projekt interessiert, kann weitere Informationen bei Werner Schaad (IFW C43.2) oder Martin Wunderli (IFW C47.1) bekommen.

Werner Schaad,
Institut für Informationssysteme

Einladung

zur

VIS-MV

am

Mittwoch, 16. Februar 1994

18:00 h

im StuZ

(mit anschliessendem Apéro)

Traktanden:

1. Begrüssung
2. Wahl der StimmenzählerInnen
3. Wahl der Protokollführerin/des Protokollführers
4. Änderung und Genehmigung des letzten Protokolls
5. Änderung und Genehmigung der Traktandenliste
6. Rechnung und Budget, Entlastung des Vorstandes
7. Mitteilungen
   a) des Vorstandes
   b) aus der AK/UK
   c) aus dem DC
   d) aus den Kommissionen
   e) der Mitglieder
8. Bestätigung der Kommissionen
9. Wahl des Präsidenten/der Präsidentin
10. Wahl des Vorstandes
11. Wahlen AK/UK, DC und RechnungsrevisorInnen
12. 10 Jahre VIS – Megafete
13. Hardware
14. Resolutionen
15. Varia

Praktikum bei der Schweizerischen Kreditanstalt

Mein Praktikum absolvierte ich in der Schweizerischen Kreditanstalt. Grund dazu war, dass ich bis dahin nur die Industrie gekannt hatte: Bevor mein Studium begann, hatte ich fast ein Jahr lang bei der Alcatel STR AG gearbeitet. Auch während des Studiums konnte ich dort noch teilweise arbeiten. Darum wollte ich auch mal eine andere Branche kennenlernen. So entschloss ich mich schon früh, bei einer Bank das obligatorische Praktikum zu absolvieren. An der Kontaktparty für Informatiker konnte ich dann auch gleich den Bewerbebogen ausfüllen. Zwei Monate später stellte ich mich in einem Gespräch vor. Kurze Zeit danach war der Arbeitsvertrag unterschrieben.

Nach dem Ende des Sommersemsters begann mein Praktikum. Ich war in der Informatik-Abteilung "Ausland und Kommerz", die erst vor kurzer Zeit in ein neues Gebäude an der Eggbühlstrasse in Oerlikon umgezogen waren. Nach der Vorstellung aller Mitarbeiter ging es bereits los. Meine Aufgabe war Projektplanungstools zu evaluieren, denn bis jetzt hatte jeder Projektleiter auf seine Art Projekte geplant. Sei es mit irgendeinem Computerprogramm oder sogar von Hand. Um eine Übersicht über alle Projekte zu gewinnen, wurden dann alle Daten von jedem Projektleiter gesammelt und eine neue Liste angefertigt. Das Ziel meiner Studie war, ein geeignetes Projektplanungstool zu finden, das möglichst einfach und komfortabel ist. Es gab also nichts zu programmieren, was mir auch recht war, da ich während den Semestern genügend Programme geschrieben hatte. Das Vorgehen konnte ich selber bestimmen. Ich musste also nicht so vorgehen, wie es in den SKA-internen Projekthandbüchern beschrieben ist. So nahm ich die Vorlesung "Informatik-Projektentwicklung" von C.A. Zehnder wieder hervor. Sehr viel geholfen hat mir auch die Vorlesung "Betriebswissenschaftliche Methoden" von meinem Nebenfach "PPS& Logistik".

Nachdem ich das Vorgehenspapier vorgestellt hatte, musste ich mich zuerst einmal in die Thematik einarbeiten, d.h. die fachlichen Begriffe im Bereich der Planung kennenlernen. Dazu hatte ich die Gelegenheit, mich in der ETH- und BWI-Bibliothek umzusehen. Nach ein paar Tagen Lektüre ging es weiter mit der Analyse des heutigen Systems. Ich befragte alle Projektleiter, welches Tool sie zur Zeit verwenden und wie sie Planung betreiben. Nachdem ich den Ist-Zustand mit Stärken und Schwächen auf Papier festgehalten hatte, begann die Hauptstudie. Es ging nun darum, alle Projektleiter nochmals in einem Gespräch zu fragen, was sie für Anforderungen an ein Planungstool stel-

len. Diese Ideen musste ich dann systematisch analysieren, um einen strukturierten Zielkatalog aufstellen zu können. Nach der Präsentation des bereinigten Zielkatalogs konnte das Pflichtenheft erstellt werden.

Als nächstes folgte die Marktanalyse: Das Ziel war, möglichst viele Informationen über Planungstools zu sammeln, die zur Zeit auf dem Markt sind. Vorgezogen wurden Programme, die auf IBM-kompatiblen Rechnern laufen, möglichst unter MS Windows. In der Abteilung sind IBM-Rechner mit OS/2 2.1 oder DOS 5 mit Windows 3.1 im Einsatz. Da Windowsprogramme auch unter OS/2 2.1 laufen, konnte die Suche auf solche beschränkt werden. Hier kamen auch grosse Unterschiede im Kundenservice einzelner Firmen zum Vorschein: Da gibt es das uns allen wohlbekannte grösste Softwarehaus Microsoft, das es nicht für nötig hält, einen Kunden zu beraten: Man erhält lediglich eine Faxnummer, von der aus man Unterlagen über das gewünschte Programm anfordern kann. Erhalten tut man aber auch nach vermehrter Anfrage nur ein Hochglanzprospekt und eine Demo-Diskette, die nur als Show dienen kann. Wie soll man denn da noch Programme evaluieren können!
Ein anderes, erfreuliches Beispiel ist die Firma Symantec: Nachdem ich eine Demoversion erhalten hatte, kam sogar noch ein Vertreter vorbei, um ihr Produkt im Hause vorzustellen, alles kostenlos versteht sich. Ich erhielt sogar eine vollständige Version des Programms inkl. Handbücher, das man nach Abschluss der Evaluation sogar behalten konnte. Nach vielem Faxen und Telefonieren hatte ich genügend Material gesammelt. Nun ging es um eine erste Ausscheidung der Tools, die die Mussziele und Restriktionen nicht einhalten. Ein wichtiges Mussziel war die Ressourcenverwaltung, die dazu dient, alle Mitarbeiter zu verwalten, was heisst, dass Ferien bei der Planung berücksichtigt werden oder die Gesamtauslastung eines Mitarbeiters über mehrere Projekte hinweg sichtbar ist und ausgeglichen werden kann. Es blieben noch drei Tools übrig, nämlich CA Superproject 3.0, Timeline 1.0 von Symantec und MS Project 3.0.

Nachdem ich diese Programme gründlich getestet und auf die geforderten Ziele überprüft hatte, führte ich eine Nutzwertanalyse durch. Dazu mussten die Kriterien gewichtet werden, was mit Hilfe der Benutzer durchgeführt wurde. Wie eine solche Nutzwertanalyse genau funktioniert, kann im Skript "Betriebswissenschaftliche Methoden" vom BWI nachgelesen werden.

Resultat dieser Analyse war, dass es kein geeignetstes Programm gibt, sondern dass das eine etwas besser als die anderen ist. Dieses bessere war MS Project 3.0. Es ist das benutzerfreundlichste und unterstützt, begrenzt auf 19 Projekte, Mehrprojektverwaltung. Meine Aufgabe war es

nun, diese Evaluation zu präsentieren, damit die Projektleiter einen Entscheid treffen konnten. Es ging vor allem darum, die Benutzer von MS Project zu überzeugen, denn kurz bevor ich mein Praktikum begann, waren einige an einer MS-Project-Demonstration gewesen , an der falsche Tatsachen verbreitet worden waren. Meine Empfehlung wurde auch noch von einer anderen Abteilung bestätigt, die gleichzeitig MS Project evaluiert hatte. Das Projektplanungstool wird jetzt nun SKA-weit eingesetzt werden. Gegen den Schluss hatte ich sogar Zeit, um eine kleine Vorstudie zu einem anderen Projekt zu machen, also doch noch eine kleine Programmierarbeit. Es ging darum, die DDE-Möglichkeiten (Dynamic Data Exchange) zwischen dem eben eingeführten Lotus Notes und anderen Windows-Applikationen zu testen, wobei auch schon erste Mängel von Notes entdeckt wurden.

Während meines Praktikums wurde ich von alle Mitarbeitern gut unterstützt und war stets willkommen. Mein Betreuer liess mir sehr viel Spielraum zur Verfügung. Da ich zwischendurch auch mal etwas warten musste, konnte ich einigen Mitarbeitern Ratschläge weitergeben, sei es im Zusammenhang mit DOS-Rechnern, mit objektorientierter Programmierung, die erst jetzt langsam im Bankgeschäft Einzug hält oder mit Lotus-Notes-Problemen.

Angestellt war ich wie ein normaler Mitarbeiter, hatte einen eigenen Computer, eigenes Telefon, bekam Mittagessensentschädigung, hatte sogar 5 Tage Ferien und mein Bankkonto freute sich... Was will man da noch mehr?

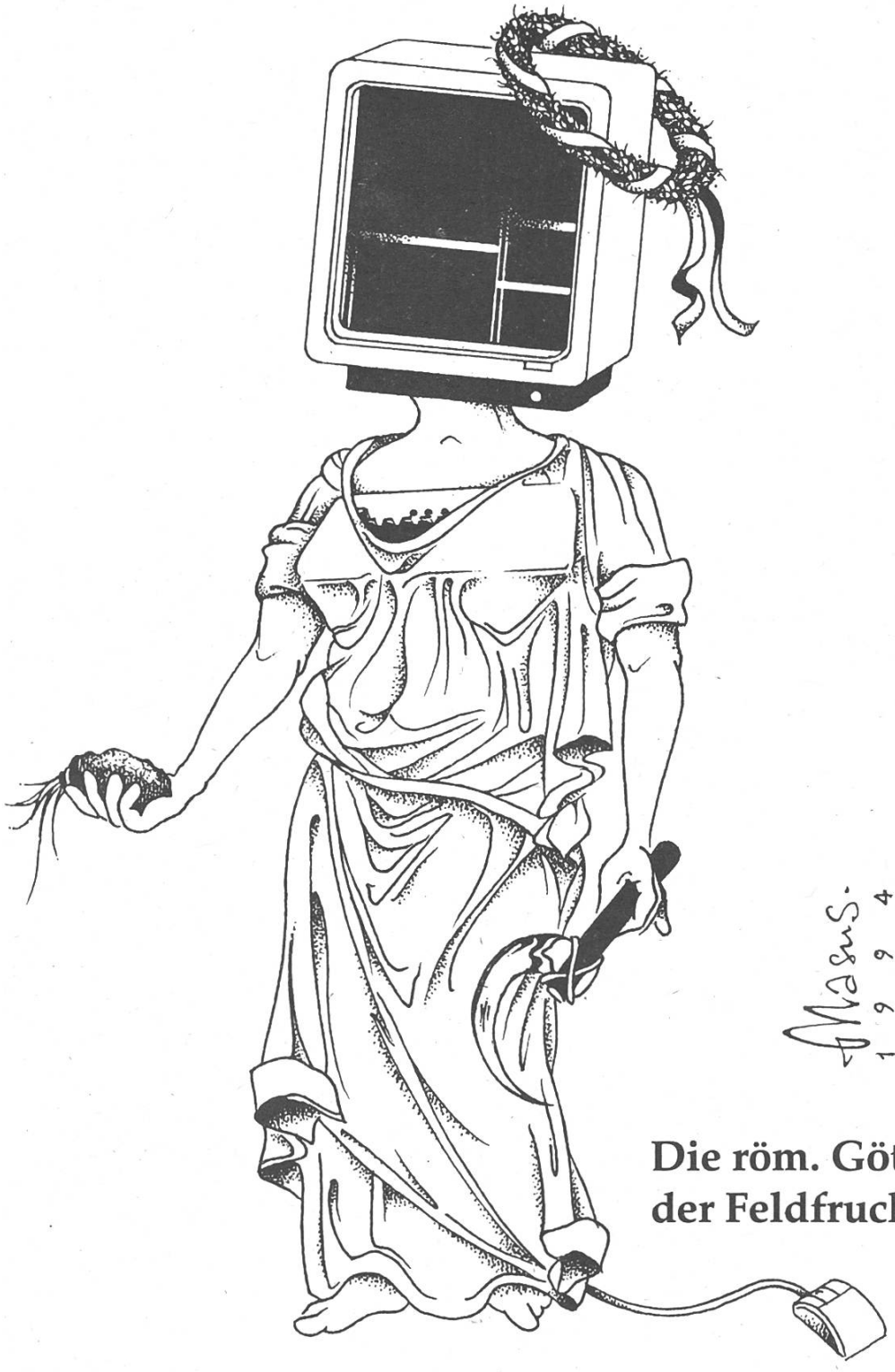
Oerlikon, 14. Oktober 1993

Daniel Ponti IIIC/7

Gratis an die Computer Graphics 94

Vom 2. bis 4. Februar findet im Kongresshaus in Zürich die Computer Graphics 94 statt. Geöffnet ist sie von 9 bis 6 Uhr, am letzten Tag eine Stunde kürzer. 140 Aussteller versammeln sich dort und informieren die Besucher über die neuesten Trends in Bereichen von CAD, CAM-CIM und CAD in der Architektur, über Animation und Simulation bis zu Virtual Reality und Multimedia.
Wir als Studenten sind an diese Messe eingeladen, die Legi öffnet uns die Türen gratis.
Interessant könnten vor allem die dort stattfindenden Vorträge werden. Eine Broschüre liegt im VIS-Büro auf.

Spielzeuge der InformatikerInnen, 7. (letzte) Folge



Die röm. Göttin der Feldfrucht

C und UNIX: Eine neue Einführungsvorlesung

In vielen Vorlesungen des Fachstudiums werden Kenntnisse von UNIX und C erwartet. Diese Kenntnisse fehlen vielen Studenten, da das Grundstudium fast ausschliesslich auf das Betriebssystem und die Sprache Oberon ausgerichtet ist.
Kurzfristig, also vor der zur Zeit diskutierten Reform des Grundstudiums, biete ich nun eine Vorlesung „UNIX und C" an, die so plaziert ist, dass sie von Studierenden des 4. und 6. Semesters besucht werden kann.

Die Vorlesung ist "empfohlen" für das 4. Semester und die darin vermittelten Kenntnisse sind insbesondere ab Herbst 1994 für folgende Veranstaltungen des Fachstudiums eine Voraussetzung:

- Kernfach Informationssysteme

- Wahlfächer:
  - Architektur und Realisierung von Datenbanksystemen I + II
  - Aufbau symbolischer Rechensysteme

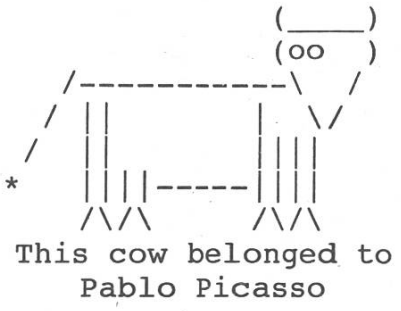
Für eine Reihe weiterer Vorlesungen sind Kenntnisse von UNIX/C empfohlen.
Entsprechende Hinweise stehen im Katalog der Lehrveranstaltungen.

Die Vorlesung im kommenden Sommersemester hat das Format 2V+2U. Sie wird jeweils Do von 10-12 stattfinden, die Übungen Mi von 16-17.

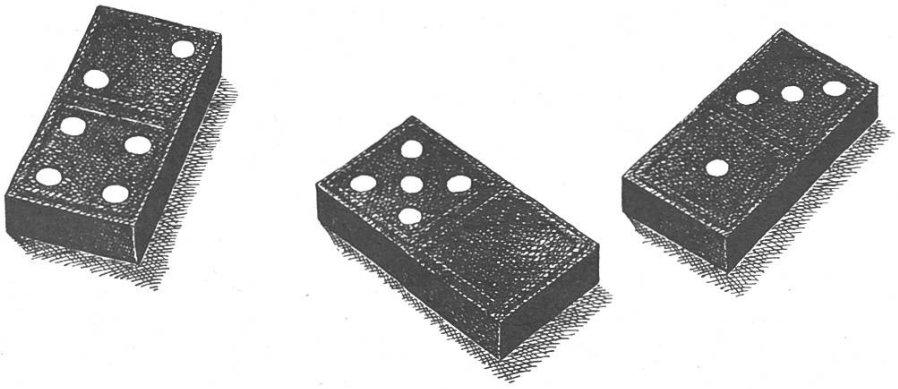
Nach einer kurzen Einführung in die Sprache C wird vor allem über Systemprogrammierung unter UNIX gesprochen. Weitere Themen sind Libraries, die Programm-Entwicklungsumgebung (Make, Debugging und Profiling, Lex und Yacc), Modularisierung und Standardisierungsfragen. In den Übungen werden kleinere und grössere Programme entwickelt.
Dabei besteht die Gelegenheit, die im Grundstudium erworbenen Kenntnisse über Algorithmen und Datenstrukturen praktisch anzuwenden.

Roman Mäder

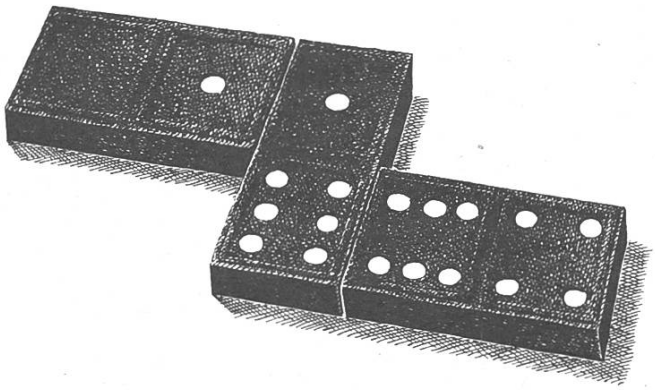
ASCII-COWS I



Andere.



Apple.



Die meisten Computer sind ohne spezielle Hard- und Software nicht netzwerkfähig. Das kostet nicht nur zusätzlich Geld, sondern erfordert auch einen hohen Installationsaufwand. Mit dem Macintosh hingegen finden Sie sowohl im Lehrerzimmer als auch an der Universität sofort Anschluss. Denn er ist mit AppleTalk von Anfang an netzwerkfähig. Und um Peripheriegeräte anzuschliessen, brauchen Sie ebenfalls nicht den Doktor zu machen. Ausserdem kann jeder Macintosh auch MS-DOS-, Windows- und OS/2-Disketten lesen und beschreiben. Endlich eine Studentenverbindung, der auch Lehrer beitreten können.

Apple

Generalvertretung für die Schweiz und Liechtenstein:
Industrade AG, Apple Computer Divison, Hertistrasse 31, 8304 Wallisellen, Tel. 01 832 81 11.

Some Swansea Problems

Diverse Anfragen haben uns dazu bewogen, ein paar der ACM-Programmieraufgaben, die an der Ausscheidung vom letzten November in Swansea gestellt wurden, hier in den Visionen abzudrukken. Lösungen dazu sind im VIS-Büro *nicht* erhältlich...

1993-4 ACM International Collegiate Programming Contest
Western European Regional at University College Swansea

Problem **A**
Chess

Please submit this problem as 'chess.c' (C) or 'chess.p' (Pascal).

Almost everyone knows the problem of putting eight queens on an 8×8 chessboard such that no Queen can take another Queen. Jan Timman (a famous Dutch chessplayer) wants to know the maximum number of chesspieces of one kind which can be put on an $m \times n$ board with a certain size such that no piece can take another. Because it's rather difficult to find a solution by hand, he asks your help to solve the problem.

He doesn't need to know the answer for every piece. Pawns seems rather uninteresting and he doesn't like Bishops anyway. He only wants to know how many Rooks, Knights, Queens or Kings can be placed on one board, such that one piece can't take any other.

Input
The first line of input contains the number of problems. A problem is stated on one line and consists of one character from the following set r, k, Q, K, meaning respectively the chesspieces Rook, Knight, Queen or King. The character is followed by the integers m ($4 \leq m \leq 10$) and n ($4 \leq n \leq 10$), meaning the number of rows and the number of columns or the board.

Output
For each problem specification in the input your program should output the maximum number of chesspieces which can be put on a board with the given formats so they are not in position to take any other piece.

Note
The bottom left square is *1, 1*.

Example

Input
```
2
r 6 7
k 8 8
```

Output
```
6
32
```

Problem **F**
Compress

Please submit this problem as 'compress.c' (C) or 'compress.p' (Pascal).

Nowadays everyone is using compression methods to reduce the space occupied by data. In some cases this is done in such a way you hardly notice it, for example tapestreamers, modems and harddisk doubling programs. In other cases you have to do it yourself by using pack, arj, zip, zoo or arc.

Most compression-methods are very complicated, but for this problem only the following simplified method is used. An ASCII-character consists of eight bits, which allows the encoding of 256 different characters. It's possible to recode the characters with a new sequence of bits. These sequences may have a different length. When you choose to give the characters which are often used a shorter sequence of bits than the characters which are seldom used the total text size will be reduced.

Recoding characters gives one other problem, which occurs when you try to get the text back. In a standard ASCII text you know the first character begins at the 1st and ends at the 8th bit, the second begins at the 9th and ends at the 16th bit and so on. When using a variable length coding you don't know where a character begins. For example when an 'a' is coded as '11' and an 'e' as '1111', given the bit-sequence '111111' you don't know if it means 'ae', 'ea' or 'aaa'.

The last problem is solved when the next principle is used. Suppose you want to give some characters a code length of 3 (and you haven't given any character a code yet). In that case you've got 8 (2 to the power 3) possibilities. Seven codes can be immediately allocated to characters. The last one depends on the fact if you need to give just an other chararacter a code or more. If you only have one left to code, the last code will do, but if you have to code more than an extension must follow. The extension has the same structure. This has to be continued until all different characters in the text are given a code.

For example you've the characters 'a', 'e', 'i', 'o', 'u' and 'y'. You choose to give the first three characters a code with length 2 and the rest an equal length. So 'a' becomes '00', 'e' becomes '01' and 'i' becomes '10'. The other characters have to be an extension of '11'. Then 'o' becomes '1100', 'u' becomes '1101' and 'y' becomes '1110'. The code '1111' is free now. If 7 characters instead of 6 characters should be given a code this last code would be sufficient. If more than 7 characters should be given a code, the last code would be extended and so on.

You only have to code the characters which occur in the text. You don't have to give the code table itself.

You must write a program that reduces a given text by recoding each character and give the minimum total filelength in bits.

Input

The first line of the input file contains the number of problems. A problem starts with the number of lines *n* to follow, followed by *n* lines. A line consists of characters and the characters 'A'..'Z', 'a'..'z', space, '.', ',' ,'-' and 'S' are allowed. The character 'S' will always be at the end of a line and cannot occur anywhere else. This character has to coded instead of the real end of line mark.

Output

For each problem in the input your program should output the minimal number of bits to code the given text.

Example

Input

```
2
3
Hello Contestant,$
Please write a program which gives$
the text Hello world.$
1
To be or not to be, that is the question.$
```

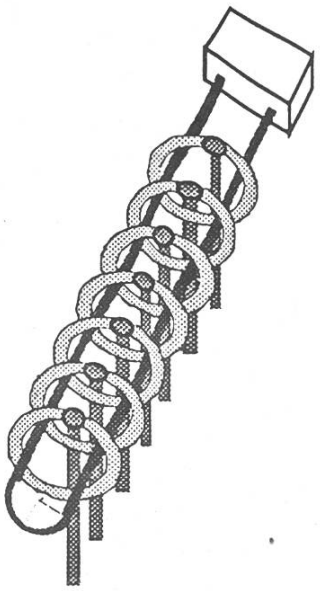
Output

```
335
167
```

Problem **B**
Spin

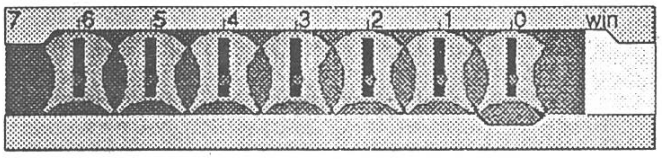
Please submit this problem as 'spin.c' (C) or 'spin.p' (Pascal)

The classic Chinese Rings puzzle comes in a variety of forms. The original version has seven rings linked together by a sliding loop threaded through them. The aim is to remove the loop by manipulating the rings (see right).

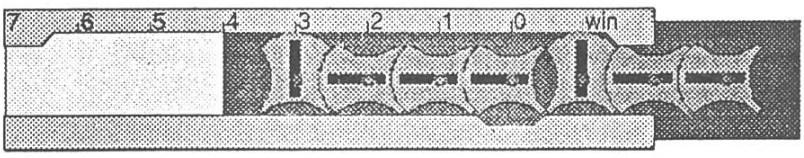


The Original Chinese Rings Puzzle

A modern implementation uses seven disks with specially shaped cut-outs mounted on a slide. The slide can move left and right. The slide can always move left until it reaches its left-most position, shown here:

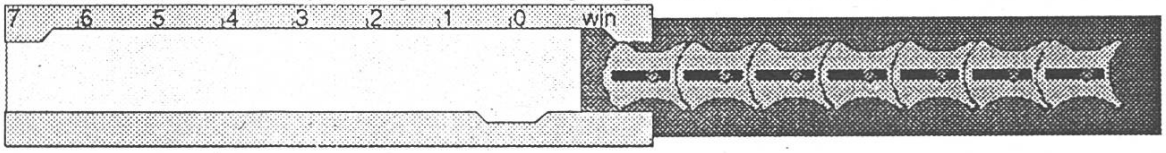


Each disk can be rotated 90°, so the long end of the black bar points either straight up (vertical) or to the left (horizontal). The slide can only move right until a vertical disk hits the end stop under the 'Win' marking:



A disk can be rotated between horizontal and vertical only if it is positioned over the indentation marked '0' *and* the disk on its right is vertical . The right-most disk can always rotate if it is in position '0' since it has no disk on its right.

The aim is to free the slide by moving it so its left edge aligns with the 'Win' mark:



Your task is to write a program which will take several part-solved puzzles and compute the number of steps needed to move the slide to position 'Win' for each puzzle.

Input

There will be several puzzles in the input file. The first line of the file will contain an integer *n* specifying the number of puzzles. There will then be *n* lines, each of the form:

length orientations position

where *length* is an integer indicating the number of disks on the slide, *orientations* is a string of *length* characters from the set [*h,v*] giving the orientation of each disk from left to right, and *position* is an integer from 0 to *length* specifying the numbered mark which aligns with the left edge of the slide.

Output

For each puzzle, your program should output one integer on a line which counts the minimum number of steps needed to win the puzzle. A step is either a movement of the slide, one unit left or right, or the rotation of a disk.

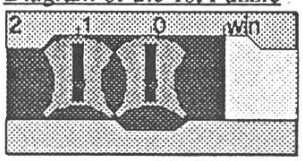
Sample Input

```
2
2 vv 2
7 vhhhvhh 4
```

Output for the Sample Input

```
7
357
```

Diagram of the 1st Puzzle



⇨ Seite 13

#include

Die Anweisung #include wird dazu verwendet, Definitionsdateien (sog. Headerfiles) in den Quelltext einzufügen. Genaueres dazu, insbesondere darüber, was denn diese geheimnisvollen aber scheint's ziemlich wichtigen Headerfiles denn sollen, kommt erst später, bei der behandlung der getrennten Übersetzung. Für jetzt sei nur soviel gesagt: Immer wenn eine Funktion von ausserhalb des aktuellen Quelltextes benutzt werden soll, also eine, die in einem anderen Quelltextfile steht, schon übersetzt wurde und einfach in unser Programm eingebunden werden soll, dann hat man die Definitionsdatei in die aktuelle "Compilation-Unit" einzubinden. Die Zeile

```
#include <modul.h>
```

oder

```
#include "modul.h"
```

veranlasst den Präprozessor, den Inhalt der angegebenen Datei einzulesen und in den gerade bearbeiteten Quelltext einzufügen. Der Unterschied zwischen den Begrenzungszeichen ist einigermassen irrelevant. Gebraucht für eigene Definitionsdateien Doppelanführungszeichen " und ", für Standarddefinitionsdateien spitze Klammern <>. Standarddefinitionsdateien sind etwa stdio.h oder string.h – zu finden im Filesystem in /usr/include oder /include. Der Vorgang entspricht de jure (mitnichten aber de facto!) dem Oberon-IMPORT.

Die erwähnten Möglichkeiten sind nicht alle, die der Präprozessor bietet. Insbesondere besteht da noch die in UNIX sehr oft gebrauchte Möglichkeit der Definition von sogenannten Makros. Informationen darüber sind in K&R zu finden.

scanf

Jetzt, wo wir gelernt haben, *wie* wir eine Funktion definieren, können wir gleich zur Erklärung einer bestimmten wichtigen Funktion schreiten:
Sie heisst scanf und dient zum Einlesen von Variablen. Sie hat folgende Definition:

```
int scanf(char fmt[], ...)
```

Na,ja – toll ist diese Definition ja nicht, kommen wir deshalb zur Erklärung: *fmt[]* ist ein Character-Array (siehe weiter unten) und sieht dem Format-String von printf sehr ähnlich: In diesen String schreibt man gewöhnlich Ausdrücke wie "%s", "%d", "%f" usw. (ganz wie bei printf). Als weitere Argumente folgen die dem Formatstring entsprechenden Variablen, die alle als Rückgabeparameter definiert sein müssen. Beispielsweise kann man mit dem Aufruf

```
scanf("%s %d %c %f", string,
&myint, &mychar, &myfloat);
```

einen String (was das genau ist, kommt gleich), eine Ganzzahl, ein Zeichen und eine Fliesskommazahl einlesen. Bei der Eingabe muss darauf geachtet werden, dass dann die unterschiedlichen Elemente durch Leerzeichen getrennt werden. Übrigens ist es unmöglich diese Eingabe zu unterbrechen. So, wie oben definiert, kann man auch keine Leerstrings einlesen. Das geht aber mit der Angabe von %[...] im Formatstring. Innerhalb der eckigen Klammern kann man nun Zeichen und Zeichenbereiche angeben, die eingelesen werden können sollen. Somit liest der Aufruf

```
scanf("%[0-9 a-z]",string);
```

einen String ein, der die Ziffern von 0-9, das Leerzeichen und die Kleinbuchstaben enhalten darf. Folgende Eingabe wird also vollständig in *string* eingelesen:

```
hallo ich bin der 0815 c hacker
```

Und folgende Eingabe wird nur bis "0815" eingelesen:

```
hallo ich bin der 0815-C-Hacker
```

Im obigen Formatstring ist ja '-' nicht zugelassen. Lassen wir es zu (indem wir das '-' ganz hinten oder ganz vorne einfügen), dann wird die Eingabe allerdings nur bis 0815- eingelesen, da wir die Grossbuchstaben auch nicht erfasst haben.

Man kann in scanf die Eingabe auch auf eine bestimmte Anzahl Zeichen begrenzen. Das macht man, indem man zwischen % und die das nachfolgende Zeichen eine Zahl setzt. Mit Hilfe eines '*' kann man eine Eingabe überspringen (damit kann man beispielsweise am Ende der Eingabezeile Zeichen auffressen). Folgender Aufruf

```
scanf("%80[0-9a-zA-Z ,.-] %*[
-~]",string);
```

liest also höchstens 80 Zeichen in *string* ein und frisst alle übrigen Zeichen auf.

Strings und Stringfunktionen

Da wir nun wissen, wie wir mit Funktionen arbeiten können, und die Grundkenntnisse über Pointer haben, sind wir in der Lage, mit einer wichtigen Datenstruktur zu arbeiten: Strings.

Strings sind Character-Arrays. Damit wissen wir schon, wie wir einen String definieren:

```
char string[80];
```

Wie auch in Oberon wird ein String mit dem ASCII-Zeichen 0 beendet. In der oben definierten Zeichenkette haben also 79 benutzergewollte Zeichen Platz. Die Zuweisung von Zeichenketten bereitet schon die ersten Schwierigkeiten: Die Anweisung:

```
string= "zensuriert ;-)";
```

funktioniert NICHT! C unterstützt nämlich Strings nur mit Hilfe eingebauter Bibliotheksfunktionen. Demnach muss die Zuweisung mit Hilfe einer Funktion geschehen. Und hier ist sie: *strcpy(a,b)* kopiert eine Zeichenkette von b nach a. Die obengeschriebene Zuweisung muss also im C-Programm folgendermassen lauten:

```
strcpy(string,"zensuriert ;-
)");
```

– mit einer Ausnahme: In C ist es möglich, eine Variable bei der Deklaration gleich zu initialisieren:

```
int i= 0;
```

Diese Zeile deklariert die Ganzzahl i und initialisiert sie mit dem Wert 0. Arrays können bekanntlich folgendermassen deklariert und initialisiert werden:

```
int a[]={ 0, 8, 15, 47, 11,
632, 72, 12 } ;
```

In dieser Anweisung wird das Array a mit den Werten 0,8,15... belegt. Gleichzeitig bestimmt der Compiler die Grösse des Arrays (8). Die muss man nicht angeben, da diese Information in der Anzahl angegebener Werte schon enthalten ist.

Im Falle eines Character-Arrays kann man das zwar auch so machen:

```
char string[]=
{'M','a','n','n',',',
 ' ','i','s','t',' ',
 'd','a','s',' ','d',
 'o','o','f','\0'};
```

Das geht aber auch kürzer:

```
char string[]= "Mann, ist das
doof";
```

Man beachte, dass im ersten Fall das 0-Zeichen angegeben werden muss, im zweiten Fall nicht.

Im Falle der *Initialisierung* eines Strings darf man also Zeichenketten mit = einander zuweisen, sonst nicht.

Wer wissen will, wie lange ein String ist (der mit dem 0-Zeichen abgeschlossen sein muss), der verwendet mit Vorteil die Funktion strlen(s). Sie gibt eine Ganzzahl als Wert zurück, die die Länge der Zeichenkette s enthält.

Zeichenketten miteinander zu vergleichen, was in Oberon so leicht mit Hilfe von =,<,>,>=,<= funktioniert, ist in C nur mit der Bibliotheksfunktion *strcmp(a,b)* zu realisieren. Strcmp() liefert als Funktionswert eine Ganzzahl i mit folgender Semantik zurück:

i>0	falls	a>b
i==0	falls	a==b
i<0	falls	a<b

Die Prozedur, die man zum Zusammenhängen von Strings benötigt, heisst *strcat(a,b)*. Sie hängt die Zeichenkette *b* an die Zeichenkette *a* an. Das Resultat findet sich in *a*.

Bevor ich weitere Funktionen und Prozeduren nenne, die auf Zeichenketten operieren, möchte ich auf eine weitere Eigenheit von C zu sprechen kommen: Bei Prozedur- und Funktionsparametern werden in C Arrays wie Pointer behandelt. Der Compiler kodiert also nicht wie in Oberon ein Array als Tupel

< Zeiger auf das erste Element, Grösse des Arrays >,

sondern ein Array ist für den C-Compiler ein Pointer.

Folgender Prozedur-Header kann also zweierlei bedeuten:

```
void dummyproz(char *a);
```

Entweder wird ein Zeichen wieder zurückgegeben oder in *a* versteckt sich eine ganze Zeichenkette. Da dies zu Verwirrungen führen kann (das habe ich nur geschrieben, damit Ihr die unleserlichen C-Programme mit der Zeit verstehen lernt), rate ich Euch (für den Fall, dass Ihr mit *a* wirklich einen String meint) zu folgender Schreibweise:

```
void dummyproz(char a[]);
```

Das bedeutet das gleiche, ist aber lesbarer.

Anders als in Oberon (oder Modula-2), wo man Strings auf zweierlei Art übergeben kann, nämlich:

```
1.) PROCEDURE dummyproc(a:
ARRAY OF CHAR);

2.) PROCEDURE dummyproc(VAR
a: ARRAY OF CHAR);
```

wird in C ein String immer als VAR-Pa-

rameter übergeben. Das heisst alle Änderungen, die Ihr an einem String in einer Prozedur vornehmt, werden im Caller übernommen. (Das liegt eben an der Tatsache, dass Arrays äquivalent mit Pointern sind.)

Folgende weitere Funktionen zur Bearbeitung von Strings sind bei den meisten C-Compilern dabei. Das "n" im Funktionsnamen deutet auf die Angabe der maximalen Bearbeitungslänge hin. Ansonsten verhalten sich die Funktionen wie ihre Pendants ohne n.

- `void strncpy(char to[], char from[], int count);`
  Kopiert von from[0..count] nach to[0..count]

- `int strncmp(char s1[], char s2[], int count);`
  Vergleicht s1[0..count] mit s2[0..count]

- `void strncat(char s[], char append[], int count);`
  Hängt append[0..count] an s an.

- `int strcasecmp(char s1[], char s2[]);`
  `int strncasecmp(char s1[], char s2[], int count);`
  Diese Funktionen machen das Gleiche wie strcmp und strncmp. Sie vernachlässigen dabei allerdings Gross- und Kleinschreibung.

Wenn Ihr noch mehr wissen wollt, dann empfiehlt sich, "man string" in ein Shell-Fenster einzutippen. Das klärt Euch über alle String-Funktionen auf, die der C-Compiler zur Verfügung stellt.

Die Hauptprozedur main

Wir kommen noch einmal zur Hauptprozedur zurück, weil wir Euch darüber noch nicht alles gesagt haben. Wie wir schon einmal erwähnt haben, hat main folgende Definition:

```
void main(int argc, char
*argv[])
```

Nun, wie Ihr bemerkt haben werdet, gibt es Programme, die auf der Eingabezeile Parameter verlangen (gcc, als Beispiel, will unter anderem als Parameter den Namen des Sourcefiles).

Das ist sehr einfach zu programmieren: argc und argv helfen uns dabei. In argc ist die Anzahl der Parameter gespeichert, wobei der Programmname auch ein Parameter ist (wenn man also *ls* eingibt enthält der Parameter argc im main von ls eine 1).

Der Parameter argv ist ein Zeiger auf einen Array von Strings, also einen Array mit Zeigern auf die eigentlichen Zeichenketten. Da sieht man nun, dass man unbedingt wissen muss, dass bei Prozedurparametern Arrays und Pointer äquivalent sind. Denn die Schreibweise *char argv[][]* lassen die C-Compiler NICHT zu. Man muss also selber wissen, was sich hinter einer solchen Definition verbirgt. (Manchmal schreibt man auch *char **argv*.)

Also: In argv findet man die einzelnen Parameter als Strings: In argv[0] steht der Programmname, in argv[1] steht das erste Argument usw.

In unserem heutigen Beispielprogramm könnt Ihr genau sehen, wie man damit umgeht.

Files

Ein- und Ausgabe sind relativ wichtige Dinge. Mit der Erklärung von *printf* und

scanf haben wir zwar einen wesentlichen Punkt dieses Kapitels abgehakt, aber, wie man weiss, ist Interaktivität nicht alles: Files sind beim Arbeiten mit dem Computer unentbehrlich. In C ist die Ein- und Ausgabe über Dateien stark verwandt mit der interaktiven Ein- und Ausgabe.

Für die gepufferte Ein- und Ausgabe nach ISO muss man das Headerfile "stdio.h" in den Source code einbinden – wie wir es ja schon das letzte Mal machten:

```
#include <stdio.h>.
```

Das sollte eigentlich sowieso bei jedem Programm in einer der ersten Zeilen stehen, das irgendwie mit Ein- und Ausgabe zu tun hat.[4]

Das Öffnen eines Files erfolgt mit der Funktion *fopen*. Sie hat folgende Definition:

```
FILE *fopen(char name[], char
mode[])
```

Fopen liefert einen Pointer auf ein FILE zurück (wie das FILE aussieht, interessiert uns im Moment nicht). Dazu muss man den Namen und den Modus als Zeichenkette angeben. Folgende Modi können gewählt werden:

- "r" oder "rb" = read: Ein File wird nur zum Lesen geöffnet.
- "w" oder "wb" = write: Ein neues File wird unter dem angegebenem Namen geöffnet. Existiert schon ein File mit diesem Namen so wird es gnadenlos überschrieben.
- "a" oder "ab" = append: Ein altes File wird zum Weiterschreiben geöffnet. Existiert das File nicht, wird ein neues erzeugt. Der Filepointer wird auf das Ende des Files positioniert. Es ist nicht möglich den alten Inhalt des Files zu überschreiben.'
- "r+" oder "r+b" oder "rb+" = Update, read and write: Ein File wird zum Lesen und Schreiben geöffnet. Es muss bereits existieren. Der Filepointer wird auf den Anfang des Files positioniert.
- "w+" oder "w+b" oder "wb+" = Update, write: Das File wird zum Schreiben geöffnet. Die Länge des Files wird auf 0 reduziert. Der Filepointer wird auf den Anfang des Files positioniert.
- "a+" oder "a+b" oder "ab+" = Update, append: Ein altes File wird zum Schreiben und Lesen geöffnet. Der Filepointer wird auf das Ende des Files positioniert. Es ist nicht möglich, den alten Inhalt des Files zu überschreiben.

Das "b" in den oben angegebenen Modi steht für *binary* und wird zum Unterscheiden von Text- und Binärdateien benutzt. In UNIX wird nicht zwischen diesen beiden Typen unterschieden. Es ist dort also wirkungslos.

Der Rückgabewert FILE ist übrigens "NULL" , falls irgendein Fehler beim Öffnen der Datei aufgetreten ist. NULL entspricht dem Oberon-NIL.

Gelesen wird mit der Funktion *fread*. Sie hat folgende Definition:

```
size_t fread(void *b, size_t
b_size, size_t n, FILE *fp)
```

Oops - Das sieht einiges komplizierter aus! Um es gleich vorwegzunehmen: Die Definition *void *b* bedeutet nichts anderes, als dass hier ein Pointer zu *irgendeinem* Datentyp angegeben werden soll. Dieser Pointer zeigt auf einen Buffer, in welchen die zu lesenden Daten geschrie-

4 Ist ein Programm, das keine Ein- und Ausgabe hat, überhaupt sinnvoll?.

ben werden sollen. Im Normalfall gibt man da ein Array an (ein Beispiel folgt sogleich). Der nächste Parameter (size_t b_size) gibt die Grösse eines Elements dieses Buffers in Bytes an. Dafür gibt es in C eine Standardfunktion, die uns sagt, wieviel Bytes ein Datentyp benötigt: Sie heisst *sizeof()*. Der Aufruf

```
s= sizeof(int);
```

weist s auf der Sun (beispielsweise) den Wert 4 zu.Der Parameter n informiert die Funktion über die Anzahl der Elemente in unserem Buffer. Und den letzten Parameter kennen wir schon: Das ist das FILE, das wir von fopen bekommen haben. Der Typ der beiden Variablen, size_t, entspricht eigentlich überall einem int.

Der Funktionswert von fread teilt uns mit, wieviele Elemente gelesen wurden. Am Ende eines Files wird dieser Wert typischerweise kleiner als n sein.

So, jetzt noch das angekündigte Beispiel: Angenommen wir wollen aus einem File 80 Text-Zeichen auf einmal herauslesen, dann würde der Aufruf von fread so aussehen:

```
#include <stdio.h>

char buffer[80];
FILE *fp;

...

fp= fopen("ReadMe","r");
wieviel= fread (buffer,
   sizeof(char),80,fp);
```

Das Schreiben besorgt (wie könnte es anders sein) eine Funktion namens *fwrite*. Sie hat haargenau dieselbe Struktur wie fread. Als Funktionswert wird die Anzahl geschriebener Elemente zurückgegeben. Sollte sich hier eine Differenz mit dem Parameter n ergeben, dann ist ein Fehler aufgetreten (beispielsweise: Diskette voll).

Dasselbe Beispiel zum Schreiben von 80 Zeichen:

```
char buffer[81];
FILE *fp;
```

strcpy(buffer,"Ich schreibe den Buffer ohne Null-Zeichen!");

```
fp= fopen("WriteMe","w");
wieviel= fwrite (buffer,
sizeof(char),80,fp);
```

Neben diesen oft verwendeten Funktionen gibt es auch die Funktionen fprintf und fscanf. Na, die beiden haben wir schon fast einmal kennengelernt. Sie haben folgende Definition:

```
· int fprintf(FILE *fp, char
  fmt[], <type> arg1,<type> arg2,
  ...)
· int fscanf(FILE *fp, char fmt[],
  <type> arg1, <type> arg2, ...)
```

Die Funktion fprintf schreibt allerdings ausschliesslich strings! Der Wert der Funktion sagt uns, wieviele Zeichen geschrieben wurden. Ein negativer Wert bedeutet, dass ein Fehler passiert ist. Der Parameter *fmt[]* ist ein Formatstring wie er in der letzten Folge erkärt wurde.

Die Funktion fscanf funktioniert wie scanf, aber mit Files. Sie gibt als Wert die Anzahl der Elemente zurück, die gelesen wurden.

Weitere File-I/O-Funktionen im Überblick:

- `char *fgets(char buffer[], int length, FILE *fp)`
  liest einen String vom FILE. Normalerweise geht das Resultat in den buffer der Länge length. Der Funktionswert ist NULL, falls ein Fehler aufgetreten ist. Ansonsten zeigt er auf den buffer.

- `int *fputs(const char s[], FILE *fp)`
  schreibt einen String s in das FILE. Wenn ein Fehler auftritt, ist der Funktionswert EOF, ansonsten ist er 0.

- `int fgetc(FILE *fp)`
  liest das nächste Zeichen (muss in ein char umgewandelt werden). Falls ein Fehler aufgetreten ist ist der Rückgabewert EOF.

- `int fputc(int c,FILE *fp)`
  schreibt ein Zeichen c in das FILE. Wenn alles gut läuft, steht im Resultat das Zeichen c, ansonsten EOF.

Das sind die wichtigsten File-I/O-Funktionen. Weitere kann man mit Hilfe der Manpages finden. Ein File wird mit der Funktion fclose wieder geschlossen. Die Definition ist

- `int fclose(FILE *fp)`

Der Rückgabewert ist 0 falls alles geklappt hat und ungleich 0 falls nicht.

Das Beispielprogramm

Unser Beispielprogramm für diesmal macht nicht sehr sinnvolle Sachen. Aber: Praktisch alles, was wir diesmal gelernt haben, wird in dem Programm behandelt: Das Programm liest ein Text-File zeilenweise in ein String-Array ein, und mischt die Zeilen durcheinander. Das Programm benötigt als Argument beim Programmaufruf den Namen des Textfiles. Wir haben das Programm mit folgenden Daten laufen lassen:

Ein Gleiches
Über allen Gipfeln
Ist Ruh
Über allen Wipfeln
Spürest du
Kaum einen Hauch
Die Vögelein schweigen im Walde
Warte nur, balde
Ruhest du auch

und folgendes Ergebnis erhalten:

0: Kaum einen Hauch
1: Warte nur, balde
2: Ruhest du auch
3: Die Vögelein schweigen im Walde
4: Spürest du
5: Über allen Gipfeln
6: Ist Ruh
7: Ein Gleiches
8: Über allen Wipfeln

—

Das wär's. Das Beispielprogramm findet Ihr auf den nächsten Seiten. Ansonsten wünschen wir Euch viele frohe Stunden (am Computer und auch sonst...) und verabschieden uns bis zum nächsten Mal.

P.S. Zur Belohnung fürs Durchhalten hier noch ein kleines Schmankerl:

```
float o=0.075,   h=1.5,T,r,O,l,I;
int _, L=80, s=3200; main() {for(
;s%L||(h-=o,T= -2),s; 4 - (r=O*O)
<(l=I*I)|++ _ ==   L&&write(1,(--
s%L?_<L?--_%6:6:7)   +"World! \n"
,1)&&(O=I=l=_=r=0,T+=o /2))O=I*2*
O+h, I=l+T-r;}
```

Eintippen (alles auf einer Zeile), compilieren, staunen... Viel Spass!

```
/*********************************************/
/* 2. Beispielprogramm zum visionaeren C-Kurs */
/* Autor: Leonhard Jaschke                    */
/* Idee: Patrick Leoni                        */
/*                                            */
/* Abstract: Das Programm liest ein Text-File */
/* ein, dessen Zeilen nicht laenger als       */
/* LENGTH-1 sein sollten. Ausserdem sollte    */
/* das Text-File nicht mehr als MAXSTRINGS    */
/* Zeichen enthalten. Die Zeilen des Files    */
/* werden zufällig vertauscht und wieder      */
/* ausgegeben.                                */
/*********************************************/

#include <stdio.h>
#include <stdlib.h> /* fuer die Funktion rand() */

#define LENGTH     80 /* maximale Laenge unserer Strings */
#define MAXSTRINGS 40 /* maximale Anzahl der Strings */

#define LF         10 /* Line Feed */
#define ENDSTRING  0

void mixit(char strings[MAXSTRINGS][LENGTH],int n)
{
   char merk[LENGTH];
   int i,j;

   for(i= 0; i<n; i++)
   {
      j= (int)(((float)(rand()>>16)/65536.0)*n);
/* rand() liefert als Wert eine 32-Bit-Zahl. Die Operation >>16
dividiert diese Zahl ganzzahlig durch 2^16 (16-maliger shift-left).
Daraus resultiert eine 16-Bit Zahl zwischen 0 und 65535. Diese
skalieren wir durch die Division auf eine Zahl zwischen 0 und 1 und
blasen sie mit der Multiplikation auf eine Zahl im Intervall [0,n[
auf */
      strcpy(merk,strings[i]);
      strcpy(strings[i],strings[j]);
      strcpy(strings[j],merk);
   }
}
```

```
void main(int argc, char *argv[])
{
   FILE *input;
   char merk[MAXSTRINGS][LENGTH];
   int strings, length, i;
   int read;
   char ch;

   if (argc!=2)
   {
      printf("USAGE: %s <filename>\n",argv[0]);
      exit(10L);
   }
   input= fopen(argv[1],"r");
   if (input)
   {
      strings= 0;
      length= 0;
      read= fread(&ch,sizeof(char),1,input);
      while(read!=0)
      {
         if (ch!=LF)
         {
            merk[strings][length]= ch;
            length++;
         }
         else
         {
            merk[strings][length]= ENDSTRING;
            strings++;
            length= 0;
         }
         read= fread(&ch,sizeof(char),1,input);
      }
      fclose(input);

      mixit(merk,strings);

      for(i= 0; i<strings; i++)
      {
         printf("%d: %s\n",i,merk[i]);
      }
   }
   else
   {
      printf("No such file!\n");
   }
}
```

Neues aus der Abteilung für Informatik

Die Testat- und Zulassungskontrolle zu den Frühjahrsprüfungen findet in der Zeit von

Montag, 7. Februar bis und mit
Donnerstag, 10. Februar 1994

statt und zwar:

12.30 bis 14.00 bei H. Hilgarth für die Vordiplome und den 1. Teil des Schlussdiploms nach Studienplan 1989

14.00 bis 16.00 bei L. Perrochon für den 2. Teil SD (Studienplan 1989) und Prüfungen nach neuem Studienplan sowie insbesondere Uebertritte dazu.

Wichtig:

Auch Studierende nach neuem Studienplan, bei dem keine Testate mehr erforderlich sind, müssen sich zur Zulassung unbedingt melden. Dies gilt auch für Repetenten aller Prüfungsstufen.

Nichterscheinen hat automatisch die Abmeldung von den Prüfungen zur Folge.

Wie bereits in früheren Jahren wurden die Termine für die Vordiplomprüfungen zugunsten einer längeren Vorbereitungszeit wiederum soweit als möglich in den hinteren Teil der Prüfungssession verschoben und zwar:

1. Vordiplom

1. Prüfung	Informatik I+II	Donnerstag, 24. März
2. Prüfung	Analysis I+II	Dienstag, 29. März
3. Prüfung	Wahrscheinlichkeit + Statistik	Dienstag, 5. April
4. Prüfung	Elektrotechnik I+II	Freitag, 8. April
5. Prüfung	Algebra I+II	Montag, 11. April

2. Vordiplom

1. Prüfung	Physik I+II	Donnerstag, 24. März
2. Prüfung	Informatik III+IV	Dienstag, 29. März
3. Prüfung	Theoretische Informatik	Dienstag, 5. April
4. Prüfung	Elektrotechnik III+IV	Freitag, 8. April
5. Prüfung	Wissenschaftliches Rechnen	Montag, 11. April

Bitte beachten: Sie erhalten den perönlichen Prüfungsplan am 22. Februar 1994. Bis dahin gelten die obigen Angaben als provisorisch und es können immer noch Änderungen eintreten.

Die Schlussdiplomprüfungen verteilen sich über die Prüfungssession vom 14. März bis 15. April 1994.

Osterfeiertage: 1. bis und mit 4. April 1994.

Das Abteilungssekretariat

Solaris

I see that Sun has chosen the name "Solaris" for their new operating system.

In Stanislaw Lem's novel of the same name, Solaris was an alien planet/ intelligence that human explorers found to be utterly incomprehensible and psychologically devastating. Encounters with Solaris drove them to madness and death.

I'm glad that someone at Sun is finally getting it right.

unknown

G.A.B. 6648 Minusio

pal

Falls unzustellbar bitte zurück an:

Verein der Informatikstudierenden
IFW B29
ETH–Zentrum

CH–8092 Zürich

Inhalt