Skript: Einführung in die Informatik

skript

Klaus Lagally
Institut für Informatik
Breitwiesenstraße 20 - 22
D - 70565 Stuttgart

Zur Vorlesung

Die Vorlesung „Einführung in die Informatik I“ soll in die Begriffswelt der Informatik einführen und Grundlage sein für ein mehrsemestriges Studium der Informatik oder auch der Softwaretechnik, auch im Rahmen eines Nebenfaches. Außenstehende sind willkommen, mögen aber gelegentlich verwundert sein darüber, daß auch recht fortgeschrittene Konzepte hier schon, wenn auch kurz und oberflächlich, angesprochen werden. Das ist Absicht; trifft man auf diese Konzepte später nochmal, so erkennt man sie zumindest wieder und hat die erste Scheu schon verloren; damit fällt der Einstieg leichter, wenn es „Ernst wird“.

Die Vorlesung versteht sich nicht als Programmierkurs und leistet dies auch nicht. Programmieren lernt man überhaupt nicht durch Zuhören oder aus Büchern, sondern durch eigenes Tun: in den Übungen, in speziellen Kursen, und durch Versuch und Irrtum am Rechner (die dabei unvermeidlichen Mißerfolge gehören dazu). Unsere Programmbeispiele haben den Zweck, an ihnen neue Konzepte ein- und vorzuführen; es lohnt sich, sie recht genau anzusehen. Gute Beispiele zu finden ist recht schwer: sind sie zu klein, so lernt man nichts aus ihnen; sind sie zu groß, so kosten sie zuviel Zeit. Manchmal bringen wir daher nur Bruchstücke; ihre Vervollständigung ist oft offensichtlich.

Zum Skript

Dieses Skript gibt es in 2 Versionen: als druckbares Dokument und als eine Kollektion von verketteten WWW-Seiten. Es ist nicht zum sequentiellen Lesen gedacht (obgleich das möglich ist), sondern zum Blättern und Herumstöbern mittels der vielen Querverweise. Das geht auch mit der gedruckten Fassung (die Verweise sind markiert), aber die Hypertext-Fassung auf dem WWW ist bequemer zu handhaben. Inhaltlich sollten beide Fassungen identisch sein bis auf die Form.

Dieses Skript will und kann kein Lehrbuch ersetzen (das brauchen Sie außerdem), sondern ergänzen. Wir stellen hier einiges anders als gewohnt dar, bringen andere Beispiele und auch zusätzliche Informationen, die nicht unbedingt zum Prüfungsstoff gehören, aber gut zu wissen sind (oder die man hier bequemer findet). Dafür lassen wir viele Dinge, die in den Standard-Lehrbüchern gut dargestellt oder weitgehend bekannt sind, hier weg.

Das Skript besteht aus einzelnen Bausteinen, die jeweils ein Teilproblem ansprechen und in der Regel nicht mehr als eine Seite umfassen. Sie sind über Querverweise mit-

einander verbunden, so daß man auch fehlende Details nachschlagen kann; verliert man dabei die Übersicht, so kann man immer zum Inhaltsverzeichnis (S. 3) oder auch zur Stichwortliste (S. 151) zurückkehren.

Das Skript erhebt keinen Anspruch auf Originalität. Vieles steht in Lehrbüchern ebenso gut oder besser, und dort nachzuschlagen und zu vergleichen, sei den Lesern nachdrücklich ans Herz gelegt. Wir gehen dennoch manchmal unsere eigenen Wege, und manches mag über den normalen Umfang einer Einführung hinausgehen; dafür kommen manche traditionell breit abgehandelten Gebiete hier recht kurz weg. Manches fehlt auch ganz, weil man es woanders leicht finden kann.


Mit diesem Zeichen (das wir von D.E.Knuth übernommen haben, ohne um Erlaubnis zu fragen), haben wir Abschnitte markiert, die beim ersten Lesen übergangen werden können. Sie gehören durchaus zum Text, aber nicht zum Prüfungsstoff.


Mit dieser doppelten Warnung sind Seitenwege und Abschweifungen markiert, die gelegentlich weit über den Stoff hinausgehen, aber vielleicht dennoch interessant sind; entscheiden Sie selbst.

Zur Darstellung

Informatik ist, obwohl das Viele glauben, kein typisches Männerfach (manche Computerspiele schon eher), sondern erfahrungsgemäß für Alle gleich gut zugänglich; und daher haben wir auf durchgehend geschlechtsneutrale Formulierungen nicht peinlich streng geachtet; der Inhalt ist uns wichtiger. Die Darstellung ist gelegentlich informell und nicht von letzter Strenge, dafür hoffentlich anschaulich. Wenn Leser(innen) glauben, einiges hier Gebotene sei falsch oder auch nur ungeschickt dargestellt, so können sie damit durchaus recht haben. Wir sind für Hinweise sehr dankbar, und der sachlichen Kritik stellen wir uns gerne; die Informatik und auch ihre Didaktik sind noch sehr im Fluß. Es gibt sicher vieles zu verbessern, und so wird es vermutlich noch lange bleiben.

Wenn Sie Fehler oder Unklarheiten finden oder Anregungen, Wünsche oder Verbesserungsvorschläge haben, so lassen Sie mich das bitte wissen. Meine E-Mail-Adresse finden Sie hier[1] und am Fuße jeder WWW-Seite; Briefe lese ich auch. Telefonische Anfragen kommen leider fast immer ungelegen.

[1] `mailto:lagally@informatik.uni-stuttgart.de`

1 Inhaltsverzeichnis

inhalt

2 Organisatorisches 1999/2000

orga

- Meine Homepage [2]
- Meine Informatik-Erfahrungen (S. 8)
- Sprechstunden und Beratung:
  - Bitte fragen Sie mich *nicht* nach den Übungen[3]; darüber weiß ich nichts, weil sich Herr Gunnar Hilling [4] darum kümmert.
  - Kommen Sie ansonsten mit Prüfungsproblemen, fachlichen Fragen und auch gerne mit persönlichen Problemen zu mir; ich helfe gerne, wenn ich kann. Halten Sie sich aber bitte, wenn es nur irgend geht, an die Sprechstunden (immer dienstags ab 14:00 Uhr, ohne Anmeldung); sonst komme ich zu nichts anderem mehr (wegen BaFöG-Bescheinigungen können Sie jederzeit kommen, die gehen schnell).
  - Wenn es mal wirklich „brennt“, oder wenn eine Sache voraussichtlich länger dauern wird, so nehmen Sie Kontakt mit mir auf für einen Sondertermin. Das geht weitaus am besten über E-Mail[5]; telefonisch (0711/7816-392) bin ich nicht zuverlässig erreichbar außer gegen Abend, weil mein Sekretariat derzeit nicht besetzt ist. E-Mail geht schnell (ich sehe oft in mein Postfach), und sie unterbricht mich nicht bei anderen Tätigkeiten. Vieles läßt sich so auch direkt erledigen.
  - Wenn Sie Magister-Student(in) sind, so nehmen Sie Kontakt mit mir auf, soweit nicht schon geschehen, und kommen Sie mit allen Problemen jederzeit. Auch Ihre Prüfungsorganisation läuft über mich.

[2] `http://www.informatik.uni-stuttgart.de/ifi/bs/people/lagally.htm`
[3] `http://balsa.informatik.uni-stuttgart.de/`
[4] `http://www.informatik.uni-stuttgart.de/ifi/bs/people/hilling.htm`
[5] `mailto:lagally@informatik.uni-stuttgart.de`

3 Werdegang Klaus Lagally

werde

Meine Informatik-Erfahrungen gehen schon recht weit zurück, sind aber etwas unordentlich; nicht jede Jahreszahl habe ich nachgeprüft:

1959	Programmierkurs an der PERM (maschinennah)
1960	ALGOL 60 an der PERM
ca. 1960	erstes großes Assemblerprogramm
ca. 1961	FORTRAN II an einer IBM 7090
1964	Telefunken TR4
ca. 1965	TR4 Betriebssystem ausgedruckt und kommentiert
ab 1966	Wartung TR4 ALGOL-Compiler
1967	Promotion in Theoretischer Physik
ab 1968	Arbeitsgruppe Betriebssysteme (BSM) an der TU München
1968	Prozeßverwaltung und Synchronisation
1969	Mitarbeit am Sprachentwurf PS440
ab 1969	Mitarbeit am PS440-Compiler
ab 1969	Mitarbeit am Systementwurf BSM
ab 1970	Wartung PS440-Compiler
1973	Wissenschaftlicher Leiter der Arbeitsgruppe BSM
1975	erste Vorlesung Betriebssysteme
1975	Ruf an die Universität Stuttgart
seit 1976	zahlreiche Vorlesungen: Einführung in die Informatik I - III, Betriebssysteme, Systemprogrammierung, Programmiersprachen, Compilerbau, Formale Semantik, Theoretische Informatik I, und diverse kleinere

Wie Sie sehen, habe ich nie Informatik studiert (und es geht doch)

Seit etwa 1992 habe ich auch ein seltsames Hobby (zu deutsch: Steckenpferd): ich schlage mich mit arabischem Textsatz herum (System ArabTeX[6]). Meine früheren Aktivitäten als Funkamateur (DJ4TU) sind in den letzten Jahren völlig eingeschlafen.

[6] `http://www.informatik.uni-stuttgart.de/ifi/bs/research/arab.html`

4 weshalb Theorie?

theorie

- „**Theorie**“ muß nicht heißen „schwer verständlich“ oder „abstrakt“ (S. 10), sondern: *den Dingen auf den Grund gehen.*
- auf Deutsch: „**Durchblick**“ oder: das Wesen eines Problems verstanden haben; Theorie kommt also aus der **Praxis**.
- Theorie ohne Praxis ist manchmal „grau“, also steril.
- Praxis ohne Theorie wird leicht Bastelei oder Murks.
- Man braucht immer beides, Theorie und Praxis; dazwischen zu wechseln erfordert Übung, diese lohnt sich aber.
- Wenn Theorie schwer verständlich aussieht, dann liegt das immer an der Darstellung; lassen Sie sich das nicht gefallen!

  Bemerkung: In Goethe's „Faust“ steht:

  > „Grau, teurer Freund ist alle Theorie,
  > doch grün des Lebens goldner Baum!“

  das sagt dort nicht Faust, sondern Mephistopheles (ein Teufel[7] in Professorengestalt) beim Versuch, einen hilflosen Studenten irrezuführen; lassen Sie sich dadurch nicht beeindrucken, sondern finden Sie den logischen Fehler in seiner Aussage!

[7] `http://www.mathematik.uni-stuttgart.de/mathB/lst1/teufel/`

5 was ist Abstraktion?

abstrakt

- „**Abstraktion**“ heißt: das Unwesentliche an der Beschreibung eines Problems weglassen.
- Dann ist das Problem immer noch da, aber seine Beschreibung ist einfacher geworden, und die Chancen, es zu lösen, sind größer.
- Um zu erkennen, was wesentlich und was unwesentlich ist, braucht man ein tieferes Verständnis, also Theorie (S. 9).
- Hier liegt das Hauptproblem bei den „Textaufgaben“, die bei Schülern (und nicht nur bei ihnen) unbeliebt sind: das Abstrahieren macht Mühe.
- Ein wichtiger Schritt wird oft vergessen: hat man ein Problem auf eine einfachere abstrakte Ebene gehoben und dort gelöst, so muß man das Ergebnis noch auf die Stufe der ursprünglichen Beschreibung zurückübersetzen. Das ist eine mechanische und lästige Arbeit: „Geben Sie das Ergebnis in Worten an“ (wer kennt und haßt das nicht), aber notwendig; sonst hat man für den ursprünglichen Auftraggeber auf seiner Denk-Ebene das Problem anscheinend nicht gelöst.
- Die Erfahrung zeigt: *abstrahieren* und später auch wieder *konkretisieren* (zurückübersetzen) kann man nicht von alleine (Genies ausgenommen), aber man kann es gut erlernen, und Üben hilft viel.

6 Was ist eine Definition?

define

Das Wort „**Definition**“ führt manchmal zu Verwirrung, weil es zweierlei Bedeutungen haben kann:

- In der **Umgangssprache** und in vielen Geisteswissenschaften bedeutet es:
  - die **Bedeutung** eines *existierenden* Wortes klären durch Vergleich mit dem Gegenteil und mit verwandten Begriffen, gegen die es sich absetzt.
- in **Fachsprachen** (S. 12) hat es eine andere Funktion:
  - für ein *neues* (oder auch für ein schon existierendes) *Wort* wird eine *neue* (!) Bedeutung festgelegt, die bis auf weiteres gültig bleibt und die bisherige Bedeutung (so es denn eine solche gab) völlig **überdeckt**.
  - Es handelt sich folglich um eine reine **Sprachregelung**; für jede Verwendung des neuen Wortes kann man (mit den nötigen grammatikalischen Anpassungen) die gegebene Erklärung einsetzen, ohne daß sich an der Bedeutung das Geringste ändert.
  - Es gibt also daran *gar nichts* zu verstehen; es kann nützlich sein, neue Definitionen auf ein Blatt zu notieren, sodaß man sie bei Bedarf zur Hand hat. Oft lohnt es sich nicht, sie auswendig zu lernen, wenn sich absehen läßt, daß man sie später nicht mehr braucht. Umgekehrt lohnt es sich oft, zur Vereinfachung der Ausdrucksweise selbst eine lokale Definition einzuführen; keine Scheu davor!
  - Eine Definition in diesem Sinne hat einen wohlbestimmten (wenn auch nicht immer ausdrücklich angegebenen) **Gültigkeitsbereich**. Wird er verlassen, so lebt die bisherige Bedeutung des Wortes (falls es eine solche hatte) wieder auf.
- Definitionen im zweiten Sinne kommen auch innerhalb von Programmen in höheren Programmiersprachen (S. 13) vor; dort heißen sie manchmal auch „Deklaration“ oder „Vereinbarung“, und auch hier muß man auf den Gültigkeitsbereich achten!

7 Fachsprache und Jargon

jargon

- **Fachsprachen** unterscheiden sich von der Umgangssprache dadurch, daß gewisse Wörter (**Termini**) in einer *eindeutigen*, gewöhnlich durch eine Definition (S. 11) festgelegten Bedeutung *ausschließlich* gebraucht werden.
- **Wortwiederholungen** gelten in der Umgangssprache und auch in literarischen Texten als schlechter Stil. In der Fachsprache ist es *umgekehrt*: ein festgelegter Begriff wird, wenn es möglich ist, *immer* durch denselben definierten Terminus ausgedrückt. Sprachliche Härten spielen keine Rolle; die Klarheit ist wichtiger.
- Unter **Jargon** versteht man das Durchsetzen alltäglicher Aussagen mit Ausdrücken aus einer Fachsprache, ohne daß das nötig oder auch nur hilfreich wäre (oft: um sich wichtig zu machen). Das Ersetzen des Fachausdrucks durch seine Bedeutung oder das Ausdrücken desselben Sachverhalts mit einfachen Worten wirkt oft sehr entlarvend!
- *Warnung*: eine häufige *Prüfungsfrage* ist:
  gegeben ein Satz im Jargon; sagen Sie das Gleiche auf Deutsch mit einfachen Worten.
- *Guter Rat*: versuchen Sie so oft wie möglich, eine Aussage in der Fachsprache einem Außenstehenden in der Umgangssprache zu erklären. Wenn Sie das können, haben Sie es verstanden.

8 wichtige Programmiersprachen

proglang

Eine Statistik im Jahre 1970 hat ca. 700 **Programmiersprachen** aufgeführt; inzwischen sind es weitaus mehr geworden. Den vollen Überblick hat wohl niemand.

Historisch interessant und teilweise heute noch wichtig (etwa chronologisch):

LISP	(LISt Processing) für KI-Anwendungen
FORTRAN	für Ingenieur-Anwendungen (viele Versionen)
COBOL	für kaufmännische Anwendungen
ALGOL	für wissenschaftliches Rechnen und zur Dokumentation; veraltet
PL/I	von IBM; als Ersatz von FORTRAN und COBOL
Assembler	maschinennahe Programmierung (viele Varianten)
APL	erste interaktive Sprache; leider schwer lesbar
SNOBOL	zur Textverarbeitung
SETL	baut auf Mengenlehre auf
PEARL	zur Steuerung von industriellen Prozessen
CHILL	für Telefon-Vermittlungsanlagen
Pascal	allgemeine Anwendungen; einfach, für Ausbildung gedacht und auch gut dafür geeignet
Modula 2	wie Pascal; getrennte Übersetzung von Programmteilen
Modula 3	eine objektorientierte Erweiterung von Modula 2
Oberon	eine andere objektorientierte Erweiterung von Modula 2
BASIC	primitiv; für „Laien“ gedacht, fast unbrauchbar
LOGO	einfacher Dialekt von LISP (für Grundschule) mit einfacher Grafik
Scheme	Erweiterung von LISP für allgemeine Anwendungen
C	ursprünglich für Betriebssysteme (UNIX) gedacht, aber auch sonst weit verbreitet
SQL	zur Abfrage von relationalen Datenbanken
QBE	Datenbankabfragen, formularbasiert
Ada	allgemeine Anwendungen, komfortabel, für industriellen Einsatz
Simula 67	erste „objektorientierte Sprache“ (1967!), leider kostenpflichtig
Smalltalk	zweite objektorientierte Sprache, sehr flexibel
C++	objektorientierte Erweiterung von C; sehr verbreitet, aber etwas unsicher und daher riskant
EIFFEL	objektorientiert; schön, aber kostenpflichtig
JAVA	objektorientiert; maschinenunabhängig, für WWW-Anwendungen
Prolog	logikbasiert; zur Wissensverarbeitung
Perl	komfortable Sprache zur Textverarbeitung

Es kommen laufend neue Sprachen und Dialekte dazu.

Einige der ersten Programmiersprachen, etwa LISP und ALGOL, wurden ursprünglich zur präzisen **Dokumentation** von Algorithmen entwickelt, die dann von Hand

in eine maschinennahe Form für die gerade verfügbare Maschine übertragen wurden. Erst später kamen Übersetzer (Verfahren zur automatischen Umwandlung in direkt ausführbaren Code) dazu.

Die „beste“ Programmiersprache gibt es nicht; aber wer einmal eine hinreichend mächtige Sprache kennengelernt hat, kann bei Bedarf schnell auf eine andere umsteigen. Dabei muß er sich oft nur an eine andere **Notation** (S. 14) gewöhnen; (deren Übersetzer hilft ihm dabei).

Diese Vorlesung verwendet Modula 2 (S. 15) zur Demonstration von Programmbeispielen.

9 Bemerkungen zur Notation

notation

Die Grundkonzepte der meisten klassischen Programmiersprachen sind gar nicht so sehr verschieden, doch sehen Programme, die dasselbe tun, oft sehr unterschiedlich aus. Ein typisches Beispiel dafür ist die Notation der Zuweisung (S. 41), die in den auf den „Urvater“ ALGOL 60 zurückgehenden Sprachen (ALGOL 68, Pascal, Modula, Oberon, Ada, Eiffel) in der Form <V> := <E> geschrieben wird. Sprachen in der Tradition von FORTRAN (PL/I, C, C++, Java etc.) schreiben dafür <V> = <E>, was leicht mit dem logischen Vergleichsoperator „ = “ verwechselt wird. Dieser muß infolgedessen anders geschrieben werden („ .EQ. “ in FORTRAN und PL/I, „ == “ in C und seinen Nachfolgern), was man immer wieder übersieht, und das gibt dann schwer zu findende Fehler. Natürlich kann man sich das durch eine entsprechende Dressur angewöhnen (manche Leute verwechseln das mit *Fachkompetenz*), aber wir finden es besser, für eine grundlegend *neue* Operation (und das *ist* die Zuweisung) auch eine *neue* Notation einzuführen. Diese muß man dann zwar auch *lernen*, aber man braucht seine sonstigen Denkgewohnheiten nicht zu *verlernen*, sondern kann sie zusätzlich in die Problemlösung mit einbringen, und das ist offensichtlich vorteilhaft.

Leider hat sich die schon 1976 vorgeschlagene Notation <E> ⇒ <V>, die klar und auch nahe an der technischen Realisierung ist, aus Beharrlichkeit (oder Trägheit, oder schlechtem Marketing) nicht durchgesetzt.

10 weshalb MODULA 2 als Beispiel-Sprache? modula

Als Programmiersprache für unsere Programmbeispiele verwenden wir hier nicht C oder C++, Ada, Java, Eiffel oder eine andere gerade „moderne“ Sprache, sondern Modula 2. Dies hat mehrere Gründe:

- Die Sprache ist “schlank“ und belastet daher das Gedächtnis nicht stark; das ist wichtig, weil jeder später auf andere Programmiersprachen (S. 13) umsteigen wird, aber welche das sein werden, ist heute noch nicht abzusehen.
- Die Sprache ist mächtig genug, um alle wesentlichen Konzepte an ihr vorzuführen, und sie ist auch für größere Projekte geeignet.
- Es gibt kostenlose oder billige Realisierungen (S. 15) für alle wichtigen Rechnersysteme.
- Die Sprache hat deutliche Grenzen in der Ausdrucksfähigkeit; dadurch lernt man, sich notfalls zu behelfen und zu improvisieren.

Die Verwendung von Modula 2 hat durchaus auch Nachteile:

- Ein Nachteil: die verfügbaren Bibliotheken sind nicht völlig standardisiert; das gibt bei der Übertragung auf ein anderes Rechnersystem womöglich Ärger (der ist allerdings recht lehrreich!)
- Ein Nachteil: die Sprache ist in der Industrie nicht recht verbreitet. Sie werden also später auf andere Sprachen (S. 13) umsteigen müssen; das sollte Ihnen aber dann leicht fallen.

Wir glauben, daß wir mit Modula 2 einen brauchbaren Kompromiß gefunden haben.

11 freie MODULA-Systeme modreal

- MODULA 2 für DOS[8] (Shareware)
- GPM[9] für verschiedene Plattformen

[8] `http://129.69.218.213/~lagally/ifi/bs/lehre/ei1/modula.zip`
[9] `ftp://pegasus.dvz.fh-aachen.de/pub/modula/gpm/`

12 Objekte und Klassen (1)

objekt1

- Unter einem **Objekt** versteht man in der Informatik zweierlei:
  - ein Ding in der realen Welt
  - ein programmiertes Modell davon

  Die Modellierung kann unterschiedlich detailliert sein, je nach Anwendungsfall.
- Ein Objekt hat einen **Zustand** (*state*), der durch **Attribute** (Eigenschaften) beschrieben wird,
  und **Methoden**, die angeben, was man mit dem Objekt tun kann
  (bei „aktiven Objekten“: was das Objekt tun kann).
- Objekte gleicher Art faßt man zu **Klasse**n zusammen.
  Ein Objekt ist eine **Instanz** einer Klasse.
  Eine Klasse liegt oft in einer (Klassen-)**Bibliothek**.

  Beispiel:

  einige Attribute eines Druckers aus der Klasse „Drucker“:
  - ein/aus?
  - Papier drin?
  - Papierformat
  - aktuelle Druckposition
  - Inhalt aktuelle Seite
  - noch zu druckende Daten

  einige Methoden eines Druckers:
  - ein/ausschalten
  - Papier einlegen
  - Papier herausnehmen
  - zu druckende Daten senden

13 Objekte und Klassen (2)

objekt2

- Eine Klasse kann von einer „Oberklasse“ Attribute und Methoden **erben**.

  Beispiel:
  Die Klasse „Drucker“ erbt von der Klasse „Gerät“:
  - Attribut „Farbe“
  - Methode „anmalen“
  - Attribut „Ort“
  - Methode „aus dem Fenster werfen“

Eine Konsequenz der **Vererbung** ist, daß ein Objekt nun in verschiedenen **Rollen** auftreten kann (**Polymorphismus**), nämlich als Instanz seiner eigenen Klasse oder als Instanz einer Oberklasse; das kann je nach Art der Verwendung zu Überraschungen führen.

Wir haben das Problem hier nicht und ignorieren es daher bis auf weiteres; Vererbung spielt in dieser Vorlesung keine Rolle.

- Auch eine *Klasse* kann *Attribute* haben (etwa die Anzahl der gerade existierenden Instanzen) und auch *Methoden* (etwa zum Einrichten oder Löschen von Instanzen).

  In manchen „objektorientierten Sprachen“ faßt man Klassen daher auch als Objekte auf, die Instanzen einer Klasse „class“ sind. Wir verwenden dies hier nicht.

Bemerkung: Oft wird Objektorientierung so erklärt, daß Objekte *aktive Einheiten* sind, welche miteinander über **Nachrichten** kommunizieren und daraufhin selbst ihre Methoden ausführen.

Wir schätzen das nicht; das ist zwar eine von mehreren möglichen Realisierungen, aber meistens geht es anders viel einfacher; es wird daher intern auch meist anders gemacht. Nur wenn die Objekte räumlich verteilt angeordnet sind, kommt man um die Nachrichten nicht herum.

14 erstes Programm in MODULA 2

hello

Unser erstes Programm soll nichts weiter können, als einen Begrüßungstext ausgeben; dabei können wir überprüfen, ob unsere Modula 2 (S. 15)-Umgebung im Prinzip funktioniert.

Wir geben den Text nicht direkt auf den Bildschirm aus, dessen genaue technische Ansteuerung wir nicht kennen (und die uns auch gar nicht interessiert), sondern auf eine gedachte **Ausgabefläche**, die aus beliebig vielen Text-Zeilen von beliebiger Länge besteht. Ihr Zustand ist bestimmt durch den aktuellen sichtbaren Inhalt und die aktuelle Schreibposition; er kann durch Methoden verändert werden, von denen es eine größere Anzahl gibt. Die Abbildung des abstrakten Objekts auf den konkreten Bildschirm veranlaßt das Betriebssystem automatisch.

```
MODULE ErsterTest;
  (* Demo fuer Ausgabe von Text *)
  (* Vorlesung EI1 WS 1999/2000 *)
  (* 19.10.1999, Klaus Lagally  *)

FROM InOut IMPORT WriteLn, WriteString;

BEGIN WriteString("Hello, world!");
      WriteLn
END ErsterTest.
```

Im Kopf des Programms steht sein Name sowie Angaben über Zweck, Kontext, Verfasser und Änderungsstand (mindestens!).

Aus der Bibliothek „InOut“ werden die Methoden “WriteLn“ und „WriteString“ der Ausgabefläche importiert. Ihre Klasse enthält nur eine einzige Instanz, daher hat (und braucht) dieses Objekt im Programm keinen eigenen Namen.

Die Methoden werden nacheinander aufgerufen; dies bedeutet der Strichpunkt (=Semikolon) in Modula. In manchen anderen Sprachen bezeichnet er das Ende einer Anweisung (S. 41). Am Programmende steht zur Kontrolle nochmal der Programmname.


Recht häufig wird das „Hello, world“-Problem auch zur Vorstellung anderer Programmiersprachen und Programmierstile[10] verwendet.

[10] http://pcbs13.informatik.uni-stuttgart.de:80/~lagally/fun/evol.txt

15 Mengen (1)

menge1

Obgleich es parallel eine Mathematik-Vorlesung gibt, sprechen wir manche Dinge, die dort behandelt werden, hier auch an, weil wir einige Resultate und Begriffe vorweg brauchen, und weil wir wissen, daß nicht alle Teilnehmer im Gymnasium Mengenlehre gehabt haben. Wir gehen dabei nicht streng vor und verzichten vorerst auf Beweise; dies wird an gegebener Stelle repariert.

> Unter einer **Menge** verstehen wir jede Zusammenfassung von bestimmten, wohlunterschiedenen Objekten unserer Anschauung oder unseres Denkens (welche **Elemente** der Menge genannt werden) zu einem Ganzen.
> (nach G. Cantor, 1895)

Mengen werden angegeben entweder durch Aufzählen ihrer Elemente:

M1 = { a, b, c }

dabei kommt es auf die Reihenfolge nicht an; oder durch Angabe einer Eigenschaft, die *alle* ihre Elemente und *nur* sie haben:

M2 = { alle 5DM-Stücke in meinem Portemonnaie }
M3 = { alle geraden ganzen Zahlen }

N = { alle natürlichen Zahlen }
Z = { alle ganzen Zahlen }
Q = { alle rationalen Zahlen }
R = { alle rellen Zahlen }

Die zuletzt genannten Mengen sind **unendlich**; wir setzen sie als bekannt voraus.
Wichtig ist noch die Menge der „**Wahrheitswerte**“:

B = { FALSE, TRUE }


Auch gegebene Mengen kann man wieder zu Mengen zusammenfassen; eine „Menge aller Mengen“ gibt es allerdings nicht! Überhaupt muß man darauf achten, daß die Elemente einer neu gebildeten Menge bereits wohlbestimmt sind, sonst kann man in logische Widersprüche kommen.

Den aus der Mathematik bekannten Zahlenmengen entsprechen in den üblichen Programmiersprachen **Datentyp**en; eigentlich sind das Klassen (S. 16), weil es auf die definierten Operationen (Methoden) auch ankommt. Neben **Grundtypen** (S. 22) (auch „einfache Typen“) gibt es auch **abgeleitete Typen** (S. 24), die mittels bereits bekannter Typen aufgebaut werden.

16 Mengen (2)

menge2

Aus gegebenen Mengen kann man neue Mengen bilden:

Die **Vereinigung** M1∪M2 von zwei Mengen M1 und M2 besteht aus den Dingen, die Elemente von M1 *oder/auch* Elemente von M2 sind.

$$M1 \cup M2 = \{ x \mid (x \in M1) \vee (x \in M2) \}$$

Der **Durchschnitt** M1∩M2 von zwei Mengen M1 und M2 besteht aus den Dingen, die sowohl Elemente von M1 wie auch *gleichzeitig* Elemente von M2 sind.

$$M1 \cap M2 = \{ x \mid (x \in M1) \wedge (x \in M2) \}$$

Nach dieser Verabredung ist der Durchschnitt von zwei Mengen, die *gar kein* Element gemeinsam haben, auch eine Menge. Diese heißt die **leere Menge** ∅; sie einzuführen, ist praktisch, weil es Sonderfälle einspart.

Wenn alle Elemente einer Menge M1 auch Elemente einer Menge M2 sind, so nennt man M1 eine **Teilmenge** von M2: M1 ⊆ M2.

$$M1 \subseteq M2 \iff (x \in M1) \Rightarrow (x \in M2)$$

Ist M1 Teilmenge von M2 und M2 *gleichzeitig* Teilmenge von M1, so betrachten wir die Mengen als *gleich*.

Man sieht sofort: der Durchschnitt M1∩M2 von zwei Mengen M1 und M2 ist Teilmenge sowohl von M1 wie von M2, und M1 und M2 sind beides Teilmengen ihrer Vereinigung M1∪M2.

Unter der **Mächtigkeit** einer Menge M verstehen wir, wenn M **endlich** ist, die Anzahl **Card(M)** der Elemente; und wir verallgemeinern dies auf unendliche Mengen: zwei Mengen heißen **gleichmächtig**, wenn man sie umkehrbar eindeutig aufeinander beziehen kann.

Die Mengen **N**, **Z** und **Q** sind gleichmächtig: sie sind **abzählbar**, man kann ihre Elemente geeignet durchnumerieren.

Die Mengen **R** und **C** (und auch die Menge der ganzzahligen Funktionen auf **N**) sind gleichmächtig: sie sind **überabzählbar**.


Ob es zwischen der Mächtigkeit von **N** und der von **R** noch eine weitere Mächtigkeit gibt, ist unbekannt. Mengen von noch höherer Mächtigkeit als **R** gibt es, etwa die Menge der reellen Funktionen.

Weitere Konzepte aus der Mengenlehre sprechen wir erst dann an, wenn wir sie brauchen.

17 Kartesisches Produkt

kartes

Für zwei Mengen M und N definieren wir als **kartesisches Produkt** M×N: die Menge aller geordneten Paare <m,n>, wobei gilt m ∈ M und n ∈ N.

$$M \times N = \{ <m,n> \mid m \in M \wedge n \in N \}$$

Entsprechend bilden wir kartesische Produkte aus mehreren Faktoren:

$$A \times B \times C = \{ <a,b,c> \mid a \in A, b \in B, c \in C \}$$

und wir betrachten die Produktbildung als **assoziativ**:

$$(A \times B) \times C = A \times (B \times C) = A \times B \times C$$

Ein kartesisches Produkt ist eine Menge wie andere auch, und es kann daher auch als Definitionsbereich einer Funktion (S. 38) vorkommen. Dabei erhalten wir etwas Wohlbekanntes im neuen Gewande zurück:

$$f : M \times N \rightarrow L$$
$$<m,n> \mapsto f(<m,n>) = g(m,n)$$

also eine Funktion von mehreren Veränderlichen, wenn wir g(m,n) anstatt f(<m,n>) schreiben.

Ebenso könnte der Bildbereich (S. 38) einer Funktion ein kartesisches Produkt sein, und Anwendungen dafür gibt es durchaus; allerdings denkt man nicht leicht daran, weil die Mathematik traditionell dafür keine passende Notation entwickelt hat, und die klassischen imperativen (S. 41) Sprachen unterstützen das auch nicht, wieder aus historischen Gründen (in funktionalen Sprachen wie LISP (S. 13) gibt es das sehr wohl).


Eine Anwendung dafür ist uns allen vertraut: die ganzzahlige Division liefert *zwei* Werte zurück, nämlich den Quotienten und den Rest. Das ist nichts anderes als ein kartesisches Produkt, aber es fehlt an einer passenden Notation; und deshalb muß man in Modula 2, wenn man beide Werte braucht, leider zweimal dividieren, obgleich bei jeder Division beide anfallen.

18 Grundtypen (1)

basis1

Programmiersprachen bieten für die aus der Mathematik kommenden Zahlenmengen (S. 19) Näherungen (**Datentyp**en) an, die man allerdings mit den Originalen nicht verwechseln sollte: die Zahlenmengen sind unendlich, während wir in einem Computer immer nur endlich viele Werte in jeweils endlicher Darstellung unterbringen können. Es ist nützlich, Datentypen als Klassen (S. 16) aufzufassen, also die definierten Operationen jeweils mit zu betrachten, und wir gehen auch bei den Zahlenmengen selbst so vor, beschränken uns allerdings auf eine Auswahl an Operationen. Alle hier besprochenen Objekte besitzen als Attribute

- einen (mathematischen) **Wert**
- eine **interne Darstellung**, die den Anwender nichts angeht,
- eine **externe Darstellung** (oder evtl. mehrere davon), die man mit dem Wert nicht verwechseln darf!

- Ausgehend von den natürlichen Zahlen **N** mit den Operationen Addition (+), Subtraktion (−), Multiplikation (*), ganzzahlige Division (DIV) und Restbildung (MOD) bietet Modula (anders als die meisten anderen Sprachen) den Datentyp **CARDINAL** an, der die Null und ein Anfangsstück der natürlichen Zahlen umfaßt (die Obergrenze ist implementierungsabhängig). Die Operationen sind die gleichen, aber man muß auf zweierlei achten:
  - Die Operation muß definiert sein (etwa keine Subtraktion größerer Zahlen von kleineren, keine Division oder Restbildung durch Null)
  - Das Ergebnis muß wieder im Bereich CARDINAL liegen. Andernfalls ist das zurückgelieferte Ergebnis schlicht falsch, oder aber unser Programm „stürzt ab“.
- Den ganzen Zahlen **Z** entspricht als Datentyp **INTEGER** ein um Null symmetrischer Ausschnitt, mit den gleichen Operationen und den gleichen Vorsichtsmaßregeln.

Für beide Datentypen (und noch andere!) sind außerdem noch eine Reihe von **Vergleichsoperationen** <, =, >, <=, >=, <> (für ungleich) definiert; sie liefern als Ergebnis:

- Wahrheitswerte aus dem Typ **BOOLEAN** = {FALSE, TRUE}. Für diesen Typ sind die aus der Logik bekannten Operationen **AND**, **OR** und **NOT** definiert (oft auch noch andere, die aber geringe Bedeutung haben).

19 Grundtypen (2)

basis2

An Stelle der rationalen Zahlen **Q** mit den Operationen Addition (+), Subtraktion (−), Multiplikation (*) und Division (/) und der rellen Zahlen **R** mit den gleichen Operationen gibt es in Modula (S. 15) wie in anderen Programmiersprachen einen Typ **REAL**, der aus einer *endlichen* Anzahl von *rationalen* Zahlen besteht. Die Ergebnisse von Rechenoperationen mit Werten dieses Typs sind *grundsätzlich ungenau*, daher machen die (vorhandenen) Vergleichsoperationen = und <> hier wenig Sinn!


Der zweckmäßige Umgang mit Werten vom Typ REAL so, daß sich die Ungenauigkeiten bei einer längeren Rechnung nicht zu sehr aufschaukeln, ist Gegenstand eines eigenen Wissenschaftszweiges: **numerische Mathematik**.

Als Grundlage für die Bearbeitung von Texten haben wir in allen modernen Sprachen schließlich noch den Datentyp **CHAR**. Er umfaßt mindestens alle druckbaren und über eine Tastatur eingebbaren Zeichen einschließlich des Leerzeichens (" "); oft sind noch andere Spezialzeichen über Hilfskonstrukte erreichbar. Der zu Grunde liegende Zeichenvorrat ist in Modula (S. 15) (und in den meisten anderen Sprachen) heute der „American Standard Code for Information Interchange“ **ASCII** (S. 25), oft mit nationalen Erweiterungen (wie die deutschen Umlaute), die bei der Übertragung auf andere Systeme manchmal für Überraschungen sorgen. Abweichend davon verwenden Großrechnersysteme (historisch bedingt) oftmals noch den von IBM entwickelten Code **EBCDIC**. Seit einigen Jahren wird unter der Bezeichnung **UNICODE** versucht, einen für alle Sprachen der Welt ausreichenden Code zu definieren, doch hat sich diese Codierung (außer in der Sprache JAVA (S. 13)) bisher nicht breit durchgesetzt.

In Modula werden Werte vom Typ CHAR in Paaren von einfachen (') oder doppelten (") Anführungszeichen notiert. Auf dieselbe Weise können auch **Zeichenketten** („Strings“) angegeben werden; doch ist zwischen einem Einzelzeichen und einer Zeichenfolge der Länge 1 logisch ein Unterschied! Die Zeichenfolge der Länge Null ("") gibt es auch.

Zeichen treten nicht nur als Bestandteile von (**alphanumerisch**en) Texten auf, sondern auch, vor allem in der Mathematik, als **Name**n von Objekten. Dabei zeigt sich oft, daß der verfügbare Zeichenvorrat nicht ausreicht; Abhilfe schafft der Gebrauch von längeren Zeichenketten, von **Bezeichnern** (S. 26), als Namen.

20 Skalare Typen

skalar

Die Grundtypen (S. 22) CARDINAL, INTEGER, BOOLEAN, CHAR in Modula 2 haben die Eigenschaft, daß die dazugehörigen Werte in einer Folge hintereinanderliegen. Solche Typen nennt man **skalare Typen**; REAL gehört nicht dazu!

In Modula 2 (und bereits in Pascal) lassen sich weitere skalare Typen bilden:

- **Aufzählungstypen** durch direkte Angabe der zugehörigen Werte. Man gibt einen Typnamen und die Folge der Namen der dazugehörigen Werte direkt an. Beispiel:

  ```
  TYPE Farbe = (rot, orange, gelb, gruen, blau, violett);
  ```

  Der Gültigkeitsbereich (S. 11) der so eingeführten neuen Bezeichner ist der zugehörige Block (S. 53).

  Die interne Darstellung der Werte eines Aufzählungstyps braucht den Benutzer nicht zu interessieren. Ein/Ausgabe-Operationen sind nicht standardmäßig vorgegeben.

  Auch BOOLEAN = (FALSE, TRUE) kann als Aufzählungstyp aufgefaßt werden, doch gibt es hier noch weitere Operationen, etwa AND, OR und NOT.

- **Bereichstypen** durch Ausschnittbildung aus einem schon vorhandenen skalaren Typ. Beispiel:

  ```
  TYPE Byte = (0 .. 255);
  ```

  Die Werte eines Bereichstyps sind auch Werte des zugehörigen Grundtyps und verhalten sich genauso; Ein- Ausgabeoperationen sind die des Grundtyps, falls sie dort vorhanden sind. An eine Variable eines Bereichstyps dürfen aber nur Werte aus diesem Bereich zugewiesen werden.

Für alle skalaren Typen sind als Operationen erklärt:

- Vergleiche =, <>, <, <=, >, >= ; bei Aufzählungstypen im Sinne der Reihenfolge
- ORD(n) gibt die Ordnungsnummer des Wertes n
- SUCC(n) ist der Nachfolger von n, falls existent
- PRED(n) ist der Vorgänger von n, falls existent

weiterhin ist mit ihnen möglich

- Zuweisung (S. 41) an Variable (S. 56) desselben Typs

- Steuerung einer CASE-Anweisung (S. 46)
- Auswahl einer RECORD-Variante (S. 60)
- Verwendung als Indextyp eines ARRAY-Typs (S. 59) (nicht mit INTEGER oder CARDINAL!)

21 Der ASCII-Code

ascii

American Standard Code for Information Interchange

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	IS4	,	<	L	\	l	|
D	CR	IS3	-	=	M	]	m	}
E	SO	IS2	.	>	N	^	n	~
F	SI	IS1	/	?	O	_	o	DEL

Dies sind insgesamt 128 Zeichen, von denen die ersten 32 und das letzte Zeichen mit Sonderfunktionen belegt sind; darauf beschränkt sich der **ASCII**-Standard.


Es gibt zahlreiche verschiedene Erweiterungen dieses Standards um weitere 128 Zeichen, etwa für Strichgraphik am Bildschirm oder für nationale Sonderzeichen; in Modula sollte man sie besser nicht verwenden, sonst ist die Austauschbarkeit der Programme gefährdet, von sonstigen Überraschungen ganz abgesehen!

In Modula können wir jedes der 128+128 Zeichen notfalls mittels der Standardfunktion CHR(n) (S. 36) erzeugen, wobei n die Platznummer in der Tabelle ist.

22 Griechisches Alphabet; Bezeichner

greek

In der Mathematik werden zur Bezeichnung allgemeiner Größen traditionell Einzelbuchstaben verwendet. Im Laufe der Zeit hat sich gezeigt, daß diese knapp werden, zumal viele von ihnen durch eine Belegung mit einer Standardbedeutung nahezu verbraucht sind. Auch das Ausweichen auf Großbuchstaben, Fraktur und fremde Alphabete hat das Problem nicht gelöst, zumal sich dann bei den üblichen Textverarbeitungssystemen (mit Ausnahme von TEX[11]) oft erhebliche Schwierigkeiten ergeben. Als Beispiel (weil man es immer wieder mal braucht) hier das **Griechisch**e Alphabet.

A	α	alpha	I	ι	iota	P	ϱ	rho
B	β	beta	K	κ	kappa	Σ	σ	sigma
Γ	γ	gamma	Λ	λ	lambda	T	τ	tau
Δ	δ	delta	M	μ	my	Υ	υ	ypsilon
E	ε	epsilon	N	ν	ny	Φ	φ	phi
Z	ζ	zeta	Ξ	ξ	xi	X	χ	chi
H	η	eta	O	o	omikron	Ψ	ψ	psi
Θ	ϑ	theta	Π	π	pi	Ω	ω	omega

Abhilfe hat in der Informatik (nach einigen Anläufen in der Mathematik, etwa bei den Standardfunktionen) die generelle Verwendung von **Bezeichner**n gebracht: Zeichenfolgen, die mit einem Buchstaben beginnen und mit Buchstaben oder Ziffern fortgesetzt werden können (die genauen Regeln sind etwas sprachabhängig); davon gibt es beliebig viele, auch bei einem kleinen Basis-Zeichensatz.

Allerdings muß man nun die **Multiplikation** durch ein eigenes Zeichen ausdrücken; der Stern (*) hat sich dafür allgemein durchgesetzt (der Punkt wird auch anderweitig gebraucht, das Malkreuz $\times$ ist oft nicht verfügbar und wird auch zu leicht mit dem Buchstaben x verwechselt).

Zur Erinnerung: Jeder **Bezeichner** hat, wie andere Abkürzungen auch, einen begrenzten **Gültigkeitsbereich** (S. 11)!

[11] `http://www.dante.de`

23 Wie löst man Probleme?

problem

Wir verstehen als Aufgabe und Tätigkeit eines Informatikers nicht in erster Linie das Schreiben von *Programmen*, sondern das *Lösen von Problemen*, sowohl von eigenen wie auch von solchen der Anwender; also über eine **Problemanalyse** zu einem **Systementwurf** zu kommen. Dazu gibt es zwar kein durchgängiges Verfahren, jedoch eine Reihe von bewährten **Strategien** und nützlichen **Tipps**.

Eine große Schwierigkeit in der Praxis besteht darin, daß viele Problembereiche außerordentlich umfangreich und komplex sind; man braucht also Strategien zur *Beherrschung der* **Komplexität**. Dabei ist es hilfreich, nicht nur ein einzelnes **Problem** anzugehen, sondern zu versuchen, eine ganze **Problemklasse** zu lösen. So hat man einen besseren Wirkungsgrad und erhält im Idealfall **wiederverwendbar**e **Baustein**e.

Häufig beobachtet man, daß ein allgemeineres Problem leichter und eleganter lösbar ist als ein spezielles (das damit *auch* gelöst ist). Der Grund dafür ist die dabei vorgenommene Abstraktion (S. 10), also das Weglassen von für die spezielle Instanz (S. 16) spezifischen Einzelheiten; man überblickt so das Wesentliche besser (das ist Theorie (S. 9)).

Den Instanzen aus einer Problemklasse kann man jeweils ein geeignetes ganzzahliges **Größenmaß** zuordnen (oft gibt es mehrere davon). Damit lautet unsere **Grundstrategie**:

- kleine Probleme: direkt lösen
- große Probleme: nach der Regel *divide et impera* (teile und herrsche) in kleinere Teilprobleme herunterbrechen und deren Lösungen wieder richtig zusammensetzen. Die Teilprobleme betrachten wir in diesem Zusammenhang erst als **„schwarzen Kasten“**, dessen Auswirkung durch einen **Kontrakt** beschrieben wird; und wir untersuchen sie anschließend für sich weiter. So folgen wir dem **Geheimnisprinzip**: wir verwenden nicht mehr Information als lokal benötigt wird.

Für das einzelne Teilproblem kommen, sofern es nicht direkt lösbar ist, jeweils zwei Vorgehensweisen in Frage:

- es ist eine Instanz der gleichen Problemklasse: **Rekursion** (S. 29)
- es gehört zu einer anderen Problemklasse: **Modularisierung** (S. 28)

Beides wird durch Modula 2 (S. 15) gut unterstützt. Wir zeigen dies an einem ausführlichen *Beispiel* (S. 31).

24 Modularisierung

modul

Unter einem **Modul** (*das* Modul; Betonung auf der zweiten Silbe) verstehen wir in Modula 2 eine selbständige Übersetzungseinheit. Davon gibt es verschiedene Sorten:

- Programm-Module: sie enthalten je ein Hauptprogramm, das nach einem Bindevorgang ausführbar ist.

  Ein Programm-Modul beginnt in Modula 2 mit dem Schlüsselwort „`MODULE`“
- Bibliotheks-Module: sie stellen Dienste zum Import für andere Programm- oder Bibliotheks-Module zur Verfügung.

Ein Bibliotheks-Modul kann (fast) beliebige Modula-Objekte zum Import durch andere Module „exportieren“. Oft sind es eine oder mehrere Objektklassen (in Modula: Datentypen), dann spricht man von einem **Typen-Modul**. Wird nur ein einziges Datenobjekt exportiert, so nennt man das Modul ein **Daten-Modul**. Ist eine Objektklasse schon anderweitig verfügbar und werden nur zusätzliche Methoden dafür zur Verfügung gestellt, so spricht man von einem **Funktions-Modul**.

Zu einem Bibliotheks-Modul gehören zwei Komponenten, ein **Definitions-Modul** und ein **Implementierungs-Modul**. Das Definitions-Modul ist eine Beschreibungsdatei, in der die **Schnittstelle** beschrieben ist; dort sind alle exportierten Modula-Objekte und Dienste aufgeführt samt allen zu ihrer Verwendung nötigen Informationen, also ihr **Kontrakt**. Die Beschreibungsdatei beginnt mit den Modula-Schlüsselwörtern „`DEFINITION MODULE`“; sie muß durch das Modula-System in eine maschinenlesbare interne Form überführt werden, um dann für „`IMPORT`“ verfügbar zu sein.

Das Implementierungs-Modul enthält die Programmteile, die alle in der Schnittstelle angebotenen Objekte und Dienste realisieren. Es beginnt mit den Schlüsselwörtern „`IMPLEMENTATION MODULE`“ und wird in ausführbaren Code übersetzt, der später mit den Aufrufern zusammen zu einem ausführbaren Programm zusammengebunden wird. Das Implementierungs-Modul bezieht sich implizit auf die Schnittstelle und *kann geändert werden,* ohne daß die es aufrufenden Programmteile davon wissen müssen, solange die Schnittstelle, und also der Kontrakt, gleich bleibt.

25 Rekursion

rekurs

Wenn bei der Lösung eines Problems aus einer Problemklasse als Bestandteil die Lösung einer *kleineren* Instanz derselben Problemklasse mit verwendet wird, so spricht man von **Rekursion**.

Das Prinzip der Rekursion ist äußerst mächtig, wird aber von Anfängern nicht gerne verwendet, weil:

- es oft an ungeeigneten Beispielen erklärt wird, bei denen sein Nutzen nicht einsichtig ist; manche Probleme haben eine einfachere nicht-rekursive Lösung.
- oft auf ungeeignete Weise versucht wird, sich von seiner korrekten Anwendung zu überzeugen; dann kann es recht unübersichtlich werden.

Geht man aber behutsam vor, so ist das Ganze sehr einfach:

- Wir betrachten die verwendete Teillösung als „**black box**“, von der wir annehmen, daß sie ihren Kontrakt einhält. Ihren Aufbau und ihre interne Funktion betrachten wir auf dieser Stufe *nicht!*
- Wir überlegen uns einen Beweis für die **Terminierung**, also das Abbrechen der rekursiven Zerlegung. Dazu genügt es, ein geeignetes ganzzahliges **Größenmaß** zu finden und sicherzustellen, daß alle Teilprobleme *echt kleiner* sind.
- Wir überlegen uns einen Beweis für die **Korrektheit**.

Beim Korrektheitsbeweis verwenden wir den „Satz vom **kleinsten Verbrecher**“:

> Jede *nichtleere* Menge von natürlichen Zahlen hat ein kleinstes Element

in folgender Weise:

Wenn unsere Lösung für *manche* Problemgrößen fehlerhaft arbeiten sollte, dann gibt es darunter eine *kleinste.* Für diese kleinste *falsche* Lösung wissen wir aber, daß alle (rekursiv verwendeten) „black boxes“ korrekt arbeiten, da sie ja kleiner sind; der Fehler kann also nur an ihrem *Zusammenspiel* liegen. Wenn sich das Zusammenspiel der Teilprobleme aber als korrekt herausstellt, so gibt es eine kleinste fehlerhaft arbeitende Instanz *gar nicht,* und somit ist unsere Lösung für alle Problemgrößen korrekt!

Dies ist im Grunde ein „auf den Kopf gestellter“ **Induktionsbeweis** nach der Problemgröße, der oft leichter zu finden ist als ein direkter Induktionsbeweis.

Einen solchen Beweis muß man sich für jede rekursive Problemlösung einmal *überlegen;* er muß aber nicht sehr formal hingeschrieben werden. Es genügt, sich vom

korrekten Zusammenspiel der Teilprobleme zu überzeugen (und darauf zu achten, daß sie echt kleiner sind!), und das gelingt, sofern die Verschachtelung der Teilprobleme nicht zu tief ist, oft durch direktes Hinsehen.

26 Faktorisierung: Problemanalyse fakt

Wir wollen ein Programm entwickeln, das einen Benutzer in die Lage setzt, zu beliebig vielen natürlichen Zahlen jeweils die Zerlegung in Primfaktoren berechnen und anzeigen zu lassen. Da wir aus der Mathematik wissen, daß das Problem grundsätzlich eindeutig lösbar ist, konzentrieren wir uns zuerst auf die **Benutzerführung** und verschieben die Frage der Berechnung auf später.

Das Programm soll sich mit einem Begrüßungstext melden und dann wiederholt Zahlen vom Benutzer abrufen, deren Faktorzerlegung zu berechnen ist. Durch Eingabe einer Null als Abbruchkriterium soll der Benutzer anzeigen können, daß er keine weiteren Berechnungen wünscht; dann soll das Programm sich geordnet abmelden. Dieses Verhalten können wir jetzt bereits ausprogrammieren, wenn wir die eigentliche Berechnung an einen anderen Programmbaustein delegieren.

```
MODULE Faktorisiere;
  (* Program zur Faktorisierung natuerlicher Zahlen *)
  (* Vorlesung EI1 WS 1999/2000 *)
  (* Klaus Lagally, 06.11.1999  *)

FROM InOut IMPORT WriteLn, WriteString, ReadCard;
FROM Zahlen IMPORT Zerlege;

VAR Zahl: CARDINAL; (* zu faktorisieren *)

PROCEDURE Prompt;
  (* fordert die naechste Eingabe vom Benutzer an  *)
  (* und legt sie in die CARDINAL-Variable "Zahl" *)
BEGIN WriteString("gib eine natuerliche Zahl oder 0 ein");
   WriteLn;
   ReadCard(Zahl);
END Prompt;

BEGIN (* hier beginnt der Ablauf des Hauptprogramms *)
   WriteString("Faktorisieren von natuerlichen Zahlen");
   WriteLn;
   WriteString("Eingabe von Null bewirkt: Programmende!");
   WriteLn;
   Prompt; (* fordert naechste Eingabe nach "Zahl" an *)

   WHILE Zahl <> 0
   DO Zerlege(Zahl); (* berechne und drucke die Zerlegung *)
```

```
      Prompt; (* naechste Eingabe nach "Zahl" anfordern *)
    END;

    WriteString("Auf Wiedersehen!"); WriteLn;
END Faktorisiere.
```

Die eigentliche Berechnung haben wir in ein noch zu definierendes Bibliotheks-Modul „Zahlen“ (S. 32) abgeschoben. Das Programm selbst verwendet die Tatsache der Faktorisierung gar nicht und kann somit, mit den nötigen Anpassungen, für andere wiederholte Berechnungen wiederverwendet werden.

27 Faktorisierung: Definitionsmodul

zahlen1

Wir wollen ein Bibliotheksmodul (S. 28) `Zahlen` entwickeln, das eine Prozedur `Zerlege` zur Berechnung und Ausgabe der Primfaktorzerlegung einer natürlichen Zahl exportiert. Dazu legen wir seine Schnittstelle in einem Definitions-Modul (S. 28) fest.

```
DEFINITION MODULE Zahlen;
  (* Modul zur Faktorisierung natuerlicher Zahlen *)
  (* Vorlesung EI1 WS 1999/2000 *)
  (* Klaus Lagally, 09.11.1999  *)

PROCEDURE Zerlege(n: CARDINAL);
  (* bestimme die Primfaktorzerlegung der Zahl n *)
  (* und gib sie in der Form  n=f1*f2*...*fk aus *)

END Zahlen.
```

Dies ist die gesamte Information, die einem Anwender der Prozedur `Zerlege` bekannt sein muß. Die interne Realisierung ist ihm nicht zugänglich, und sie soll ihn auch nicht interessieren; wir behalten uns vor, sie abzuändern (und dabei hoffentlich zu verbessern), ohne daß er dies wissen oder gar sein Programm neu übersetzen muß; ein neuer Bindevorgang wird dann allerdings nötig sein, da sich ja der ausführbare Code geändert hat.

Bevor das Definitions-Modul verwendet werden kann, muß es noch in eine geeignete interne Darstellung übersetzt werden.

Die Realisierung für das Bibliotheks-Modul findet sich in einem gleichbenannten Implementierungs-Modul (S. 33).

28 Faktorisierung: Implementierungsmodul (1) zahlen2

Für das Implementierungs-Modul müssen wir uns nun ein Verfahren überlegen. Der einfachste Ansatz ist wohl, aufsteigend ab 2 alle möglichen Teiler zu testen und jeweils abzudividieren, solange das geht; dabei geben wir sie auch gleich aus. Eigentlich brauchen wir nur Primzahlen zu testen, die haben wir aber hier nicht zur Verfügung. So testen wir auch auf zerlegbare Zahlen als mögliche Teiler, werden aber keine finden, weil deren Primfaktoren kleiner sind und also vorher schon entfernt wurden. Unser Verfahren bricht ab, wenn nur noch ein letzter Teiler übrig ist.

Das Ganze wird durch das folgende Modul geleistet:

```
IMPLEMENTATION MODULE Zahlen;
  (* Modul zur Faktorisierung natuerlicher Zahlen *)
  (* Vorlesung EI1 WS 1999/2000 *)
  (* Klaus Lagally, 09.11.1999  *)

FROM InOut IMPORT WriteCard, WriteString, WriteLn;

PROCEDURE Zerlege(n: CARDINAL);
  (* bestimme die Primfaktorzerlegung der Zahl n    *)
  (* und gib sie in der Form  n = f1*f2*...*fk aus  *)
  (* suche und drucke aufsteigend alle Teiler von n *)
VAR Teiler: CARDINAL;
BEGIN WriteCard(n,1); WriteString(* = *);
      Teiler := 2;
   WHILE Teiler < n
   DO WHILE (n MOD Teiler) = 0
      DO WriteCard(Teiler,1); WriteString(" * ");
         n := n DIV Teiler
      END;
      Teiler := Teiler + 1
   END;
   WriteCard(n,1); WriteLn
END Zerlege;

END Zahlen.
```

Wir haben hier das Verfahren ausprogrammiert, das wir bei der Arbeit mit Papier und Bleistift (oder Taschenrechner) ohne Computer wohl auch verwendet hätten; solch ein **direkter Angriff**, also die Übertragung von alltäglichen Verfahren auf den Computer, führt oft schnell zu guten Ansätzen. Um sich von der korrekten

Funktion zu überzeugen, überlegt man sich am besten, was das Verfahren tut, wenn n die Werte 0, 1, 2 hat oder eine Primzahlpotenz oder eine allgemeine zerlegbare Zahl ist; trotz der beiden geschachtelten Schleifen ist das beim **Schreibtischtest** noch zu überblicken (von der Verwendung eines *„Debuggers“* halten wir nicht viel). Überraschenderweise gibt es auch eine äußerst einfache rekursive Lösung (S. 34).

29 Faktorisierung: Implementierungsmodul (2) zahlen3

Das Problem der Faktorzerlegung läßt sich überraschenderweise unter Verwendung von Rekursion (S. 29) sehr einfach und übersichtlich lösen, wenn man die Problemklasse erweitert (darauf muß man allerdings erst einmal kommen).

Wir nutzen aus, daß wir die Faktoren in aufsteigender Reihenfolge bestimmen; dann wissen wir also zwischendurch, daß kleinere Teiler als der gerade getestete nicht vorkommen können. Wir definieren dazu eine Routine „Zerlege2(n,t)“, welche die Zerlegung von n in Faktoren liefert, wobei n keine Teiler kleiner als t hat. Diese Routine verwenden wir nur intern und nehmen sie nicht in die Schnittstelle auf (das wäre wohl möglich, aber vermutlich braucht sie sonst niemand).

Das Ganze wird durch das folgende Modul geleistet:

```
IMPLEMENTATION MODULE Zahlen;
  (* Modul zur Faktorisierung natuerlicher Zahlen *)
  (* zweite, rekursive Version  *)
  (* Vorlesung EI1 WS 1999/2000 *)
  (* Klaus Lagally, 09.11.1999  *)

FROM InOut IMPORT WriteCard, WriteString, WriteLn;

PROCEDURE Zerlege(n: CARDINAL);
  (* bestimme die Primfaktorzerlegung der Zahl n    *)
  (* und gib sie in der Form  n = f1*f2*...*fk aus  *)
  (* suche und drucke aufsteigend alle Teiler von n *)
BEGIN WriteCard(n,1); WriteString(* = *);
  Zerlege2(n, 2) (* noch sind alle Teiler > 1 moeglich *)
END Zerlege;

PROCEDURE Zerlege2(n, t: CARDINAL);
  (* liefert Zerlegung von n, wenn kein Teiler < t existiert *)
  (* Vorbedingung: t > 1 *)
BEGIN IF t >= n
```

```
    THEN (* n kann nicht weiter zerlegt werden *)
      WriteCard(n,1); WriteLn
    ELSIF (n MOD t) = 0
    THEN (* Teiler t gefunden *)
      WriteCard(t,1); WriteString(" * ");
      Zerlege2(n DIV t, t) (* liefert die restliche Zerlegung *)
    ELSE (* t ist nicht Teiler *)
      Zerlege2(n, t+1) (* zerlege: kein Teiler < t+1 existiert *)
    END
  END Zerlege2;

  END Zahlen.
```

Daß die Teilprobleme richtig zusammenspielen, „sieht man“; wie sieht es aber mit der Terminierung (S. 29) aus, und was ist ein sinnvolles Größenmaß für die Instanz(n,t) (S. 16) in der Problemklasse von „Zerlege2“?

Es zeigt sich, daß „n-t“, die Differenz der Argumente, das Gewünschte leistet: es nimmt bei den beiden rekursiven Anwendungen von „Zerlege2“ jedesmal echt ab, und wenn es $<= 0$ ist, lösen wir das Problem direkt (und richtig). Unsere Lösung ist also korrekt.

30 Typtransfer-Funktionen

transf

Beim Aufbau von Ein- oder Ausgaberoutinen steht man vor dem Problem, beliebige druckbare Zeichen anhand ihrer Position in der ASCII (S. 25)-Tabelle auszuwählen, oder auch umgekehrt zu einem vorgelegten Zeichen seine Position in der Tabelle zu bestimmen; dieses Problem tritt bereits bei den Ziffern auf!

Dafür bietet Modula zwei **Umwandlungsfunktionen** standardmäßig an:

- CHR(n) ist das Zeichen (vom Typ CHAR), das an der Stelle n in der Tabelle steht;
- ORD(c) ist die Platznummer (vom Typ CARDINAL) des Zeichens c.
  (ORD gibt es auch für andere „Aufzählungstypen (S. 24)“, die wir noch kennenlernen werden).

Ein gutes Beispiel (S. 37) für die Verwendung dieser **Transfer-Funktionen** und auch die Anwendung der Rekursion (S. 29) ist die Ausgabe von CARDINAL-Werten zur Basis 16 (**hexadezimal**).

Auch für andere Datentypen (S. 22) gibt es Umwandlungsfunktionen: so ist in der Mathematik die Menge **N** eine Teilmenge von **Z**, und **Z** ist Teilmenge sowohl von **Q** wie von **R**. Für die entsprechenden Datentypen CARDINAL, INTEGER, REAL in Modula gilt dies aber keineswegs, weil die interne Darstellung der Werte verschieden ist.

- FLOAT(n) ergibt einen REAL-Wert, wenn n INTEGER oder CARDINAL ist,
- TRUNC(r) ergibt den ganzzahligen INTEGER-Anteil des REAL-Wertes r,
- INTEGER(n) wandelt n von CARDINAL nach INTEGER,
- CARDINAL(n) wandelt n von INTEGER nach CARDINAL.

Daß die beiden letzten Funktionen ebenso heißen wie die Ergebnis-Datentypen, ist merkwürdig, aber praktisch, da leicht zu merken. Natürlich muß sich das Ergebnis im neuen Datentyp intern darstellen lassen, sonst kann beliebig Fürchterliches passieren.


Auch in anderen Programmiersprachen gibt es solche Umwandlungsfunktionen, die allerdings meist etwas anders heißen und auch etwas anders arbeiten; manchmal wird auch automatisch bei Bedarf umgewandelt. Hier ist man von einer Standardisierung leider weit entfernt. Auch die oben genannten Funktionen sind in Modula nicht wirklich genormt, sondern implementierungsabhängig.

31 HEX-Ausgabe

hexout

Gesucht ist eine Ausgaberoutine, die es gestattet, einen gegebenen Wert n vom Typ CARDINAL in der **Stellenwertdarstellung** zur Basis 16 (hexadezimal) in ein Ausgabefeld der minimalen Breite k auszugeben; als Ziffern für 10 - 15 sollen die Großbuchstaben 'A' - 'F' verwendet werden.

Zuerst bauen wir eine Hilfsroutine auf, die eine einzelne HEX-Ziffer mittels der Standard-Ausgaberoutine Write(c) ausgibt; dabei nutzen wir aus, daß im ASCII-Code (S. 25) (und in allen anderen uns bekannten Codierungen) die Ziffern '0' .. '9' und die Buchstaben 'A' ..'F' jeweils unmittelbar hintereinander liegen. Zur Umwandlung zwischen den Datentypen CARDINAL und CHAR verwenden wir die Transfer-Funktionen (S. 36) ORD und CHR.

```
FROM InOut IMPORT Write;

PROCEDURE WriteHexDigit(n: CARDINAL);
(* PRE n < 16 *)
BEGIN IF n < 10
      THEN Write(CHR(ORD('0') + n))
      ELSE Write(CHR(ORD('A') + n - 10))
      END
END WriteHexDigit;
```

Weiterhin nutzen wir die bekannte Tatsache aus, daß bei der Stellenwertdarstellung einer Zahl n zu einer Basis B die letzte Ziffer gerade den Rest (n MOD B) codiert, während die Ziffern davor die Stellenwertdarstellung des Quotienten (n DIV B) darstellen, falls dieser Quotient nicht 0 ist.

```
PROCEDURE WriteHex(n, k: CARDINAL);
VAR i: CARDINAL;
BEGIN IF n < 16 (* nur eine Stelle; vorne auffuellen *)
      THEN FOR i := 1 TO k-1 DO Write(' ')
           END
      ELSE WriteHex(n DIV 16, k-1) (* vordere Stellen *)
      END;
      WriteHexDigit(n MOD 16)
END WriteHex;
```

Um führende Zwischenräume auszugeben, haben wir hier eine Zählschleife (S. 48) verwendet. Die Prozedur sollte, obgleich sie rekursiv (S. 29) ist, bei näherer Betrachtung selbsterklärend sein.

32 Funktionen

funkt

Unter einer (totalen) **Funktion** f von einer Menge M in eine Menge N verstehen wir eine *bestehende* **Zuordnung**, die für *jedes* Element m von M *eindeutig* ein Element n von N vorgibt, das wir mit f(m) bezeichnen. Wir schreiben dafür

$$f: M \to N \qquad \text{oder auch} \qquad m \mapsto n = f(m)$$

Die Zuordnung f muß *nicht* durch eine **Rechenvorschrift** gegeben sein; ist M endlich, so ist die einfachste Beschreibung von f eine **Wertetabelle**! M und N können ganz beliebig sein (nicht nur Zahlenmengen); wir unterscheiden auch nicht zwischen Funktionen und **Abbildungen**.

M heißt **Definitionsbereich** von f, N heißt **Wertevorrat**. Anders als vom Gymnasium her gewohnt, verlangen wir, daß f auf *ganz M* definiert ist; dagegen müssen nicht alle Elemente von N als Werte vorkommen. Die Teilmenge der Elemente von N, die als Funktionswerte tatsächlich angenommen werden, heißt **Bildbereich** von f, geschrieben **Img(f)**. Ist Img(f) = N, so heißt die Funktion f **surjektiv**.

Oft verwendet man auch **partielle Funktionen**, die nicht auf ganz M definiert sind; dann schreiben wir

$$f: M \nrightarrow N$$

und bezeichnen die Teilmenge von M, auf der f definiert ist, mit **Def(f)**; wir nennen sie wieder *„Definitionsbereich“*. Die im Gymnasium häufig betrachteten *reellen Funktionen* sind partielle Funktionen von **R** nach **R** in unserem Sinne. Eine *partielle* Funktion ist *totale* Funktion auf ihrem Definitionsbereich!

Die Funktionen von M nach N bilden selbst eine Menge (in Modula 2: einen Datentyp); wir bezeichnen diese mit [M→N], lassen aber gelegentlich die Klammern weg. Wollen wir andeuten, daß wir die partiellen Funktionen meinen, so schreiben wir [M ↛N].

Ist eine Funktion f durch einen **Rechenterm** gegeben, der von einer Veränderlichen x abhängt, so berechnet sich das Ergebnis der **Anwendung** von f auf ein **Argument** a durch konsistentes Ersetzen von x durch a und Auswerten des entstehenden neuen Terms. Hängt der Term von *mehreren* Veränderlichen ab, so müssen wir kenntlich machen, *welche* der Veränderlichen bei der Anwendung zu ersetzen ist; dies leistet die Lambda-Notation (S. 39), der in Programmiersprachen die *Parameter-Angabe* entspricht.


Unser hier eingeführter Funktionsbegriff ist so allgemein, daß er auch solche Fälle umfaßt, die in manchen mathematischen Darstellungen als *Funktionale* oder **höhere Funktionen** (S. 40) bezeichnet werden.

33 Lambda-Notation; Funktionsprozeduren

lambda

Wollen wir eine Funktion (S. 38) durch einen Rechenterm definieren, der von mehreren Veränderlichen abhängt, so müssen wir kenntlich machen, welche der Veränderlichen bei der Anwendung durch das Argument zu ersetzen ist; die anderen Veränderlichen bleiben **frei**.

Beispiel: gegeben sei der Term x*x + y*y. Damit können wir zwei *verschiedene* Funktionen bilden:

$$x \mapsto f1(x) = x*x + y*y, \qquad f1(a) = a*a + y*y$$

aber auch

$$y \mapsto f2(y) = x*x + y*y, \qquad f2(a) = x*x + a*a$$

Dafür kann man auch schreiben:

$$f1 = \lambda x\ .\ (x*x + y*y), \qquad f2 = \lambda y\ .\ (x*x + y*y)$$

Der durch eine solche **Lambda-Angabe** *gebundene* (S. 55) Bezeichner für den **Parameter** kann innerhalb seines Gültigkeitsbereichs (S. 11) (hier: die Funktionsdefinition) überall durch einen anderen Bezeichner ersetzt werden, solange dadurch keine Kollisionen entstehen; dadurch ändert sich die Bedeutung nicht.

In Programmiersprachen werden Funktionen durch **Funktionsprozedur**en vertreten, den Lambda-Angaben entspricht die *Parameterliste.* In Modula 2 sind bei den Parametern und beim Ergebnis jeweils noch die *Datentypen* anzugeben; damit sieht eine Funktionsvereinbarung beispielsweise für die Funktion f2 folgendermaßen aus:

```
PROCEDURE f2(y: REAL): REAL;
BEGIN RETURN x*x + y*y
END f2;
```

Sogar die Anwendung von Funktionen, die keinen Namen tragen, sondern die *nur* durch einen Term gegeben sind, auf ein Argument läßt sich so anschreiben (zur Klarheit setzen wir Klammern):

$$(\lambda x\ .\ (x*x + y*y))(a) = a*a + y*y$$

Das ist dann interessant, wenn sich der Term selbst erst durch eine Berechnung ergibt, etwa die Bildung der Ableitung beim symbolischen Rechnen.

Es müssen übrigens nicht nur *Veränderliche* sein, die durch eine Lambda-Angabe markiert und bei der Funktionsanwendung ersetzt werden; läßt man hier auch *Funktionsbezeichner* zu, so kommt man zu den sogenannten **höheren** (S. 40) **Funktion**en; manche davon sind uns bereits wohlbekannt!

34 Höhere Funktionen

hfunkt

Unser Funktionsbegriff (S. 38) ist so allgemein, daß er auch **Funktionale** und **höhere Funktionen** umfaßt; darunter versteht man gelegentlich Operationen, bei denen in einem Term nicht nur Variablen, sondern auch Funktionssymbole durch Argumente (geeigneten Typs) ersetzt werden.

Solche Operationen sind schon vom Gymnasium her bekannt, werden aber dort meist nicht als Funktionen aufgefaßt.
Beispiel: die **Differentiation**

D: $[\mathbf{R} \to \mathbf{R}] \to [\mathbf{R} \to \mathbf{R}]$

überführt eine reelle (differenzierbare) *Funktion* f in ihre *Ableitungsfunktion* D(f) = f'. Das kann geschrieben werden:

$D = \lambda f\ .\ f'$ oder deutlicher $D = \lambda f\ .\ (\lambda x\ .\ f'(x))$

Ein **bestimmtes Integral** auf einem *festen* Intervall (a,b) überführt eine *reelle Funktion* f in einen reellen *Wert*, ist also vom Typ $[\mathbf{R} \to \mathbf{R}] \to \mathbf{R}$ und kann geschrieben werden:

$$I = \lambda f. \int_a^b f(t)\,dt$$

Das Bilden einer **Stammfunktion** macht dagegen aus einer *reellen Funktion* wieder eine *reelle Funktion*, ist also vom gleichen Typ $[\mathbf{R} \to \mathbf{R}] \to [\mathbf{R} \to \mathbf{R}]$ wie die Differentiation. Wir bekommen die Stammfunktion, indem wir das bestimmte Integral von der oberen Grenze abhängen lassen, das geht mit der Lambda-Notation (und *nur* mit ihr) sehr bequem:

$$S = \lambda f.(\lambda b. \int_a^b f(t)\,dt)$$

Die Operation: **Auswertung** einer beliebigen *reellen Funktion* f an einer gegebenen *festen* reellen Stelle a ist vom gleichen Typ $[\mathbf{R} \to \mathbf{R}] \to \mathbf{R}$ wie das Bilden des bestimmten Integrals und kann geschrieben werden:

$A = \lambda f\ .\ f(a)$ im Gegensatz zur Funktion f selbst: $f = \lambda x\ .\ f(x)$

Modula 2 ist (neben LISP) eine der wenigen Programmiersprachen, in denen sich höhere Funktionen bequem programmieren lassen, da es hier auch Prozedur-Typen (S. 115) gibt.

35 Imperative Programmierung

imper

Im Modell der **imperativen Programmierung** besteht der **Ablauf** eines Programms aus einer *zeitlichen Folge* von **Aktionen**; jede Aktion ergibt sich durch Ausführung einer **Anweisung** der zu Grunde liegenden Programmiersprache.

Eine Aktion kann in eine *zeitliche Folge* von Teil-Aktionen zerfallen; dem entspricht in der Programmiersprache eine **strukturierte Anweisung**, die angibt, *welche* Teilanweisungen in *welcher Folge* auszuführen sind.

Diese (rekursive) Zerlegung sowohl der Aktionen wie auch der strukturierten Anweisungen kann mehrstufig sein; das ist das Grundprinzip der **strukturierten Programmierung**, wobei man sich auf eine *feste* Menge von Zerlegungsmustern (**Kontrollstrukturen** (S. 43)) für Anweisungen beschränkt.

Eine Konsequenz des strukturierten Vorgehens, und auch der tiefere Grund für seine Nützlichkeit, ist die Tatsache, daß der Aufwand für das Erstellen und das Verständnis eines korrekten Programms dann nicht schneller anwächst als die Programmlänge; nur auf diese Weise lassen sich mit Aussicht auf Erfolg größere Programmsysteme entwickeln. Dabei kommt es auf die Art der zugelassenen Zerlegungsmuster nicht entscheidend an, solange es sich dabei immer um Anweisungen bzw. Aktionen handelt.

Die rekursive Zerlegung endet bei Anweisungen, die nicht weiter zerlegt werden *können* oder *sollen.* Dies sind (in Modula 2 und in ähnlichen Sprachen):

- die „**leere Anweisung**“. Sie tut nichts und wird auch meist nicht eigens hingeschrieben. Ihr einziger Zweck ist, triviale Sonderfälle und Schreibfehler abzufangen.
- die (Wert)-**Zuweisung** der Gestalt <V> := <E>. Dabei ist <V> der Name einer **Variablen** (S. 56), und <E> ist ein Ausdruck, der ausgewertet wird; das Ergebnis wird der *neue* Inhalt der Variablen <V>.

  In vielen Programmiersprachen (beispielsweise in C und FORTRAN) wird statt „ := “ für den **Zuweisungsoperator** „ = “ geschrieben; das bietet Gelegenheit zur Verwechslung mit dem **Vergleichsoperator**, der dann anders geschrieben werden muß („ == “ bzw. „ .EQ. “)! Beides hat übrigens mit dem **Gleichheitszeichen** in der Mathematik, das die Tatsache einer *bestehenden* Gleichheit ausdrückt, nichts zu tun!
- ein **Prozeduraufruf** (S. 49) der Gestalt <Prozedur-Name>(<Argument-Liste>), dabei ist die <Argument-Liste> eine durch Kommas getrennte Liste von **Argumenten**. Ein Prozeduraufruf bedeutet den Aufruf eines **Bausteins**, der ein Teilproblem aus einer *Klasse* von Problemen löst. Die Argumentliste gibt an, um welche Instanz der Klasse es sich handelt.

Die Argumente sind meist **Ausdrücke**, deren **Wert** das betreffende Teilproblem auswählt, können aber auch **Name**n (Referenzen) von lokal sichtbaren Objekten sein, auf die der Baustein Zugriff haben soll. Das Nähere regelt der Kontrakt (in Modula 2: mittels einer **Prozedur-Vereinbarung** (S. 50)).

Die Wirkung des Bausteins ist gegeben durch einen **Kontrakt** (oft außerhalb der Programmiersprache), also eine Verabredung, unter welchen vom Aufrufer zu schaffenden **Voraussetzungen** der Baustein welche **Leistung** zu erbringen hat. Ansonsten betrachten wir ihn hier als „schwarzen Kasten“, und wir untersuchen seine interne Funktion hier nicht weiter; dies ist Sache seines Anbieters (**Geheimnisprinzip**).

Eine Variante des Prozeduraufrufs ist der **Makro-Aufruf**. Er kann äußerlich genauso aussehen (meist ist das nicht so!), wird aber technisch anders behandelt: in einem Vorlauf der Übersetzung wird die **Realisierung** des „schwarzen Kastens“ an der Aufrufstelle selbst (mit den nötigen Änderungen) **textuell** *eingesetzt.* Modula 2 bietet dies nicht an, wohl aber C und „Verwandtschaft“.

Es mag auffallen, daß unter den elementaren Anweisungen weder der explizite **Sprung** an eine markierte Anweisung noch das explizite Setzen einer solchen **Marke** vorkommen. Dies ist Absicht; beide Konstrukte lassen sich nicht in natürlicher Weise als Aktionen deuten, denen die Lösung eines Teilproblems entsprechen würde, und ihre unkritische Anwendung kann zu beliebig unübersichtlichen Programmabläufen führen. Entgegen manchen Ansichten in der älteren Literatur bedeutet ihre Vermeidung noch nicht automatisch „strukturierte Programmierung“. Umgekehrt wird ein Schuh daraus: bei strukturiertem Vorgehen braucht man sie nicht (außer direkt auf der Ebene der Maschinenbefehle, um höhere Konstrukte nachzubilden; dann tun sie aber auch keinen Schaden).

36 Kontrollstrukturen

control

Um *strukturierte Anweisungen* (S. 41) aus Teilanweisungen aufzubauen, bieten alle imperativen Programmiersprachen (S. 41) einen Satz von **Kontrollstrukturen** an, die sich durch **Struktogramme** (S. 44) gut veranschaulichen lassen. Der Grundvorrat ist bei allen Sprachen etwa derselbe, bis auf Abweichungen in der Schreibung:

- die **Anweisungsfolge** oder **Anweisungssequenz**:

  <S1>; <S2>

  In Modula 2 (und seiner Verwandtschaft) werden mehrere Anweisungen durch **Strichpunkt** (Semikolon) *verbunden*; das bedeutet ihre *zeitliche Aufeinanderfolge* (in ALGOL 68 bezeichnet die dort auch vorkommende Verbindung durch **Komma** die mögliche **parallele Ausführung**).

  In manchen Sprachen (wie etwa C) *schließt* ein Semikolon dagegen eine Anweisung *ab;* die Glieder einer Sequenz werden dort einfach hintereinander angeschrieben. (Macht man das in Modula 2 versehentlich ebenso, so würde nach dem letzten Glied eine Fehlermeldung auftreten, wenn es die leere Anweisung nicht gäbe).

- die **Alternative** (Auswahl aus 2 Möglichkeiten):

  IF <C> THEN <S1> ELSE <S2> END

  Wenn die Bedingung <C> wahr ist, wird die Anweisung <S1> ausgeführt, andernfalls die Anweisung <S2>. Wenn <S1> oder <S2> leer sind, ergeben sich Vereinfachungen; dafür und auch für die Auswahl aus mehreren Möglichkeiten bietet Modula 2 eigene Konstrukte zur **Fallunterscheidung** (S. 46) an.

- die **Wiederholungsanweisung** (Schleife):

  WHILE <C> DO <S> END

  führt eine Anweisung <S> wiederholt aus, solange jeweils *vorher* die Bedingung <C> erfüllt war; ist dies bereits zu Anfang nicht der Fall, so geschieht gar nichts (abweisende Schleife (S. 47)).

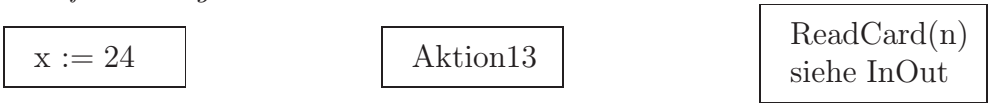
  Diese Art der Wiederholung ist im Prinzip ausreichend, um alle Fälle zu erfassen, und sie ist am einfachsten auf Korrektheit zu prüfen. Dennoch bietet Modula 2 (wie andere Programmiersprachen auch) für häufige Sonderfälle einen Satz von weiteren **Schleifenstrukturen** (S. 47) an.

37 Struktogramme (1)

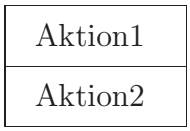
strukt1

Die der **Strukturierten Programmierung** (S. 41) zu Grunde liegenden wichtigsten Kontrollstrukturen (S. 43) lassen sich graphisch gut veranschaulichen; wir weichen allerdings aus gutem Grunde von der genormten (!) Darstellung etwas ab.

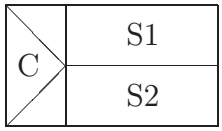
- Eine Aktion (S. 41) wird immer durch ein **Rechteck** dargestellt, das entweder die *Beschreibung* ihrer Funktion (oder einen Verweis darauf) enthält, oder aber nach *festen Regeln* weiter *unterteilt* wird.



- Eine **Sequenz** von Aktionen ergibt eine senkrecht aufeinander passende Folge von Rechtecken, oder gleichbedeutend: ein durch waagrechte Trennungen (denen jeweils ein Semikolon entspricht) in weitere Rechtecke unterteiltes Rechteck. Sie bedeutet die Ausführung der Teilaktionen in der Folge von oben nach unten.

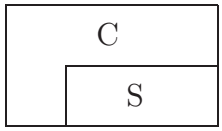


- Eine **Alternative** IF <C> THEN <S1> ELSE <S2> END stellen wir *abweichend vom Standard* wie folgt dar:

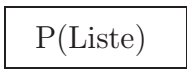


  Standardmäßig steht sonst <C> oben, <S1> links und <S2> rechts; unsere Notation hat den Vorteil, daß das Format von <S1> und <S1> bequemer ist, und daß man über die Zeichnung die Schlüsselwörter aus Modula 2 (und Pascal) direkt an die passenden Stellen schreiben kann.

- Die **Schleife** WHILE <C> DO <S> END ist wohl selbsterklärend:



- Für einen **Prozeduraufruf** „P(Liste)“ bleibt der Kasten



  stehen, und ist allenfalls *in einem neuen Diagramm* weiter zu zerlegen.

38 Struktogramme (2)

strukt2

Beim Arbeiten mit Struktogrammen stellen sich von selbst gute **Arbeitsregeln** heraus:

- Die Zerlegung darf nicht zu tief werden; wenn bei normaler Schriftgröße ein Blatt DIN A4 nicht ausreicht, sollte man einen Baustein oder mehrere davon (nach Festlegung ihres Kontraktes!) herausziehen.
- Eine Zerlegung in mehr als allenfalls 3 Ebenen ist nicht mehr gut zu überschauen, und daher vermutlich erst mal falsch!
- Auch wenn man nicht mehr mit Struktogrammen arbeitet (sie kommen vor allem in Schulbüchern und Einführungsvorlesungen vor), gelten beim Arbeiten am Bildschirm (oder mit Papier und Bleistift, auf einzelnen Blättern) entsprechende Erfahrungsregeln.
- Ein Baustein, der mehr als einen Schirm oder eine Seite füllt, ist zu groß (außer es handelt sich um eine rein schematische Wiederholung einander entsprechender Einträge).
- Was sich nicht mit einem Blick erfassen läßt, sollte man besser untergliedern, und dabei jeweils einen Kontrakt formulieren. Das fördert erheblich das Problemverständnis, und damit die Chancen auf eine einfache und korrekte Lösung.

Für einige hier noch nicht besprochene Varianten (S. 47) der grundlegenden Kontrollanweisungen gibt es auch Vorschläge für entsprechende Struktogramme, die mehr oder weniger gelungen sind. Vieles ist hier Geschmackssache; jedenfalls kommt man mit den oben angegebenen Grundstrukturen im Prinzip allein aus, und in der Praxis zumindest recht weit.

Bemerkung zu Sprüngen

Ein Konzept läßt sich mit Struktogrammen, und auch mit den hier besprochenen Kontrollstrukturen (S. 43), *nicht* ausdrücken, und das ist gerade eine ihrer Stärken: die Vorstellung von einem Prozessor, der den Programmtext durchläuft und dabei gelegentlich an eine andere Stelle springt. Tatsächlich kommt es uns bei der Strukturierten Programmierung (S. 41) gar nicht darauf an, **Sprünge** zu vermeiden, wie man anfangs glaubte; auf der Ebene der Maschinenbefehle sind sie sogar unverzichtbar. Zu vermeiden sind nur Sprünge an *beliebige Stellen,* (die sogenannte „**Spaghetti-Programmierung**“), und unsere Konstrukte erlauben dies nicht, denn sie enthalten das Konzept der „**Stelle**“ überhaupt nicht.

39 Fallunterscheidung

case

Ist eine von mehr als 2 Alternativen auszuwählen, so kann man in Modula 2 die **CASE-Anweisung** verwenden. Die Idee dazu ist recht alt und in mehreren Programmiersprachen früher schon versucht worden; aber erst in Modula 2 wurde eine wirklich übersichtliche, praktisch brauchbare Notation gefunden.

Eine „**CASE-Anweisung**“ hat folgende Gestalt:

CASE <E> OF <Fälle> ELSE <Rest> END

Die Liste der <Fälle> ist wie folgt aufgebaut:

<Fälle> → <Fälle> | <Fall>
<Fälle> → <Fall>

Der <Rest> ist eine Anweisungsfolge; der Teil „ELSE <Rest>“ kann fehlen.

Die einzelnen Fälle sind also durch den senkrechten Strich „ | “ getrennt (das war die entscheidende gute Idee, denn alle früheren Ansätze waren schwer lesbar). Der einzelne <Fall> besteht aus einer Markenliste und einer Anweisungsfolge:

<Fall> → <Markenliste> : <Anweisungsfolge>
<Fall> → <leer>

und die Markenliste ist eine ggf. durch Kommas getrennte Liste von Konstanten oder Konstantenbereichen. Alle hier vorkommenden Konstanten, und auch der Ausdruck <E>, müssen aus demselben skalaren Typ (S. 24) sein. Der leere Fall ist nur dazu da, einen senkrechten Strich hinter der letzten Alternative zu erlauben.

Bei der Ausführung wird der Ausdruck <E> ausgewertet. Wird ein Fall gefunden, zu dem das Ergebnis paßt, so wird die zugehörige Anweisungsfolge ausgeführt, andernfalls der <Rest>. Fehlt der <Rest> und paßt kein Fall, so wird in Modula 2 ein Laufzeitfehler ausgelöst.

Beispiel

```
TYPE Farbe = (rot, gelb, gruen):
VAR ampel: Farbe;

CASE ampel OF
  rot: Halten |
 gelb: KreuzungFrei |
ELSE   Fahren
END;
```

40 Schleifenstrukturen

loops1

Die Schleifenstruktur

WHILE <C> DO <S> END

prüft wiederholt die Bedingung <C> und führt dann jeweils die Anweisungsfolge <S> aus; sobald <C> nicht erfüllt ist, wird die Schleife verlassen. Ist die Bedingung <C> bereits beim ersten Mal nicht erfüllt, so wird <S> *gar nicht* ausgeführt (**abweisende Schleife**).

Modula bietet aus Bequemlichkeitsgründen *weitere Schleifenstrukturen* an, obwohl die beschriebene WHILE-Schleife im Prinzip ausreichend ist; sie ist auch am einfachsten zu verifizieren.

Die Konstruktion

REPEAT <S> UNTIL <C>

ist gleichbedeutend mit

<S> ; WHILE NOT <C> DO <S> END

Die Anweisungsfolge <S> wird also auf jeden Fall mindestens einmal ausgeführt (**annehmende Schleife**).

Darüberhinaus gibt es noch die **Zählschleife** (S. 48) und die **allgemeine Schleife** (S. 48).

Übrigens ist die Folge

WHILE <C> DO <S1> END ; <S2>

in der Auswirkung gleichbedeutend zu einem Aufruf der parameterlosen (rekursiven) Prozedur:

```
PROCEDURE Schleife;
BEGIN IF <C> THEN <S1> ; Schleife
      ELSE <S2>
      END
END Schleife;
```

Das Schleifenkonstrukt ist also prinzipiell überflüssig, sobald man rekursive Prozeduren zur Verfügung hat. Umgekehrt lassen sich nach demselben Schema gebaute „rechts-rekursive“ Prozeduren in Schleifen umwandeln (Entrekursivierung (S. 129)).

41 Zählschleife und allgemeine Schleife

loops2

Wenn die Anzahl der Schleifendurchläufe bereits vorweg bekannt ist, so kann man die **Zählschleife**

FOR <V> := <E1> TO <E2> BY <E3> DO <S> END

mit Vorteil verwenden. Sie bedeutet folgendes:

Die drei Ausdrücke <E1> für den **Startwert**, <E2> für den **Endwert** und <E3> für die **Schrittweite** der **Schleifenvariablen** <V> werden ausgewertet (der Teil „BY <E3>“ fehlt oft, dann wird E3 = 1 angenommen). Die Variable <V> wird auf den Startwert gesetzt und dann jeweils mit der Schrittweite fortgeschaltet, solange sie nicht jenseits des Endwertes angekommen ist. Dabei wird jedesmal der **Rumpf** <S> der Schleife ausgeführt; der Wert der Variablen <V> kann dort verwendet werden.

Warnung: liegt der Startwert (in der durch die Schrittweite gegebenen Richtung) bereits jenseits des Endwertes, so wird der Rumpf <S> gar nicht ausgeführt.

Warnung: den Wert von <V> im Rumpf <S> der Schleife **explizit** zu verändern, ist in Modula 2 möglich, aber *weniger empfehlenswert*! Auch die Weiterverwendung des Endwertes von <V> nach Verlassen der Schleife ist schlechter Programmierstil.

Bemerkung: die Ausdrücke <E1> und <E2> und die Schleifenvariable <V> können aus einem beliebigen skalaren Typ (S. 24) sein; <E3> ist INTEGER und gibt an, um wie viele Schritte in der Folge der Werte des Typs jeweils weiter bzw. zurück gegangen wird.

Die **allgemeine Schleife**

LOOP <S> END

wird beliebig oft durchlaufen; verlassen wird sie nur durch Ausführung einer irgendwo in der Anweisungsfolge <S> enthaltenen Anweisung **EXIT**. Ihre Verifikation ist in der Regel nicht einfach.


Bemerkung: Die in den Sprachen C, C++ und Java verwendete Schleifenkonstruktion

for (<S1> ; <C> ; <S2>) <S3>

ist gleichbedeutend mit dem Modula 2-Konstrukt

<S1> ; WHILE <C> DO <S3> ; <S2> END

führt aber auf Grund der extrem knappen Notation öfters zu Fehlern.

42 Prozeduraufruf

call

Der Aufruf einer **Prozedur** (S. 50) ist eine Anweisung (S. 41) der Gestalt

<Prozedurname> <Argumentliste>

die sich durch die folgenden Regeln beschreiben läßt:

<Prozedurname> → <Bezeichner>
<Argumentliste> → (<Argumente>)
<Argumentliste> → <leer>

Die Argumentliste kann also fehlen.

<Argumente> → <Argumente> , <Argument>
<Argumente> → <Argument>

Die Argumente werden also durch Kommas getrennt.

<Argument> → <Ausdruck>

Dies ist ein **Wert-Argument**; der Ausdruck wird ausgewertet, und das Ergebnis wird der Prozedur übergeben.

<Argument> → <Variable>

Dies ist ein **Referenz-Argument**; die Variable (S. 56) bezeichnet ein Objekt, das die Prozedur ansprechen und auch verändern können soll.

Wert-Argumente und Referenz-Argumente sind in Modula 2 an der **Aufrufstelle** der Prozedur oft nicht zu unterscheiden; der Unterschied zeigt sich in der Form der Prozedur-Vereinbarung (S. 50), und natürlich gehört er zum Kontrakt.

Ein **Funktionsaufruf** sieht (fast) genauso wie ein **Prozeduraufruf** aus, doch steht er an einer anderen Stelle: nicht als *Anweisung,* sondern als *Ausdruck,* oder als *Operand* eines Ausdrucks, da er ja einen *Ergebniswert* zurückliefert. Hier *sollten* nur Wert-Argumente verwendet werden, und bei (selten vorkommenden) parameterlosen Funktionen muß in Modula 2 merkwürdigerweise dennoch eine leere Parameter-Klammer geschrieben werden. (Die in der Sprache C gegebene Möglichkeit, einen Funktionsaufruf auch als Anweisung zu verwenden und den Ergebniswert „wegzuwerfen“, ist logisch unsauber und fehleranfällig).


Andere Programmiersprachen bieten gelegentlich noch weitere „Parameter-Übergabemechanismen“ wie „call by value/result“ oder „call by name“ an; wir gehen darauf hier nicht ein.

43 Prozedur-Vereinbarung

proz

Die **Vereinbarung** einer Prozedur (S. 49) (oder Funktionsprozedur (S. 51)) besteht aus einem **Prozedurkopf** und einem **Prozedurrumpf**.

Im Kopf steht die Beschreibung der **Parameter**, also der beim Aufruf durch Argumente zu ersetzenden Bezeichner mit Angaben über ihre Verwendung, und als wichtiger (dennoch leider oft weggelassener!) Bestandteil die *Beschreibung des* **Kontrakt**es (S. 41), also der zu erbringenden Leistung unter Angabe der Voraussetzungen. Diese gesamte Information gehört auch, sofern die Prozedur durch ein Modul exportiert wird, in das zugehörige Definitionsmodul (S. 28); sie enthält alles, was der Aufrufer über die Prozedur wissen muß, und nicht mehr.

In Modula 2 hat ein Prozedurkopf folgendes Format:

PROCEDURE <Prozedurname> <Parameterangabe> ;

Dazu kommt als Kommentar die Leistungsbeschreibung.

Die <Parameterangabe> ist nach folgenden Regeln gebaut:

<Parameterangabe> → (<Parameterliste>)
<Parameterangabe> → <leer>

Die Prozedur braucht also keine Parameter zu haben; dann besteht ihr Effekt nur aus „**Nebenwirkungen**“ oder „Seiteneffekten“.

Die Parameterliste ist falls nötig durch Strichpunkte getrennt:

<Parameterliste> → <Parameterliste> ; <Parameter>
<Parameterliste> → <Parameter>

Für **formale Parameter** gibt es (in Modula 2 und verwandten Sprachen) zwei Formate:

<Parameter> → <Bezeichner> : <Typ>

Dies ist ein „**Wert-Parameter**“; er kann wie eine *lokale* **Hilfsvariable** verwendet werden, deren Inhalt anfangs der beim aktuellen Aufruf übergebene **Wert** (S. 41) des Arguments ist.

<Parameter> → VAR <Bezeichner> : <Typ>

Dies ist ein „**Referenz-Parameter**“; er ist ein *zusätzlicher* **Name**, über den das beim aktuellen Aufruf angegebene externe Objekt angesprochen und auch verändert werden kann.

Die Schreibweise kann oft vereinfacht werden: mehrere aufeinanderfolgende Parameterangaben gleicher Art und gleichen Typs kann man zu einer durch Kommas getrennten Liste zusammenfassen.

Der Rumpf ist ein **Block** (S. 53), also eine Anweisungsfolge zusammen mit lokalen Objekten, für deren Bezeichner er den **Gültigkeitsbereich** darstellt. Auch die ganze Vereinbarung ist ein Block, der den Rumpf umfaßt; hier kommen als neue gebundene Bezeichner die Parameter hinzu.

Alle Bezeichner im Rumpf, die weder durch eine lokale Vereinbarung noch eine Parameter-Angabe gebunden sind, sind **frei**; über sie kann auf **globale Objekte** zugegriffen werden, sofern diese in einem umfassenden Block vereinbart sind. Solche **Seiteneffekte** umgehen die Schnittstelle und sind daher riskant, aber gelegentlich unvermeidlich (etwa bei Ausgabeoperationen).

44 Funktionsvereinbarung

proz2

Der Kopf einer **Funktionsvereinbarung** sieht in Modula fast genauso aus wie eine Prozedurvereinbarung (S. 50):

PROCEDURE <Prozedurname> <Parameterangabe> : <Typ>;

das **Funktionsresultat** wird aus dem Rumpf, wo immer es berechnet wurde, durch eine Anweisung RETURN <Ausdruck> an den Aufrufer übergeben.

Es ist gute Praxis, bei Funktionsvereinbarungen *nur* Wertparameter zu verwenden und auf *Seiteneffekte* gänzlich zu verzichten!

Beispiel

Gesucht ist eine Funktion HexDigit, die zu einer gegebenen Zahl n mit 0 <= n < 16 die zugehörige HEX-Ziffer liefert; als Ziffern für 10 - 15 sollen die Großbuchstaben 'A' - 'F' verwendet werden. Dabei nutzen wir aus, daß im ASCII-Code (S. 25) (und in allen anderen uns bekannten Codierungen) die Ziffern '0' .. '9' und die Buchstaben 'A' .. 'F' jeweils unmittelbar hintereinander liegen. Zur Umwandlung zwischen den Datentypen CARDINAL und CHAR verwenden wir die Transfer-Funktionen (S. 36) ORD und CHR.

```
PROCEDURE HexDigit(n: CARDINAL): CHAR;
(* PRE n < 16 *)
BEGIN IF n < 10
      THEN RETURN CHR(ORD('0') + n)
      ELSE RETURN CHR(ORD('A') + n - 10)
      END
END HexDigit;
```

45 Speicherkonzepte

store

Der **Speicher** unseres Rechners besteht aus einzelnen **Zellen** fester Länge, die ab 0 bis zu einem vom Ausbau abhängigen Maximalwert durchnumeriert sind; die Nummern nennen wir **Hausnummern**, um sie von den im Adreßteil von Maschinenbefehlen vorkommenden **Adressen** bei Bedarf unterscheiden zu können; je nach Prozessor kann sich eine Adresse auch auf ein Register beziehen, oder die Adressen werden beim Speicherzugriff noch durch eine eigene Hardware-Einheit umgerechnet.

Wir fassen Folgen von Zellen, die wir zur Ablage von **Daten** verwenden wollen, logisch zu Speicherobjekten zusammen. Ein **Speicherobjekt** hat folgende Attribute:

- einen **Typ**, der die interne Darstellung der Daten und die Länge bestimmt,
- einen **Inhalt**: die Daten selbst in interner Darstellung,
- einen **Ort**: die Hausnummer,

Als Methoden haben wir

- **lesen**: auf den Inhalt zugreifen,
- **schreiben**: den Inhalt ändern.

Wie alle Objekte kann man auch Speicherobjekte **erzeugen** (belegen) und **löschen** (freigeben). Ein Speicherobjekt kann, wenn die Daten eine **Struktur** tragen, in weitere Speicherobjekte **zerfallen**.

Auf der Ebene der höheren Programmiersprachen entsprechen den Speicherobjekten **Variablen** (S. 56), doch ist die Zuordnung nicht umkehrbar eindeutig. Zu jedem *Datentyp* T, also einer Klasse von Werten zusammen mit den darauf definierten Operationen, betrachten wir eine Klasse **Var(T)** der Behälter für Werte aus T. Ein **Behälter** hat die folgenden Attribute:

- den Typ T
- einen **Zugriffspfad**, die **Referenz**, auf ein zugeordnetes Speicherobjekt
- dessen Inhalt (vom Typ T)
- evtl. eine **Benennung** oder mehrere davon

Als Methoden kommen in Betracht:

- **auswerten**: Ausliefern des Inhalts
- **besetzen**: Ändern des Inhalts

Wann (und ob) wir einer Variablen ein Speicherobjekt **zuordnen** (daran binden (S. 55)), hängt von der Lebensdauer (S. 57) der Variablen ab. Die Zuordnung kann man auch wieder **auflösen**.

46 Blöcke (1)

block1

Ein Grundkonzept bei den **blockorientierten** Sprachen (das sind die Familien von Algol, Pascal, C und einige weitere; auch die meisten objektorientierten Sprachen gehören dazu) ist der **Block**.

- Unter einem **Block** verstehen wir einen abgeschlossenen, zusammenhängenden Teilbereich im **Programmtext**. Blöcke können ineinandergeschachtelt sein, aber sie überlappen sich nicht.
- Ein Block bestimmt den **Gültigkeitsbereich** der **Bezeichner** für in ihm erklärte **lokale Objekte** (Konstanten, Typen, Variablen (S. 56), Prozeduren (S. 50) und Funktionen (S. 50)). Diese Bezeichner sind durch ihre Vereinbarung gebunden (S. 55) und **verdecken** gleichnamige Objekte in umfassenden Blöcken; alle dann noch **sichtbar**en Objekte aus äußeren Blöcken sind **global** und bilden die **Umgebung** des Blockes. Der den Block direkt umfassende Block bestimmt die **unmittelbar**e **Umgebung**, von dort aus kommt man schrittweise weiter.
- Zu einer Prozedur (oder Funktion) gehören eigentlich *zwei* Blöcke: der Rumpf der Prozedur mit den zugehörigen lokalen Objekten, und ein ihn umfassender Block, in dem die Parameter-Angaben hinzukommen (ein intelligenter Compiler wird die beiden Blöcke womöglich zusammenfassen). Ebenso gehören zu dem äußersten Block eines Programms oder Moduls die **Importe** und als Umgebung ein *impliziter* umfassender Block zum Zugriff auf die ohne Vereinbarung zugänglichen **Standard-Objekte**.
- Ein Block enthält normalerweise *Anweisungen.* Damit diese ausgeführt werden können, muß der Block **aktiviert** und seine lokalen Objekte erzeugt werden; seine Umgebung muß bereits aktiviert sein.
- Der äußerste Block eines Programms und eines Moduls wird beim *Programmstart* aktiviert; seine Umgebung ist dann schon aktiv. Eine Prozedur wird beim *Aufruf,* eine Funktion bei der *Verwendung* aktiviert. Ein *eingeschachtelter* Block, der gleichzeitig eine Anweisung ist (in manchen Sprachen wie etwa ALGOL gibt es das), wird beim *Betreten* aktiviert.
- Ein Block wird beim Verlassen wieder **deaktiviert**; bei mehreren aktiven Blöcken geschieht die Deaktivierung in der *umgekehrten Reihenfolge* ihrer Aktivierung (**Last-In-First-Out**, **Keller-Prinzip**, **Stack-Verhalten**). Dieses *Verhalten* ist wesentlich, nicht seine technische Realisierung!

47 Blöcke (2)

block2

Die **Speicherverwaltung** für die Objekte eines Blocks geschieht auf folgende Weise:

- Bei der Aktivierung wird dem Block ein **Aktivierungsobjekt** (auf Englisch: **activation record**) zugeordnet (daran gebunden (S. 55)). Dies ist ein Speicherobjekt (S. 52), in dem die den *lokalen Objekten* des Blockes bei ihrer Einrichtung zugeordneten *Speicherobjekte* Platz finden, sowie Platz für diverse weitere Informationen, etwa:
  - die **Parameterangaben**, falls es solche gibt,
  - der **Resultatwert**, falls es einen solchen gibt,
  - ein Zugriffspfad auf die *unmittelbare Umgebung,* also auf das gerade gültige Aktivierungsobjekt des direkt umfassenden Blockes (im Compilerbau **statische Kette** genannt),
  - ein Zugriffspfad auf das *vor Aktivierung* des Blockes gültige Aktivierungsobjekt, das beim Verlassen des Blockes wieder zugänglich wird (im Compilerbau **dynamische Kette** genannt),
  - **Rückkehr-Information**, die angibt, wo und auf welche Weise nach Verlassen und Deaktivieren des Blockes mit der Verarbeitung fortgefahren wird.

  Beim Verlassen des Blockes wird das Aktivierungsobjekt wieder freigegeben.
- Wird ein Block **mehrfach aktiviert**, etwa bei rekursivem Aufruf (S. 29) einer Prozedur oder Funktion, so wird ihm jeweils ein *neues* Aktivierungsobjekt zugeordnet, er besitzt also dann mehrere Aktivierungsobjekte; damit existieren auch von jedem lokalen Objekt dann mehrere **Inkarnationen**; nur eine davon ist jeweils zugänglich.
- Das Gleiche gilt auch, wenn eine Prozedur nicht von sich selbst direkt *aufgerufen* wird, sondern aus einer von ihr aufgerufenen *anderen* Prozedur, eventell sogar nach weiteren Aufrufen. Das nennt man **indirekte Rekursion**.

Ist ein im Block deklariertes lokales Objekt eine Prozedur oder eine Funktion, so wird daran bei der Aktivierung des Blockes *dessen* gerade erzeugtes Aktivierungsobjekt als *direkte Umgebung* gebunden (S. 55) (**Hülle**, auf Englisch: **closure**). Diese Zuordnung bleibt erhalten, auch wenn die Prozedur später in einem *anderen* Kontext aktiviert wird, etwa aus einem inneren Block heraus, oder gar an einer ganz anderen Stelle, nachdem sie *als Parameter* weitergegeben wurde (in einigen Sprachen, darunter Modula 2, ist das möglich).

48 Bindung

bind

Der Begriff **Bindung** kommt in der Informatik, insbesondere bei Programmiersprachen, in mehreren Bedeutungen vor, die man daher leicht verwechselt:

- Im *Programmtext* kann ein Bezeichner durch eine **Vereinbarung** (Deklaration) eingeführt werden; davon kann es mehrere geben. Hier spricht man vom **definierend**en **Auftreten** (**defining occurrence**). Jedes davon verhält sich wie eine Definition (S. 11); deren Gültigkeitsbereich (S. 11) bestimmt sich durch die Blockstruktur (S. 53).

  Jede andere Verwendung desselben Bezeichners im Programmtext ist ein **angewandt**es **Auftreten** (**applied occurrence**). In einem sinnvollen Programm ist jedes solche angewandte Auftreten an ein passendes definierendes Auftreten **gebunden**, also ihm zugeordnet; welches das ist, bestimmt sich aus der Schachtelung der Gültigkeitsbereiche, und es ist Aufgabe der „**Namensanalyse**“ eines Compilers, diese Zuordnung herauszufinden.

  Wir sprechen bei dieser Art von Bindung von **Namensbindung**. Ein so gebundener Bezeichner kann ohne Änderung der Bedeutung des Programms konsistent (ohne Kollisionen!) *umbenannt* werden.

Die eben besprochene *Namensbindung* hat mit dem statischen *Programmtext* zu tun und legt teilweise seine Bedeutung fest. Zur *Laufzeit* des Programms kommen andere Vorgänge hinzu, die traditionell (leider) auch „Bindung“ heißen:

- Bei der Aktivierung (S. 54) eines Blockes werden dem Block selbst und den darin deklarierten Variablen (S. 56) (Referenzen auf) Speicherobjekte (S. 52) zugeordnet, die als Behälter für Werte dienen. An das *Objekt* (nicht an seinen Bezeichner!) wird ein Zugriffspfad (S. 52) auf seine aktuelle Realisierung **gebunden**. Dies wollen wir hier **Objektbindung** nennen.

  Ein Spezialfall davon ist die **Umgebungsbindung**: einer lokal deklarierten Prozedur (S. 50) oder Funktion wird das aktuelle Aktivierungsobjekt (S. 54) als *unmittelbare Umgebung* zugeordnet.
- Wird eine Prozedur oder Funktion aufgerufen (S. 49), so ist eine Korrespondenz zwischen den aktuellen Argumenten (S. 49) und den in der Prozedurvereinbarung erklärten formalen Parametern (S. 50) herzustellen; diese Korrespondenz nennen wir **Argumentbindung**.
- Schließlich betrachten wir im Programmtext von Prozeduren und Funktionen noch die **Parameterbindung**, welche die Korrespondenz zwischen den Parameterangaben (S. 50) im Prozedurkopf und den angewandten Vorkommen dieser Parameterbezeichner im Rumpf der Prozedur beschreibt. Sie ist ein Spezialfall der *Namensbindung*.

49 Variable: Bedeutung

var1

Das Wort „**Variable**“ wird in verschiedenen Wissensgebieten in unterschiedlicher Bedeutung gebraucht (manchmal sogar gleichzeitig!):

- In der *Mathematik* und der *mathematischen Logik* ist eine „Variable“ eine Bezeichnung, die in einem Term oder einer Formel vorkommt. Sie ist **Platzhalter** für einen *Wert,* der an ihrer Stelle eingesetzt werden kann. Gibt es dazu eine Definition (S. 11) (oder eine andere Art der Bindung), so ist die Variable dadurch **gebunden** (S. 55) und kann *konsistent umbenannt* werden (solange es dabei nicht zu Kollisionen kommt). Andernfalls heißt die Variable *frei* (manchmal auch *Parameter,* mit anderer Bedeutung des Wortes als in den Programmiersprachen!); sie kann dann in einem umfassenden Kontext abermals gebunden werden.

  Eine andere Art der Verwendung ist auch gebräuchlich: bei einem funktionalen Zusammenhang der Art y = f(x) bezeichnet man oft x als *unabhängige* und y als *abhängige* Variable.

- In der **Physik** bezeichnet man mit „Variable“ eine **Größe**, deren Wert von der Zeit oder einer anderen Größe abhängt.

Die dafür übliche Notation weicht übrigens vom Gebrauch in der Mathematik ab, ohne daß man darauf normalerweise achtet; das kann zu Überraschungen führen. Ein Beispiel:

Der freie Fall kann beschrieben werden durch

z = f(t) = - g*t*t/2 ; v = f'(t) = - g*t ;

z hängt mit v folgendermaßen zusammen:

z = h(v) = - v*v/(2*g) ;

Aus Bequemlichkeit schreibt man nun oft z(t) und z(v) dafür, bezeichnet mit z also die **Größe** und nicht den **Funktionsterm**; dieser heißt hier f(t) bzw. h(v). Auf die hier lauernden Mißverständnisse fällt man spätestens in der Thermodynamik herein.

- In der **Informatik**, vor allem bei Programmiersprachen, versteht man unter „Variable“ einen Bezug (Referenz) auf einen **Behälter** (S. 52), der Werte eines gegebenen Typs aufnehmen kann. Dem Behälter können ein oder mehrere **Speicherobjekte** (S. 52) zugeordnet sein. Eines davon ist jeweils *aktuell*, und sein Inhalt kann *ausgelesen* und auch durch **Zuweisung** (S. 41) *verändert* werden; dabei wird der Bezug angegeben.

50 Variable: Lebensdauer

var2

Variablen lassen sich klassifizieren nach der **Lebensdauer** der zugeordneten *Speicherobjekte.* Man unterscheidet meist:

- **Persistente Variablen** bezeichnen globale Objekte (z.B. Dateien), die außerhalb des Programmlaufes Bestand und Bedeutung haben.
- **Statische Variablen**: ihnen ist über die Laufzeit des Programms hinweg *genau ein* Speicherobjekt zugeordnet, das beim Programmstart eingerichtet und bei Programmende wieder gelöscht wird. So verhalten sich die explizit deklarierten Variablen im *äußersten* **Block** (S. 53) eines Programms bzw. eines Moduls.
- **Automatische Variablen** gehören zu *inneren Blöcken*; ihnen wird erst bei der **Aktivierung** (S. 54) des Blockes ein Speicherobjekt (als Teil des Aktivierungsobjektes) zugeordnet, das beim Verlassen des Blockes wieder freigegeben wird. Weil Blöcke mehrfach aktiviert sein können (etwa durch rekursiven Aufruf einer Prozedur), können derselben Variablen so mehrere **Inkarnationen** (S. 54) zugeordnet sein; und eine davon (in der Regel die jüngste) ist jeweils sichtbar.
- **Dynamische Variablen** werden im Programmlauf *explizit* erzeugt und auch explizit gelöscht; ihre Lebensdauer ist von der Blockstruktur (S. 53) unabhängig. Man stellt sich vor, daß sie aus einem eigenen Speicherbereich genommen werden, der die **Halde** genannt wird (auf Englisch: heap; dieses Wort hat mehrere Bedeutungen!) Dynamische Variablen sind gewöhnlich **anonym** und werden über einen *Zugriffspfad* angesprochen, der bei ihrer Erzeugung ausgeliefert wird.

  Es kann gleichzeitig mehrere Zugriffspfade geben, dann spricht man von einem **Alias-Phänomen** (S. 58); es ist gefährlich, denn eine Veränderung über einen der Zugriffspfade wirkt sich auch auf die anderen Pfade aus, ohne daß man dies lokal erkennt.

  Gehen alle Zugriffspfade verloren, so bleibt das der Variablen zugeordnete Speicherobjekt dennoch erhalten, und es belegt weiterhin Platz, ist aber nicht mehr zugänglich. Man spricht hier von **Garbage** oder **Daten-Müll**. In Programmiersystemen, in denen dieser Effekt häufig oder gar ständig auftritt (dazu gehören LISP, PROLOG und fast alle objektorientierten Sprachen), benötigt man daher eine *automatische* **Speicherbereinigung**, auch **Garbage Collection** genannt, die von Zeit zu Zeit solche unzugänglich gewordenen Speicherobjekte aufsucht und freigibt. Dies kostet Rechenzeit und tritt meist zu unvorhersehbaren Zeitpunkten auf; das kann bei manchen Anwendungen lästig oder störend sein.

Arbeitet die Speicherbereinigung fehlerhaft, oder gibt der Programmierer, sofern es keine solche gibt, nicht alle dynamischen Variablen wieder frei, die nicht mehr benötigt werden, so kommt es zu **Speicherschwund** („memory leaks“) mit dem Effekt, daß womöglich der freie Speicher zur Neige geht und der Programmlauf daher vorzeitig beendet wird.

Bemerkung:
Oft fragt man auch nach dem *Gültigkeitsbereich* einer *Variablen*: das gibt es nicht! Gemeint ist der Gültigkeitsbereich eines zugeordneten *Bezeichners,* also ein Block (S. 53), ggf. mit Ausnahme innerer Blöcke, in denen der Bezeichner verdeckt ist.

51 Alias-Phänomene

var3

Von einem **Alias-Phänomen** spricht man, wenn *ein und dasselbe* Speicherobjekt (S. 52) über mehrere Zugriffspfade zugänglich ist; dann verändert ein Schreibzugriff über *einen* der Pfade auch den über die *anderen* Pfade erreichbaren Wert. Dies kann natürlich zu Überraschungen führen und ist oft lokal nicht leicht zu erkennen.

Alias-Phänomene können auf unterschiedliche Weise zustande kommen:

- Zwei mit *unterschiedlichen* Index-Variablen indizierte **Komponente**n eines Array beziehen sich dennoch auf dasselbe Speicherobjekt, wenn die *Inhalte* der Index-Variablen gleich sind.
- Wenn zu einem Zugriffspfad ein **Pointer-Wert** existiert, kann dieser in *mehreren* **Pointer-Variable**n liegen.
- Ein *Referenz-Parameter* bestimmt einen Zugriffspfad. *Mehrere* Referenz-Parameter könnten an dasselbe aktuelle Objekt als Argument gebunden sein.
- Ein Objekt kann gleichzeitig sowohl *global* sichtbar sein, wie auch über einen *Referenz-Parameter.*
- In manchen Programmiersprachen lassen sich **explizite Referenz**en auf Variablen und damit *zusätzliche* Zugriffspfade konstruieren (das ist etwa in C mit dem &-Operator möglich und manchmal auch nötig, um Ergebnisse einer Prozedur zurückzuliefern, da es dort keine Referenz-Parameter gibt).

Kombinationen aller dieser Möglichkeiten können in der Praxis vorkommen, je nach der Mächtigkeit der Programmiersprache; sie zu erkennen, ist in der Regel recht schwierig und manchmal sogar vor dem Programmablauf nachweislich unmöglich. Dies hat leider Auswirkungen auf die Praxis des Compilerbaus: manche Strategien zur Umwandlung eines Programms in eine effizientere Form sind nicht korrekt anwendbar, wenn man mit Alias-Phänomenen zu rechnen hat.

52 Reihungen = Arrays

array

Um einzelne Datenelemente zu größeren Einheiten zusammenzufassen, bieten Programmiersprachen unterschiedliche Mechanismen an; eine einfache Möglichkeit ist es, **Reihungen** (Folgen fester Länge) zu bilden, die man auch als *Tabellen* interpretieren kann. Oft spricht man auch von *Feldern* oder *Vektoren*, doch haben diese Bezeichnungen noch andere Bedeutungen, die hier nicht gemeint sind, und daher vermeiden wir sie hier.

In Modula 2 (und Pascal und weiteren Sprachen) können wir neue Datentypen einführen durch eine **Arrayvereinbarung** der folgenden Art:

TYPE <T> = **ARRAY** [<Indextyp>] OF <Elementtyp>

Dabei sind <Indextyp> und <Elementtyp> Bezeichnungen für bereits bekannte Datentypen. Ein Datenelement vom Typ <T> können wir als eine **Tabelle** auffassen, deren Zeilen mit Werten vom <Indextyp> „numeriert“ sind und deren Einträge Werte vom <Elementtyp> sind; wir können sie als *Wertetabelle einer Funktion* des Typs [<Indextyp>→<Elementtyp>] deuten, doch steht dieser Aspekt meist nicht im Vordergrund.

Der <Indextyp> muß ein skalarer Typ (S. 24) sein, allerdings, aus Speicherplatzgründen, nicht INTEGER oder CARDINAL; also ein Bereichstyp (S. 24) oder ein Aufzählungstyp (S. 24). Der <Elementtyp> ist (fast) ganz beliebig. Der Typ <T> heißt ein **Arraytyp**.

Eine Variable (S. 56) von einem *Arraytyp* <T> ist eine **Arrayvariable**, auf ihre **Komponente**n kann man durch **Indizierung** zugreifen: Ist <V> eine Variablenbenennung vom Arraytyp <T>, so benennt <V>[<I>] eine *Variable* vom <Elementtyp>. <I> ist dabei ein *Wert* vom <Indextyp>.

Warnung: Ein häufiger *Programmierfehler* besteht darin, mit dem Wert des Indexausdruckes <I> die Grenzen des Indextyps zu überschreiten. Dies kann vom Übersetzer nicht immer erkannt werden und führt zu unbeabsichtigten Speicherzugriffen auf andere Objekte, oft mit katastrophalen Folgen!

Leider bieten die meisten Programmiersprachen für **Arraykonstanten** keine bequeme Notation an; statt eines konstanten Array muß man dann eine Arrayvariable verwenden, deren Komponenten man einzeln vorbesetzt.

Beispiele:

```
TYPE Dreivektor = ARRAY [1..3] OF REAL;

TYPE Farbe = (rot, gelb, gruen);
TYPE Zustand = (aus, an);
```

```
TYPE Ampel = ARRAY [Farbe] OF Zustand;
VAR Ampel1, Ampel2, Ampel3, Ampel4: Ampel;

Ampel1[rot] := an; Ampel1[gelb] := aus; Ampel1[gruen] := aus;
IF Ampel1[gruen] = an THEN fahre ELSE warte END
```

Ist der Elementtyp selbst ein Arraytyp, so erhält man mehrdimensionale Strukturen, beispielsweise **Matrizen**.

Beispiele:

```
TYPE Kreuzung = ARRAY [1..4] OF ARRAY [Farbe] OF Zustand;
VAR Kreuzung1: Kreuzung;

Kreuzung1[3][rot] := an;
```

Eine verkürzte Schreibweise ist zulässig und bequem:

```
TYPE DreiDreiMatrix = ARRAY [1..3, 1..3] OF REAL;

TYPE Kreuzung = ARRAY [1..4, Farbe] OF Zustand;

Kreuzung1[3, rot] := an;
```

Manchmal ist auch *teilweise Indizierung* nützlich:

```
Ampel3 := Kreuzung1[3];
```

53 Verbunde = Records

record

Mittels Reihungen (S. 59) kann man nur Datenelemente gleichen Typs zusammenfassen; will man Komponenten *unterschiedlicher* Typen zu einem **Verbund** kombinieren, den man als Ganzes verwalten will, so bietet sich ein **Verbundtyp** an:

TYPE <T> = **RECORD** <Felder> END

Jedes der <Felder> hat die Gestalt

<Feldname> : <Feldtyp> ;

sieht also wie eine Variablenvereinbarung aus; dabei ist <Feldname> ein *Bezeichner,* über den auf die **Komponente** des Verbundes zugegriffen werden kann, und <Feldtyp> der Typ der Komponente.

Auf die Komponenten wird durch **Selektion** zugegriffen: Ist <V> eine *Variablenbenennung* vom Verbundtyp <T>, so benennt <V>.<N>, wenn der **Selektor** <N> der <Feldname> einer der Komponenten ist, eine *Variable* vom zugehörigen <Feldtyp>.

Das mengentheoretische Analog dazu ist das kartesische Produkt (S. 21) der Feldtypen, also eine Menge von *Tupeln,* und die Feldnamen vertreten die *Projektionsfunktionen* auf die einzelnen Komponenten der Tupel (die merkwürdigerweise in der Mathematik keine Standard-Bezeichnung haben).

Der Gültigkeitsbereich (S. 56) des Feldnamens <N> in einer Selektion der Gestalt <V>.<N> ist die Selektion selbst. Der Gültigkeitsbereich kann erweitert werden durch eine **WITH-Anweisung**: ist <V> eine Benennung einer Verbundvariablen, so ist *innerhalb* der Anweisung

WITH <V> DO ... END

der Feldname <N> mit gleicher Bedeutung wie eine *Variablenbenennung* für die Komponente <V>.<N> verwendbar, und er kann auch äußere Deklarationen eines gleichnamigen Bezeichners <N> verdecken.

Leider bieten die meisten Programmiersprachen für explizite **Verbundkonstanten** keine bequeme Notation an; statt eines konstanten Verbundes muß man dann eine **Verbundvariable** verwenden, deren Komponenten man einzeln vorbesetzt.

Beispiele:

```
TYPE Datum = RECORD
                    Tag: 1 .. 31;
                  Monat: 1 .. 12;
                   Jahr: INTEGER;
             END;

TYPE Person = RECORD
                  Name: Namenstyp;
              geboren: datum;
              END;

VAR Person1: Person;

Person1.Name := "Peter Ustinov";

WITH Person1.geboren DO
  Tag := 16; Monat := 4; Jahr := 1921;
END;
```

54 Sequenzen; Formale Sprachen

sequenz

Um Zeichenfolgen über einem endlichen Zeichenvorrat A zu beschreiben, beginnen wir zweckmäßig mit dem kartesischen Produkt (S. 21) von A mit sich selbst.

Die Menge A×A besteht aus den geordneten Paaren <a1,a2> mit a1, a2 ∈ A. Ebenso besteht A×A×A aus den Tripeln von Elementen aus A, und ebenso für beliebig viele Faktoren A; wir sprechen hier von **Sequenzen** über A, und schreiben sie oft mit spitzen Klammern.

Wir können Sequenzen miteinander verknüpfen oder **verketten**, indem wir sie aneinanderhängen:

<a1,a2><a3,a4,a5> = <a1,a2,a3,a4,a5>

diese Operation ist *assoziativ*.

Ist A ein **Alphabet**, also ein endlicher Zeichenvorrat, so lassen wir bei den Sequenzen meist die spitzen Klammern und Kommas weg und sprechen von **Wörter**n über dem Alphabet A. Die Menge A selbst entspricht dann den Wörtern der Länge 1, und oft nehmen wir ein **leeres Wort** ε, also die Sequenz <> der Länge 0, mit hinzu; es ändert bei der Verkettung nichts (es ist ein **neutrales Element**).

Die Menge *aller* endlichen Wörter über einem Alphabet A:

$$\mathbf{A^*} = \{\varepsilon\} \cup A \cup (A\times A) \cup (A\times A\times A) \cup \ldots$$

heißt in der Algebra **freies Monoid** über A; weshalb, tut hier nichts zur Sache.

Jede Teilmenge L von A* nennen wir eine **Formale Sprache** (über A); fast alle davon sind *unendlich*, und wir können nur mit solchen praktisch umgehen, von denen es eine *endliche* Beschreibung gibt!

A* ist *abzählbar.* (S. 20) und damit auch alle Formalen Sprachen über A (soweit sie nicht ohnehin endlich sind); aber die *Menge aller Formalen Sprachen* über A ist *überabzählbar.* (S. 20) Andererseits gibt es nur *abzählbar* viele endliche Beschreibungen. Unsere Möglichkeiten erscheinen also recht stark eingeschränkt; aber für die Praxis genügen sie bei weitem.

Es ist oft bequem, auch eine assoziative *Verkettung* für Formale Sprachen einzuführen:

$$MN = \{\ mn \mid m\in M \wedge n\in N\ \}$$

Auch hier gibt es ein *neutrales Element* oder **Einselement**, nämlich $\{\varepsilon\}$, die Menge, die nur ε enthält. Die **leere Sprache** $\emptyset$ ist dagegen ein **Nullelement**:

$$M\{\varepsilon\} = \{\varepsilon\}M = M, \qquad M\emptyset = \emptyset M = \emptyset$$

55 Grammatiken (1)

gramm1

Die meisten interessanten Formalen Sprachen (S. 62) sind *unendlich*; um mit ihnen zu arbeiten, brauchen wir eine *endliche* Beschreibung. Der Linguist Noam Chomsky hat dafür einen bequemen Mechanismus angegeben; dieser war für natürliche Sprachen gedacht, ist aber für Formale Sprachen noch besser geeignet.

Eine **Chomsky-Grammatik** G wird gegeben durch 4 Komponenten (N, T, P, S):

- T = eine endliche Menge von Symbolen, genannt **Terminalsymbole**.
- N = eine endliche Menge von *Hilfsbegriffen*, genannt **Nichtterminalsymbole**.
- P = eine endliche Menge von **Produktionen** oder **Regeln**,
- S ∈ N, das **Startsymbol**.

(Die merkwürdige Ausdrucksweise mancher Theoretiker: „Eine Grammatik G *ist* ein Viertupel (N, T, P, S)“ bedeutet genau dasselbe!)

Wir verlangen: N∩T = ∅, und es ist praktisch, V = N∪T als **Vokabular** einzuführen (unsere Bezeichnungsweise entspricht dem Standard; im Lehrbuch „Skriptum Informatik“ steht es merkwürdigerweise anders).

Die Regeln in P werden in der Form (α,β) oder meistens $\alpha \to \beta$ geschrieben; dabei sind α und β nahezu beliebige Zeichenfolgen aus V*, aber in α muß mindestens ein Element aus N vorkommen, also: $\alpha \in V^* \times N \times V^*$, $\beta \in V^*$.

Ein **Ableitungsschritt** xαy ⇒ xβy ist ein Übergang von einer beliebigen Zeichenfolge xαy aus V*, in der α vorkommt, zur Zeichenfolge xβy ; dabei ist $\alpha \to \beta$ eine Regel aus P.

Eine **Ableitung** S ⇒* w ist eine Folge von 0 oder mehr Ableitungsschritten, die bei S anfängt. Das Ergebnis w heißt eine **Satzform** (S ist also auch eine Satzform).

Ist eine Satzform w ∈ T*, enthält sie also kein Element von N mehr, so heißt sie ein **Satz**.

Die *Menge aller Sätze*, also der Zeichenfolgen *aus T*,* die durch Ableitungen aus S erzeugt werden können, heißt *die durch G erzeugte Sprache* **L(G)**. Die Hilfsbegriffe aus N kommen also darin nicht mehr vor; L(G) kann in Sonderfällen sogar leer sein.

Es gibt Grammatiken von der Art, daß in einer Ableitung dieselbe Satzform mehrfach auftreten kann. Das ist nutzlos und unerwünscht; man kann die Grammatik stets so umformen, ohne die erzeugte Sprache zu verändern, daß dies nicht vorkommen kann, und wir nehmen an, dies sei geschehen. Dann hat jede Satzform eine Ableitung endlicher Länge, und man kann die möglichen Ableitungen systematisch so erzeugen, daß jede Satzform genau einmal gefunden wird. Dann wird erst recht jeder Satz gefunden: die Sprache L(G) ist **aufzählbar**.

56 Grammatiken (2)

gramm2

Wenn wir auch ein Erzeugungsverfahren (S. 63) besitzen, das die zur Sprache L(G) gehörenden Wörter systematisch alle aufzählt, so sind wir doch meist auch an einem **Entscheidungsverfahren** interessiert, das für ein vorgelegtes Wort w ∈ T* feststellt, ob es in L(G) liegt oder nicht. Leider braucht es das allgemein nicht zu geben. Daher teilt man die Grammatiken (und die zugeordneten Sprachen) in Klassen ein danach, ob es ein Entscheidungsverfahren gibt, und womöglich gar ein effizientes. Die Klassifikation richtet sich nach der Gestalt der Regeln $\alpha \to \beta$:

- **Klasse 0**: keine Einschränkung.

  Hier muß kein Entscheidungsverfahren existieren.

- **Klasse 1** (**kontextsensitiv**e Grammatiken): |α| <= |β|;
  also darf die linke Seite einer Regel nicht länger als die rechte Seite sein.

  Hier gibt es immer ein Entscheidungsverfahren, aber seine Kosten sind womöglich astronomisch hoch.

- **Klasse 2** (**kontextfrei**e Grammatiken): α ∈ N;
  die linke Seite ist ein *einziges* Nichtterminalsymbol, das bei einem Ableitungsschritt unabhängig von seiner Umgebung ersetzt wird, daher der Name *kontextfrei.*

  Die Kosten des Entscheidungsverfahrens wachsen für Wörter der Länge n höchstens wie n**3 an, in günstigeren Fällen gar nur proportional zu n. Dies ist in der Praxis brauchbar; hierher gehören die meisten Programmiersprachen.

- **Klasse 3** (**reguläre** Grammatiken): zusätzlich β ∈ T oder auch β ∈ TN;
  das Entscheidungsverfahren macht für ein Wort der Länge n nur n Schritte!

In allen Fällen erlauben wir als Ausnahme die Regel $S \Rightarrow \varepsilon$, falls S auf *keiner* rechten Seite vorkommt. Das macht einige Sätze einfacher, etwa den folgenden:

> Für die Mengen der zugehörigen Sprachen gilt:
> $Klasse3 \subset Klasse2 \subset Klasse1 \subset Klasse0$

Hier haben wir ausgenutzt, daß man die *Sprachen* auch in Klassen einteilen kann nach der höchsten Nummer einer Klasse, in der es zu der Sprache eine Grammatik gibt (es gibt zu einer Sprache immer viele Grammatiken, deren Äquivalenz man nicht immer leicht sieht).

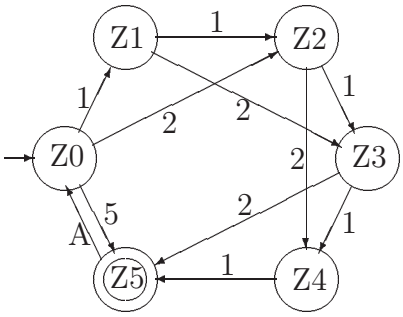
57 Endliche Automaten (1)

auto1

Wir wollen einen idealisierten Zigarettenautomaten simulieren, der aus einem unerschöpflichen Vorrat einer einzigen Sorte nach Einwurf des passenden Geldbetrags und Drücken der Ausgabetaste eine Schachtel auswirft. Der Preis soll DM 5 betragen; der Automat akzeptiert Geldstücke zu DM 1, DM 2 und DM 5 sowie die Taste A. Was bei Überzahlungen geschehen soll (abweisen, vereinnahmen, oder gar wechseln), und ob es eine Rückgabetaste geben soll, lassen wir vorerst offen; die Erweiterung ist leicht, macht aber unsere Beschreibung komplizierter.

Den *inneren* **Zustand** (S. 16) des Automaten beschreiben wir durch ein Element aus einer endlichen **Zustandsmenge** Q = {Z0 .. Z5}; jeder Zustand bedeutet den bislang gespeicherten Geldbetrag. Z0 ist der **Startzustand**, Z5 ein **Endzustand**.

Die Wirkungsweise des Automaten können wir graphisch veranschaulichen durch ein **Übergangsdiagramm**; die Übergänge zwischen den Zuständen sind jeweils durch die aktuelle **Eingabe** bestimmt.



Bei größeren Zustandszahlen ist eine **Übergangstafel** übersichtlicher. Sie kann als Wertetabelle einer Funktion (S. 38) vom Typ [Q×E]→Q mit zwei Argumenten gelesen werden, wobei E = {'1','2','5','A'} die Menge der zulässigen Eingaben ist, oder auch als zweidimensionale Matrix (S. 59) vom Typ ARRAY [Q,E] OF Q. Die leeren Felder geben an, in welchen Situationen wir das Verhalten bisher noch nicht festgelegt haben.

	1	2	5	A
Z0	Z1	Z2	Z5	
Z1	Z2	Z3		
Z2	Z3	Z4		
Z3	Z4	Z5		
Z4	Z5			
Z5				Z0

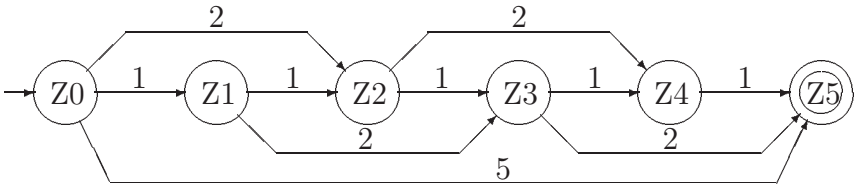
Mit diesen Angaben den Automaten per Programm zu simulieren, sei den Lesern als Übungsaufgabe überlassen, ebenso die Erweiterung der Beschreibung um die Ausgabe eines Wortes über einem geeigneten **Ausgabealphabet** bei jedem Übergang.

58 Endliche Automaten (2)

auto2

Die zulässigen Eingabefolgen, die vom Startzustand Z0 zum Endzustand Z5 führen, bilden offensichtlich eine Formale Sprache (S. 62), die unser Automat **erkennen** kann. Wir wollen diesen Zusammenhang weiter untersuchen.

Wir lassen die Simulation der Ausgabetaste weg und zeichnen das Übergangsdiagramm etwas um:



Jeder zulässige Weg vom Startzustand Z0 zum Endzustand Z5 ist ein Wort einer Formalen Sprache (S. 62); wir suchen dafür eine Grammatik (S. 63). Tatsächlich können wir sie aus dem Diagramm oder auch aus der Übergangstafel direkt ablesen:

Z0 → 1 Z1
Z0 → 2 Z2
Z0 → 5 Z5
Z1 → 1 Z2
Z1 → 2 Z3
Z2 → 1 Z3
Z2 → 2 Z4
Z3 → 1 Z4
Z3 → 2 Z5
Z4 → 1 Z5
Z5 → ε

Dies sind Regeln einer regulären Grammatik (S. 64) für die Sprache der zulässigen Eingaben, bis auf die störende letzte Regel, die wir aber beseitigen können, indem wir Z5 auf den rechten Seiten einfach weglassen.

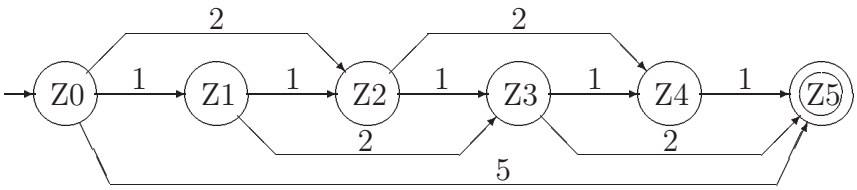
Umgekehrt können wir zu jeder regulären Grammatik (S. 64) systematisch einen endlichen Automaten für dieselbe Sprache konstruieren: Jedem Nichtterminalsymbol (S. 63) entspricht ein Zustand, das Startsymbol (S. 63) wird Anfangszustand. Die Regeln, auf deren rechter Seite nur ein Terminalsymbol (S. 63) steht, ergänzen wir um ein *neues* Nichtterminalsymbol, das Endzustand wird. Die Übergänge ergeben sich direkt aus den Grammatikregeln.

59 Endliche Automaten (3)

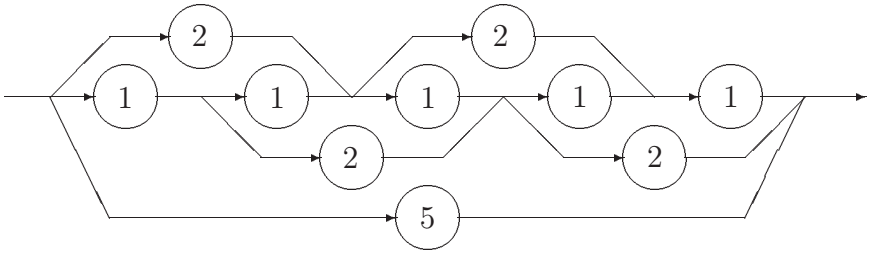
auto3

Der aus einer regulären Grammatik schematisch erhaltene Automat liefert uns ein Erzeugungsverfahren (S. 64), ist aber leider meistens **nichtdeterministisch**: von einem Zustand aus kann es bei gleicher Eingabe *mehrere* mögliche Übergänge geben. Man kann ihn systematisch in einen äquivalenten **deterministisch**en Automaten umwandeln; wie das geht, erfahren wir in der **Automatentheorie**. So erhalten wir auch ein Entscheidungsverfahren (S. 64); der neue Automat kann allerdings recht groß werden.

Das Übergangsdiagramm unseres Automaten



betont die *Zustände,* deren Bezeichnung aber für die erzeugte Sprache gar keine Rolle spielt. Wenn wir die Zustände unterdrücken und stattdessen die *Übergänge* hervorheben, erhalten wir eine gleichwertige Darstellung, aus der man die erzeugte Sprache noch besser ablesen kann:



Dies ist ein *Sonderfall* der **Syntaxdiagramme**, wobei im Diagramm *nur* Terminalsymbole vorkommen; wie wir sehen konnten, ist er gleichmächtig mit den endlichen Automaten, und auch mit den regulären (S. 64) Grammatiken (Klasse 3). Jeder zulässige Weg vom Eingang zum Ausgang definiert ein Wort der erzeugten Sprache.

Im allgemeinen Fall können in Syntaxdiagrammen außer den hier rund gezeichneten Terminalsymbolen auch *Nichtterminalsymbole* vorkommen, die man üblicherweise durch einen rechteckigen Kasten kennzeichnet; für jedes davon ist ein beliebiges daraus ableitbares Wort einzusetzen. Die genauen Regeln der Abarbeitung besprechen wir später. Diese allgemeinen Syntaxdiagramme erfassen die *kontextfreien* (S. 64) Sprachen (Klasse 2).

60 Syntaxdiagramme: Aufbau

syntax1

Jeder Chomsky-Grammatik (S. 63) der Klasse 2 oder 3 kann man folgendermaßen eine Kollektion von Syntaxdiagrammen zuordnen:

- Aus jeder Produktion machen wir einen gerichteten Graphen, der für jedes Symbol auf der rechten Seite einen Knoten enthält. Für Terminalsymbole zeichnen wir runde bzw. ovale Knoten, die mit dem Symbol markiert sind; ebenso verwenden wir rechteckige Kästen zur Veranschaulichung der Nichtterminalsymbole. Die Symbole bzw. ihre Darstellungen sind in der Reihenfolge der Aufschreibung verkettet; dazu kommt als Eingang noch ein Pfeil, der von außen auf das erste Symbol der rechten Seite zeigt, und als Ausgang ein Pfeil vom letzten Symbol nach außen. Ist eine rechte Seite leer, so entspricht ihr ein Pfeil, der sowohl Eingang wie Ausgang ist.
- Alle so aufgebauten Ketten, die zum *gleichen* Nichtterminalsymbol gehören, bekommen einen gemeinsamen Eingang und einen gemeinsamen Ausgang; das so entstandene Gebilde ist ein Syntaxdiagramm für das Nichtterminalsymbol. Es gibt dann also für jedes Nichtterminalsymbol genau ein Diagramm; oft läßt es sich noch vereinfachen, indem man am Eingang oder am Ausgang gemeinsame Teile zusammenfaßt.


Umgekehrt kann man aus jeder Kollektion von Syntaxdiagrammen durch Aufspalten und womöglich Einführen von neuen Nichtterminalsymbolen eine Grammatik herleiten, welche dieselbe Sprache erzeugt. Wir gehen darauf nicht ein.


Ist die Grammatik von der Klasse 3, so lassen sich in den Diagrammen die Nichtterminalsymbole entfernen. Das Verfahren ist zu mühsam, um es hier vorzuführen; das Diagramm direkt hinzuschreiben, ist meist einfacher.

61 Syntaxdiagramme: Formularmaschine

syntax2

Um aus einer Kollektion von Syntaxdiagrammen (S. 68) Wörter der durch sie definierten Sprache zu *erzeugen*, können wir eine **Formularmaschine** verwenden:

- Für jedes **Diagramm** gibt es einen Satz **Formulare** (Kopien nach Bedarf).
- Es gibt ein *aktuelles Blatt,* mit dem wir arbeiten, und einen **Stapel** von teilweise bearbeiteten Blättern; anfangs ist er leer, und das aktuelle Blatt gehört zum Startsymbol.

- wir suchen auf dem aktuellen Blatt einen zulässigen Weg vom Eingang zum Ausgang und notieren die Folge der dabei akzeptierten Symbole als Ergebnis des Blattes:
  - wenn wir auf einen (runden) **Terminalknoten** stoßen, so akzeptieren wir das zugehörige Symbol und gehen weiter;
  - stoßen wir auf einen (eckigen) **Nichtterminalknoten**, so notieren wir die aktuelle Stelle und das bisherige Ergebnis auf dem Blatt und legen es oben auf den Stapel; wir beginnen ein neues Formular der Sorte, die zu dem Nichtterminal gehört.
- Haben wir den Ausgang des aktuellen Formulars erreicht, so notieren wir das Resultat und legen das Blatt beiseite. Nun gibt es zwei Fälle:
  - ist der Stapel nicht leer, so holen wir das oberste Blatt als aktuelles Formular, übertragen das aktuelle Resultat als Ergebnis des bearbeiteten Nichtterminalknotens, und fahren fort;
  - war der Stapel leer, so ist unser Resultat das Endergebnis.
- Das beschriebene Verfahren funktioniert *so nicht,* wenn die Diagramme **linksrekursiv** sind, wenn man also beim Aufruf weiterer Diagramme, ohne ein Zeichen gelesen oder produziert zu haben, zum Ausgangsdiagramm zurückkommt. Hier hilft eine *Umformung* der Diagramme weiter.

Jedem Exemplar eines Diagramms können wir einen *endlichen Automaten* zuordnen; wir haben also einen **Stapel** oder **Keller** von endlichen Automaten: einen **Kellerautomat**en.

Da wir jede Grammatik (S. 64) der Klasse 2 in Syntaxdiagramme umformen können, sind Kellerautomaten also ebenso mächtig.


Man kann versuchen, die Formularmaschine auch zum **Erkennen** von Wörtern der beschriebenen Sprache zu verwenden; dann muß aber an jeder Verzweigung jedes Diagramms an Hand der Eingabe erkennbar sein, wie es weiter geht. Dies ist oft, aber leider nicht für alle Grammatiken so (die **LL(1)-Bedingungen** müssen erfüllt sein); dann braucht man andere Methoden, die in der Theoretischen Informatik und im Compilerbau besprochen werden.

62 Sprache und Metasprache

meta

Wenn man sich mit der Untersuchung einer (realen oder formalen) Sprache beschäftigt, so tut man dies auch mit sprachlichen Mitteln; man muß dann sorgsam darauf achten, auf welcher *Sprachebene* man sich bewegt. Wir unterscheiden zwischen der **Objektsprache**, die untersucht wird, und der **Metasprache**, in der wir über die Objektsprache reden. Verwechslung der beiden Sprachebenen, wenn wir etwa in der *Objektsprache* über Sätze der Objektsprache reden, führt leicht zu logischen Paradoxien, wie etwa: „Dieser Satz ist falsch“.

Die Problematik tritt auch bei formalen Sprachen (S. 62) auf. Will man eine Grammatik (S. 63) so präzise niederschreiben, daß man sie maschinell weiterverarbeiten kann, etwa um einen Erkennungsmechanismus automatisch zu generieren, so muß man 4(!) Zeichenvorräte sorgfältig auseinanderhalten:

- den *Grund-Zeichensatz,* den unsere Ein/Ausgabegeräte beherrschen (heute meist ASCII (S. 25) oder eine Erweiterung davon),
- die Terminalsymbole (S. 63),
- die Nichtterminalsymbole (S. 63),
- die **Metazeichen**: Zeichen des Grund-Zeichensatzes, die weder als Terminalsymbole noch als Nichtterminalsymbole dienen, sondern zur Aufschreibung und ggf. Gliederung der *Grammatikregeln* selbst benutzt werden.

Wir geben im Folgenden einige Mechanismen an, um Grammatiken bzw. Sprachen der Klassen (S. 64) 2 und 3 zu notieren. In diesen Klassen kann man jedem Nichtterminalsymbol eine formale Sprache zuordnen, nämlich die Menge der daraus ableitbaren Folgen von Terminalsymbolen. Damit läßt sich auch eine Folge von Terminal- und Nichtterminalsymbolen als Definition einer Sprache interpretieren, und man kann die Grammatikregeln als **Mengengleichungen** für Wortmengen lesen; deren Lösungen sehen allerdings gelegentlich unübersichtlicher aus als die ursprünglichen Regeln selbst.

63 Reguläre Ausdrücke (1)

regula1

Ein oft bequemes Hilfsmittel zur Notation von Sprachen der Klasse 3 (S. 64) sind **reguläre Ausdrücke** über einem (Terminal-)Alphabet A. Jeder solche Ausdruck R bezeichnet eine Menge von Wörtern aus A*, und die Ausdrücke sind durch folgende Regeln definiert:

- ε ist ein regulärer Ausdruck,
- a ist ein regulärer Ausdruck für jedes a∈A,
- (R) ist ein regulärer Ausdruck, wenn R ein regulärer Ausdruck ist (Klammerung),
- R1|R2 ist ein regulärer Ausdruck, wenn R1 und R2 reguläre Ausdrücke sind (Alternative),
- R* ist ein regulärer Ausdruck, wenn R ein regulärer Ausdruck ist (Stern-Bildung),
- sonst gibt es nichts.

Die Symbole { ε ( ) | * } sind hier Metasymbole (S. 70), die nicht zu A gehören. Die Bedeutung der regulären Ausdrücke ist folgende:

- ε bezeichnet die *Menge* $\{\varepsilon\}$, die nur aus dem *leeren Wort* besteht;
- a bezeichnet die *(einelementige)* Menge {a};
- (R) bedeutet *dieselbe* Menge wie R, die Klammern dienen nur der Zusammenfassung;
- R1|R2 bedeutet die *Vereinigung* R1∪R2;
- R* bezeichnet die Menge, die durch 0-mal oder öfter durchgeführte Verkettung (S. 62) der Menge R mit sich selbst entsteht, also:
  $R^* = \{\varepsilon\} \cup R \cup RR \cup RRR \cup \ldots$

Manchmal verwendet man auch R+ als Abkürzung für R R* = R* R = R* - $\{\varepsilon\}$.

Die regulären Ausdrücke sind ebenso mächtig wie die anderen Mechanismen zur Beschreibung von Sprachen der Klasse 3 (S. 64).

In der Kommandosprache des Betriebssystems *UNIX* (einschließlich seiner Varianten wie *Linux*) werden häufig Erweiterungen der angegebenen Notation verwendet; sie bringen nichts Neues, erleichtern aber öfters die Aufschreibung.

64 Reguläre Ausdrücke (2)

regula2

Reguläre Ausdrücke sind besonders geeignet zur Definition der *Grundsymbole* von Programmiersprachen (S. 13), z.B.:

- Ziffer = '0' | '1' | ... | '9'
- Buchstabe = 'a' | 'b' | ... | 'z' | 'A' | ... | 'Z'
  dabei muß man statt der Punkte die Regeln einmal voll hinschreiben;
- Ganzzahl = Ziffer (Ziffer)* oder auch: Ganzzahl = Ziffer+
- Bruchzahl = Ziffer* '.' Ziffer+
  dabei haben wir wie oben den Apostroph ' als Metazeichen (S. 70) zur Bezeichnung unserer ASCII-Grundzeichen verwendet. Auch das Gleichheitszeichen = ist hier ein Metazeichen.
- Bezeichner = Buchstabe (Buchstabe | Ziffer)*

Diese Beschreibung (es gibt für Modula 2 noch ein paar ähnliche Regeln mehr), läßt sich direkt maschinell verarbeiten, etwa zu einem Lesemodul für einen Übersetzer; auch Syntaxdiagramme oder Erkennungsautomaten lassen sich daraus leicht ablesen.

65 BNF und EBNF

bnf

Die regulären Ausdrücke (S. 71) gehen in ihrer Mächtigkeit nicht über die regulären Sprachen, also die Sprachklasse 3 (S. 64) hinaus; in der Praxis ist jedoch auch die Klasse 2 von großer Bedeutung, daher benötigen wir weitere Mechanismen.

Der Chomsky-Formalismus (S. 63) hat nur ein einziges Metazeichen (S. 70), den Pfeil $\rightarrow$; für die Terminal- und Nichtterminalsymbole braucht er je ein eigenes Alphabet. Für die automatische Verarbeitung ist er daher kaum geeignet; anders ist es mit den folgenden Mechanismen:

BNF (Backus-Naur-Formalismus) wurde ca. 1959 zur Notation der (kontextfreien (S. 64)) Grammatikregeln für die Programmiersprache (S. 13) ALGOL 60 eingeführt:

- Metazeichen (S. 70) sind ::= für den Pfeil, < > | und ' (Apostroph).
- Nichtterminalsymbole (S. 63) werden als *Bezeichner* in spitzen Klammern < und > notiert.
- Terminalsymbole (S. 63) sind *Sonderzeichen* oder Kombinationen davon wie := , die sich selbst bezeichnen, oder **Wortsymbole**: Bezeichner eingeschlossen in Apostrophe ('begin').

- Mehrere rechte Seiten zum gleichen Nichtterminalsymbol können durch den Strich | zu Alternativen zusammengefaßt werden.

EBNF (Erweiterter Backus-Naur-Formalismus) ist eine bequeme Erweiterung von *BNF,* wenn auch mit etwas anderen Konventionen:

- Metazeichen (S. 70) sind = ( ) | * " und der Punkt.
- Der Pfeil → wird durch = wiedergegeben; am Ende jeder Regel steht zur Klarheit ein Punkt.
- Nichtterminalsymbole werden durch *Bezeichner* wiedergegeben.
- Terminalsymbole werden in Anführungszeichen " eingeschlossen. Sie können Einzelzeichen, Kombinationen davon oder Bezeichner (Wortsymbole) sein, die genau so wie in den Regeln geschrieben werden müssen.
- Auf der rechten Seite einer Regel kann ein beliebiger regulärer Ausdruck (S. 71) aus Terminal- *und Nichtterminalsymbolen* stehen. Für jedes Nichtterminalsymbol gibt es nur eine einzige Regel.

Die Syntax der meisten Programmiersprachen wird heutzutage meist mittels EBNF definiert; daneben gibt es oft noch Syntaxdiagramme (S. 68) zur Veranschaulichung, die sich jedoch zur automatischen Verarbeitung nicht eignen.

Als Beispiel für EBNF folgen hier die Regeln für den zulässigen Aufbau regulärer Ausdrücke:

R = ε | A | "(" R ")" | R "|" R | R "*" .

dabei haben wir die Menge A der Grundzeichen als bekannt und gegeben vorausgesetzt.

Mit BNF bzw. EBNF beschreiben wir *kontextfreie Grammatiken* in einer Form, die für die automatische Verarbeitung geeignet ist; es gibt **Parser-Generatoren**, die dies mit Vorteil ausnutzen, um automatisch ein Erkennungsverfahren zu generieren. Nicht für alle Grammatiken der Klasse 2 ist dies möglich, aber für eine praktisch ausreichend große Teilmenge davon.

Auch die Regeln für das EBNF-Format selbst lassen sich wiederum in EBNF angeben! Damit kann man einen Parsergenerator schrittweise aufbauen: ein Prototyp braucht nur die Grammatik der EBNF selbst zu verstehen; mit seiner Hilfe lassen sich dann mächtigere Versionen generieren. Diese Methode, genannt **Bootstrap**, kann den Gesamtaufwand erheblich reduzieren (wir haben vor einigen Jahren, neben der Compilerbau-Vorlesung, einen Generator auf diese Art innerhalb von 3 Wochen geschrieben).

66 Attributierte Grammatiken

attrib

Am Beispiel der Türme von Hanoi (S. 88) läßt sich gut ein Konzept demonstrieren, das beim Compilerbau sehr hilfreich ist, um über den formal korrekten *Aufbau* eines Textes hinaus Teile der *Bedeutung* zu erfassen: es gibt dafür eine **Attributierte Grammatik**.

Damit ist folgendes gemeint: Wir gehen aus von einer kontextfreien (S. 64) Grammatik und ordnen jedem Nichtterminalsymbol eine Klasse (S. 16) zu; in den Regeln stehen dann jeweils Instanzen (S. 16) dieser Klassen, die wir, falls nötig, durch Indizes unterscheiden. Jedes Objekt hat einen Satz von Attributen (S. 16), welche mit seiner *Bedeutung* zu tun haben können, und die Attributwerte stehen innerhalb einer Regel in *Beziehungen* untereinander, die als Zusatzforderungen zu den kontextfreien Regeln hinzutreten und etwa die Menge der *syntaktisch korrekt* aufgebauten Texte in Richtung der *sinnvollen* Texte einschränken.

In unserem Beispiel hat eine *Scheibe* ein Attribut *Durchmesser,* und ein *Turm* hat einen *Durchmesser* und eine *Höhe.* Wir schreiben die Attribute wie Komponenten eines Verbundes (S. 60), und bekommen damit folgende Regeln:

Turm → <leer>

Turm.Höhe = 0
Turm.Durchmesser = 0

Turm → Scheibe Turm_1

Turm.Höhe = Turm_1.Höhe + 1
Turm.Durchmesser = Scheibe.Durchmesser
Scheibe.Durchmesser > Turm_1.Durchmesser

Damit haben wir auch die Nebenbedingungen unseres Problems in einer Form erfaßt, die nahe an den Grammatikregeln liegt, und die von einem intelligenten Generator womöglich automatisch verarbeitet werden kann.

67 Algorithmen

algor1

Unter einem **Algorithmus** verstehen wir eine *präzise, endliche Verarbeitungsvorschrift*, die auf endlich vielen wohldefinierten Grundoperationen von jeweils endlicher Ausführungszeit aufbaut. Er definiert eine *Funktion* von seiner Eingabe in seine Ausgabe, und er *beschreibt* eine *Realisierung* dieser Funktion.

Nicht jede Verarbeitungsvorschrift betrachten wir als Algorithmus; wir verlangen von einem Algorithmus:

- er löst eine *Klasse* von Problemen; die Klasse ist sein *Definitionsbereich*;
- die Länge seiner Beschreibung und der Platzbedarf während der Ausführung sind endlich;
- für jede Eingabe aus dem Definitionsbereich ergibt sich das Ergebnis nach endlich vielen Schritten (der Algorithmus *terminiert*);
- wir lassen zu, daß während der Ausführung eines Algorithmus für den nächsten Schritt noch Wahlmöglichkeiten bestehen; dann nennen wir ihn *nichtdeterministisch,* andernfalls *deterministisch.*


Manchmal betrachtet man auch nicht-terminierende Algorithmen, etwa bei Steuerungsaufgaben oder im Bereich der Betriebssysteme. Sie lösen eine von vornherein nicht beschränkte Folge von Problemen gleicher Art nacheinander; jeder dieser Schritte ist dann ein Algorithmus im obigen Sinne.

Eine reale oder gedachte Funktionseinheit, welche die einzelnen Arbeitsschritte eines Algorithmus ausführt, nennen wir eine **Maschine**.

Beispiele dafür sind etwa: die endlichen Automaten (S. 65), die Kellerautomaten (S. 68), unsere Formularmaschine (S. 68) für Syntaxdiagramme, die virtuelle Modula-Maschine, auf der unsere Modula-Programme ablaufen, und die später besprochene Turing-Maschine (S. 77). Auch unser Computer selbst gehört natürlich dazu!

68 Berechenbarkeit

berechn

Unter einer (totalen) Funktion (S. 38) f von einer Menge M in eine Menge N verstehen wir eine *bestehende Zuordnung*, die für *jedes* Element m von M *eindeutig* ein Element n von N vorgibt, das wir mit f(m) bezeichnen. Eine Funktion f nennen wir **berechenbar**, *wenn man sie berechnen kann,* wenn es also einen Algorithmus (S. 75) gibt, der für jedes Element m aus dem Definitionsbereich M das zugeordnete f(m) nach endlich vielen Schritten liefert.

Man möchte vermuten, daß jede präzise festgelegte Funktion auch berechenbar ist; leider ist das aber nicht der Fall. Das kann man folgendermaßen einsehen:

- Ein Algorithmus hat eine endliche Beschreibung über irgend einem endlichen Zeichensatz; es gibt abzählbar (S. 62) unendlich viele endliche Texte, und daher nur abzählbar viele Algorithmen-Beschreibungen. Daher gibt es erst recht nur *abzählbar* viele berechenbare Funktionen.
- Betrachten wir die auf **N** definierten Funktionen f, die als Ergebnisse nur die Werte 0 oder 1 liefern; jede davon definiert eine Teilmenge Mf $\subset$ **N** *derjenigen* Zahlen m $\in$ **N**, für die f(m) = 1 ist. Umgekehrt gibt es für jede Teilmenge M $\subset$ **N** eine solche Funktion fM, die **charakteristisch**e Funktion der Menge M. Die Menge der Teilmengen M $\subset$ **N** ist aber *überabzählbar,* (S. 62) ebenso also die Menge ihrer charakteristischen Funktionen; und damit die Menge *aller* Funktionen erst recht.

Demnach muß es also *weitaus mehr* nicht-berechenbare als berechenbare Funktionen geben, und einige davon kennt man auch! Für praktische Zwecke sind natürlich nur berechenbare Funktionen brauchbar, und von ihnen gibt es auch genug.

Beispiele:

Funktionen, die durch einen Rechenausdruck gegeben sind, sind natürlich berechenbar.

Die folgenden Funktionen wären von enormer praktischer Bedeutung, wenn sie berechenbar wären; aber das sind sie leider nicht:

- Das *Halteproblem:* Sei A die Menge aller Algorithmus-Beschreibungen (es kommt nicht darauf an, über welchem Zeichensatz und in welchem Formalismus), und E die Menge aller möglichen Eingaben dafür. Die Funktion f: A$\times$E $\rightarrow$ Boolean, welche f(a,e) = *true* liefert, falls ein Algorithmus a$\in$A angewandt auf eine Eingabe e$\in$E terminiert, ist nicht berechenbar.

  Das wird in der Theoretischen Informatik bewiesen und bedeutet: es gibt kein allgemeines Verfahren, um zu entscheiden, ob ein gegebenes Programm bei einer vorgegebenen Eingabe anhalten wird.
- Das *Äquivalenzproblem:* Die Funktion g: A$\times$A $\rightarrow$ Boolean, welche g(a1,a2) = *true* liefert, falls zwei Algorithmen a1 und a2 dieselbe Funktion berechnen, ist nicht berechenbar.

  Es ist also nicht allgemein entscheidbar, ob zwei Programme dasselbe tun.

Natürlich kann man die genannten Probleme sehr oft dennoch lösen; aber ein Verfahren dafür, das in *allen* Fällen brauchbar ist, gibt es nachweislich nicht.

69 Turing-Maschine (1)

turing1

Die **Turing-Maschine** ist ein Gedankenmodell einer Maschine (S. 65), das von Alan Turing bereits 1936 eingeführt wurde, um Algorithmen und Fragen der Berechenbarkeit (S. 75) und Entscheidbarkeit zu untersuchen. Für praktische Anwendungen war es nicht gedacht (und ist auch dafür nicht besonders geeignet, wohl aber für theoretische Untersuchungen).

Die Turingmaschine ist leicht zu verstehen, wenn wir von dem berühmten Marsfahrzeug „Pathfinder“ ausgehen und es nur geringfügig vereinfachen und verallgemeinern. Wir betrachten die Marsoberfläche als unendlich ausgedehnt, fahren aber nur in einer festen Richtung in einzelnen Schritten hin und her; dabei entnehmen wir als Proben Zeichen aus einem festen Zeichensatz und hinterlassen auch entsprechende Spuren, die wir später wieder lesen können.

Genauer: wir betrachten ein unendlich langes Band aus einzelnen Zellen, die Zeichen aus einem Alphabet, dem **Bandalphabet** B tragen. Ein Zeichen davon, das **Leerzeichen** z, ist ausgezeichnet; damit sind alle Zellen beschriftet bis auf einen endlich langen Ausschnitt. Anfangs stehen dort nur Zeichen aus einer Teilmenge von B, dem **Eingabealphabet** I⊂B, welches z nicht enthält. Als Steuerung unseres „Fahrzeugs“ verwenden wir einen endlichen Automaten (S. 65) mit der **Zustandsmenge** Q und dem **Anfangszustand** q0∈Q. Die Übergangstafel des Automaten beschreibt, was die Maschine tun kann: abhängig vom aktuellen Zustand q und dem gerade gelesenen Zeichen x bekommen wir einen neuen Zustand q’, ein Zeichen x’, das an die Stelle des gerade gelesenen Zeichens auf das Band geschrieben wird, und eine der drei Aktionen L (gehe einen Schritt nach links), R (gehe nach rechts) und H (halte an; fertig!).

Wir setzen die Turing-Maschine auf ein vorbereitetes Band an (meist auf das linke Ende des nichtleeren Abschnitts); sie kann jeweils das Zeichen lesen und überschreiben, auf dem sie gerade steht. Sie führt, gesteuert durch ihr „Programm“ in der Übergangstafel, eine Reihe von Operationen aus und bleibt schließlich stehen, wenn sie die Aktion H erreicht; dann steht das „Ergebnis“ der Berechnung auf dem Band. Es kann auch vorkommen, daß sie *nicht* stehen bleibt: dann gibt es *kein* Ergebnis, weil die Berechnung nicht endet. Die Maschine kann auch „steckenbleiben“, wenn sie in der Steuertafel ein leeres Feld findet, in dem keine Reaktion vorgegeben ist; das ist ein Fehlerfall, und auch hier gibt es kein sinnvolles Ergebnis.

70 Turing-Maschine (2)

turing2

Wir zeigen die Arbeitsweise einer Turing-Maschine (S. 77) an einem Beispiel: wir programmieren eine primitive **Subtraktionsmaschine** für natürliche Zahlen, die wir in der einfachsten möglichen Darstellung codieren, nämlich als Folgen eines einzigen Zeichens a.

Unser Bandalphabet soll B = {a,b,z} sein, und anfangs steht auf dem Band die Angabe „x - y“ in folgender Form: x-mal das Zeichen a, mindestens ein Trennzeichen b, y-mal das Zeichen a; rechts und links davon stehen beliebig viele Leerzeichen z (tatsächlich genügt hier jeweils eines davon).

Die Maschine nutzt aus, daß gilt:

x - y = x falls y = 0
x - y = (x-1) - (y-1) sonst

Die Zustände haben folgende Bedeutung:

q0 = nimm ein Zeichen a von x weg (bilde x-1)
q1 = laufe nach rechts bis zur Trennung b
q2 = laufe nach rechts bis y und nimm dort ein a weg (bilde y-1)
q3 = prüfe, ob jetzt y=0 ist
q4 = laufe nach links bis an den Anfang von x
q5 = ebenso, aber lösche alle b
q6 = fertig!

Als „Programm“ haben wir die Tafel

	a	b	z
q0	q1 z R		
q1	q1 a R	q2 b R	
q2	q3 b R	q2 b R	
q3	q4 a L		q5 z L
q4	q4 a L	q4 b L	q0 z R
q5	q5 a L	q5 z L	q6 z R
q6	q0 a H		

Wir wollen nicht behaupten, daß dieses Programm besonders gut lesbar wäre; aber ein kleines Beipiel selbst durchzuspielen klärt vieles.

71 Entscheidbarkeit

entsch

In unserem Beispiel (S. 78) haben wir nur natürliche Zahlen verarbeitet; wie sich herausgestellt hat, lassen sich aber ganz beliebige Daten und Texte codieren, auch Beschreibungen von Algorithmen und sogar von Turing-Maschinen selbst! Wir haben hier also ein zwar sehr primitives und gähnend langsames, aber dafür extrem mächtiges und flexibles Analysewerkzeug vor uns.

Über die Theorie der Turing-Maschinen (S. 77) hinaus sind weitere Formalismen untersucht worden, um den Begriff der Berechenbarkeit genauer zu fassen, etwa die sog. **Markoff-Algorithmen**, oder die (partiell oder total) *rekursiven Funktionen* (mit unserer Definition von Rekursion (S. 29) haben sie nur am Rande zu tun). Hierher gehört auch die Frage nach der **Entscheidbarkeit** von Formalen Sprachen (S. 62), also ob es für eine Formale Sprache ein Entscheidungsverfahren (S. 64) gibt.

Dabei hat sich herausgestellt, daß es zu jeder Grammatik (S. 63) von der Klasse 0 (also zu *jeder* Grammatik!) eine äquivalente Turing-Maschine gibt, und umgekehrt. Für die Klasse 1 kann man das Band sogar außerhalb der Eingabe abschneiden, und für die anderen Klassen ergeben sich noch stärkere Spezialisierungen der Maschinen, bis hin zu den endlichen Automaten.

Alle diese Untersuchungen haben immer denselben Berechenbarkeitsbegriff ergeben, und das führt auf die berühmte **Church'sche These**:

> Jede berechenbare Funktion läßt sich programmieren.

Diese These, von der es auch andere Formulierungen gibt, läßt sich nicht beweisen, weil die verwendeten Begriffe nicht präzise genug definiert sind. Sie ist heute dennoch allgemein akzeptiert.

Wichtig ist ihre Umkehrung:

> Es gibt praktisch wichtige, wohldefinierte Probleme, die sich nicht entscheiden und nicht programmieren lassen.

Dies mahnt zur Bescheidenheit; allerdings gibt es keinen Anlaß zur Resignation: wichtige und entscheidbare Probleme gibt es noch gerade genug!

72 Programmaufbau

program

Grundsätzlich ist ein Programmodul (S. 28) folgendermaßen aufgebaut:

MODULE <Modulname>;
<Importliste>
<Block>
END <Modulname>.

Die **Importliste** besteht aus Einträgen folgender Arten:

- FROM <Klasse> IMPORT <Name>, <Name> ... ;
- IMPORT <Klasse>;

Im ersten Fall werden aus der Schnittstelle (S. 28) der Klasse einzelne <Namen> zur lokalen Verwendung bereitgestellt, im zweiten Fall *alle* dort definierten Namen ohne Einzelangabe. Im zweiten Fall können sie aber nur in der Form <Klasse>.<Name> als **qualifizierter Name** angesprochen werden; anderenfalls gäbe es zu viele unbeabsichtigte Kollisionen mit lokal eingeführten Bezeichnern.

Der **Block** (S. 53) besteht aus *Vereinbarungen* und Anweisungen (S. 41).

Durch Vereinbarungen (S. 80) werden neue lokale Bezeichner (S. 26) für *Objekte* unterschiedlicher Art eingeführt:

- Konstantenvereinbarungen (S. 80)
- Typenvereinbarungen (S. 80)
- Variablenvereinbarungen (S. 83)
- Prozedur (S. 50)- und Funktionsvereinbarungen (S. 51)

Die *Anweisungen* des Blockes im Hauptprogramm werden ausgeführt, sobald seine Ablaufumgebung zur Verfügung steht (dazu gehören auch die Importe, deren Module initialisiert sein müssen).

Für ein Implementierungsmodul (S. 28) sieht es im Grunde ebenso aus; jedoch kommen die Vereinbarung aus seinem Definitionsmodul (S. 28) noch implizit dazu. Sein Block wird ausgeführt, sobald alle seine Importe zur Verfügung stehen, aber vor dem Block des Hauptprogramms. Es ist Sache des Binders, eine geeignete Initialisierungsreihenfolge bei mehreren Moduln herauszufinden.

73 Deklarationen

decl

Durch Vereinbarungen werden neue lokale Bezeichner (S. 26) für *Objekte* unterschiedlicher Art eingeführt:

- Konstantenvereinbarungen (S. 80)
- Typenvereinbarungen (S. 80)
- Variablenvereinbarungen (S. 83)
- Prozedurvereinbarungen (S. 50) und Funktionsvereinbarungen (S. 51)

Eine **Konstantenvereinbarung** hat die Form

CONST <Bezeichner> = <Ausdruck>;

der Wert des Ausdrucks muß bereits *zur Übersetzungszeit* bestimmt werden können. Eine Konstantenvereinbarung empfiehlt sich, wenn eine Konstante mehrfach vorkommt oder unbequem zu notieren ist, oder wenn sie eine Bedeutung hat, die über ihren Zahlenwert hinausgeht (etwa die Dimension eines Vektorraumes). Dies steigert die Lesbarkeit des Programms und erleichtert eine Erweiterung auf verwandte Fragestellungen:

one person's constant is another person's variable

Über eine **Typvereinbarung** wird ein Datentyp *umbenannt* oder ein neuer Datentyp eingeführt durch Ableitung von bereits bekannten Typen. Wir unterscheiden dabei

- Aufzählungstypen (S. 24)
- Bereichstypen (S. 24)
- Arraytypen (S. 59)
- Verbundtypen (S. 60)
- Mengentypen (S. 82)
- Pointertypen (S. 84)

Eine **Variablenvereinbarung** hat die Form

VAR <Bezeichner>: <Typ>;

sie führt einen *Bezeichner* als **Benennung** für einen Behälter (S. 52) ein, der Werte eines gegebenen *Typs* aufnehmen kann.

Eine Prozedurvereinbarung (S. 50) bzw. Funktionsvereinbarung (S. 51) beschreibt sowohl die Aufrufschnittstelle (S. 49) einer Prozedur oder Funktion, wie auch ihre Realisierung.

74 Mengentypen

set

Ein **Mengentyp** umfaßt die **Potenzmenge** der Wertemenge zu einem gegebenen **Basistyp**, also die Menge aller seiner Teilmengen:

TYPE <T> = SET OF <BT>;

dabei ist der Basistyp <BT> ein skalarer Typ (S. 24), also meist eine Aufzählung (S. 24) oder ein Ausschnittyp (S. 24). Sein zulässiger Umfang ist bei den meisten Implementierungen unangenehm stark eingeschränkt.

Konstante *Werte* eines Mengentyps <T> haben die Form <T>{ c1, ... , cn }, dabei sind c1 bis cn Konstanten des Basistyps <BT>. Zu jedem Mengentyp <T> gehört mit <T>{} auch ein Exemplar der leeren Menge Ø. Variable Elemente vom Basistyp kann man in eine Menge aufnehmen oder ausschließen durch die *Anweisungen:*

- INCL(S, x) mit der Bedeutung: $S := S \cup \{x\}$
- EXCL(S, x) mit der Bedeutung: $S := S \setminus \{x\}$

Die Grundoperationen der Mengenlehre sind für Mengentypen vorhanden, werden aber anders geschrieben:

- S1 + S2 : die Vereinigung $S1 \cup S2$
- S1 * S2 : der Durchschnitt $S1 \cap S2$
- S1 - S2 : die Differenz $S1 \setminus S2 = \{ x \in S1 \mid \neg x \in S2 \}$
- S1 / S2 : die symmetrische Differenz $S1 \oplus S2 = S1 \setminus S2 \cup S2 \setminus S1$

Hier sind die Ergebnisse wieder Mengen desselben Typs; daneben gibt es noch die Abfragen mit Ergebnis BOOLEAN:

- S1 <= S2 bedeutet: $S1 \subseteq S2$
- S1 >= S2 bedeutet: $S2 \subseteq S1$
- x IN S bedeutet: $x \in S$

Vordefiniert ist der Mengentyp **BITSET** als SET OF [0 .. N-1], dabei ist N typisch die Wortlänge des Rechners. Er ist vor allem zur Manipulation von *Bitmustern* gedacht. Bei Konstanten dieses Typs braucht die Typangabe nicht geschrieben zu werden; alle anderen Mengentypen werden intern auf diesen Typ abgebildet, daher auch die Größeneinschränkungen. In manchen Implementierungen gibt es auch noch **SET OF CHAR**, mit eigener Realisierung.

75 Variablendeklaration

vardef

Eine **Variablenvereinbarung** hat die Form

VAR <Bezeichner>: <Typ>;

sie führt einen *Bezeichner* als **Benennung** für einen Behälter (S. 52) ein, der Werte eines gegebenen *Typs* aufnehmen kann. Bei Aktivierung (S. 54) des Blockes (S. 53), zu dem die Variable gehört, wird dem Behälter ein Speicherobjekt (S. 52) zugeordnet, das über einen Zugriffspfad (S. 52), die **Referenz**, erreichbar ist.

Kommt eine Variablenbenennung im Programmtext vor, so bedeutet dies je nach dem Kontext *Verschiedenes:*

- auf **Referenz-Position**, also auf der linken Seite einer Zuweisung (S. 41) oder als Referenz-Argument (S. 49) einer Prozedur oder Funktion, ist die *Referenz,* also der **Bezug** auf den Behälter gemeint;
- auf **Wert-Position**, also im Rahmen eines *Ausdrucks* oder als Wert-Argument (S. 49) einer Prozedur oder Funktion, bedeutet die Benennung den **Inhalt** des Behälters.

Bezeichnet eine Variablenbenennung eine Variable eines strukturierten Typs, so können von ihr weitere Benennungen für die Komponenten abgeleitet werden, denen ebenfalls Referenzen entsprechen. So ist

- wenn a eine Array-Variable (S. 59) ist, a[i] eine Benennung für die Komponente des Array a zum Index-Wert i,
- wenn a eine Verbund-Variable (S. 60) ist, a.x eine Benennung für die durch den Selektor x ausgewählte Verbundkomponente.
- Indizierung und Selektion können womöglich mehrfach angewandt werden; so entstehen kompliziertere Variablenbenennungen, denen jeweils wieder ein Zugriffspfad auf ein Speicherobjekt entspricht.

Die einer Variablenbenennung entsprechende *Referenz* ist eine *verallgemeinerte Adresse* des zugeordneten Behälters. Sie braucht nicht direkt eine *Maschinenadresse* oder Hausnummer (S. 52) zu sein; manche Prozessoren besitzen allgemeinere Adressierungsmodi. Es gibt Programmsysteme, wo die Referenz ihren Wert sogar während des Programmlaufs ändern kann, ohne daß der Programmierer das bemerken muß!

Die *Referenzen* auf Behälter für Werte eines Typs T bilden *selbst* eine Klasse **Ptr(T)**, den T zugeordneten Pointertyp (S. 84). Mit Werten eines Pointertyps kann nur sehr eingeschränkt gerechnet werden (außer in C und seinen Verwandten, und dort ist es riskant), aber man kann über sie auf den *Inhalt* des Behälters lesend und schreibend zugreifen.

76 Pointertypen

pointer

Die *Referenzen* auf Behälter für Werte eines Typs T bilden *selbst* eine Klasse **Ptr(T)**; ihre Instanzen nennt man auch **Zeiger** oder **Pointer**.

Auch in Modula können wir zu jedem gegebenen Typ <T> einen Pointertyp <PT> definieren:

TYPE <PT> = POINTER TO <T>;

Pointerwerte sind Bezüge (S. 83) auf Speicherobjekte, entstehen also erst zur Laufzeit unseres Programms; daher gibt es für sie auch keine Standardbezeichnungen. Eine Ausnahme bildet der Pointerwert **NIL** (anderswo auch: NULL oder *void*); er ist in jedem Pointertyp vorhanden und bezieht sich auf *gar nichts*, ist aber wohldefiniert.

Auf Pointerwerten sind nur wenige Operationen definiert:

- die Vergleiche = und <>
- die Zuweisung eines Pointerwertes an eine **Pointervariable** (vom Typ <PT>)
- die **Dereferenzierung**: der Zugriff auf den zugehörigen Behälter.

Die *Dereferenzierung* eines Pointerwertes p wird p^ oder p↑ (manchmal auch als p ->) geschrieben; dies bedeutet eine Variablenbenennung (S. 83) für den Behälter, auf den p verweist. Ist p kein legaler Pointerwert, verweist also p nicht *wirklich* auf einen *existierenden* Behälter, so spricht man von einem **dangling pointer**; ein Zugriffsversuch kann katastrophale Konsequenzen haben!

In Modula 2 und Pascal bekommen wir Pointerwerte als Referenzen auf *neue* Speicherobjekte: ist p eine Variable eines Pointertyps <PT>, so erzeugt die Standardprozedur **NEW**(p) in einem *eigenen* Speicherbereich, genannt die Halde (S. 57) (Heap) ein Speicherobjekt von dem Typ <T>, auf den die Pointerwerte in <PT> verweisen, und legt die Referenz in p ab. Die dadurch definierte dynamische Variable (S. 57) kann über die *Benennung* p↑ angesprochen werden und lebt (S. 57) so lange, bis wir sie mittels der Standardprozedur **DISPOSE**(p) wieder frei geben, oder bis zum Ende des Programmlaufes. Das gilt selbst dann, wenn uns zwischendurch alle Zugriffspfade auf das Objekt abhanden gekommen sind; und kommt dies zu oft vor, so kann die Halde erschöpft werden.

In C und verwandten Sprachen kann man eine Referenz auf eine Variable a in der Form &a erzeugen; das braucht man dort auch zur Simulation von Referenzargumenten (S. 49), die es ansonsten nicht gibt. Dort bedeutet auch eine Benennung einer Arrayvariablen (S. 59) eine Referenz auf das erste Element des Array, und damit kann man sich Referenzen auf die weiteren Elemente berechnen; eine leider äußerst fehleranfällige Technik!

77 Aufwand und Effizienz

effiz

Eines der Kriterien zur Beurteilung eines Algorithmus, und keineswegs immer das wichtigste, ist die Frage nach dem **Aufwand** (gemessen in einem geeigneten Maß) für die Lösung eines gegebenen Problems. Andere Eigenschaften wie Einfachheit, Klarheit und Verständlichkeit, Wartbarkeit und Änderbarkeit, Übertragbarkeit und Anpaßbarkeit, Robustheit und Fehlertoleranz (und natürlich die Korrektheit!) sind oft weitaus bedeutsamer. Von Interesse ist der Aufwand oder die **Effizienz** dann, wenn die bei konkretem Einsatz entstehenden **Kosten** nicht vernachlässigbar oder gar unerträglich hoch sind.

Im Einzelfall wird man den Aufwand einfach *messen,* aber um für weitere Einsätze eine Vorhersage machen zu können, sollte man etwas über die Abhängigkeit des Aufwandes von der Problemgröße wissen, zumal dann, wenn mehrere Verfahren zur Auswahl stehen. Man wird also versuchen, den Aufwand zu berechnen. Dies kann sehr detailliert geschehen und ist dann oft sehr schwierig; oft genügen aber schon globale Aussagen der Art, wie stark der Aufwand mit der Problemgröße wächst, und sie sind oft recht einfach zu erhalten. Eine solche **asymptotische Abschätzung** ist recht grob und nur bis auf einen konstanten Faktor bekannt, dennoch kann sie zur Auswahl eines geeigneten Verfahrens ausreichen.

Um über das **Wachstumsverhalten** von Funktionen reden zu können, definieren wir:

> Die **Größenordnung** einer Funktion f: **N**→**N** ist die Menge der Funktionen g, die schließlich *nicht schneller anwachsen* als f:
>
> $\mathcal{O}$(f) = { g: **N**→**N** | ∃ c>0 ∃ n0 mit g(n) <= c*f(n) ∀ n >= n0 }
>
> f(n) ist also für genügend große n bis auf einen konstanten Faktor c eine *obere Schranke* für g(n).

Für g∈$\mathcal{O}$(f) sagt man gerne: g(n) ist *von der Größenordnung* f(n). Gebräuchlich sind auch die Sprechweisen:

- g∈$\mathcal{O}$(1) : g *hat konstanten Aufwand*
- g∈$\mathcal{O}$(log n) : g *wächst logarithmisch*
- g∈$\mathcal{O}$(n) : g *wächst linear*
- g∈$\mathcal{O}$(n**k) für ein k > 1 : g *wächst polynomiell*
- g∈$\mathcal{O}$(a**n) für ein a > 1 : g *wächst exponentiell*
- es gilt: $\mathcal{O}(1) \subset \mathcal{O}(\log n) \subset \mathcal{O}(n) \subset \mathcal{O}(n^{**}k) \subset \mathcal{O}(a^{**}n)$

78 ggT: größter gemeinsamer Teiler (1) ggt1

Wir fragen nach einem Verfahren, den größten gemeinsamen Teiler **ggT(a,b)** von zwei natürlichen Zahlen a und b zu berechnen. Das aus dem Gymnasium bekannte Verfahren verwendet die Primzahlen, die wir nicht bequem zur Verfügung haben; wir machen deshalb einen direkten Ansatz. Es kommt uns ohnehin nicht auf die *Lösung* des Problems an (die ist schon seit 2000 Jahren bekannt), sondern auf den *Lösungsweg*, der womöglich lehrreich ist.

Wir beginnen mit einigen einfachen mathematischen Beobachtungen. Wir stellen fest:

- wenn a einen Teiler d hat, also $\exists$ x$\in$**N** : a = x*d, dann hat auch jedes Vielfache n*a den Teiler d: n*a = (n*x)*d
- alle Vielfachen von d haben den Teiler d, und d ist die kleinste Zahl davon
- wenn a und b den Teiler d haben, dann hat ihn auch jede Linearkombination davon: sei a = x*d, b = y*d, so ist m*a + n*b = (m*x + n*y)*d
- wenn eine Zahl c den Teiler d hat, so ist d <= c
- die Menge der ganzzahligen positiven Linearkombinationen von a und b ist nicht leer und hat also ein kleinstes Element, das den Teiler d hat
- dies gilt für *alle* gemeinsamen Teiler von a und b

Daraus ergibt sich nun:

> der größte gemeinsame Teiler von zwei gegebenen Zahlen a und b ist deren kleinste positive Linearkombination

Wir suchen nun durch systematisches Subtrahieren möglichst kleine Kombinationen aufzubauen; das ist korrekt, weil eine Linearkombination von Linearkombinationen wieder eine Linearkombination der Ausgangswerte ist. Kommen wir dabei zum Ende, so haben wir wegen ggT(n,n) = n den gesuchten Wert gefunden; und wir *müssen* zum Ende kommen, weil unsere Zahlen immer kleiner werden. So kommen wir auf das Verfahren:

```
PROCEDURE ggT(a, b: CARDINAL): CARDINAL;
(* PRE: a>0, b>0 *)
BEGIN IF a=b THEN RETURN a
      ELSIF a>b THEN RETURN ggT(a-b, b)
      ELSE RETURN ggT(b-a, a)
      END
END ggT;
```

79 ggT: größter gemeinsamer Teiler (2)

ggt2

Das soeben gefundene Verfahren ist rekursiv (S. 29); als Problemgröße (S. 29) nehmen wir das jeweils größere der beiden Argumente, und dieses nimmt bei jedem Schritt echt ab; also terminiert das Verfahren.

Leider ist das Verfahren sehr schlecht: hat etwa b den Wert 1, so kommen wir nur bei jedem Schritt um 1 nach unten, und wir brauchen a Schritte, was sehr viel sein kann. Um es zu verbessern, nehmen wir statt der Differenz den Divisionsrest, der auch eine Linearkombination ist, aber immer kleiner als die kleinere der beiden Zahlen. So springen wir in viel größeren Schritten. Allerdings müssen wir nun aufpassen: der Rest kann 0 werden, und dann sind wir fertig. So bekommen wir den neuen Ansatz:

```
PROCEDURE ggT(a, b: CARDINAL): CARDINAL;
(* PRE: a>=0, b>=0 *)
BEGIN IF a=0 THEN RETURN b
      ELSIF b=0 THEN RETURN a
      ELSIF a>b THEN RETURN ggT(a MOD b, b)
      ELSE RETURN ggT(b MOD a, a)
      END
END ggT;
```

Diese Version, die **Euklidischer Algorithmus** genannt wird, ist schon seit ca. 2000 Jahren bekannt, und hat sich bewährt. Wir wollen abschätzen, wie gut sie ist.

Dazu betrachten wir den *schlimmsten Fall* (**worst case**), wir suchen also Paare (a,b), wo das Verfahren besonders viele Schritte braucht. Dazu sollen a und b *teilerfremd* sein (sonst sind wir früher fertig), und der Sprung von der größeren Zahl a nach a MOD b soll möglichst klein sein. Das ist so, wenn a DIV b = 1 ist, und wir haben dann a MOD b = a - (a DIV b)*b = a - b, wir kommen also bei unserem ersten Verfahren heraus, wenn dies bei *jedem* Schritt so ist.

Nennen wir die Folge der jeweils größeren Zahlen Z(n), Z(n-1), ... bei insgesamt n Schritten, so muß sein: Z(n) = Z(n-1) + Z(n-2), und man sieht leicht, daß auch Z(2) = Z(1) = 1 sein muß. Wir laufen also im schlimmsten Falle die Folge der Fibonacci-Zahlen (S. 90) herunter. Man weiß (S. 143), daß diese etwa exponentiell (S. 85) anwachsen: F(n) ≈ c*ϕ**n; und wegen log F(n) ≈ log c + n*log ϕ ist also die Schrittzahl n etwa proportional zum Logarithmus der größeren Zahl F(n).

Wir haben also sogar im schlimmsten Falle logarithmischen (S. 85) Aufwand, und damit kann man sehr zufrieden sein. Die Kosten der bekannten Alternative, nämlich die gemeinsamen Primfaktoren herauszuziehen, sind demgegenüber *geradezu astronomisch!* Wenn man die Rekursion durch eine Schleife ersetzt (wir übergehen dies hier), so ergibt sich noch eine Verbesserung um einen konstanten Faktor.

80 Türme von Hanoi

hanoi

Die Sage berichtet, daß in einem buddhistischen Tempel bei Hanoi die Mitglieder eines Mönchsordens seit Urzeiten damit beschäftigt seien, eine mühsame, aber für die Welt als Ganzes sehr wichtige Arbeit zu verrichten:

Dort stehen drei Pfosten, auf denen 64 zylindrische Scheiben von jeweils verschiedenem Durchmesser liegen. Anfangs lagen alle Scheiben auf dem ersten Pfosten, und sie sollen auf den dritten Pfosten einzeln umgelegt werden; dabei kann der zweite Pfosten als Zwischenablage verwendet werden, aber es darf nur immer eine kleinere auf einer größeren Scheibe zu liegen kommen.

Wenn diese Arbeit einst vollendet sein wird, ist der Zweck des Weltalls erreicht; die Menschheit wird vom Zwang der ewigen Wiedergeburt erlöst und darf ins Nirvana eingehen.

Das Problem sieht sehr kompliziert aus, vor allem was die dabei nötige Buchhaltung betrifft, und wir wollen die Mönche bei ihrem sehr lobenswerten Werk durch eine Computersimulation unterstützen. Nach längerer Überlegung zeigt sich, daß es nützlich ist, einige Hilfsbegriffe einzuführen:

- ein *Turm* ist ein Stapel von *Scheiben;* dabei liegt immer eine kleinere Scheibe auf einer größeren;
- die *Höhe* eines Turmes ist die Anzahl der Scheiben; sie kann 0 sein, weil wir auch *leere* Türme betrachten wollen;
- der *Durchmesser* eines Turmes sei gleich dem Durchmesser seiner untersten Scheibe, oder 0 für einen leeren Turm;
- ein Turm ist entweder leer, oder er besteht aus einer Scheibe und einem darauf stehenden Turm kleineren Durchmessers und um 1 kleinerer Höhe.

Aus dieser nunmehr rekursiven (S. 29) Beschreibung der Situation können wir nun den Ansatz einer ebenfalls rekursiven Lösung für den Transport eines Turmes gewinnen:

- für einen leeren Turm ist nichts zu tun;
- um einen nichtleeren Turm von einem Pfosten A auf einen Pfosten E zu verlagern, bringen wir den auf der untersten Scheibe stehenden kleineren Turm (auf die gleiche Weise) auf den dritten Pfosten Z, legen die verbliebene Scheibe von A nach Z um, und holen dann den kleineren Turm (auf die gleiche Weise) vom Hilfspfosten Z nach E.

Das Verfahren ist nun leicht auszuprogrammieren:

```
TYPE Pfosten = (Anfang, Zwischen, Ende);

PROCEDURE bewegeTurm(A, Z, E: Pfosten; h: Cardinal);
BEGIN IF h > 0 (* sonst ist nichts zu tun *)
      THEN bewegeTurm(A, E, Z, h-1);
           umlegeScheibe(A, E);
           bewegeTurm(Z, A, E, h-1)
      END
END bewegeTurm;
```

Die Prozedur *bewegeTurm* transportiert vom Pfosten A die obersten h Scheiben auf den Pfosten E, und sie verwendet dabei den dritten Pfosten Z als Zwischenspeicher.

Daß das Verfahren terminiert (S. 29), ist leicht zu sehen: wir können als Größenmaß (S. 29) für das Problem wahlweise die Höhe h oder den Durchmesser des Turmes ansehen, und in beiden Fällen sind die Teilprobleme kleiner. Nicht so leicht zu sehen ist, daß das Verfahren *korrekt* ist insoweit, daß dabei immer kleinere Scheiben auf größeren zu liegen kommen. Dies zu beweisen ist Übungsaufgabe; uns kommt es hier auf etwas anderes an, nämlich auf den **Aufwand**.

Sei A(n) der Aufwand, einen Turm der Höhe n zu bewegen, und c1, c2, c3 irgendwelche Konstanten. Dann kann man aus dem Programm sofort ablesen:

A(n) = 2 * A(n-1) + c1 für n > 0, A(0) = c2

Dies ist eine **Differenzengleichung**, und deren Theorie wird heute kaum noch gelehrt; wir lösen sie hier *ad hoc,* durch teilweises Probieren:

A(n) + c1 = 2 * (A(n-1) + c1)
oder mit B(n) = A(n) + c1, A(n) = B(n) - c1:
B(n) = 2 * B(n-1) = c3 * 2**n mit irgend einem Faktor c3

Wenn wir noch B(0) = c3 = A(0)+c1 = c1+c2 verwenden und nach A(n) auflösen, erhalten wir:

A(n) = (c1+c2) * 2**n - c1

also eine extrem schnell wachsende Funktion. Wie schnell die Funktion 2**n wächst, zeigen einige Zahlenbeispiele:

2**10 = 1024 ≈ 10**3, 2**20 ≈ 10**6, 2**60 ≈ 10**18,

und mit 64 Scheiben und nur 10 Sekunden für das Umlegen einer Scheibe kommt man bei ca. 5 Billionen Jahren heraus, was mit „astrophysikalischer Genauigkeit“, also etwa in der Größenordnung, die Annahmen der Astronomen über die weitere Lebensdauer der Sonne um den Faktor 20 übertrifft. An der Sage könnte also wohl etwas dran sein, aber derzeit besteht noch kein Anlaß zur Besorgnis.

81 Fibonacci-Zahlen (1) fib1

Leonardo di Pisa, genannt *Fibonacci,* hat in seinem 1202 erschienenen Rechenbuch *Liber abaci* ein Modell dafür angegeben, wie sich die Kaninchen vermehren (nicht die Art und Weise, die war schon bekannt, sondern die Geschwindigkeit).

Die ihm zu Ehren so genannte Folge der **Fibonacci-Zahlen** tritt in der Mathematik, der Informatik, der Architektur und auch in vielen anderen Bereichen immer wieder unvermutet wie ein Springteufel auf (einmal habe ich mit ihrer Hilfe sogar in der Spielbank gewonnen), und es lohnt sich, sie genauer zu untersuchen; an ihrem Beispiel lassen sich auch viele Programmiertechniken gut studieren.

Das (etwas unrealistische) Modell unserer Kaninchen-Population betrachtet Paare von ewig lebenden Kaninchen, die nach ihrem ersten Lebensmonat jeden Monat ein weiteres Paar produzieren. Innerhalb eines beliebigen Monats sind also die Kaninchen noch da, die im Monat vorher vorhanden waren; dazu kommen die Nachkommen derjenigen, die damals schon einen Monat alt waren, also einen Monat früher schon da waren. Wir fangen mit einem neugeborenen Paar im Monat 1 an; wenn wir mit F(n) die Anzahl der Paare im Monat n bezeichnen, so gilt also

$$\begin{array}{lll} (1) & F(n) = F(n\text{-}1) + F(n\text{-}2) & \text{für } n > 2 \\ (2) & F(1) = F(2) = 1 & \end{array}$$

Die Folge der ersten Werte 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 läßt sich bequem von unten herauf berechnen (will man das unbedingt vermeiden, so kann man sie auch an einem Kunstwerk in der Neuen Staatsgalerie in Stuttgart ablesen).

Aus der rekursiven (S. 29) Definition ergibt sich unmittelbar eine Funktionsprozedur (S. 51):

```
PROCEDURE Fib(n: CARDINAL): CARDINAL;
BEGIN IF n <= 2 THEN RETURN 1
      ELSE RETURN Fib(n-1) + Fib(n-2)
      END;
END Fib;
```

Die Prozedur ist ganz offensichtlich korrekt; aber wenn man sie in einem geeigneten Rahmen ablaufen läßt, so stellt man mit einiger Verwunderung fest, daß sie schon für mäßig hohe Werte von n überraschend lange braucht, während uns die direkte Berechnung der ersten Werte weder Schwierigkeiten noch merklichen Zeitaufwand eingebracht hat. Dies ist erstaunlich; und wir werden später untersuchen (S. 143), woran das liegen mag. Vorerst geben wir uns mit einer korrekten Lösung zufrieden.

82 Abstrakte Datentypen (1)

adt1

Pointer (S. 84) und dadurch zugängliche dynamische Variable (S. 84) sind ein bequemes Mittel, um **dynamische Datenstruktur**en aufzubauen, deren Umfang sich zur Laufzeit des Programms ändern kann. Diese lassen sich wiederum zu neuen *Datentypen* zusammenfassen, und wenn man diese in einer Weise definiert, daß ihr *Verhalten* für die Anwendung hinreichend genau bekannt ist, ohne aber die interne Realisierung im Detail festzulegen, so nennt man sie **abstrakte Datentypen**.

Besonders bequem kann man abstrakte Datentypen durch eine *funktionale* oder **algebraische Spezifikation** definieren. In diesem Sinne besteht ein abstrakter Datentyp aus:

- einer Menge von *Typen,* hier meist **Sorten** genannt;
- einer Menge von *Operationen,* also Funktionen, deren Definitionsbereich und Wertevorrat, also deren **Funktionalität** oder **Arität** angegeben sind; *Konstanten* lassen sich als *nullstellige Funktionen* hier mit unterbringen;
- einer Menge von **Regeln** oder **Axiome**n über bestehende Zusammenhänge zwischen den Operationen.

Mit den Operationen lassen sich nun ganz beliebige *formal korrekt* aufgebaute (**wohlgeformte**) **Term**e bilden, und in der so gewonnenen **Termalgebra** kann man mittels der gegebenen Regeln *rechnen.* Wir erhalten so eine **algebraische Struktur** (die uns schon bekannten Strukturen wie Halbgruppen, Gruppen, Ringe, Körper, Vektorräume etc. passen hier auch hinein!) Durch die Regeln zerfällt die Termalgebra in *Äquivalenzklassen* von Termen, und diese sind die *Objekte* unseres Datentyps. Manchmal unterscheidet man bei den Operationen auch **Konstruktor**en, mittels deren wir Objekte aufbauen, und **Projektor**en, über die wir auf Komponenten oder Attribute von Objekten zugreifen. Allein aus Konstruktoren aufgebaute Terme nennt man **Grundterm**e. Oft enthält jede Äquivalenzklasse genau einen Grundterm, auf den sich die anderen Elemente der Klasse mittels der Regeln reduzieren lassen, und der damit das Objekt repräsentiert.

Wir nennen eine *algebraische Struktur* oft kurz eine **Algebra** oder, wenn es uns auf ihre Verwendung innerhalb von Programmen ankommt, eine **Rechenstruktur**. In diesem Falle werden wir nicht direkt auf der Termalgebra arbeiten (das ist meist recht unbequem), sondern eine äquivalente, womöglich effizientere oder leichter handhabbare Implementierung wählen. Deren *Schnittstelle* ist durch die Angabe der Sorten und der Funktionalität der Operationen gegeben, während ihr gewünschtes *Verhalten* durch die Regeln näher spezifiziert wird. Für die Frage, ob die Implementierung korrekt ist, hat man gerade die Einhaltung der Regeln nachzuprüfen.

83 Abstrakte Datentypen (2)

adt2

Als Beispiel für einen abstrakten Datentyp (S. 91) betrachten wir lineare Listen (S. 94) beliebiger Länge von Elementen eines vorerst noch nicht festgelegten Typs T; wir haben also einen **parametrisierten** oder **generischen** Typ, manchmal auch **Schema** oder **Schablone** (**template**) genannt, den wir später durch Festlegung von T noch spezialisieren können.

Die im folgenden verwendete Notation ist typisch, aber nicht genormt.

ADT Lists(T) IS
SORTS:
    list, T, boolean
OPERATIONS:
    newlist: $\rightarrow$ list
    cons: T $\times$ list $\rightarrow$ list
    head: list $\nrightarrow$ T
    tail: list $\nrightarrow$ list
    isempty: list $\rightarrow$ boolean
RULES:
    isempty(newlist) = true
    isempty(cons(x, l)) = false
    head(cons(x, l)) = x
    tail(cons(x, l)) = l
END Lists.

Die Operationen *newlist* und *cons* sind Konstruktoren, die anderen Operationen sind Projektoren. Grundterme werden mit *newlist* und *cons* aufgebaut, oder sind Grundterme der Sorten *boolean* oder T. Beispielsweise läßt sich die Sequenz <1, 2, 3, 4> als Term schreiben: cons(1, cons(2, cons(3, cons(4, newlist)))); und daß hier so viele Klammern stehen, wie öfters in LISP, ist kein Zufall.

Die Operationen *head* und *tail* sind partielle Funktionen (S. 38); manche Terme wie *tail(newlist)* lassen sich daher nicht auf Grundterme (S. 91) zurückführen. Sie bezeichnen demnach *keine Objekte;* in einer Implementierung ergeben sie keinen Sinn, und sie sollten möglichst als Fehler erkannt werden.

Abstrakte Datentypen werden durch *funktionale Sprachen* und auch manche *objektorientierte Sprachen* recht gut unterstützt; in *imperativen Sprachen* wie etwa Modula 2 ergeben sich oft rein technisch bedingte Einschränkungen, bei denen man sich geeignet behelfen muß. So sind meist nicht alle Typen als Funktionsresultate zugelassen; dann arbeitet man statt dessen mit Ergebnisparametern, muß aber daher Resultate zwischenspeichern.

84 Abstrakte Datentypen (3)

adt3

Bereits in der Termalgebra (S. 91) können wir mit Listen rechnen und auch zusätzliche Funktionen definieren, wie etwa zum Verketten von zwei Listen:

OPERATIONS:
append: list × list → list
RULES:
append(newlist, l) = l
append(cons(x, l1), l2) = cons(x, append(l1, l2))

Durch diese rekursive (S. 29) Definition ist die Funktion vollständig festgelegt; ihr Ergebnis läßt sich immer auf Grundterme (S. 91) reduzieren. Nach dem gleichen Schema lassen sich weitere nützliche Funktionen aufbauen.

Algebraische Spezifikationen sind schon lange bekannt. So sind die Grundterme (S. 91) des folgenden abstrakten Datentyps (S. 91)

ADT Nat IS
SORTS: Nat
OPERATIONS:
eins: → Nat
nach: Nat → Nat
RULES:
nach(x) = nach(y) ⇒ x = y
END Nat .

gerade die natürlichen Zahlen; dieses System hat darüberhinaus noch andere Modelle. Die Forderung, daß die Zahl 1 *nicht Nachfolger* ist, und auch das *Induktionsaxiom,* das wir in unserem Formalismus nicht formulieren können, ergeben sich gerade durch die Beschränkung auf die Grundterme. Die arithmetischen Operationen erhalten wir in gewohnter Weise durch

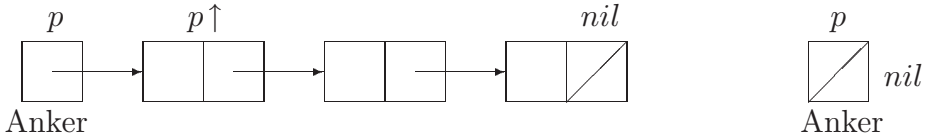
OPERATIONS:
add: Nat × Nat → Nat
mult: Nat × Nat → Nat
RULES:
add(x, eins) = nach(x)
add(x, nach(y)) = nach(add(x, y))
add(x, y) = add(y, x)
mult(x, eins) = x
mult(x, nach(y)) = add(mult(x, y), x)
mult(x, y) = mult(y, x)

85 Verkettete Listen: Aufbau

list1

Wenn wir den abstrakten Datentyp list (S. 92) in Modula 2 nachbilden wollen, so haben wir das Problem, daß die bisher besprochenen Mechanismen alle Objekte fester Länge beschreiben. Die Objekte des Datentyps *list* haben aber offensichtlich variablen Umfang, und daher kann man ihnen in einer konkreten Realisierung keine Speicherobjekte (S. 52) fester Länge zuordnen; es bleibt nur die Möglichkeit, sie durch ein **Geflecht** von **Knoten** zu realisieren, dessen Elemente auf der Halde (S. 84) liegen. Die Knoten müssen Verbunde (S. 60) sein, da sie neben den eigentlichen Nutzdaten vom Typ T auch noch die Strukturinformation über ihren Zusammenhang tragen müssen.

In unserem Fall ist dies jeweils ein Verweis auf den Nachfolger, und am Ende der Liste steht stattdessen ein NIL-Wert. Die Liste als Ganzes wird angesprochen über einen **Anker**, also eine Variable, deren Inhalt ein Verweis auf das erste Listenelement ist, oder NIL bei einer leeren Liste.



Wir definieren daher:

```
TYPE atom = ? (* irgend ein Typ *)
     liste = POINTER TO knoten;
     knoten = RECORD wert: atom;
                     nach: liste;
              END;
```

Aus historischen Gründen erlaubt Modula 2 nicht beliebige Typen als Funktionsresultate, daher ersetzen wir einige Funktionen durch Prozeduren mit Ergebnisparametern:

```
PROCEDURE newlist(VAR l: liste):
BEGIN l := NIL;
END newlist;

PROCEDURE cons(a: atom; l: liste; VAR r: liste);
(* r: = cons(a,l) *)
BEGIN NEW(r);
      r^.wert := a; r^.nach := l;
END cons;
```

```
PROCEDURE isempty(l: liste): BOOLEAN;
BEGIN RETURN l = NIL;
END isempty;

PROCEDURE head(l: liste; VAR a: atom);
BEGIN a := l^.wert;
END head;

PROCEDURE tail(l: liste; VAR r: liste);
BEGIN r := l^.nach;
END tail;
```

Diese Prozeduren sind extrem einfach; und hat man sich ihren Aufbau einmal eingeprägt, so wird man auf ihren weiteren Gebrauch verzichten und die betreffenden Anweisungen jeweils direkt hinschreiben.

Allerdings haben wir noch nicht berücksichtigt, daß *head* und *tail* partiell (S. 38) definiert sind und auf eine leere Liste nicht angewandt werden dürfen. Dies müssen wir noch korrigieren; und weil Modula 2, anders als Sprachen wie Ada und Java, keinen Standard-Fehlermechanismus anbietet, rufen wir hier eine Prozedur *error* auf, die noch geeignet zu definieren wäre. Eine Fortsetzung des Programmlaufs nach ihrem Aufruf macht natürlich wenig Sinn, da ja der Aufruf von *head* bzw. *tail* unzulässig war; es muß also ein *logischer Fehler* vorliegen, den man aufsuchen und beseitigen sollte.

```
PROCEDURE head(l: liste; VAR a: atom);
BEGIN IF l = NIL THEN error("access to empty list")
      ELSE a := l^.wert END;
END head;

PROCEDURE tail(l: liste; VAR r: liste);
BEGIN IF l = NIL THEN error("access to empty list")
      ELSE r := l^.nach END;
END tail;
```

Durch wiederholte Aufrufe von *newlist* und *cons* können wir nun Listen von beliebigem Inhalt aufbauen; für einige Standard-Anwendungen, die sich wieder als abstrakte Datentypen (S. 91) gut modellieren lassen, werden wir spezielle Mechanismen kennenlernen.

86 Opake Typen

opak

Abstrakte Datentypen (S. 91) sind eng verwandt mit Klassen (S. 16) in *objektorientierten Sprachen* und mit Bibliotheksmoduln (S. 28) in Modula 2. Der zu einem Bibliotheksmodul gehörige Kontrakt (S. 28) kann besonders präzise durch die algebraische Spezifikation (S. 91) angegeben werden, wenngleich diese normalerweise *funktional* formuliert ist; die notwendigen Änderungen sind meist offensichtlich.

```
DEFINITION MODULE Listen;
(* lineare Listen von Elementen des Typs atom *)
(* hierher kommt die algebraische Spezifikation *)

TYPE liste;

PROCEDURE newlist(VAR l: liste):

PROCEDURE isempty(l: liste): BOOLEAN;

PROCEDURE cons(a: atom; l: liste; VAR r: liste);
(* r: = cons(a,l) *)

PROCEDURE head(l: liste; VAR a: atom);

PROCEDURE tail(l: liste; VAR r: liste);

END Listen.
```

Ein Typenmodul (S. 28) exportiert einen Datentyp, und wenn dessen Realisierung nicht im Definitionsmodul angegeben ist (dann heißt dieser Typ ein **opaker Typ**), tritt ein praktisches Problem auf: um Wertargumente dieses Typs zu übergeben, oder auch um Speicherobjekte für deklarierte Variablen dieses Typs anzulegen, benötigt der Modula-Übersetzer Angaben über den Platzbedarf der durch den Typ beschriebenen Werte, und diese Information ist nicht verfügbar; sie entsteht ja erst bei der Übersetzung des Implementierungsmoduls. Daher gibt es eine *Implementierungseinschränkung*:

> Der Speicherbedarf für Werte eines opaken Typs darf nicht größer sein als der eines Pointerwertes (S. 84).

Diese Einschränkung stört nicht sehr, weil opake Typen tatsächlich meist als Pointertypen realisiert *sind,* und manche Implementierungen fordern dies sogar.

87 Verkettete Listen: Freigabe

list2

Wie einfach unsere angegebene Realisierung des abstrakten Datentyps *liste* (S. 92) in Modula 2 auch aussehen mag, so enthält sie doch ein paar ganz böse **Fallen**, die zu logischen Fehlern geradezu verleiten. Sie hängen damit zusammen, daß unser abstrakter Datentyp eine Menge von *Werten,* die Realisierung aber eine Menge von *Objekten* darstellt, und dieser Unterschied ist hier wesentlich.

Unser oben eingeführter Typ *liste* enthält tatsächlich *keine Listenwerte,* wie sie unsere algebraische Spezifikation (S. 92) beschreibt, sondern *Verweise* auf solche Werte, weil er ein Pointertyp (S. 84) ist. Deshalb haben sowohl Zuweisung (S. 41) wie Vergleich (S. 22) jetzt andere Auswirkungen als von früher her gewohnt, und wir müssen außerdem jetzt den durch unsere Geflechte (S. 94) belegten Speicher immer explizit freigeben. Dies kann etwa durch folgende rekursive Prozedur geschehen:

```
PROCEDURE delete(l: liste);
BEGIN IF l <> NIL
      THEN delete(l^.nach);
           DISPOSE(l);
      END;
END delete;
```

Vorsicht! Nach ihrem Aufruf zeigt der Inhalt von l *immer noch* auf die Halde, ist also ein dangling pointer (S. 84) geworden! Man sollte l anschließend besser mit NIL besetzen (das könnte eine Variante von *delete* auch direkt tun), wenn man nicht sofort ein neues Geflecht daranhängt. Es ist auch gute Praxis, jede neu deklarierte Variable eines *Pointertyps* explizit mit dem Wert NIL vorzubesetzen (in objektorientierten Sprachen geschieht das oft automatisch, in Modula 2 leider nicht).

Für die rekursive Prozedur *delete* gibt es auch eine iterative Variante, deren Wirkungsweise interessant ist (man sollte ein Beispiel durchspielen). Hier durchläuft der Wertparameter l als *lokale Hilfsvariable* die ganze Liste bis zum Ende, während ein zweiter Zeiger q immer ein Listenelement weit „nachhinkt“. Manchmal nennt man das **Schleppzeigertechnik**.

```
PROCEDURE delete(l: liste);
VAR q: liste;
BEGIN WHILE l <> NIL
      DO q := l; l := l^.nach;
         DISPOSE(q);
      END;
END delete;
```

88 Verkettete Listen: tiefe Kopie

list3

Wenn wir zwei Variable p und q vom Typ liste (S. 94) haben, wobei wir an p ein (irgendwann über newlist (S. 94) und cons (S. 94) aufgebautes) Geflecht angehängt haben, so erzeugt die Zuweisung q := p *keineswegs* ein *neues* Geflecht gleichen Inhalts, sondern einen *weiteren Zugriffspfad* auf dasselbe Geflecht, also ein Alias (S. 58). Eine Veränderung über einen der beiden Pfade p und q wird damit auch über den *anderen* Pfad sichtbar, da es sich ja um dasselbe Objekt handelt. Außerdem haben wir, falls q bereits einen legalen Pointer auf ein Haldenobjekt enthielt, diesen jetzt überschrieben, und womöglich ist dieses Objekt dadurch sogar *unerreichbar* geworden (sofern wir nicht daran gedacht haben, es freizugeben, wenn es nicht mehr benötigt wird).

Um solche Überraschungen zu vermeiden, ist es dringend zu empfehlen, sich den Effekt *jeder* Pointer-Manipulation durch eine *Handskizze* zu veranschaulichen!

Will man statt eines Alias (S. 97) wirklich ein *neues* Listenobjekt mit gleichem Inhalt wie ein gegebenes Objekt haben, das man dann unabhängig vom anderen verändern kann, so kann das nur über eine explizite **tiefe Kopie** geschehen, die einen neuen Zugriffspfad liefert:

```
PROCEDURE copy(l: liste; VAR p: liste);
BEGIN IF l = NIL THEN p := NIL
      ELSE NEW(p);
           p^.wert := l^.wert;
           copy(l^.nach, p^.nach);
      END
END copy;
```

Auch hier gibt es eine nicht ganz leicht zu verstehende iterative Version:

```
PROCEDURE copy(l: liste; VAR p: liste);
VAR q: liste;
BEGIN IF l = NIL THEN p := NIL;
      ELSE NEW(p); q := p;
           WHILE l <> NIL
           DO q^.wert := l^.wert; l := l^.nach;
              IF l = NIL THEN q^.nach := NIL
              ELSE NEW(q^.nach); q := q^.nach;
              END;
           END;
      END;
END copy;
```

89 Verkettete Listen: Wertvergleich

list4

Wenn wir zwei Variable p und q vom Typ liste (S. 94) haben, wobei wir an p und q je ein (irgendwann über newlist (S. 94) und cons (S. 94) aufgebautes) Geflecht angehängt haben, so liefert der Vergleich p = q nicht etwa die Information, ob die beiden Listen dieselben Datenwerte in derselben Reihenfolge enthalten, sondern stattdessen die Information, ob es sich um *dasselbe konkrete Objekt* handelt.

Die Abfrage auf gleiche Wertefolgen ist aufwendiger; sehen wir uns dazu erst ihre funktionale Definition (S. 91) an:

isequal(newlist, newlist) = true
isequal(newlist, cons(a,x)) = false
isequal(cons(a,x), newlist) = false
isequal(cons(a,x), cons(b,y)) = ((a = b) ∧ isequal(x,y))

Bei der Implementierung in Modula 2 müssen wir darauf achten, daß wir keine NIL-Werte dereferenzieren dürfen, und daß ein Objekt sich selbst gleich ist.

```
PROCEDURE isequal(p, q: liste): BOOLEAN;
BEGIN IF p = q THEN RETURN TRUE
      ELSIF p = NIL THEN RETURN FALSE
      ELSIF q = NIL THEN RETURN FALSE
      ELSE RETURN (p^.wert = q^.wert)
                  AND isequal(p^.nach, q^.nach)
      END;
END isequal;
```

Dies läßt sich auch iterativ formulieren, wobei die beiden Wertparameter p und q die beiden Listen soweit wie nötig durchlaufen; hier haben wir ein Beispiel dafür, daß eine LOOP-Schleife (S. 48) nicht mit EXIT, sondern mit RETURN verlassen wird.

```
PROCEDURE isequal(p, q: liste): BOOLEAN;
BEGIN LOOP IF p = q THEN RETURN TRUE
           ELSIF p = NIL THEN RETURN FALSE
           ELSIF q = NIL THEN RETURN FALSE
           ELSIF p^.wert <> q^.wert THEN RETURN FALSE
           ELSE p := p^.nach; q := q^.nach;
           END;
      END;
END isequal;
```

90 Verkettete Listen: Modifikation (1)

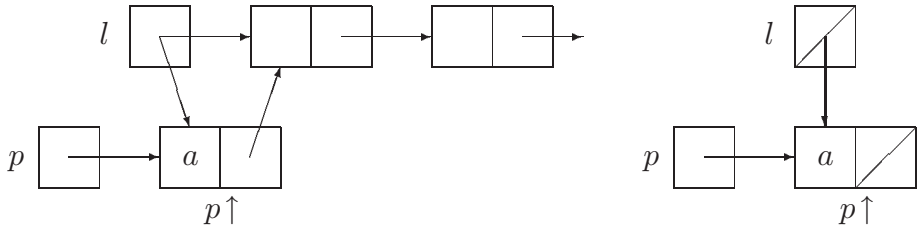
list5

Wenn wir lineare Listen (S. 94) zur Ablage einer variablen Anzahl von Nutzdaten verwenden wollen, besteht oft der Wunsch, die Liste selbst zu modifizieren, statt eine geänderte Kopie abzulegen. Wir wollen uns über die dazu vorhandenen Möglichkeiten und ihren Aufwand einen Überblick verschaffen.

Die einfachsten Operationen sind das Einfügen und das Entfernen von Datenelementen *am Listenanfang.* Sie haben offensichtlich konstanten Aufwand (S. 85) und lassen sich von den schon bekannten Operationen `cons` und `tail` leicht ableiten, wobei wir jetzt auf dem Original der Liste arbeiten; daher ist der Parameter `l` hier ein **Durchgangsparameter**:

```
PROCEDURE einfuege(a: atom; VAR l: liste);
VAR p: liste;
BEGIN NEW(p);
      p^.wert := a; p^.nach := l; l := p;
END einfuege;

PROCEDURE entferne(VAR a: atom; VAR l: liste);
VAR p: liste;
BEGIN p := l; a := p^.wert; l := p^.nach;
      DISPOSE(p);
END entferne;
```

Wie immer bei Pointer-Manipulationen veranschaulichen wir uns die Verhältnisse durch eine Skizze, auch für den Fall einer leeren Liste. Daß `entferne` nicht auf eine leere Liste angewendet werden darf, muß der Aufrufer sicherstellen; andernfalls enthält sein Programm einen *logischen* Fehler. Wir prüfen darauf deshalb hier nicht zur Laufzeit.



Bemerkenswert ist bei Verwendung dieser Routinen, daß Datenwerte, die nacheinander eingetragen werden, beim Abholen in *umgekehrter* Reihenfolge wieder erscheinen (**LIFO**-Prinzip, „Last-In-First-Out“). Ebenso verhält sich ein *Stapel* von losen Seiten, und dieses Verhalten kommt so häufig in Anwendungen vor, daß es sich lohnt, es (später) durch einen eigenen abstrakten Datentyp stack (S. 103) zu modellieren.

91 Verkettete Listen: Modifikation (2)

list6

Eine Variante der Routinen zum Einfügen oder Löschen am Listenanfang (S. 100) wird gelegentlich verwendet, um sich lokal eine **Freiliste** an Listenelementen zu halten für den Fall, daß DISPOSE nicht richtig funktioniert; bei manchen Implementierungen ist (oder war?) das leider so.

```
VAR Frei: liste; (* mit NIL initialisieren! *)

PROCEDURE gibKnoten(VAR p: liste);
BEGIN IF Frei = NIL THEN NEW(p)
      ELSE p := Frei; Frei := p^.nach;
      END;
END gibKnoten;

PROCEDURE nimmKnoten(VAR p: liste);
BEGIN p^.nach := Frei; Frei := p;
END nimmKnoten;
```

Außer am Listenanfang kommen oft auch Modifikationen *im Inneren* einer Liste vor, an einer etwa durch einen Suchvorgang gefundenen Stelle. Dabei kommt es im Grunde gar nicht so sehr auf die Liste als konkrete *Speicherstruktur* an, sondern vielmehr auf die lineare Folge der *Datenwerte.*

Das Einfügen oder Entfernen eines Datenelements *hinter* einem über einen Zeiger q zugänglichen **Einstiegs-Element** geht fast genauso wie am Listenanfang (S. 100); der Nachfolgerverweis im Einstiegselement wirkt als Anker für den Rest der Liste:

```
PROCEDURE einfuegenach(a: atom; q: liste);
VAR p: liste;
BEGIN NEW(p);
      p^.wert := a; p^.nach := q^.nach; q^.nach := p;
END einfuegenach;

PROCEDURE entfernenach(VAR a: atom; q: liste);
VAR p: liste;
BEGIN p := q^.nach; a := p^.wert; q^.nach := p^.nach;
      DISPOSE(p);
END entfernenach;
```

Der Zeiger q braucht hier kein Referenzparameter zu sein, da er nicht verändert wird (es schadet aber auch nichts).

92 Verkettete Listen: Modifikation (3)

list7

Wenn wir *vor* einem Einstiegselement (S. 101) auf analoge Weise ein Datenelement eintragen oder aber das Einstiegselement entfernen wollen, so brauchen wir dazu Zugriff auf den Vorgängerknoten des Einstiegselements, da sich dessen Nachfolger ändert; und diesen Zugriff können wir nicht billig bekommen, weil wir vom Listenanfang her suchen müssen. Es gibt allerdings einen *Kunstgriff,* der die Tatsache ausnutzt, daß es uns auf die *Datenwerte* und nicht auf die *Listenknoten* ankommt; leider funktioniert er beim Löschen des letzten Listenelements nicht.

Beim Einfügen tragen wir das Einstiegselement als seinen eigenen *Nachfolger* ein und überschreiben es dann mit dem neuen Datenwert:

```
PROCEDURE einfuegevor(a: atom; VAR q: liste);
VAR p: liste;
BEGIN NEW(p);
      p^.wert := q^.wert; p^.nach := q^.nach;
      (* oder besser: p^ := q^ *)
      q^.nach := p; q^.wert := a; q := p;
END einfuegevor;
```

Die letzte Anweisung bewirkt, daß der Zeiger `q` wieder auf den als Einstieg gebrauchten *Datenwert* zeigt. Jetzt *muß* `q` Referenzparameter sein, weil das Einstiegselement seinen Ort ändert!

Beim Entfernen überschreiben wir das Einstiegselement mit seinem Nachfolger, *sofern* es einen solchen gibt, und löschen diesen.

```
PROCEDURE entfernebei(VAR a: atom; q: liste);
VAR p: liste;
BEGIN p := q^.nach;
      q^.wert := p^.wert; q^.nach := p^.nach;
      (* oder besser: q^ := p^ *)
      DISPOSE(p);
END entfernebei;
```

Der Verweis `q` ist anschließend ein dangling pointer (S. 84); und das zu Recht, weil wir ja das darüber vorher zugängliche Einstiegselement gelöscht haben.

Gibt es keinen Nachfolger, wollen wir also das letzte Datenelemen der Liste löschen, so bleibt uns nichts anderes übrig, als vom Listenanfang her zu suchen, und dann haben wir (leider) linearen Aufwand (S. 85) bezüglich der Listenlänge.

93 ADT Stapel = Stack

stack1

In den Anwendungen tritt recht häufig die Situation auf, daß eine von vorneherein nicht begrenzte Zahl von Datenelementen so aufbewahrt werden muß, daß jeweils das zuletzt verwendete Datenelement direkt zugreifbar ist, während die anderen in umgekehrter Reihenfolge der Eintragung beim Abruf wieder erscheinen. Dieses *Last-In-First-Out*-Verhalten (**LIFO**) ist uns bereits bei der Formularmaschine (S. 68) zur Abarbeitung von Syntaxdiagrammen (S. 67) begegnet, und es ist generell typisch für Probleme, die bei der Analyse und Verarbeitung von kontextfreien (S. 64) Sprachen (insbesondere Programmiersprachen (S. 13)) auftreten, oder die sich (bei hinreichend tiefer theoretischer Einsicht) über solche Sprachen beschreiben lassen.

Eine Datenstruktur, die sich so verhält, nennt man einen **Stapel**, auch **Keller** oder **Stack**, und sie ist leicht algebraisch zu modellieren:

ADT Stacks(T) IS
SORTS:
stack, T, boolean
OPERATIONS:
newstack: $\rightarrow$ stack
push: T $\times$ stack $\rightarrow$ stack
top: stack $\nrightarrow$ T
pop: stack $\nrightarrow$ stack
isempty: stack $\rightarrow$ boolean
RULES:
isempty(newstack) = true
isempty(push(x, l)) = false
top(push(x, l)) = x
pop(push(x, l)) = l
END Stacks.

Dies stimmt bis auf die Bezeichnung der Funktionen **push**, **pop** und **top** mit den bereits besprochenen Listen (S. 92) überein, jedoch arbeiten wir hier nicht funktional mit Stack-Werten, sondern der Objekt-Aspekt steht im Vordergrund. Die dafür angegebenen Realisierungen (S. 100) können wir mit den notwendigen Änderungen direkt übernehmen.


Benötigt man nur einen einzigen Stapel, so kann man ihn auch in einem Array (S. 59) fester Länge ablegen, muß aber jetzt die aktuelle Höhe des Stapels mitführen (dies geschieht automatisch) und darauf achten, daß der Stapel nicht überläuft. Die Realisierung ist eine naheliegende Übungsaufgabe.

94 ADT Schlange = Queue

queue1

Eine Datenstruktur zur Zwischenablage von Datenwerten, bei der die Werte in der gleichen Reihenfolge entnommen werden, in der sie eingetragen wurden (First-In-First-Out, **FIFO**), nennen wir eine (Warte-)**Schlange** oder **Queue**. Eine Schlange hat einen **Schwanz**, an den wir mittels der Operation **enqueue** Datenelemente anhängen, und einen **Kopf**, wo wir das aktuell erste Element *first(q)* mit der Operation **dequeue** abholen können; übrig bleibt *rest(q).*

Die algebraische Modellierung ist etwas mühsam und benötigt die Hilfsfunktionen *first* und *rest* vor allem wegen einiger Unvollkommenheiten der üblichen mathematischen Notation.

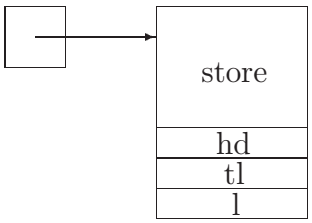
ADT Queues(T) IS
SORTS:
queue, T, boolean, cardinal
OPERATIONS:
newqueue: $\rightarrow$ queue
enqueue: T $\times$ queue $\rightarrow$ queue
isempty: queue $\rightarrow$ boolean
length: queue $\nrightarrow$ cardinal
dequeue: queue $\nrightarrow$ T $\times$ queue
first: queue $\nrightarrow$ T
rest: queue $\nrightarrow$ queue
RULES:
isempty(newqueue) = true
isempty(enqueue(x, q)) = false
length(newqueue) = 0
length(enqueue(x, q)) = length(q) + 1
dequeue(q) = <first(q), rest(q)>
first(enqueue(x, q)) = IF isempty(q) THEN x ELSE first(q)
rest(enqueue(x, q)) = IF isempty(q) THEN newqueue
ELSE enqueue(x, rest(q))
END Queues.

Wir besprechen einige Realisierungen im Einzelnen; man macht dabei leicht Fehler, weil es nicht nur den Sonderfall der *leeren Schlange* zu berücksichtigen gilt, sondern auch den Fall einer *einelementigen Schlange,* bei der Kopf und Schwanz identisch sind.

95 Schlangen (2)

queue2

Ist die Maximallänge einer Schlange (S. 104) von vorne herein beschränkt, so kann man sie in einem Array (S. 59) fester Länge unterbringen, den man zu einem logischen **Ringpuffer** schließt. Da im Allgemeinen nur ein Teil des Array belegt sein wird, müssen wir uns die Position von Kopf und Schwanz merken, und wir rechnen auch die aktuelle Länge mit.



Zur Einübung formulieren wir unsre Lösung mittels eines opaken (S. 96) Typs *queue*, den wir als Pointer (S. 84) auf einen Verbund unserer Beschreibungsgrößen und Datenwerte realisieren. Die einzelnen Felder des Verbundes sprechen wir über eine WITH-Anweisung (S. 60) an; das ist bequem.

```
TYPE queue = POINTER TO qrecord;
     index = 1 .. max; (* maximaler Umfang *)
     storeTyp = ARRAY [index] OF atom;
     qrecord = RECORD store: StoreTyp;
                      hd, tl: index;
                      l: CARDINAL;
               END;

PROCEDURE newqueue(VAR q: queue);
BEGIN NEW(q);
      WITH q^ DO l := 0; hd := 1; tl := max;
      END;
END newqueue;
```

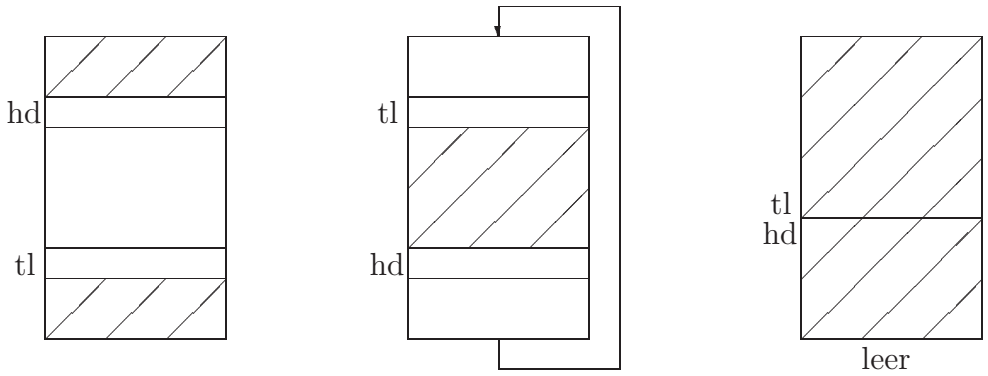
Die hier gesetzten Anfangswerte für hd und tl ergeben sich beim Austragen des einzigen Elements einer einelementigen Schlange, die ganz am Ende des Array liegt; andere Besetzungen wären ebenso möglich. Eine Schlange ist leer genau dann, wenn sie die Länge Null hat.

Achtung! Eine Schlange der *maximalen* Länge l=max sieht bei dieser Implementierung bis auf die Längenangabe *genauso* wie eine *leere* Schlange aus! Daher müssen wir hier unbedingt die Länge mitrechnen, selbst wenn sie der Anwender nicht brauchen sollte.

96 Schlangen (3)

queue3

Zum Eintragen eines neuen Datenelements am Schwanz (S. 104) der Schlange (S. 105) hängen wir es hinten an den belegten Teil des Array an; und ebenso holen wir uns vorne das aktuelle Kopf-Element ab. Würden wir dabei über das Ende des Array hinauslaufen, so kommen wir von vorne wieder herein:




```
PROCEDURE enqueue(x: atom; VAR q: queue);
BEGIN WITH q^
      DO IF tl = max THEN tl := 1
         ELSE tl := tl + 1
         END;
         store[tl] := x; l := l + 1;
      END;
END enqueue;

PROCEDURE dequeue(VAR y: atom; VAR q: queue);
BEGIN WITH q^
      DO y := store[hd]; l := l - 1;
         IF hd = max THEN hd := 1
         ELSE hd := hd + 1
         END;
      END;
END dequeue;
```

Statt der IF-Abfrage hätten wir auch mit der MOD-Funktion arbeiten können, und hätten uns dabei vermutlich erst einmal verrechnet. Die hier gegebene Lösung ist so primitiv, daß ihre Korrektheit unmittelbar einsichtig ist; daher ist sie unbedingt vorzuziehen (schneller ist sie außerdem auch noch!)

97 Schlangen (4)

qhead1

Für Schlangen (S. 104) von prinzipiell unbekannter Länge bietet sich eine Realisierung über verkettete Listen (S. 100) an. Wir wissen bereits, daß wir am Ende einer Liste leicht Elemente anhängen, und am Anfang Elemente löschen können. Zur Beschreibung einer Schlange brauchen wir demnach Informationen darüber, wo Kopf und Schwanz liegen, und vielleicht auch noch die Länge, falls sie oft benötigt wird. Diese Beschreibungsgrößen packen wir in einen Beschreibungsblock, den wir über einen Pointer adressieren. Für die Listenelemente (S. 94) selbst können wir es bei den schon früher verwendeten Datentypen belassen:

```
TYPE queue = POINTER TO qrecord;
   qrecord = RECORD l: CARDINAL;
                    hd, tl: liste;
             END;
```

Eine leere Schlange hat einen nichtleeren Beschreibungsblock, aber die daran hängende Liste ist leer; wir setzen (redundant) hd = tl = NIL und l = 0. Wird eine Schlange nicht mehr gebraucht, so müssen wir nicht nur die Listenelemente, sondern auch den Beschreibungskopf freigeben!

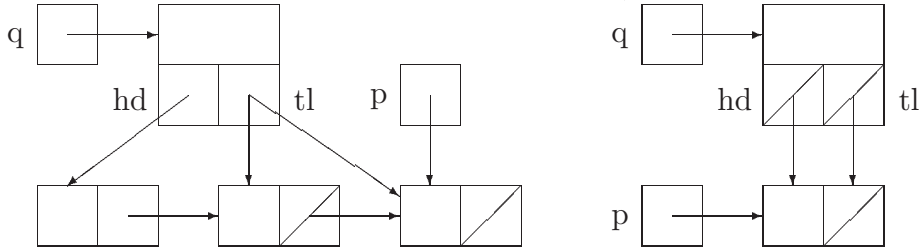
```
PROCEDURE newqueue(VAR q: queue);
BEGIN NEW(q);
      WITH q^ DO hd := NIL; tl := NIL; l := 0;
      END;
END newqueue;

PROCEDURE isempty(q: queue): BOOLEAN;
BEGIN RETURN q^.hd = NIL;
END isempty;

PROCEDURE killqueue(VAR q: queue);
BEGIN delete(q^.hd);
      DISPOSE(q);
END killqueue;
```

Bei einer *einelementigen* Schlange ist der Kopf gleichzeitig das letzte Element; dies gibt beim Eintragen und Aushängen Sonderfälle. Das versuchte Aushängen aus einer leeren Schlange fangen wir nicht dynamisch ab; der Benutzer kann (und sollte) es durch Abfragen selbst vermeiden!

98 Schlangen (5)

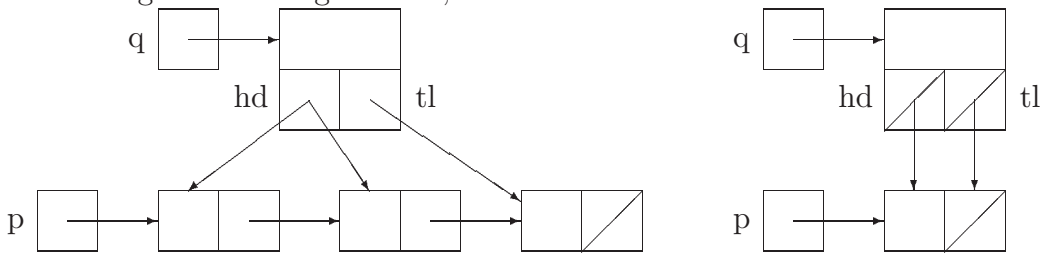
qhead2

Beim Anhängen an eine Schlange erzeugen wir ein neues Listenelement. Bei einer nichtleeren Schlange hängen wir es hinten an; bei einer leeren Schlange wird es der Kopf. In beiden Fällen ist es das letzte Listenelement, also der Schwanz.




```
PROCEDURE enqueue(x: atom; VAR q: queue);
VAR p: liste;
BEGIN WITH q^
      DO NEW(p); p^.wert := x; p^.nach := NIL;
         IF hd = NIL THEN hd := p ELSE tl^.nach := p
         END;
         tl := p; l := l + 1;
      END;
END enqueue;
```

Beim Austragen aus einer (nichtleeren) Schlange hängen wir das Kopfelement ab; ist die Schlange dann leer geworden, so hat sie auch keinen Schwanz mehr!

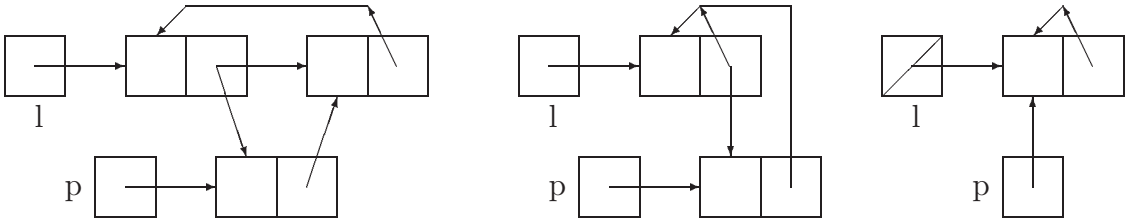



```
PROCEDURE dequeue(VAR y: atom; VAR q: queue);
VAR p: liste;
BEGIN WITH q^
      DO p := hd; y := p^.wert; hd := p^.nach;
         IF hd = NIL THEN tl := NIL
         END;
         DISPOSE(p); l := l - 1;
      END;
END dequeue;
```

99 Ringlisten

ring1

Bisher haben wir als **Endemarkierung** einer verketteten Liste immer einen Vorwärtsverweis mit dem Wert NIL verwendet (**gestreckte Liste**). Manchmal ist es vorteilhaft, stattdessen dort einen Verweis auf das erste Element der Liste einzutragen, wenn dieses über einen Anker (S. 94) zugänglich ist; so erhalten wir eine **Ringliste**. Sie kann ebensogut zur Ablage von Datenwerten verwendet werden, aber man muß auf Sonderfälle beim Übergang zwischen einer einelementigen und einer leeren Liste achten, und das Kopfelement muß seinen Platz behalten, weil ein Zeiger vom Ende her darauf verweist:




```
PROCEDURE einfuege(a: atom; VAR l: liste);
VAR p: liste;
BEGIN NEW(p);
      IF l = NIL THEN l := p ELSE p^ := l^
      END;
      l^.nach := p; l^.wert := a;
END einfuege;
```

Beim Entfernen geht alles sinngemäß rückwärts:

```
PROCEDURE entferne(VAR a: atom; VAR l: liste);
VAR p: liste;
BEGIN a := l^.wert; p := l^.nach;
      IF l = p THEN l := NIL ELSE l^ := p^
      END;
      DISPOSE(p);
END entferne;
```

Als Ausgleich für diese Komplikationen kommt jetzt eine neue Möglichkeit hinzu: die *Datenelemente* sind jetzt ein **logischer Ring**, und indem wir den Anker weiter wandern lassen, können wir ihn bei einem beliebigen Element beginnen lassen.

Zum Löschen einer Ringliste hängen wir am einfachsten alle Elemente aus.

```
PROCEDURE delete(l: liste);
VAR p: liste;
BEGIN WHILE l <> NIL
      DO p := l^.nach;
         IF l = p THEN l := NIL ELSE l^.nach := p^.nach
         END;
         DISPOSE(p);
      END;
END delete;
```

100 Ringlisten (2)

ring2

Mittels Ringlisten (S. 109) lassen sich Schlangen (S. 104) überraschend elegant implementieren, wenn man den Anker (S. 94) auf das logisch letzte Element verweisen läßt; das folgende Element ist dann der Kopf, und man kann vom Anker aus effizient sowohl den Kopf entfernen, wie auch durch Eintragen dahinter und Weiterschalten des Ankers ein neues logisch letztes Element eintragen. Wenn man die Länge nicht explizit benötigt, kommt man mit dem Zeigertyp *liste* zur Repräsentation der Schlangen aus. Die Routinen (S. 109) ändern sich nur geringfügig:

```
TYPE queue = liste;

PROCEDURE enqueue(x: atom; VAR q: queue);
VAR p: queue;
BEGIN NEW(p);
      IF q = NIL THEN q := p ELSE p^.nach := q^.nach
      END;
      q^.nach := p; q := p; p^.wert := x;
END enqueue;

PROCEDURE dequeue(VAR x: atom; VAR q: queue);
VAR p: queue;
BEGIN p := q^.nach; x := p^.wert;
      IF q = p THEN q := NIL ELSE q^.nach := p^.nach
      END;
      DISPOSE(p);
END dequeue;
```

101 Doppelschlange = Deque

deque

Eine **Doppelschlange** (**Deque**, **Deck**) ist eine lineare Datenstruktur, bei der an *beiden* Enden Datenelemente eingefügt oder abgehängt werden können. Eine Doppelschlange kann also auch wahlweise als Stapel (S. 103) oder Schlange (S. 104) verwendet werden, und dies in beiden Richtungen. Ein konkretes Analogon dazu ist etwa ein Güterzug, bei dem an beiden Enden rangiert werden kann.

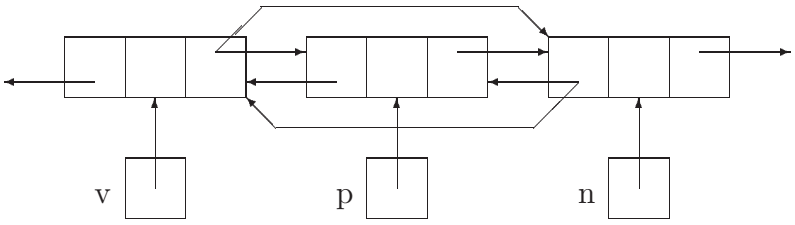
Damit die Zugriffsoperationen effizient (mit konstantem (S. 85) Aufwand) realisiert werden können, sollte es möglich sein, von beiden Enden her in die darunterliegende Datenstruktur hineinzulaufen, und dies gelingt am besten über eine **doppelt verkettete** Liste, in der jeder Knoten je einen Verweis auf den Nachfolger und den Vorgänger trägt.

Analog zu früher können wir jetzt definieren:

```
TYPE liste = POINTER TO Knoten;

TYPE Knoten = RECORD wert: atom;
                     vor, nach: liste;
              END;
```

Diese Datenstruktur ist auch für sich gesehen interessant, weil Einfügen und Löschen jetzt auch im Inneren einer Liste einfach möglich ist. Wenn wir die beiden Verkettungen einzeln für sich betrachten, so erhalten wir jeweils wieder eine verkettete Liste, und wir können jeweils noch vorgeben, ob dies eine gestreckte (S. 94) Liste oder eine Ringliste (S. 109) sein soll.

Als Beispiel geben wir eine Operation für das Entfernen eines Knotens aus einer doppelt verketteten Ringliste an:




```
PROCEDURE entferne(VAR y: atom; VAR p: liste);
VAR n, v: liste;
BEGIN y := p^.wert; v := p^.vor; n := p^.nach;
      v^.nach := n; n^.vor := v;
      DISPOSE(p);
END entferne;
```

Der Anwender muß darauf achten, daß er noch einen weiteren Zeiger in die Liste behält, sonst wird sie unzugänglich! Einfügen vor oder hinter einem gegebenen Element geht ganz analog, und ebenso einfach.

Wir haben hier zwei Hilfszeiger v und n eingeführt, um das Programm übersichtlich zu halten. Statt der zweiten Zeile im Prozedurrumpf hätten wir auch schreiben können:

```
p^.vor^.nach := p^.nach; p^.nach^.vor := p^.vor;
```

Dann brauchen wir die Hilfszeiger nicht, und die Routine wird schneller, aber weniger leicht zu lesen. Ein wirklich guter, moderner Compiler macht die Verbesserung automatisch. (Sie kann erheblich sein; auf einem TR4-Prozessor aus den 60-er Jahren braucht das Entfernen nur 6 Maschinenbefehle!)

Natürlich ist für die neu gewonnene Flexibilität ein Preis zu zahlen, gemäß der Regel, die gelegentlich **TAANSTAAFL-Law** genannt wird:

> There Ain't Absolutely No Such Thing As A Free Lunch

Hier ist der notwendige Zusatzaufwand der Platz, den der zweite Verweis in jedem Knoten belegt.

Die Struktur der doppelt verketteten Liste enthält erhebliche **Redundanz** (im Grunde überflüssige Information): für jedes über einen Zeiger p zugängliche Element einer doppelt verketteten Liste muß gelten

```
p^.vor^.nach = p    und    p^.nach^.vor = p
```

Diese Redundanz läßt sich ausnutzen: ist etwa in einer doppelt verketteten Ringliste ein *einziger* Zeiger zerstört (aber nur so weit, daß ein Zugriff über ihn nicht auf einen Laufzeitfehler führt!), so kann dieser Fehler nicht nur *erkannt*, sondern sogar *korrigiert* werden. Allerdings bleibt dann noch die womöglich schwierige Aufgabe, herauszufinden, wie es zu dem Fehler überhaupt kommen konnte!

Historische Abschweifung

Wir haben in einem größeren Implementierungsprojekt Anfangs der 70-er Jahre die doppelt verketteten Listen als Grundlage für alle anderen vorkommenden Listenstrukturen verwendet (damals waren Kenntnisse über Datenstrukturen noch nicht in dem Maße Allgemeingut, wie sie es heute sind oder zumindest sein sollten), und das hat uns öfters geholfen, Fehler beim Zugriff durch mehrere parallel ablaufende Prozesse auf gemeinsame Datenstrukturen zu finden. Wie solche parallelen Zugriffe korrekt zu synchronisieren sind, haben wir dadurch besser verstanden.

102 Bäume und Wälder

tree1

Zur Ablage von **hierarchisch** strukturierten Informationen verwendet man häufig **Baumstruktur**en. Wir beginnen mit zwei *ernst gemeinten* Definitionen:

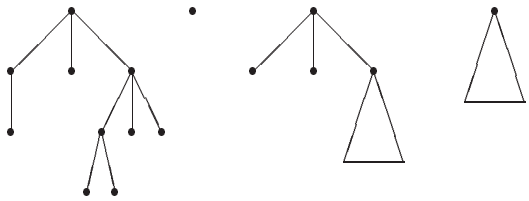
- Ein **Wald** ist eine Menge von *Bäumen.*
- Ein **Baum** besteht aus einem Knoten, genannt **Wurzel**, und einem *Wald* der **Unterbäume**.

Nach dieser Definition kann ein *Wald* leer sein, ein *Baum* dagegen nicht, da er zumindest die Wurzel enthält. Wir definieren weiter:

- Ein **Teilbaum** ist eine Teilmenge eines Baumes, die selbst ein Baum ist.
- Ein **Blatt** ist ein Teilbaum, der keine Unterbäume mehr enthält.

Es ist eine bequeme Tradition, im Zusammenhang mit Bäumen die Analogie zu einer (merkwürdigerweise meist rein männlichen) Familie auszunutzen, und von **Sohn**, **Vater**, **Bruder** eines Knotens zu sprechen (Großväter, Onkel und Neffen kommen auch gelegentlich vor).

Wir veranschaulichen uns Bäume meist als verzweigte Strukturen, oder aber einfach als Dreiecke mit der Wurzel oben(!), wenn es uns auf die innere Struktur nicht ankommt.



Auch in der Informatik wachsen die Bäume nicht in den Himmel; daß man gewöhnlich die Wurzel oben zeichnet, liegt wohl daran, daß man meist mit ihr beginnt und die Zeichenfläche von oben nach unten füllt; die Darstellung mit links liegender Wurzel trifft man gelegentlich an, aber eher selten.

Es liegt nahe, einen Wald als eine verkettete Liste (S. 94) der Bäume zu realisieren; dann sind, weil die Unterbäume eines Baumes ein Wald sind, deren Wurzeln auch miteinander verkettet. Als Anker (S. 94) für diese Liste verwenden wir einen Verweis vom gemeinsamen *Vater* dieser Wurzelknoten auf den Listenkopf, den *ältesten Sohn,* und die Wurzelverkettung der Unterbäume geht von ihm auf die *jüngeren Brüder.* Damit sind von der Wurzel des *ältesten Baumes* im Wald alle Knoten je über eine Folge von Zeigern erreichbar.

Wir kommen hier, wie auch schon bei den doppelt verketteten (S. 111) Listen, mit zwei Verweisen je Knoten aus, jedoch ergibt sich eine völlig andere Struktur.

103 Binäre Bäume

btree1

In der Praxis ebenso wichtig wie Wälder (S. 113) und allgemeine Bäume (S. 113), wenn nicht sogar wichtiger, sind *Binärbäume.* Sie lassen sich algebraisch einfach spezifizieren:

ADT BTrees(T) IS
SORTS:
    btree, T, boolean
OPERATIONS:
    newtree: → btree
    node: T × btree × btree → btree
    isempty: btree → boolean
    root: btree ↛ T
    left: btree ↛ btree
    right: btree ↛ btree
RULES:
    isempty(newtree) = true
    isempty(node(x, l, r)) = false
    root(node(x, l, r)) = x
    left(node(x, l, r)) = l
    right(node(x, l, r)) = r
END BTrees.

Ein **Binärbaum** ist *kein Baum!* Die wesentlichen Unterschiede sind:

- ein Binärbaum kann *leer* sein,
- wenn nur ein Unterbaum vorhanden ist, so macht es einen Unterschied, ob er der *linke* oder der *rechte* ist!

Die Darstellung von Binärbäumen als Verweise auf Knoten mit einem Datenfeld und zwei Zeigern ist naheliegend:

```
TYPE btree = POINTER TO brecord;
   brecord = RECORD wert: atom;
               left, right: btree;
             END;
```

sie liefert uns auch sofort eine Darstellung für einen Wald (S. 113) durch einen **äquivalent**en Binärbaum. Für jeden (allgemeinen) Baum des Waldes gilt: in den linken Unterbaum des zugeordneten Binärbaumes kommen die Nachkommen, in den rechten Unterbaum die jüngeren Brüder samt ihren Nachkommen.

104 Prozedurtypen; Durchwanderung

travers

Wenn man ein Geflecht (S. 94) zur Ablage von Daten verwendet, tritt gelegentlich das Problem auf, eine vorgegebene *Operation* auf jedes Datenelement anzuwenden, also das Geflecht geeignet zu **durchwandern**. Dabei werden wir einen Knoten (S. 94) womöglich mehrmals **besuchen**, aber nur *genau einmal* **bearbeiten**. Als Operationen kommen beispielsweise in Frage: bilde die Summe, suche das kleinste Element, gib das Element in externer Darstellung lesbar aus; es gibt noch mehr Anwendungen.

Für lineare Listenstrukturen (S. 94) ist das einfach zu lösen: man läßt einen Zeiger über die Liste laufen, und man wendet auf jedes gefundene Element die gewünschte Operation an. Für kompliziertere Strukturen muß man sich jeweils eine Strategie zur **Durchwanderung** oder **Traversierung** überlegen, und davon kann es mehrere geben. Dieselbe *Strategie* will man womöglich für mehrere *Operationen* verwenden, und diese Operationen haben gemeinsam, daß sie jeweils auf Elemente eines gegebenen Typs angewendet werden.

Modula 2 bietet (fast ohne Konkurrenz, neben ALGOL 68 und Ansätzen in Pascal) dafür ein eigenes Konzept an: **Prozedurtypen**. Ein Prozedurtyp umfaßt eine Klasse von Prozeduren mit gleicher Funktionalität (S. 91), aber womöglich verschiedener Auswirkung. Beispiel: der Typ

```
TYPE Bearbeiter = PROCEDURE(btree);
```

hat die Bedeutung: tu irgendwas mit einem Wert-Argument vom Typ *btree*. Hat nun eine Durchwanderungsstrategie ein Argument vom Typ *Bearbeiter,* so kann man dafür im konkreten Anwendungsfall eine beliebige Operation passender Funktionalität einsetzen. Auch Variablen von einem Prozedurtyp sind möglich und können mit konkreten Funktionen passenden Typs besetzt werden.


Prozedur- bzw. Funktionstypen treten auch in anderem Zusammenhang auf: bei der Bildung des bestimmten Integrals einer beliebigen integrierbaren reellen Funktion f über einem festen Intervall [a,b]

$$I(f) = \int_a^b f(t)\,dt$$

mit einem geeigneten numerischen Integrationsverfahren

```
PROCEDURE Integriere(f: realfct; a, b: real): real;
```

ist der Integrand vom Typ

```
TYPE realfct = PROCEDURE(real): real;
```

105 Durchwanderung: Wald und Binärbäume

btree2

Bei der Durchwanderung (S. 115) eines Binärbaumes (S. 114) müssen wir, um alle Knoten zu finden, die Wurzel eines Teilbaumes in der Regel mehrmals besuchen (S. 115), und wir können noch auswählen, wann wir sie dabei bearbeiten (S. 115) wollen. Dabei sind mehrere Strategien gebräuchlich, die etwa gleich wichtig sind:

- die **Präfix**-Strategie: die Wurzel wird *vor* den Unterbäumen bearbeitet,
- die **Infix**-Strategie: die Wurzel wird *zwischen* den Unterbäumen bearbeitet,
- die **Postfix**-Strategie: die Wurzel wird *nach* den Unterbäumen bearbeitet.


Man könnte noch nach der Reihenfolge der Unterbäume differenzieren; meist lohnt sich das nicht.

Die drei Strategien sind unmittelbar auszuprogrammieren:

```
PROCEDURE prefix(b: btree; p: Bearbeiter);
BEGIN IF b <> NIL
      THEN p(b); (* bearbeite die Wurzel *)
           prefix(b^.left, p);  (* besuche l. UB *)
           prefix(b^.right, p); (* besuche r. UB *)
      END;
END prefix;
```

Dabei ist der Parameter p eine beliebige passende Bearbeitungs-Prozedur (S. 115). Die Strategien *infix* und *postfix* gehen sinngemäß ebenso.

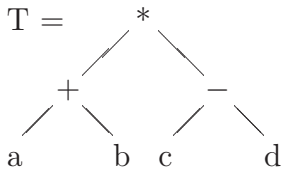
Auch für allgemeine Bäume und Wälder lassen sich entsprechende Durchwanderungsstrategien definieren, (eine Infix-Strategie macht allerdings keinen Sinn), und erfreulicherweise besteht eine enge Beziehung zu den Durchwanderungen des äquivalenten (S. 114) Binärbaumes:

- die *Präfix-Durchwanderung* des Binärbaumes bearbeitet jeden Knoten *vor* seinen Nachkommen und *vor* seinen jüngeren Geschwistern, ist also auch eine *Präfix-Strategie,*
- die *Infix-Durchwanderung* des Binärbaumes bearbeitet jeden Knoten *nach* seinen Nachkommen und *vor* seinen jüngeren Geschwistern, ist also hier eine *Postfix-Strategie,*
- die *Postfix-Durchwanderung* des Binärbaumes bearbeitet jeden Knoten *nach* seinen Nachkommen und *nach* seinen jüngeren Geschwistern, ist also ebenfalls eine *Postfix-Strategie,* allerdings mit anderer Reihenfolge der Geschwister.

106 Durchwanderung: Binärbäume

btree3

Für die Durchwanderung (S. 116) eines Binärbaumes (S. 114) betrachten wir ein *Beispiel:* ein arithmetischer Ausdruck sei als ein Binärbaum dargestellt, und wir wollen diesen Binärbaum **verstrecken** (manchmal **flattening** genannt), also in eine lineare Zeichenfolge überführen, aus der sich die ursprüngliche Struktur wiedergewinnen läßt. In der Schreibweise erlauben wir uns im Interesse der Klarheit einige Freiheiten, und wir setzen Klammern nach Bedarf.



Unsere drei Strategien liefern nun (leicht erweitert wegen der Klammern):

```
prefix(T) = *( +(a, b), -(c, d))

infix(T) = ((a + b) * (c - d))

postfix(T) = a b + c d - *
```

- Die Präfix-Form entspricht der funktionalen Schreibweise:
  prefix(T) = prod(sum(a,b),dif(c,d)) oder (* (+ a b) (- c d))
- Die Infix-Strategie liefert die normale Term-Schreibweise zurück, mit so vielen Klammern, daß wir auf den Vorrang der Operationen nicht achten müssen.
- Die Postfix-Form kommt ganz ohne Klammern aus; deutlicher ist die Variante:
  postfix(T) = a b ADD c d SUB MUL
  dabei beziehen sich die Operationen jeweils auf die davor schon eingelesenen Operanden. Manche Prozessoren arbeiten so (einschließlich einiger besserer Taschenrechner), und auch bereits die alten Registrierkassen, bei denen das Eingeben eines Postens und das anschließende Saldieren getrennte Aktionen waren.

Die Präfix-Form und die Postfix-Form werden gelegentlich auch **polnische** (**PN**) bzw. **umgekehrte polnische** Notation **UPN**) genannt, nach dem polnischen Logiker Lukasiewicz, der sie ca. 1928 eingeführt hat. Sie kommen ohne Klammern aus, sofern die Stelligkeit (S. 91) der Operationen bekannt ist.

Aus den drei Verstreckungen läßt sich der ursprüngliche Binärbaum schematisch wiedergewinnen; für die Präfix- und die Postfix-Darstellung können die Leser vielleicht erraten, wie das geht. Man braucht als Hilfs-Datenstruktur einen expliziten Stapel (S. 103) für Zwischenresultate.

107 Vereinigungstypen, Varianten

variant

Gelegentlich will man mehrere Datentypen mit gleicher Bedeutung, aber unterschiedlichen internen Formats zu einem **Vereinigungstyp** zusammenfassen, der logisch die **disjunkte Vereinigung** der gegebenen Typen ist. Der Zugriff auf ein Datenelement hängt dann davon ab, in welchem internen Format die Daten abgelegt sind, und daher muß jedes Datenelement zusätzlich eine Angabe über den Grundtyp enthalten, die zur Laufzeit des Programms ausgewertet werden kann.

Modula 2 bietet dazu Verbunde mit **Variante**n an. Ein solcher Verbund besitzt einen gemeinsamen Teil (er kann leer sein), ein **Schalterfeld** von einem geeigneten skalaren Typ, sowie mehrere alternative Fortsetzungen des gemeinsamen Teils. Der aktuelle Inhalt des Schalterfeldes gibt an, welche Variante gerade aktuell ist. Die Varianten belegen *denselben* Speicherplatz.

Beispiel: Punkte in der Ebene können wahlweise durch Kartesische Koordinaten oder durch Polarkoordinaten angegeben werden. Dies läßt sich durch folgenden Vereinigungstyp ausdrücken:

```
TYPE Punkt = RECORD CASE kp: (K, P) OF
                          K: x, y: REAL; |
                          P: r, phi: REAL;
                    END;
             END;
```

Zur Laufzeit kann der Verbund, der einen Punkt beschreibt, aus den Feldern (kp, x, y) bestehen, dann muß kp=K gelten; oder aber er besteht aus (kp, r, phi), dann muß kp=P sein. Programme, die mit solchen Verbunden arbeiten, können durch Abfrage des Schalterfeldes kp feststellen, welche Darstellung vorliegt. Ein Zugriff über die Selektoren der anderen Variante ist dann natürlich unsinnig.

In Sonderfällen kann das Schalterfeld fehlen (allerdings nicht seine Typangabe!) dann muß das Programm anderswo her wissen, welche Variante gerade gültig ist. Generell muß davon abgeraten werden; solche Programme sind extrem schwer korrekt zu bekommen.

In diesem Falle ist es möglich, und es kann sogar manchmal sinnvoll sein, auf denselben Dateninhalt über Selektoren verschiedener Varianten zuzugreifen. So kann man mit der Typvereinbarung

```
TYPE ptrval = RECORD CASE : BOOLEAN OF
                      TRUE: ptr: liste; |
                      FALSE: val: CARDINAL;
                    END;
              END;
```

auf die einem Pointerwert vom Typ *liste* entsprechende interne Adressen-Darstellung als CARDINAL-Wert zugreifen (vorausgesetzt, der Umfang von CARDINAL ist groß genug), und sogar neue Pointer-Werte erzeugen. Das ist etwa notwendig, wenn man selbst eine Halden-Verwaltung schreiben muß; natürlich ist ein solches Programm extrem vom Rechnertyp abhängig, und man muß ganz genau wissen, was man tut. Für Menschen mit schwachen Nerven ist das nichts!

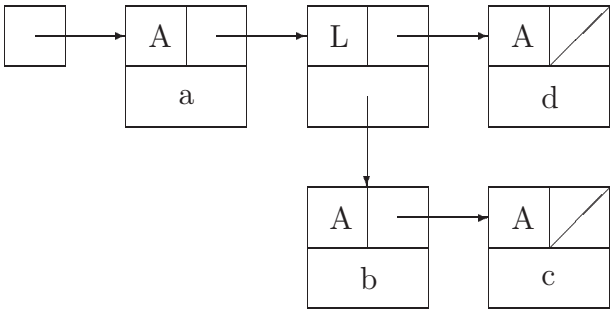
108 Rekursive Listen

rlist1

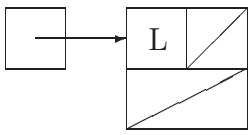
In der Programmiersprache LISP (S. 13) können die Listen, die dort als grundlegende Datenstruktur verwendet werden, als *Elemente* neben Atomen auch selbst Listen enthalten. Solche **rekursive Listen** lassen sich in Modula 2 nachbilden: wir erweitern unseren früher (S. 94) verwendeten Typ *knoten* zu einem Verbund mit Varianten (S. 118) und einem Schalterfeld *ka:*

```
TYPE knoten = RECORD nach: liste;
                     CASE ka: (A, L) OF
                          A: wert: atom; |
                          L: vws: liste;
                     END;
              END;
```

Damit kann die rekursive Liste (a (b c) d) jetzt als Geflecht dargestellt werden:



Die rekursive Liste (()) ist nicht etwa leer, sondern enthält als einziges Element eine leere Liste () !



109 Suchbäume (1)

stree1

Wenn wir Nutzdaten in einem Geflecht ablegen, so wollen wir sie auch effizient wiederfinden und womöglich aktualisieren oder löschen können. Dazu versehen wir sie mit einem **Suchschlüssel** aus einem linear geordneten Datentyp. Zur Ablage eignet sich ein **Suchbaum**, ein Binärbaum mit einem zusätzlichen Schlüsselfeld *key* in jedem Knoten und der Zusatzregel, die für *jeden* Knoten gilt:

> Im linken Unterbaum liegen nur Datenelemente mit *kleinerem* Suchschlüssel, im rechten Unterbaum nur Elemente mit *größerem* Schlüsselwert als im Knoten selbst.

Mit den Datentypen

```
TYPE stree = POINTER TO sknoten;

TYPE sknoten = RECORD key: keytype;
                      wert: atom;
                left, right: stree;
               END;
```

kann man nun das Eintragen eines Datenwertes x unter dem Schlüssel k wie folgt realisieren:

```
PROCEDURE insert(k: keytype; x: atom; VAR b: stree);
BEGIN IF b = NIL
      THEN NEW(b); b^.left := NIL; b^.right := NIL;
           b^.key := k; b^.wert := x;
      ELSIF k < b^.key THEN insert (k, x, b^.left)
      ELSIF k > b^.key THEN insert (k, x, b^.right)
      ELSE (* k = b^.key *) b^.wert := x;
      END;
END insert;
```

Beim Eintragen in einen noch leeren Baum wird der Baum neu erzeugt; und wenn der Schlüssel im Baum noch nicht vorkommt, so landen wir beim Absteigen in die Teilbäume irgendwann bei einem Blatt, an welches dann ein neues Datenelement als Unterbaum angehängt wird.

Wir haben hier vorausgesetzt, daß ein Datenwert, dessen Schlüssel bereits belegt ist, durch einen neuen Wert überschrieben werden soll; ist das nicht erwünscht, so wäre im ELSE-Zweig stattdessen eine geeignete Fehlermeldung zu erzeugen, oder die Tatsache anderweitig, etwa über einen Ergebnisparameter, zurückzumelden.

110 Suchbäume (2)

stree2

Das **Aufsuchen** eines Datenelements in einem Suchbaum (S. 120) unter Angabe seines Suchschlüssels (S. 120) geht ganz ähnlich wie das Eintragen: man steigt in den Suchbaum hinab, bis man das Element an der Wurzel eines Teilbaumes gefunden hat, oder bis man weiß, daß es nicht vorhanden sein kann. Weil man oft die gefundenen Daten modifizieren möchte, liefern wir hier nicht die Daten selbst, sondern nur einen Verweis darauf zurück, der im Mißerfolgsfalle leer ist.

```
PROCEDURE lookup(k: keytype; b: stree): stree;
BEGIN IF b = NIL THEN RETURN NIL
      ELSIF k < b^.key THEN RETURN lookup(k, b^.left);
      ELSIF k > b^.key THEN RETURN lookup(k, b^.right);
      ELSE (* k = b^.key *) RETURN b;
      END;
END lookup;
```

Diese Routine ist nicht nur rekursiv (S. 29), sondern sogar **rechtsrekursiv**. Darunter verstehen wir das folgende:

- im rekursiven Fall ist nur ein einziges Teilproblem zu lösen,
- dessen Lösung ist *unmittelbar* die Lösung des Gesamtproblems.

Rechtsrekursive Routinen können wir leicht iterativ und damit schneller machen, sofern sie nur Wertparameter haben oder allenfalls Ergebnisparameter, die identisch durchgereicht werden. Wir stellen das hier zurück, weil wir später die Suchverfahren (S. 124) unter einem allgemeineren Aspekt besprechen wollen, und dabei wird die Lösung mit abfallen.

So elegant unser Verfahren auch ist, gibt es doch ein paar Probleme. Wenn man die Daten zufällig in aufsteigender Folge der Schlüssel einträgt, so entartet der Baum zu einer *linearen Kette,* und damit wird das Eintragen und Suchen mit $\mathcal{O}(n)$ unangenehm langsam; es gibt noch andere Eingabefolgen mit dieser Eigenschaft. Beim Eintragen in *zufälliger* Reihenfolge geschieht dies zwar nicht; dann haben wir Aufwand $\mathcal{O}(\log n)$. Aber wenn man die schlimmsten Fälle unbedingt vermeiden will, so muß man zusätzliche Information im Baum mitführen, etwa die Höhe oder die Knotenzahl der Teilbäume, und notfalls den Baum gelegentlich reorganisieren. Die Einzelheiten davon und geeignete Varianten (**2-3-Bäume**, **AVL-Bäume**) gehören in eine Vorlesung über Datenstrukturen.

111 Suchbäume (3)

stree3

Das **Austragen** eines Datenelements mit dem Schlüssel k aus einem Suchbaum (S. 120), der über eine Pointervariable (S. 84) b zugänglich ist, bedarf einiger Behutsamkeit, weil wir mehrere Bedingungen erfüllen müssen:

- der restliche Baum muß über b zugänglich bleiben,
- der Baum darf beim Löschen eines inneren Knotens nicht auseinanderfallen,
- der Baum muß ein Suchbaum bleiben.

Um durch die Komplexität nicht überwältigt zu werden, gehen wir in mehreren Schritten vor: wir suchen erst das auszutragende Element auf und liefern einen Verweis p darauf zurück; freigeben kann es der Aufrufer selbst, und vielleicht hat er auch andere Pläne damit:

```
PROCEDURE delete(k: keytype, VAR b, p: stree);
BEGIN IF b = NIL THEN p := NIL
      ELSIF k < b^.key THEN delete(k, b^.left, p)
      ELSIF k > b^.key THEN delete(k, b^.right, p)
      ELSE (* k = b^.key *) unlink(b, p)
      END;
END delete;
```

Mittels *unlink(b, p)* hängen wir die Wurzel eines über b zugänglichen Teilbaumes nach p um und reorganisieren ihn; dabei brauchen wir eine neue Wurzel, falls er zwei Unterbäume hatte, sonst fällt er auseinander. Wir verwenden dafür den „infix-ersten (S. 116)“ Knoten des rechten Unterbaumes, so bleibt der Baum ein Suchbaum (rechts und links vertauscht ginge auch):

```
PROCEDURE unlink(VAR b, p);
BEGIN p := b;
      IF p^.left = NIL THEN b := p^.right
      ELSIF p^.right = NIL THEN b := p^.left
      ELSE unlinkfirst(p^.right, b);
           b^.left := p^.left; b^.right := p^.right;
      END;
END unlink;
```

112 Suchbäume (4)

stree4

Wir wollen den infix-ersten (S. 116) Knoten eines Suchbaumes (S. 120) abhängen. Das ist entweder seine Wurzel, falls der linke Unterbaum leer ist, oder der erste Knoten dieses Unterbaumes. Der Baum ist vorher und auch nachher über b zugänglich, auch wenn sich dabei seine Wurzel ändert; nach p kommt ein Verweis auf den abgehängten Knoten:

```
PROCEDURE unlinkfirst(VAR b, p);
BEGIN IF b^.left = NIL THEN p := b; b := p^.right;
      ELSE unlinkfirst(b^.left, p)
      END;
END unlinkfirst;
```

Diese Routine ist überraschenderweise korrekt, obgleich das erst nicht so aussieht: sie reorganisiert den über ihren VAR-Parameter b adressierten Baum. Ist kein linker Unterbaum vorhanden, so bekommt der Baum eine andere Wurzel; beim rekursiven Aufruf bleibt aber die Wurzel erhalten, und der linke Unterbaum wird reorganisiert; dabei kann er eine neue Wurzel bekommen, die an der gleichen Stelle angekettet wird!


Die Routinen *delete* und *unlinkfirst* und auch unsere oben angegebene Routine *insert* zum Einfügen (S. 120) neuer Elemente sind rekursiv, sogar rechtsrekursiv (S. 121); aber ihre Ergebnisparameter werden nicht alle identisch durchgereicht. Sie in iterative Routinen umzuformen ist dennoch möglich, aber etwas mühsam, und das Ergebnis ist unübersichtlich. Wir sparen uns den Versuch deshalb, weil wir annehmen, daß in der Regel das Neueintragen und Austragen gegenüber dem Suchen und Modifizieren eher selten vorkommt, sodaß sich der Aufwand wohl nicht lohnt.

Eine interessante Beobachtung ist noch, daß eine Infix-Durchwanderung (S. 116) eines Suchbaums jedes Datenelement *nach* den Elementen mit kleinerem Schlüssel und *vor* den Elementen mit größerem Schlüssel bearbeitet (S. 115), also genau in aufsteigend sortierter Reihenfolge; wir haben also gratis ein **Sortierverfahren** erhalten. Das Verfahren ist gar nicht schlecht, aber es gibt noch bessere mit geringerem Speicherbedarf, und daher wird man es nur anwenden, wenn der Suchbaum aus anderen Gründen ohnehin vorliegt.

113 Suchverfahren (1)

search1

Das **Suchen** nach einem Datenelement mit einem gegebenen Suchschlüssel wird üblicherweise von Fall zu Fall behandelt, in Abhängigkeit von der Datenstruktur, in der die Nutzdaten abgelegt und verwaltet werden. Tatsächlich ist aber die dahinterstehende Grundlogik immer gleich; Unterschiede bestehen nur in den Zugriffsarten auf die Daten und in den darüber bekannten Informationen.

Wir behandeln hier das Suchproblem so allgemein wie möglich, indem wir nur soviel Information verwenden, wie wir unbedingt benötigen. Unser Zugang ist notwendigerweise recht abstrakt (S. 10), weil wir alles an konkreten Details weglassen, was nicht zum Problem im engeren Sinne gehört; dafür stehen uns die Einzelheiten auch nicht im Wege.

Um überhaupt suchen zu können, setzen wir voraus:

- *Datenelemente,* die irgendwelche Nutzdaten tragen und über einen *Suchschlüssel* von einem geeigneten linearen Typ *keytype* identifiziert werden;
- für jedes Datenelement gibt es eine geeignete *Ortsangabe;*
- eine *Datenstruktur,* die die Datenelemente enthält, beschrieben durch geeignete *Beschreibungsgrößen;*
- ein *Kriterium,* ob die Datenstruktur *leer* ist;
- eine Ortsangabe als *Einstieg* in eine nicht leere Datenstruktur, die auf irgend ein Datenelement verweist.

Damit kann man die Suche als **Pseudoprogramm** bereits ausprogrammieren; in dieser Form wird es allerdings noch kein Übersetzer akzeptieren. Die nötigen Anpassungen sind im konkreten Fall offensichtlich.

```
PROCEDURE suche(k: keytype; d: Daten; VAR wo: Ort);
BEGIN IF leer(d) THEN wo := NIRGENDS
      ELSE wo := Einstieg(d);
           IF k <> key(wo)
           THEN suche(k, Teilstruktur(d), wo)
           END;
      END;
END suche;
```

Teilstruktur(d) ist eine geeignete Teilmenge von d, in der das Element zum Schlüssel k noch liegen kann. Der Einstieg gehört sicher nicht dazu, aber die Teilmenge kann durchaus wesentlich kleiner als der Rest sein, je nachdem, wie viel man über den Aufbau der Datenstruktur weiß.

114 Suchverfahren (2)

search2

Die angegebene Suchroutine (S. 124) ist rechtsrekursiv (S. 121), und sie hat nur Wertparameter bis auf das Resultat *wo,* das unverändert durchgereicht wird. Damit läßt sich eine effiziente **iterativ**e Variante herleiten, und mit einer kleinen Zusatzüberlegung ergibt sie sich auch direkt auf sehr übersichtliche Weise.

Während des Suchens sind wir in einem von drei Zuständen: entweder, wir haben das Gesuchte schon gefunden, oder wir wissen schon, daß es nicht da sein kann, oder wir wissen noch nichts darüber. Diesen **Suchzustand** rechnen wir mit:

```
PROCEDURE suche(k: keytype; d: Daten; VAR wo: Ort);
VAR Zustand: (suchen, ja, nein);
BEGIN Zustand := suchen;
      WHILE Zustand = suchen
      DO IF leer(d) THEN wo := NIRGENDS; Zustand := nein;
         ELSE wo := Einstieg(d);
              IF k = key(wo) THEN Zustand := ja
              ELSE d := Teilstruktur(d);
              END;
         END;
      END;
END suche;
```

Für unsere frühere Suchroutine für Suchbäume (S. 121) bekommen wir durch die nötigen Anpassungen jetzt die iterative Version:

```
PROCEDURE lookup(k: keytype; b: stree): stree;
VAR Zustand: (suchen, ja, nein);
BEGIN Zustand := suchen;
      WHILE Zustand = suchen
      DO IF b = NIL THEN Zustand := nein
         ELSIF k < b^.key THEN b := b^.left;
         ELSIF k > b^.key THEN b := b^.right;
         ELSE (* k = b^.key *) Zustand := ja;
         END;
      END;
      RETURN b;
END lookup;
```

Daß wir hier ein Funktionsresultat zurückliefern anstatt eines Ergebnisparameters, ist nicht wesentlich.

115 Suchverfahren (3)

search3

Unser allgemeines Suchschema (S. 124) läßt sich auch auf ganz andere Datenstrukturen anwenden, sowohl in der rekursiven (S. 124), wie auch in der iterativen (S. 125) Form.

Als Beispiel betrachten wir eine lineare Tabelle a der Länge max, in der die Daten *aufsteigend geordnet* ab 1 bis zu einer Füllhöhe n abgelegt sind. Unsere Teil-Datenstrukturen sind Ausschnitte aus der Tabelle mit den Grenzen l und r, und als Einstieg verwenden wir jeweils die Mitte einer Teiltabelle. Für l>r ist die Teiltabelle leer. Ein Ergebnis wo=0 soll bedeuten: nichts gefunden.

```
PROCEDURE binsuche(k: keytype; n: CARDINAL; VAR wo: CARDINAL);
VAR Zustand: (suchen, ja, nein);
BEGIN Zustand := suchen;
      WHILE Zustand = suchen
      DO IF l > r THEN wo := 0; Zustand := nein;
         ELSE wo := (l + r) DIV 2;
              IF k = a[wo].key THEN Zustand := ja
              ELSIF k < a[wo].key THEN r := wo - 1
              ELSE (* k > a[wo].key *) l := wo + 1
              END;
         END;
      END;
END binsuche;
```

Dieses Verfahren nennt man **binäre Suche** oder auch **logarithmische Suche**, weil es den Aufwand $\mathcal{O}$(log n) hat; so etwa suchen wir auch in einem Telefonbuch.

D.E.Knuth schreibt darüber 1968 in seinem Standardwerk “The Art of Computer Programming” folgendes:

> Although the basic idea of binary searching is comparatively straightforward, the details can be somewhat tricky, and many good programmers have done it wrong the first few times they tried.

Mit unserem Ansatz, der den Suchzustand mitrechnet, muß man sich inzwischen wohl eher anstrengen, um Fehler zu machen. Wir vermuten: das liegt daran, daß wir auf einer recht hohen Abstraktionsebene eingestiegen sind, auf der man klar sieht, worauf es ankommt.

116 Suchverfahren (4)

search4

Unser allgemeines Suchschema (S. 124) liefert eine elegante Lösung für ein klassisches Problem, das traditionell in einer Weise angegangen wird, welche die Verwendung von Sprüngen oder des Abbruchs einer Schleife nahelegt; wir kommen hier bequem ohne sie aus. Es handelt sich um das Suchen in einer rechteckigen Matrix:

```
TYPE Zeilen = 1 .. zmax;
     Spalten = 1 .. smax;
     Daten = RECORD wert: atom;
                    key: keytype;
             END;
VAR a: ARRAY [Zeilen, Spalten] OF Daten;

PROCEDURE suche(k: keytype; VAR found: BOOLEAN;
                VAR z: Zeilen; VAR s: Spalten);
VAR Zustand: (suchen, ja, nein);
BEGIN Zustand := suchen; z := 1; s := 1;
      WHILE Zustand = suchen
      DO IF a[z,s].key = k then Zustand := ja;
         ELSIF s < smax THEN s := s + 1;
         ELSIF z < zmax THEN z := z + 1; s := 1;
         ELSE Zustand := nein;
         END;
      END;
      found := Zustand = ja;
END suche;
```

Traditionell wird dies mit zwei geschachtelten Schleifen und jeweils vorzeitigem Ausprung programmiert; unsere Lösung finden wir klarer, und sie ist mindestens ebenso effizient.

```
PROCEDURE suche(k: keytype; VAR found: BOOLEAN;
                VAR z: Zeilen; VAR s: Spalten);
BEGIN found := FALSE;
      FOR z := 1 TO zmax
      DO FOR s := 1 TO smax
         DO IF a[z,s].key = k THEN found := TRUE; EXIT;
            END;
         END; IF found THEN EXIT END;
      END;
END suche;
```

In manchen Sprachen kann man mit EXIT auch mehrere geschachtelte Schleifen verlassen; dann fällt der zweite Aussprung weg.

Abgesehen vom Programmieren auf Maschinenebene, wo die höheren Konstrukte fehlen, ist uns kein Beispiel (mehr) bekannt, in denen explizite Sprünge von Vorteil wären.

117 Suchverfahren (5)

search5

Leider gibt es Fälle, bei denen unser allgemeines Suchschema (S. 124) nicht unmittelbar auf eine iterative Lösung führt, nämlich dann, wenn die Datenstruktur, in der wir suchen, beim Entfernen des Einstiegselements in mehrere Teile zerfällt. Ein Beispiel ist etwa das Suchen in einem Wald (S. 113) oder auch in seinem äquivalenten (S. 114) Binärbaum.

```
PROCEDURE lookup(k: keytype; b: btree; VAR p: btree);
BEGIN IF b = NIL THEN p := NIL;
      ELSE lookup(k, b^.left, p);
           IF p = NIL THEN lookup(k, b^.right, p);
      END;
END lookup;
```

Der zweite Aufruf von *lookup* ist rechtsrekursiv (S. 121) und kann wie gewohnt entfernt werden, der erste aber nicht:

```
PROCEDURE lookup(k: keytype; b: btree; VAR p: btree);
VAR Zustand: (suchen, ja, nein);
BEGIN Zustand := suchen;
      WHILE Zustand = suchen
      DO IF b = NIL THEN p := NIL; Zustand := nein;
         ELSE lookup(k, b^.left, p);
              IF p <> NIL THEN Zustand := ja;
              ELSE b := b^.right;
              END;
         END;
      END;
END lookup;
```

Behelfen könnte man sich mit einer zusätzlichen Liste der noch unerledigten Teilprobleme; dies macht unser Übersetzer beim rekursiven Aufruf aber ohnehin automatisch.

118 Rekursion und Iteration (1) iter1

Bei der rekursiven (S. 29) Zerlegung eines Problems (S. 27) in kleinere Teilprobleme tritt ein Sonderfall häufig auf (leider nicht oft genug), den wir **rechtsrekursiv** nennen, und der sich in eine effiziente nichtrekursive **iterativ**e Lösung umwandeln läßt:

- wenn sich das Problem nicht *direkt* lösen läßt, so ist nur noch *ein einzelnes* Teilproblem zu lösen,
- die Lösung dieses Teilproblems gibt *direkt* die Lösung des Gesamtproblems.

Die hier verwendete Bezeichnung *rechtsrekursiv* leitet sich her aus einer Analogie mit den Formalen Sprachen (S. 62): die Grammatik (S. 64)

Problem → lösedirekt
Problem → reduziere Problem

ist regulär (S. 64) (Klasse 3) und läßt sich in einen regulären Ausdruck (S. 71) umwandeln:

Problem → (reduziere)* lösedirekt

Als Pseudo-Programm entspricht das der rekursiven Fassung

```
PROCEDURE loese(P: Problem; VAR E: Ergebnis);
BEGIN IF NOT direktloesbar(P)
      THEN P := Teilproblem(P); loese(P, E);
      ELSE E := loesedirekt(P);
      END;
END loese;
```

Wichtig ist, daß hier P, die Beschreibung der Probleminstanz, Wertparameter (S. 50) ist, und daß der rekursive Aufruf genauso wie der Erstaufruf aussieht; dem entspricht die iterative Form:

```
PROCEDURE loese(P: Problem; VAR E: Ergebnis);
BEGIN WHILE NOT direktloesbar(P)
      DO P := Teilproblem(P);
      END;
      E := loesedirekt(P);
END loese;
```

119 Rekursion und Iteration (2) iter2

Hinter der gezeigten Reduktion (S. 129) einer rechtsrekursiven Problemlösung auf eine iterative Version steht die Idee, den Aktivierungsblock (S. 54) der ersten Inkarnation (S. 54) der Prozedur *wiederzuverwenden*, statt für die rekursiv aufgerufene Instanz ein neues Exemplar anzulegen. Dies ist möglich, weil sein Inhalt hier nach der Rückkehr aus dem rekursiven Aufruf nicht mehr benötigt wird bis auf die Rückkehr-Information (S. 54) und den Ergebnisverweis, die beide noch richtig stehen; das rekursiv gerufene (bzw. hier direkt aktivierte!) Exemplar benötigt nur neue Eingangsparameterwerte, und die haben wir selbst korrekt umbesetzt.

Wie das Prinzip arbeitet, sehen wir etwa an einem schon früher angesprochenen Beispiel: Kopieren (S. 98) einer linear verketteten Liste. An der unmittelbar naheliegenden rekursiven Version

```
PROCEDURE copy(l: liste; VAR p: liste):
BEGIN IF l <> NIL
      THEN NEW(p);
           p^.wert := l^.wert; copy(l^.tail, p^.tail);
      ELSE p := NIL;
      END;
END copy;
```

stört uns, daß sich der Ergebnisparameter p beim rekursiven Aufruf auf ein anderes Objekt als beim Erstaufruf bezieht. Wir verwenden für das Kopieren des Listenschwanzes nun eine neue Routine *copytail,* der wir den *Bezug* auf den *bereits existierenden* Kopf der neuen Liste als *Wertparameter* mitgeben, und die nur den Schwanz der Liste kopieren soll:

```
PROCEDURE copy(l: liste; VAR p: liste):
BEGIN IF l <> NIL
      THEN NEW(p);
           p^.wert := l^.wert; copytail(l, p);
      ELSE p := NIL;
      END;
END copy;
```

Die Routine *copytail* wird nun nicht mehr mit einer leeren Liste aufgerufen; der Sonderfall ist nun, daß die Liste nur aus dem Kopf besteht.

120 Rekursion und Iteration (3)

iter3

Die Routine *copytail* hat nun nur noch Wertparameter, und sie ist rechtsrekursiv; daher können wir die Rekursion *schematisch* entfernen:

```
PROCEDURE copytail(l, q: liste):
BEGIN IF l^.nach <> NIL
      THEN NEW(q^.nach); q := q^.nach; l := l^.nach;
           q^.wert := l^.wert; copytail(l, q);
      ELSE q^.nach := NIL;
      END;
END copytail;
```

```
PROCEDURE copytail(l, q: liste):
BEGIN WHILE l^.nach <> NIL
      DO NEW(q^.nach); q := q^.nach; l := l^.nach;
         q^.wert := l^.wert;
      END;
      q^.nach := NIL;
      END;
END copytail;
```

Die Routine *copytail* ist nun nicht mehr rekursiv und wird nur noch einmal aufgerufen; wenn wir ihren Rumpf anstelle ihres Aufrufs nach *copy* kopieren, brauchen wir noch eine Hilfszelle für ihren (ehemaligen) Parameter q (den wir deshalb auch schon so genannt haben), und den wir einmal mit p besetzen.

```
PROCEDURE copy(l: liste; VAR p: liste):
VAR q: liste;
BEGIN IF l <> NIL
      THEN NEW(p); q := p; q^.wert := l^.wert;
           WHILE l^.nach <> NIL
           DO NEW(q^.nach); q := q^.nach; l := l^.nach;
              q^.wert := l^.wert;
           END;
           q^.nach := NIL;
      ELSE p := NIL;
      END;
END copy;
```

Diese Routine ist bis auf Details identisch mit der früher (S. 98) angegebenen, *ad hoc* geschriebenen Version; das ist sehr beruhigend.

121 allgemeine Graphen

graph1

Alle bisher besprochenen Datenstrukturen lassen sich auffassen als Sonderfälle einer allgemeineren Struktur, die **Graph** heißt. Graphen sind von Bedeutung weit über die Informatik hinaus, von der Mathematik bis hin zur Betriebswirtschaft, und es gibt einen eigenen Wissenschaftszweig, die *Graphentheorie.* Leider ist die Sprechweise dort nicht immer ganz einheitlich; das führt manchmal zu Irritationen.

Ein Graph (V, E) besteht aus einer Menge V von **Knoten** (Vertices), die verbunden sein können durch eine Menge E von **gerichtet**en oder **ungerichtet**en **Kanten** (Edges). Entsprechend spricht man von *gerichteten* und *ungerichteten* Graphen. Die Kanten kann man als *geordnete* bzw. *ungeordnete* Paare von Knoten auffassen; die Kanten in einem *gerichteten* Graphen nennt man oft auch **Pfeil**e.

Eine Folge von aneinanderstoßenden Kanten, bzw. von Pfeilen mit gleicher Richtung, nennt man einen **Weg** oder **Pfad**. Ein *geschlossener* Weg heißt ein **Kreis** oder ein **Zyklus**. Ein Kreis der Länge 1 heißt **Schlinge**.

Je nach der Anwendung sind ungerichtete oder gerichtete Graphen zur Modellierung zweckmäßiger; dazwischen überzugehen ist einfach. Zu jedem gerichteten Graphen gibt es einen zugeordneten ungerichteten Graphen, den man bekommt, indem man die Richtung der Pfeile ignoriert; dabei verliert man natürlich Information. Umgekehrt kann man einen ungerichteten Graphen durch einen gerichteten Graphen darstellen: man ersetzt jede Schlinge durch einen geschlossenen Pfeil, und jede Kante, die keine Schlinge ist, durch ein Paar einander entgegengerichteter Pfeile.

Ein ungerichteter Graph, in dem es von jedem Knoten zu jedem anderen Knoten einen Weg gibt, heißt **zusammenhängend**. Allgemein gibt es zu jedem Knoten seine **Zusammenhangskomponente**: den größten zusammenhängenden Teilgraphen, in dem er liegt, also die Menge der Knoten, die von ihm aus durch Wege erreichbar sind. Wenn es zwischen je zwei Knoten je nur maximal *einen* Weg gibt, heißt der Graph **einfach zusammenhängend**.

Zur technischen Darstellung von Graphen gibt es mehrere Möglichkeiten:

- eine Liste der Knoten und eine Liste der Kanten als Knotenpaare; dies ist die platzsparendste Lösung, aber das Suchen nach Kanten kann unbequem sein.
- die **Inzidenzmatrix** oder **Adjazenzmatrix**: eine mit den Knoten indizierte quadratische Boolesche Matrix, in der jedes Feld ein Paar bezeichnet und angibt, ob eine (gerichtete) Kante besteht; der Speicheraufwand ist womöglich hoch, und oft ist die Matrix sehr dünn besetzt.
- zu jedem Knoten eine Liste der Nachbarn bzw. der ausgehenden Kanten. Bei gerichteten Graphen genügt die Liste der Nachfolger oder auch der Vorgänger allein; die jeweils andere Liste läßt sich leicht berechnen.

122 Binäre Relationen (1) rel1

Zwischen gerichteten Graphen (S. 132) und *binären Relationen* besteht ein unmittelbarer Zusammenhang: eine **binäre Relation** R über einer Grundmenge M ist eine Menge von Paaren R $\subseteq$ M×M, welche bestehende Beziehungen zwischen den Elementen von M beschreibt; für <a,b>∈R schreibt man gerne aRb. Der zugehörige gerichtete Graph (M,R) heißt **Relationsgraph** oder auch **Pfeildiagramm** zu R; er beschreibt anschaulich dieselbe Information.

Für die technische Darstellung von binären Relationen haben wir dieselben Möglichkeiten zur Verfügung wie bei gerichteten Graphen (S. 132); dazu kommen noch ein paar Sonderfälle. Oft wird die Inzidenzmatrix (S. 132) nicht mit Wahrheitswerten, sondern mit den Werten 0 (für FALSE) und 1 (für TRUE) besetzt, das kann manchmal bequem sein.

Aus der Darstellung als Inzidenzmatrix sieht man leicht, daß die Anzahl der möglichen Relationen auf einer gegebenen Grundmenge außerordentlich groß ist, nämlich 2**(n**2) für n Knoten (für nur 3 Knoten gibt es bereits 512 Relationen, und die meisten davon sind völlig uninteressant).

Für die Praxis bedeutsam sind Relationen mit speziellen Eigenschaften:

- eine Relation R heißt **reflexiv**, wenn gilt:

  $\forall x: xRx$

- eine Relation R heißt **irreflexiv**, wenn gilt:

  $\forall x: \neg(xRx)$

- eine Relation R heißt **symmetrisch**, wenn gilt:

  $\forall x,y: xRy \iff yRx$

- eine Relation R heißt **asymmetrisch**, wenn gilt:

  $\forall x<>y: xRy \Rightarrow \neg(yRx)$

- eine Relation R heißt **antisymmetrisch**, wenn gilt:

  $\forall x<>y: xRy \iff \neg(yRx)$

- eine Relation R heißt **transitiv**, wenn gilt:

  $\forall x,y,z: xRy \wedge yRz \Rightarrow xRz$

123 Binäre Relationen (2) rel2

Eine Relation (S. 133), die *reflexiv, symmetrisch* und *transitiv* ist, heißt eine **Äquivalenzrelation**. Solche Relationen sind wichtig, weil sie die Tatsache ausdrücken, daß zwei Elemente ein geeignetes *Merkmal gemeinsam* haben.

Zu jeder Äquivalenzrelation R gehört eine **Klassenzerlegung** der Grundmenge M, und umgekehrt: die **Äquivalenzklasse** [a] eines Elements a besteht aus allen zu a äquivalenten Elementen von M. Zwei verschiedene Klassen sind **disjunkt**, haben also kein Element gemeinsam, und insgesamt schöpfen sie die Grundmenge aus.

Umgekehrt können wir einer beliebigen disjunkten Klassenzerlegung der Grundmenge eine Äquivalenzrelation zuordnen: zwei Elemente sind genau dann äquivalent, wenn sie in derselben Klasse liegen.

Eine Relation R, die *irreflexiv, asymmetrisch* und *transitiv* ist, heißt eine (schwache) **Halbordnung**; sie kann ausdrücken, daß ein Element bezüglich geeigneter Merkmale *kleiner* als ein anderes ist. Dabei ist es nicht nötig, daß für ein beliebiges Paar <a,b> von Elementen genau eine der drei Beziehungen aRb, bRa oder a=b gilt; manche Paare können **unvergleichbar** sein.

Beispiele für Halbordnungen:

- Größenvergleich natürlicher Zahlen
- Teilbarkeit natürlicher Zahlen
- Punkte in der Ebene mit (x1,y1)R(x2,y2) $\iff$ x1<x2 $\wedge$ y1<y2
- die Teilmengenrelation $\subset$

Eine Halbordnung, bei der alle Paare von Elementen vergleichbar sind, heißt eine (schwache) **Totalordnung**. Sie bestimmt für die Elemente der Grundmenge eine eindeutige Reihenfolge.

Ist eine Relation R asymmetrisch, transitiv und *reflexiv* anstatt *irreflexiv*, so nennt man sie eine *starke* Halbordnung (ebenso kann man starke Totalordnungen einführen). Für sie gilt: aRb $\wedge$ bRa $\Rightarrow$ a=b

Eine Halbordnung läßt sich stets in eine Totalordnung einbetten, oft auf mannigfache Weise. Dazu legt man für Paare von vorher unvergleichbaren Elementen willkürlich eine Reihenfolge fest und nimmt jeweils gleich die Beziehungen dazu, die sich aus der Forderung der Transitivität ergeben. Ein systematisches Verfahren, aus einer Halbordnung eine Totalordnung zu machen, heißt **topologisch**es *Sortieren.* Wir gehen hier darauf nicht ein.

124 Relationenalgebra rel3

Mit binären Relationen läßt sich ein algebraischer Kalkül aufbauen:

- die **Nullrelation** *0:* dabei bestehen gar keine Beziehungen, alle Elemente der Inzidenzmatrix sind 0 (FALSE)
- die **Allrelation** *1:* jedes Element steht zu jedem in Beziehung (auch zu sich selbst!), alle Elemente der Inzidenzmatrix sind 1 (TRUE)
- zu jeder Relation R gibt es das **Komplement** $\neg R$*:* es besteht eine Beziehung genau dann, wenn in R keine besteht, alle Elemente der Inzidenzmatrix werden invertiert
- zu zwei Relationen R und S gibt es die **Konjunktion** $R \wedge S$*:* die Elemente der Inzidenzmatrizen werden mit $\wedge$ verknüpft
- zu zwei Relationen R und S gibt es die **Disjunktion** $R \vee S$*:* die Elemente der Inzidenzmatrizen werden mit $\vee$ verknüpft

Damit haben wir bereits eine **Boolesche Algebra** erhalten; wir nehmen weitere Elemente und Operationen hinzu:

- die **Identität** *Id:* jedes Element steht nur zu sich selbst in Beziehung, die Inzidenzmatrix ist die Einheitsmatrix
- zu zwei Relationen R und S gibt es das **Relationenprodukt** $R.S$*:*

  $$\text{aR.Sb} \iff \exists \text{c: aRc} \wedge \text{cSb}$$

  Die Inzidenzmatrizen werden nach den Regeln der Matrixalgebra multipliziert; dabei wird aber + durch $\vee$ und $\times$ durch $\wedge$ ersetzt. Wenn man die Matrixelemente 0 und 1 verwendet, kann man auch in der Matrixalgebra rechnen und dann das Resultat „stutzen“, also Werte >1 auf 1 reduzieren. Den gleichen Kunstgriff kann man auch bei der Disjunktion verwenden; die Konjunktion kommt schon richtig heraus.
- Für das Relationenprodukt ist die Relation *Id* das Einselement, die Nullrelation *0* das Nullelement; mit der Disjunktion anstelle einer Addition ergibt sich eine Ringstruktur.

Die so erhaltene **Relationenalgebra** kann man noch erweitern auf Relationen zwischen verschiedenen Grundmengen, und um mehrstellige Relationen; der erhaltene Formalismus ist wichtig als Grundlage der Theorie der **Datenbanken**.

125 Transitive Hülle

rel4

Zu jeder binären Relation R, auch wenn sie nicht transitiv ist, lassen sich umfassende transitive Relationen finden, und die kleinste davon ist eindeutig bestimmt. Sie heißt **transitive Hülle** Trans(R).

Daß die transitive Hülle existiert und eindeutig bestimmt ist, läßt sich am einfachsten über den Relationsgraphen (S. 133) einsehen: <a,b> ∈ R bedeutet: es gibt einen Pfeil von a nach b; und die Transitivitätsbedingung sagt aus: wenn es einen Pfeil von a nach b und einen von b nach c gibt, also einen *Weg* von a nach c über b, so gibt es auch einen *Pfeil* von a nach c. Im Graphen einer transitiven Relation gibt es also für jeden *Weg* einen *Pfeil* vom Anfangspunkt zum Endpunkt, und um die transitive Hülle zu bilden, betrachtet man alle Wege und nimmt die genannten Pfeile dazu, sofern sie nicht schon vorhanden waren, und keine weiteren.

Um die transitive Hülle zu berechnen, kann man also für jedes Tripel <a,b,c> in der Transitivitätsbeziehung die Pfeile von a nach c ergänzen, etwa folgendermaßen:

```
TYPE knoten = 1 .. n;
TYPE vorgaenger = SET OF knoten;
TYPE relation = ARRAY [knoten] OF vorgaenger;

PROCEDURE Trans(rel: relation; VAR trel: relation);
VAR i, k: knoten;
BEGIN trel := rel;
      FOR i := 1 TO n
      DO FOR k := 1 TO n
         DO IF i IN trel[k]
            THEN trel[k] := trel[k] + trel[i];
            END;
         END;
      END;
END Trans;
```

Dies ist der berühmte Algorithmus von **Warshall**, und man möchte glauben, daß man ihn mehrmals ablaufen lassen müßte, bis sich nichts mehr ändert. Überraschenderweise ist das aber nicht nötig, weil bei jedem Schritt in der äußeren Schleife die Ergebnisse früherer Schritte mit verwendet werden. Das sieht man am besten an einem Beispiel (ausprobieren!); der Beweis ist etwas schwieriger. Wer ihn nachlesen möchte, findet ihn in:

ACM Journal 9/1, 11 - 12 (1962)

126 Markierung von Graphen

graph2

Wenn wir in einem allgemeinen Graphen (S. 132) Datenelemente ablegen und wiederfinden wollen, so brauchen wir ein Durchwanderungsverfahren (S. 115), um jedes Datenelement aufzufinden und genau einmal zu bearbeiten. Wenn eine Liste aller Knoten vorliegt, so ist dies einfach; öfters hat man aber nur Information über die Kanten. Da die Kanten im Prinzip ganz beliebig liegen können, haben wir zuwenig Strukturinformation, um von vorneherein sicherstellen zu können, daß jedes Datenelement gefunden und genau einmal bearbeitet wird; wir müssen uns also Informationen über den bisherigen Ablauf merken, und eine Möglichkeit dazu ist eine **Markierung** oder „**Färbung**“ der bereits besuchten Knoten. Dabei ist klar, daß wir ausgehend von einem beliebigen Knoten über die Kanten nur seine Zusammenhangskomponente (S. 132) erreichen können; Knoten außerhalb davon müssen wir auf andere Weise suchen, und wir nehmen an, daß eine solche Möglichkeit gegeben ist.

Wir wollen jede Komponente eines Graphen mit einer eigenen Farbe markieren; dabei wird auch jeder Knoten genau einmal markiert (und kann dabei auch bearbeitet (S. 115) werden).

```
TYPE Farbe = 1 .. fmax;

PROCEDURE markiereGraph;
VAR x: Knoten; f: Farbe;
BEGIN f := 1;
      WHILE exist(unmarkierteKnoten)
      DO x := any(unmarkierteKnoten);
         markiereKomponente(x, f);
         f := f + 1;
      END;
END markiereGraph;

PROCEDURE markiereKomponente(x: Knoten; f: Farbe);
VAR y: Knoten;
BEGIN markiereKnoten(x, f);
      FOR ALL y IN Nachbarn(x)
      DO IF NOT markiert(y)
         THEN markiereKomponente(y, f);
         END;
      END;
END markiereKomponente;
```

127 Tiefensuche

graph3

Das angegebene Verfahren (S. 137) zur Markierung eines ungerichteten Graphen entfernt sich vom Startknoten immer möglichst weit, (bei einem Baum würde es jeweils ganz in einen Unterbaum absteigen), daher nennt man es **Tiefensuche**. Es ist schon aus dem Altertum und aus der griechischen Sage bekannt unter dem Namen *Labyrinth-Suche;* das bezieht sich auf die Erzählung von dem griechischen Helden Theseus, der nach Kreta reiste, um dort ein Untier namens Minotauros zu erlegen, das dort in einem unterirdischen Höhlensystem, genannt **Labyrinth**, sein Unwesen trieb.

Nach einer anderen Lesart war es kein Höhlensystem, sondern der Königspalast von Knossos selbst, in dem man sich in der Tat sehr leicht verlaufen kann; jedenfalls fand Theseus den Minotauros mittels Tiefensuche, und als Markierungshilfsmittel verwendete er das Wollknäuel aus dem Strickzeug der Königstochter Ariadne, mit der er sich angefreundet hatte. Die Sache ging übrigens nicht gut aus: dem König Minos war das Ganze nicht so recht (klassische Schandmäuler behaupten: der Königin noch weniger), und so mußte die beiden jungen Leute rasch aus Kreta verschwinden, vertrugen sich aber dann doch nicht so gut, sodaß Theseus Ariadne auf der Insel Naxos sitzen ließ, wie eine wunderschöne Oper von Richard Strauss erzählt. Ihre Trauer drang bis zu den Göttern im Olymp, und Gott Bacchus soll sie getröstet haben (Skeptiker vermuten, mittels seines berühmten Produktes); und bei der Rückkehr des Theseus aufs griechische Festland ertränkte sich sein Vater auf Grund eines bedauerlichen Mißverständnisses. Man sieht hier sehr schön die ungeheure Tragweite von Informatik-Konzepten, von der Mythologie bis zur spätromantischen abendländischen Musik.

Der **Ariadne-Faden** markiert übrigens gerade die bisher besuchten Knoten, und da er wieder aufgewickelt wird, verhält er sich wie ein Stack (S. 103). Wenn wir ihn explizit mitrechnen, bekommen wir einen *nichtrekursiven* Algorithmus:

```
PROCEDURE markiereKomponente(x: Knoten; f: Farbe);
VAR y, z: Knoten; s: STACK OF Knoten;
BEGIN newstack(s); markiereKnoten(x, f); push(s, x);
      WHILE NOT isempty(s)
      DO pop(s, z);
         FOR ALL y IN Nachbarn(z)
         DO IF NOT markiert(y)
            THEN markiereKnoten(y, f); push(y, f);
            END;
         END;
      END;
END markiereKomponente;
```

128 Breitensuche

graph4

Betrachtet man die Folge des Stapelinhalte, die beim Ablauf des genannten Verfahrens auftreten, so stellt man fest, daß sie gerade einen Baum beschreiben, dessen Wurzel der Einstiegsknoten ist, und in den der Reihe nach alle Knoten aufgenommen werden, jeder genau einmal. Einen solchen Baum nennt man oft „**Spanning Tree**“ oder auch **Gerüst**. Er kann für die weitere Bearbeitung des Graphen sehr nützlich sein. Für mehrere Komponenten erhält man entsprechend einen Wald.

Das angegebene nichtrekursive Verfahren (S. 138) zur Tiefensuche in einem ungerichteten Graphen stellt mittels des als Hilfsspeicher verwendeten Stapels sicher, daß jeder Knoten der Komponente genau einmal bearbeitet wird. Die *last-in-first-out*-Eigenschaft ist dafür aber gar nicht wesentlich; dasselbe könnte man auch mit einer Schlange (S. 104) erreichen, die analog verwendet wird (ersetze die Operationen sinngemäß!). Nun ist immer noch sicher, daß jeder Knoten genau einmal zur Bearbeitung herangezogen wird, aber die Reihenfolge ist anders: der neue Algorithmus bleibt solange wie möglich in der Nähe des Startknotens, er heißt **Breitensuche**.

Für beide Verfahren gibt es wichtige Anwendungen. Eine davon ist die bereits früher angesprochene Speicherbereinigung (S. 57) (Garbage Collection) auf der Halde. Hier geht es darum, alle noch erreichbaren Haldenobjekte aufzufinden und den Rest an die Verwaltung zurückzugeben. Dies findet in mehreren Phasen statt:

- entferne alle Markierungen auf der Halde.
- laufe über alle deklarierten, gültigen Pointer-Variablen (S. 84) und markiere alle Knoten des jeweils daran hängenden Geflechtes.
- gib alle nun noch unmarkierten Halden-Objekte zurück.
- vielleicht kann man dabei die Halde noch reorganisieren, um die weitere Verwaltung zu vereinfachen. Dabei müssen die betroffenen Pointer-Werte womöglich umgerechnet werden, falls ihre Bezugsobjekte verlagert wurden.
- ob man Tiefensuche oder Breitensuche zur Markierung verwendet, ist eine Zweckmäßigkeitsfrage, die durch die Art der geplanten Reorganisation beeinflußt wird; für die ansonsten korrekte Funktion spielt die Entscheidung keine Rolle.

Damit dies überhaupt möglich ist, müssen einige Voraussetzungen erfüllt sein:

- jedes Haldenobjekt muß als solches erkennbar sein.
- für jedes Haldenobjekt muß sein aktueller Typ feststellbar sein, insbesondere muß feststehen, welche seiner Felder weitere Pointer enthalten.
- alle Pointer müssen wohldefiniert oder mit NIL besetzt sein.

Diese Bedingungen sind leider nicht in allen Programmiersprachen erfüllt, in denen es Pointer gibt; für manche Sprachen ist daher eine automatische Speicherbereinigung nicht möglich. Umgekehrt ist sie für Sprachen unverzichtbar, die mit Haldenobjekten arbeiten, aber gar keine explizite Freigabe anbieten; dazu gehören wichtige objektorientierte Sprachen wie JAVA, aber auch LISP und PROLOG.

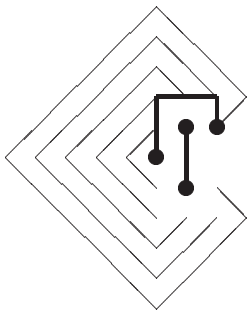
129 Breitensuche (2)

graph5

Bei der Planung der Leitungsführung auf einer Leiterplatte (**Entflechtung**) läßt sich das Verfahren der Breitensuche (S. 139) gut verwenden; wir betrachten ein etwas idealisiertes Modell:

Gegeben ist eine rechteckige Platine, die in ein festes Punktraster eingeteilt ist. Auf manchen Rasterpunkten liegen Anschlüsse für Bauteile; einige davon sollen paarweise verbunden werden. Manche Bereiche der Platte sind blockiert, einige Verbindungen bestehen bereits und müssen umgangen werden. Die Verbindungen dürfen nur parallel zu den Platinenseiten auf Rasterlinien geführt werden (evtl. mit Ecken), sollen aber möglichst kurz sein.

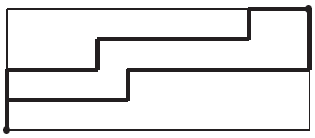
Wir fassen den freien Bereich der Platine als Graphen auf: Knoten sind alle freien Rasterpunkte, Kanten jeweils die Verbindungen zu den Nachbarn. Wir zeichnen einen Startpunkt und einen Zielpunkt aus und suchen dazwischen eine möglichst kurze Kantenfolge. Dazu starten wir eine Breitensuche vom Startpunkt aus und markieren jeden dabei gefundenen Knoten mit dem Abstand zum Startpunkt (die Markierung kann man als Höhe auffassen). Ist der Zielpunkt erreicht, so folgen wir einer möglichst steil nach unten folgenden Kantenfolge; sie ist *eine* kürzeste Verbindung zum Startpunkt (es kann davon mehrere geben).



Im Beispiel haben wir in der Mitte eine bestehende vertikale Verbindung vorgegeben, die zu umgehen ist; der Startpunkt liegt links (das Raster selbst haben wir hier nicht dargestellt). Man sieht an den diagonal liegenden Höhenlinien, wie sich schichtenweise ein Gebirge aufbaut, bis es den rechts liegenden Zielpunkt erreicht, von dem aus man dann wieder absteigt.

Gibt es mehrere Verbindungen durchzuschalten, so kommt es auf eine geschickte Reihenfolge an: man kann sich leicht den weiteren Weg versperren! Unter Umständen muß man also zurücksetzen und eine andere Reihenfolge versuchen. Dennoch ist das Verfahren (das von Lee angegeben wurde) praktisch brauchbar auch bei Verwendung durch Laien, und führt manchmal zu überraschenden Lösungen.

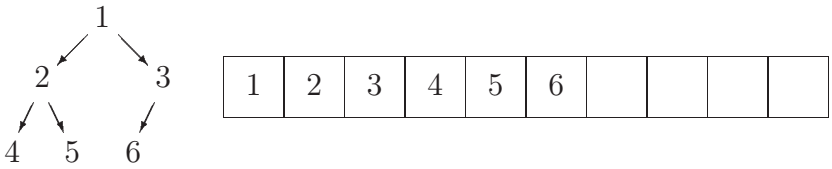
Interessant ist, daß es bei der hier verwendeten Definition der Länge eines Kantenzugs, der **Manhattan-Metrik**, zu zwei Punkten *mehrere* kürzeste Verbindungen geben kann, wie auch in einer Großstadt mit rechtwinkligem Straßennetz!



130 Vollständige Binärbäume

fulltree

Ein interessanter Sonderfall einer Datenstruktur, deren Strukturinformation implizit gegeben ist und daher nicht gespeichert werden muß, sind **vollständig**e Binärbäume. Diese haben bei gegebener Knotenzahl die minimale Höhe und damit die kürzesten Suchwege; Anwendung finden sie beispielsweise bei dem (hier nicht besprochenen) Sortierverfahren **Heapsort**.



Ein vollständiger Binärbaum wird schrittweise aufgebaut, indem man beginnend an der Wurzel die folgenden „Ebenen“ von links beginnend jeweils soweit möglich und nötig auffüllt. Die laufende Nummer eines Knotens bestimmt somit seine Position im Baum vollständig, und man kann die Dateninhalte einfach hintereinander in einem ab 1 indizierten Array ablegen. Ausgehend von der Platznummer eines Knotens im Array findet man die Nummer seines Vaters durch Halbieren; der ältere Sohn hat die doppelte Platznummer, und dessen Bruder liegt unmittelbar dahinter. So kann man bequem im Baum auf- und absteigen, ohne irgendwelche Zeiger mitführen zu müssen.

Nr(Vater(n)) = Nr(n) DIV 2
Nr(Sohn1(n)) = Nr(n) * 2 Nr(Sohn2(n)) = Nr(n) * 2 + 1

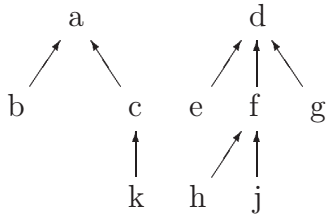
131 Äquivalenzrelationen

equiv

Zu einer Äquivalenzrelation (S. 134) gehört ein Relationsgraph (S. 133), der in disjunkte Komponenten (S. 132) zerfällt. Jede Komponente entspricht einer Äquivalenzklasse, und innerhalb der Komponente ist jeder Knoten mit jedem anderen durch eine Kante verbunden, sowie auch mit sich selbst durch eine Schlinge (S. 132).

Dieselbe Information ist auch schon in einem Gerüst (S. 139) des Graphen vorhanden, also in einem Wald, dessen Bäume gerade die Klassen aufspannen; die Wurzeln dieser Bäume kann man als Repräsentanten der Klassen ansehen, und zwei Knoten sind genau dann äquivalent, wenn sie zur selben Wurzel gehören. Der Baum für einen isolierten Knoten besteht nur aus der Wurzel allein.

Manchmal ist man nur an zwei Operationen interessiert, die traditionell *Find* und *Union* genannt werden: **Find** findet zu einem Knoten die zugehörige Klasse, und **Union** vereinigt die Klassen zweier gegebener Knoten, fügt also eine neue Äquivalenzbeziehung hinzu. Dies läßt sich platzsparend realisieren durch das Gerüst, wobei man nur von jedem Knoten zur Wurzel seines Baumes gelangen können muß. Es genügt also jeweils ein Verweis auf den *Vaterknoten,* um die Wurzel zu finden, und zur Vereinigung zweier Bäume nimmt man einen Verweis von einer der beiden Wurzeln auf einen beliebigen Knoten des anderen Baumes hinzu.



Damit können wir die Operationen *Find* und *Union* bereits korrekt realisieren, doch läßt sich die Effizienz noch erheblich verbessern durch einen Kunstgriff, der **Pfadkompression** genannt wird. *Find* arbeitet um so schneller, je kürzer der dabei zurückgelegte Weg zur Wurzel ist, und daher ersetzen wir, sobald wir die Wurzel kennen, für den Ausgangsknoten und alle Zwischenknoten den Vaterverweis durch einen Zeiger auf die Wurzel, falls er nicht bereits dorthin weist. So wird der Baum zusehends immer flacher, enthält aber immer noch dieselben Knoten und dieselbe Wurzel, und erfüllt also noch denselben Zweck. Unser Verfahren beschleunigt sich also durch den Gebrauch selbst!

Das Verfahren ist, abgesehen von seiner praktischen Brauchbarkeit, auch theoretisch interessant, weil sich bei seiner genaueren Analyse für seinen Aufwand eine Funktion ergibt, die zwar nicht konstant ist, aber geradezu unvorstellbar langsam anwächst.

132 Fibonacci-Zahlen: Aufwand

fib2

Wir versuchen, den Aufwand für die Auswertung der Funktionsprozedur Fib(n) (S. 90) zu berechnen; er sei A(n) für die n-te Fibonacci-Zahl F(n) (S. 90). Aus dem Programmcode liest man ab:

A(n) = A(n-1) + A(n-2) + c1 für n > 2
A(1) = A(2) = c2 mit irgendwelchen Konstanten c1 und c2.

und der Ansatz B(n) = A(n) + c1 führt uns sofort auf die Lösung

A(n) = (c1 + c2) * F(n) - c1

Wir sind also wieder bei den Fibonacci-Zahlen herausgekommen; wir versuchen, ihr Wachstumsverhalten zu bestimmen.

Wir beobachten, daß aufeinander folgende Fibonacci-Zahlen von den ersten paar Werten abgesehen immer um einen Faktor ≈ 1.6 anwachsen, und versuchen daher einen Potenzansatz F(n) = c * x**n für die Gleichung (1) (S. 90). Dies führt auf die quadratische Gleichung x**2 - x - 1 = 0 mit den beiden Lösungen x1 ≈ 1.618, x2 ≈ -0.618 (es ist üblich, deren positive Lösung x1 = $(\sqrt{5} + 1) / 2$ mit ϕ oder **goldener Schnitt** zu bezeichnen).

Die beiden Werte für x geben zwei linear unabhängige Lösungen F1(n) und F2(n) für die Gleichung (1), und jede Linearkombination a*F1(n) + b*F2(n) davon ist auch eine Lösung; Gleichung (2) bestimmt die Koeffizienten a und b eindeutig. Deren genaue Werte interessieren uns hier nicht; nachdem F2(n) für wachsende n schnell abnimmt, kommt es uns nur auf den Anteil von F1(n) an, und der ist von der Größenordnung $\mathcal{O}(\phi$**n). Also wachsen auch die Fibonacci-Zahlen F(n) *exponentiell* an, schneller als jede Potenz von n; und das gilt auch für den Aufwand der Auswertung von Fib(n).

Andererseits haben wir aber das Anfangsstück oben (S. 90) von Hand sehr einfach und schnell berechnen können, mit einem *konstanten* Zusatzaufwand für jede weitere Zahl, also insgesamt mit *linearem* Aufwand. Weshalb ist unsere Prozedur so schlecht, und wie können wir sie *systematisch* verbessern?

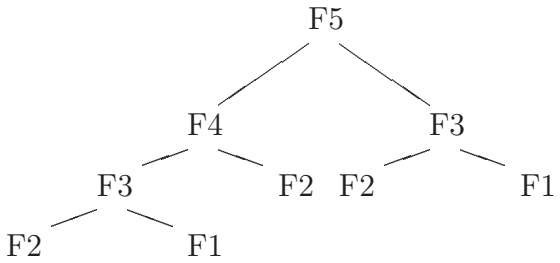
Durch die rekursiven Aufrufe (S. 29) in unserer Prozedur (S. 90) zerlegen wir das Problem der Größe n in kleinere Teilprobleme der Größen n-1 und n-2, und bei deren Auswertung geht dieser Zerlegungsprozeß weiter. Dabei zeigt sich bei genauerer Betrachtung, daß dabei die gleichen Teilprobleme mehrmals auftreten, und sie mehrmals zu lösen, ist offensichtlich überflüssig, gemäß dem Grundsatz:

effizient arbeiten bedeutet, nichts Überflüssiges tun.

Daran haben wir uns bei unserer Berechnung *von Hand* anscheinend gehalten!

133 Dynamische Programmierung (1) fib3

Bei der Berechnung beispielsweise der fünften Fibonacci-Zahl (S. 90) ergibt sich der folgende Baum von zu lösenden Teilproblemen:



Man sieht sofort, daß alle überhaupt benötigten Teilprobleme auch in dem ganz links stehenden Ast liegen. Unsere Rekursionsgleichung (S. 90)

(1) F(n) = F(n-1) + F(n-2) für n > 2
(2) F(1) = F(2) = 1

können wir statt als Zerlegungsvorschrift auch als Aufbauvorschrift lesen und die Knotenwerte im linken Ast von unten nach oben berechnen; die restlichen Werte im Baum fallen dabei mit ab, und genau so haben wir es auch von Hand gemacht.

Die hier verwendete *Lösungsstrategie:*

- verschaffe dir einen Überblick über die beteiligten Teilprobleme,
- löse alle Teilprobleme in geeigneter Reihenfolge, dabei jedes nur einmal

ist bekannt unter der merkwürdigen Bezeichnung **dynamische Programmierung** (dynamic programming). Sie ist an vielen Stellen anwendbar und stammt ursprünglich aus dem Bereich der **Planungsrechnung** (**Unternehmensforschung** oder **Operations Research**), einem Teilgebiet der Betriebswirtschaftslehre.

Aus demselben Gebiet kommen weitere für uns befremdliche Begriffe wie **lineare**, **quadratische**, **ganzzahlige** *Programmierung* (besser heute: lineare **Optimierung** etc). Dort geht es darum, aus einer Schar von möglichen Zuständen, die durch geeignete Einflußgrößen unter festgelegten Nebenbedingungen gegeben sind, eine optimale Lösung im Sinne einer vorgegebenen (z.B. linearen) Zielfunktion auszuwählen. Mit Programmierung in unserem Sinne hat das primär nichts zu tun, obgleich man natürlich heute Computerhilfe dabei in Anspruch nimmt, sogar in großem Umfang.

134 Dynamische Programmierung (2) fib4

Bei Anwendung der dynamischen Programmierung (S. 144) brauchen wir im allgemeinen eine Buchhaltung über die einzelnen Teilprobleme und ihre Lösungen; oft bietet sich dafür ein Array an. In unserem Spezialfall der Fibonacci-Zahlen (S. 90) geht es einfacher, weil wir auf dem Aufstieg durch den linken Ast des Lösungsbaumes (S. 144) nur die beiden jeweils letzten Werte brauchen; diese können wir aber direkt verwalten:

```
PROCEDURE Fib(n: CARDINAL): CARDINAL;
VAR fb, fvor, falt, i: CARDINAL;
BEGIN IF n <= 2
      THEN RETURN 1
      ELSE fb := 1; fvor := 1;
           FOR i := 3 TO n
           DO falt := fvor; fvor := fb;
              fb := falt + fvor;
           END;
           RETURN fb;
      END;
END Fib;
```

Daß dieses Verfahren, das wir auch bei der Berechnung von Hand intuitiv verwendet haben, *linearen* Aufwand benötigt, sieht man unmittelbar.

Übrigens kann man in manchen Büchern lesen, die Effizienzverbesserung käme davon, daß wir die Rekursion durch eine Iteration ersetzt haben, oder daß wir Variablen an Stelle von funktionaler Programmierung eingesetzt haben. Dies trifft nicht zu; es gibt auch eine (allerdings etwas künstliche) funktional-rekursive Lösung (S. 148), die mit linearem Aufwand auskommt. Die Verbesserung kommt hier wirklich aus der dynamischen Programmierung, und auch für die genannte rekursive Lösung ergibt sie eine Reduktion des Aufwandes, allerdings nur noch um einen konstanten Faktor.

Auch an anderen Stellen kann *dynamische Programmierung* nützlich sein, oder sie steckt bereits verborgen in den gebräuchlichen Verfahren. Bei der Berechnung der **Fakultätsfunktion**

$$\text{fak}(1) = 1, \qquad \text{fak}(n) = n * \text{fak}(n\text{-}1) = n*(n\text{-}1)*(n\text{-}2)* \ldots *1$$

ist das Aufsammeln der einzelnen Faktoren von unten herauf so naheliegend, daß man wohl gar erst keinen rekursiven Ansatz versuchen würde; daher eignet sich

dieses Beispiel auch nicht gut dazu, den möglichen Nutzen der Rekursion glaubhaft zu machen, denn hier hat sie keine erkennbaren Vorteile.


Besser sieht es bei den **Binomialkoeffizienten** aus:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \qquad \binom{n}{1} = \binom{n}{n} = 1$$

die man am besten und schnellsten von unten herauf über das bekannte **Pascal**sche **Dreieck** ausrechnet. Auch dahinter steckt die Idee der dynamischen Programmierung, und sie verbessert den Aufwand dramatisch gegenüber einer rekursiven Lösung.

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
```

135  Memoisierung

fib5

Eine weitere globale Verbesserung der Effizienz läßt sich erreichen, wenn man sich über mehrere Aufrufe der Lösungsfunktion hinweg bereits berechnete Werte merkt, und diese später nur noch nachschlägt. Diese Vorgehensweise wird **Memoisierung** genannt, und wir führen das *Prinzip* hier vor, obgleich es sich bei unserem einfachen Beispiel nicht lohnt.

Die Lebensdauer (S. 57) bereits berechneter Werte muß nun über den Aufruf der Funktion hinausgehen, und wir brauchen daher dafür eine globale Datenstruktur, im Hauptprogramm oder besser (wegen des Geheimnisprinzips (S. 27)) in einem eigenen Modul (S. 28), das die Realisierung verbirgt.

```
DEFINITION MODULE FibMod;
(* Modul fuer Fibonacci-Zahlen, robust *)

CONST MaxArg = 24;
(* fuer groessere Argumente laeuft der Zahlbereich ueber *)

PROCEDURE Fib(n: CARDINAL): CARDINAL;
(* liefert die n-te Fibonacci-Zahl, falls 0 < n < MaxArg,
   sonst den illegalen Funktionswert 0 *)

END FibMod.
```

Im Implementierungsmodul berechnen wir nun alle jemals benötigten Werte bereits zur Zeit der Initialisierung des Moduls:

```
IMPLEMENTATION MODULE FibMod;

TYPE Index = 1 .. MaxArg;
VAR FibZahl: ARRAY [Index] OF CARDINAL;
    Ind: Index;

PROCEDURE Fib(n: CARDINAL): CARDINAL;
BEGIN IF (n < 1) OR (n > MaxArg) THEN RETURN 0
      ELSE RETURN FibZahl[n]
      END:
END Fib;

BEGIN (* Modul-Initialisierung *)
      FibZahl[1] := 1; FibZahl[2] := 1;
      FOR Ind := 3 TO MaxArg
      DO FibZahl[Ind] := FibZahl[Ind-1] + FibZahl[Ind-2];
      END;
END FibMod.
```


Der Aufruf von *Fib* hat nun konstanten Aufwand. Will man nicht alle Werte auf Verdacht vorweg berechnen, so kann dies auch in *Fib* nachgeholt werden:

```
IMPLEMENTATION MODULE FibMod;

TYPE Index = 1 .. MaxArg;
VAR FibZahl: ARRAY [Index] OF CARDINAL;
    Ind: Index;

PROCEDURE Fib(n: CARDINAL): CARDINAL;
BEGIN IF (n < 1) OR (n > MaxArg) THEN RETURN 0
      ELSE WHILE Ind < n
           DO Ind := Ind + 1;
              FibZahl[Ind] := FibZahl[Ind-1] + FibZahl[Ind-2];
           END;
           RETURN FibZahl[n]
      END:
END Fib;
```

```
BEGIN (* Modul-Initialisierung *)
      FibZahl[1] := 1; FibZahl[2] := 1; Ind := 2;
END FibMod.
```

Zur Zeit der Modul-Initialisierung berechnen wir jetzt nur die Startwerte F(1) und F(2); alle anderen bis zum höchsten jeweils benötigten Wert werden erst bei Bedarf nachgezogen.

136 Fibonacci-Zahlenpaare

fib6

Wir haben oben (S. 144) den exponentiellen Aufwand bei der rekursiven Berechnung der Fibonacci-Zahlen (S. 90) dadurch reduziert, daß wir uns über den Zusammenhang der benötigten Teilprobleme einen Überblick verschafft und die Teilprobleme dann von unten herauf gelöst hatten, gemäß der Vorgehensweise der *Dynamischen Programmierung*. Dabei sind wir auf eine iterative (S. 145) Lösung gestoßen, und nach der „*herrschenden Meinung*“ kommt daher der nun nur noch lineare Aufwand.

Tatsächlich ist dies aber gar nicht der Fall, wie wir zeigen können: es gibt eine *rekursive* Lösung des Problems, die ebenfalls *linearen* Aufwand besitzt, also bis auf einen (allerdings recht großen!) konstanten Faktor ebenso effizient ist wie unsere iterative Lösung.

Der Kern unseres Ansatzes liegt in der Beobachtung, daß unsere Rekursionsgleichung

(1) $\quad F(n) = F(n\text{-}1) + F(n\text{-}2) \qquad$ für $n > 2$
(2) $\quad F(1) = F(2) = 1$

immer unmittelbar benachbarte Fibonacci-Zahlen verknüpft; fassen wir sie zu Paaren zusammen, so bekommen wir für zwei benachbarte Paare eine **einstufig**e Rekursionsbeziehung:

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} F(n-1) + F(n-2) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F(n-1) \\ F(n-2) \end{pmatrix} = A \cdot \begin{pmatrix} F(n-1) \\ F(n-2) \end{pmatrix}$$

oder auch

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = G(n) = A \cdot G(n-1), \qquad G(2) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

mit einer konstanten Matrix A. Schon hier kann man sehen, daß wir jetzt nur noch $\mathcal{O}(n)$ Teilprobleme zu lösen haben, und jeder Schritt bringt einen konstanten Beitrag, also ist der gesamte Aufwand höchstens linear!

Es gibt einen Trick, mit dem man die Gleichung für G(n) sogar mit *logarithmischem* Aufwand direkt lösen kann!

137 Fibonacci-Zahlenpaare (2) fib7

Zur Lösung unserer einstufigen (S. 146) Rekursionsgleichung nehmen wir zwischendurch die Existenz eines Abstrakten Datentyps (S. 91) *Paar* an

ADT Paar IS
SORTS: Paar, CARDINAL
OPERATIONS:
Komb: CARDINAL × CARDINAL → Paar
Teil1: Paar → CARDINAL
Teil2: Paar → CARDINAL
RULES:
Teil1(Komb(x,y)) = x
Teil2(Komb(x,y)) = y
END Paar.

oder als Definitionsmodul (S. 28) geschrieben:

```
DEFINITION MODULE Paar;

TYPE Paar;

PROCEDURE Komb(x, y: CARDINAL): Paar;

PROCEDURE Teil1(z: Paar): CARDINAL;

PROCEDURE Teil2(z: Paar): CARDINAL;

END Paar.
```

Damit können wir nun eine rein funktional-rekursive Lösung aufbauen:

```
PROCEDURE Fib(n: CARDINAL): CARDINAL;
BEGIN RETURN Teil1(FPaar(n);
END Fib;

PROCEDURE FPaar(n: CARDINAL): Paar;
BEGIN IF n <= 2 THEN RETURN Komb(1, 1)
      ELSE RETURN FKomb(FPaar(n-1))
      END:
END FPaar;
```

```
PROCEDURE FKomb(x: Paar): Paar;
BEGIN RETURN Komb(Teil1(x)+Teil2(x), Teil1(x))
END FKomb;
```

Damit hätten wir im Prinzip unser Problem gelöst, wenn wir den Typ *Paar* zur Verfügung hätten. Leider ist das aber nicht so; wir können ihn in Modula 2 nicht verwenden, ja nicht einmal realisieren: für einen opaken Typ (S. 96) belegen seine Werte zu viel Speicher, und er ist auch als Resultat einer Funktion nicht zulässig, weil Modula 2 dort aus historischen Gründen nur einfache Grundtypen erlaubt.


Wir müssen uns also irgendwie behelfen: entweder durch Übergang auf eine mächtigere Programmiersprache wie etwa Ada (S. 13) oder LISP (S. 13); oder aber indem wir den Typ *Paar* auflösen und Hilfsvariablen sowie Ergebnisparameter (S. 49) verwenden.

```
PROCEDURE Fib(n: CARDINAL): CARDINAL;
VAR x, y: CARDINAL;
BEGIN FPaar(n, x, y); RETURN x;
END Fib;

PROCEDURE FPaar(n: CARDINAL; VAR x, y: CARDINAL);
VAR u, v: CARDINAL;
BEGIN IF n <= 2 THEN x := 1; y := 1;
      ELSE FPaar(n-1, u, v);
           x := u+v; y := u;
      END:
END FPaar;
```

Der Algorithmus und sein Verhalten ändert sich dabei nicht grundsätzlich: wir haben hier auch nichts anderes getan, als was eigentlich ein Compiler für uns erledigen sollte!

Wenn wir nun auf diese letzte Lösung wieder Dynamische Programmierung (S. 144) anwenden und die Teillösungen von unten nach oben einsammeln, so kommen wir im wesentlichen bei unserer früheren iterativen Lösung (S. 145) heraus. So hat sich der Kreis geschlossen.

Index