Grundzüge der Informatik I

Elemente der sequentiellen Programmierung und
Programmierkurs MODULA-2

Notizen zur Vorlesung im WS 1994 /95

Joachim Biskup
Universität Hildesheim
Samelsonplatz 1
31141 Hildesheim

© Joachim Biskup

Joachim Biskup, Universität Hildesheim, WS 1994/95

Grundzüge der Informatik I:

Elemente der sequentiellen Programmierung und Programmiersprache MODULA-2

Vorbemerkungen:

Diese Vorlesungsnotizen sollen den Verlauf der Vorlesung und eigenständige Eigenarbeiten begleitender Literatur erleichtern, aber nicht ersetzen.

Das sind Notizen und kein ausgefeiltes Buch.

Ich versuche grundlegende Konzepte und ihre handwerkliche Anwendung in MODULA-2 eng verzahnt vorzutragen. Dadurch soll der Zugang zur Theorie erleichtert und die Praxis besser begründet werden. Andererseits leidet die Schönheit der Theorie und die Vollständigkeit des Handwerklichen darunter. Für eine Einführung scheint mir das ein annehmbares Übel zu sein.

Beim Programmierkurs stütze ich mich auf N. Wirth, Programmieren in Modula-2, Springer 1985, wohl wissend, daß das neue bewährte Modula-System manche Abweichungen enthält. Die muß der Leser selbst erfahren.

Modula-2-Sprachelemente beschreibe ich im allgemeinen unvollständig, um das neue behandelte Konzept hervorzuheben. Über die genaue und vollständige Erklärung eines Sprachelements will, muß in der Literatur schauen.

Literatur

A. MODULA-2

unbedingt: Niklaus Wirth, Programming in MODULA-2, Springer 1983

empfohlen: Gustav Pomberger, Softwaretechnik und Modula-2, Hanser, 1984.

ferner gibt es eine Reihe von Büchern über Programmieren in MODULA-2, die man sich nach eigenem Geschmack aussuchen sollte

B. Programmierung allgemein

empfohlen: W.A. Wulf, M. Shaw, P.N. Hilfinger, L. Flon, Fundamental Structures of Computer Science, Addison-Wesley 1981

F.L. Bauer, G. Goos, Informatik I, Springer, 1982.

weiterführend: F.L. Bauer, H. Wössner, Algorithmische Sprache und Programmentwicklung, Springer, 1981.

David Gries, The Science of Programming, Springer 1981.

ferner gibt es eine Vielzahl von anderen Büchern.

neuere Bücher, u.a.:

H.-J. Appelrath, Jochen Ludewig,
Skripten Informatik,
Teubner 1991.

André Spiegel, Jochen Ludewig, H.-J. Appelrath,
Aufgaben zur Skripten Informatik,
Teubner 1991.

Rainer Gmehlich, Heinrich Rust,
Mehr als nur Programmieren,
Vieweg 1993.

Peter Rechenberg,
Was ist Informatik,
Hanser 1991

A. Einführung

1.	Informatik	2
2.	Einführung in einige Grundbegriffe der Informatik	14
3.	Einige Ebenen der Programmierung	12
4.	Programmierumgebung	4
5.	Die Grenzen des Berechenbaren	8
6.	Einige historische Stichworte	4
7	Euklidisches Algorithmus zur Bestimmung des g.g.T. – ein Beispielprogramm in MODULA-2	5

B. Grundkurs Programmierung mit MODULA-2

8.	Alphabete, Worte, Syntaxdiagramme – Programmodule	8
9.	Skalare Datentypen I	21
10.	Wertzuweisung	7
11	Einführung in die axiomatische Semantik	9
12.	Strukturierte Anweisungen	42

1. Zusammensetzungen 2
2. Fallunterscheidungen 14
3. Wiederholung mit WHILE 15
4. Wiederholung mit REPEAT 2
5. beschränkte Wiederholung mit FOR 6
6. allgemeinste Fall: Wiederholung mit LOOP 3

13.	Übersetzung strukturierter Anweisungen in eine maschinennahe Sprache	11
14.	Skalare Datentypen II	6
15.	Einfache Ein-/Ausgabe	0
16.	Ein Vorgriff: Felder	4

C. Systematisches Programmieren

17. Allgemeine Methoden 15
18. Prozeduren 9
19. Lokalität 21
20. Funktionsprozeduren 5
21. Prozedurtypen 10
28 Module 7
22 Aufwandsanalyse: Suchen bei sequentieller Speicherung 30

D. Datentypen

23. Allgemeines und ein Rückgriff (: Felder) 11
24. Kartesische Produkte 12
25. Disjunkte Vereinigungen 5
26. Potenzmengen 3
27. Zeiger 26
28 Variable 5

Nachwort

29. Grenzen des Programmierens 4

A. Einführung

1. Informatik

Meyers Lexikon definiert:

> "Informatik /lat./ (engl. computer science), die Wissenschaft von den elektronischen Datenverarbeitungsanlagen (EDV-Anlagen) und den Grundlagen ihrer Anwendung".

Diese Definition wird an deutschen Universitäten im allgemeinen als zu eng und vordergründig angesehen. Nicht die Datenverarbeitungsanlagen, so wie sie zur Zeit hergestellt werden, stehen im Vordergrund des Interesses, sondern das Grundsätzliche jeglicher planmäßiger Verarbeitung von Information, wer oder was diese Verarbeitung letztlich auch durchführt.

So finden Sie im Studien- und Forschungsführer Informatik, der gemeinsam von der Gesellschaft für Mathematik und Datenverarbeitung, GMD, und dem Deutschen Akademischen Austauschdienst, DAAD, herausgegeben wird, folgende Begriffsbestimmung:

> "Informatik ist die Wissenschaft von der systematischen Verarbeitung von Informationen - insbesondere der automatischen Verarbeitung mit Hilfe von Digitalrechnern. Sie und die Technologie der Datenverarbeitungssysteme bilden die Grundpfeiler der Datenverarbeitung.
> Die Informatik erforscht die grundsätzlichen Verfahrensweisen der Informationsverarbeitung und die allgemeinen Methoden der Anwendung solcher Verfahren in den verschiedensten Bereichen. Ihre Aufgabe ist es, durch Abstraktion und Modellbildung von speziellen Gegebenheiten sowohl der technischen Realisierung existierender Datenverarbeitungsanlagen als auch von den Besonderheiten spezieller Anwendungen abzusehen und dadurch zu den allgemeinen Gesetzen, die der Informationsverarbeitung zugrunde liegen, vorzustoßen sowie Standardlösungen für Aufgaben der Praxis zu entwickeln."

Diese weitverbreitete, um Vollständigkeit bemühte Definition der Informatik stützt sich auf den Begriff der Information. Leider ist dieser selbst sehr vielschichtig und schwierig, und in Ihren ersten Lehrveranstaltungen wird er nicht im Mittelpunkt stehen. Als Leitfaden für die Anfängervorlesungen wähle ich daher eine weniger erschöpfende, aber bestimmtere und handlichere Begriffsbestimmung.

> Informatik ist die Wissenschaft und die Kunst, formale (Programmier-) Sprachen zur Beschreibung von abstrakten Dingen und abstrakten Handlungsabläufen zu erfinden, zu verwirklichen und zu benutzen.

Eine Sprache zu erfinden beinhaltet insbesondere, ihr Form zu verleihen und Bedeutung zu geben. Bedeutung kann letztlich nur durch Rückgriff auf schon vorhandene Verständigungsmittel festgelegt werden, wobei Mathematik und Logik eine besondere Rolle spielen. Die erfundenen Sprachen kann man zunächst als reine Gebilde des Geistes ansehen, die nicht unmittelbar den leidigen Beschränkungen

von Raum und Zeit unterworfen sind.

Eine Sprache zu verwirklichen heißt, sie über gedankliche Zwischenebenen schließlich auf die Ebene von wirklichen Maschinen zurückzuführen. In diesen Maschinen müssen mit den Mitteln der Technik die abstrakten Dinge des Geistes beobachtbar oder meßbar dargestellt werden und die abstrakten Handlungsabläufe des Geistes beobachtbar oder meßbar nachgebildet werden. Für diese Aufgabe muß man selbstverständlich Raum, Zeit und derzeitige Technologie berücksichtigen.

Eine Sprache zu benutzen verlangt, eine Aufgabenstellung mit den Ausdrucksmitteln der Sprache zu beschreiben. Die großen Aufgaben werden dabei in der Regel auch Forderungen der Wirtschaftlichkeit und Ansprüche der Gesellschaft umfassen.

Die obige Definition deutet auch eine Einordnung der Informatik in das Gefüge der Wissenschaften an: sie trägt einerseits Züge einer auf Mathematik und Logik abgestützten Geisteswissenschaft und andererseits Züge der Ingenieurwissenschaften, die der Kunst des Möglichen verpflichtet sind und zielgerichtet Dinge entwerfen, bauen und unterhalten. Selbstverständlich ist die Informatik dabei keine Überwissenschaft, die etwa über ihren Geburtshelfern Mathematik und Nachrichtentechnik schwebt.

2. Einführung in einige Grundbegriffe der Informatik

Dieser Abschnitt dient der zunächst noch recht ungenauen Einführung der wichtigsten für die Programmierung bedeutsamen Begriffe der Informatik. Diese Begriffe werden später – bei verschiedenen Gelegenheiten unter verschiedenen Sichtweisen – häufig wieder auftauchen.

1. Werte

Werte sind wohlunterscheidbare Dinge, die Gegenstand des Handelns sein sollen.

Bsp

0	a	ahoi
1	aa	dortmund
⋮	abc	
10		
⋮		
1023		

FALSE
⋮
TRUE

elementare Eigenschaften: für je zwei Werte kann man feststellen, ob sie **gleich** oder **ungleich** sind (wohlunterscheidbar)

einige Probleme: – wie kann eine Maschine "wissen", daß 10 eine "Zahl", aber ahoi ein "Wort" ist ?

2. Wertemengen

Werte werden zu Wertemengen zusammengefaßt. Manchmal "denkt man" sich dazu alle vorkommenden Mengen disjunkt, d.h. jeder Wert gehört zu genau einer Wertemenge (damit z.B. eine Maschine "Zahlen" von "Worten" unterscheiden kann)

Bsp: natürliche Zahlen (Kardinalzahlen) kleiner als 2^{10} : $\{0, 1, \ldots, 1023\}$

Worte der Länge ≤ 2 (Zeichenfolgen) : $\{$ ε, (Länge 0 (sog. leeres Wort))

$a, b, \ldots, z$ Länge 1

$aa, \ldots, az,$
$ba, \ldots, bz,$ Länge 2
$\vdots$
$za, \ldots, zz$ $\}$

Wahrheitswerte : $\{$FALSE, TRUE$\}$

einige Eigenschaften: - jeder Wert gehört zu einer Wertemenge

- Wertemengen "denkt man" sich häufig angeordnet z.B. Zahlen entsprechend ihrer "Größe", Worte wie im Lexikon (nicht wie oben), Worte nach Länge und dann wie im Lexikon (wie oben!)

Wahrheitswerte wie oben, d.h. FALSE < TRUE

einige Probleme: - eine Maschine "kennt" nur einige elementare Wertemengen; wie definiert man sich neue Wertemengen, z.B wie definiert man sich "Worte", wenn die Maschine nur Buchstaben "kennt"

3. (Berechenbare) Wertausdrücke

Wertausdrücke liefern Werte aus einer Wertemenge.

Wertausdrücke werden (induktiv) aufgebaut aus

1) Werten
2) Funktionszeichen
3) gewissen Trennzeichen, z.B. „(", „,", „)":

- jeder Wert ist ein Wertausdruck (er liefert sich selbst)
- ist f ein n-stelliges Funktionszeichen und sind $a_1, \dots, a_n$ Wertausdrücke, so ist auch $f(a_1, \dots, a_n)$ ein Wertausdruck (er liefert denjenigen Wert, den man erhält, wenn man die durch f bezeichnete Funktion auf die durch $a_1, \dots, a_n$ gelieferten Werte anwendet)

Ein Wertausdruck **liefert** den zugehörigen Wert durch eine **Auswertung** oder **Berechnung**.

Bsp:
- $+(3,7)$ liefert 10
- $+(2,8)$ liefert 10
- 10 liefert 10
- $+(2, \cdot(2,4))$ liefert im ersten Schritt $+(2,8)$, im zweiten Schritt 10

sub^{o}_{a} (oho) liefert aha

↑

"substituiere o durch a"

sub^{o}_{a} (dortmund) liefert dartmund

^{+}top (dortmund) liefert d

↑

"erstes Zeichen von"

rest (dortmund) liefert ortmund

sub^{o}_{a} (top (oho)) liefert im ersten Schritt sub^{o}_{a} (o),
liefert im zweiten Schritt a

conc (dort , mund) liefert dortmund

↑

"Konkatenation"

conc (top (dortmund) , top (rest (dortmund)))

liefert im einem (parallelen) ersten Schritt
conc (d, top (ortmund)) ,

liefert im zweiten Schritt conc (d, o) ,
liefert im dritten Schritt do

einige Probleme:

- eine Maschine "kennt" nur einige elementare Funktionszeichen und die zugehörigen Funktionen; wie definiert man sich neue Funktionszeichen samt den zugehörigen Funktionen?
- die Argumente müssen zum Funktionszeichen "passen" (was sollte wohl $+(a, \sigma)$ bedeuten?)
- die Funktionen sind nicht immer definiert (was liefert rest (ε) ?)

4. Bezeichner

Bezeichner sind Namen für Objekte, insbesondere für Werte.

Bsp:

summand 1	x	wort
summand 2	y	ort
summe	z	plz

elementare Eigenschaften: - für zwei Bezeichner kann man feststellen, ob sie gleich oder ungleich sind.

- verschiedene Bezeichner sollen im allgemeinen Namen für verschiedene Objekte sein

einige Probleme: - wie kann eine "Maschine" "Bezeichner" von "Werten" unterscheiden?

- wie kann man festlegen, welcher Art (z.B. aus welchen Wertemenge) das bezeichnete Objekt sein soll?
- wie kann man insbesondere wenn man sehr viele Bezeichner benötigt, die "Übersicht über die verwendeten Bezeichner behalten?

5. Zustände

Zustände sind Abbildungen der Form

$$\Sigma: \text{Menge von Bezeichnern} \longrightarrow \text{Wertemenge}$$

Bsp:

summand1	summand2	summe
3	7	0

, d.h. hier etwa $\Sigma(\text{summe}) = 0$

x	y
3	7

ort
dortmund

ort	plz
dortmund	4600

wort	← Bezeichner
oho	← Wert

6. (berechenbare) Ausdrücke

Ausdrücke werden (induktiv) aufgebaut aus

1) Werten und Bezeichnern
2) Funktionszeichen
3) gewissen Trennzeichen:

- jeder Wert und jeder Bezeichner ist ein Ausdruck
- ist f ein n-stelliges Funktionszeichen und sind $a_1, \ldots, a_n$ Ausdrücke, so ist auch $f(a_1, \ldots, a_n)$ ein Ausdruck

Bsp:

3	sub_a^o (wort)
summand2	top (ort)
+(3, summand2)	sub_a^o (top (wort))
+(summand1, summand2)	
+(x, y)	

Ein Ausdruck zusammen mit einem Zustand, dessen Definitionsbereich alle im Ausdruck vorkommenden Bezeichner enthält, liefert einen Wert durch

1) Ersetzung der Bezeichner durch die im Zustand angegebenen Werte (dann liegt ein Wertausdruck vor)
2) Auswertung des entstandenen Wertausdrucks

Bsp

$+(x,y)$ mit $\binom{x}{3}\,\binom{y}{7}$ liefert durch Ersetzung $+(3,7)$,

Auswertung 10

$+(3, \text{summand2})$ mit $\binom{\text{summand2}}{7}$ liefert durch Ersetzung $+(3,7)$,

Auswertung 10

$+(3, \text{summand2})$ mit $\binom{\text{summand1}\,|\,\text{summand2}}{8\,|\,\_}$ liefert durch Ersetzung $+(3,7)$,

Auswertung 10

$\text{sub}^{\circ}_{a}(\text{wort})$ mit $\binom{\text{wort}}{\text{oho}}$ liefert durch Ersetzung $\text{sub}^{\circ}_{a}(\text{oho})$,

Auswertung aha

elementare Eigenschaften:

- Ausdrücke beschreiben Funktionen: zu jedem (geeigneten) Zustand wird ein (Funktions-)Wert geliefert; der Zustand beschreibt die Argumentwerte.

Bsp: i) als Wertemenge liege zugrunde $\mathbb{N} = \{0,1,2,\ldots\}$

$+(x,y)$ beschreibt Funktion vom Typ $\mathbb{N}\times\mathbb{N} \to \mathbb{N}$,

wobei an die Addition gedacht ist.

ii) als Wertemenge sei wieder $\mathbb{N} = \{0,1,2,\ldots\}$ zugrunde gelegt:

$+(\text{summand1}, \text{summand2})$ beschreibt die gleiche Funktion

+(3, summand2) beschreibt eine Funktion vom Typ $\mathbb{N} \to \mathbb{N}$, wobei der dritte Nachfolger des von summand2 bezeichneten Arguments genommen ist.

iii) Als Wertemengen für Argumente sei $\mathbb{N}$, für Ergebnisse sei {FALSE, TRUE} zugrunde gelegt

$\leq(x,y)$ beschreibt Funktion vom Typ $\mathbb{N} \times \mathbb{N} \to$ {FALSE, TRUE}

einige Probleme:

- wie entwickelt man zu einer gegebenen Funktion einen Ausdruck, der sie beschreibt?
- wie kann man überprüfen, ob ein gegebener Ausdruck eine gegebene Funktion beschreibt?
- wie kann eine "Maschine" "Werte", "Bezeichner" und "Funktionszeichen" unterscheiden?

7. Zustandsüberführungen

Überführung eines Zustands ζ in einen Folgezustand ζ', $\zeta \vdash \zeta'$.

Bsp $\begin{pmatrix} x \\ 3 \end{pmatrix} \begin{pmatrix} y \\ 7 \end{pmatrix} \vdash \begin{pmatrix} x \\ 3 \end{pmatrix} \begin{pmatrix} y \\ 7 \end{pmatrix} \begin{pmatrix} z \\ 10 \end{pmatrix}$

$\begin{pmatrix} x \\ 3 \end{pmatrix} \begin{pmatrix} y \\ 7 \end{pmatrix} \vdash \begin{pmatrix} x \\ 10 \end{pmatrix}$

$\begin{pmatrix} x \\ 3 \end{pmatrix} \begin{pmatrix} y \\ 7 \end{pmatrix} \begin{pmatrix} z \\ 0 \end{pmatrix} \vdash \begin{pmatrix} x \\ 3 \end{pmatrix} \begin{pmatrix} y \\ 7 \end{pmatrix} \begin{pmatrix} z \\ 10 \end{pmatrix}$

$\begin{pmatrix} x \\ 3 \end{pmatrix} \begin{pmatrix} y \\ 7 \end{pmatrix} \vdash \begin{pmatrix} z \\ 10 \end{pmatrix}$

8. Anweisungen

Anweisungen beschreiben (Klassen von) Zustandsüberführungen.

Eine typische einfache Anweisung ist etwa:

> bestimme den von einem Ausdruck mit einem (Ausgangs-) Zustand gelieferten Wert und bezeichne diesen!

Eine Anweisung zusammen mit einem (geeigneten) Zustand bestimmt einen (Folge-) Zustand, manchmal auch eine endliche oder gar abzählbar unendliche Folge von Zuständen.

Bsp: "bestimme den durch $+(x,y)$ (mit einem Ausgangs-Zustand) gelieferten Wert und bezeichne ihn mit dem neuen Bezeichner z; die durch x und y bezeichneten Werte bleiben unverändert"

Diese Anweisung überführt z.B

$$\begin{pmatrix} x & y \\ 3 & 7 \end{pmatrix} \text{ in } \begin{pmatrix} x & y & z \\ 3 & 7 & 10 \end{pmatrix},$$

$$\begin{pmatrix} x & y \\ 4 & 8 \end{pmatrix} \text{ in } \begin{pmatrix} x & y & z \\ 4 & 8 & 12 \end{pmatrix} \quad \text{usw.}$$

einige Probleme:

- eine Maschine "kennt" nur einige elementare Anweisungen; wie definiert man sich neue zusammengesetzte Anweisungen?

- wie erfährt man zu gegebenen Zustandsüberführungen eine (zusammengesetzte) Anweisung, auch Programm genannt, die diese beschreibt?

- wie kann man überprüfen, ob eine gegebene Anweisung gegebene Zustandsüberführungen beschreibt?

- die Zustandsüberführung ist nicht immer definiert, oder die Anweisung beschreibt (unbeabsichtigt) eine unendliche Folge von Zuständen.

9. Maschinen

(gedachte) Geräte, die die bisher genannten Aufgaben durchführen können:

- zu jedem elementaren Funktionszeichen benötigen wir eine Funktionsmaschine,
  z.B. für (das Zeichen) + eine "Additionsmaschine"
  für (das Zeichen) $succ_0$ eine "Such- und Ersetzungs-Maschine"

- zur Analyse und Auswertung von Ausdrücken benötigen wir eine Auswertungsmaschine, die sich dabei der Funktionsmaschinen bedient.

- zur Analyse und Durchführung von Anweisungen benötigen wir eine Rechenmaschine, die sich dafür die Anweisungsmaschine bedient.

10. Programmiersprache, ihre Syntax und Semantik

Eine Programmiersprache dient dazu, Funktionen bzw. Zustandsüberführungen durch Ausdrücke bzw. Anweisungen zu beschreiben.

Die Syntax legt fest, welche Zeichenfolgen zum Aufschreiben von Werten, Ausdrücken, Anweisungen usw. erlaubt sind.

Die Semantik legt die Bedeutung erlaubter Zeichenfolgen durch Angabe der Maschinen fest. Einige Formen der Semantik-Definition benutzen nicht "Maschinen" in irgendeinem technischen Sinn, sondern definieren die bezeichneten Funktionen bzw. Klassen von Zustandsüberführungen mit den Mitteln der Mathematik oder Logik.

11. Programmieren

Entwicklung eines Ausdrucks, bzw. einer (zusammengesetzten) Anweisung (d.h. eines Programms), die ein vorgegebenes Problem formal beschreiben und damit der maschinellen Bearbeitung zugänglich machen.

Die Entwicklung beinhaltet im allgemeinen die folgenden Stufen:

- Erfassung und Verdeutlichung des "Problems"

  (wobei sich häufig das ursprünglich gemeinte "Problem" erheblich wandeln wird)

- genaue Festlegung der Eigenschaften des zu entwickelnden Programms; insbesondere:

  welche Variablen bzw. welche Zustandsüberführungen soll beschrieben werden?

  welche Betriebsmittel (Zeit, Speicher) der verwendeten Maschine dürfen (maximal) verbraucht werden?

- Evaluierung eines beschreibenden Ausdrucks, bzw. eines beschreibenden Programms

- Nachweis der festgelegten Eigenschaften

12 Prozesse

Ein Prozess ist die Durchführung (durch eine Maschine) eines Programms.

Prozesse können u.a. gestartet, unterbrochen, beendet werden.
Mehrere Prozesse können gleichzeitig ablaufen und untereinander kommunizieren.
Mehrere Prozesse können gleichzeitig das gleiche Programm durchführen.

In dieser Vorlesung behandeln wir Elemente der sequentiellen Programmierung bei der wir im allgemeinen nur an einem einzigen Prozess denken.
Bei der parallelen Programmierung muß insbesondere das Zusammenspiel mehrerer Prozesse beschrieben werden. Dieser schwierige, aber wichtige Gebiet bleibt späteren Vorlesungen überlassen.

3. Einige Ebenen der Programmierung

Die "Ebene" einer Programmiersprache wird beschrieben durch

- die elementaren Funktionszeichen und elementaren Anweisungen
- den Regeln, mit denen neue Funktionszeichen und neue zusammengesetzte Anweisungen definiert werden können

(bzw. jeweils den entsprechenden Maschinen)

Diese Gegebenheiten können z.B.

- auf die Erfordernisse der zu beschreibenden Probleme zugeschnitten sein (problemorientiert);
- im Hinblick auf die (jeweils, heute) vorhandenen tatsächlichen Geräte entworfen sein (maschinenorientiert);
- Programmierung durch Entwicklung von Ausdrücken vorsehen (applikativ);
- Programmierung durch Entwicklung von Anweisungen vorsehen (prozedural).
- Programmierung durch Angabe der Eigenschaften der gewünschten Werte bzw. Zustandsübergänge vorsehen (prädikativ)

Wir wollen dazu ein Beispiel in mehreren Ebenen behandeln:

Problem:

Es soll die Summe zweier natürlicher Zahlen bestimmt werden können.

(wobei als Beispiel stets Summe von 3 und 2 betrachtet werde).

①

Voraus: "Auswertungsmaschine" kennt

als Wertemenge: $\{0, 1, 2, \ldots\}$

als Funktionszeichen: + und Additions-Maschine

Bsp + (3, 2) liefert durch Auswertung 5

②

Voraus: Auswertungsmaschine kennt nur

als Wertemenge $\{0, 1, 2, \ldots\}$

als Funktionszeichen

INC und "Nachfolger-Maschine" (increase)

DEC und "Vorgänger-Maschine" (decrease)

pos mit "Nulltest-Maschine"

[pos(0) liefert FALSE,
pos(1), pos(2), ... liefern TRUE]

if then else fi "(Wert-) Fallunterscheidung

Verfahren: es gilt in der Arithmetik:

für alle natürlichen Zahlen a, b: $a + b = \begin{cases} (a+1) + (b-1) & \text{falls } b > 0 \\ a & \text{falls } b = 0 \end{cases}$

definiere neues Funktionszeichen add durch

add(x,y):
if pos(y) then add(INC(x), DEC(y)) else x fi

Bsp

add(x,y) liefert mit Zustand $\binom{x}{3} \mid \binom{y}{2}$:

durch Ersetzung der Bezeichner: add(3,2)
durch Einsetzen der Definition: if pos(2) then add(INC(3), DEC(2)) else 3 fi
durch Auswertung : add(4,1)
durch Einsetzen der Definition: if pos(1) then add(INC(4), DEC(1)) else 4 fi
durch Auswertung : add(5,0)
durch Einsetzen der Definition: if pos(0) then add(INC(5), DEC(0)) else 5 fi
durch Auswertung : 5

(3)

Man betrachte die oben im Laufe der Rechnung auftretenden Argumentpaare für add:

3	2
4	1
5	0

Diese könnte man auch explizit bezeichnen:

3 durch u_0, 2 durch v_0

4 durch u_1, 1 durch v_1

5 durch u_2, 0 durch v_2;

allgemein benötigte man mit zweitem Argument b insgesamt $2\cdot(b+1)$ viele Bezeichner (d.h. die Anzahl hängt vom zweiten Argument ab!):

u_0 v_0

u_1 v_1

$\vdots$ $\vdots$

u_i v_i

u_{i+1} v_{i+1}

$\vdots$ $\vdots$

u_b v_b,

wobei u_{i+1} bezeichnet Wert von INC(u_i)

v_{i+1} bezeichnet Wert von DEC(v_i).

Nachdem man jeweils INC(u_i) bzw. DEC(v_i) ausgewertet hat, benötigt man die durch u_i bzw. v_i bezeichneten Werte nicht mehr, d.h. man benötigt keine Bezeichner mehr für sie. Also könnte man die alten Bezeichner wiederverwenden, nämlich zur Bezeichnung der neuen Werte.

Dann kommt man mit zwei Bezeichnern u, v aus (unabhängig von der Größe vom zweiten Argument!):

[anfang] bezeichne 3 durch u , 2 durch v

[schrittweise] bezeichne Wert von INC(u) wieder durch u, ( ) wie

bezeichne Wert von DEC(v) wieder durch v

oder etwas formaler:

[anfang] (u, v) := (x, y);

[schrittweise] WHILE pos(v) DO (u,v) := (INC(u), DEC(v)) END;

[Ergebniswert] add := u

↑ Bezeichner für errechneten Wert (u,v wird nicht mehr gebraucht)

Voraus: Rechenmaschine kennt

Zuweisung von Werten an Bezeichner

Bildung von repetitiven Anweisungen (WHILE ... DO ... END)

Auswertungsmaschine kennt pos, INC, DEC

Bsp

$$\begin{pmatrix} x & y \\ 3 & 2 \end{pmatrix}$$

vermöge [anfang] $\vdash$ $\begin{pmatrix} x & y & u & v \\ 3 & 2 & 3 & 2 \end{pmatrix}$

vermöge [schrittweise] $\vdash$ $\begin{pmatrix} x & y & u & v \\ 3 & 2 & 4 & 1 \end{pmatrix}$

$\vdash$ $\begin{pmatrix} x & y & u & v \\ 3 & 2 & 5 & 0 \end{pmatrix}$

vermöge [Ergebniswert] $\vdash$ $\begin{pmatrix} x & y & add \\ 3 & 2 & 5 \end{pmatrix}$

MODULA-2 – Formen für ①, ②, ③

zu ① + ist bekannt für Werte vom Typ CARDINAL, d.h. für (natürliche) Kardinalzahlen, wie man die natürlichen Zahlen auch nennt.

Bsp: 3+2

zu ② Wegen ① eigentlich unnötig, aber trotzdem als Beispiel:

Vereinbarung (Definition):

```
PROCEDURE add (x: CARDINAL; y: CARDINAL): CARDINAL;
BEGIN IF y > 0 THEN RETURN add (INC(x), DEC(y))
                  ELSE RETURN x
      END
END add;
```

Kopf

Rumpf

werden als bekannt vorausgesetzt

Aufruf: add (3,2) innerhalb eines Ausdrucks

zu ③ Ebenfalls wegen ① eigentlich unnötig, aber trotzdem als Beispiel:

Vereinbarung (Definition):

```
PROCEDURE add (x: CARDINAL; y: CARDINAL): CARDINAL;
    VAR u: CARDINAL; v: CARDINAL;
    BEGIN u := x; v := y;
          WHILE v > 0 DO u := INC(u); v := DEC(v) END;
          RETURN u
    END add;
```

Aufruf: add (3,2)

Wir wollen zeigen, daß diese Vereinbarung korrekt ist, genauer daß

add(x,y) den Zustand $\begin{pmatrix} x \\ a \end{pmatrix} \begin{pmatrix} y \\ b \end{pmatrix}$ in den Zustand $\begin{pmatrix} * \\ a \end{pmatrix} \begin{pmatrix} y \\ b \end{pmatrix} \begin{pmatrix} add \\ a+b \end{pmatrix}$

überführt für alle natürlichen Zahlen a, b.

kurz: add = x+y f.alle erlaubten Werte für x, y

Dazu versehen wir unsere Anweisungen von rückwärts mit Aussagen, die für die jeweils möglichen Zustände gelten sollen:

```
(* x∈ℕ und y∈ℕ *)
u := x ; v := y ;
(* u+v = x+y *)
WHILE v > 0 DO
                (* u+v = x+y *)

                u := INC(u) ; v := DEC(v)
                (* u+v = x+y *)
                END ;

(* u+v = x+y und v = 0 *)
(* u = x+y *)
RETURN u ;
(* add = x+y *)
```

Man muß nun nachprüfen, ob die Aussage vor einer Anweisung die Aussagen nach einer Anweisung impliziert, d.h. genauer für:

für (*Vorb*) S (*Nachb*) gilt:

jeder Zustand „z'", für den die Aussage Vorb gilt, wird durch die Anweisung S in einen Folgezustand z' überführt, so daß für z' die Aussage Nachb gilt, oder S liefert überhaupt keinen "wohldefinierten Folgezustand."

Induktiv ist das hier einfach.

Schließlich müssen wir zeigen, daß die repetitive Anweisung einen Folgezustand liefert; das folgt

wegen der Aussage $y \geq 0$, bzw. $v \geq 0$ zu Beginn

weil die Folge der durch v bezeichnete Werte streng monoton fallend ist;

wegen der Arithmetik!

④ Wir beginnen nun auf der Ebene der Codierung von Zahlen durch Worte über dem Alphabet $\{0,1\}$:

Jede natürliche Zahl n läßt sich (bis auf führende Nullen) eindeutig durch ein Wort $n_k \ldots n_0 \in \{0,1\}^*$ darstellen:

$$n = \sum_{i=0}^{k} n_i \cdot 2^i$$

Dualdarstellung

Bsp:
$3 = 1 \cdot 2^1 + 1 \cdot 2^0$ — 11
$2 = 1 \cdot 2^1 + 0 \cdot 2^0$ — 10
$5 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$ — 101

Die Addition läßt sich stellenweise wie im Dezimalsystem durchführen

Bsp:

x	11	Argumente
y	10	
u	100	Überlauf
binadd	101	Ergebnis

Wir nehmen an, daß folgende Funktionszeichen bekannt sind:

letzt liefert rechtes Zeichen eines Elementes aus $\{0,1\}^*$;
letzt(ε) liefere 0!

anfang liefert den Anfang eines Elementes aus $\{0,1\}^*$ bis auf das rechte Zeichen;
anfang(ε) liefere ε!

3.10

aufg Liefert die Verknüpfung von einem Zeichen aus $\{0,1\}$ und einem Wort aus $\{0,1\}^*$

vas "Volladdiersumme" : $\{0,1\} \times \{0,1\} \times \{0,1\} \to \{0,1\}$

vaü "Volladdierüberlauf" : $\{0,1\} \times \{0,1\} \times \{0,1\} \to \{0,1\}$

Liefert stellenweise Summe und Überlauf:

a	b	ü	vas	vaü
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

3.14

add (x, y):

[anfangs] $(s, \ddot{u}) := (\varepsilon, 0)$;

[schrittweise] WHILE $(x \neq \varepsilon)$ OR $(y \neq \varepsilon)$ OR $(\ddot{u} = 1)$ DO

$(s, \ddot{u}, x, y) :=$

(anfüg (vas (letzt (x), letzt (y), ü) , s) ,

vaü (letzt (x), letzt (y) , ü) ,

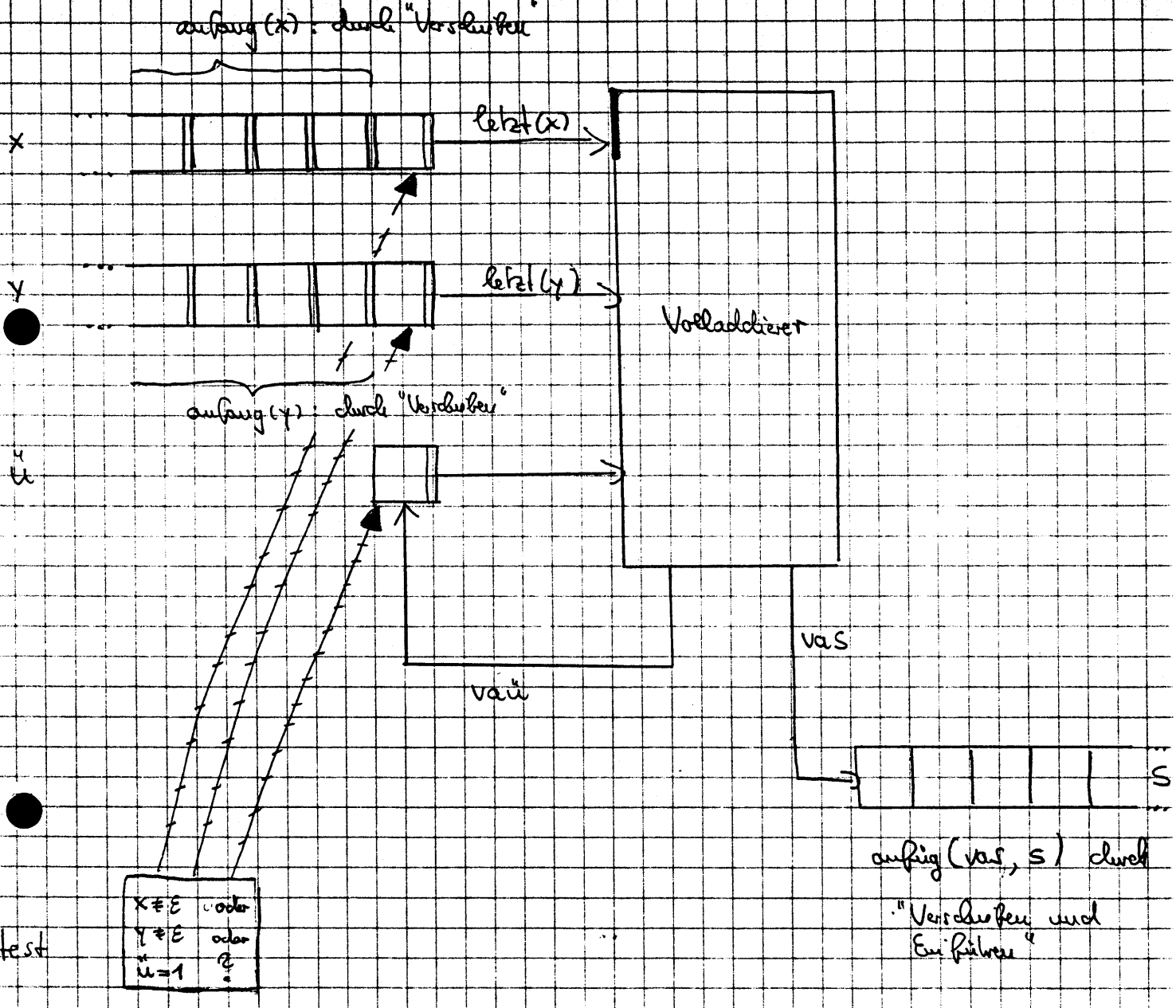
anfang (x) ,

anfang (y));

END;

[Ergebniswert] add := s

Die zu den Funktionsweisen gehörenden Maschinen kann man nun verbinden:



Solch eine Maschine arbeitet "getaktet":

Die Speicherbausteine ☐ übernehmen den neuen Wert jeweils erst beim Eintreffen des Taktimpulses.

4. Programmiersprachen

aus §2.10: eine Programmiersprache dient dazu, Funktionen bzw. Zustandsüberführungen durch Ausdrücke bzw. Anweisungen zu beschreiben.

Funktionen sind gegeben durch Angabe der

- Werteverläufe für Argumente und Ergebnisse
- Zuordnung von Argumenten und Ergebnissen,

d.h. eine Programmiersprache muß erlauben

- Dinge des Handelns (Werte, Objekte)
- Handlungsabläufe (Zuordnungen)

präzise, endlich und elementar ausführbar zu beschreiben.

aus §2.11: Programmieren macht ein vorgegebenes Problem der maschinellen Bearbeitung zugänglich.

Eine Programmiersprache

- wird vom Menschen zur formalen Beschreibung von Funktionen bzw. Klassen von Zustandsüberführungen benutzt,
- muß für eine "Maschine" verständlich sein.

Die von uns benutzte Programmiersprache MODULA-2 (bzw. ihre Semantik) soll durch eine gedachte "MODULA-2-Maschine" beschrieben (bzw. beschrieben) werden.

Andererseits benutzen wir wirkliche Rechnermaschinen des Herstellers SUN.

Diese Maschinen arbeiten auf verschiedenen Ebenen; die Distanz muß nun aber überbrückt werden.

Dies geschieht insbesondere durch einen Übersetzer (Compiler), der jedes MODULA-2-Programm in ein (äquivalentes) SUN xyz-Programm übersetzt; dieser Übersetzer ist selbst ein Programm mit

Argument-Wertemenge: Zeichenfolgen über dem Alphabet der Sprache MODULA-2

Ergebnis-Wertemenge: - SUN xyz - Programme
- Fehlermeldungen (falls die Zeichenfolge kein syntaktisch richtiges Programm war)

Diese Beschreibung ist noch recht ungenau:

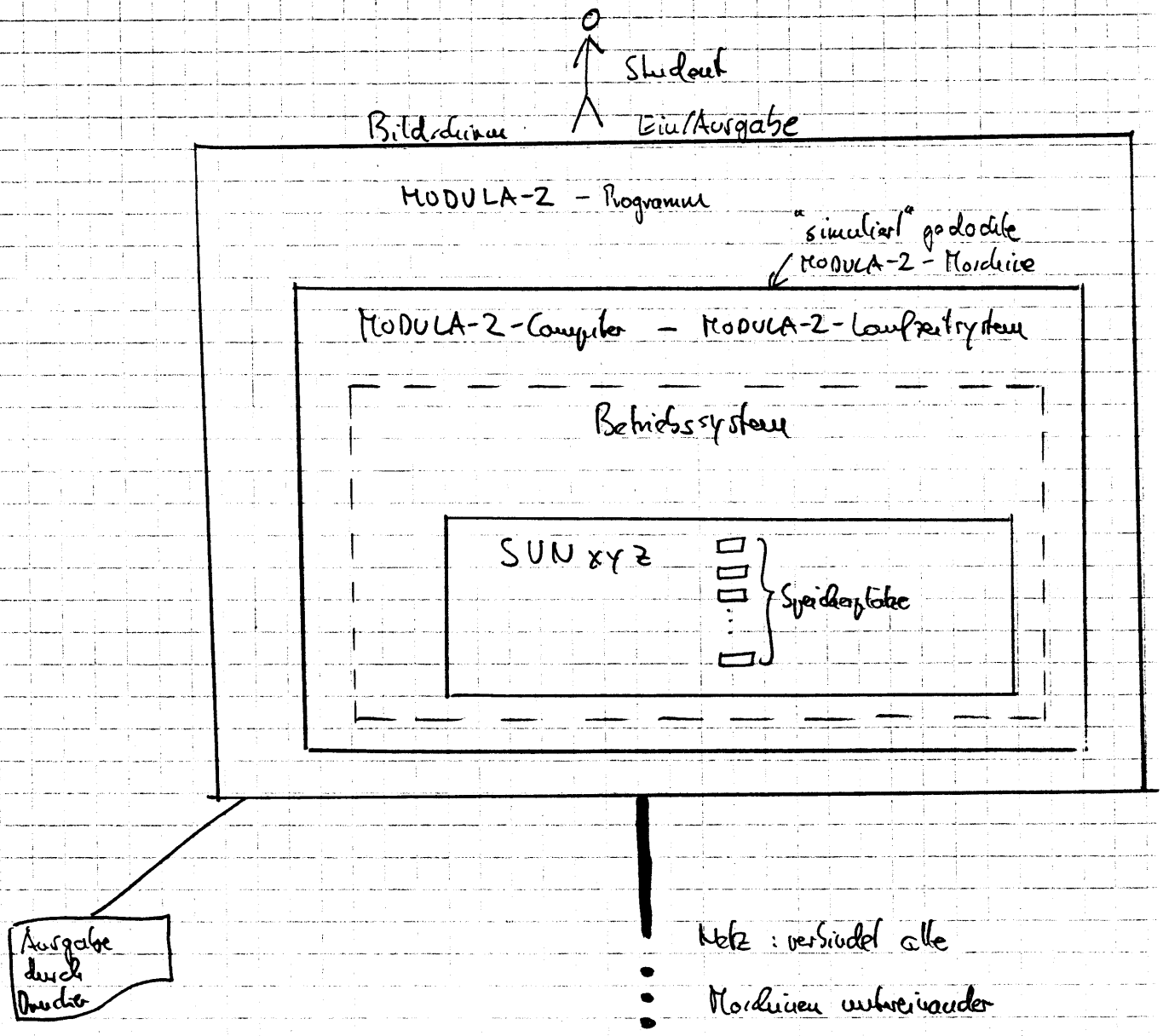
um ein MODULA-2-Programm auf einer SUN xyz auszuführen, braucht man u.a.

MODULA-2-Compiler
MODULA-2-Laufzeitsystem
Betriebssystem

} alles Programme; sie sind Teil der allgemeinen "Software"

Das Gesagte soll noch durch ein Bild verdeutlicht werden:

"Informatik ist eine schwarze Kunst –
sie beherrscht die Denkweise des schwarzen Kastens
(black box)



<u>software:</u> Programme in einer Programmiersprache (gedachte "Maschinen")

<u>hardware:</u> <u>wirkliche</u> Maschinen mit Ein/Ausgabe-Geräten (können aufgefaßt werden als technisch realisierte "Programme")

alte Programmierer-Weisheit:

" Eine Rechenmaschine tut genau das, was man ihr sagt "

1. Interpretation

man: MODULA-2 - Programmierer

Rechenmaschine: gedachte MODULA-2-Maschine, d.h.

hardware (SUN xyz) und allgemeine software (insb. Compiler, Laufzeitsystem, Betriebssystem)

sagen: Eingabe eines MODULA-2-Programmes

2. Interpretation

man: Systemprogrammierer-Team

Compilerprogrammierer-Team

Rechnerbetriebsgruppe

⋮

MODULA-2-Programmierer

Rechenmaschine: hardware

sagen: siehe Babylon

Gemäß dieser 2. Interpretation müssten Sie darauf gefaßt sein, daß die Rechenmaschine manchmal etwas tut, was weder Sie noch irgendjemand anderer auf Anhieb versteht.

In dieser Vorlesung bezeichnen wir in allgemeinen die 1. Interpretation.

5 Die Grenzen des Berechenbaren

aus §2.10: eine Programmiersprache dient dazu, Funktionen bzw. Zustandsüberführungen zu beschreiben.

"Zustandsüberführungen" kann man auch als Funktionen deuten, z.B. kann man die Zustandsüberführungen

$$\left\{ \begin{pmatrix} x \\ a \end{pmatrix} \begin{pmatrix} y \\ b \end{pmatrix} \vdash \begin{pmatrix} x \\ a \end{pmatrix} \begin{pmatrix} y \\ b \end{pmatrix} \begin{pmatrix} z \\ a+b \end{pmatrix} \;\middle|\; a, b \text{ natürliche Zahlen} \right\}$$

als Funktion der Addition auf natürlichen Zahlen (mit Verabredungen über Bezeichnern) deuten.

<u>in diesem Abschnitt</u> reden wir dann also davon, daß

jedes Programm P (in MODULA-2 gültige Zeichenfolge)

eine Funktion $\sigma(P)$ beschreibt (oder <u>berechnet</u>).

Zur Verwirklichung beschreiben wir Argument-Werte-mengen und Ergebnis-Werte-menge ebenfalls als Menge von Zeichenfolgen

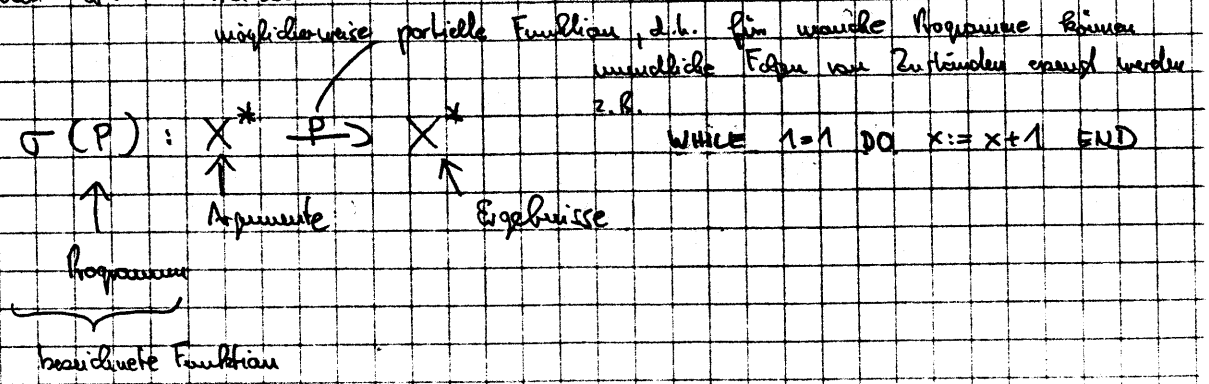
(z.B. beschreibt man zwei Argumente "drei" und "zwei" etwa durch <u>eine</u> Zeichenfolge:

$$< 2 , 3 >$$

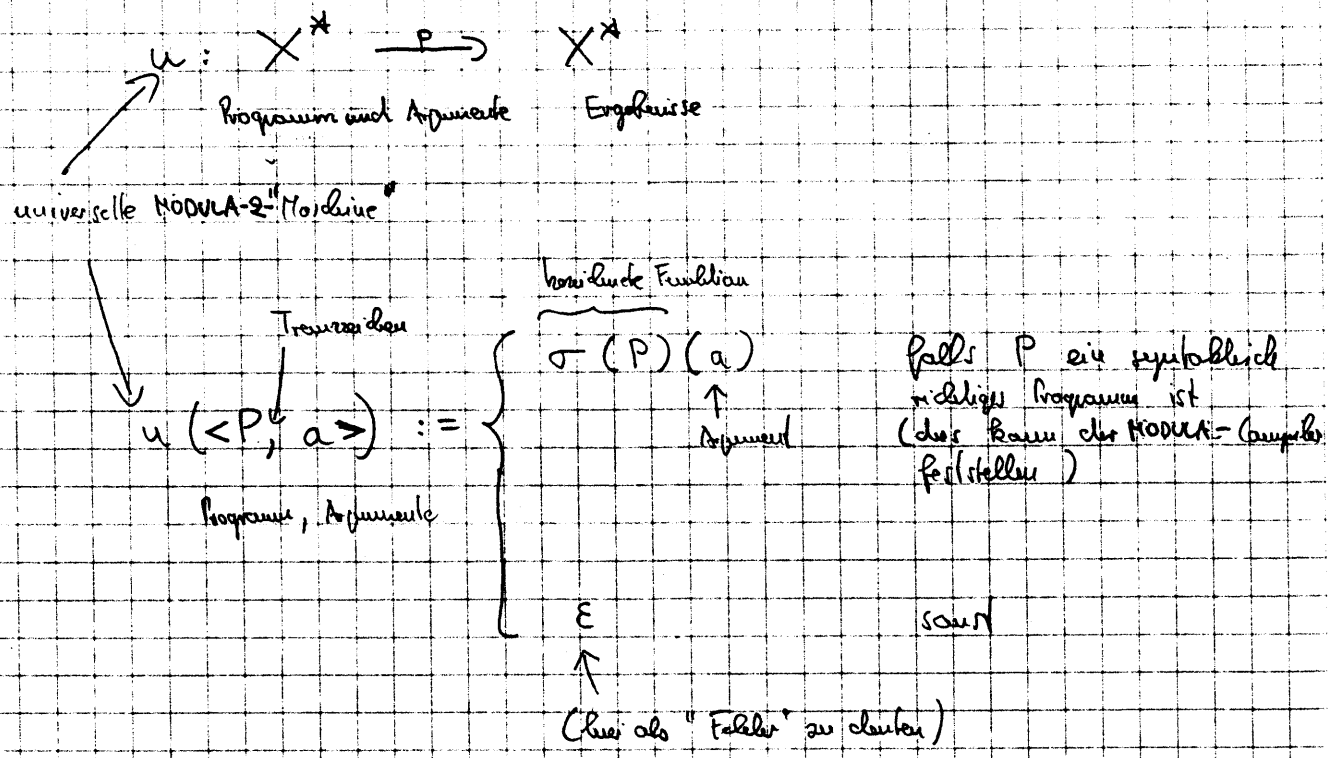
)

Die Menge der Zeichenfolgen sei mit X^* bezeichnet.

Dann können wir schreiben



Die Arbeitsweise der gedachten MODULA-2-Maschine läßt sich dann wie folgt angeben:



Wir wollen nun zwei Klassen von Funktionen betrachten:

$$\mathbb{P}_{\text{MODULA}} := \{ \sigma(P) \mid P \text{ MODULA-Programm} \}$$

$$\mathbb{P}_{\text{berechenbar}} := \{ f \mid f: X^* \to X^* \text{ ; „es gibt eine Maschine, die } f \text{ berechnet“} \}$$

Unter gewissen Idealisierungen (z.B. Argumente dürfen beliebig groß sein; beliebig lange (aber endliche) Rechenzeit; allgemein: unter Vernachlässigung der Beschränkungen der „wirklichen Zeit“ und des „wirklichen Raums“) ist man davon überzeugt (dies ist der Kern der Church'schen These), daß gilt:

$$\mathbb{P}_{\text{MODULA}} = \mathbb{P}_{\text{berechenbar}} \quad , \tag{2}$$

d.h. MODULA-2 ist eine universelle Sprache in dem Sinne, daß jede „berechenbare Funktion“ durch ein MODULA-2-Programm beschrieben werden kann.

Die obige Funktion u nun ist nach dem bisher Gesagten durch eine Maschine berechenbar, d.h. selbst durch ein MODULA-2-Programm beschreibbar (P_u)

Wir wollen eine Aussage plausibel machen (C wird später in der Theoretischen Informatik alles präziser gemacht), die die Grenze des Berechenbaren markiert.

Aussage (Unentscheidbarkeit des Halteproblems)

Es gibt kein MODULA-2 – Programm, das folgendes leistet:

Argumente: $< P, a >$

↑ Programm ↑ Argument

Ergebnis: $\begin{cases} 1 & \text{falls } \sigma(P)(a) \text{ ist } \underline{\text{definiert}}, \text{ d.h. die MODULA-2 – Maschine, gestartet mit Programm } P \text{ und Argument } a, \text{ liefert ein Ergebnis} \\ 0 & \text{sonst} \end{cases}$

d.h. die Funktion $h: X^* \to X^*$ mit

$$h(< P, a >) := \begin{cases} 1 & \sigma(P)(a) \text{ definiert} \\ 0 & \text{sonst} \end{cases} \qquad (3)$$

ist nicht berechenbar.

Beweisidee

Wir führen einen indirekten Beweis.

Annahme: Es gibt ein MODULA-2 – Programm P_h, das h berechnet.

5.5

wir definieren eine weitere Funktion [?]

$$diag : X^* \to X^*$$

(5)

$$diag(P) := \begin{cases} conc(!, \sigma(P)(P)) & \text{falls } \sigma(P)(P) \text{ definiert} \\ ! & \text{falls } \sigma(P)(P) \text{ nicht definiert} \end{cases}$$

! irgendein gewähltes Zeichen

Erläuterung zur Definition von diog:

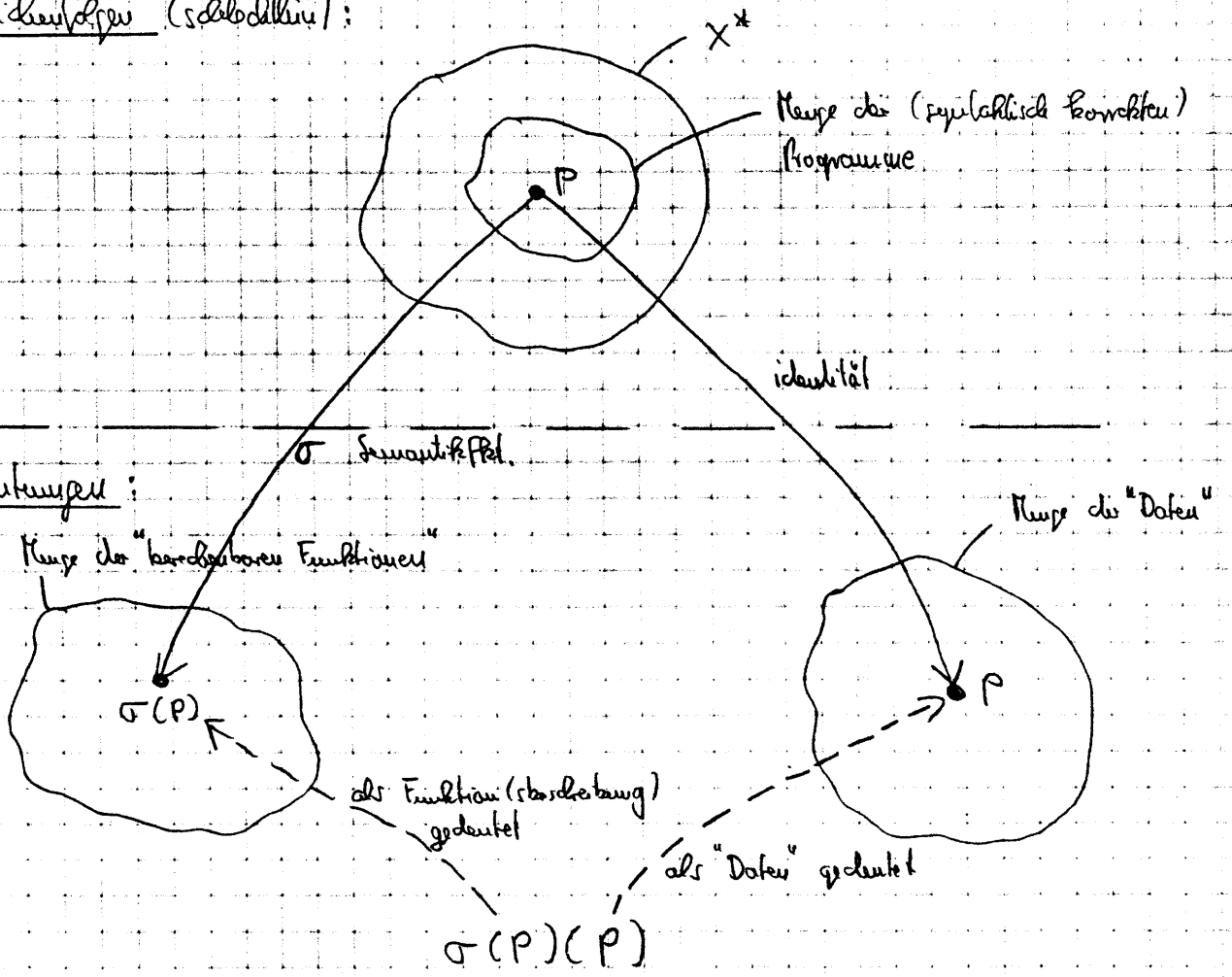
Die behandelten Objekte sind alles Zeichenfolgen, d.h. Elemente aus X^*;
manche Zeichenfolgen sind (syntaktisch korrekte) MINOR-2-Programme.

Zeichenfolgen können verschieden gedeutet werden, und zwar

- als "Programme", d.h. Beschreibungen von Funktionen
- als "Daten", d.h. Argumente für Funktionen.

In der Definition von diog werden beide Deutungen benutzt.

Zeichenfolgen (syntaktisch):



diag ist berechenbar vermöge folgendem Verfahren:

1) berechne P_R zu "$\sigma(P)(P)$ definiert" zu entscheiden

$\Updownarrow$ (3)

$h(\langle P, P\rangle) = 1$

$\Updownarrow$

P_R gestartet mit $\langle P, P\rangle$, befolgt 1

2) <u>falls</u> $\sigma(P)(P)$ definiert ist

<u>dann</u>

berechne P_u zu $\sigma(P)(P)$ zu berechnen

$\|$ (2)

$u(\langle P, P\rangle)$

und folge wenn ein "!" zu

<u>andernfalls</u>

befolge ! als Ergebnis

[beachte den <u>Trick</u>: $diag(P) \neq \sigma(P)(P)$ für <u>alle</u> Programme P]

Gemäß (2) gibt es dann ein MODULA-2 – Programm für diag, P_{diag}, mit

$\sigma(P_{diag}) = diag$

Wir betrachten nun $\underbrace{\sigma(P_{diag})}_{=diag}(P_{diag}) = diag(P_{diag})$:

Da diag eine <u>totale</u> Funktion ist, ist $\sigma(P_{diag})(P_{diag})$ definiert und

in (5) kommt der erste Fall zum Tragen:

$$diag(P_{diag}) = conc(!, \overset{\perp}{\sigma}(P_{diag})(P_{diag}))$$

$$= conc(!, diag(P_{diag}))$$

was einen Widerspruch ergibt [$x \neq !x$ für alle endlichen Zeichenfolgen]

ein anderes Argument:

X^* ist <u>abzählbar</u> unendlich, d.h.

X^* läßt sich mit den natürlichen Zahlen durchnumerieren:

ε	0
a_1	1
$\vdots$	$\vdots$
a_n	n
a_1a_1	$n+1$
a_1a_2	$n+2$
$\vdots$	$\vdots$
a_na_n	$n+n^2$
$a_1a_1a_1$	$n+n^2+1$
$\vdots$	$\vdots$

Aber: es gibt <u>überabzählbar</u> viele Funktionen der Art

$$f: X^* \to X^*$$

(denn: alle abzählbar unendlichen Folgen über $\{0,1\} \subset X$, d.h. $\mathbb{R}$ darstellbar)

6. Einige historische Stichworte

frühe mathematische Wurzeln:

Babylonier (ca. 2000 v. Chr.) : Rechnen im Stellenwertsystem zur Basis 60

Griechen (Euklid, ca. 300 v. Chr.) : Euklid'scher Algorithmus zur Bestimmung des g.g.T zweier natürlicher Zahlen

Araber (Al Chwarizmi, 800 u. Chr.) : Verfahren zur Behandlung algebraischer Relationen

Adam Riese (1492-1559) : Verfahren der Arithmetik

Leibniz (1646-1710) : Rechnen im Stellenwertsystem zur Basis 2

Maschinen:

Schickard : Addiermaschine (1623)

Pascal : Addiermaschine (1645)

Leibniz : Maschine für vier Grundrechenarten (1674) großer Entwurf einer "Rechenmaschine"

Babbage : Analytic Engine (1850), Ansätze einer programmgesteuerten Maschine

Zuse : Z3 (1941), erste funktionsfähige programmgesteuerte Rechenmaschine

Aiken : Mark I Computer (1944)

Eckert / Mauchly : ENIAC (1946), erste mit Röhren arbeitende Rechenmaschine

v. Neumann : Princeton-Maschine (1948), Prototyp der heutigen universellen, speicherprogrammierten Rechenanlagen

1. Generation 1953-58 : Röhren, Kernspeicher, Magnetbänder

2. Generation 58-66 : Transistoren, Kernspeicher, Platten

3. Generation 66-74 : Großschaltkreise, Dialogstationen, Halbleiterspeicher

4. Generation 74-82 : hoch integrierte Schaltkreise

5. Generation 82- : laufende Entwicklung: Höchstintegration, "wissensbasierte" Programmierung, Netzwerke

künstliche Sprachen (der Logik)

Lullus (ca. 1300) : ars magna, "allgemeines Verfahren zum Auffinden aller Wahrheiten"

Leibniz (1646-1710) : zeichenmäßige Darstellung von Inhalten

Boole (1815-1946) : Algebraisierung der Aussagenlogik

Frege (1848-1925) : "Begriffsschrift" (1879)

Russel / Whitehead : "Principia mathematica (1913)

mathematische Präzisierungen des Begriffs "berechenbare Funktion" durch

Herbrand (1932), Gödel (1934), Kleene (1936) : Gleichungskalkül

Turing (1937) : Turingmaschine

Post (1936), Markow (1947) : Zeichenfolgen-Manipulations-Maschine

Church (1936) : Lambda-Kalkül

Programmiersprachen

Plankalkül — Zuse (1948)

FORTRAN — Backus u.a. bei IBM (1957)

ALGOL 60
algorithmic language — Revised Report on the Algorithmic Language ALGOL 60 (1960)

Autoren: J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden, M. Woodger

Algol 60 war der Ausgangspunkt für eine ganze Familie von Programmiersprachen; wichtige Vertreter sind

ALGOL 68 — van Wijngaarden u.a.

PASCAL (1968 entworfen, 1970 Lauffähig, 1973 revidiert)
N. Wirth, Zürich

SIMULA (1965-67) O.J. Dahl, B. Myhrhaug, K. Nygaard

→ MODULA-2 (1979-1981 als Weiterentwicklung von PASCAL entstanden) N. Wirth, Zürich

MODULA-3

OBERON

⋮

andere bekannte Sprachen:

COBOL	common business oriented language	(1960)
PL/1	programming language 1	(1965)
LISP	McCarthy (1959): Ausdruck-orientiert, enge Verbindung zum Lambda-Kalkül	
ADA	1980; revidiert 1983; wird weiter laufende Entwicklung, vom US-Verteidigungsministerium gefördert	
PROLOG	"logische Programmiersprache", prädikative Sprache	
C	weit verbreitet in der Welt des Betriebssystems UNIX (gehört in die ALGOL 60-Familie)	
C++	objektorientierte Weiterentwicklung (Stroustrup 1986)	
SMALLTALK	objektorientierte Sprache; greift SIMULA-Konzepte auf (Goldberg/Robson 89)	
APL	für technisch-naturwissenschaftliche Anwendungen	
ML		

:

insgesamt: viele Hundert Sprachen,

Turm von Babylon

7. Euklid'scher Algorithmus zur Bestimmung des g.g.T.

– ein Beispielprogramm in MODULA-2 –

Aufgabe: Entwickle ein MODULA-2-Programm zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen.

Verdeutlichung der Aufgabe:

a, b, x, y, z, w stehe für natürliche Zahlen, d.h. Elemente aus $\mathbb{N} = \{0, 1, 2, \ldots\}$

x te y (teilt) : gdw $y \neq 0$ und ex. a mit $y = a \cdot x$

[manchmal auch $y = 0$ zugelassen: dann gilt insbesondere x te 0 f. alle x]

$$ggt(x,y) = \text{Maximum}_{(\mathbb{N}, \leq)} \underbrace{\{ z \mid z \text{ te } x \text{ und } z \text{ te } y \}}_{:= G_{x,y}}$$

erschöpfende Fallunterscheidung für x, y:

Fall 1: $x = 0$ oder $y = 0$: $G_{x,y} = \emptyset$, d.h. $\text{Maximum}_{(\mathbb{N}, \leq)}$ undefiniert!

Fall 2: $x \neq 0$ und $y \neq 0$: $1 \in G_{x,y} \subset \{0, \ldots, \max(x,y)\}$

d.h. $G_{x,y}$ nichtleer, endlich und $\text{Maximum}_{(\mathbb{N}, \leq)}$ definiert.

Prädikative Form der Aufgabe, Spezifikation

```
PROCEDURE ggt (x: CARDINAL; y: CARDINAL): CARDINAL;
```

Kopf einer Funktionsprozedur in MODULA-2

```
(* x > 0 und y > 0 *)
```

Kommentar in MODULA-2

```
BEGIN
    RETURN Maximum {z | z te x und z te y}
END ggt;
```

Rumpf der Funktionsprozedur

$\underbrace{\qquad\qquad}_{(\mathbb{N}, \leq)}$

ggf. so wählt in MODULA-2

entwickle einen Plan: - zerlege Aufgabe in (einfache) Teilaufgaben, die ggf. nur für besondere Fälle der Gewünschte leisten
- prüfe die Vollständigkeit der Fallunterscheidung

hier: Übersetzung in rekursive Form

es gilt

1. [Fall: beide Argumente gleich] $ggt(x,x) = x$ ← triviale Teilaufgabe

2. [Fall: erstes Argument kleiner als zweites]

für $x < y$: $ggt(x,y) = ggt(x, y-x)$ ← vereinfachte Teilaufgabe

denn $x<y$, $w \text{ te } x$, $w \text{ te } y \Rightarrow$ es a,b mit $x = a \cdot w$, $y = b \cdot w$, $a < b$

$\Rightarrow$ es a,b mit $\underbrace{y-x}_{\neq 0} = b \cdot w - a \cdot w = \underbrace{(b-a)}_{\neq 0} \cdot w$

$\Rightarrow w \text{ te } y-x$

3. [Fall: erstes Argument größer als zweites]

$ggt(x,y) = ggt(y,x)$ ← vereinfachte Teilaufgabe

```
PROCEDURE ggt (x: CARDINAL; y: CARDINAL): CARDINAL;
(* x>0 und y>0 *)
BEGIN
   IF x=y THEN RETURN x
   ELSIF x<y THEN RETURN ggt(x, y-x)
   ELSE RETURN ggt(x-y, y)
   END
END ggt;
```

Wir müssen überprüfen:

a) jeder Rekursionsschritt erhält den gg.T:

folgt aus 1)-3) oben

b) die Rekursion endet:

in jedem Schritt wird ein Argument echt verkleinert, bleibt aber größer als 0;

d.h. sind x_0, y_0 Anfangswerte

x_i, y_i Werte für i-ten Schritt

dann gilt:

$0 < x_i \leq x_0$, $0 < y_i \leq y_0$ f. alle i

$x_{i+1} < x_i$ oder $y_{i+1} < y_i$ f. alle i

repetitive Form

```
PROCEDURE ggt (x: CARDINAL; y: CARDINAL): CARDINAL;
(* x>0 und y>0 *)
BEGIN
  WHILE x # y DO
    IF x<y THEN y := y-x
    ELSE (* x>y *)  x := x-y
    END
  END;
  (* x=y *)
  RETURN x
END ggt;
```

Das sind zwei mögliche MODULA-2 Formen zur Vereinbarung der neuen Funktionszeichen ggf.

Wir wollen nun ein ausführbares MODULA-2 Programm angeben, mit dem man das Funktionszeichen benutzen kann.

```
MODULE anwenden;

FROM InOut IMPORT ReadCard, WriteCard, WriteString, WriteLn;

VAR a,b: CARDINAL;

PROCEDURE ggt (x: CARDINAL; y: CARDINAL): CARDINAL;
(* x>0 und y>0 *)
BEGIN
   WHILE x # y DO
     IF x<y THEN y:= y-x
     ELSE (* x>y *) x := x-y
     END
   END;
   (* x=y *)
   RETURN x
END ggt;

BEGIN
   ReadCard(a); ReadCard(b);
   WriteString("  a = "); WriteCard(a,10); WriteLn;

   WriteString("  b = "); WriteCard(b,10); WriteLn;

   WriteString(" ggt = "); WriteCard(ggt(a,b),10); WriteLn
END anwenden.
```

Vereinbarungen

Anweisungen

Übersetzung

Vereinbarung des Funktionszeichens ggt

gibt die Stellenzahl an

'Aufruf' des oben definierten Funktionszeichens ggt mit den durch a bzw. b bezeichneten Werten

Übung: 1) man führe das Programm 'per Hand' aus für gewählte Eingabe-Werte und notiere die Zustandsübergänge (vgl. §3)!

2) auf der Workstation ausführen lassen!

B. MODULA-2 – Grundkurs

8. Alphabete, Worte, Syntaxdiagramme

aus §2.10: Die Syntax einer Programmiersprache legt fest, welche Zeichenfolgen (Worte) zum Aufschreiben von Werten, Ausdrücken, Anweisungen usw erlaubt sind.

Wir benötigen zunächst ein Beschreibungsmittel für Syntax:

Def Ein Alphabet ist eine endliche Menge (von Zeichen).

Bsp

$B = \{a, b, .., y, z, A, B, .., Y, Z\}$ Buchstaben

$Z = \{0, 1, .., 9\}$ Ziffern

$O = \{+, -, *, /, \text{DIV}, \text{MOD}\}$

$A = \{\text{IF}, \text{THEN}, \text{ELSIF}, \text{ELSE}, \text{END}, \text{WHILE}, \text{REPEAT}, \text{MODUL}, \text{IMPORT}, \text{PROCEDURE}\}$

Def Ein Wort über einem Alphabet X ist eine endliche (möglicherweise leere) Folge von Elementen aus X. X^* sei die Menge der Worte über X.
$\varepsilon \in X^*$ sei leeres Wort.

Bsp

ggt Wort über B

x1 Wort über $B \cup Z$

IF x=0 THEN y:=0 END Wort über $A \cup B \cup \{-, :=\} \cup Z$

Operationen auf Worten:

$\bullet : X^* \times X^* \longrightarrow X^*$

$x \bullet y := xy$

Hintereinanderschreiben, Konkatenation

$x \bullet \varepsilon = \varepsilon \bullet x = x$

Bsp

$a \bullet b = ab$

$\bullet$ jo $\bullet$ achim = joachim

$(X^*, \bullet)$ ist Monoid, d.h.
$\bullet$ assoziativ
ε neutrales Element

Operationen auf Wortmengen (Elemente aus $\mathcal{P}X^*$)

wenige Regelbare Operationen: $\cap, \cup, \setminus$

Konkatenation $X \cdot Y = \{u \cdot v \mid u \in X, v \in Y\}$

Bsp $B \cdot B = \{aa, ab, \dots aZ, ba, \dots, bZ, \dots, ZZ\}$

Potenz $X^0 = \{\varepsilon\}$

$X^{n+1} = X \cdot X^n$

Bsp $X^1 = X \cdot X^0 = X \cdot \{\varepsilon\}$

$= \{u \cdot \varepsilon \mid u \in X\}$

$= X$

es gilt offensichtlich: $X^* = \bigcup_{i \in \mathbb{N}} X^i = X^0 \cup X^1 \cup X^2 \cup \dots$

$= \{\varepsilon\} \cup X \cup X \cdot X \cup \dots$

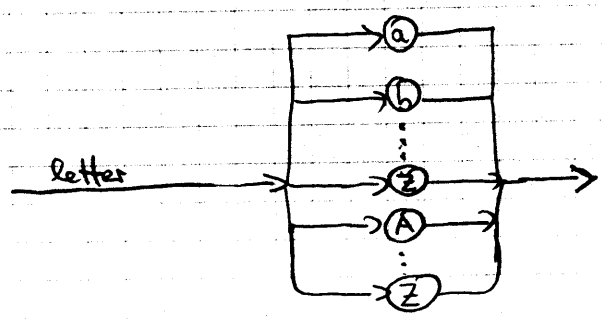
Syntaxdiagramme:

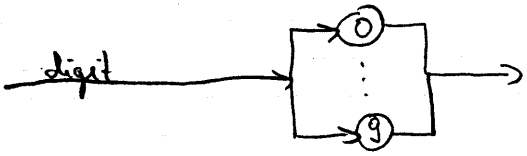
gerichtete Graphen mit
- einem Eingang (mit Namen der zu definierenden Menge)
- einem Ausgang
- Knoten für Zeichen x des Alphabets X (x)
  (einem der Einermengen $\{x\}$ für $x \in X$)
- Knoten für Wortmenge W

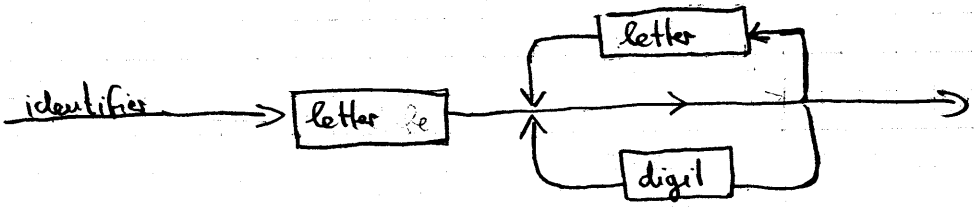
W

ein Syntaxdiagramm definiert die Menge der Worte, die man wie folgt bilden kann: man folgt einem gerichteten Weg vom Eingang zum Ausgang, entnimmt dabei jedem Knoten (d.h. der dort angegebenen Menge) ein Element und konkateniert diese Elemente.

Bsp





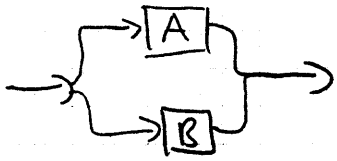


Bezeichner sind: x
aha
x1y
HochschuleHildesheim

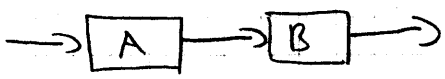
keine Bezeichner sind: 1
1a

zwischen einigen der obigen Operationen auf Wortemengen und Syntaxdiagrammen besteht ein enger Zusammenhang:

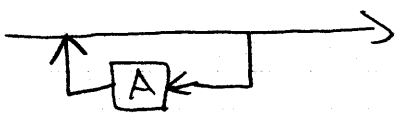
$A \cup B$



$A \cdot B$

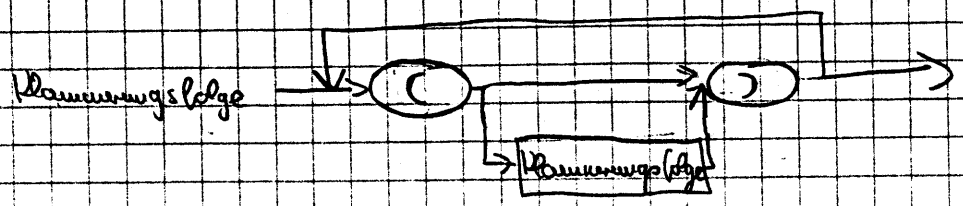


A^*



ein Syntaxdiagramm darf auch einen Knoten mit Namen enthalten; dann muß es aber stets einen Weg finden, der keinen solchen Knoten trifft.

Bsp



Klammerungsfolgen sind ( )

(( )) ( )

Klammerungs-F.

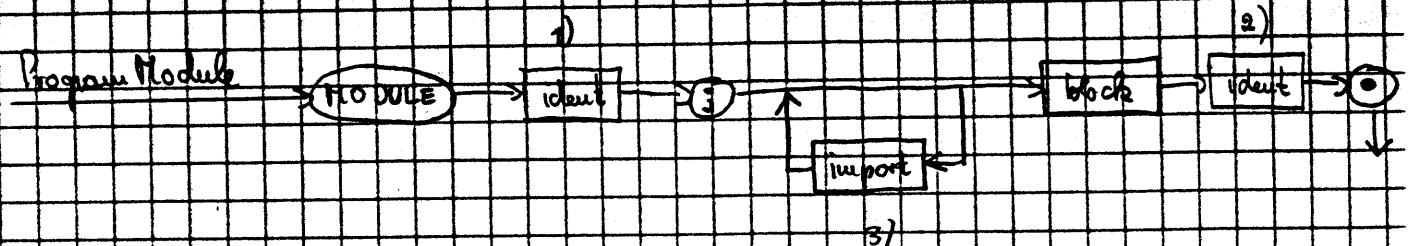
Klammerungs-F.

Klammerungs-F.

Klammerungs-F.

Keine Klammerungsfolge ist (( )

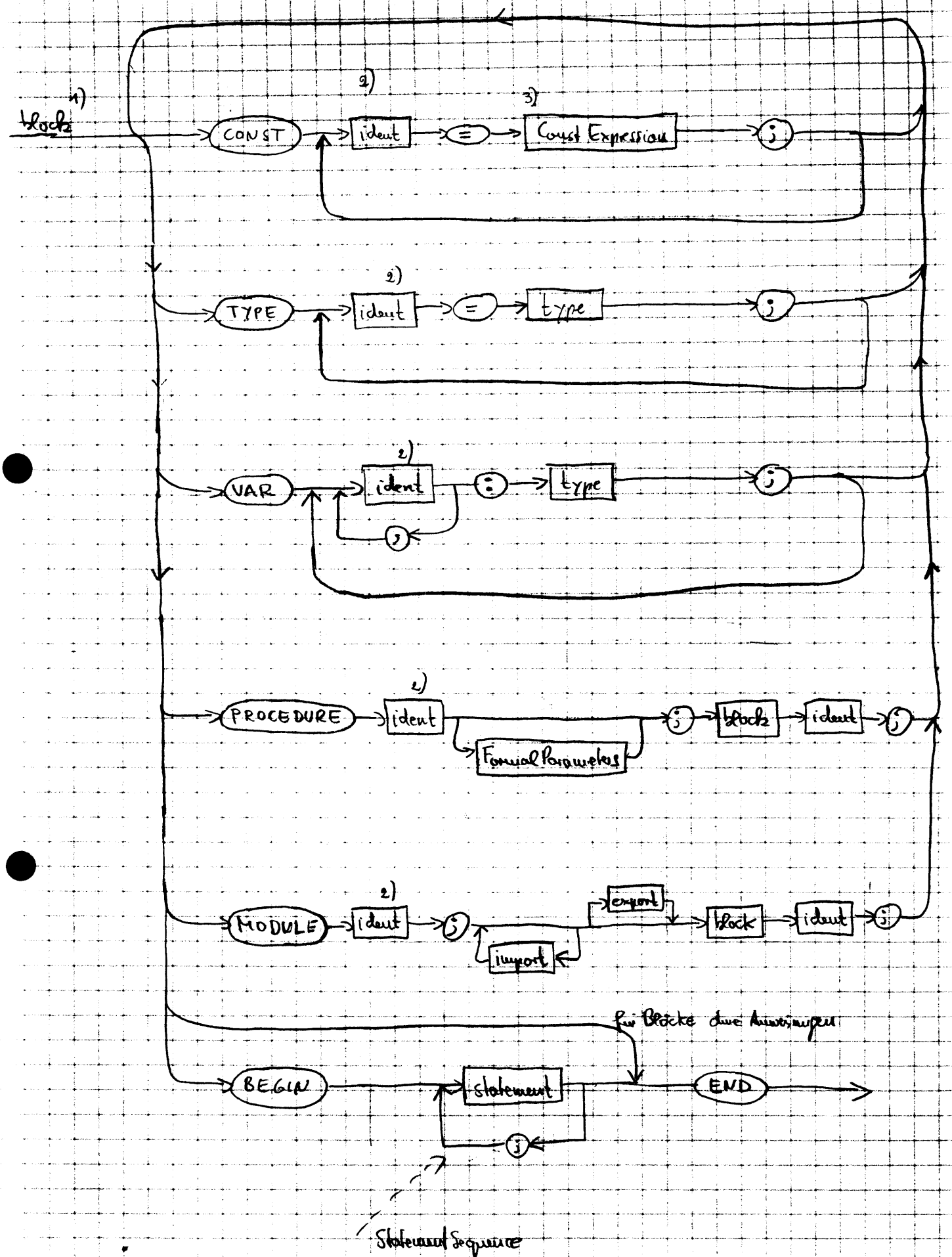
Der allgemeine Aufbau von MODULA-2 - Programmen (wie weit der folgenden Angaben noch nicht vollständig):



1) Programme selbst sind Werte (Objekte) und werden bezeichnet. Der hier festgelegte Bezeichner (identifier) benennt das folgende Programm innerhalb der "Umgebung" (Standardmodule, Laufzeitsystem, Betriebssystem, ...).

2) Der abschließende Bezeichner muß gleich dem einführenden sein.

3) Ein MODULA-2 - Programm kann "Größen von außen" benutzen. Diese Größen müssen zunächst in das Programm eingeführt werden (importiert werden), indem sie durch ihre Bezeichner aufgezählt werden.
Ein Teil der standardmäßigen Umgebung, nämlich das Modul InOut bietet Größen zum Lesen und Schreiben von und zu der Standard - Ein/Ausgabe an.


block
1)
CONST
2)
ident
=
3)
Const Expression
;
TYPE
2)
ident
=
type
;
VAR
2)
ident
:
type
;
,
PROCEDURE
2)
ident
;
block
ident
;
Formal Parameters
MODULE
2)
ident
;
import
export
block
ident
;
Pour Blocks avec Instructions
BEGIN
statement
;
END
Statement Sequence

1) Die Reihenfolge der Vereinbarungen ist im wesentlichen wahlfrei und sollte nach inhaltlichen Gesichtspunkten und für eine gute Lesbarkeit gewählt werden.

2) Alle in den Anweisungen bzw. in untergeordneten Blöcken vorkommenden Bezeichner (außer Standardbezeichner gemäß 3), weiteren der MODULA-2-Maschine bekannten Standardbezeichnern, importierten Bezeichnern) müssen entsprechend ihrer beabsichtigten Verwendung vereinbart werden. Dies dient u.a.

- der besseren Lesbarkeit der Programme
- dem "disziplinierten Programmieren"
- dem "sicheren Programmieren" (z.B. können spätere Schreibfehler häufig erkannt werden)
- der Bereitstellung von Speicherplatz und Verarbeitungsregeln durch den Compiler und das Laufzeitsystem
- der Kontrolle (durch den Compiler), ob Funktionszeichen mit "geeigneten Operanden" versehen sind

Alle diese Bezeichner sollen verschieden sein von

- Begrenzern, z.B. BEGIN, END, VAR, ...
- Standardbezeichnern, z.B. CARDINAL, INTEGER, ...

Jeder Bezeichner muß stets zusammen mit seinem vereinbarten Typ behandelt werden; man spricht von getypten Bezeichnern.

3) Elementare Werte haben vorbelegte Standardbezeichner, genannt Konstante. Im allgemeinen identifiziert man diese Standardbezeichner mit den Werten und sagt dann, "die Konstanten bezeichnen sich selbst". Aus solchen Konstanten können Ausdrücke aufgebaut werden.

4) Für elementare Werte kann der Benutzer eigene Bezeichner definieren

z.B.
```
CONST pi = 3.14;
CONST mwstsatz = 0.14;
CONST näherungsschrittweite = 10;
```

- Das dient u.a
  - der besseren Lesbarkeit des Programms
  - der besseren Änderbarkeit des Programms

9. Skalare Datentypen

9.0

Wertemengen, welche wir immer zusammen mit den darauf definierten Operationen betrachten:

Datentyp = Wertemenge und Operationen

Ein Datentyp wird <u>skalar</u> genannt, wenn

- seine Werte <u>unstrukturiert</u> sind (bzw. die Operationen diese Unstrukturiertheit erhalten lassen)
- eine <u>lineare Ordnung</u> auf der Wertemenge definiert ist

$(X, \leq)$ heißt lineare Ordnung : gdw

1) $\leq$ zweistellige Relation über X; $\leq \subset X \times X$, d.h. eine beliebige Paar von Elementen aus X
2) $x \leq x$ f.alle $x \in X$ (reflexiv)
3) $x \leq y$ und $y \leq x \Rightarrow x = y$ f.alle $x, y \in X$ (antisymmetrisch)
4) $x \leq y$ und $y \leq z \Rightarrow x \leq z$ f.alle $x, y, z \in X$ (transitiv)
5) $x \leq y$ oder $y \leq x$ f.alle $x, y \in X$ (linear)

Für alle skalaren Datentypen X sind folgende Operationen definiert:

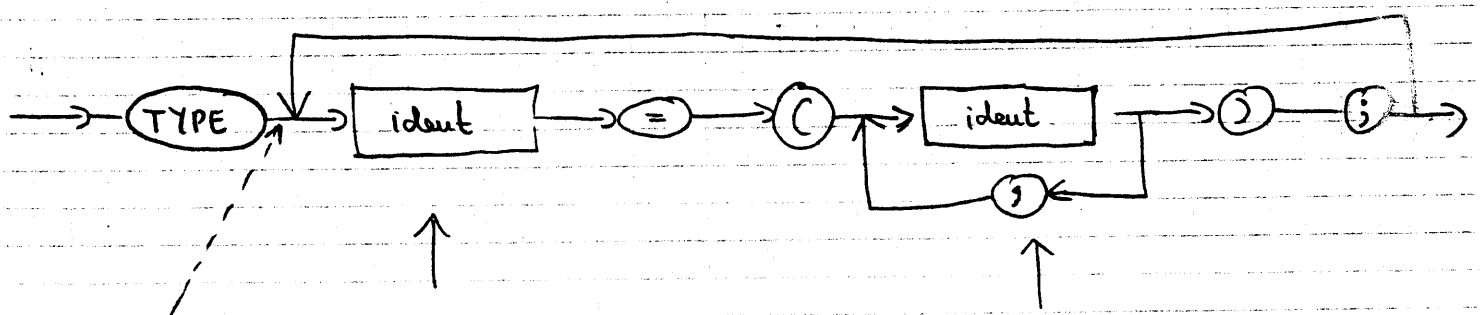
Vergleichs-

		MODULA-2-Funktionszeichen
Gleichheit	$= : X \times X \to \{false, true\}$	=
Ungleichheit	$\neq : X \times X \to \{false, true\}$	<> oder #
Vergleiche	$< : X \times X \to \{false, true\}$	<
	$\leq : X \times X \to \{false, true\}$	<=
	$> : X \times X \to \{false, true\}$	>
	$\geq : X \times X \to \{false, true\}$	>=

Ist die lineare Ordnung diskret (wenn der Standardtyp REAL (reelle Zahl)) wird als nicht diskret behandelt), so sind die folgenden Funktionen definiert:

$INC: X \xrightarrow{p} X$	Nachfolger (increase)	
$DEC: X \xrightarrow{p} X$	Vorgänger (decrease)	
$ORD: X \to \mathbb{N}$	Ordnungszahl (ordinal)	(nicht für Standardtyp INTEGER (ganze Zahl), vgl. 9.2)

Skalare Typen definiert man in MODULA-2 durch Aufzählung:



TypeDeclaration — kennzeichnet den Typ — kennzeichnet die Elemente der Wertemenge in der Reihenfolge ihrer Aufzählung

Bsp

TYPE Farbe = (rot , gelb , grün , blau) ;

ORD(rot) = 0
ORD(gelb) = 1
ORD(grün) = 2
ORD(blau) = 3

INC(rot) = gelb
INC(gelb) = grün
INC(grün) = blau
[INC(blau) undefiniert]

[DEC(rot) undefiniert]
DEC(gelb) = rot
DEC(grün) = gelb
DEC(blau) = grün

- TYPE wochentag = (mon , die , mit , don , fre , sam , son) ;
  arbeitstag = (moa , dia , mia , doa , fra) ;

  Werden mehrere Typen vereinbart, so müssen die Bezeichner alle untereinander verschieden sein (damit kann man einem Bezeichner nicht ansehen, von welchem Typ er ist).

- In MODULA-2 (wie in beinahe allen anderen Programmiersprachen) sind die Datentypen für
  - Wahrheitswerte
  - ganze Zahlen
  - natürliche Zahlen
  - reelle Zahlen
  - Zeichen
  - Teilmengen von {0,...,N-1} für ein implementierungsabhängiges N

  standardmäßig definiert.

9.1 Der Standardtyp Wahrheitswert

Wertemenge: FALSE, TRUE (standardmäßig definierte Bezeichner)

es gilt: ORD(FALSE) = 0
ORD(TRUE) = 1

Der Standardbezeichner für diesen Typ ist: BOOLEAN

Operationen:

- Standardoperationen für skalare Datentypen
- Die Operationen NOT Negation
  OR Disjunktion, Oder
  AND Konjunktion, Und

deren Wertverlauf durch folgende Tabelle gegeben ist:

x	y	NOT x	x AND y	x OR y
FALSE	FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE
TRUE	TRUE	FALSE	TRUE	TRUE

Es gibt insgesamt $2^4 = 16$ verschiedene zweistellige Funktionen auf $\{FALSE, TRUE\}$

(f = FALSE ; t = TRUE)

x	y	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}
f	f	f	f	f	t	f	t	f	t	f	t	f	t	f	t	f	t
f	t	f	f	t	t	f	f	t	t	f	f	t	t	f	f	t	t
t	f	f	f	f	f	t	t	t	t	f	f	f	f	t	t	t	t
t	t	f	f	f	f	f	f	f	f	t	t	t	t	t	t	t	t

b_0: Kontradiktion, FALSE

b_1: Und, AND, $x \wedge y$

b_2: Äquivalenz, $x \Leftrightarrow y$

b_3: Projektion, $x = y$

b_4: Oder, OR, $x \vee y$

b_{15}: Tautologie, TRUE

Bem: alle diese Funktionen lassen sich mit Hilfe von NOT, AND, OR und den Vergleichsoperatoren (für Wahrheitswerte) ausdrücken! Übung!

Bem: alle diese Funktionen lassen sich auch nur mit Hilfe von NOT, AND, OR (und FALSE, TRUE) ausdrücken! Übung!

Bsp $b_9(x,y) = \begin{cases} TRUE & falls\ x=y \\ FALSE & falls\ x \neq y \end{cases}$

$= x = y$

↑ Standard-Vergleichsoperation für $\{FALSE, TRUE\}$

$= x\ AND\ y\ OR\ NOT\ x\ AND\ NOT\ y$

= (x AND y) OR ((NOT x) AND (NOT y))

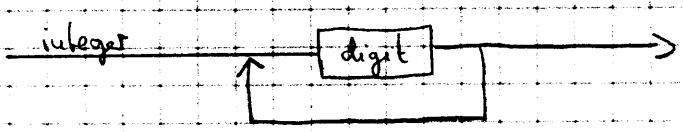
d.h. NOT bindet stärker als AND
AND bindet stärker als OR

Solche Identitäten kann man stets durch Wertverlaufstabellen nachprüfen.

9.2 Der Standardtyp ganze Zahl

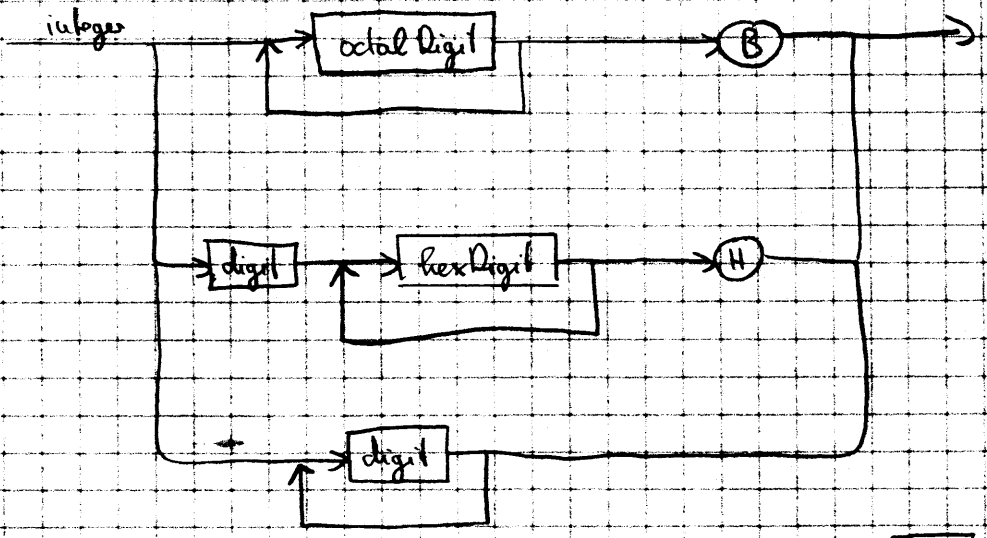
Wertemenge: { ..., -2, -1, 0, 1, 2, ... } endlicher Abschnitt der ganzen Zahlen

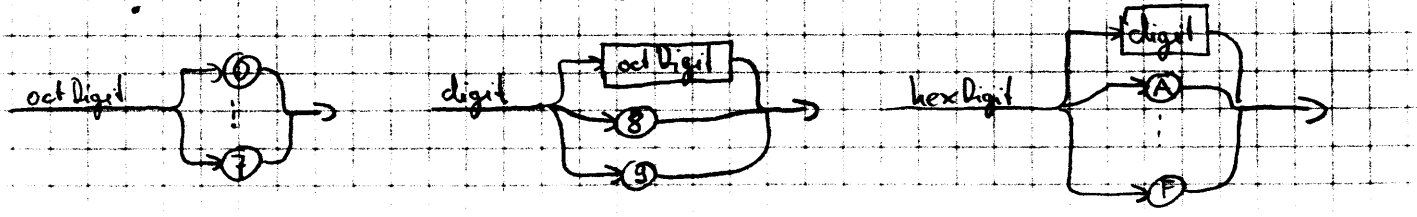
für dezimale Darstellung (nichtnegative Zahlen):



(negative Zahlen werden durch Ausdrücke der Form – Zahl beschrieben)

oktale und hexadezimale Darstellungen sind auch zulässig:

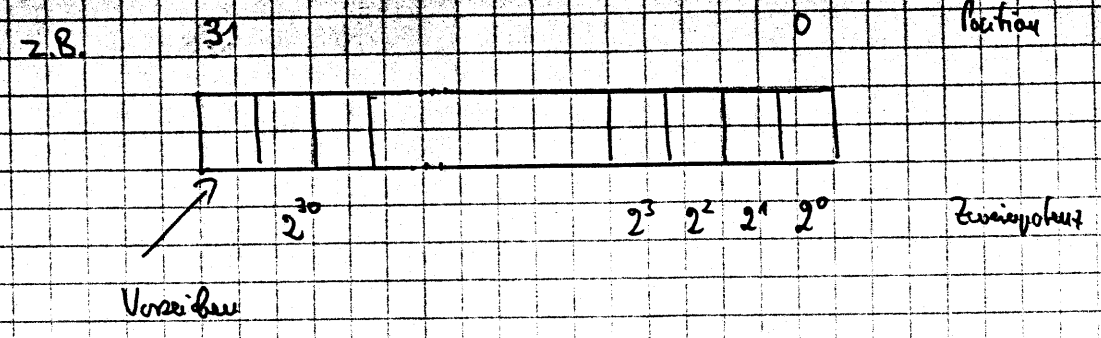




Bsp. 0
13
11111

interne Darstellung im Dualsystem:

z.B.



größte Zahl $2^{31}-1 = 2\,147\,483\,647$, bezeichnet durch MaxInt

d.h. nur 9 Dezimalstellen sind garantiert

Der Standardbezeichner für diesen Typ ist: INTEGER

Operationen

- Standardoperationen für skalare Datentypen, außer ORD
- * Multiplikation
  DIV ganzzahlige Division (Teile, ignoriere "Rest")
  MOD ganzzahliger Rest: a MOD b = a - ((a DIV b) * b)
- \+ Addition (zweistellig), Identität (einstellig)
  \- Subtraktion (zweistellig), Inverses (einstellig)

ferner sind häufig einige wichtige Funktionen standardmäßig definiert, z.B.

ABS(x) Absolutwert von x

ODD(x) liefert Wahrheitswert: TRUE, falls x ungerade; FALSE, falls x gerade

Achtung: wegen der Begrenzung des Zahlenraumes gilt:

i) *, +, - bezeichnen partielle Funktionen, die nur dann definiert sind, wenn das Ergebnis noch in der Wertemenge liegt

Falls der Wertebereich überschritten wird, spricht man von einem "Überlauf" (der Speicherzelle), overflow, und die Programmausführung wird vom Laufzeitsystem abgebrochen (das ist der beste Fall) oder es entsteht ein "Unsinnsergebnis".

ii) die Gesetze der Arithmetik gelten deshalb nicht uneingeschränkt:

z.B. sei "unsinnig": $a+b > MaxInt$

$a+b-c < MaxInt$

$a-c \geq 0$

dann liefert

$\underbrace{(a+b)}_{\text{undefiniert}}-c$ keinen Wert, da Ausdrücke unter Beachtung der Klammern nach Vorrangregeln von links nach rechts ausgewertet werden

aber

$(a-c)+b$ liefert den "sinnvollen" Wert

9.3. Der Standardtyp natürliche Zahl

Wertemenge: $\{0, 1, 2, \ldots\}$ endlicher Abschnitt der natürlichen Zahlen ((endliche) Kardinalzahlen)

Bezeichner, interne Darstellung, Operationen sind sinngemäß wie bei ganzen Zahlen, wobei natürlich die einstellige Operation – nicht erlaubt ist.

Man beachte, daß keine gemischten Ausdrücke aus Operanden vom Typ INTEGER und vom Typ CARDINAL gebildet werden dürfen; häufig werden jedoch sog. Typ-Transferfunktionen bereitgestellt:

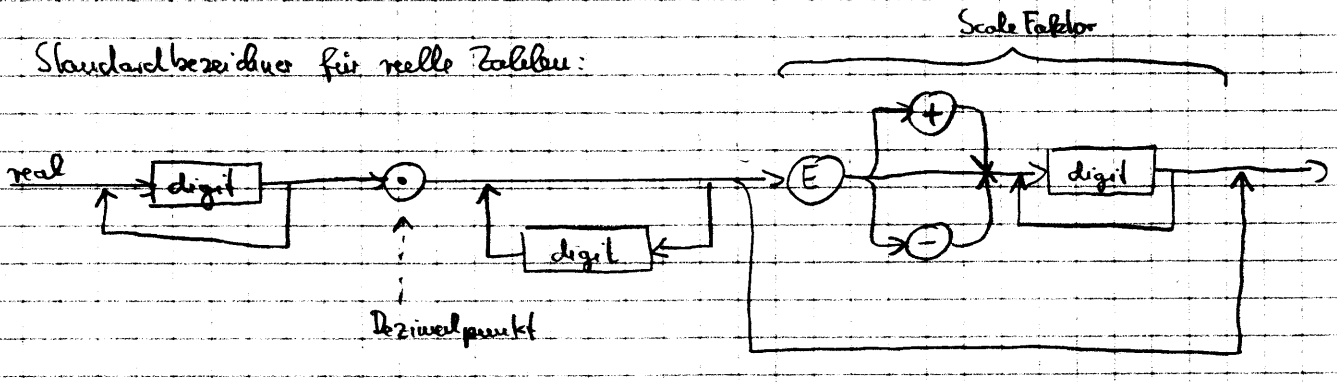
$$\text{INTEGER}: \{0, 1, \ldots\} \to \{\ldots, -1, 0, 1, \ldots\}, \quad \text{INTEGER}(c) = c$$

$$\text{CARDINAL}: \{\ldots, -1, 0, 1, \ldots\} \to \{0, 1, \ldots\}, \quad \text{CARDINAL}(i) = \begin{cases} i & \text{falls } i \geq 0 \\ \text{undef.} & \text{sonst} \end{cases}$$

9.4 Der Standardtyp reelle Zahl

Wertemenge: endliche Teilmenge der reellen Zahlen

Standardbezeichner für reelle Zahlen:



Bsp.: 1.0

13.7E20 steht für $13{,}7 \cdot 10^{20}$

0.137E-2 steht für $0{,}137 \cdot 10^{-2} = 0{,}00137$

Intern in Gleitkomma-Darstellung (floating point notation)

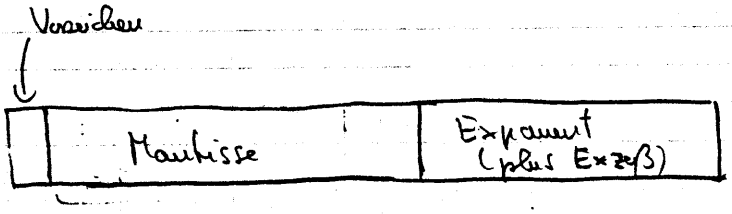
Sei b (z.B. $= 16$) Basis, dann wird eine reelle Zahl x dargestellt als

$$x = m \cdot b^e$$

mit Normalisierung für die Mantisse m: $\frac{1}{b} \leq |m| < 1$

für den Exponenten e: e ganzzahlig

(wobei zum Exponenten i.allg. ein Exzeß addiert wird, um negative Zahlen zu vermeiden)



z.B. etwa ... insgesamt 64 Positionen

größte Zahl: $\sim 7{,}2 \cdot 10^{75}$

kleinste positive Zahl: $\sim 5{,}4 \cdot 10^{-79}$

Diese Art der Zahlendarstellung erlaubt es, einen großen Bereich reeller Zahlen approximativ darzustellen. Die Arithmetik folgt den üblichen Rechengesetzen beim Rechnen mit Brüchen und Potenzen, jedoch muß man wegen der begrenzten Stellenzahlen auf manchmal erhebliche Rundungsfehler gefaßt sein. Insbesondere gelten die Gesetze der Assoziativität und Distributivität i.allg. wegen Stellenauslöschungen nicht.

Bsp Basis: $b = 10$

Mantisse: 4 Dezimalstellen und Vorzeichen

Exponent: 1 Dezimalstelle und Vorzeichen

sei $a = 0,001235$

$b = 0,5512$

$c = -0,5511$

in $\mathbb{R}$ gilt: $(a+b)+c) = a+(b+c) = 0,001335$

in Gleitkomma-Arithmetik

$$\left(\langle +0,1235 \mid -2\rangle \oplus \langle 0,5512 \mid 0\rangle\right) \oplus \langle -0,5511 \mid 0\rangle$$

$$= \langle +0,5524 \mid 0\rangle \oplus \langle -0,5511 \mid 0\rangle = \langle +0,1300 \mid -2\rangle$$

↑ die beiden letzten Stellen von a gehen verloren

aber

$$\langle +0,1235 \mid -2\rangle \oplus \left(\langle +0,5512 \mid 0\rangle \oplus \langle -0,5511 \mid 0\rangle\right)$$

$$= \langle +0,1235 \mid -2\rangle \oplus \langle +0,1000 \mid -3\rangle = \langle +0,1335 \mid -2\rangle$$

Die Auslöschung geschieht hier, weil zwei Summanden betragsmäßig entgegengesetzt gleich sind, während der weitere Summand in der Größenordnung der Summe ist.

Allgemein gilt: Der Programmierer muß sich über den schlimmstmöglichen Fall der Auslöschung im klaren sein und entsprechende Vorsichtsmaßnahmen treffen.

Insbesondere sollten errechnete reelle Zahl nicht auf exakte Gleichheit, sondern nur auf kleine Abweichung, $|a-b| < \varepsilon$, getestet werden.

Der Standardbezeichner für diesen Typ ist REAL

Operationen:

Die Vergleichsoperationen (für skalare Datentypen): =, #, <, <=, >, >=

* Multiplikation

/ Division

\+ Addition (zweistellig), Identität (einstellig)

\- Subtraktion (zweistellig), Inverses (einstellig)

ferner sind häufig weitere Funktionen standardmäßig definiert.

9.5 Der Standardtyp Zeichen

siehe § 14

9.6 Der Standardtyp "Teilmenge von von {0, .., N-1}

siehe § 26 (und Übung)

9.7 Ausdrücke

Datentyp : Wertemenge und Operationen (bezeichnet durch Funktionszeichen)

Operationen kann man zusammensetzen : einsetzen, Hintereinanderausführen,

solche zusammengesetzten Operationen werden durch Ausdrücke (vgl. § 2.6) beschrieben.

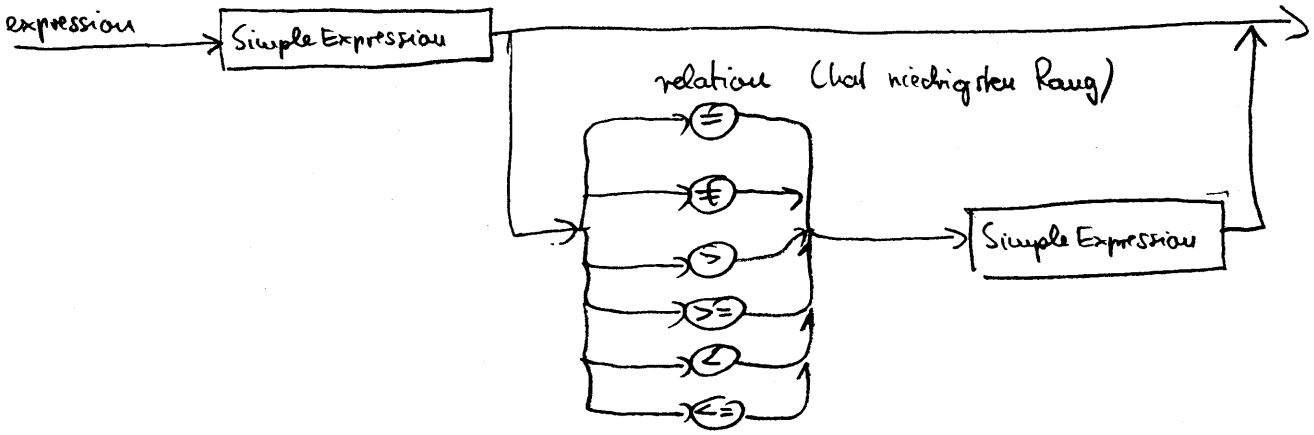
Hierbei muß man beachten:

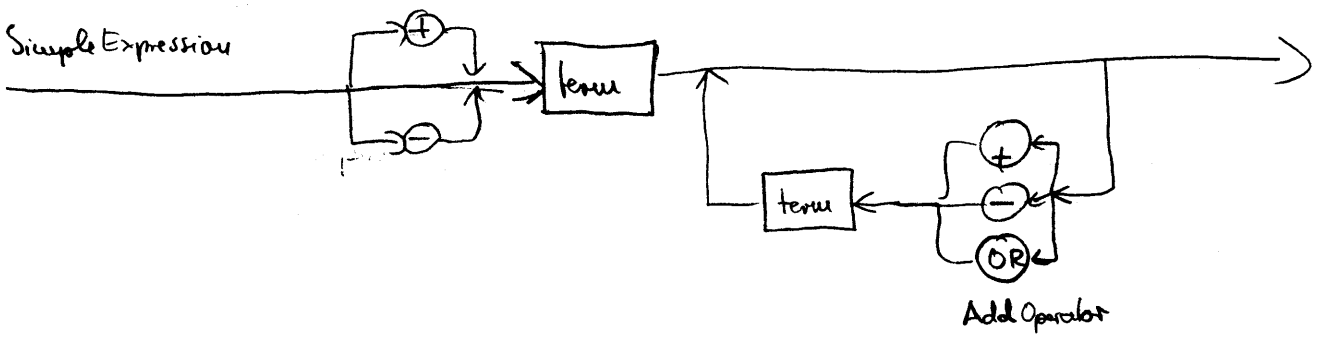
- Argumente müssen vom richtigen Typ sein.
- Auswertung geschieht
  - entsprechend Klammerung
  - entsprechend Rang mit folgender Rangfolge (: von stark nach schwach)

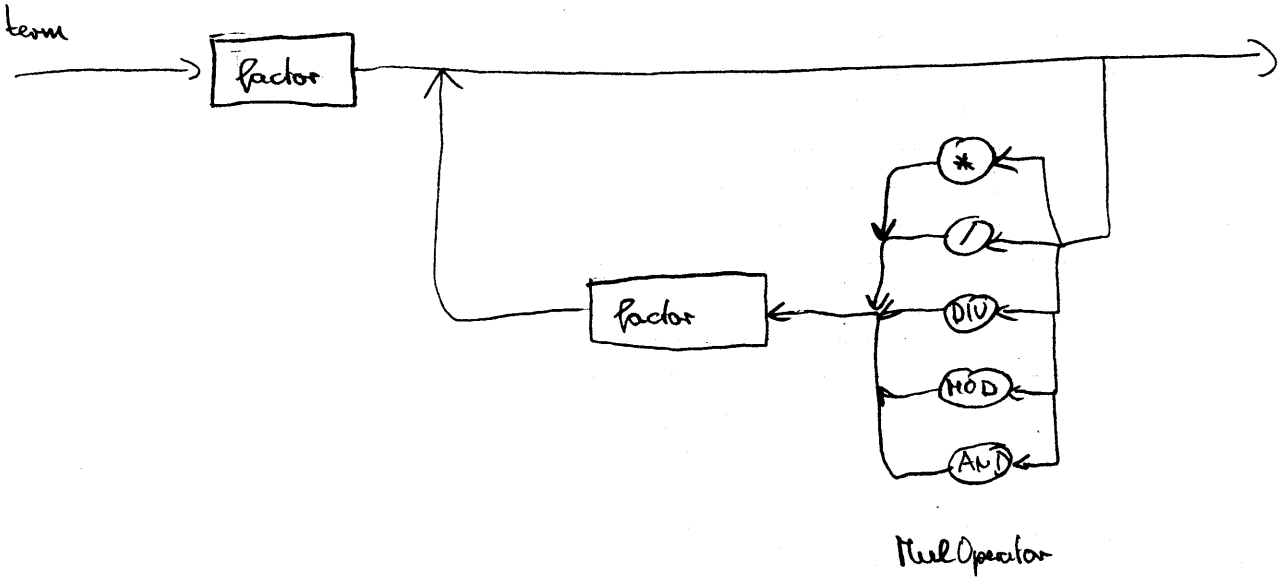
not, +(einstellig), -(einstellig)	
*, /, DIV, MOD, AND	multiplikative Operationen
+(zweistellig), -(zweistellig), OR	additive Operationen
=, #, <, <=, >, >=	Vergleichsoperationen

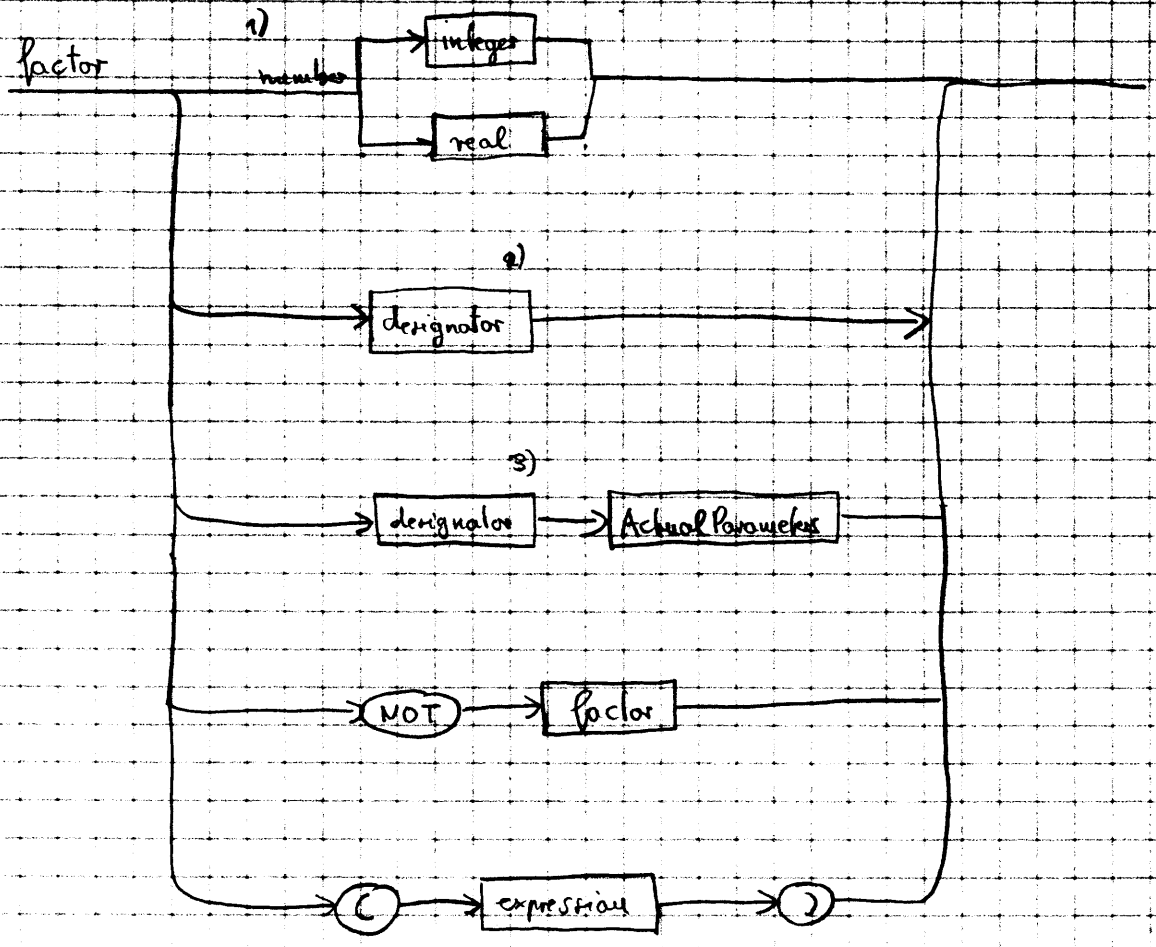
  - von links nach rechts bei gleichem Rang

Auswertungsregel wird durch skizzieren Aufbau der Syntaxdiagramme wiedergegeben:


expression
Simple Expression
relation (hat niedrigsten Rang)
=
#
>
>=
<
<=
Simple Expression


Simple Expression
+
-
term
term
+
-
OR
Add Operator


term
factor
factor
*
/
DIV
MOD
AND
Mul Operator



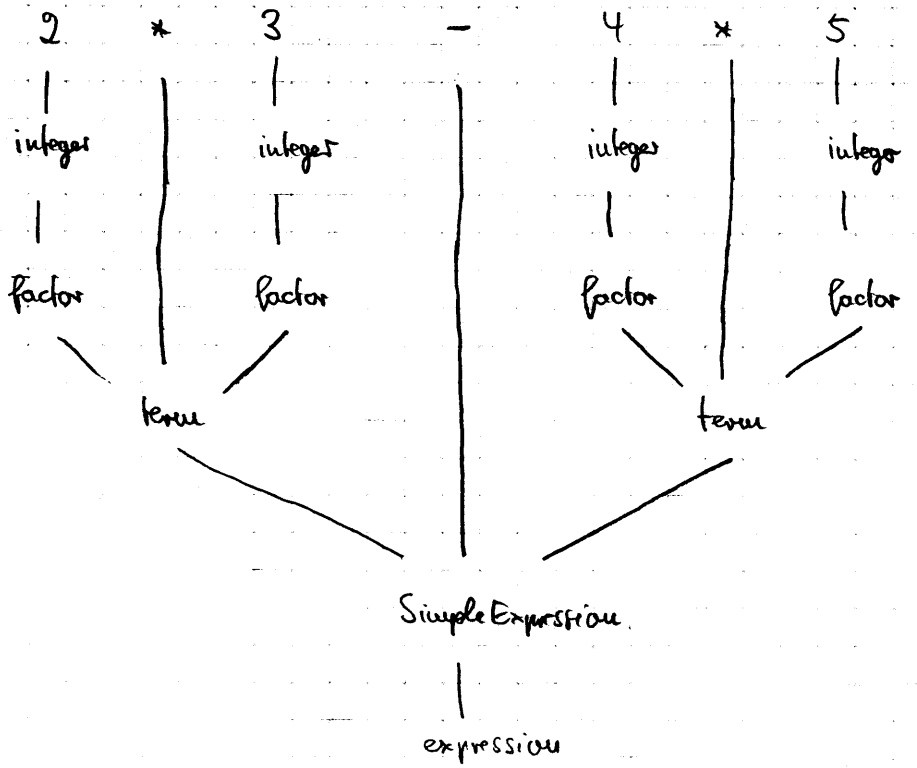
1) Für noch nicht besprochene Datentypen sind auch andere Konstanten erlaubt.

2) Zur Zeit nur mit VAR vereinbarte Bezeichner, später allgemeiner.

3) Mit PROCEDURE als Funktionsprozedur vereinbarte Bezeichner.

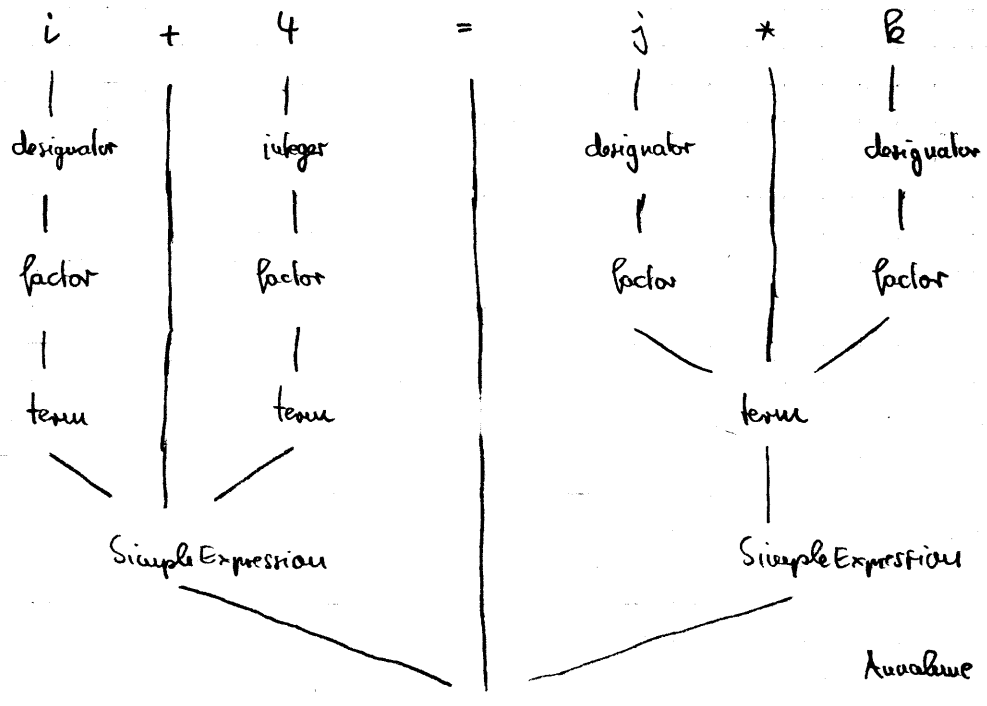
In den folgenden Beispielen soll der Aufbau der Ausdrücke jeweils durch einen sog. Strukturbaum dargestellt werden. Ein Strukturbaum zeigt, wie der Ausdruck anhand der Syntaxdiagramme entstanden ist.

a)



$2*3-4*5$ liefert $(2*3)-(4*5) = 6-20 = -14$

b)



Annahme folgende Vereinbarung:

VAR i,j,k : INTEGER;

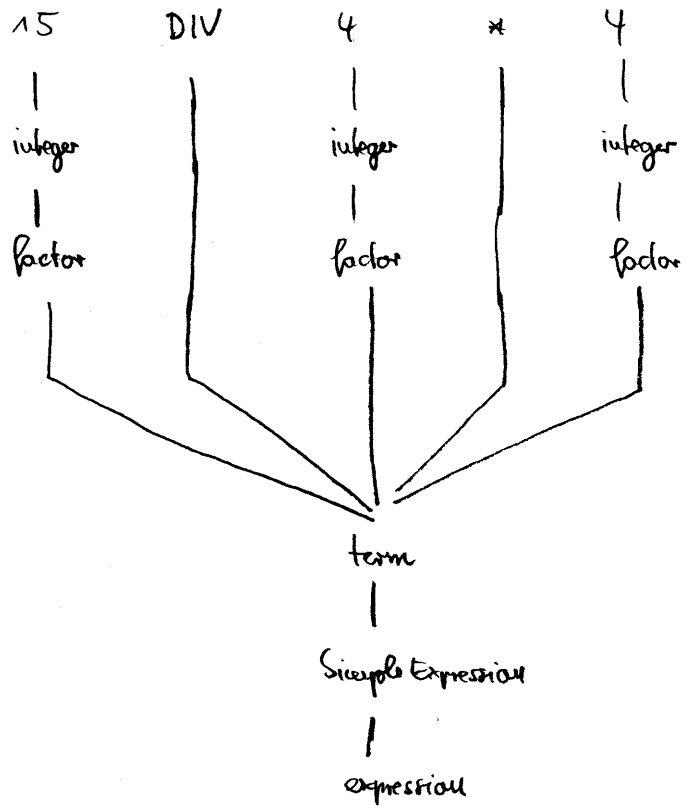
$i + 4 = j * k$ liefert im Zustand $\begin{pmatrix} i \\ 3 \end{pmatrix} \begin{pmatrix} j \\ 2 \end{pmatrix} \begin{pmatrix} k \\ 4 \end{pmatrix}$:

durch Ersetzung: $3 + 4 = 2 * 4$

durch schrittweise Auswertung: $7 = 8$

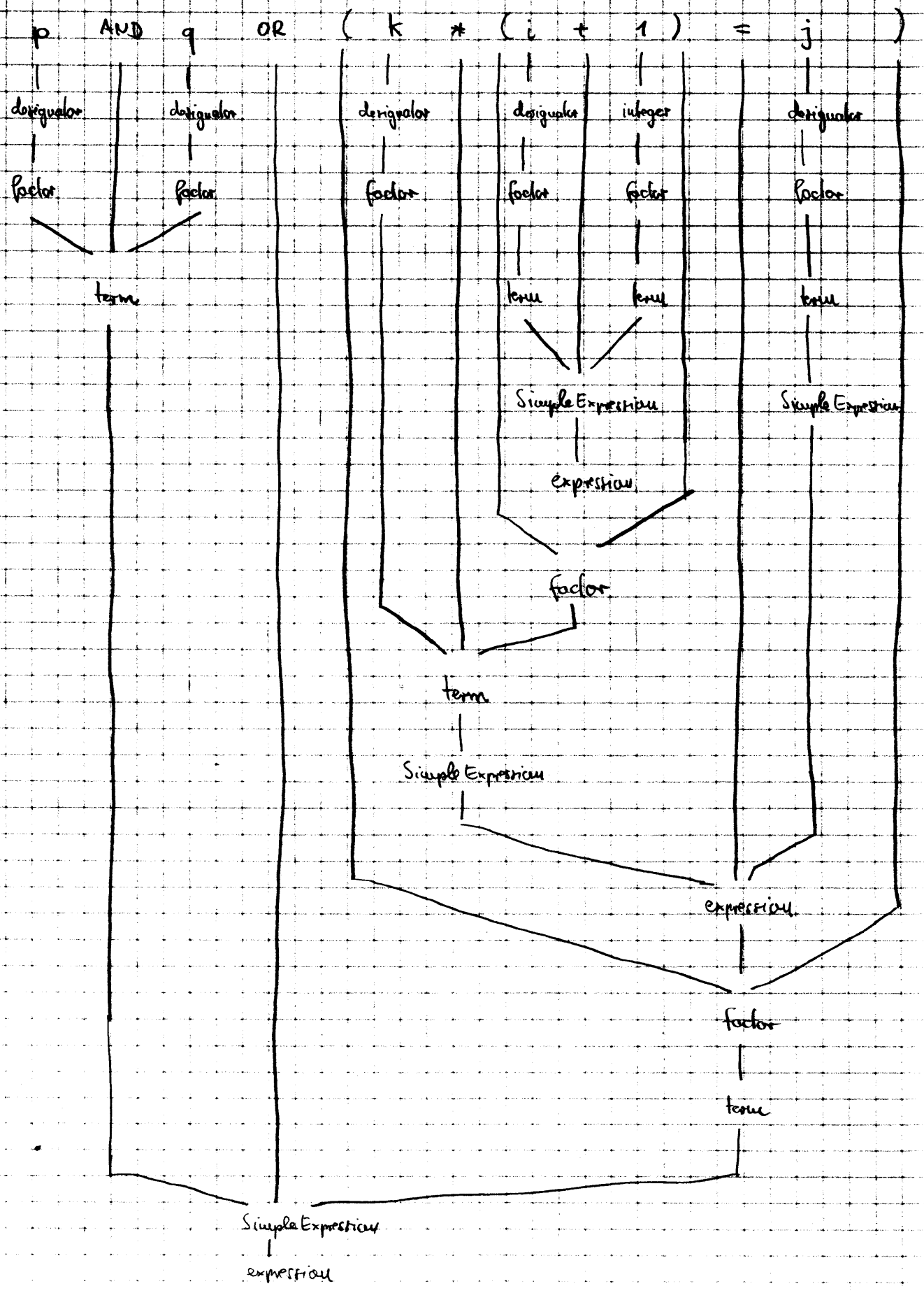
FALSE

c)

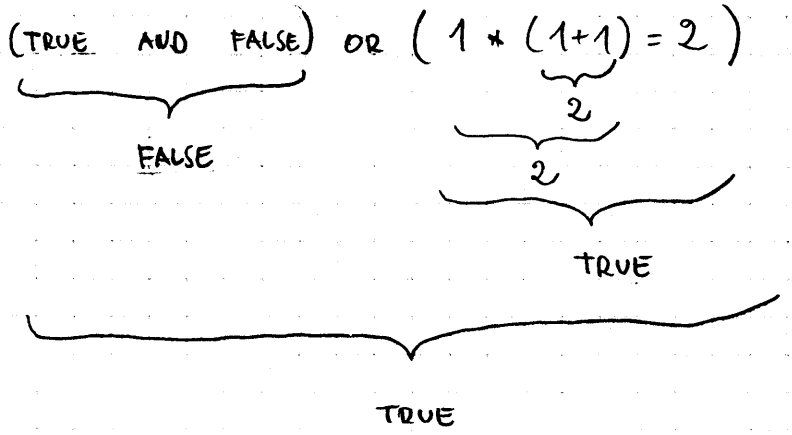


15 DIV 4 * 4 liefert (15 DIV 4) * 4 = 3 * 4 = 12

d) Anweisung: VAR i, j, k : INTEGER ;
p, q : BOOLEAN ;



p AND q OR (k * (i+1) = j) liefert im Zustand (p: TRUE | q: FALSE | i: 1 | j: 2 | k: 1):



Ein Ausdruck liefert zusammen mit einem Zustand einen Wert, sofern

- alle im Ausdruck vorkommenden Bezeichner auch im Zustand vorkommen
- diese Bezeichner zu den Funktionszeichen passende Werte haben;

andernfalls ist der Ausdruck undefiniert!

Der Typ des gelieferten Wertes wird durch das "äußerste" (d.h. das bei der Auswertung zuletzt durchgeführte) Funktionszeichen bestimmt.

Bsp 2*3 - 4*5 liefert Wert vom Typ INTEGER

15 DIV 4 * 4 " " " " INTEGER

p AND q OR (k*(i+1)=j) " " " " BOOLEAN

Ausdrücke, die einen Wert vom Typ INTEGER, CARDINAL oder REAL liefern heißen arithmetische Ausdrücke.

Ausdrücke, die einen Wert vom Typ BOOLEAN liefern, heißen Boolsche Ausdrücke; sie dienen insbesondere der Steuerung des Programmablaufs.

Typanpassungen

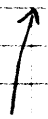
in der Mathematik: $\mathbb{Z} \subset \mathbb{R}$

in der "reinen Informatik": Wertebereiche oft disjunkt, d.h. bzw. ausgedrückt

$$\text{INTEGER} \cap \text{REAL} = \emptyset$$

In MODULA-2 können Typanpassungen vom Programmierer angefordert werden mit Hilfe standardmäßig definierter Operationen, z.B.

INTEGER : $\{0,1,\ldots\} \longrightarrow \{\ldots,-1,0,1,2,\ldots\}$



Diese Bezeichnung dient hier als Funktionszeichen für die Einbettungsfunktion der Kardinalzahlen in die ganzen Zahlen

FLOAT : $\{0,1,\ldots\} \longrightarrow \mathbb{R}$

Einbettungsfunktion der Kardinalzahlen in die reellen Zahlen

TRUNC : $\mathbb{R}^+ \longrightarrow \{0,1,2,\ldots\}$

truncate, liefert den ganzzahligen Anteil, z.B. TRUNC (3.7) liefert 3

Typanpassungen sind häufig implementierungsabhängig; i.allg. gibt es weitere Möglichkeiten.

10. Wertzuweisung

Werte müssen bezeichnet werden:

- ein beim Aufschreiben des Programmes bekannter Wert kann in MODULA-2 durch die Standardbezeichner für die elementaren Werte angegeben werden (bzw. durch die CONST-Klausel einen neuen Bezeichner erhalten)
- ein beim Aufschreiben des Programmes noch nicht bekannter Wert wird durch einen Bezeichner benannt, der durch die VAR-Klausel vereinbart wird (oder z.B. im Kopf einer Funktionsprozedur erscheint),

bei diesen Werten handelt es sich insbesondere um

- "von außen" kommende "Eingabe-Werte"
- "Zwischenergebnisse" in iterativen Anweisungen
- von Anweisungen gelieferte "Ergebniswerte", die "später" wieder verwendet werden sollen, also zwischenzeitlich gespeichert werden sollen

Die zwischen vorhandenen Bezeichner mit den bezeichneten Werten stellen den veränderlichen Zustand (vgl. § 2.5) dar.

Elementare Änderungen eines Zustandes, d.h. elementare Anweisungen (vgl. § 2.8) sind:

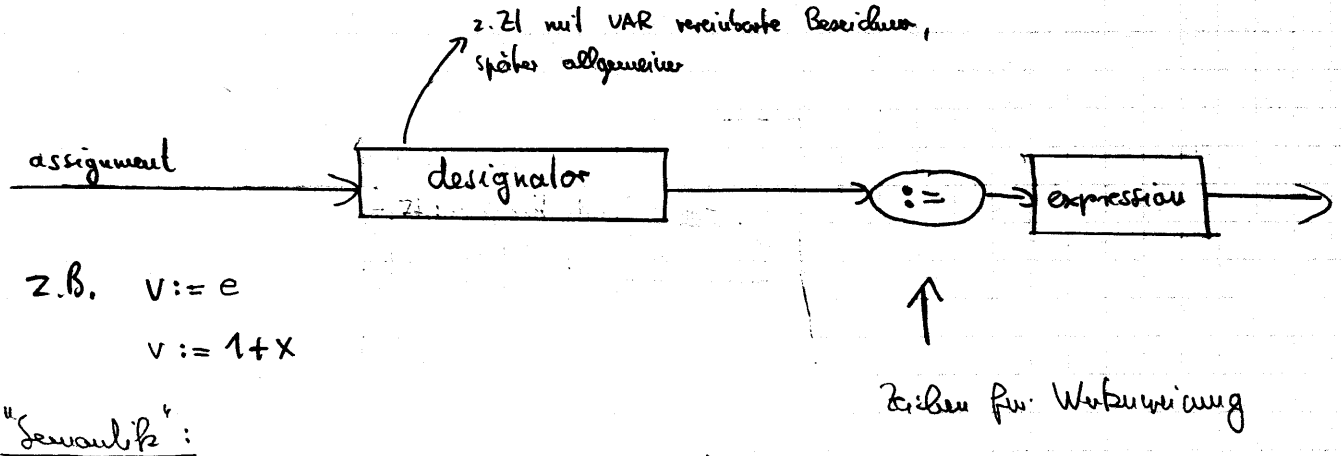
- Einführung neuer Bezeichner:

  bewirkt wesentlich durch: VAR-Klausel;

  beachte: in MODULA-2 wird (leider) nicht sofort ein Wert bezeichnet; hierfür ist in MODULA-2 der Programmierer verantwortlich

- (Wieder-)Verwendung eines Bezeichners für einen neuen errechneten Wert (und "Vergessen" des alten Wertes!)

dieses wird in MODULA-2 als 'Wertzuweisung (assignment)' notiert:



z.B. v := e

v := 1+x

"Semantik":

"werte den Ausdruck aus und bezeichne das Ergebnis v mit dem Bezeichner"

Häufig schreibt man statt v:=e auch:

v ← e Vorstellung: Designator v repräsentiert einen Speicherplatz der Maschine; das vom Ausdruck e gelieferte Ergebnis wird an diesem Speicherplatz gebracht

v → e Vorstellung: an das vom Ausdruck e gelieferte Ergebnis wird ein Designator angeheftet

Der für den Designator vereinbarte Typ muß übereinstimmen mit dem Typ des vom Ausdruck gelieferten Ergebnisses. Allerdings erlaubt MODULA-2 einige Ausnahmen davon, insbesondere kann ein CARDINAL-Wert einem INTEGER-Designator zugewiesen werden. Genaueres findet man in Büchern unter dem Stichwort "assignment compatible".

Bsp:

```
MODULE teiler;
FROM InOut IMPORT WriteInt, WriteLn;
VAR x, y : INTEGER;
BEGIN   x := 12 ; y := 9 ;
        WHILE x # y DO
            IF x < y THEN y := y - x
            ELSE (* x > y *)    x := x - y
            END
        END;
        WriteInt (x, 10) ; WriteLn
END teiler.
```

In diesem konkreten Beispiel erfolgt keine Eingabe von draußen; das Programm arbeitet nur für x = 12 und y = 9

Die folgende Tabelle zeigt den Verlauf der durch x und y im Programm berechneten Werte. Im Programm werden stets auch zwei weitere Werte, nämlich x # y und x < y berechnet; diese Werte erhalten keine ausdrückliche Bezeichnung im Programm und können somit nicht direkt angesprochen werden.

x : INTEGER	y : INTEGER	x # y : BOOLEAN	x < y : BOOLEAN
12	9	TRUE	FALSE
3	6	TRUE	TRUE
	3	TRUE	TRUE
		FALSE	

Werte abwechselnd über die Bezeichner x, y zugänglich

Werte "flüchtig"

In § 3 kamen in Beispielen sog. <u>kollektive Wertzuweisungen</u> vor:

$$(u,v) := (x,y)$$
$$(u,v) := (\,\mathrm{INC}(u)\,,\,\mathrm{DEC}(v)\,)$$

$$(s,ü,x,y) := (\ \mathrm{anfüg}(\mathrm{vas}(\mathrm{letzt}(x),\mathrm{letzt}(y),ü),s)\ ,\ \mathrm{vaü}(\mathrm{letzt}(x),\mathrm{letzt}(y),ü)\ ,\ \mathrm{anfang}(x)\ ,\ \mathrm{anfang}(y)\ )$$

allgemein: $\underbrace{(v_1,\dots,v_n)}_{\text{paarweise verschiedene Designatoren (mit VAR vereinbart)}} := \underbrace{(e_1,\dots,e_n)}_{\text{Ausdrücke vom passenden Typ}}$

$(u,v) := (x,y)$ beschreibt $\left(\dots\left|\begin{matrix}x\\n_1\end{matrix}\right|\begin{matrix}y\\n_2\end{matrix}\right|\begin{matrix}u\\n_3\end{matrix}\left|\begin{matrix}v\\n_4\end{matrix}\right|\dots\right) \vdash \left(\dots\left|\begin{matrix}x\\n_1\end{matrix}\right|\begin{matrix}y\\n_2\end{matrix}\right|\begin{matrix}u\\n_1\end{matrix}\left|\begin{matrix}v\\n_2\end{matrix}\right|\dots\right)$

$(x,y) := (y+1,x+1)$ beschreibt $\left(\dots\left|\begin{matrix}x\\n_1\end{matrix}\right|\begin{matrix}y\\n_2\end{matrix}\right|\dots\right) \vdash \left(\dots\left|\begin{matrix}x\\n_2+1\end{matrix}\right|\begin{matrix}y\\n_1+1\end{matrix}\right|\dots\right)$

Mit solchen kollektiven Wertzuweisungen kann man häufig die gewünschten Zustandsüberführungen problemgerecht formulieren,

<u>aber</u>, in MODULA-2 <u>nicht</u> erlaubt,

<u>deshalb</u> <u>Sequentialisierung</u> erforderlich!

häufiger Phänomen:

"natürliche" Formulierung (des Problems) beinhaltet Parallelität,

Programmiersprache (zugeschnitten auf traditionelle Rechner) erzwingt strenge Serialität.

$(u, v) := (x, y)$ läßt sich serialisieren zu

$u := x$ $\quad v := y$ $\quad$ oder $\quad v := y$ $\quad u := x$ $\quad$ d.h. beliebig

$(x, y) := (y+1;\ x+1)$ läßt sich nicht direkt serialisieren:

z.B. $x := y+1$ ist nicht richtig, denn
$y := x+1$

$$\left(\ldots \left|\begin{matrix} x \\ 2 \end{matrix}\right| \begin{matrix} y \\ 3 \end{matrix} \ldots\right) \xrightarrow{(x,y):=(y+1,x+1)} \left(\ldots \left|\begin{matrix} x \\ 4 \end{matrix}\right| \begin{matrix} y \\ 3 \end{matrix} \ldots\right)$$

aber $\left(\ldots \left|\begin{matrix} x \\ 2 \end{matrix}\right| \begin{matrix} y \\ 3 \end{matrix} \ldots\right) \xrightarrow{x:=y+1} \left(\ldots \left|\begin{matrix} x \\ 4 \end{matrix}\right| \begin{matrix} y \\ 3 \end{matrix} \ldots\right)$

$$\xrightarrow{y:=x+1} \left(\ldots \left|\begin{matrix} x \\ 4 \end{matrix}\right| \begin{matrix} y \\ 5 \end{matrix} \ldots\right)$$

also: Sequentialisierung erfordert Umrechnung.

innen anwendbare Methoden

1) benutze neue (d.h. sonst nicht gebrauchte) "Hilfs"-Designatoren $h_1, .., h_n$ für die Ergebnisse von $e_1, .., e_n$:

dann ist $(v_1, .., v_n) := (e_1, .., e_n)$

äquivalent (im Hinblick auf die Variablen durch $v_1, .., v_n$ bezeichneten Werte) zu

$$(h_1, .., h_n) := (e_1, .., e_n);$$
$$(v_1, .., v_n) := (h_1, .., h_n);$$

beide Anweisungen lassen sich stets bedeutungsäquivalent sequentialisieren

2) benutze neue "Hilfs"-Designatoren $h_1, .., h_n$ für die "alten" Werte von $v_1, .., v_n$

und ersetze in allen Ausdrücken v_i jeweils durch h_i:

dann ist $(v_1, .., v_n) := (e_1, .., e_n)$

äquivalent zu $(h_1, .., h_n) := (v_1, .., v_n);$
$(v_1, .., v_n) := (e_1[v_1/h_1, .., v_n/h_n], .., e_n[v_1/h_1, .., v_n/h_n])$

und beide Anweisungen lassen sich stets bedeutungsäquivalent sequentialisieren

Bsp belasse $(x, y) := (y+1, x+1)$

gemäß 1): $(h1, h2) := (y+1, x+1);$
$(x, y) := (h1, h2)$

in MODULA-2 sequentialisiert z.B.
```
h1 := y+1
h2 := x+1
x := h1
y := h2
```

gemäß 2) $(h1, h2) := (x, y)$;
$(x, y) := (h2+1, h1+1)$

in MODULA-2 sequentialisiert z.B
```
h1 := x;
h2 := y;
x := h2+1;
y := h1+1;
```

Empfehlung: Bei der Entwicklung eines Programms

- zunächst kollektive Verknüpfungen verwenden, wenn sie dem bearbeiteten Problem angemessen sind
- dann sequentialisieren
- entsprechend Kommunizieren: die sequentielle Notation läßt nicht mehr ohne weiteres die problemeigene Parallelität erkennen.

Aussage: $(v_1, \ldots, v_n) := (e_1, \ldots, e_n)$ ist äquivalent zu

$v_1 := e_1$;
$\vdots$
$v_n := e_n$;

falls für $i = 2, \ldots, n$ gilt: in e_i kommt $v_1, \ldots, v_{i-1}$ nicht vor.

Beweis: Übung!

11. Einführung in die axiomatische Semantik

Wir werden im Folgenden die Semantik von Anweisungen auf drei Ebenen angeben:

1) umgangssprachlich

2) operational durch
   - Flußdiagramme (bei strukturierten Anweisungen)
   - Übersetzung in ein Programm einer maschinennäheren Ebene (in §13)

3) axiomatisch durch Kalkültransformationen.

Dabei muß natürlich gelten:

- 1), 2), 3) müssen äquivalent sein; wir werden diese Äquivalenzen hier nicht nachweisen, sondern an die Intuition appellieren.

  Aber: dieses ist in der Informatik eine höchst wichtige Aufgabe:

  Bsp: Die Semantik von MODULA-2 kann (ein wesentlicher) (gemäß 3) axiomatisch definiert werden; diese Definition ist dann losgelöst von allen realen Computern.

  Einen MODULA-2 - Computer zu entwickeln heißt dann, (gemäß 2) eine Übersetzung von MODULA-2-Programmen in die "Maschinensprache" des verwendeten Computers festzulegen, deren Semantik bekannt sei.

  Forderung: Übersetzung soll korrekt sein in Bezug auf die axiomatische Festlegung.

- die umgangssprachliche Semantik soll möglichst genau die formalen Semantiken beschreiben

11.1a

Aussage: sprachliches Gebilde,
das bezüglich einer Interpretation
- der Funktionszeichen
- [der Prädikatenzeichen]
- [der Konstantenzeichen]
- bei gegebenem Universum

entweder wahr oder falsch ist,

z.B. 1) "3+3 ist gerade" ist wahr bzg. Interpretation:
+ : Addition
gerade: {0, 2, 4, 6, ...}
3 : drei
Universum: ℕ

2) "7 ist gerade" ist falsch bzg dieser Interpretation

3) "f.alle x : $x \overset{\text{ist}}{\vee}$ gerade oder $x \overset{\text{ist}}{\vee}$ ungerade" ist wahr bzg...

4) "f.alle x : $x+x \overset{\text{ist}}{\vee}$ gerade" ist wahr bzg...,

5) "f.alle x : x+1 ist gerade" ist falsch bzg...

Aussageform: sprachliches Gebilde mit "Platzhaltern"
(frei vorkommende Variablen):
bebgl man die "Platzhalter", so erhält man eine Aussage

z.B. "x ist gerade" ist wahr bei Belegung von x durch 3+3 (bzg. Interpretation s.o.);
ist falsch bei Belegung von x durch 7

11.1 b

Aussagen kann man auch über

einen Zustand oder

eine Zustandsüberführung

bilden.

Eine Aussageform enthält dann

"Platzhalter" für Zustände:

im folgenden

implizit durch $3, 3'$ aufgeschrieben,

explizit durch (einige) der im Zustand vorkommenden

Bezeichner, etwa für $3 = \begin{pmatrix} a & b & x & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix}$: a, b, x

Bedeutung von $3, 3'$: wenn betrachtet dann den

jeweils bezeichneten Wert.

zu 3) axiomatische Semantik

Sei S eine Anweisung, z.B. $x := a+b$

S bestimmt Zustandsüberführungen $z' \vdash z$ (vgl. § 2.8)

z.B. $\underbrace{\left(\ldots \middle| \begin{matrix} x \\ n_1 \end{matrix} \middle| \begin{matrix} a \\ n_2 \end{matrix} \middle| \begin{matrix} b \\ n_3 \end{matrix} \middle| \ldots\right)}_{z'} \overset{S}{\vdash} \underbrace{\left(\ldots \middle| \begin{matrix} x \\ n_2+n_3 \end{matrix} \middle| \begin{matrix} a \\ n_2 \end{matrix} \middle| \begin{matrix} b \\ n_3 \end{matrix} \middle| \ldots\right)}_{z}$

Für jeden vorkommenden Bezeichner muß man unterscheiden

bezeichneter Wert vor der Ausführung von S, abgekürzt durch mit ' versehenen Bezeichner

bezeichneter Wert nach der Ausführung von S, abgekürzt durch den Bezeichner

d.h. zur Vereinfachung der Schreibweisen identifizieren wir hier Bezeichner mit den bezeichneten Werten;

falls eindeutig: ' fortlassen

Über Zustände kann man Aussagen (sprachliche Gebilde, die entweder wahr oder falsch sind) machen.

z.B. $a' > 0 \wedge b' > 0$: wahr für $z_1' = \left(\ldots \middle| \begin{matrix} a \\ 3 \end{matrix} \middle| \begin{matrix} b \\ 4 \end{matrix} \middle| \ldots\right)$

$x > 0$: falsch für $z_2 = \left(\ldots \middle| \begin{matrix} x \\ -1 \end{matrix} \middle| \ldots\right)$

Über Zustandsüberführungen kann man ebenfalls Aussagen machen.

z.B. $(*\ a' > 0 \wedge b' > 0\ *)\ x := a+b\ (*\ x > 0\ *)$;

lies: "falls $a' > 0 \wedge b' > 0$ vor Ausführung von $x := a+b$, dann $x > 0$ nach Ausführung von $x := a+b$"

Definition[1]

Seien P, Q Aussagen (über Zustände) (genauer: Aussagenformen, die Bedingungen für Zustände ausdrücken: für jeden konkreten Zustand entweder wahr oder falsch)

S Anweisung (Programm)

(* P *) S (* Q *) : (sei wahr) : gdw

f. alle Zustände z':

Falls P wahr ist für den Zustand z' vor Ausführung von S

dann überführt S z' in einen Folgezustand z und

Q ist wahr für z

man sagt: S ist (streng) korrekt bzg. Vorbedingung P und Nachbedingung Q

diese Redeweise zieht daher:

P und Q 'spezifizieren' die gewünschten Zustandsüberführungen, bzw. eine Funktion;

S soll das Gewünschte durch eine Anweisung (Programm) beschreiben.

Diese Beschreibung ist korrekt genau dann, wenn

(* P *) S (* Q *) (wahr ist).

1) Bemerkung: in manchen Darstellungen der axiomatischen Semantik wird nicht erfordert, daß S einen Folgezustand erzeugt, d.h. S braucht nicht zu terminieren!

Die Idee der axiomatischen Semantik ist nun, daß man die Bedeutung von S dadurch festlegt, daß man alle Aussagen über Zustandsüberführungen angibt, die wahr sein sollen.

Problem: wie kann man alle solchen Aussagen angeben?

Lösung:

1) Man schreibt gewisse grundlegende Aussagen explizit als Axiome auf: diese seien per Axiom wahr.

2) Alle Folgerungen (im Sinne der Logik) aus den Axiomen sind dann ebenfalls wahr.

3) Nur die Axiome und ihre Folgerungen gelten als wahr.

d.h. man braucht nicht alle Aussagen, sondern nur die Axiome anzugeben.

Um die Axiome (für eine Anweisung S) ihrerseits so "stark" wie möglich zu machen, (aussagekräftig) geht man hier wie folgt vor:

man schreibt nicht einzelne Axiome auf, sondern Axiomenschemas $\mathcal{H}_Q$ gemäß folgender Idee:

- sei Q eine Aussage über Zustände
- dann betrachte man alle Aussagen $P_1, P_2, P_3, \ldots$, so daß $(\!* P_i *\!)\ S\ (\!* Q *\!)$ (wahr ist)
- unter diesen suche man eine schwächste Bedingung P

d.h. falls P_i für einen Zustand z wahr ist, dann ist auch P für z wahr

die schwächste Vorbedingung zu Q bezüglich S bezeichnet

man mit

$$wp(S, Q)$$

↑

weakest precondition

- als Axiome nimmt man dann, für alle Aussagen Q:

$$(* wp(S,Q) *) \quad S \quad (* Q *)$$

[dies ist als Axiomenschema aufgeschrieben: für jede Aussage Q wird dies ein Axiom]

Definition $wp(S, Q)$ ist durch folgende Bedingungen definiert:

1) $(* wp(S,Q) *) \quad S \quad (* Q *)$

2) falls $(* P *) \; S \; (* Q *)$, dann $\quad P \Rightarrow wp(S, Q)$

wp ist eine Prädikatstransformation.

Achtung: zwei Dinge auseinander halten:

(1) für die einzelnen Sprachelemente S von MODULA-2 wird $wp(S, Q)$ axiomatisch festgelegt. [für diese Axiome muß man allerdings prüfen, ob sie die umgangssprachliche Semantik richtig erfassen]

Mit der Festlegung der Prädikattransformation wp haben wir auch bestimmt, welche Zustandsüberführungen ein Programm S beschreibt (vgl. Redeweise aus §2.10).

Genauer:

Jedes Programm S beschreibt Zustandsüberführungen $\pi(S)$ der Form

$\pi(S) = \{ (z_a, z_e) \mid$ startet man Programm S im "Anfangszustand" z_a dann läuft das Programm mit "Endzustand" z_e $\}$

Die Semantik-Funktion π hat also die Form

$$\pi : \text{Menge der Programme} \longrightarrow \mathcal{P}(\text{Menge der Zustände} \times \text{Menge der Zustände})$$

($\mathcal{P}$: Potenzmenge)

Bei gegebener Prädikattransformation wp kann man π wie folgt bestimmen:

$$\pi(S) = \{ (z_a, z_e) \mid wp(S, \overbrace{x = z_e}^{Q(x)}) \text{ ist wahr für } z_a \}$$

(x: Bezeichner für Zustände)

Begründung

I. Beh: $wp(S, m = 3_e)$ wahr für $3_a \Rightarrow (3_a, 3_e) \in \pi(S)$.

Begründung:

(1) : Sei $wp(S, m = 3_e)$ wahr für 3_a.

Gemäß Eigenschaft 1) aus Def. für wp (Seite 11.5) gilt:

$$(* \; wp(S, m = 3_e) \; *) \quad S \quad (* \; m = 3_e \; *)$$

d.h. gemäß Def auf S. 11.3:

(2) $\underbrace{wp(S, m = 3_e)}_{P}$ wahr ist für 3_a vor Ausführung von S

dann überführt S 3_a in einen "Folgezustand" 3 und

$\underbrace{m = 3_e}_{Q(m)}$ ist wahr für 3

d.h. 3 ist gleich 3_e

aus (1) und (2) folgt $(3_a, 3_e) \in \pi(S)$.

II. Beh: $(3_a, 3_e) \in \pi(S) \Rightarrow wp(S, m = 3_e)$ wahr für 3_a

Begründung:

Sei $(3_a, 3_e) \in \pi(S)$, dann gilt

$$(* \; m = 3_a \; *) \quad S \quad (* \; m = 3_e \; *)$$

Gemäß Eigenschaft 2) aus Def. für wp (S. 11.5) folgt

$$m = 3_a \Rightarrow wp(S, m = 3_e)$$

insbesondere also: $wp(S, m = 3_e)$ wahr für 3_a

(2) Um die Korrektheit eines Programmes T in MODULA-2 bezüglich einer Spezifikation der Form < Vorbedingung P, Nachbedingung Q > nachzuweisen, muß man (*P*) T (*Q*) beweisen, d.h. man muß aus den Axiomen mit Mitteln der mathematischen Logik (*P*) T (*Q*) ableiten. Eine zweckmäßige Methode hierfür besteht aus folgenden zwei Schritten:

1) suche $wp(T, Q)$,

2) beweise $P \Rightarrow wp(T, Q)$;

denn dann folgt:

$P \Rightarrow wp(T, Q)$ "f. alle z': falls P wahr für z', dann $wp(T,Q)$ wahr für z'"

$$\frac{(*\ wp(T,Q)\ *)\ T\ (*Q*)}{(*P*)\ T\ (*Q*)}\ \text{log. Folgerung}$$

Übung: Nachprüfen, daß obige Schlußweise richtig!

Einige elementare Eigenschaften von wp

1) isoton: falls $Q_1 \Rightarrow Q_2$, dann $wp(S, Q_1) \Rightarrow wp(S, Q_2)$

2) konjunktionstreu:
$$wp(S, Q_1) \wedge wp(S, Q_2) \Leftrightarrow wp(S, Q_1 \wedge Q_2)$$

3) schwach disjunktionstreu:
$$wp(S, Q_1) \vee wp(S, Q_2) \Rightarrow wp(S, Q_1 \vee Q_2)$$

4) wunderfrei: $wp(S, \text{falsch}) = \text{falsch}$

5) gehorsam $wp(\overbrace{\text{"leere Anweisung"}}^{\text{skip}}, Q) = Q$

wir kommen zur Wertzuweisung zurück:

Wertzuweisungsaxiom

Sei $x := e$ eine Wertzuweisung, insbesondere e ein Ausdruck (ohne Seiteneffekte).

$$wp(x := e, Q(x)) \Leftrightarrow Q[^x/_e] \quad (\wedge\ e \text{ liefert Wert})$$

$Q[^x/_e]$: ersetze jedes Vorkommen von x durch e

($\wedge\ e$ liefert Wert): wichtig für praktische Arbeit, aber in Theorie oft vernachlässigt

Bsp $wp(x := \underbrace{y+3}_{\equiv e}, \underbrace{x > 0}_{\equiv: Q(x)}) \Leftrightarrow x > 0\,[^x/_{y+3}] \Leftrightarrow y+3 > 0$

$$wp(x := x+1,\ x \leq n+1) \Leftrightarrow x \leq n+1\,[^x/_{x+1}]$$

$$\Leftrightarrow x+1 \leq n+1$$

$$\Leftrightarrow x \leq n$$

für $Q(y)$, in dem x nicht vorkommt:

$$wp(x := e, Q(y)) \Leftrightarrow Q(y)[^x/_e] \Leftrightarrow Q(y)$$

d.h. Wertzuweisung $x := e$ lässt die durch $y \leftarrow e$ beschriebenen Werte unverändert (ohne "Seiteneffekt")

in der $v \leftarrow e$ Vorstellung, d.h. v repräsentiert einen Speicherplatz bedeutet dies:

verschiedene Bezeichner repräsentieren verschiedene Speicherplätze;
bringt (den Wert von e) zu einem Speicherplatz (v repräsentiert), so sind alle anderen Speicherplätze davon unberührt.

Definition

Seien P, Q Aussage(forme)n über Zustände,

S Anweisung (Programm).

$(*\ P\ *)\quad S\quad (*\ Q\ *)\qquad$ (sei wahr)

P: Vorbedingung — Q: Nachbedingung

: gdw für alle Zustände z':

falls P wahr ist für den Zustand z' vor Ausführung von S,

dann überführt S z' in einen Folgezustand z und Q ist wahr für z.

(S streng korrekt bez. P und Q)

einige Blickwinkel:

1. Wirkung von Anweisungen ist bekannt:
   welche Aussagen über strenge Korrektheit gelten?
2. Wirkung von Anweisungen soll definiert werden:
   durch Angabe derjenigen Aussagen über strenge Korrektheit, die gelten sollen!

zu 1.: Wirkung von Anweisungen ist bekannt:

Definition: Die Prädikattransformation $wp(S, Q)$ ist definiert durch

1. $(*\ wp(S,Q)\ *)\quad S\quad (*\ Q\ *)$
2. falls $(*\ P\ *)\quad S\quad (*\ Q\ *)$
   dann $P \Rightarrow wp(S, Q)$

Die Transformation wp liefert

zu jeder Aussage(form) Q als gewünschte Nachbedingung

jedem Programm S

diejenige schwächste Aussage(form) P,

die als Vorbedingung $(*\ P\ *)\quad S\quad (*\ Q\ *)$

wahr macht.

wp transformiert "sprachliche Gebilde".

zu 2.: Wirkung von Anweisungen soll definiert werden:

- definiere eine Prädikattransformation $wp(S, Q)$
- die Wirkung von S wird axiomatisch festgelegt durch die Forderung, daß alle Aussagen der folgenden Form gelten sollen:

$$(*\ wp(S, Q)\ *)\quad S\quad (*\ Q\ *)$$

Verbindung von 1. mit 2.

A) Die beabsichtigte Wirkung von Anweisungen sei anschaulich, umgangssprachlich bekannt:

Bestimme daraus die Prädikattransformation wp!

B) Benutze wp zur axiomatischen Festlegung der Wirkung von Anweisungen.

Die beabsichtigte Wirkung von Anweisungen ist dann formal definiert.

C) Dann kann man von einem Programme T formal beweisen, daß es seine formale Spezifikation, gegeben durch eine Vorbedingung P und eine Nachbedingung Q, erfüllt:

Beweise $(*\ P\ *)\ T\ (*\ Q\ *)$

als logische Folgerung aus den Axiomen.

D) Eine zweckmäßige Beweismethode für

$\cdot\ (*\ P\ *)\ \ T\ \ (*\ Q\ *)$:

1. Bestimme $wp(T, Q)$!

2. Beweise $P \Rightarrow wp(T, Q)$!

logische Folgerung

$(*\ P\ *)\ \ T\ \ (*\ Q\ *)$

elementare Eigenschaften von wp :

unter Blickwinkel 1: als Folgerung aus der bekannten Wirkung
unter Blickwinkel 2: Axiome dementsprechend gewählt

1. isoton : falls $Q_1 \Rightarrow Q_2$,
   dann $wp(S, Q_1) \Rightarrow wp(S, Q_2)$

2. konjunktionstreu: $wp(S, Q_1) \wedge wp(S, Q_2)$
   $\Leftrightarrow wp(S, Q_1 \wedge Q_2)$

3. schwach disjunktionstreu:
   $wp(S, Q_1) \vee wp(S, Q_2)$
   $\Rightarrow wp(S, Q_1 \vee Q_2)$

4. "wunderfrei" : $wp(S, \text{falsch}) \equiv \underbrace{\text{falsch}}_{\text{f. alle Zustände falsch}}$

5. "gehorsam" : $wp(\underbrace{\text{skip}}_{\text{leere Anweisung}}, Q) \equiv Q$

Prädikattransformation wp :

Fallunterscheidung nach dem syntaktischen Aufbau von Anweisungen:

Wertzuweisung: $wp(x := e, Q) \Leftrightarrow Q[x/e]$
und e liefert Wert

Zusammensetzung: $wp(S_1; S_2, Q)$
$\Leftrightarrow wp(S_1, wp(S_2, Q))$

Fallunterscheidung: $wp(\text{IF } B \text{ THEN } S_1 \text{ ELSE } S_2 \text{ END}, Q)$
$\Leftrightarrow (B \Rightarrow wp(S_1, Q)) \wedge (\neg B \Rightarrow wp(S_2, Q))$

12. Strukturierte Anweisungen

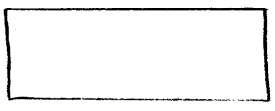
Auch große und mächtige Programme müssen schließlich (zumindest in einer maschinennahen Ebene) aus elementaren Anweisungen (Zuweisung im wesentlichen nur "Ausdruck auswerten und Wert bezeichnen", d.h. Wertzuweisung) aufgebaut werden. Damit die beabsichtigte Wirkung eines Programms, d.h. die beabsichtigte Klasse von Zustandsüberführungen, übersichtlich und durchschaubar beschrieben werden kann, sind in problemorientierten Programmiersprachen eine Reihe von "Grundstrukturen" zum Aufbau von Programmen verfügbar, und zwar in Form von sog. strukturierten Anweisungen. Das sind insbesondere

- Fallunterscheidungen
- Wiederholungen

Der Struktur dieser Anweisungen muß nun erkennbar gemacht werden:

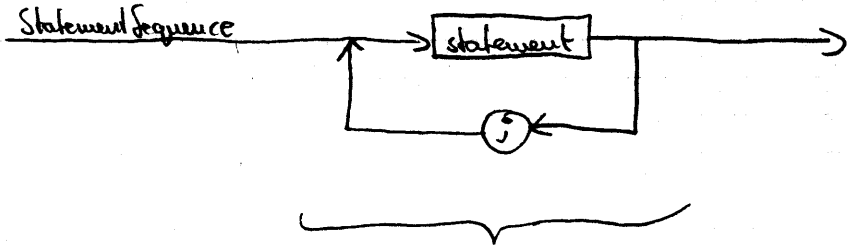
- Da für die Mitteilung an den Rechnerarchive auch solche strukturierten Anweisungen linear als Zeichenfolgen aufgeschrieben werden müssen, wird die Struktur durch spezielle Wortsymbole (Trennzeichen) angezeigt, z.B. IF, THEN, ELSIF, ELSE, END, WHILE, DO, REPEAT, FOR, CASE, ...

- Für die Entwicklung eines Programms, bzw. die Mitteilung an einen Menschen empfiehlt es sich häufig, die Struktur durch geschickte flächenmäßige Aufschreibung sichtbar zu machen. Ein guter Hilfsmittel dafür sind Nassi-Shneiderman Programme, die aus ineinander einsetzbaren Kästen für Anweisungen (elementare oder strukturierte) aufgebaut werden:

Anweisung: 

1. Zusammensetzungen

Anweisungen werden in MODULA-2 sequentiell bearbeitet, und zwar entsprechend der Reihenfolge des Aufschreibens in einem Block:



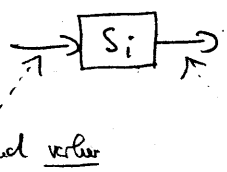
Folge von Anweisungen, jeweils durch ; getrennt

Sei $S_1 ; S_2 ; \dots ; S_n$ eine Folge von Anweisungen

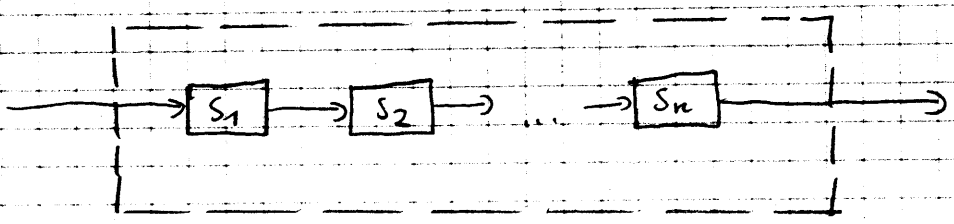
umgangssprachliche Semantik: führe $S_1, \dots, S_n$ sequentiell aus

durch Flußdiagramm:

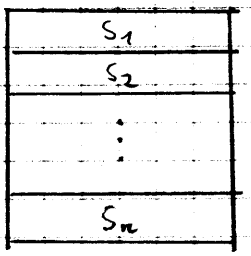
die durch S_i beschriebene Zustandsüberführung sei dargestellt durch den "schwarzen Kasten"



dann wird die durch $S_1 ; \dots ; S_n$ beschriebene Zustandsüberführung dargestellt durch den "schwarzen Kasten":



Nassi-Shneidermann-Diagramm



axiomatische Semantik:

Zusammensetzungsaxiom:

$$wp(S_1; S_2, Q) \Leftrightarrow wp(S_1, wp(S_2, Q))$$

Bsp: $wp(y := x ; z := x, y > 0 \wedge z > 0)$

Zusammensetzungs-Axiom
$\Leftrightarrow$ $wp(y := x, wp(z := x, y > 0 \wedge z > 0))$

Wertzuweisungs-Axiom
$\Leftrightarrow$ $wp(y := x, y > 0 \wedge x > 0)$

Wertzuweisungs-Axiom
$\Leftrightarrow$ $x > 0 \wedge x > 0$

$\Leftrightarrow$ $x > 0$

2. Fallunterscheidungen

Bei der Spezifikation eines Problems verwendet man häufig Fallunterscheidungen, z.B.

Addition (§3):

$$a+b = \begin{cases} (a+1)+(b-1) & \text{falls } b>0 \\ a & \text{falls } b=0 \end{cases}$$

ggT (§7):

$$ggt(x,y) = \begin{cases} x & \text{falls } x=y \\ ggt(x,y-x) & \text{falls } x<y \\ ggt(x-y,y) & \text{falls } x>y \end{cases}$$

Absolutwert:

$$|x| = \begin{cases} x & \text{falls } x \geq 0 \\ -x & \text{falls } x \leq 0 \end{cases}$$

lineare Gleichungen:

x löst $a \cdot x + b = 0$:

$$\begin{cases} x = \frac{-b}{a} & \text{falls } a \neq 0 \\ \text{"alle } x\text{"} & \text{falls } a=0 \text{ und } b=0 \\ \text{"kein } x\text{"} & \text{falls } a=0 \text{ und } b \neq 0 \end{cases}$$

allgemeinste Form: bewachte Anweisungen

```
IF   B1   THEN   S1
[]   B2   THEN   S2
:
[]   Bn   THEN   Sn   END
```

"Wächter"

wobei $B_1, \ldots, B_n$ Boolsche Ausdrücke und $S_1, \ldots, S_n$ Anweisungen sind

nichtdeterministische Semantik:

"suche ein i, so daß B_i liefert TRUE, und führe dann S_i aus"

Probleme:

1) was ist, wenn für alle i B_i FALSE liefert?
2) was ist, wenn für ein i B_i keinen Wert liefert (undefiniert ist)?
3) was ist, wenn die Fallunterscheidung nicht erschöpfend ist?
4) was ist, wenn für $i \neq j$ B_i und B_j den Wert TRUE liefern?

Antworten:

zu 1) "Anweisung undefiniert"
oder "Anweisung läßt Zustand unverändert"

zu 2) Schwieriges Problem: die Frage, ob ein Ausdruck einen Wert liefert ist äquivalent zum Halteproblem, d.h. nicht entscheidbar!

Falls die "Rechenmaschine" nicht schon einen anderen Wächter gefunden hat, der TRUE liefert, muß die Berechnung von B_i immer fortgesetzt werden (die Maschine kann nicht wissen, wann sie aufhören soll)

meistens: "Anweisung insgesamt undefiniert"

zu 3)

sollte man stets vermeiden: das sind häufig die vergessenen Spezialfälle!

also nur vollständige bewachte Anweisungen verwenden.

zu 4)

dann ist die Anweisung nichtdeterministisch, das Problem also noch nicht hinreichend genau spezifiziert (was in einem frühen Schritt der Programmentwicklung durchaus sinnvoll sein kann!).

Die bewachte Anweisung beinhaltet Parallelität:

"suche ein i, so daß B_i liefert true" kann so durchgeführt werden, daß alle B_i parallel ausgewertet werden

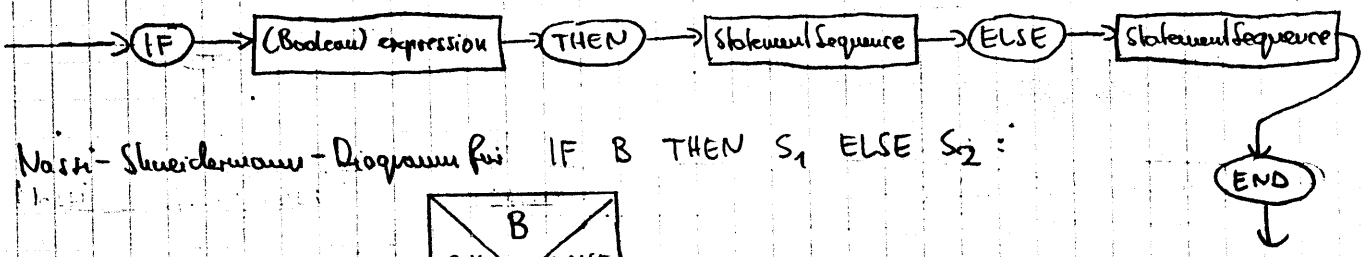
in MODULA-2 ist die bewachte Anweisung so nicht erlaubt;

MODULA-2 stellt einen sequentiellen Spezialfall zur Verfügung.

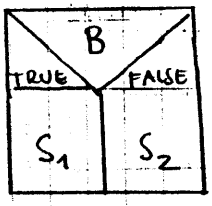
Empfehlung: Bei der Entwicklung eines Programm

- zunächst vollständige bewachte Fallunterscheidung verwenden
- dann eventuellen Nichtdeterminismus entfernen
- dann durch MODULA-2-Konstruktion sequentialisieren

2.1 zweigliedrige Fallunterscheidung



Nassi-Shneiderman-Diagramm für IF B THEN S_1 ELSE S_2:



entspricht der betrachteten Anweisung (in MODULA-2 notiert)

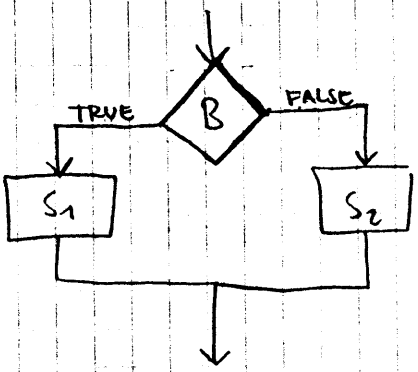
IF B THEN S_1
| NOT B THEN S_2 END

Bsp: IF x > 0 THEN x := x - 1 ELSE x := 0 END

umgangssprachliche Semantik:

"Falls der Ausdruck den Wert TRUE liefert, dann führe S_1 aus,
falls der Ausdruck den Wert FALSE liefert, dann führe S_2 aus" –

Flußdiagramm:



axiomatische Semantik:

Fallunterscheidungsaxiom

$$wp(\text{IF } B \text{ THEN } S_1 \text{ ELSE } S_2 \text{ END}, Q)$$

$$\Leftrightarrow \big((B \Rightarrow wp(S_1, Q)) \wedge (\neg B \Rightarrow wp(S_2, Q))\big)$$

es gilt: $(B \Rightarrow P_1) \wedge (\neg B \Rightarrow P_2)$

$$\Leftrightarrow [(\neg B \vee P_1) \wedge (\underbrace{\neg\neg B}_{\Leftrightarrow B} \vee P_2)]$$

$$\Leftrightarrow [\underbrace{(\neg B \wedge B)}_{\Leftrightarrow \text{FALSE}} \vee (\neg B \wedge P_2) \vee (P_1 \wedge B) \vee (P_1 \wedge P_2)]$$

$$\Leftrightarrow [(B \wedge P_1) \vee (\neg B \wedge P_2) \vee (P_1 \wedge P_2)]$$

$$\Leftrightarrow [(B \wedge P_1) \vee (\neg B \wedge P_2) \vee (\underbrace{B \vee \neg B}_{\text{TRUE}} \wedge P_1 \wedge P_2)]$$

$$\Leftrightarrow [(B \wedge P_1) \vee (\neg B \wedge P_2) \vee (B \wedge P_1 \wedge P_2) \vee (\neg B \wedge P_1 \wedge P_2)]$$

$$\Leftrightarrow (B \wedge P_1) \vee (\neg B \wedge P_2)$$

also läßt sich das Fallunterscheidungsaxiom auch so ausdrücken:

$$wp(\text{IF } B \text{ THEN } S_1 \text{ ELSE } S_2 \text{ END}, Q)$$

$$\Leftrightarrow \big((B \wedge wp(S_1, Q)) \vee (\neg B \wedge wp(S_2, Q))\big)$$

Aufgabe: Der Bezeichner x sei vereinbart mit VAR x: INTEGER

beweise: (* TRUE *) IF x>0 THEN x:=x-1 ELSE x:=0 END (* x ≥ 0)

d.h. nach Ausführung der Anweisung bezeichnet x immer (P ⇐ TRUE)
eine nichtnegative Zahl (Q ⇔ x ≥ 0).

gemäß Methode aus §11 zu zeigen:

TRUE ⇒ wp(IF x>0 THEN x:=x-1 ELSE x:=0 END , x ≥ 0)

äquivalent zu zeigen: TRUE ⇐ wp(IF x>0 THEN x:=x-1 ELSE x:=0 END , x ≥ 0)

wp(IF x>0 THEN x:=x-1 ELSE x:=0 END, x ≥ 0)

Fallunterscheidungs-A
⇐ $(x>0 \Rightarrow wp(x:=x-1, x\geq 0)) \wedge (\neg(x>0) \Rightarrow wp(x:=0, x\geq 0))$

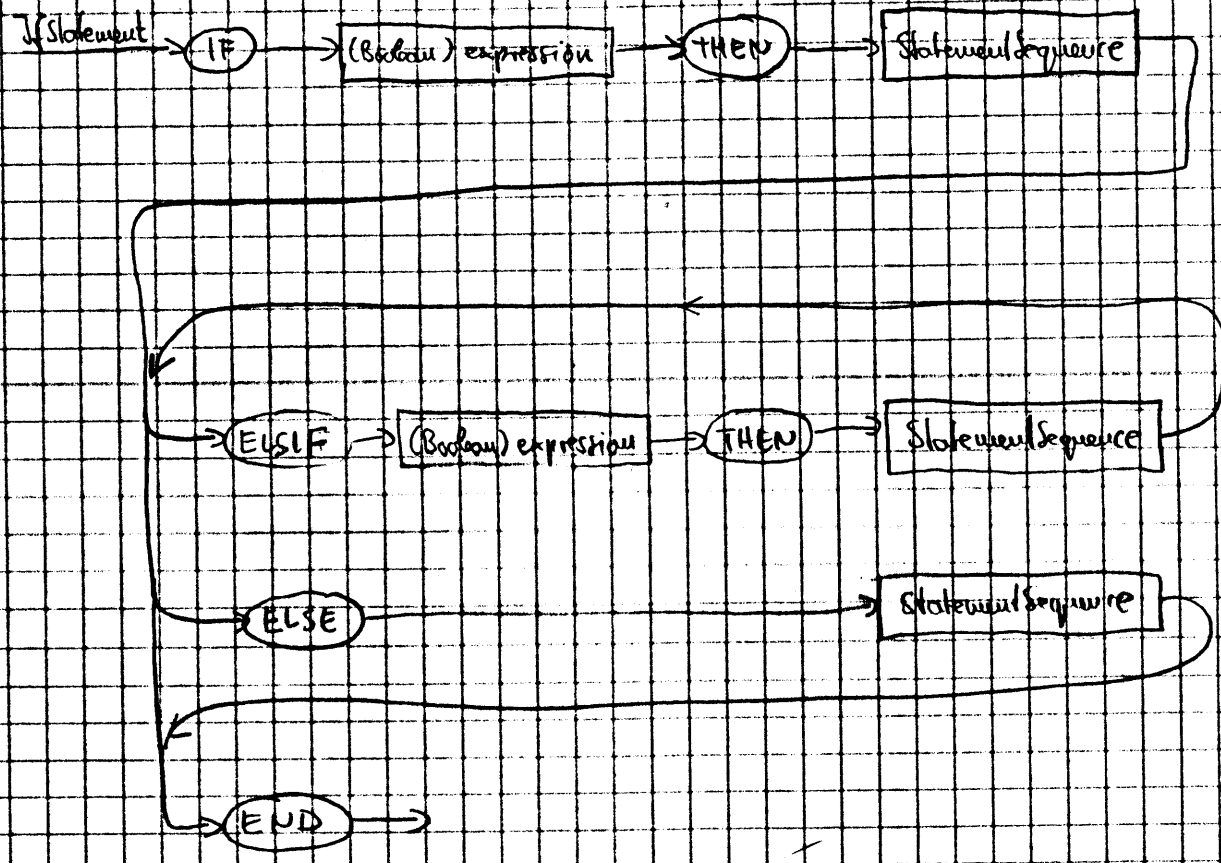
Verkettungs-A
⇐ $(x>0 \Rightarrow x-1\geq 0) \wedge (x\leq 0 \Rightarrow 0\geq 0)$

Arithmetik: wir setzen X:INTEGER voraus!
⇐ TRUE

2.2 mehrgliedrige Fallunterscheidung, IfStatement

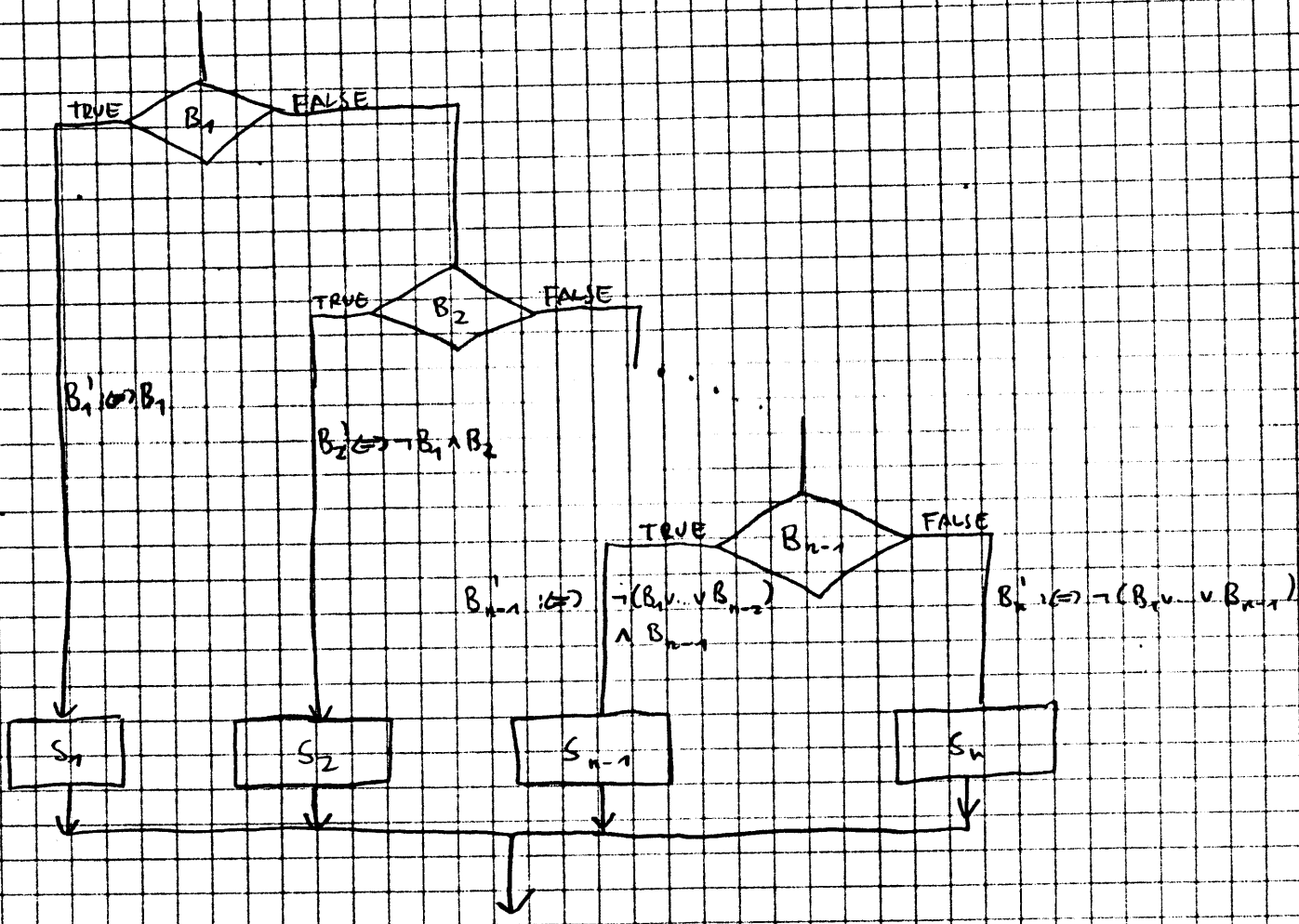
Mehrgliedrige Fallunterscheidungen können stets sequentialisiert werden (aber gewisse Spezialfälle sind auch anders handhabbar durch die Selektionsanweisung, vgl. § 12.).

Dazu stellt MODULA-2 das IfStatement zur Verfügung, von dem wir in 2.1 schon einen Spezialfall kennengelernt:




```
IF      B1      THEN    S1
ELSIF   B2      THEN    S2
:       :       :       :
ELSIF   Bn-1    THEN    Sn-1
ELSE                    Sn
END
```

Flußdiagramm:



Bei der Sequentialisierung darauf achten, daß

- häufigster Fall noch vorn
- "Rastloch" B_n' wirklich "alles erfaßt"

Wird in den Zweigen einer (zweigliedrigen) Fallunterscheidung einem mit BOOLEAN vereinbarten Designator TRUE bzw. FALSE zugewiesen, d.h. liegt die Form vor:

```
VAR b: BOOLEAN;
⋮
IF B THEN b:=TRUE ELSE b:=FALSE END;
```

so drückt man dies natürlich besser aus durch:

```
b := B
```

Beispiel: Lineare Gleichung $ax + b = 0$ für $a, b \in \mathbb{Z}$ lösen!

Falls eindeutige Lösung $\Leftrightarrow a \neq 0$: Funktion $lsg : \mathbb{Z} \times \mathbb{Z} \not\to \mathbb{R}$

$lsg(a,b)$ = Lösung von $ax = -b$

$= \frac{-b}{a}$

Falls keine eindeutige Lösung $\Leftrightarrow a = 0$: Funktion $kommentar : \mathbb{Z} \to Text$

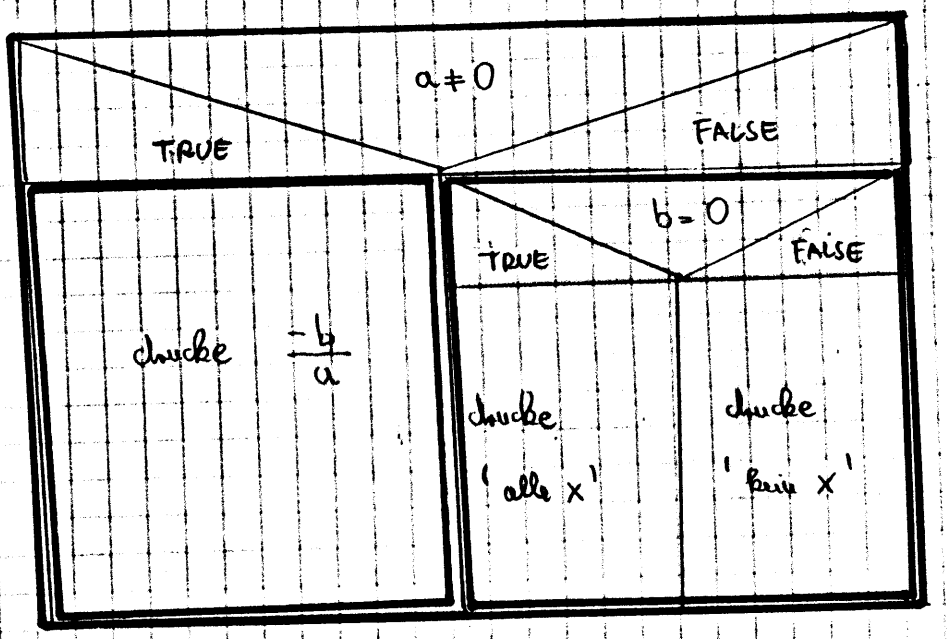
$$kommentar(b) = \begin{cases} \text{"alle x"} & \text{falls } b = 0 \\ \text{"kein x"} & \text{falls } b \neq 0 \end{cases}$$

Wächter

Wächter

```
IF a ≠ 0 THEN druck: lsg (a,b)
[] a = 0 THEN druck: kommentar (b); END
```

Nassi-Shneiderman-Diagramm:



wie kann man $\frac{-b}{a}$ für INTEGER-Zahlen b, a ausrechnen?

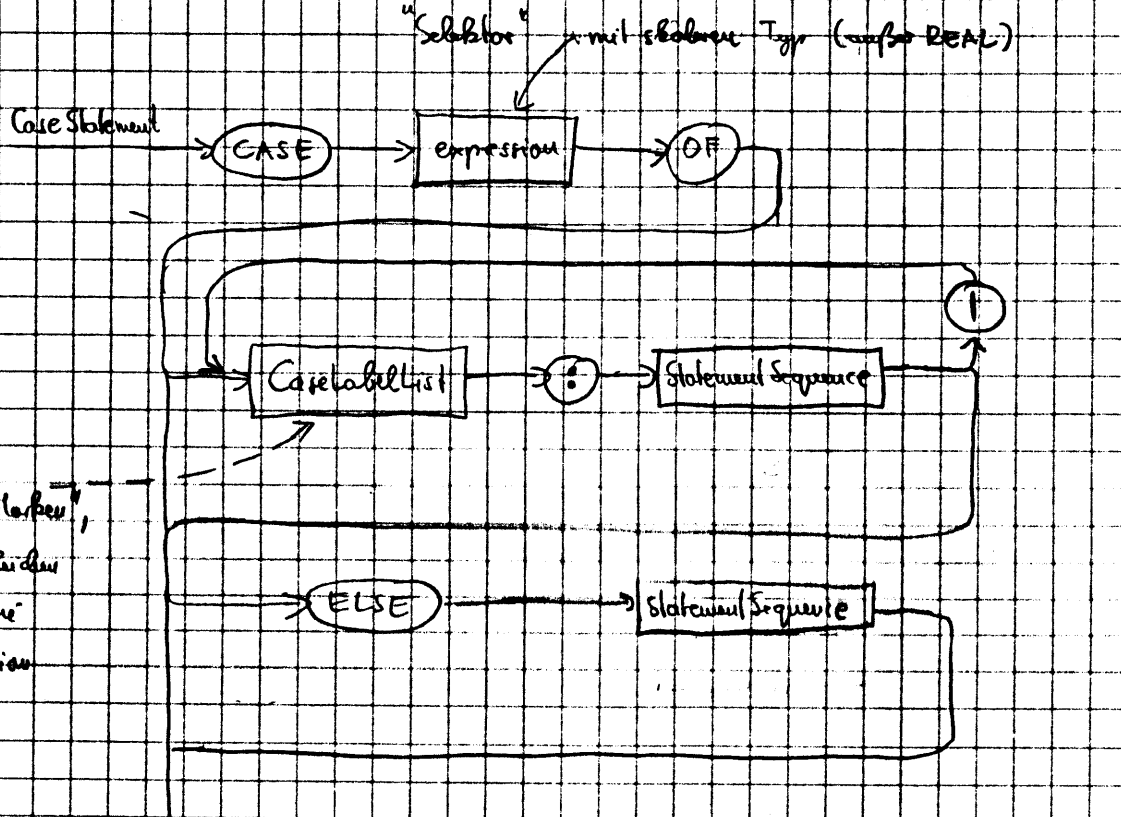
b \ a	> 0	< 0
> 0	- FLOAT(b) / FLOAT(a)	FLOAT(b) / FLOAT(-a)
< 0	FLOAT(-b) / FLOAT(a)	- FLOAT(-b) / FLOAT(-a)

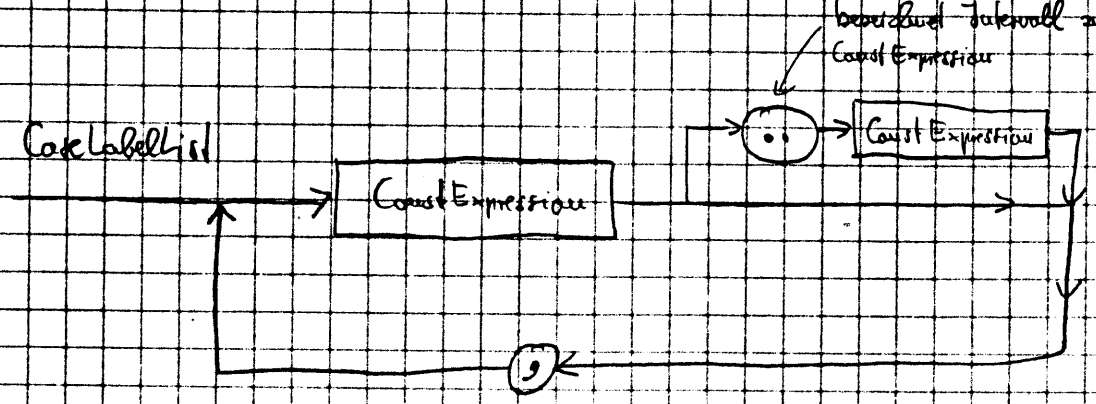
denn FLOAT : {0, 1, ...} → $\mathbb{R}^+$

```
MODULE LineareGleichung;
FROM InOut IMPORT ReadInt, WriteString, WriteLn;
FROM RealInOut IMPORT WriteReal;

VAR a, b: INTEGER      (* Argumente *)  ; Loesung : REAL;
BEGIN
  ReadInt(a); ReadInt(b);

  IF a # 0 THEN
        IF   b = 0                          THEN  Loesung := 0.0
        ELSIF (a > 0) AND (b > 0)           THEN  Loesung := - FLOAT(b) / FLOAT(a)
        ELSIF (a > 0) AND (b < 0)           THEN  Loesung := FLOAT(-b) / FLOAT(a)
        ELSIF (a < 0) AND (b > 0)           THEN  Loesung := FLOAT(b) / FLOAT(-a)
        ELSE (* (a<0) AND (b<0) *)                Loesung := - FLOAT(-b) / FLOAT(-a)
        END;
        WriteString("Loesung = "); WriteReal(Loesung, 20)
  ELSE (* a = 0 *)
        IF b=0 THEN WriteString("alle x") ELSE WriteString("kein x") END
  END;
  WriteLn
END LineareGleichung.
```

2.3 Selektionsanweisung, Case Statement

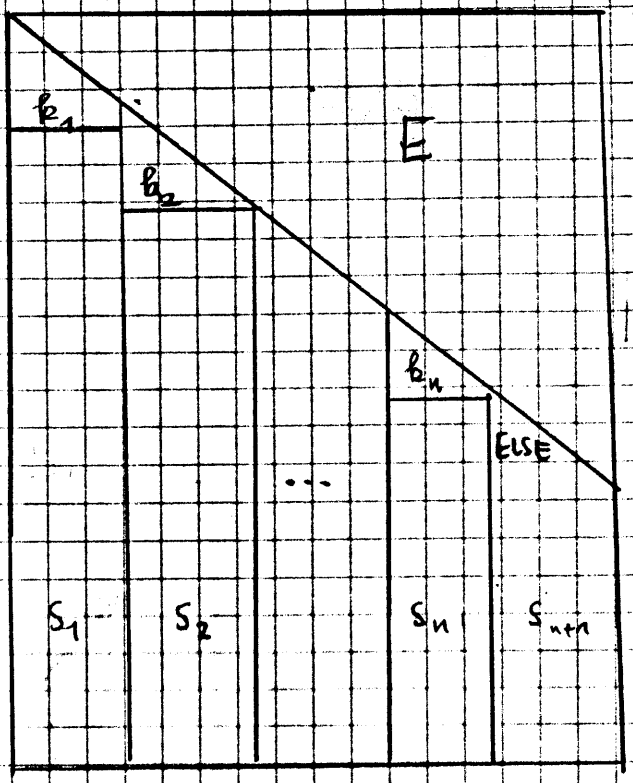
Haben die Wächter B_i eine Bewachten Anweisung der Form $B_i \Leftrightarrow E = k_i$, wobei die k_i verschiedene Werte (vom gleichen Typ) sind, so kann man in MODULA-2 die Selektionsanweisung verwenden:





Nassi-Shneiderman-Diagramm für

```
CASE  E  OF
   k1 : S1  |
   k2 : S2  |
   :
   kn : Sn
  ELSE  Sn+1
END
```



Bsp

```
TYPE Wochentag = (mon, die, mit, don, fre, sam, son);
VAR  zeit : Wochentag;
:

CASE  zeit  OF
   mon, die :  WriteString ("nachmittags")  |
   mit      :  WriteString ("frei")         |
   do       :  WriteString ("ganztägig")    |
   fre..son :  WriteString ("geschlossen")
END
```

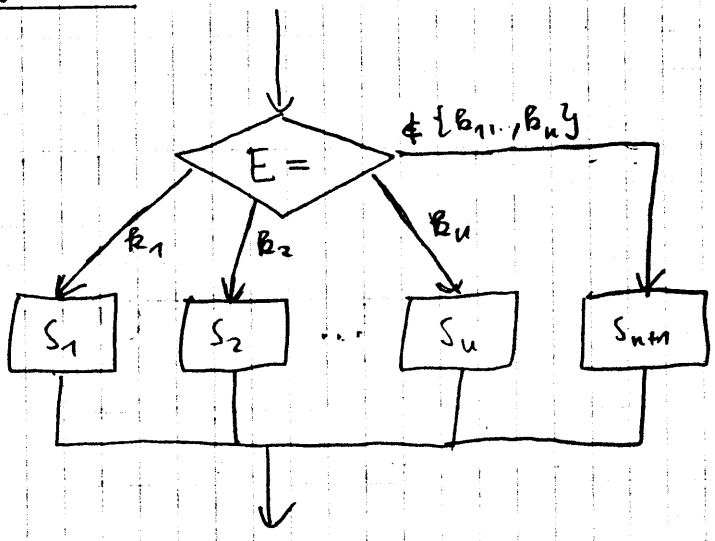
```
TYPE  op = ( plus, minus, mal );
VAR   z : op;
      x,y : INTEGER;
      :
CASE  z  OF
      plus  : x := x+y  |
      minus : x := x-y  |
      mal   : x := x*y
END
```

Ausgangssprachliche Semantik

" werte den "Selektor"-Ausdruck aus und führe diejenige Anweisung aus, die mit dem errechneten Wert markiert ist

(Anweisung wird übersprungen, falls keine "Voll-Marke" für errechneten Wert angegeben und der ELSE-Zweig fehlt.)

Flußdiagramm



axiomatische Semantik: „Schlussaxiom"

$$\text{wp}(\text{CASE } E \text{ OF } k_1 : S_1 \mid k_2 : S_2 \mid \ldots k_n : S_n \text{ ELSE } S_{n+1} \text{ END}, Q)$$

$$\Leftrightarrow (E = k_1 \Rightarrow \text{wp}(S_1, Q))$$

$$\wedge (E = k_2 \Rightarrow \text{wp}(S_2, Q))$$

$$\vdots$$

$$\wedge (E = k_n \Rightarrow \text{wp}(S_n, Q))$$

$$\wedge (E \neq k_1 \wedge \ldots \wedge E \neq k_n \Rightarrow \text{wp}(S_{n+1}, Q))$$

3. Wiederholung mit WHILE

alte Beispiele: Addition (§3), ggt (§7)

neues Beispiel: Addiere alle natürlichen Zahlen bis zur Zahl n:

$$\text{auflauf} : \mathbb{N} \to \mathbb{N}$$

$$\text{auflauf}(n) = \sum_{i=0}^{n} i = (0+1+2 \ldots + n)$$

rekursive Def.: $\underbrace{\underset{\substack{\uparrow \\ \text{auflauf}(0)}}{0} + 1 + 2 + \ldots + (n-1)}_{\text{auflauf}(n-1)} + n$

d.h. $\text{auflauf}(0) = 0$

$n > 0$: $\text{auflauf}(n) = \text{auflauf}(n-1) + n$

in der Schreibweise aus §3 (in MODULA-2 so nicht erlaubt!):

auflauf(n): <u>if</u> n=0 <u>then</u> 0 <u>else</u> auflauf(n-1)+n <u>fi</u>

liefert z.B. im Zustand $\binom{n}{3}$:

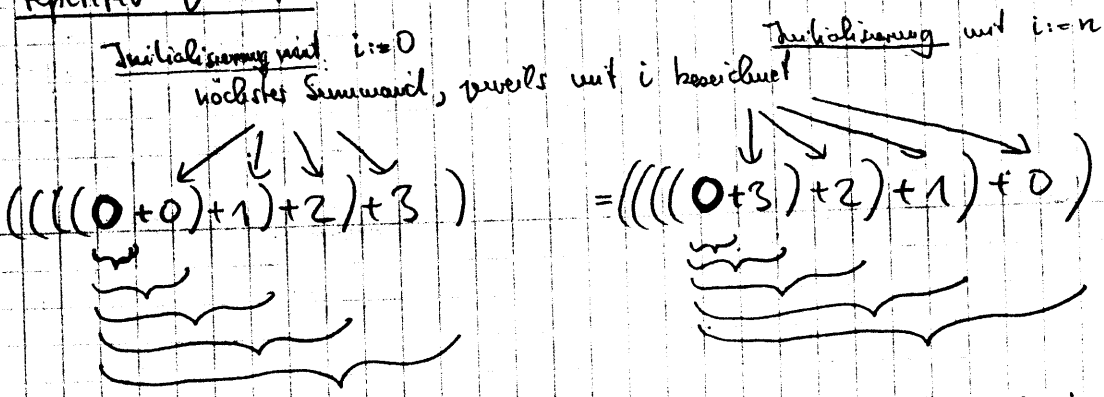
nicht keine Werte ausrechenbar

umklammern, damit <u>Teilergebnisse</u> <u>schon</u> ausrechenbar

$$\begin{cases} \text{auflauf}(3) \\ \text{auflauf}(2)+3 \\ (\text{auflauf}(1)+2)+3 = \text{auflauf}(1)+(2+3) \\ ((\text{auflauf}(0)+1)+2)+3 = \text{auflauf}(0)+(1+(2+3)) \end{cases}$$

$$((0+1)+2)+3 = 0+(1+(2+3))$$

repetitiv gedacht:



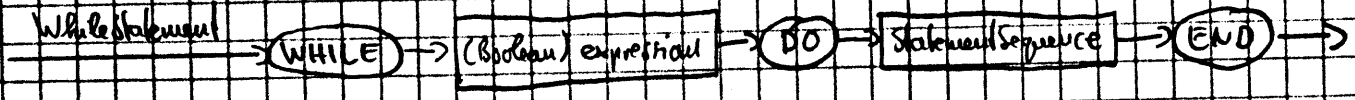
Zwischensumme jeweils mit summ bezeichnet;
Initialisierung mit summ := 0

```
summ := 0;
i := 0;
WHILE i <= n DO
    summ := summ + i;
    i := i+1
END;
```

```
summ := 0;
i := n;
WHILE i >= 0 DO
    summ := summ + i;
    i := i-1
END;
```

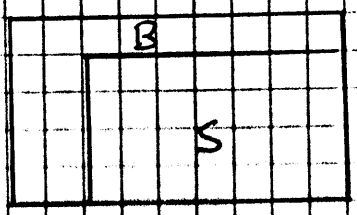
vollständiges MODULA-2 - Programm, z.B.

```
MODULE auflauf;
FROM InOut IMPORT ReadCard, WriteCard, WriteLn;

VAR n: CARDINAL    (* Argument *);
    summ: CARDINAL (* Ergebnis *);
    i : CARDINAL;
BEGIN
  ReadCard(n);
  summ := 0; i := 0;
  WHILE i <= n DO
      summ := summ + i; i := i+1
  END;
  WriteCard(summ, 10); WriteLn
END auflauf.
```



Nassi-Shneidermann-Diagramm für

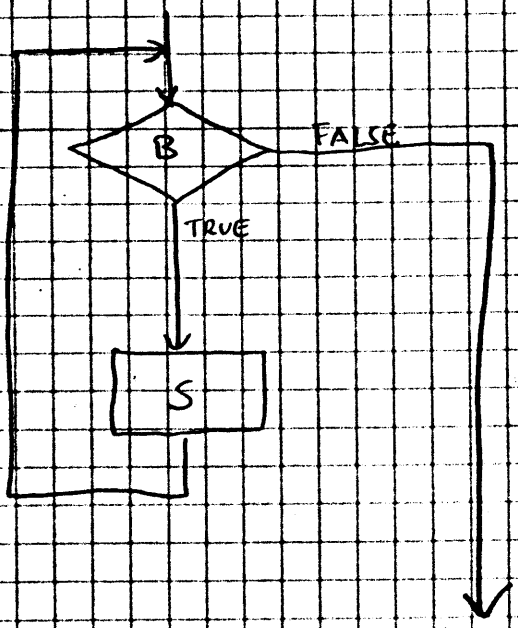
```
WHILE  B  DO
         S
END
```



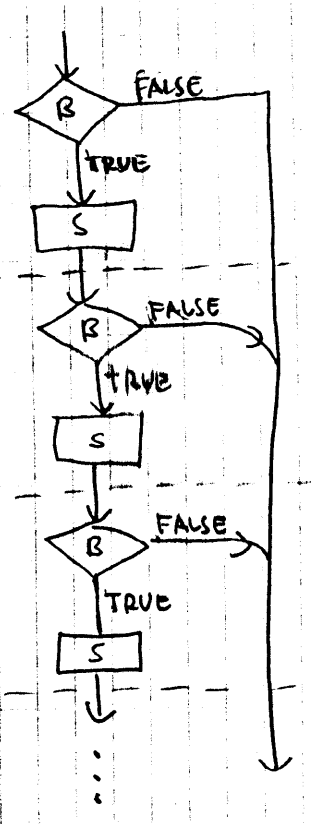
umgangssprachliche Semantik:

" Falls der Ausdruck den Wert FALSE liefert, so tue nichts;
Falls der Ausdruck den Wert TRUE liefert, so führe die
Anweisungen aus und wiederhole den gesamten Vorgang."

Flußdiagramm (für WHILE B DO S END):



offensichtlich ist dies gleichbedeutend mit:



d.h. WHILE B DO S END
ist äquivalent mit

```
IF B THEN
   S;
   WHILE B DO S END
END
```

äquivalent mit

```
IF B THEN
   S;
   IF B THEN
      S;
      WHILE B DO S END
   END
END
```

(bzw. in der Möglichkeit des (bedingten) Rücksprungs)

in dieser "Umschreibung" liegt die Mächtigkeit (und Gefährlichkeit) der Wiederholungsanweisung:

1) Liefert anfänglich B den Wert TRUE, so wird in die "Schleife" eingetreten.
Um aus der Schleife herauszukommen, muß B schließlich einmal den Wert FALSE erhalten, d.h. insbesondere die Anweisung S muß mindestens eine Wertzuweisung an einen in B vorkommenden Bezeichner enthalten (sonst könnte der Wert von B nie geändert werden; die Anweisung liefert kein Ergebnis; sie "kreist").

2) Die Anzahl der Wiederholungen muß hier nicht im voraus bestimmbar sein, aber wichtig ist: der Programmierer muß sicherstellen, daß stets nur endlich viele Wiederholungen möglich sind.

3) Die Wiederholungs-Anweisung reicht zusammen mit Wertzuweisung, Addition und Subtraktion im wesentlichen aus, um alle berechenbaren Funktionen zu beschreiben

[alle anderen Sprachelemente dienen der "Bequemlichkeit"]

4) Das Unentscheidbarkeitsproblem besagt, daß es kein HOCUS-FOCUS-Programm geben kann, das für eine beliebige Anweisung S und beliebigen Zustand Z feststellt, ob die Schleife S irgendwann wieder verlassen werden kann.

5) Neben der wünschbaren partiellen Definierbarkeit von Funktionen sind Wiederholungsanweisungen Ursache von undefinierten Ergebnissen.

axiomatische Semantik:

Das Axiom muß ausdrücken:

"es gibt i, so daß die Schleife noch i Durchläufen verlassen wird und dann gilt Q"

für festes i:

$i=0$: $\neg B \wedge Q \quad \Leftrightarrow: H_0$

$i=1$: $B \wedge wp(S, \underbrace{\neg B \wedge Q}_{\Leftrightarrow H_0}) \quad \Leftrightarrow: H_1$

$i=2$: $B \wedge wp(S, B) \wedge \underbrace{wp(S;S, \neg B \wedge Q)}_{\text{Zusammensetzwp-Ax}}$

$\underbrace{wp(S, wp(S, \neg B \wedge Q))}_{\text{Kompatibilität}}$

$wp(S, \underbrace{B \wedge wp(S, \neg B \wedge Q)}_{\Leftrightarrow H_1})$

allgemein: $H_{i+1} :\Leftrightarrow B \wedge wp(S,B) \wedge \dots \wedge wp(\underbrace{S;\dots;S}_{(i+1)\text{-mal}}, \neg B \wedge Q)$

$\Leftrightarrow B \wedge [wp(S,B) \wedge \dots \wedge wp(S, wp(\underbrace{S;\dots;S}_{i\text{-mal}}, \neg B \wedge Q))]$ Zusammensetzwp-Axiom

$\Leftrightarrow B \wedge [wp(S, B \wedge \dots \wedge wp(\underbrace{S;\dots;S}_{i\text{-mal}}, \neg B \wedge Q))]$ Kompatibilität

$\Leftrightarrow B \wedge wp(S, H_i)$

Wiederholungsaxiom (für WHILE)

Abstrakt wieder i Durchläufen

$wp(\text{WHILE } B \text{ DO } S \text{ END}, Q) \Leftrightarrow$ es gibt $i \geq 0 : H_i(Q)$. $(\Leftrightarrow H_0(Q) \vee H_1(Q) \vee \dots)$

mit $H_0(Q) :\Leftrightarrow \neg B \wedge Q$

$H_{i+1}(Q) :\Leftrightarrow B \wedge wp(S, H_i(Q))$

Schwierig zu handhaben: Abstrakte Beschreibung durch "es gibt i" usw.

"unendliche Disjunktion" ausgedrückt.

Für die auf S. 11.6 angesprochene Methode zum Korrektheitsbeweis, d.h. hier zum Beweis von (*P*) WHILE B DO S END (*Q*), ist das Widerholungsaxiom häufig zu unhandlich. Deshalb benutzt man hierfür meistens folgende Beweismethode:

man zeige: 1) die Schleife bricht ab
2) nach Verlassen der Schleife gilt Q

zu 2)

suche eine sogenannte Schleifeninvariante R mit folgenden Eigenschaften

a) , (* B ∧ R *) S (* R *) , d.h. die durch R ausgedrückte Eigenschaft des Zustands bleibt bei jeder Ausführung eines Schleifendurchlaufs erhalten; R ist eine Formalisierung der (jeweiligen) der Widerholungsanweisung zugrunde liegenden Programmieridee der Form: "Nach Ausführung von S liegt im wesentlichen wieder die gleiche Situation vor wie vor Ausführung von S, so daß man den Vorgang wiederholen kann".

b_1) $P \Rightarrow R$, d.h. vor Ausführung der Widerholungsanweisung gilt die Schleifeninvariante (in den Beispielen wird häufig P identisch mit R sein, so daß diese Eigenschaft trivial wird).

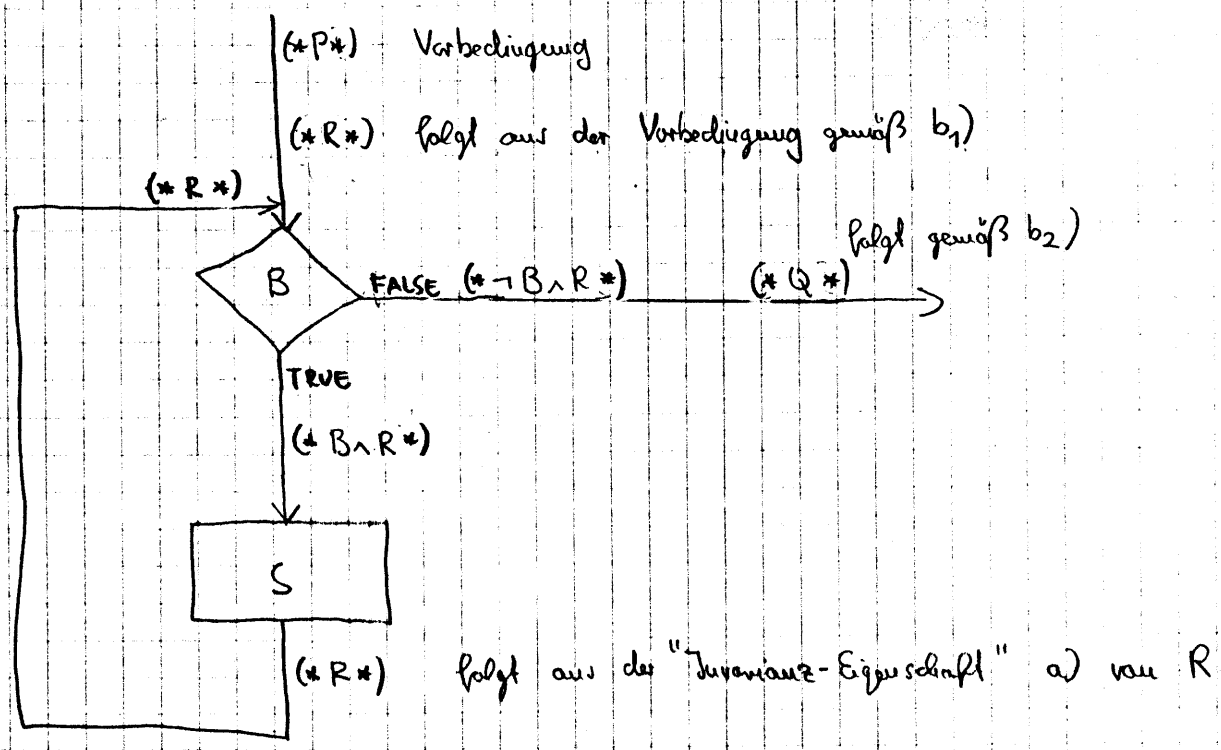
b_2) $\neg B \wedge R \Rightarrow Q$, d.h. bei Verlassen der Schleife ($\neg B$) sichert die Schleifeninvariante die gewünschte Nachbedingung Q.

Man kann dann nachprüfen, daß die Beweismethode richtig ist; genauer gilt folgende <u>Behauptung</u>:

<u>Falls</u> die Schleife abbricht [Teil 1) der Beweismethode] und es eine Schleifeninvariante R mit den obigen Eigenschaften gibt [Teil 2) der Beweismethode],

<u>dann</u> gilt (* P *) WHILE B DO S END (* Q *)

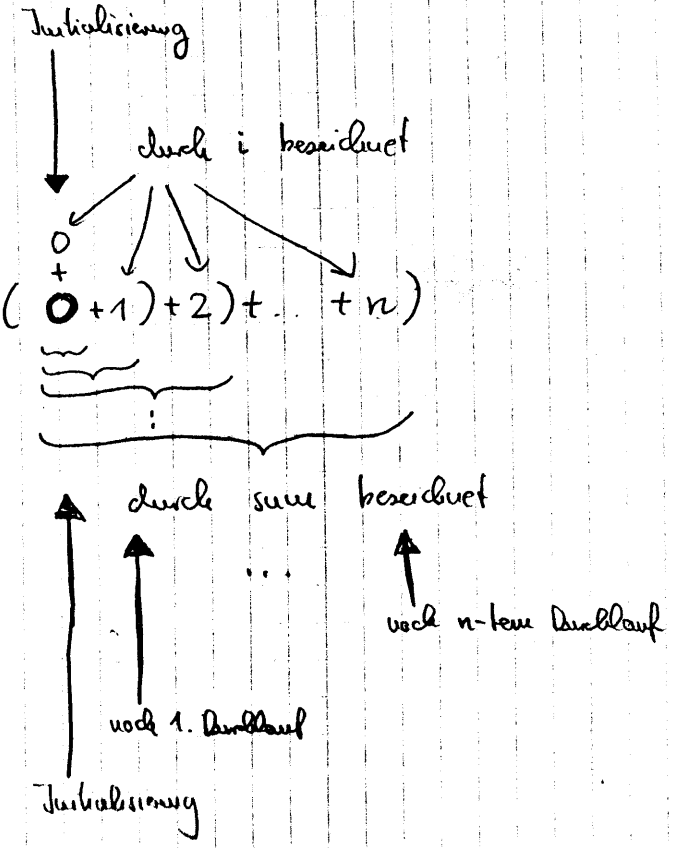
<u>anschauliche Rechtfertigung der Behauptung</u>:



<u>zu 1)</u> suche einen Ausdruck A,

- der Werte in einer <u>fundiert geordneten</u> Menge liefert (d.h. jede echt absteigende Kette $x_1 > x_2 > x_3 > \ldots$ bricht ab! Standardbeispiel $\mathbb{N} = \{0, 1, 2, \ldots\}$ mit natürlicher Ordnung)
- dessen gelieferte Werte nach jedem Schleifendurchlauf <u>echt kleiner</u> werden.

12.24



Initialisierung

durch i bezeichnet

Beispiel:

$$\text{aufkauf}(n) = \sum_{i=0}^{n} i = (\ldots(\underbrace{\underbrace{\underbrace{0}_{0}+1}_{+}+2}_{\vdots})+\ldots+n)$$

durch sum bezeichnet

...

noch n-tem Durchlauf

noch 1. Durchlauf

Initialisierung

Spezifikation: $(*\ 0 \le n\ *)\ S\ (*\ sum = 0+\ldots+n\ *)$

Nachbedingung: $Q :\Leftrightarrow sum = 0+\ldots+n$

$\Leftrightarrow sum = 0+\ldots+i \wedge i = n$

$\Leftrightarrow \underbrace{(sum = 0+\ldots+i \wedge 0 \le i \le n)}_{\Leftrightarrow R} \wedge \neg \underbrace{(i<n)}_{\Leftrightarrow :B}$

Schleifeninvariante

Wächter

Initialisierung:

$wp(sum := 0;\ i := 0,\ sum = 0+\ldots+i \wedge 0 \le i \le n)$

Zusammensetzungs-Ax. $\Leftrightarrow wp(sum := 0,\ wp(i := 0,\ sum = 0+\ldots+i \wedge 0 \le i \le n))$

Wertzuweisungs-Ax. $\Leftrightarrow wp(sum := 0,\ sum = \underbrace{0+\ldots+0}_{=0} \wedge \underbrace{0 \le 0}_{\Leftrightarrow \text{TRUE}} \le n)$

Wertzuweisungs-Ax. $\Leftrightarrow 0 = 0 \wedge 0 \le n$

$\Leftrightarrow 0 \le n$

Schleife: zu zeigen: $B \wedge R \Rightarrow wp(i := i+1 ; sum := sum + i , R)$ vgl. Seite 11.6

$\Leftrightarrow R$

$wp(i := i+1 ; sum := sum + i , \underbrace{sum = 0 + .. + i \wedge 0 \le i \le n}_{\Leftrightarrow R})$

Zusammensetzungs-Ax.

$\Leftrightarrow wp(i := i+1 , wp(sum := sum + i , sum = 0 + .. + i \wedge 0 \le i \le n))$

Wertzuweisungs-Ax

$\Leftrightarrow wp(i := i+1 , sum + i = 0 + .. + i \wedge 0 \le i \le n)$

Wertzuweisungs-Ax

$\Leftrightarrow sum + (i+1) = \underbrace{0 + .. + i}_{0 + .. + i + (i+1)} + (i+1) \wedge 0 \le i+1 \le n$

Arithmetik

$\Leftarrow sum = 0 + .. + i \wedge 0 \le i < n$

Arithmetik

$\Leftrightarrow \underbrace{i < n}_{\Leftrightarrow B} \wedge \underbrace{(sum = 0 + .. + i \wedge 0 \le i \le n)}_{\Leftrightarrow R}$

damit ergibt sich:

```
        (* 0 ≤ n *)
        sum := 0; i := 0;
R       (* sum = 0 + .. + i ∧ 0 ≤ i ≤ n *)
        WHILE i < n DO
B∧R         (* i < n ∧ (sum = 0 + .. + i ∧ 0 ≤ i ≤ n) *)
            i := i+1; sum := sum + i
R           (* sum = 0 + .. + i ∧ 0 ≤ i ≤ n *)
        END;
¬B∧R    (* ¬(i < n) ∧ (sum = 0 + .. + i ∧ 0 ≤ i ≤ n) *)
```

falls Schleife abbricht!

Abbruch: betrachte $A = n - i$

. sei A_i Wert von A nach i-ter Durchlauf

$A_0 = n - 0 = n \geq 0$

$A_{i+1} = n - (i+1) = (n-i) - 1 = A_i - 1 \geq 0$ wegen $i < n \Leftrightarrow i < n$

d.h. $A_0 > A_1 > \ldots_i \geq 0$

Bemerkung:

1) Typisch ist, daß Programm und Verifikation zusammen entwickelt werden.

2) Typisch ist, daß Konstruktion über den zugrunde liegenden Datentyp (hier: natürliche Zahlen) verwendet werden müssen.

3) Aktuelle Forschung:

algorithmische Unterstützung für Verifikation:
"induktive Beweiser", "Invarianten-Inspirierer"

Beispiel: Approximiere sinus von x.

Aufsummierung, jeweils durch summ bezeichnet

Es gilt: $\sin(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - + \ldots (-1)^n \frac{x^{2n+1}}{(2n+1)!} \pm \ldots$

Glieder, jeweils durch glied bezeichnet

"Laufzahl", jeweils durch lza bezeichnet

es gelten folgende Rekursionen:

```
(lza, glied, summ) := ( lza' + 2,
                        - glied' * x * x / ( (lza'+1) * (lza'+2) ),
                        summ' - glied' * x * x / ((lza'+1) * (lza'+2)) )
```

mit der Initialisierung:

```
(lza, glied, summ) := ( 1, x, x )
```

Abbruchbedingung:

Mathematik: die Folge $(summ_i)_{i \in \mathbb{N}}$ konvergiert.
Wir wollen abbrechen, falls das letzte Glied keinen "wesentlichen" Beitrag zur Aufsummierung mehr erbracht hat:

$$\frac{|glied|}{|summ|} < eps$$

Der Wert für EPS muß so gewählt werden, daß er größer als die Rechengenauigkeit des Typs REAL ist!

Die Rekursion läßt sich gleichfalls (d.h. Rechenzeit sparend!) sequentialisieren:

```
lza := lza' + 2;
glied := -glied' * x * x / ((lza-1) * lza);
summe := summe' + glied;
```

Die Rekursionsformeln enthalten Ausdrücke und konstante Werte:

```
x * x
1
```

Enthält ein solcher Ausdruck noch Funktionszeichen (hier: x * x), so durch einen Bezeichner ersetzen, dann ein Verb in der Initialisierung zuweisen wird: Herausziehen konstanter Werte.

Damit ergibt sich folgendes MODULA-2 - Programm:

```
MODULE sinus;
FROM InOut IMPORT WriteLn;
FROM RealInOut IMPORT ReadReal, WriteReal;
CONST eps = 0.1E-5;
VAR   x : REAL (*Argument*);
      summ : REAL (*Ergebnis*);

      lza, glied, xx : REAL;

BEGIN
ReadReal(x)

lza := 1;  glied := x;  summ := x;
xx := x*x;

WHILE  abs(glied) > eps * abs(summ)  DO
            lza := lza + 2;
            glied := - glied * xx /((lza-1)*lza);
            summ := summ + glied
END;

WriteReal(summ, 10); WriteLn
END sinus.
```

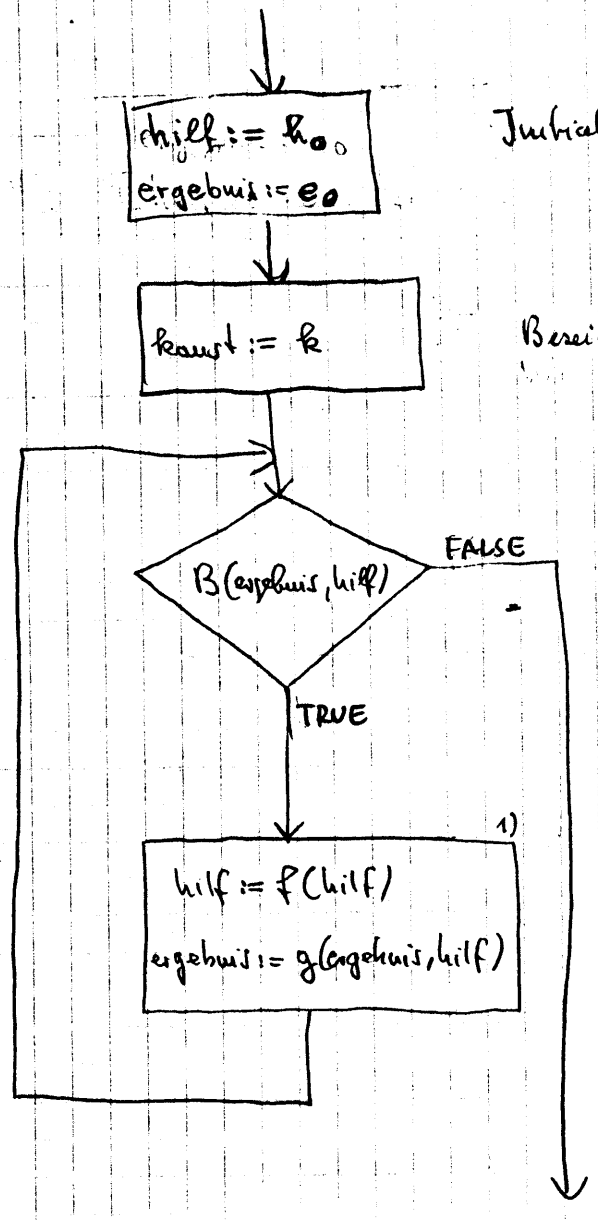
Initialisierung der Reihenglieder

Herausziehen des konstanten Ausdrucks!

Wörter

Reihenglieder

<u>allgemeines Schema:</u> in dieser oder ähnlicher Form oft anwendbar:



Initialisierung

Bezeichnen von konstanten Werten

Abbruchbedingung, Wächter

Rekursion

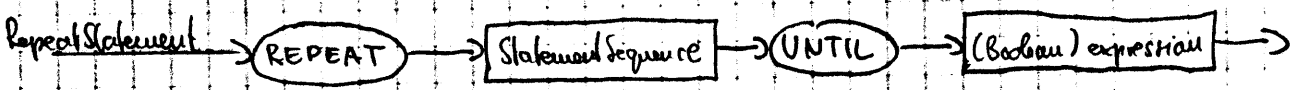
```
ergebnis := e0;
hilf     := h0;
WHILE B(ergebnis, hilf) DO
        hilf := f(hilf);
        ergebnis := g(ergebnis, hilf)
END;
```

1) Manchmal ist auch eine umgekehrte Sequentialisierung angebracht!

4. Wiederholung mit REPEAT

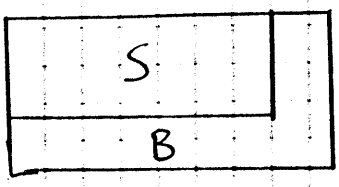
In WHILE B DO S END wird die Abbruchbedingung B getestet, bevor die Anweisung S ausgeführt wird.

Manchmal ist es zweckmäßig, dies nach der Ausführung von S zu tun.



Nassi-Shneidermann-Diagramm für

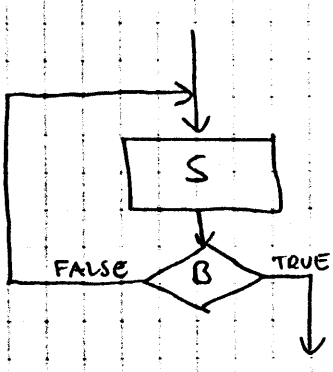
```
REPEAT  S
UNTIL   B
```



umgangssprachliche Semantik:

" führe Anweisungen aus;
falls dann der Ausdruck den Wert TRUE liefert, so beende die Wiederholung,
andernfalls wiederhole den Vorgang."

Flußdiagramm:



d.h. äquivalent mit

```
S;
WHILE NOT(B) DO S END;
```

beachte: S wird mindestens einmal ausgeführt!

axiomatische Semantik:

1) zurückführen auf Semantik von Zusammensetzung und WHILE-Anweisung
· gemäß obiger Ersatzkonstruktion;

2) entwickeln der Vorbedingung wp bei WHILE-Anweisung

$$wp(\text{REPEAT } S \text{ UNTIL } B, Q) \Leftrightarrow \underbrace{\text{es gibt } i \geq 1 : H_i(Q)}_{\Leftrightarrow H_1(Q) \vee H_2(Q) \vee \ldots}$$

mit $H_1(Q) :\Leftrightarrow wp(S, B \wedge Q)$

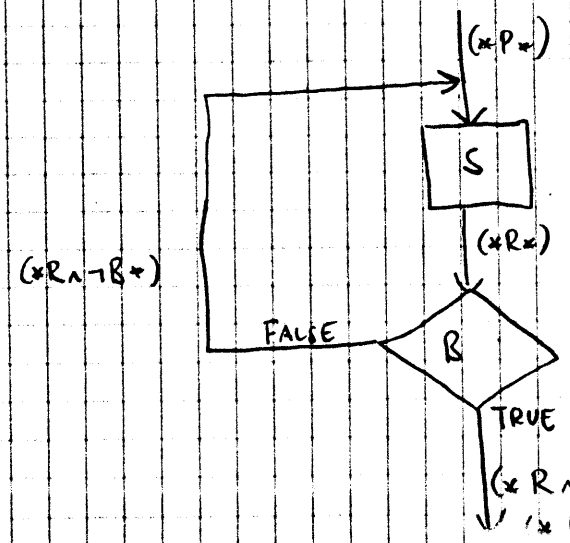
$H_{i+1}(Q) :\Leftrightarrow wp(S, \neg B \wedge wp(H_i(Q)))$

Für praktische Verwendung zum Nachweis von (* P *) REPEAT S UNTIL B (*Q*)

1) Schleife bricht ab (wie bei WHILE-Anweisung)
2) suche Schleifeninvariante R mit

i) (* P *) S (* R *)
ii) (* R ∧ ¬B *) S (* R *)
iii) R ∧ B ⇒ Q

falls Schleife abbricht, dann gilt (* P *) REPEAT S UNTIL B (* Q *)



5. Ein Spezialfall: beschränkte Wiederholung mit FOR

Das allgemeine Schema für die Verwendung der WHILE-Anweisung vereinfacht sich für den folgenden Spezialfall:

1) hilf bezeichnet im Verlauf der Wiederholung aufeinanderfolgende Werte eines skalaren Typs (außer REAL).

   Durch eine zusätzliche Angabe kann auch eine andere Schrittweite als 1 festgelegt werden.

2) Der Wächter B beobachtet nur hilf, d.h. er hat Form

   $$B(hilf)$$

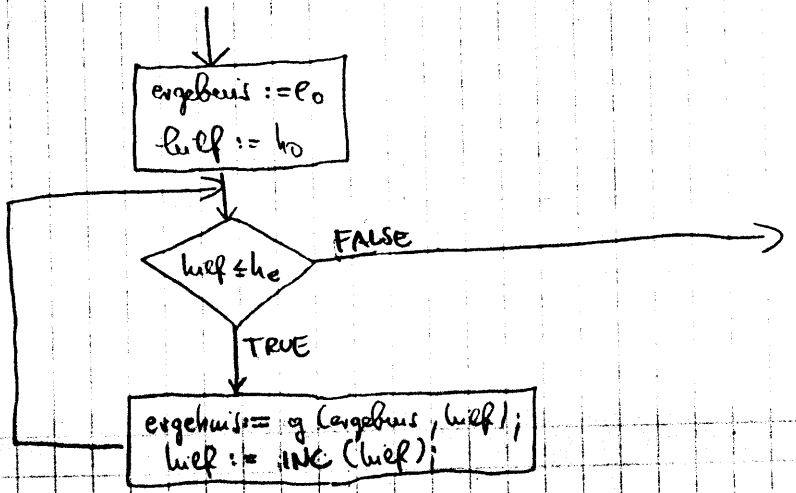
3) Die Anzahl der Wiederholungen kann im voraus errechnet werden, d.h. die Rekursion soll durchgeführt werden für

   $$h_0,\ INC(h_0),\ INC^2(h_0),\ \dots,\ INC^n(h_0) =: h_e$$

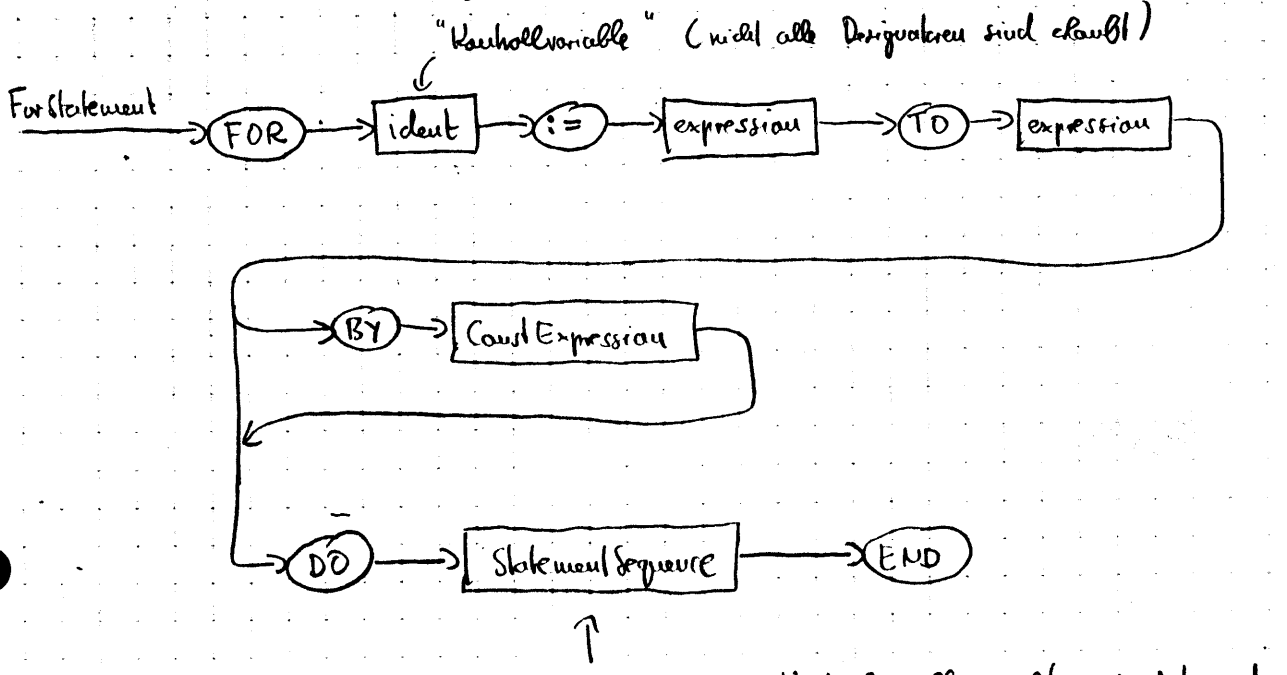
   wobei h_e im voraus errechnet werden kann. Dann hat der Wächter die Form

   $$hilf \leq h_e$$

Es liegt also folgender Spezialfall vor (Herausziehen der Konstanten vernachlässigt)



beschränkte Wiederholung, Laufanweisung



hier darf der Wert der Kontrollvariablen nicht geändert werden

Bsp
```
VAR i, sum, n : CARDINAL;
:
sum := 0      (* Initialisierung für Ergebnis *),
FOR i := 0 TO n DO sum := sum + i END;
```
↑ Initialisierung für Hilfsgröße (Kontrollvariable)

Bsp:
```
TYPE Farbe = (rot, gelb, grün, blau);
VAR  hilf : Farbe;
:
FOR  hilf := rot TO grün DO ... END;
```

Achtung: 1) nach Verlassen der Laufanweisung gilt der Wert der Kontrollvariablen als undefiniert! Die Kontrollvariable sollte lokal ausschließlich für diesen Zweck verwendet werden.

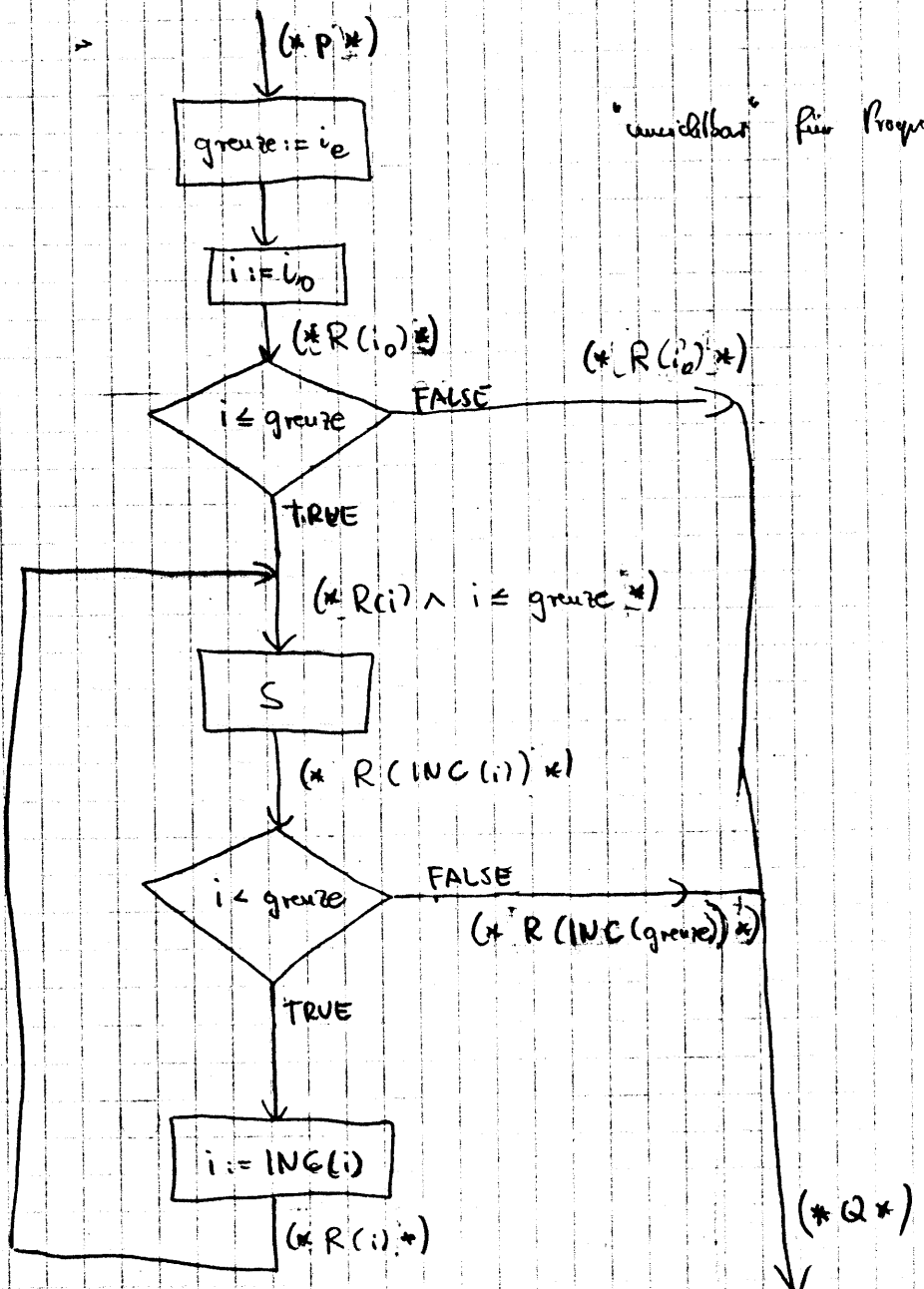
2) der den Anfangswert bzw. den Endwert, sowie der die Schrittweite festlegende Ausdruck werden nur einmal am Anfang ausgewertet, und dann werden die Werte intern festgehalten; sie können nicht mehr über die Anweisungen verändert werden

<u>umgangssprachliche Semantik:</u> FOR $i := i_0$ TO i_e DO S END

"führe S. aus mit den aufeinanderfolgenden Werten $i_0, \dots, i_e$"

[falls $i_e < i_0$, dann wie leere Anweisung]

<u>Flußdiagramm</u>



"unsichtbar" für Programmierer

axiomatische Semantik

enthält nicht den Bezeichner i !

$$wp(\text{FOR } i:=i_0 \text{ TO } i_e \text{ DO } S \text{ END}, Q)$$

$$\Leftrightarrow (i_0 > i_e \wedge Q)$$

$$\vee (i_0 \le i_e \wedge wp(i:=i_0; S; i:=INC(i); S; \ldots, i:=\boxed{i_e}; S, Q)$$

↑ Wert vor der for-Anweisung (grenze)

formale Verwendung:

zum Nachweis von (* P *) FOR $i:=i_0$ TO i_e DO S END (* Q *)

1) Schleifenabbruch ist trivialerweise immer gegeben!

2) suche Schleifeninvariante R mit

- R enthält u.a. den Bezeichner i (hängt von i ab)
- $P \Rightarrow R(i_0)$ [muß initialisiert durch geeignete Initialisierung der in Q interessanten Bezeichner sichergestellt werden]
- (* $R(i) \wedge i_0 \le i \le \boxed{i_e}$ *) S (* $R(INC(i))$ *)
- $\begin{cases} i_0 > i_e \Rightarrow [R(i_0) \Rightarrow Q] \\ i_0 \le i_e \Rightarrow [R(INC(\boxed{i_e})) \Rightarrow Q] \end{cases}$

↑ als MODULA-2-Ausdruck möglicherweise nicht definiert, aber zur Verifikation bei geeigneter Definition verwendbar!

Aufgabe: beschreibe $sum: \mathbb{Z} \to \mathbb{N}$ mit

$$sum(n) = \sum_{0 \le j \le n} j = \begin{cases} 0 & \text{falls } n \le 0 \\ 0+\ldots+n & \text{falls } n \ge 0 \end{cases}$$

noch nicht deterministisch!

```
IF  n ≤ 0  THEN  sum := 0
[]  n ≥ 0  THEN  sum := 0+...+n  END
```

es gilt:

$$0 = 0 + \sum_{0 \le j \le n} j \quad \text{für } n \le 0$$

$$0+\ldots+n = 0 + \sum_{0 \le j \le n} j \quad \text{für } n \ge 0$$

} d.h. beide Zweige können durch identische beschrieben

$$(n \le 0 \vee n \ge 0) \Leftrightarrow \text{TRUE}$$

d.h. wir erhalten durch Vereinfachen der Zweige

```
IF TRUE: THEN  sum := 0 + Σ_{0≤j≤n} j   END
```

so in MODULA-2 nicht hinschreibbar, also

oder axiomatisch: ein geeignetes Programm S genügt mit gleicher Wirkung

$$\underbrace{(*\ \text{true}\ *)}_{P}\ S\ \underbrace{(*\ sum = 0 + \sum_{0 \le j \le n} j\ *)}_{Q(n)}$$

Schleifeninvariante: $R(i) :\Leftrightarrow sum = 0 + \sum_{0 \le j < i} j$

dann gilt:

falls $0 \le n$: $R(n+1) \Leftrightarrow sum = 0 + \sum_{0 \le j < n+1} j \Leftrightarrow sum = 0 + \sum_{0 \le j \le n} j \Leftrightarrow Q(n)$

falls $n < 0$: $R(0) \Leftrightarrow sum = 0 + \sum_{0 \le j < 0} j \Leftrightarrow sum = 0 + \sum_{0 \le j \le n} j \Leftrightarrow Q(n)$

ferner für S gleich sum := sum + i :

$$(*\ \underbrace{sum = 0 + \sum_{0 \le j < i} j}_{\Leftrightarrow R(i)} \wedge \underbrace{0 \le i \le n}_{i_0 \le i \le i_e}\ *)\quad \underbrace{sum := sum + i}_{S}\quad (*\ \underbrace{sum = 0 + \sum_{0 \le j < i+1} j}_{\Leftrightarrow R(i+1)}\ *)$$

schließlich für Initialisierung von sum :

$$R(0) \Leftrightarrow sum = 0 + \sum_{0 \le j < 0} j$$

$$\Leftrightarrow \quad sum = 0$$

d.h. $\quad (*\ \underbrace{TRUE}_{0=0}\ *)\ sum := 0\ (*\ \underbrace{R(0)}_{sum=0}\ *)$

insgesamt erhält man:

```
MODULE summiere;
FROM InOut IMPORT ReadInt, WriteInt, WriteLn;
VAR  n : INTEGER (* Argument *);
     sum: INTEGER (* Ergebnis *);
     i : INTEGER;
BEGIN  ReadInt(n);
       (* TRUE *)
          sum := 0   (* R(0) *);
          FOR i := 0 TO n DO
                (* R(i) ∧ 0 ≤ i ≤ n *)
                  sum := sum + i
                (* R(i+1) *)
          END;
    (* (n < 0 ∧ R(0)) ∨ (0 ≤ n ∧ R(n+1)) *)
       WriteInt(sum, 10); WriteLn
END summiere.
```

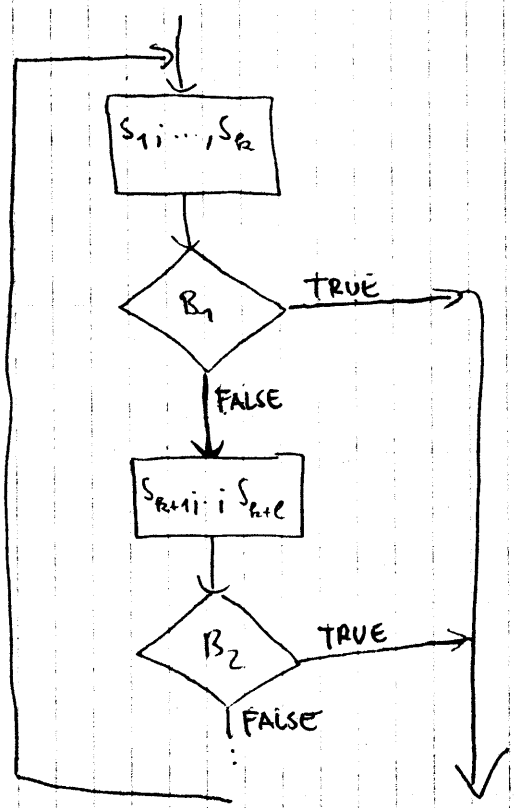
6. Allgemeinster Fall : Wiederholung mit LOOP

bisher: Abbruchbedingung wird vor oder nach Ausführung einer Anweisungsfolge getestet

allgemein: Anweisungsfolge mit Tests von Abbruchbedingungen innerhalb der Folge

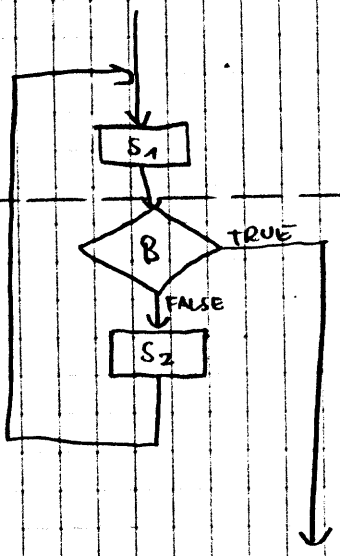
```
LOOP
    S1 ; ... ; Sk ;
    IF B1 THEN EXIT END ;
    Sk+1 ; ... ; Sk+l ;
    IF B2 THEN EXIT END ;
    :
END
```

Flußdiagramm



Einfache Fälle sind häufig mit WHILE oder REPEAT besser ausdrückbar,
z.B. betrachte folgende Schleife:

```
LOOP
    S1;
    IF B THEN EXIT END;
    S2
END
```



aufsprechen mit WHILE

Schleife wird nach "$\frac{1}{2}$ + n Durchläufen" verlassen.

In MODULA-2 mit WHILE-Anweisung:

1. Durchlauf / 1. Hälfte : Initialisierung

i-ter Durchlauf / 2. Hälfte
(i+1)-ter Durchlauf / 1. Hälfte
} : Wiederholung

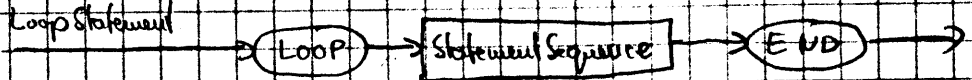
d.h.

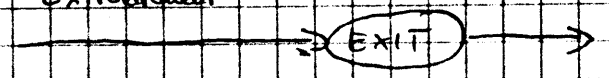
```
S1;
WHILE NOT(B) DO
    S2;
    S1
END
```

In Anwendungen kommt es darauf an, daß "$\frac{1}{2}$ + n Durchläufe" oder "n + $\frac{1}{2}$ Durchläufe" erforderlich sind, häufig ist:

gesucht Initialisierung oder Nachbehandlung.

allgemeine Form der LOOP-Anweisung:





Exit-Anweisungen dürfen nur innerhalb von LOOP ... END auftreten. Ihre Verwendung ist nur innerhalb einer prozeduralen Anweisung sinnvoll (sonst ist das Programmverhalten bis zum Suche des Loops nutzlos).

Eine typische Beispielsituation für die LOOP-Anweisung: führe eine Folge von Anweisungen zyklisch wiederholt durch; falls zwischendurch "Fehlersituationen" auftreten, so starte "Fehlerbehandlung" und breche die Wiederholung sofort ab.

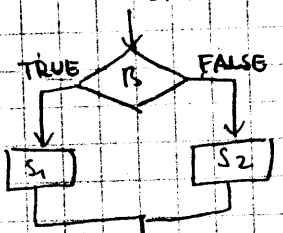
```
MODULE
  :
  LOOP
    :
    IF B1 (* Fehler 1 *) THEN Fehlerbehandlung1; EXIT END;
    :
    IF Bi (* Fehler i *) THEN Fehlerbehandlung2; EXIT END;
    :
  END;
```

Manche Programmiersprachen stellen prozedurale Sprachelemente für solche die Sicherheit steigernden Figuren bereit.

13. Übersetzung strukturierter Anweisungen in eine maschinenorientierte Sprache

Anweisungen werden in der Reihenfolge durchgeführt, in der sie aufgeschrieben sind. Wir haben bisher zwei Arten von Anweisungen kennengelernt, mit deren Hilfe man "im Programm springen" kann – sofern man die Flußdiagramm-Semantik der Angst hat – und damit also die Reihenfolgeregel durchbrechen kann:

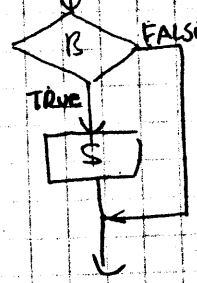
① IF B THEN S_1 ELSE S_2 END



führe S_1 aus und überspringe (im Programmtext) S_2

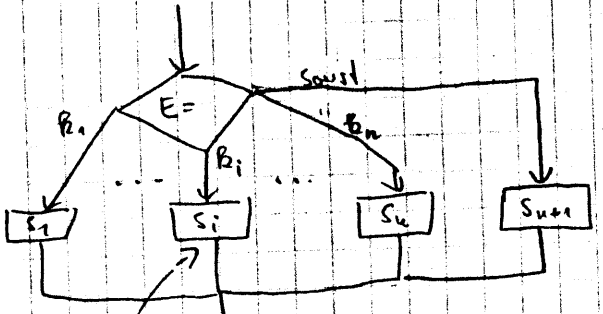
überspringe (im Programmtext) S_1 und führe S_2 aus

IF B THEN S END



überspringe (im Programmtext) S

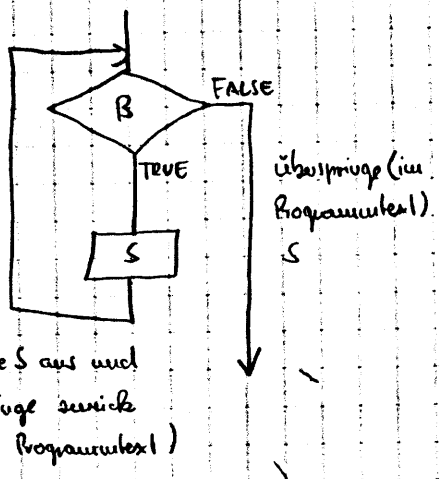
CASE E OF
k_1 : S_1 |
k_i : S_i |
k_n : S_n
ELSE : S_{n+1}
END



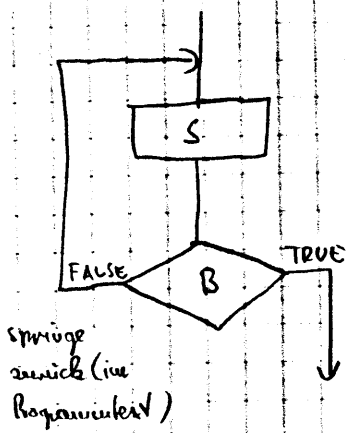
überspringe (im Programmtext) $S_1, \ldots, S_{i-1}$,
führe S_i aus und
überspringe (im Programmtext) $S_{i+1}, \ldots, S_{n+1}$

(2)

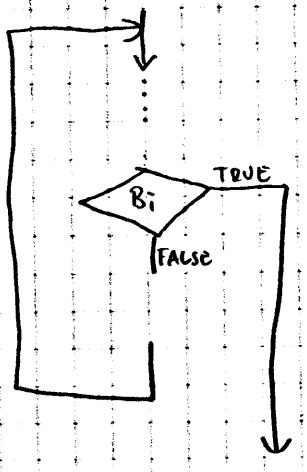
```
WHILE B DO
    S
END
```




```
REPEAT
    S
UNTIL B
```




```
LOOP
    IF Bi THEN EXIT END;
END
```



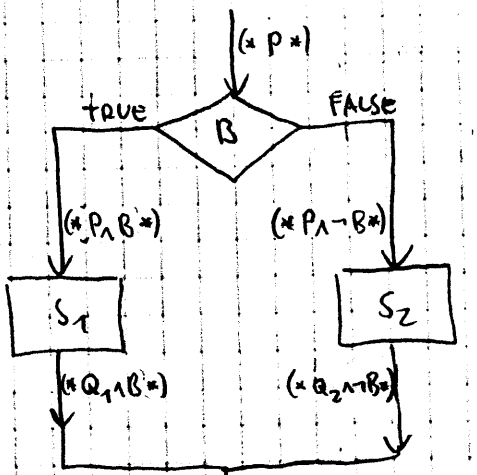
Mit diesen Sprungtypen (und den später behandelten Prozeduraufrufen) kommt man aus; sie sind übersichtlich und den meisten Aufgabenstellungen angepaßt.

eine allgemeine Überlegung: [merken der Vorgeschichte]

Die Ausführung eines Programmes erfolgt schrittweise, nämlich Anweisung für Anweisung. Die Maschine muß sich die wesentlichen Ergebnisse der vorherigen Schritte, d.h. der vorherigen Vorgeschichte der Rechnung, merken. Dies geschieht in zwei Formen:

- durch die Werte der Designatoren,
- durch die Stelle des Programmtextes, die gerade bearbeitet wird.

Als Beispiel betrachte folgende Situation (in der wir zur Vereinfachung annehmen, daß S_1 und S_2 den Wert von B unverändert lassen, d.h. $(*B*)\ S_i\ (*B*)$ und $(*\neg B*)\ S_i\ (*\neg B*)$):



die Stellen der Zweige enthalten als Ergebnis, die Vorgeschichte den Wert von B (zur Zeit der Fallunterscheidung)

(* $Q_1 \wedge B \vee Q_2 \wedge \neg B$ *) ← an dieser Stelle ist die Information über den Wert von B (zur Zeit der Fallunterscheidung) nicht mehr in der Stelle enthalten; es herrscht zunächst eine gewisse Unsicherheit über das erreichte Ergebnis, ausgedrückt durch die Disjunktion.

Falls $Q_1 \Rightarrow Q$ und $Q_2 \Rightarrow Q$, wobei Q das gewünschte Ergebnis sei, so ist die Unsicherheit nur formal, denn dann gilt:

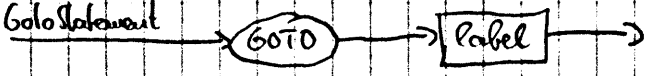
$$(Q_1 \wedge B) \vee (Q_2 \wedge \neg B) \Rightarrow (Q \wedge B) \vee (Q \wedge \neg B)$$
$$\Rightarrow Q \wedge (B \vee \neg B) \Rightarrow Q$$

Auf der Ebene der problemorientierten Programmierung erfolgt das Merken der Vorgeschichte über die Stelle des Programmtextes nur in

- triviale Form: bei applikativer Programmierung (ausdrucksorientiert) über den Aufbau des Ausdrucks
- in übersichtlicher Form: bei prozeduraler Programmierung (anweisungsorientiert) mit Hilfe strukturierter Anweisungen, §12.

Auf der Ebene der maschinenorientierten Programmierung stehen die Bildung von (nichttrivialen) Ausdrücken bzw. strukturierte Anweisungen i.allg. nicht direkt zur Verfügung; daher müssen solche Strukturen in Folgen elementarer Anweisungen übersetzt werden.

Dafür benötigt man insbesondere die Sprunganweisung (, die in MODULA-2 weder vorhanden noch wünschenswert ist).



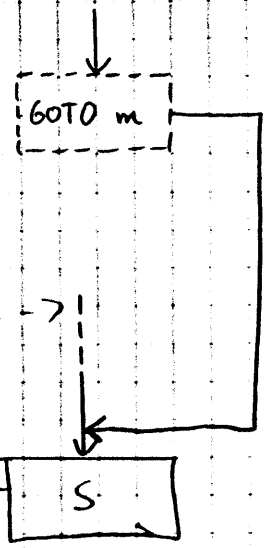
Die Marke (Label), d.h. das Sprungziel, muß genau einmal vorkommen als Kennzeichnung einer Anweisung, z.B.



umgangssprachliche Semantik GOTO m

"springe zur mit der Marke m gekennzeichneten Anweisung"

Flußdiagramm:



Anweisungen, die hier stehen, sind nur durch andere GOTO's erreichbar

Beachte: alle Pfeile im Flußdiagramm bedeuten "Sprünge"!

Es ist unentscheidbar, ob in einem Programm P bei Aufruppwand 3 eine Anweisung S bei der Ausführung tatsächlich erreicht wird (äquivalent zum Halteproblem).

Die Übersetzung von MODULA-2-Programmen in die Maschinensprache unseres Rechners leistet u.a. der MODULA-2-Compiler.

Allgemein muß auch ein Programmierer häufig eine derartige Arbeit leisten:

"Problem" zunächst "strukturiert" beschreiben;

"Maschine" benennt die "Strukturen" nicht,

d.h. "Übersetzung" oder "Beschreibungsüberführung" ist notwendig.

Als Beispiel für eine derartige Übersetzung wollen wir zeigen, wie

strukturierte Anweisungen in MODULA-2

übersetzt werden können in

Anweisungsfolgen einer einfacheren, maschinennäheren Sprache, SMAL,

die im wesentlichen unserem Flußdiagramm entspricht. Damit werden wir gleichzeitig, wie in § 11 angekündigt, die Semantik der strukturierten Anweisungen durch Übersetzung beschreiben.

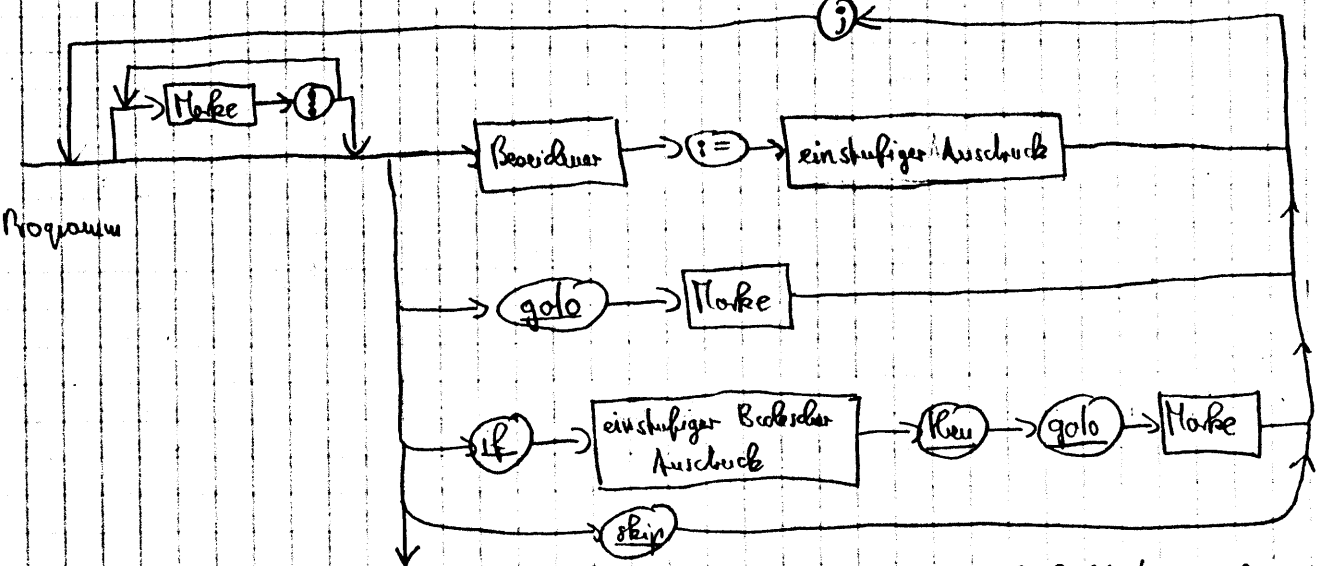
SMAL:

Anweisungen können auch mehrfach markiert sein [das machen wir nun, um die Übersetzung einfach zu beschreiben],

nur einstufige Ausdrücke, d.h. jeweils nur höchstens ein Funktionszeichen,

als Anweisungen nur

Verzweigung
Sprunganweisung
bedingte Sprunganweisung



SMAL-Programm

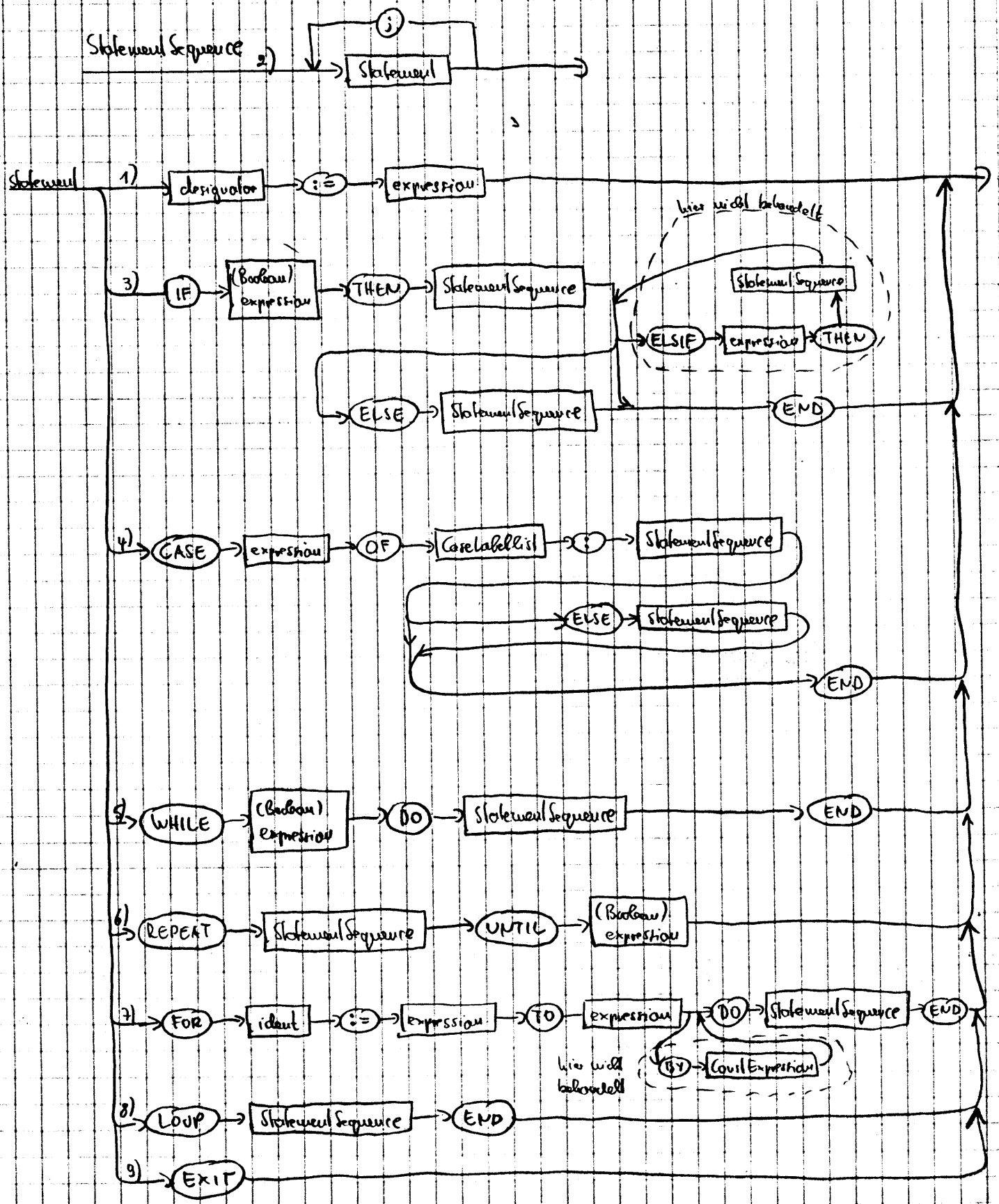
Marken eindeutig, Vereinbarung und Typen usw. hier der Einfachheit halber weggelassen.

Wir beschreiben eine Übersetzungsfunktion

rep : { Folgen (strukturierter) MODULA-2-Anweisungen } → { SMAL-Programme }

SMAL-Programme : siehe oben

MODULA-2-Anweisungsfolgen sind rekursiv definiert



Die Übersetzungsfunktion rep werden wir nur skizzieren, insbesondere soll ohne ausschließliche Notierung (zur Vereinfachung der Schreibweisen) ein

Umbenenner für Marken und Designatoren benutzt sein.

Rekursionsanfang:

nur zwei Zweige des Syntaxdiagramms sind rekursionsfrei, nämlich für die Wertzuweisung und für den EXIT-Sprung.

Den EXIT-Sprung behandeln wir später!

1) für einen einfachigen Ausdruck e sei

$$rep(x := e) \equiv x := e$$

Rekursion

1*) für einen mehrfachigen Ausdruck e der Form e_1 op e_2 (bzw. analog für op e_1) sei

$$rep(x := e) \equiv \begin{array}{l} rep(t_1 := e_1); \\ rep(t_2 := e_2); \\ x := t_1 \text{ op } t_2 \end{array} \quad \text{bzw.} \quad \begin{array}{l} rep(t_1 := e_1); \\ x := \text{op } t_1 \end{array}$$

wobei – t_1, t_2 neue Bezeichner,

– in $rep(t_1 := e_1)$ und $rep(t_2 := e_2)$ früher eingeführte neue Marken und Bezeichner durch Umbenennung verschieden gewählt werden!

2) $\text{rep}(S_1; \ldots S_n) \equiv \text{rep}(S_1);$
$\vdots$
$\text{rep}(S_n)$

(mit geeigneten Umbenennungen)

3) $\text{rep}(\text{IF } B \text{ THEN } S_1 \text{ ELSE } S_2 \text{ END})$

$\equiv \text{rep}(t := B);$
$\underline{\text{if}}\ t\ \underline{\text{then}}\ \underline{\text{goto}}\ m_1;$
$\text{rep}(S_2);$
$\underline{\text{goto}}\ m_2;$
$m_1 : \text{rep}(S_1);$
$m_2 : \underline{\text{skip}}$

(mit m_1, m_2 neue Marken, geeignete Umbenennungen usw.)

3*) $\text{rep}(\text{IF } B \text{ THEN } S \text{ END}) \equiv \text{rep}(t := \text{NOT}(B));$
$\underline{\text{if}}\ t\ \underline{\text{then}}\ \underline{\text{goto}}\ m;$
$\text{rep}(S);$
$m : \underline{\text{skip}}$

4) $\text{rep}(\text{CASE } e \text{ OF } k_1 : S_1 \mid \ldots \mid k_n : S_n \text{ ELSE } S_{n+1} \text{ END})$

$\equiv \text{rep}(t := e);$
$\underline{\text{if}}\ t = k_1\ \underline{\text{then}}\ \underline{\text{goto}}\ m_1;$
$\vdots$
$\underline{\text{if}}\ t = k_n\ \underline{\text{then}}\ \underline{\text{goto}}\ m_n;$
$\text{rep}(S_{n+1});$
$\underline{\text{goto}}\ m;$
$m_1 : \text{rep}(S_1);\ \underline{\text{goto}}\ m;$
$\vdots$
$m_n : \text{rep}(S_n);\ \underline{\text{goto}}\ m;$
$m : \text{skip}$

(entspricht nicht den tatsächlich üblichen Übersetzungen)

5) rep(WHILE B DO S END) ≡ m_1 : rep(t := NOT(B));
if t then goto m_2;
rep(S);
goto m_1;
m_2 : skip

6) rep(REPEAT S UNTIL B) ≡ m : rep(S);
rep(t := NOT(B));
if t then goto m

7) rep(FOR i := e_1 TO e_2 DO S END)
≡ rep(t := e_2);
rep(i := e_1);
if i > t then goto m_2;
m_1 : rep(S);
if i ≥ t then goto m_2;
i := INC(i);
goto m_1;
m_2 : skip

8) rep(LOOP S END) ≡ m_1 : skip;
rep(S);
goto m_1;
m_2 : skip

9) Die Übersetzung des EXIT-Sprungs ist Kontext-abhängig : sei m_2 die Schlußmarke der Übersetzung der engsten LOOP ... END - Anweisung, in der der EXIT-Sprung vorkommt, dann sei

rep(EXIT) ≡ goto m_2

Beispiel:

. rep (i := 0 ; s := 0 ; WHILE i < 100 DO s := s + i ; i := i + 1 END)

≡ rep (i := 0) ;
rep (s := 0) ;
rep (WHILE i < 100 DO s := s + i ; i := i + 1 END)

≡ i := 0 ;
s := 0 ;
m_1 : rep (t := NOT (i < 100)) ;
if t then goto m_2 ;
rep (s := s + i ; i := i + 1) ;
goto m_1
m_2 : skip

≡ i := 0 ;
s := 0 ;
m_1 : t1 := i < 100 ;
t := NOT t1 ;
if t then goto m_2 ;
s := s + i ;
i := i + 1 ;
goto m_1 ;
m_2 : skip

14. Skalare Datentypen II

14.1 Der Standardtyp Zeichen

bisher behandelte Standardtypen:

Wahrheitswerte:	BOOLEAN	insbesondere für Steuerung des Programmablaufs
ganze Zahl:	INTEGER	} für numerische Rechnungen
natürliche Zahl:	CARDINAL	
reelle Zahl:	REAL	

Programmierung befaßt sich jedoch nicht nur mit numerischen Rechnungen, sondern im weitesten Sinne mit Zeichenverarbeitung, d.h. mit der Verarbeitung von Zeichenfolgen. Jede programmierbare Aufgabe erscheint auf gewissen Ebenen als eine Aufgabe der Zeichenverarbeitung. Auf einer der untersten Ebenen, der logischen Ebene der Geräte, werden nur Zeichenfolgen über dem Alphabet {0, 1} verarbeitet. Die Werte dieses Alphabets, nämlich 0 und 1 werden dabei – je nach Zusammenhang – als Darstellungen der Wahrheitswerte (etwa 0 ≙ FALSE, 1 ≙ TRUE) oder als Ziffern von Dualzahlen oder als Ziffern von binären Verschlüsselungen von Ausweisen oder auch als noch andere Größen gedeutet.

In dieser Vorlesung haben wir schon einige dieser Zeichenverarbeitungs-Aufgaben kennengelernt, z.B.

einige Beispiele in §2,
Ziffernalkül der Addition in §3,
Syntaxbeschreibung durch Syntaxdiagramme, Erkennung von syntaktisch richtigen Programmen und ihre Übersetzung in maschinennähere Programme durch einen Compiler in §8 bzw. §13.

Die zuvor beschriebenen Funktionen lassen sich dabei stets als Wertfunktionen deuten;

z.B. Addition, $add : \{0,1\}^* \times \{0,1\}^* \longrightarrow \{0,1\}^*$

korrekte Syntax, $syntax : X^*_{\text{MODULA-2}} \longrightarrow \{\underline{\text{korrekt}}, \underline{\text{fehlerhaft}}\} \times \underbrace{\{a,\dots,z,\dots\}^*}_{\text{„Fehlermeldungen“}}$

Übersetzung, $rep : X^*_{\text{MODULA-2}} \xrightarrow{p} X^*_{\text{SMAL}}$

Weil die Darstellung von Zeichen nicht auf allen handelsüblichen Geräten gleich ist, läßt die Sprachbeschreibung hier weiter für die Implementierung der Sprache offen. Sehr gebräuchlich ist der ASCII-Zeichensatz (American Standard Code for Information Interchange), der neben den druckbaren Schriftzeichen (Buchstaben, Ziffern, Sonderzeichen) noch (nicht druckbare) Steuerzeichen (z.B. für Zeilenvorschub beim Drucken) enthält

Wertemenge: ASCII-Zeichensatz gemäß der folgenden Tabelle, die jedem Zeichen eine Zahl zwischen 0 und $(127)_{10} = (177)_8 = 2^7 - 1$ in der Oktaldarstellung (d.h. zur Basis 8) zuordnet.

$1 \cdot 64 + 7 \cdot 8 + 7 \cdot 1 = 127$

Die Anordnung dieser Wertemenge ist entsprechend dieser Zuordnung, d.h. die Funktionen

INC, DEC, ORD

arbeiten mit dieser Zuordnung.

z.B.: $40_8 + 13_8$
$= 32_{10} + 11_{10} = 43_{10}$

decimal		0	16	32	48	64	80	96	112
	octal →	0	20	40	60	100	120	140	160
	↓								
0	0	nul	dle		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	10	bs	can	(	8	H	X	h	x
9	11	ht	em	)	9	I	Y	i	y
	12	lf	sub	*	:	J	Z	j	z
	13	vt	esc	+	;	K	[	k	{
	14	ff	fs	,	<	L	\	l	|
	15	cr	gs	-	=	M	]	m	}
	16	so	rs	.	>	N	↑	n	~
15	17	si	us	/	?	O	←	o	del

Layout characters

bs	backspace
ht	horizontal tabulator
lf	line feed
vt	vertical tabulator
ff	form feed
cr	carriage return

Separator characters

fs	file separator
gs	group separator
rs	record separator
us	unit separator

Der Standardbezeichner für diesen Typ ist: CHAR

Standardbezeichner für Zeichen (in MODULA-2-Programmen)

in ' ' oder " " eingeschlossenes Zeichen

Bsp 'a'
'?'
' ' (Blank-Zeichen, häufig als '␣' notiert)
das Zeichen ' wird durch " ' " bezeichnet,
das Zeichen " wird durch ' " ' bezeichnet

Operationen:

- Standardoperationen für skalare Datentypen:
- Transfer-Funktionen:

ORD: {Zeichen} → {0, ..., 127}, Standardoperation

CHR: {0, ..., 127} → {Zeichen}

mit CHR (ORD (c)) = c und ORD (CHR (i)) = i

z.B. ORD ("a") = $(141)_8$ = $(97)_{16}$

ORD ("?") = $(77)_8$ = $(63)_{10}$

ORD (" ") = $(40)_8$ = $(32)_{10}$

Bsp Folgendes Programm liest eine Zeichenfolge, bis zum ersten Mal ein Zeichen ungleich einer Ziffer erscheint, und ermittelt den Wert der Ziffernfolge in der Dezimaldarstellung.

```
MODULE zahl;
FROM InOut IMPORT Read, WriteCard, WriteLn;

VAR   ziffer : CHAR    (* Zeichen der Argument-Zeichenfolge *);
      wert : CARDINAL (* Ergebnis *);

CONST ordvon0 = 0F0H  (* Ordnung der Ziffer 0 in hexadezimaler Darstellung *);
                ↑  ↑ zeigt hexadezimale Darstellung an
                erstes Zeichen muß eine Ziffer sein
BEGIN
   wert := 0;
   Read (ziffer);
   WHILE ("0" <= ziffer) AND (ziffer <= "9")
      DO   wert := 10 * wert + (ORD(ziffer) - ordvon0);
           Read (ziffer)
      END;
   WriteCard (wert, 10); WriteLn
END zahl.
```

manchmal wird auch verwendet:

EBCDIC - Zeichensatz mit $256 = 2^8$ Zeichen

Extended Binary Coded Decimal Interchange Code

aus E. Horowitz: Programming Languages

Diese Tabelle ordnet jedem Zeichen eine Zahl zwischen 0 und $(255)_{10} = (FF)_{16} = 2^8 - 1$ in der Hexadecimaldarstellung (d.h. zur Basis 16) zu

Bits

0-3	4-7

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3																
4		A	B	C	D	E	F	G	H	I	¢	,	<	(	+	⌶
5	&	J	K	L	M	N	O	P	Q	R	!	$	*	)	;	¬
6	—	/	S	T	U	V	W	X	Y	Z	:	,	%	—	>	?
7	&	∧	¨	Δ		[{]			v	`	:	≢	@	'	=	"
8	~	a	b	c	d	e	f	g	h	i	↑	↓	≤	⌈	⌊	→
9	[]	j	k	l	m	n	o	p	q	r	⊃	⊂		⊙	·	←
A	—	~	s	t	u	v	w	x	y	z	∩	∪	⊥	[	≥	▲
B	α	∈	ι	p	w		x	\	÷		Δ	∇	T	]	≠	|
C	<	A	B	C	D	E	F	G	H	I	⍱	⍲	⍳	⌽	Y	⍉
D	>	J	K	L	M	N	O	P	Q	R	⌶	!	⍒	⍋	[]	A
E	\		S	T	U	V	W	X	Y	Z	⌿	⍀	⌹	∘	{+}	⍕
F	0	1	2	3	4	5	6	7	8	9	|	⍱		∘	⍎	

For example, the EBCDIC character 0 has a hex value FO.

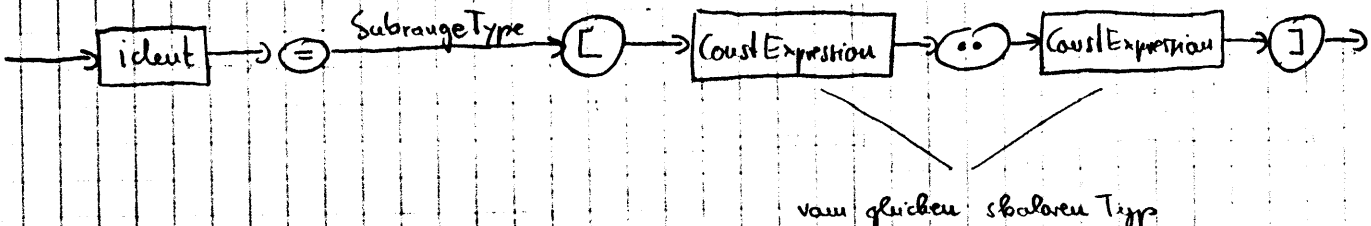
Table 3-2: APL Character Set in EBCDIC
source:IBM 5110 APL Reference Manual

14.2 Skalare Unterbypen

vielleicht besserer Name: skalare "Ausschnitte" oder skalare "Intervalle"

Für skalare Datenbypen, außer reelle Zahlen, kann man Unterbypen vereinbaren, die jeweils einen vollständigen Ausschnitt des Typs umfassen. Solch ein Ausschnitt wird durch das kleinste und das größte Element des Ausschnitts definiert.

In der TYPE-Vereinbarung werden Typbezeichner für Unterbypen wie folgt vereinbart



Bsp

```
TYPE  wochentag = ( mon, die, mit, don, fre, sam, son );
      arbeitstag = [ mon .. fre ];
      ziffer = [ "0" .. "9" ];
      kleinbuchst = [ "a" .. "z" ];
```

Alle Operationen für Unterbypen sind i.allg. die Operationen des zugehörigen skalaren Datenbypes zugelassen, wobei man natürlich sicherstellen muß, daß die Operationen nicht aus dem Unterbyp herausführen.

Unterbypen dienen u.a.

- der besseren Lesbarkeit von Programmen
- (bei entsprechender Implementierung) der Sicherheit von Programmen

15. Einfache Ein-/Ausgabe

siehe Probleme!

16. Ein Vorgriff: Felder

Um mehr Übungsmöglichkeiten und interessantere Beispiele zu haben, soll hier ein Vorgriff auf §24 erfolgen.

Man betrachte folgende Aufgabe:

Auswertung eines Polynoms $a_0 + a_1 x + \ldots + a_{10} x^{10}$

d.h. wir sollen die Funktion

$$\text{polynom} : \mathbb{R}^{11} \times \mathbb{R} \to \mathbb{R}$$
$$\text{polynom}(a_0, \ldots, a_{10}, x) := \sum_{i=0}^{10} a_i x^i$$

beschreiben.

Zur Beschreibung dieser Funktion benötigen wir Bezeichner für die Koeffizienten $a_0, \ldots, a_{10}$. In der üblichen mathematischen Schreibweise benutzt man zweiteilige Bezeichner, aufgebaut aus einem "Namen", a, und einem "Index", i.

Dadurch kann man, in gewissem Sinne, den gesamten Bereich "ansprechen". In MODULA-2 kann man dies unmittelbar, im Beispiel durch die Vereinbarung

```
VAR a : ARRAY [0..10] OF REAL;
```

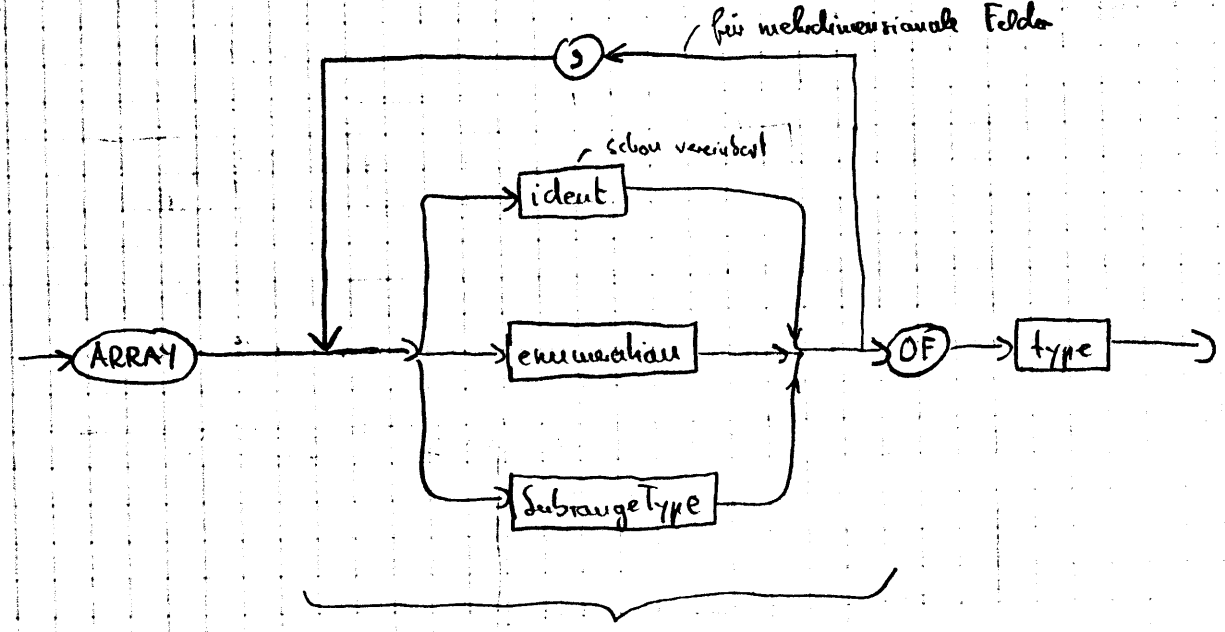
Dadurch wird ein Bezeichner a für ein Feld (array) von 11 Komponenten vereinbart, die durch die Elemente des Unterbereichstyps [0..10] ausgewählt werden können; jede Komponente a[i] kann einen Wert vom Typ REAL bezeichnen.

Sehr ausführlich gesprochen, kann man nun 11 real-Werte einheitlich durch Angabe von "Schoße" und "Hausnummer" ansprechen:

```
a[0]
a[1]
 :
a[10]
```

"Schoße" — "Hausnummer"

Für diesen Vorgriff genügt folgende Syntaxvorschrift zur Vereinbarung von Bezeichnern für Felder:



Die Feldelemente werden dann durch Angabe von Ausdrücken für die Indizes angesprochen, z.B.

```
a[0]
a[i]
a[i-j]
```

Beispiel: Auswertung eines Polynoms $a_0 + a_1 x + \ldots + a_{10} x^{10}$

```
MODULE auswerten;
FROM InOut IMPORT WriteLn;
FROM RealInOut IMPORT ReadReal, WriteReal;

CONST   n = 10;
TYPE    index = [0..n];

VAR     a : ARRAY index OF REAL;    (* Argumente *)
        x : REAL;                   (* Argument *)
        y : REAL;                   (* Ergebnis *)
        i : index;

BEGIN
    y := 0;
    FOR i := 0 TO n DO ReadReal (a[i]) END;
    ReadReal (x);
    FOR i := n TO 0 BY -1 DO y := a[i] + x * y END;
    WriteReal (y, 10); WriteLn
END auswerten.
```

Bem: Für die Folge der Zwischenergebnisse kann die Bezeichnung y stets wiederverwendet werden!

nach Horner-Schema:

$$a_0 + a_1 x + a_2 x^2 + \ldots + a_{10} x^{10}$$

$$= a_0 + x\,(a_1 + x\,(a_2 + \ldots + x\,(a_9 + x \cdot a_{10}) \ldots))$$

y_{10}, y_9, y_1, y_0

Bsp. Matrix-Multiplikation

$$\begin{pmatrix} a_{1,1} & \cdots & a_{1,10} \\ \vdots & & \\ a_{10,1} & \cdots & a_{10,10} \end{pmatrix} \cdot \begin{pmatrix} b_{1,1} & \cdots & b_{1,10} \\ & \vdots & \\ b_{10,1} & \cdots & b_{10,10} \end{pmatrix} = \begin{pmatrix} & c_{i,j} & \\ & & \end{pmatrix}_{\substack{1 \le i \le 10 \\ 1 \le j \le 10}} \quad \text{mit}$$

$$c_{i,j} = \sum_{k=1}^{10} a_{i,k} \cdot b_{k,j}$$

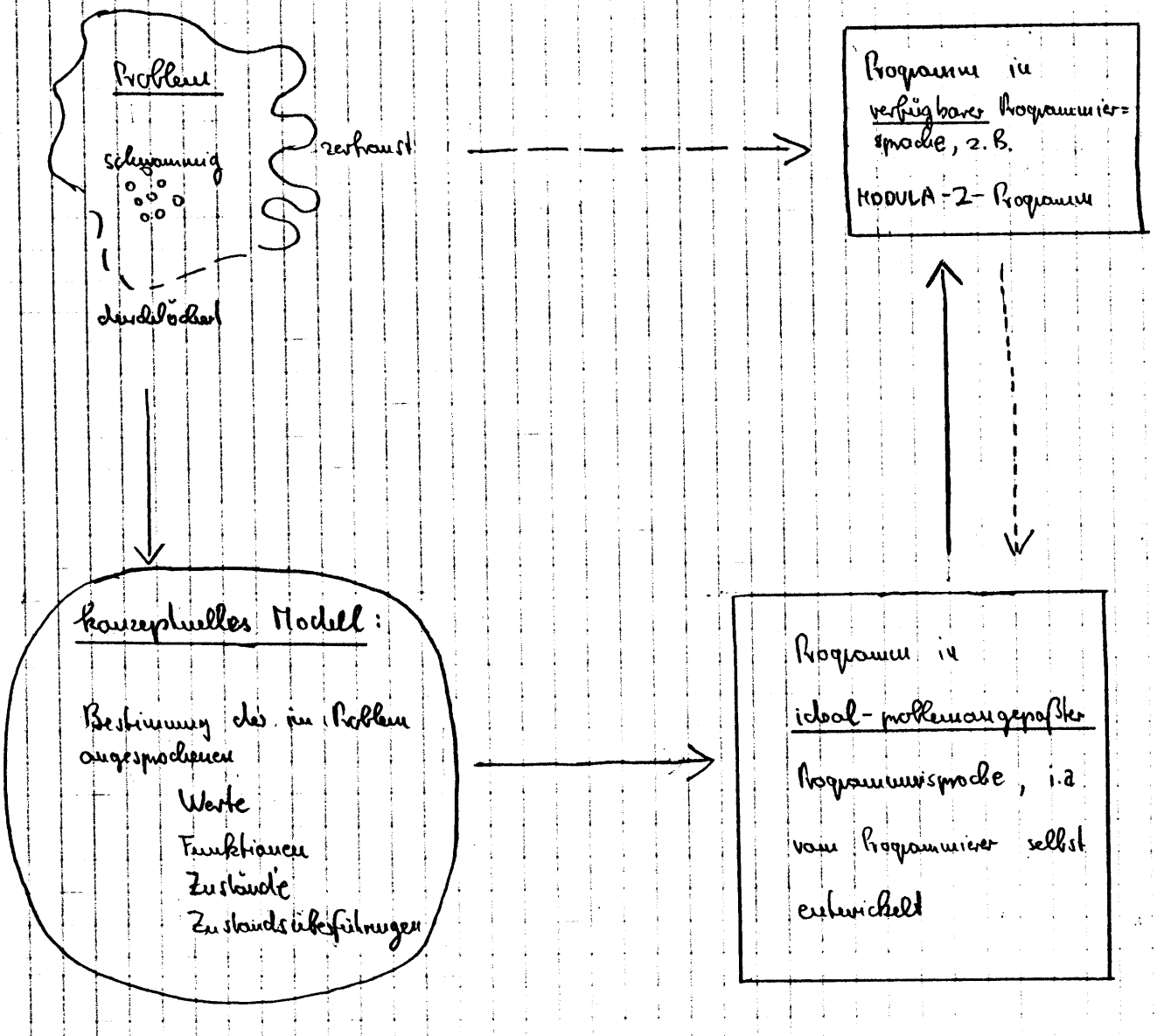
```
MODULE matrixmult;
  :
CONST n = 10;
TYPE index = [1.. n];
     matrix = ARRAY index, index OF REAL;

VAR a, b: matrix;  (* Argumente *)
      c: matrix;  (* Ergebnis *)

    i, j, k: index;
       s: REAL;

BEGIN
  : (* Wertzuweisungen an alle a[i,j] und b[i,j] *)

FOR i:=1 TO n DO
  FOR j:=1 TO n DO
      s:=0;
      FOR k:=1 TO n DO s:= s+ a[i,k]*b[k,j] END;
      c[i,j] := s
  END
END
:
END matrixmult.
```

C. Systematisches Programmieren

17. Allgemeine Methoden

aus § 12.1, Programmieren:

Entwicklung eines Ablaufs bzw. eines Programms, die ein vorgegebenes Problem formal beschreiben und damit der maschinellen Bearbeitung zugänglich machen.



Ganz allgemein gesprochen:

- Das "vorgegebene" Problem muß zunächst hinreichend genau erfaßt und verdeutlicht werden, wobei durch Abstraktion ein Modell entsteht.

- Zwischen den Anforderungen des Modells und den Ausdrucksmöglichkeiten der verfügbaren Programmiersprache wird im allgemeinen eine große Lücke klaffen: die verfügbaren Worte, Begriffe, Anweisungen usw. sind zu einfach, um eine direkte Problemlösung zu erlauben. (Bei einer universellen Sprache, § 5, läßt sich das Problem, wenn überhaupt, stets auf irgendeine Weise beschreiben; aber hier geht es darum, systematisch eine gute Beschreibung zu finden!)

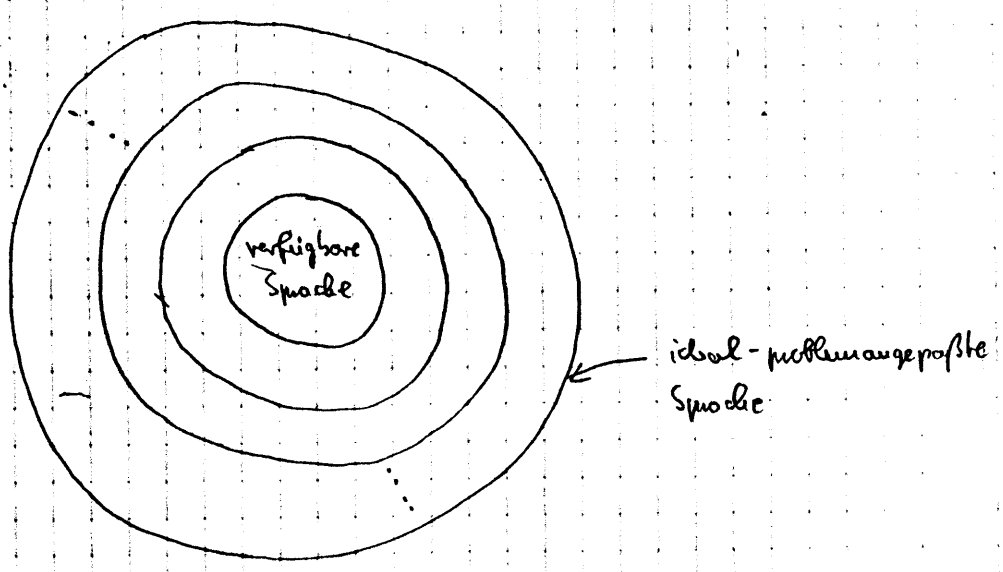
- Die Lücke muß nun schrittweise überbrückt werden: Dies kann im Grundsatz in zwei Richtungen erfolgen:

  von oben nach unten (top-down):

  Die "höheren" Sprachelemente der vom Modell verlangten ideal-problemangepaßten Programmiersprache werden auf "niedrigere" Sprachelemente, schließlich der verfügbaren Programmiersprache, zurückgeführt.

  von unten nach oben (bottom-up)

  Aus den ("niedrigeren") Sprachelementen der verfügbaren Programmiersprache werden "höhere" Sprachelemente, schließlich der ideal-problemangepaßten Programmiersprache, aufgebaut.



Die Zwischenebenen werden schrittweise erstellt, indem man sich geeignete neue

Werkzeuge
Funktionen
Anweisungen

definiert (vgl. § 2), die dann jeweils die tieferen Ebenen abschirmen.

Allgemein kann man dieses Vorgehen kennzeichnen als

Problemlösen durch Zerlegung:

Ein (schwieriges, vielfältige Gesichtspunkte ansprechendes) Problem wird zerlegt in (leichtere) Teilprobleme, mit denen man das Vorgehen wiederholt, bis eine Lösung unmittelbar möglich ist (top-down);

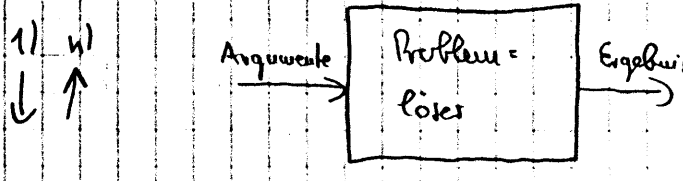
aus unmittelbar lösbaren Teilaufgaben werden Lösungen für (schwierigere) Oberaufgaben aufgebaut, mit denen man das Vorgehen wiederholt, bis eine Lösung des Gesamtproblems möglich ist (bottom-up).

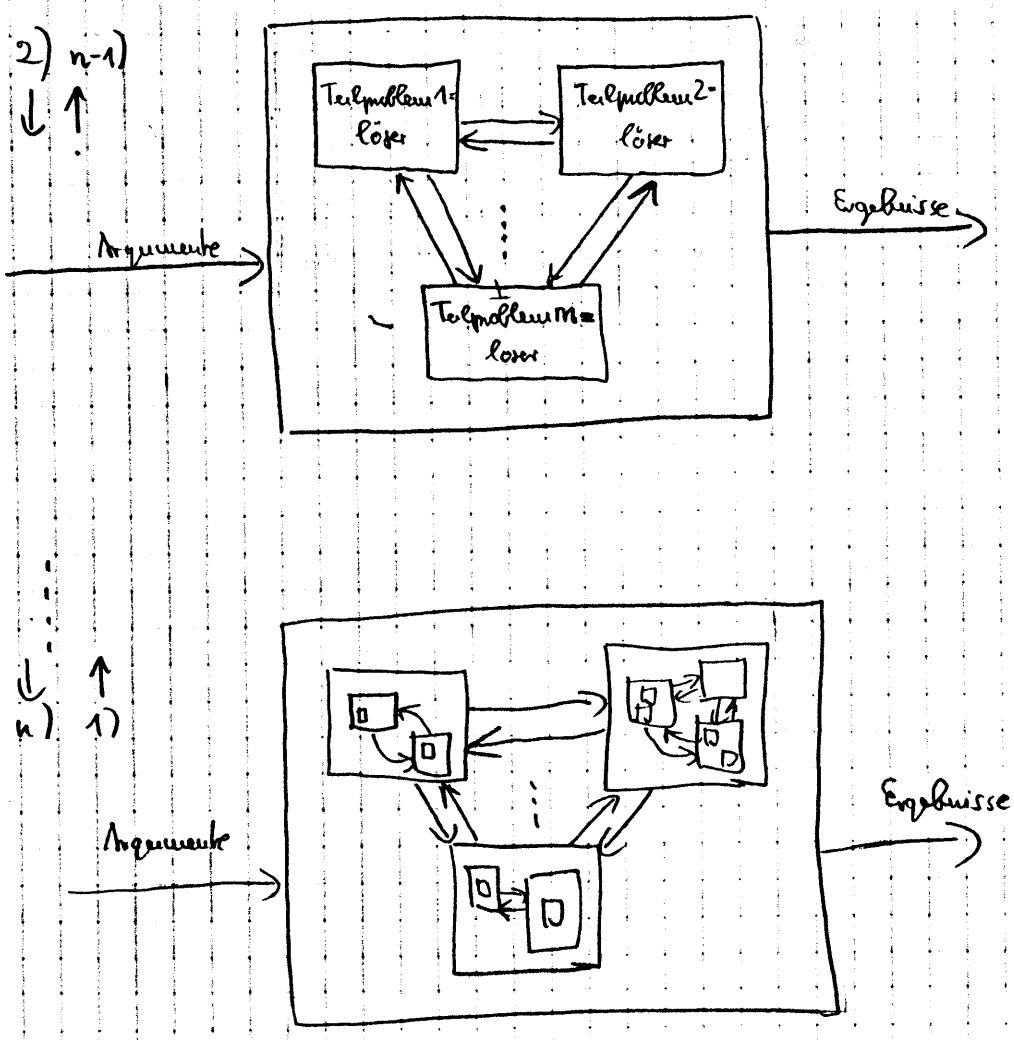
Über die bessere Richtung des Vorgehens kann man trefflich streiten, aber kaum allgemeingültige Aussagen machen:

- ein unartiges Problem wird man im ersten Anlauf wohl von oben nach unten zu zerlegen versuchen;
- die dabei erworbenen Erfahrungen, insbesondere die häufig auch auf höheren Ebenen immer noch bestehende Schwierigkeit, erwünschte Sprachelemente durch tatsächlich verfügbare darzustellen, können dann zu einer Überarbeitung der Problemlösung von unten nach oben führen;
- wie auch immer man vorgeht, gewisse Wiederholungsschritte während der Entwicklung werden i.allg. notwendig sein.

Der Entwicklung des konzeptuellen Modells wird im allgemeinen auch im obigen Sinne schrittweise erfolgen und Wiederholungsschritte erfordern.

Graphisch kann man sich das schrittweise Vorgehen wie folgt veranschaulichen:





Wichtig ist hier die Art und Weise, wie ein einzelner Baustein beschrieben wird, nämlich durch folgende Angaben (Schnittstellen-Beschreibung):

- Werte für "Argumente"
- Werte für "Ergebnisse"
- beschreibene Funktion
- beschreibene Zustandsveränderungen

} wobei von außen gesehen der Baustein diese und nur diese Wirkung haben darf

ferner

- über welche anderen Bausteine er verfügen darf
- welche seiner Teile von anderen Bausteinen benutzt werden dürfen

Dieses ist genau die in §4 schon angesprochene "Kunst des schwarzen Kastens", die

- genau angibt, was der Baustein von außen gesehen bewirkt und wie seine Bezeichnungen / an anderen Bausteinen aussehen;
- offen läßt, wie er im Inneren arbeitet; das bedeutet insbesondere, daß man das Innere getrost verändern (verbessern) darf, solange man seine Wirkung nach außen nicht verändert, d.h. man kann Änderungen vornehmen, die sich im Gesamtgefüge nur scharf begrenzt bemerkbar machen können.

Wenn man solche "schwarzen Kästen" baut, erreicht man noch zwei weitere wichtige Eigenschaften:

Sicherheit: Da das Innere abgeschirmt ist, d.h. der außen stehende "Benutzer" kann das Innere nicht kann es somit auch nicht (direkt) beeinflussen, kann man nicht "aus Versehen" (oder wenn man böse Absicht unterstellt: "aus Heimtücke") von außen den Fehler in das Innere einbauen.

Lokalität von Bausteinen

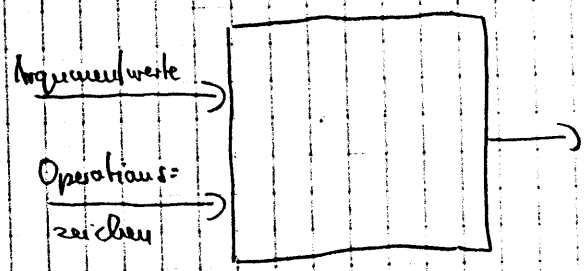
Benutzen:
Ein Baustein muß mit zwei Arten von anderen Bausteinen bzw. Werten V
- von außen zur Verfügung gestellte
- lokal vereinbarte.

Die Bezeichnung der lokal benutzten Bausteine bzw. Werte kann unabhängig von den Bezeichnungen lokal benutzter Bausteine bzw. Werte anderer Bausteine erfolgen. Damit wird die Beschreibung erheblich vereinfacht und übersichtlicher (vgl. §2.4).

Bei der Einführung von (skalaren) Datentypen haben wir verabredet

Datentyp – Wertemenge und Operationen.

Auch Datentypen kann man sich also als "schwarze Kästen" vorstellen:



Das Innere führt die gewünschte Operation aus, ohne daß man von außen die Art und Weise feststellen kann.

Diese Sichtweise ist aber auch genau die in §2.9 unter dem Stichwort "Maschine" angegebene.

Anschaulich sehr verschiedene Dinge erweisen also unter den hier besprochenen Gesichtspunkten als gleichartig; ein gemeinsamer Name für diese Dinge ist Rechenstrukturen:

Rechenstrukturen – Einheit von Wertemenge(n) und zugehörigen Operationen unter Abschirmung der Art und Weise, wie im Inneren Werte dargestellt und Operationen durchgeführt werden.

Bisher wurden die Ideale eines systematischen Programmierens durch

Zerlegung und Abschirmung

behandelt.

Im folgenden müssen wir sehen, wie man diese Ideale mit dem vorhandenen Werkzeug, d.h. hier der Programmiersprache MODULA-2 verwirklichen kann. Manchmal wird MODULA-2 nicht alles eigentlich Wünschenswerte leisten.

Aber das ist kein Grund, die oben entwickelten allgemeinen Methoden nicht zu beachten, denn

- MODULA-2 stellt für unser Team hier nur die unterste Ebene dar, d.h. auf höheren Ebenen sind wir noch bedeutend freier,
- auf unserer untersten Ebene können wir uns den Idealen immerhin weit annähern und durch entsprechende Disziplin (d.h. hier auch Beschränkung) und Kommentierung das Wünschenswerte noch sichtbar machen.
- neuere und noch zu entwickelnde Programmiersprachen unterstützen unsere Ideale vielleicht besser.

Die grundlegende Bedeutung und Notwendigkeit des systematischen Programmierens ergibt sich erst in voller Schärfe bei der Entwicklung wirklich großer Programme (Programmsysteme), die im Laufe mehrerer Jahre von einer Gruppe erstellt werden. Wegen der langen Dauer und der Beteiligung mehrerer Personen müssen - neben den (mehr oder weniger) Informatik-typischen Methoden des systematischen Programmierens - auch

der allgemeinen Methoden der Projektführung beachtet werden.
Insbesondere im gewerblichen Bereich müssen vorrangig auch die wirtschaftlichen Randbedingungen (Kosten, Termine, usw.) berücksichtigt werden.

Unter dem Stichwort "Programmentwicklung", oder häufiger schon unter Einschluß bestimmter Ausdrucksmittel "software engineering", versucht man heute eine durchgängige Methodik zu entwerfen, die dann auch durch geeignete "Werkzeuge" (Dokumentationshilfen, Versionenverwaltung, Programmbibliotheken) unterstützt werden soll.
Dies ist ein weites Gebiet und wird in weiterführenden Vorlesungen behandelt. In § 2.11 haben wir schon ganz grob die Entwicklungsphasen angesprochen.

Wir wollen hier nur noch eine Liste von besonders zu beachtenden Qualitätsmerkmalen von Programmen auflisten:

korrekt: Programm beschreibt die gewünschten Funktionen bzw. Zustandsüberführungen. Sofern das "Gewünschte" schon formal spezifiziert wird, kann dies – zumindest im Grundsatz – bewiesen werden (vgl. § 11), allerdings sind die tatsächlichen Möglichkeiten hierzu zur Zeit noch recht begrenzt. Deshalb spielt das Testen eine große Rolle.

effizient: Programm verwendet möglichst wenig Betriebsmittel, d.h. insbesondere Zeit, Speicher, Übertragungswege.
Im allgemeinen kann man Wechselbeziehungen zwischen diesen Größen erwarten.

zuverlässig und robust: Programm arbeitet immer (auch unter nicht vorhergesehenen Umständen) korrekt oder zumindest durchschaubar; bei Fehlern,

insbesondere falsche Bedienung, bewahrt es seine Funktionsfähigkeit ("stürzt nicht ab").

wartbar und erweiterbar:

Während der Lebensdauer eines Programms gefundene Fehler bzw. neu hinzukommende Anforderungen können leicht behoben bzw. durch Ergänzungen erfüllt werden.

überhaupt:

Das Programm hängt möglichst wenig und dann vorhersehbar von besonderen Eigenschaften der benutzten Umgebung ab; alle bei Umgebungsveränderungen durchzuführenden Änderungen sind einfach und begrenzt.

wiederverwendbar:

Das Programm, oder Teile davon, ist auch für (abstrahierende) weitere Zwecke verwendbar.

Die bisher genannten Qualitätsmerkmale lieben die Wünsche des "Anwenders" des Programms hervor, die das Programm im wesentlichen "von außen" betrachten. In bezug auf das "Innere" des Programms fordert man das folgende Merkmal:

strukturiert und lesbar

Notwendige Voraussetzung zur Erfüllung aller anderen Merkmale. Hierfür ist insbesondere eine sorgfältige und durchgängige Dokumentation erforderlich.

17.11

Ein (notgedrungen) kleines Beispiel:

Problem: Ein Text soll verschlüsselt werden, und zwar (obwohl es dafür viele raffiniertere Verfahren gibt) auf folgende einfache Art:

1) jeweils zwei aufeinanderfolgende Buchstaben werden vertauscht;
2) zeilenweise ersetze man den an der i-ten Stelle der Zeile stehenden Buchstaben x, wobei x der k-te Buchstabe im Alphabet sei, durch den (k+i)-ten Buchstaben.

Lösung und Verdeutlichung, konzeptuelles Modell

Werte: Texte: Zeichenfolgen über dem Alphabet der Buchstaben, die in Zeilen eingeteilt sind

Funktion: schlüssel: Texte $\longrightarrow$ Texte × Meldungen

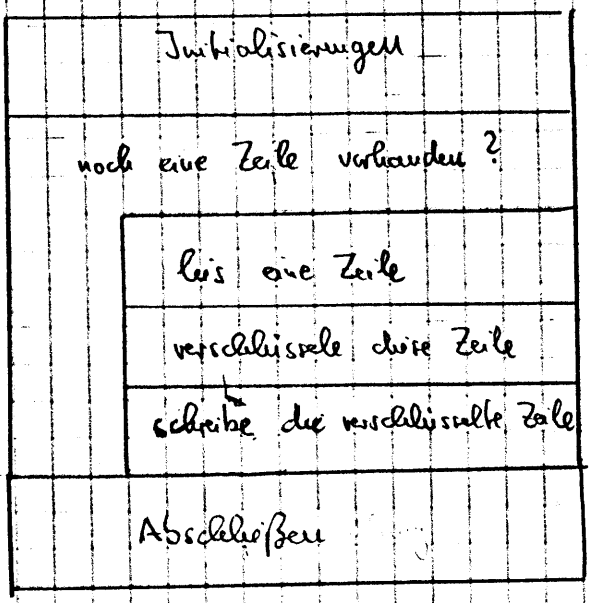
$$\text{schlüssel}(x) = \begin{cases} (\text{verschlüsseltes } x, \text{"erfolgreich"}) & \text{falls } x \text{ Text im obigen Sinne} \\ (\ ?\ , \text{Fehlermeldung}) & \text{falls } x \text{ kein Text, z.B. andere Zeichen als Buchstaben enthält} \end{cases}$$

genauere Festlegungen:

Fehlerbehandlung: Falls ein Zeichen entdeckt wird, das kein Buchstabe ist, so ersetze man dieses Zeichen durch den Buchstaben a und erzeuge eine Fehlermeldung.

17.12

erste Zerlegung:



Bem.: Dieses Diagramm enthält zwei typische Dreischritt-Programmfiguren:
initialisieren – arbeiten – abschließen
lesen – arbeiten – auswerten (schreiben)

"lies eine Zeile":

Argumente: zeilenlänge Zeichen — von Standard-Eingabe
(zeilennummer / Kardinalzahl) — von Initialisierungen bzw. "lies eine Zeile"

Ergebnisse: (Karte / Feld mit zeilenlänge Buchstaben) — an "verschlüssele diese Zeile"

(zeilennummer / Kardinalzahl) — an "lies eine Zeile"

ggf. Fehlermeldung mit Zeilennummer — an Standard-Ausgabe-File output

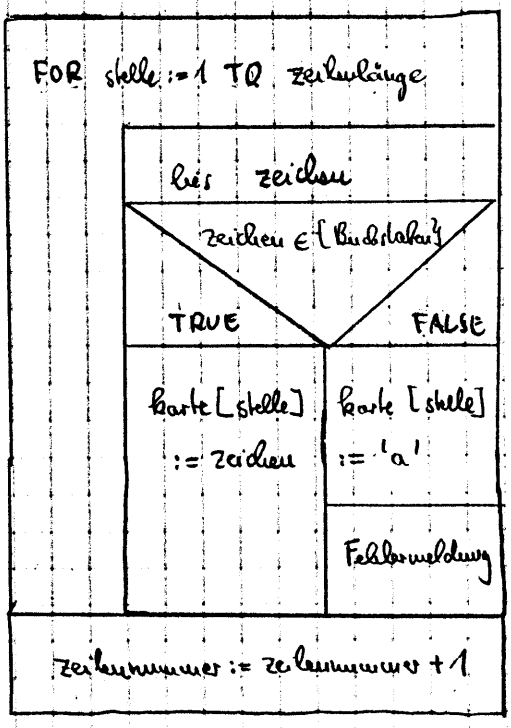
Zustandsüberführung:

(Karte | zeilennummer i) ⊢ (Karte eingelesenes Feld, bei Fehler 'a' eingesetzt | zeilennummer i+1)

benutzt: Lesen von Standard-Eingabe, Schreiben auf Standard-Ausgabe

Bem.: In dieser Darstellung liegt bereits eine wesentliche Entscheidung, nämlich mit Wiederholungen zu arbeiten, wobei Ergebnisse über Zustände weitergegeben werden. Diese beruht nicht allein auf der Logik des Problems, sondern hängt auch ab von der hier benutzten prozeduralen Ebene:

- Die Zeilennummer muß für Fehlermeldungen und ggf. Abschlußbehandlung ausdrücklich bezeichnet und damit über den gesamten Programmablauf zugänglich sein.
- Der durch Karte bezeichnete Wert könnte auch (applikativ gedacht) als Wert eines vom "verschlüsselte diese Zeile" benutzten Ausdrucks angesehen werden.



Bem.: Hier bleibt noch offen, wie zeilenlänge bestimmt wird:

- als von vornherein bekannte Konstante?
- durch Test auf ein Endzeichen?

"verschlüssele eine Zeile"

Argumente: (Karte, Feld und zeilenlänge Buchstaben) von "bis eine Zeile"

Ergebnis: (Karte, Feld und zeilenlänge Buchstaben) aus "schreibe die verschlüsselte Zeile"

Zustandsüberführung:

(Karte, Feld und zeilenlänge Buchstaben) ⊢ (Karte, verschlüsseltes Feld und zeilenlänge Buchstaben)

verschlüssele durch verkuscheln
verschlüssele durch "addiere die Stelle"

Teilaufgaben: "verschlüssele durch verkuscheln"

```
stelle := 2
stelle <= zeilenlänge
    (karte[stelle - 1], karte[stelle]) := (karte[stelle], karte[stelle - 1])
    stelle := stelle + 2
```

Teilproblem: "verschlüssele durch addieren der Stelle"

```
FOR stelle := 1 TO zahllänge
    karte [stelle] :=
        Buchstabenaddition
        (karte [stelle], stelle)
```

Teil-Teilproblem: "Buchstabenaddition"

Argumente: (buchstabe | zahl / Buchstabe | Kardinalzahl) mit [buchstabe / karte [stelle]] [zahl / stelle]

Ergebnis: (Buchstabe)

Funktion: buchstabenaddition (buchstabe, zahl) =

positionsinvers ((position (buchstabe) + zahl) mod, 26)

Teil-Teilteilproblem: position bzw. positionsinvers

Argumente: (Buchstabe) bzw. (0 ≤ Kardinalzahl ≤ 25)

Ergebnis: (0 ≤ Kardinalzahl ≤ 25) bzw (Buchstabe)

Funktion:

a	b	c	d	e	...	z
0	1	2	3	4		25

Hier wollen wir das Beispiel abbrechen, um zunächst MODULA-2-Sprachelemente kennenzulernen, die derartiges systematisches Programmieren durch "Zerlegen und Abstrahieren" unterstützen.

18. Prozeduren

Das Sprachmittel der Prozedur gibt uns die Möglichkeit, eine Reihe von Aufgaben zu behandeln, bzw. Wünsche zu erfüllen:

- §2: Wie definiert man sich neue Funktionsrufe?
  Wie definiert man sich neue (zusammengesetzte) Anweisungen?
- §17: Wie erstellt man (Teil-) Bausteine für (Teil-) Probleme?

- §2: Wie kann man Überschriften über die verwendeten Bezeichner behalten?
- §17: Wie kann man Bezeichner als lokal einführen und damit abschirmen?

- §17: Wie kann man Schnittstellen zwischen den Bausteinen beschreiben?

- Wie kann man ein Programmstück, das man an verschiedenen Stellen des gesamten Programms benötigt, nur einmal aufschreiben, aber mehrfach verwenden?

- Wie kann man ein Programmstück, das man an verschiedenen Stellen, aber jeweils für andere benannte Zustände benötigt, mehrfach verwenden?

Beispiel: "Baustein" zur Vertauschung zweier Zeichen (vgl. § 17 "verschlüssele durch vertauschen"):

a) Vereinbarung

Bezeichner für "neue Anweisung"

formale Parameter

Typ der Argumente und Ergebnisse

Prozedurkopf

```
PROCEDURE vertauschen (VAR x, y : CHAR);
```

Argument-Ergebnis-Parameter, durch VAR angezeigt, d.h. die Prozedur beschreibt Zustandsüberführungen der Form

$$\left(\begin{array}{c|c} x & y \end{array}\right) \vdash \left(\begin{array}{c|c} x & y \end{array}\right)$$

$$\left(* \left(\begin{array}{c|c} x & y \\ a & b \end{array}\right) \vdash \left(\begin{array}{c|c} x & y \\ b & a \end{array}\right) *\right)$$

Prozedurrumpf

```
VAR z : CHAR;
```

lokaler Bezeichner, nur sichtbar innerhalb der Prozedur

```
BEGIN   z := x;
        x := y;
        y := z
END vertauschen;
```

b) Aufrufe, z.B.

```
VAR eins, zwei : CHAR;
:
vertauschen (eins, zwei);
```

aktuelle Parameter (verschiedene Bezeichner vom gleichen Typ der formalen Parameter)

$$\left(* \left(\begin{array}{c|c} eins & zwei \\ a & b \end{array}\right) \vdash \left(\begin{array}{c|c} eins & zwei \\ b & a \end{array}\right) *\right)$$

man führe die Anweisungen des "geeignet modifizierten" Prozedurrumpfes aus:

- formale Argument-Ergebnis-Parameter durch aktuelle Parameter "ersetzt";
- lokale Bezeichner von allen anderen gültigen Bezeichnern "verschieden gemacht";
- dann ausführen.

z.B.

```
TYPE indexbereich = [1 .. zeilenlänge] ;
VAR  stelle : indexbereich ;
     karte : ARRAY indexbereich OF CHAR ;
```

aktuelle Parameter: auch "indizierte Bezeichner" zugelassen

```
vertauschen ( karte [stelle-1] , karte [stelle] ) ;
```

$$(* \quad \left(\begin{array}{c|c} \text{karte[stelle-1]} & \text{karte[stelle]} \\ a & b \end{array} \right) \vdash \left(\begin{array}{c|c} \text{karte[stelle-1]} & \text{karte[stelle]} \\ b & a \end{array} \right) *)$$

<u>Beispiel:</u> Baustein zur Überführung von a in a^b für ganze Zahlen a und $b \geq 0$, d.h.

$$\left(\begin{array}{c|c} x & n \\ \text{ganze Zahl } a & \text{ganze Zahl } b \end{array} \right) \vdash \left(\begin{array}{c|c} x & n \\ a^b & b \end{array} \right)$$

a) <u>Vereinbarung:</u>

formale Parameter

Prozedurkopf

```
PROCEDURE potenz ( VAR x : INTEGER; n : INTEGER) ;
```

Argument-Ergebnis-Parameter vom Typ INTEGER

Argument-Parameter vom Typ INTEGER

$$(* \quad \left(\begin{array}{c|c} x & n \\ a & b \end{array} \right) \vdash \begin{cases} \left(\begin{array}{c|c} x & n \\ a^b & b \end{array} \right) & \text{falls } b \geq 0 \\ \left(\begin{array}{c|c} x & n \\ 0 & b \end{array} \right) & \text{falls } b < 0 \end{cases} *)$$

Prozedurrumpf

```
VAR z : INTEGER ;
```

lokaler Bezeichner

ferner wird der Argument-Parameter n als lokaler Bezeichner angesehen

```
BEGIN
    IF n < 0
    THEN x := 0
    ELSE
        z := 1;
        (* z · x^n = a^b   , wobei a Anfangswert von x
                             b Anfangswert von n   *)
        WHILE n > 0 DO
            IF ODD(n) THEN z := z * x END;
            x := x * x;
            n := n DIV 2
        END;
        (* z · x^n = a^b  und n = 0 , d.h. z = a^b   *)
        x := z
    END
END potenz;
```

(Übung: man verifiziere dieses Programmstück)

Aufruf z.B. VAR zahl: INTEGER;
m, n : INTEGER;

aktueller Parameter für Argument-Parameter: Ausdruck vom Typ des formalen Parameters

aktueller Parameter für Argument-Ergebnis-Parameter: Bezeichner vom Typ des formalen Parameters

potenz (zahl, m+n+1)

$$\left(* \begin{pmatrix} zahl \\ a \end{pmatrix} \middle| \begin{matrix} m \\ i \end{matrix} \middle| \begin{matrix} n \\ j \end{matrix} \right) \vdash \left(\begin{matrix} zahl \\ a^{i+j+1} \end{matrix} \middle| \begin{matrix} m \\ i \end{matrix} \middle| \begin{matrix} n \\ j \end{matrix} \right) \quad *)$$

man führe die Anweisungen des "geeignet modifizierten und erweiterten" Prozedurrumpfes aus:

- im Rumpf die formalen Argument-Ergebnis-Parameter durch aktuelle Parameter (das müssen paarweise verschiedene Bezeichner, evtl. "erreichbar" (allgemeiner: Variablen), vom passenden Typ sein) ersetzen.

  [Bem: tatsächlich wird keine "textuelle Ersetzung" vorgenommen, sondern eine geeignete Korrespondenz hergestellt]

- lokale Bezeichner, d.h. die zu Beginn des Rumpfes vereinbarten und die formalen Argument-Parameter, von allen anderen "gültigen" Bezeichnern "unterscheidbar" machen (siehe dazu §19).

  [Bem: tatsächlich wird beim Aufruf der Prozedur neuer Speicherplatz für diese lokalen Bezeichner reserviert, der nach Beendigung der Ausführung wieder frei wird.]

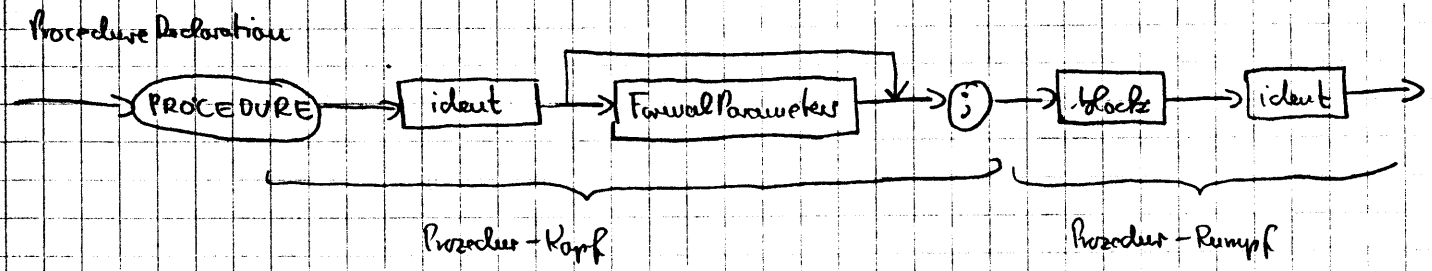
- die Werte der aktuellen Argument-Parameter (das müssen Ausdrücke vom passenden Typ sein) bestimmen und den (evtl. umbenannten) formalen Argument-Parametern zuweisen;

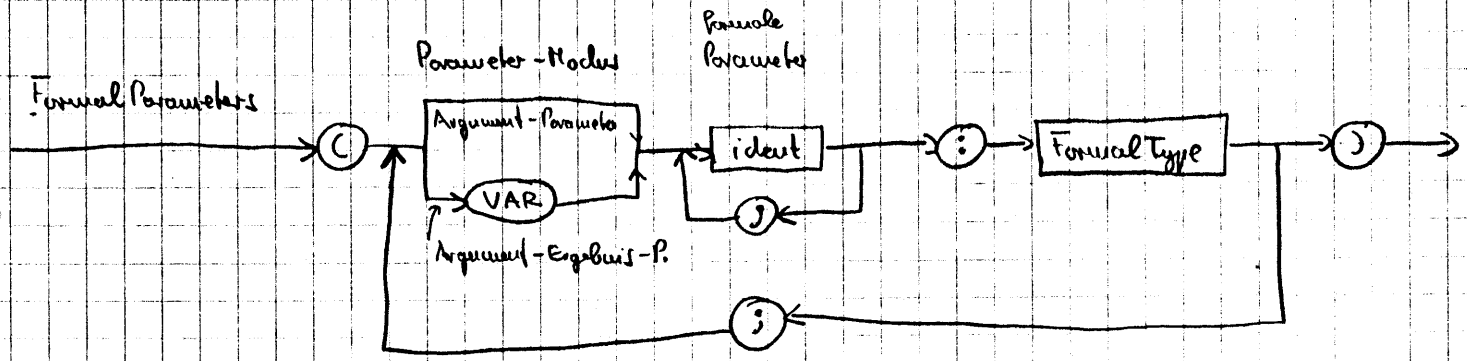
- dann die (evtl. durch Ersetzung und Umbenennung modifizierten) Anweisungen des Rumpfes ausführen.

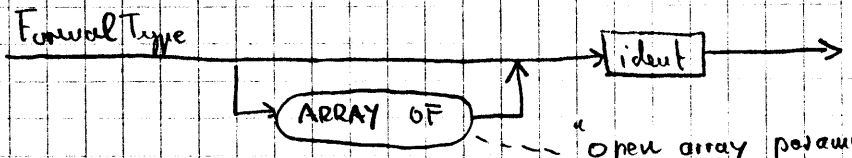
Das ist die umgangssprachliche Semantik von Prozeduraufrufen.

Ein Prozeduraufruf, d.h. ein vorher mit PROCEDURE vereinbarter Bezeichner mit passenden aktuellen Parametern, ist eine Anweisung.

MODULA-2 - Syntax der Prozedur-Vereinbarung ; vgl. dazu 8.6, Syntax von block:







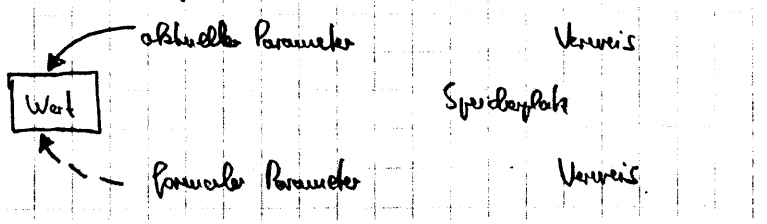
"open array parameter": Indexbereich 0..N, HIGH(s) liefert Feldlänge

es gibt also 2 Arten von formalen Parametern (in MODULA-2):

- Argument-Parameter (value parameter; Modus call-by-value):
  erwartet passenden Ausdruck als aktuellen Parameter

- Argument-Ergebnis-Parameter (variable parameter; Modus call-by-reference)
  d.h. der formale Parameter verweist nach Aufruf auf den selben Speicherplatz wie der aktuelle Parameter, woher folgende Ausdrucksweise rührt (vgl. 10.2):

(... | aktueller Parameter / Wert | ...) wird schlimmstenfalls gedeutet als



erwarten passenden Designator als aktuellen Parameter

Alle in den Anweisungen des Rumpfes benutzten Designatoren müssen vereinbart sein, und zwar

- als formale Parameter im Kopf,
- als lokale Bezeichner am Anfang des Rumpfes,
- als globale Bezeichner am Anfang des Moduls oder einer "übergeordneten" Prozedurvereinbarung (siehe § 19).

Das bedeutet, daß ein Prozeduraufruf die Werte von Designatoren ändern kann,

- die als aktuelle Parameter für formale Argument- Ergebnis- Parameter stehen

  [dies sollte der "Normalfall" sein]

- die als globale Bezeichner vorkommen

  [dies sollte wegen der Schwierigkeit, Seiteneffekte richtig zu übersehen, die Ausnahme bleiben; falls man Seiteneffekte verwendet, sollte man "stets" in einem Kommentar zum Prozedur-Kopf
  - alle globalen Bezeichner ausdrücklich aufzählen
  - die Wirkung der Prozedur auch für diese globalen Bezeichner beschreiben.]

Zu der Redeweise von 17.5.:

"Argumente" können sein: die beim Aufruf vorliegenden Werte der
- aktuellen Parameter für formale Argument-Parameter,
- aktuellen Parameter für formale Argument-Ergebnis-Parameter,
- im Rumpf vorkommenden globalen Bezeichner.

"Ergebnisse" können sein: die beim Verlassen der Prozedur vorliegenden Werte der
- aktuellen Parameter für formale Argument-Ergebnis-Parameter
- im Rumpf vorkommenden globalen Bezeichner

Die Werte aller lokalen Bezeichner sind nach dem Verlassen der Prozedur nicht mehr zugänglich.

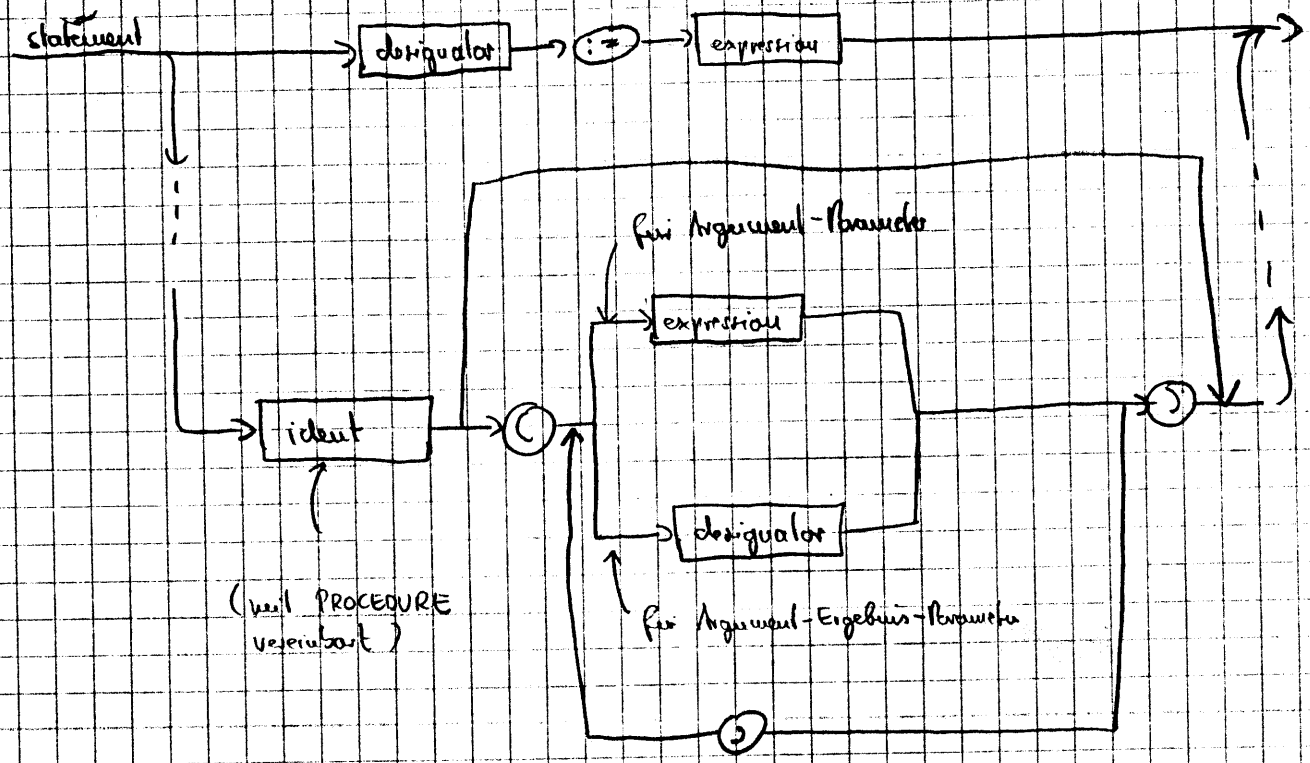
Einige Bemerkungen (die z.T. Schwächen von MODULA-2 aufzeigen)

1) Prozeduren können keine reinen Ergebnisse.

2) MODULA-2 ermöglicht Prozeduren mit beinahe beliebigen Seiteneffekten, was nicht im Sinne der systematischen Programmierung ist.

3) Formale Argument-Parameter benötigen beim Aufruf der Prozedur
Speicherplatz und Rechenzeit,
um die Werte der aktuellen Parameter den formalen Parametern zuzuweisen.
Für Argument-Parameter, die (große) Felder bezeichnen, mag dies aus Effizienzgründen unvermeidlich sein; dann muß man diese (entgegen ihrer Bedeutung im Sinne der systematischen Programmierung) als Argument-Ergebnis-Parameter behandeln, weil dafür nur ein Verweis gesetzt werden muß.

Das Syntaxdiagramm für MODULA-2 - Anweisungen (vgl S. 13.7 kann neu erweitert werden:



19. Lokalität

1. Gültigkeitsbereiche von Vereinbarungen

§ 2.4: Bezeichner sind Namen für Objekte.
Als Objekte (außer Variablen) kennengelernt (vgl. 8.6): Marken, Typen, skalare Werte, Felder, Prozeduren (andere Objekte werden noch folgen)

§ 8/§ 18: Bezeichner müssen unter Angabe des beabsichtigten Verwendungszweckes vereinbart werden.

Zwei Fragen sollen beantwortet werden:

1) Wo ist eine solche Vereinbarung gültig?
2) Wo bzw. wie lange kann man ein bezeichnetes Objekt mit Hilfe des Bezeichners "wiederfinden"?

Um die Frage 1) zu beantworten, definiert man zu jeder Bezeichner-Vereinbarung einen Gültigkeitsbereich.

Innerhalb eines Moduls gelten die in der Regel über die von den Blöcken des Moduls gebildeten Strukturen.

(Die Regeln bei mehreren Blöcken werden wir später behandeln, siehe § 28)

In MODULA-2 darf (im Gegensatz zu manchen anderen Programmiersprachen) ein Block nur als Modulrumpf direkt hinter dem Modulkopf, bzw. als Prozedurrumpf direkt hinter dem Prozedurkopf stehen. Im Folgenden wollen wir einfach den Programmodul ebenfalls wie eine Prozedur behandeln.

In anderen Sprachen, z.B. ALGOL oder SIMULA können zusammengesetzte Anweisungen gebildet werden und wie Blöcke behandelt werden; z.B.

die Anweisungsfolge $S_1 ; \ldots ; S_n$

wird zu einer zusammengesetzten Anweisung durch Einklammern:

BEGIN $S_1 ; \ldots ; S_n$ END

wird dann auch lokale Bezeichner vereinbaren, so daß ein Block entsteht:

BEGIN Vereinbarungen ; $S_1 ; \ldots ; S_n$ END

Vereinbarungs-Struktur (statische, textuelle Struktur):

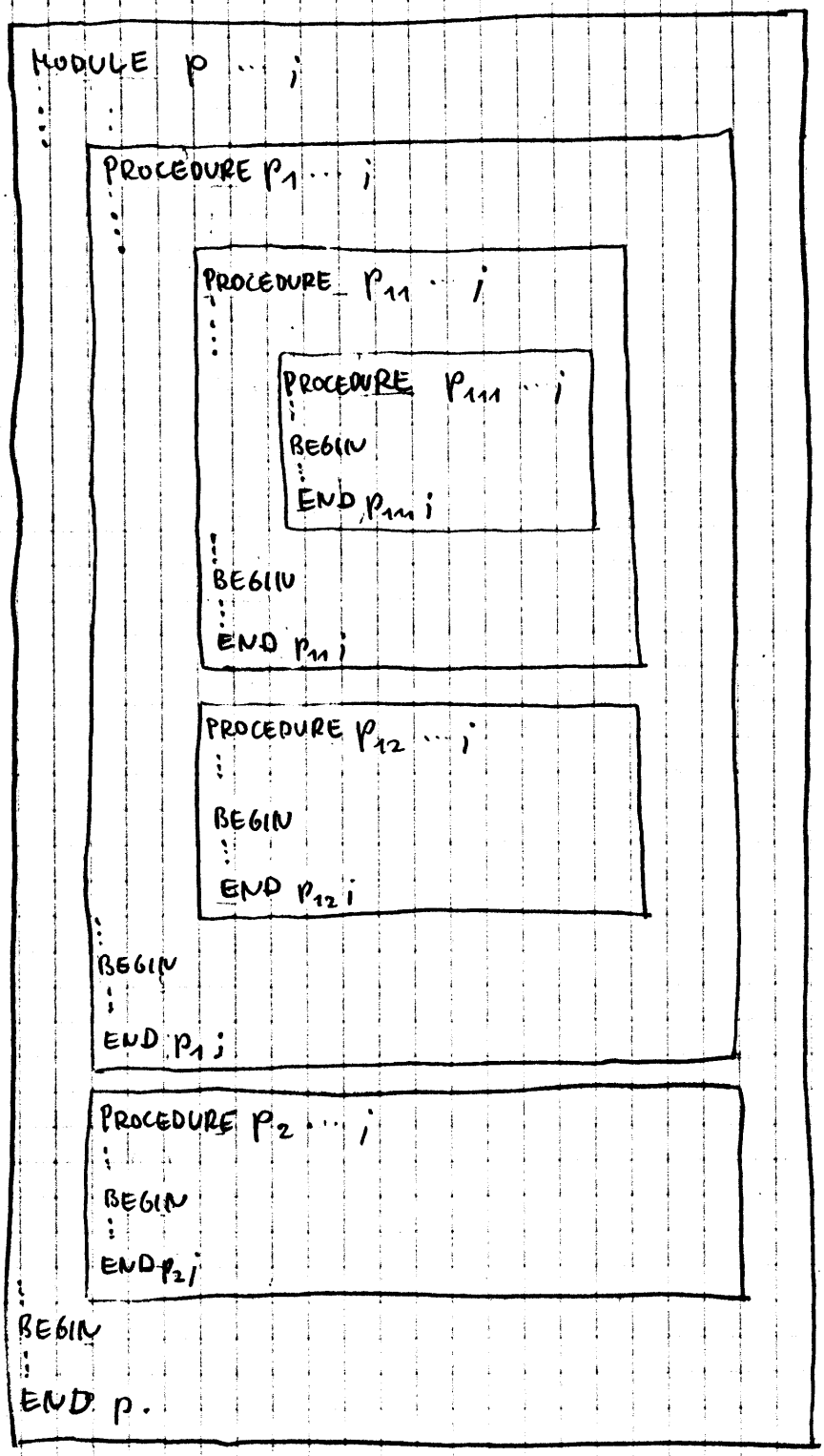
Sei PT := Menge der Prozedurtexte eines MODULA-2 - Programmoduls p
Es gilt $p \in PT$. Auf PT definieren wir eine zweistellige Relation $\sqsubset$ durch

$x \sqsubset y$: gdw x ist (textuell) in y enthalten

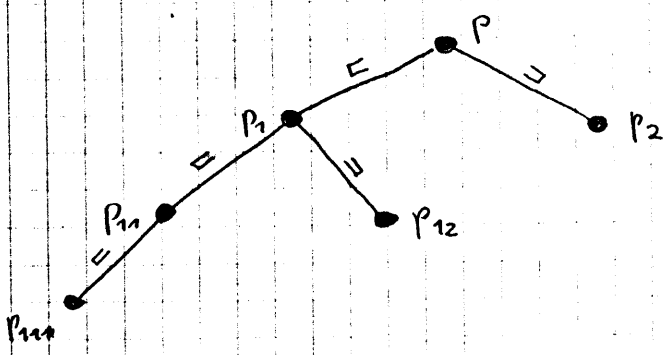
(es gelte auch $x \sqsubset x$)

$(PT, \sqsubset)$ ist eine spezielle Halbordnung, wie sie aus den Syntaxdiagrammen ergibt: Programmbereich ist dies ein Baum.

z.B.



Unter Abstraktion der gesamten Texte und Benennung der Texte durch die zuständigen Bezeichner ergibt sich folgender Baum:



Für MODULA-2 wird diese Vereinbarungsstruktur benutzt, um den Gültigkeitsbereich einer Bezeichner-Vereinbarung zu definieren. (Es gibt Programmiersprachen, die anders vorgehen.)

0) Eine Bezeichner-Vereinbarung ist das Vorkommen eines Bezeichners

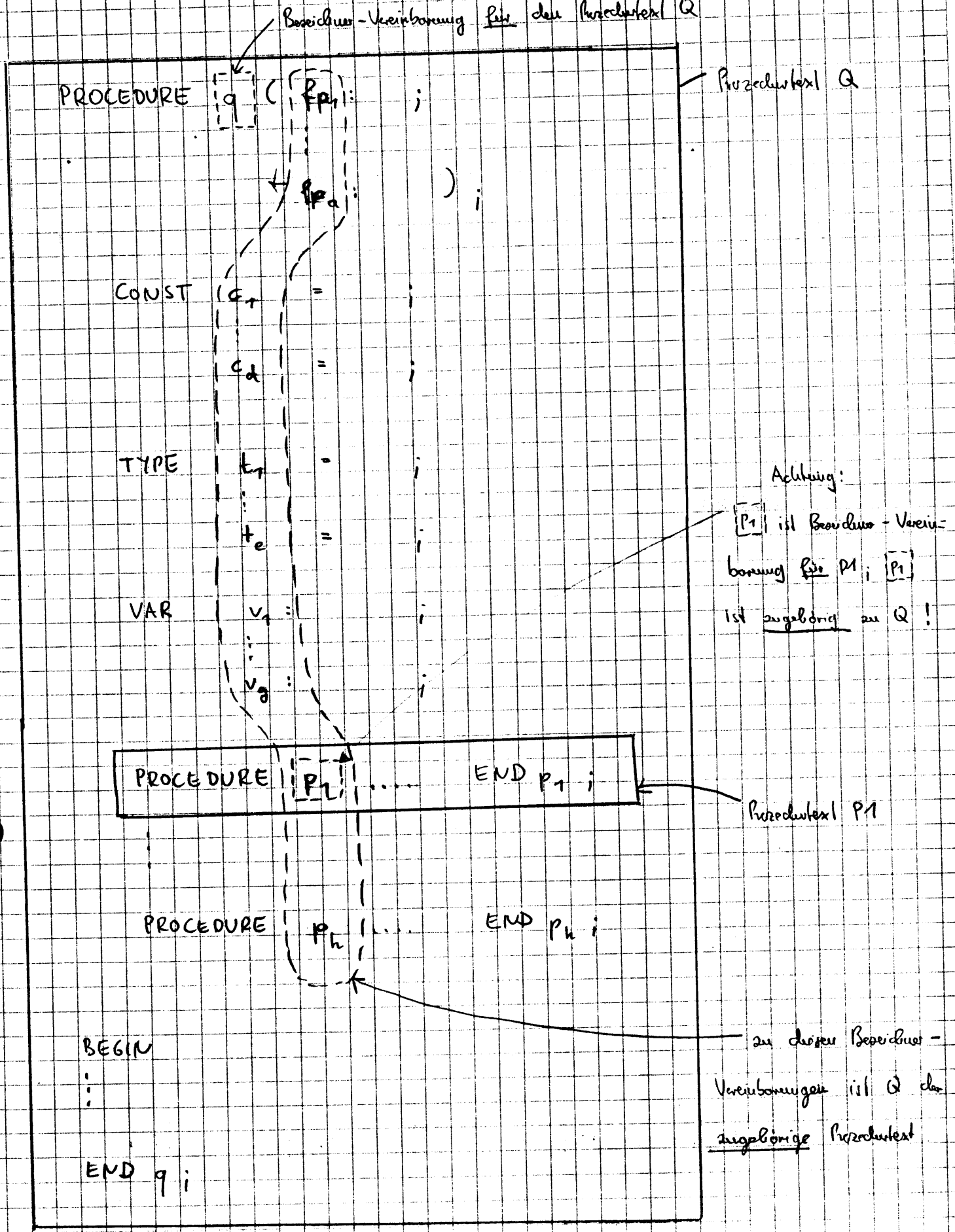
- im Kopf eines Prozedurtextes q als formaler Parameter oder
- im Vereinbarungsteil des Rumpfs eines Prozedurtextes q "direkt" hinter den Trennzeichen CONST, TYPE, VAR, PROCEDURE (bzw. MODULE).

Man sagt dann, daß für solch eine Bezeichner-Vereinbarung q der zugehörige Prozedurtext ist. Zur Erläuterung siehe Schema auf Seite 19.3.

1) Der Gültigkeitsbereich einer solchen Bezeichner-Vereinbarung ist dann der zugehörige Prozedurtext q, einschließlich aller $\sqsubset$-höheren Prozedurtexte, mit folgender Ausnahme: (d.h. aller r mit $r \sqsubset q$)

2) Ist $\bar{q} \neq q$ und ist $\bar{q}$ zugehörig zu einer weiteren Bezeichner-Vereinbarung mit dem selben Bezeichner, so gehört der Prozedurtext $\bar{q}$ (einschließlich aller $\sqsubset$-höheren Prozedurtexte) nicht zum Gültigkeitsbereich der Bezeichner-Vereinbarung.

3) Für alle Standardbezeichner, bzw. den Programmodul-Bezeichner, gibt es einen zugehörigen gedachten Prozedurtext, in dem der Modul direkt enthalten ist.



Schema zur Definition von "Bezeichner-Vereinbarung" und von "zugehöriger Prozedurtext" (Seite 19.4)

Bsp: alle Bezeichner-Vereinbarungen durch □ᵢ gekennzeichnet

```
MODULE [p]1 ;

  VAR [x]2 : i
      [y]3 : i
      [z]4 : i

  PROCEDURE [p1]5 ([y]6 : );

    VAR [z]8 : i

    BEGIN

    END p1;

BEGIN

END p.
```

Prozedurtext P

[P]1 ist Bezeichner-Vereinbarung für den Programmodultext (Prozedurtext) P

Prozedurtext P1

[p1]5 ist Bezeichner-Vereinbarung für den Prozedurtext P1

Bezeichner-Vereinbarung	Gültigkeitsbereich
P_1 *	IMAGINÄR (geladene umfassende Rozedur)
X_2	P
Y_3	P außer P_1
Z_4	P außer P_1
$P1_5$	P
Y_6	P_1
Z_8	P_1

Bemerkungen: 1) Die Definition von Gültigkeitsbereich ist deshalb so kompliziert, weil Bezeichner wiederholt vereinbart werden können*) und wir somit vom Verdecken von Bezeichnern sprechen müssen.

2) Wird ein Bezeichner mehrfach vereinbart, so sind die Gültigkeitsbereiche der (betreffenden) Bezeichner-Vereinbarungen überlappungsfrei: kommt ein Bezeichner in einer Anweisung vor, so gibt es genau eine Bezeichner-Vereinbarung, in deren Gültigkeitsbereich die Anweisung liegt; diese Bezeichner-Vereinbarung ist dann maßgeblich; da MODULA-2 die statische Vereinbarungsstruktur benutzt, kann man die maßgeblichen Vereinbarungen aus dem Programmtext ersehen (ohne das Programm auszuführen)

*) aber: Prozedurname und die Bezeichner in den direkt darauffolgenden Bezeichner-Vereinbarungen müssen paarweise verschieden sein!

3) Der Gültigkeitsbereich einer Bezeichner-Vereinbarung ist stets ein Teilbaum der Vereinbarungs-Struktur, der allerdings bei manchen Vereinbarungen "Löcher" haben kann.

Die Frage 2) ist schwieriger zu beantworten, weil wir dafür nicht nur die Struktur des Programmtextes, sondern auch den Ablauf der Programmausführung betrachten müssen.

<

Formale Aufruf-Struktur

Sei (wieder) PT := Menge der Prozedurtexte eines Programmes p.

Auf PT definieren wir eine zweistellige Relation rf, "ruft formal", durch

x rf y : gdw x enthält im Rumpf eine Anweisung, die "einen Aufruf von y beinhaltet".

d.h. der Rumpf von y, geeignet abgeändert wie auf S. 18.5 beschrieben, soll aufgerufen werden.

Bemerkungen:

1) Diese Struktur beschreibt "formale Aufrufe", d.h. sie beachtet nur den Programmtext, nicht aber die Programmausführung. x rf y bedeutet nicht notwendig, daß es irgendeinen Programmlauf gibt, in dem der den Aufruf von y enthaltende Anweisung erreicht wird; dieses Problem ist unentscheidbar, vgl. §5, §13, und verbleibt nicht danach der vollständigen algorithmischen Behandlung (etwa durch den Compiler).

2) Der formale Aufruf-Struktur ist graphentheoretisch ein gerichteter Graph mit Schlingen (d.h. Prozeduren dürfen sich auch selbst aufrufen).

3) Die formale Aufruf-Struktur muß verträglich sein mit der auf der Vereinbarungs-Struktur gestützten Gültigkeitsbereichen:

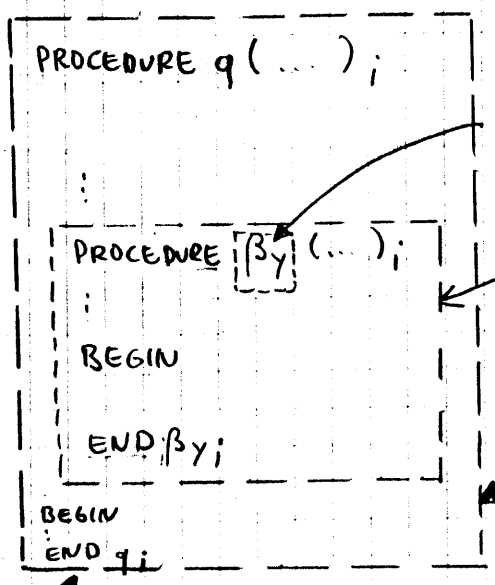
x rf y ist nur möglich, wenn x im Gültigkeitsbereich der Bezeichner-Vereinbarung für y enthalten ist, d.h.

$x \sqsubset_{(=)}$ Gültigkeitsbereich der Bezeichner-Vereinbarung für y

Prozedertext, etwa beginnend mit procedure β_y ...

Bezeichner-Vereinbarung für y

schematisches Beispiel:



x rf y ist möglich, wenn x innerhalb dieses Programmtexts, evtl. außerhalb der Lücken liegt.

Dieser Programmtext, evtl. mit Lücken bei mehrfacher Vereinbarung von β_y, ist der Gültigkeitsbereich der Bezeichner-Vereinbarung für y.



Der formale Aufruf-Struktur liegt (statisch) fest, welche Aufrufpfeile formal möglich sind.

Jede Ausführung des Programms bestimmt einen (gegebenenfalls "pulsierenden") Weg im Graphen der formalen Aufrufstruktur:

dieser Weg beginnt beim Programmodul p,

jeder tatsächliche Aufruf einer Prozedur bedeutet, eine Kante im Graphen der formalen Aufruf-Struktur weiterzugehen,

jede Beendigung der Ausführung bedeutet, die beim Aufruf begangene Kante wieder zurückzugehen.

also: Wenn das Programm ordnungsgemäß endet, dann muß jeder Hinschritt um Rückung der Kanten später durch einen Rückschritt zurückgenommen werden.

Die jeweils noch nicht zurückgenommenen Hinschritte bilden die <u>dynamische Aufrufkette</u>: diese enthält also die Kette der momentan erfolgten, aber noch nicht beendeten Aufrufe von Prozeduren.

Schließlich müssen wir noch die Konzepte des <u>Bindungszeitpunkt</u> und der <u>Lebenszeit</u> kennenlernen. Dazu muß man (wie in §10 bei der Interpretation $v \leftarrow e$ der Wertzuweisung) einen Bezeichner für Werte (Designator) ansehen als einen Repräsentanten von Speicherplatz. In dieser Eigenschaften redet man auch von <u>Variablen</u>: der Speicherplatz kann "variable Werte" vom vereinbarten Typ enthalten.

<u>Bindungszeitpunkt</u>: Zeitpunkt der Zuordnung von Bezeichner zu Speicherplatz

<u>Lebenszeit</u>: Zeit-(spanne) vom Bindungszeitpunkt bis zum Zeitpunkt der Auflösung der Verbindung.

Bindungszeitpunkt und Lebenszeit können auf verschiedene Arten geregelt werden:

statisch: alle Variablen werden zum Beginn des Programmlaufs gebunden und leben bis zum Programmende;

lokal: Variable werden beim Aufruf einer Prozedur (in manchen Programmiersprachen auch gemäß S. 19.2) gebunden und leben jeweils bis zur Beendigung der Ausführung.

dynamisch: durch ausdrückliche Anweisungen, siehe §27.

MODULA-2 verwendet die <u>lokale</u> Regelung (neben der später zu besprechenden dynamischen):

19.12

Gemäß der umgangssprachlichen Semantik von Prozeduraufrufen werden die lokalen Bezeichner lokal gebunden und leben bis zur Beendigung der Ausführung der Prozedur.

Bei mehrfach verschachtelten Bezeichnern und bei sich selbst und sich gegenseitig aufrufenden Prozeduren ist wichtig zu beachten, daß bei jedem Aufruf jedem lokalen Bezeichner jeweils neuer Speicherplatz zugeordnet wird.

Damit können wir Frage 2) beantworten:

"wo kann wie lange kann ein bezeichnetes Objekt mit Hilfe des Bezeichners (Variablen) "wiedergefunden" ?

Antwort 1:

Während der Lebenszeit des Bezeichners mit folgender Ausnahme:
erfolgt während der Lebenszeit ein Aufruf einer Prozedur q, die den Bezeichner nochmals als lokalen Bezeichner enthält, so wird während der Ausführung von q.

Antwort 2:

Kommt in einer Anweisung ein Bezeichner (Variable) vor, so ist folgende Zuordnung zwischen Bezeichner und Speicherplatz maßgeblich:

- suche die Bezeichner-Vereinbarung, in deren Gültigkeitsbereich q die Anweisung liegt;
- suche in der dynamischen Kette den zeitlich letzten Aufruf von q;
- die bei diesem Aufruf erfolgte Zuordnung ist maßgeblich.

Einige Bemerkungen zur Semantik von Prozeduraufrufen

Grundkurs

1) umgangssprachlich

2) operational
durch Flußdiagramm
bzw. Übersetzung in SMAL:

- statischer Programmtext
- dynamisches Merken der momentan bearbeiteten Stelle des Programmtexts

Prozeduren

1) umgangssprachlich

2) operational

ohne rekursive Aufrufe

theoretisch möglich:

vor Ausführung jedes Prozeduraufrufs textuell ersetzen durch erweiterten und modifizierten Rumpf (vgl. S. 18.5)

dann: – statischer Programmtext
– dynamisches Merken der momentan bearbeiteten Stelle des Programmtexts (wie im Grundkurs)

mit rekursiven Aufrufen

nicht möglich, da sich ein unendlicher Text ergeben würde!

tatsächlich (aus Effizienzgründen):

Prozedurtexte werden nicht vervielfacht, sondern mit geeigneter Erweiterung und Modifikation für jeden Aufruf wiederverwendet:

bei Aufruf: – neuen Speicherplatz bereitstellen
– an den Anfang des erweiterten und modifizierten Rumpfes springen

bei Beendigung: – Speicherplatz wieder freigeben
– zurückspringen direkt hinter die Aufrufstelle

tatsächlich:

momentan bearbeitete Stelle im Prozedurtext für jeden Block in der dynamischen Kette

zusätzlich muß sich das Laufzeitsystem merken: dynamische Kette und

19.13

3) axiomatische

ohne rekursive Aufrufe

Hierzu muß man in der Beschreibungssprache, der Logik, die "Parameterübergaben" geeignet nachbilden;

die Beweisverfahren der Programmierung sind auch auf die Beschreibungssprache der Logik anwendbar!

Dies alles ist nicht schwer, aber etwas zeitraubend und soll deshalb nicht vorgeführt werden

2) man kann zeigen: beliebige rekursive Prozeduren kann man mit Hilfe der while-Anweisung und "geeigneter Datenstrukturen" simulieren (siehe Laufzeitsysteme!)

3) axiomatische

und rekursive Aufrufen

Die hier auftretenden Probleme sind recht schwierig und gehören z. Bsp in eine Vorlesung "Theorie der Programmierung":

die im Grundkurs behandelte Wiederholungsanweisung (und while) stellt den einfachsten Spezialfall dar (vgl. S. 12.19):

19.14

1) eine Anweisung der Form

WHILE B DO S END

läßt sich durch einen Aufruf der folgenden Prozedur simulieren:

```
PROCEDURE simuliert;
BEGIN
  IF B THEN S; simuliert END

END simuliert;
```

Ein größeres Beispiel zu den Strukturen auf PT

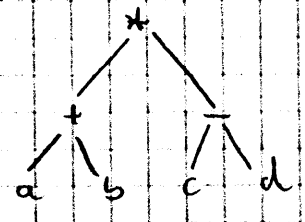
Ausdrücke in MODULA-2 werden in Infix-Schreibweise angegeben, d.h. bei zweistelligen Funktionszeichen steht das Funktionszeichen zwischen (infix) den Argumenten. Diese Schreibweise ist uns geläufig, doch für manche Zwecke unhandlich, z.B. manchmal zum Klammern. Eine andere Schreibweise ist die Postfix- (bzw. Präfix-) Schreibweise, bei der das Funktionszeichen hinter (bzw. vor) den Argumenten steht.

Bsp.

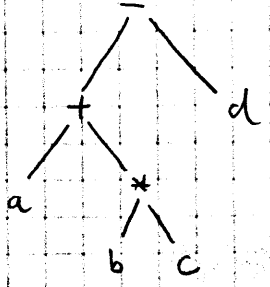
	Infix	Postfix
a)	(a+b) * (c-d)	a b + c d - *
b)	a+b * c-d	a b c * + d -
c)	a + b * (c-d)	a b c d - * +

Die Postfix- (bzw. Präfix-) Schreibweise kann man systematisch bestimmen, z.B. über die Strukturbäume:

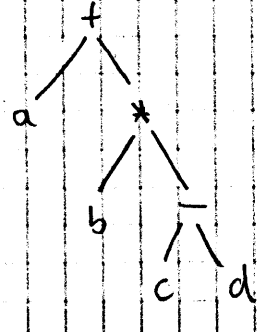
a)



b)



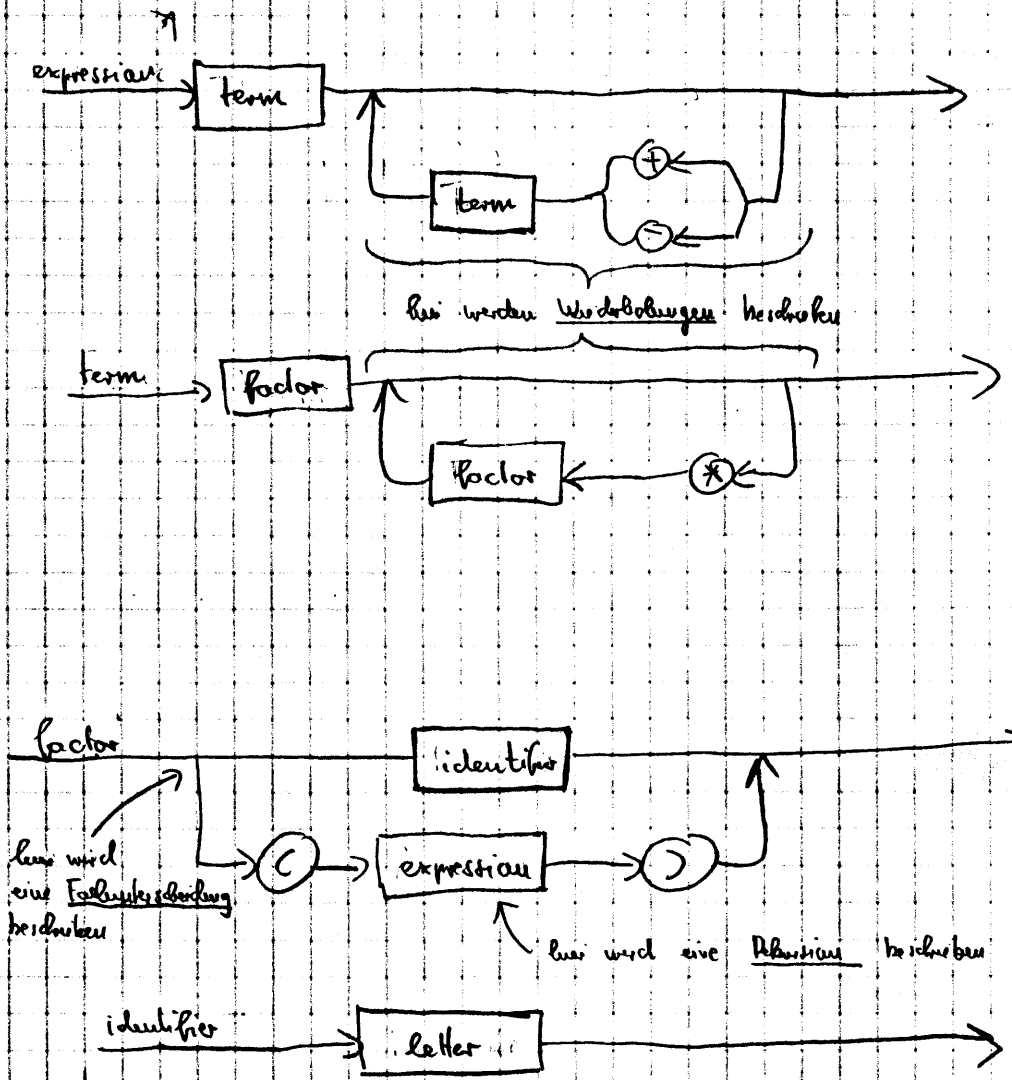
c)



Man erhält die Postfix-Schreibweise, indem man den Baum nach folgender (rekursiver) Regel durchläuft:

- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum
- besuche Wurzel

Aufgabe: Wandle einen Ausdruck von Infix-Schreibweise in Postfix-Schreibweise um. Zur Vereinfachung wollen wir uns auf Ausdrücke der folgenden einfachen Form beschränken:



Das folgende Programm erstellt nicht ausschließlich die Strukturbäume; stattdessen bilden wir die rekursive Natur der obigen Syntaxdiagramme durch Prozeduren nach, die sich rekursiv aufrufen können.

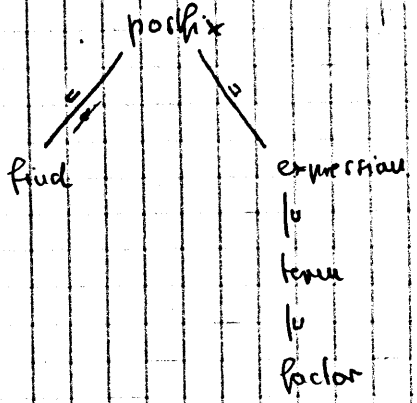
Das Programm ist übernommen aus:

Jensen / Wirth: PASCAL-User Manual and Report, S. 75

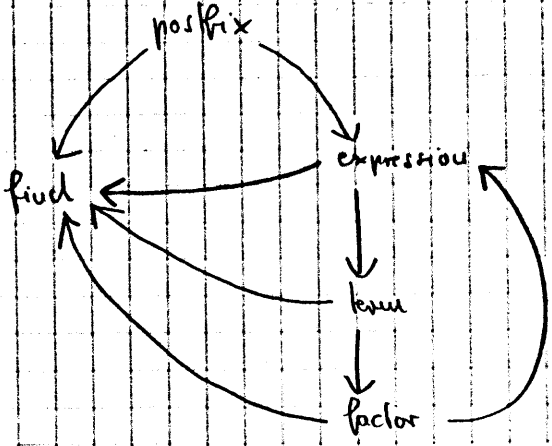
Achtung: es werden keine "Fehler" abgefangen! / Links dann Ausdruck stehen ein '.'!

```
MODULE postfix;
FROM InOut IMPORT Read, Write, WriteLn, WriteString;
  VAR ch: CHAR;   (* beobachtet jeweils das zuletzt gelesene Zeichen *)
  PROCEDURE find;
  BEGIN  REPEAT Read(ch) UNTIL ch # " ";
  END find;

  PROCEDURE expression;
    VAR op: CHAR;
    PROCEDURE term;
      PROCEDURE faclor;
      BEGIN  IF ch = "(" THEN find; expression (*ch=")"*)
                         ELSE (* ch identifier *) Write(ch)      END;
             find
      END faclor;
    BEGIN  faclor;
           WHILE ch = "*" DO find; faclor; WriteString("*") END
    END term;
  BEGIN  term;
         WHILE (ch = "+") OR (ch = "-") DO
           op := ch; find; term; Write(op)
         END
  END expression;

BEGIN  find;
       REPEAT WriteString(" "); expression; WriteLn UNTIL ch = ".";

END postfix.
```

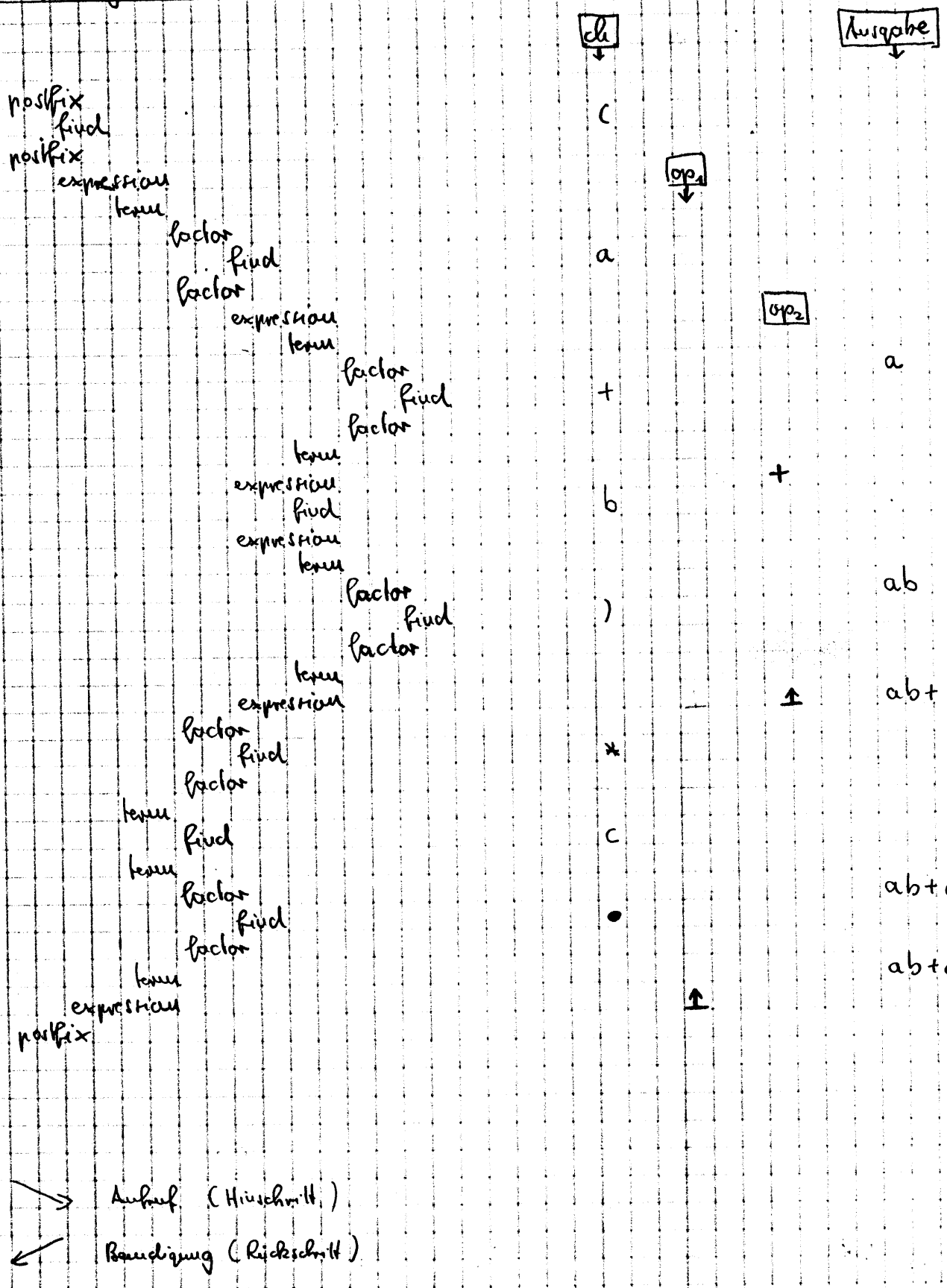
Vereinbarungs-Struktur:



formale Aufruf-Struktur:



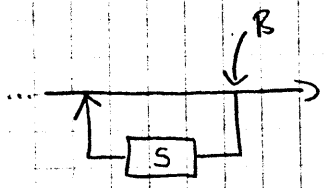
Auswertung von (a+b)*c. bestimmt folgenden Weg:



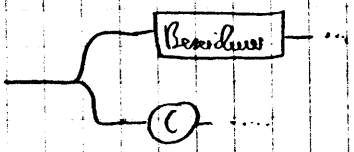
Das Programm postfix sollte jeder sehr gründlich studieren. Viele bislang besprochene Gesichtspunkte kann man an diesem Beispiel betrachten; z.B:

1. Ein wesentlicher Teil der Aufgabe ist durch die Beschreibung der Argumente in Form der Syntaxdiagramme gegeben.

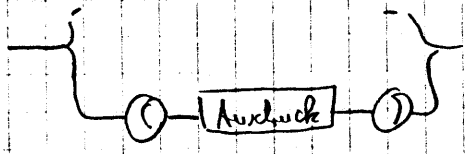
   Das Programm postfix spiegelt sehr genau die verschiedenen Konstrukte der Syntaxdiagramme wieder:



durch WHILE B DO S END – Konstruktion in term und expression



durch IF_THEN_ELSE_END Konstruktion in factor



durch Rekursion in factor

2. Prozeduren sind jeweils lokal dort vereinbart, wo sie wirklich benötigt werden.

3. Der Bezeichner op is lokaler Bezeichner in der Prozedur expression. Das ist notwendig, damit für jeden (geschachtelten) Aufruf von expression neuer Speicherplatz (zum Merken des additiven Operators) bereitgestellt wird (und somit die noch benötigten Operatoren nicht zerstört werden)

4. In der Prozedur term "merkt" man sich den einzigen Operator * "über die Stelle".

5. Die Prozedur Find ist abgesichert gegen den Fall, daß eine Eingabe (vernünftigerweise) Leerzeichen enthält.

Insgesamt ist das Programm nicht gegen beliebige Eingabefehler abgesichert; für praktische Zwecke müßten die Zuverlässigkeit und Robustheit hoher überall Tests eingebaut werden, die systematisch falsche Eingaben erkennen und dann Fehlermeldungen ausgeben.

6. Entgegen dem allgemeinen Rat von Seite 18.7 wird hier als globales Bezeichner verwendet: man spricht in diesem Zusammenhang von einem unterdrückten Parameter.

Da die Situation hier insgesamt überschaubar ist, konnte (der Parameter für) das jeweils zuletzt gelesene Zeichen unterdrückt werden.

7. Bei rekursiven Prozeduren muß man sicherstellen, daß jede Ausführung schließlich abbricht:

Für die Wiederholungs-Anweisung WHILE B DO S END ist folgende Bedingung notwendig für den Abbruch: S muß eine Anweisung enthalten, die den Wert von B ändert.

Für eine rekursive Prozedur P ist entsprechend die folgende Bedingung notwendig für den Abbruch:

P muß eine bewachte Anweisung beinhalten, die für mindestens einen Wächter rekursionsfrei ist und die in den Zweigen mit Rekursion die Werte der Wächter verändert.

Die Prozedur Factor ist nach dieser Regel aufgebaut.

20. Funktionsprozeduren

So wie man mit Prozeduren neue Anweisungen definieren kann, kann man mit dem Sprachmittel der Funktionsprozedur neue Funktionszeichen definieren.

Im wesentlichen überträgt sich alles Gesagte über Prozeduren auch auf Funktionsprozeduren, wobei diese genau ein Ergebnis haben (sollten), das besonders gekennzeichnet wird.

Beispiel: Funktion zur Berechnung der 3. Potenz einer reellen Zahl

a) Vereinbarung

Typ des Funktionsergebnis

```
PROCEDURE kubik (x : REAL) : REAL
```

formaler Argument-Parameter vom Typ REAL

Bezeichner, "neues Funktionszeichen"

```
BEGIN   RETURN x * x * x
END kubik;
```

dem Funktionszeichen wird der Wert eines Ausdrucks vom passenden Typ zugewiesen; dieser ist das gelieferte Ergebnis

b) Aufruf

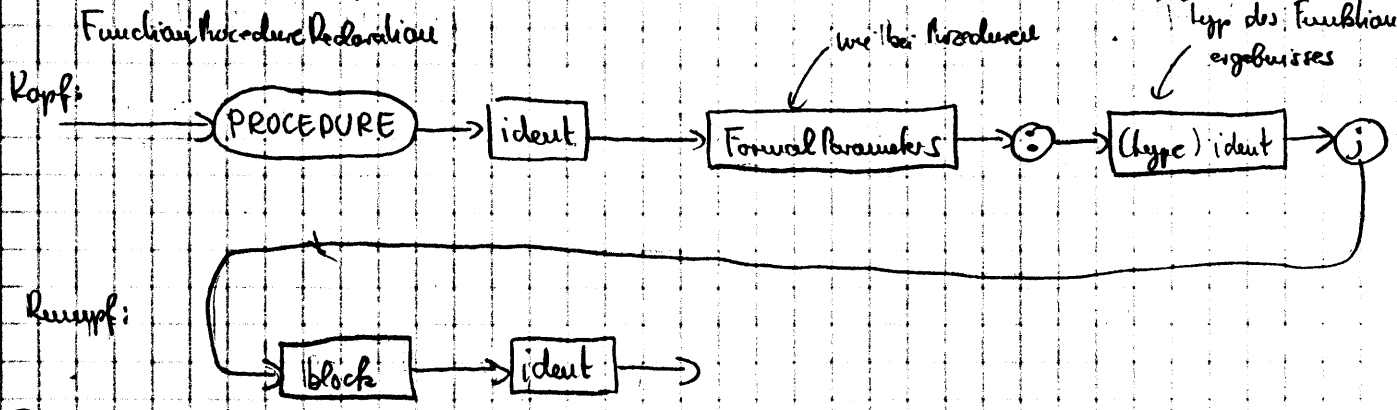
z.B.

```
VAR a,b : REAL;
  :
  b := kubik (a + 1.5);
```

aktueller Parameter, Ausdruck vom Typ REAL

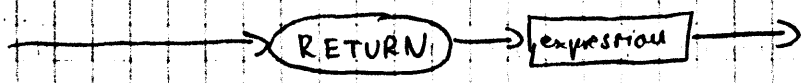
MODULA-2 - Syntax der Funktionsprozedur-Vereinbarung, vgl. dazu 8.6, Syntax von Block

Funktionsprozedurdeklaration:



Der Typ des Funktionsergebnisses muß sein: skalar, Untertyp (oder Zeiger, ⇒§27)

Der Rumpf muß mindestens eine RETURN-Anweisung enthalten:



Ausgangssprachliche Semantik: der Ausdruck wird ausgewertet und durch den Funktionsprozedur-Bezeichner bezeichnet; anschließend wird die Ausführung der Funktionsprozedur beendet

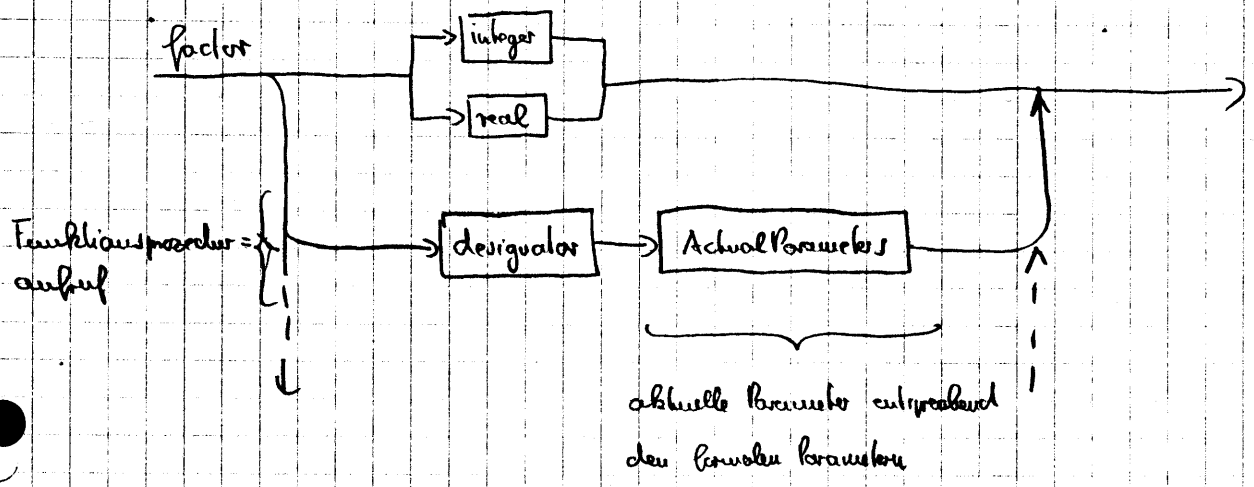
Achtung: Enthält der Rumpf "globale Bezeichner" oder formale Argument- - Ergebnis-Parameter, so liefert ein Funktionsaufruf nicht nur einen Wert, sondern bewirkt auch einen Seiteneffekt:

abgelieferter Wert: gemäß ausgeführter RETURN-Anweisung

Seiteneffekt: mögliche Änderungen der aktuellen Argument- Ergebnis-Parameter und globale Bezeichner

Im allgemeinen sollten Funktionen Seiteneffekt-frei geschrieben werden, um im Sinne des systematischen Programmierens schreibbare Programme zu vermeiden.

Funktionsprozedur - Aufruf innerhalb von Ausdrücken, vgl. § 9.7:



Dies verdeutlicht noch einmal die Forderung, Seiteneffekt-frei zu programmieren: die Auswertung eines Ausdrucks sollte keine bezeichneten Werte verändern!

Beispiel: Berechne $f(x) = \sum_{i=0}^{2} a_i x^i$ für Werte von x im Intervall $[-10, 10]$ in der Schrittweite 0,5.

```
MODULE Tabelle;

FROM InOut IMPORT WriteString, WriteLn;
FROM RealInOut IMPORT ReadReal, WriteReal;

CONST   grad = 2;
        unten = -10.0;
        oben = +10.0;
        schritt = 0.5;
        epsilon = 0.1;

TYPE    index = [0.. grad];
        koeffizienten = ARRAY index OF REAL;

VAR     a : koeffizienten;
        i : index;
        x : REAL;
        grenze : REAL;

PROCEDURE horner ( grad: INTEGER; a : koeffizienten; arg: REAL ) : REAL;
  VAR   i : index;
        s : REAL;
  BEGIN s := 0;
    FOR i := grad TO 0 BY -1 DO s := a[i] + arg * s END;
    RETURN s
  END horner;
```

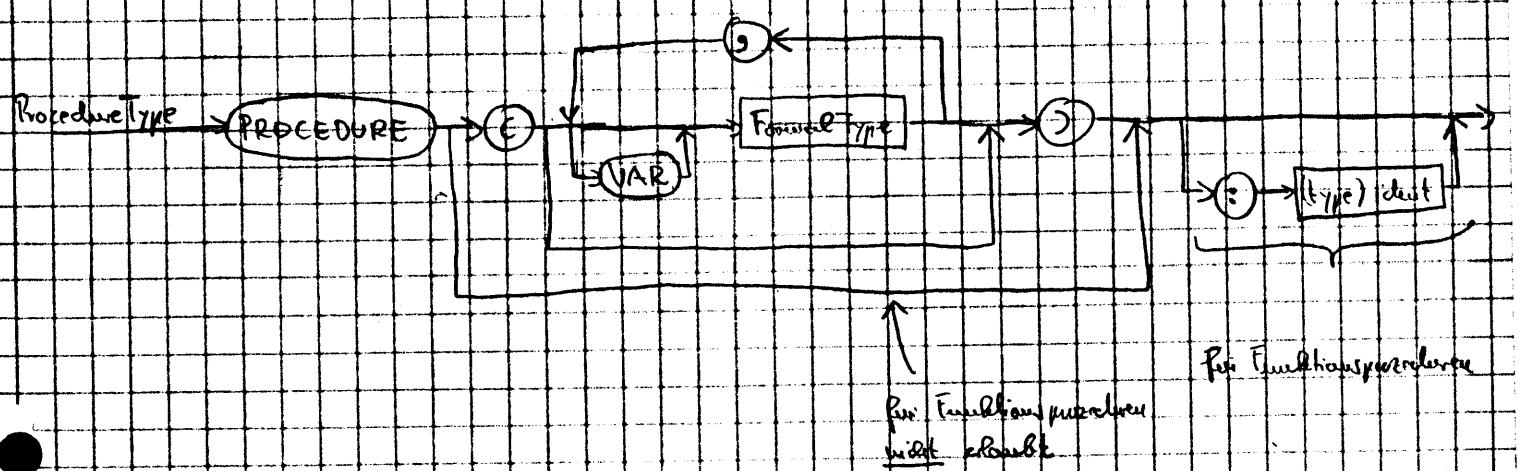
aus Effizienzgründen, für großen Grad, hier evtl. VAR anfügen

```
BEGIN
    FOR i := 0 TO grad DO ReadReal (a[i]);
    grenze := oben + epsilon;
    x := unten;

    WHILE x <= grenze DO
        BEGIN WriteString("x = "); WriteReal(x);
              WriteString("f(x) = "); WriteReal( horner(grad, a, x ));
              WriteLn;
              x := x + schritt
        END
END tabelle.
```

Einige Funktionsprozeduren sind im allgemeinen standardmäßig definiert, z.B. ODD, CHR, INC, DEC.

21. Prozedurtypen

Eine Prozedur kann als ein Objekt angesehen werden.
In der Prozedurvereinbarung wird es bezeichnet, so wie in der Konstantenvereinbarung ein Wert bezeichnet wird.
MODULA-2 erlaubt darüberhinaus (mit einigen Einschränkungen) auch Prozeduren variabel zu bezeichnen, so wie Werte variabel bezeichnet werden.
Dies geschieht für Prozeduren wie für Werte mittels der Wertzuweisung und der Parameterübergabe beim Prozeduraufruf. Variable vom Typ PROCEDURE werden wie folgt vereinbart



Beispiel:

```
TYPE Funktion = PROCEDURE (REAL) : REAL ;
     Prozedur = PROCEDURE (CARDINAL, CARDINAL) ;

VAR        f : Funktion ;
           p : Prozedur ;

PROCEDURE sinus (argument : REAL) : REAL ;
  :
  f := sinus ;
  :
  p := WriteCard ;
```

Beispiel:

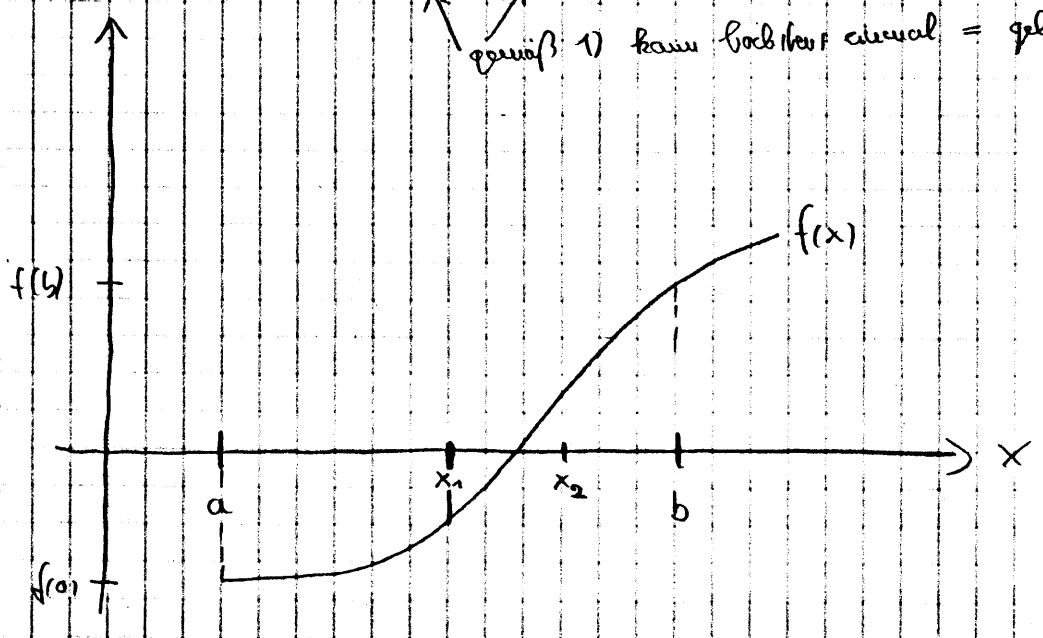
Man berechne die Nullstelle einer stetigen Funktion $f(x)$, die in einem gegeben Intervall $[a,b]$ mit $a<b$ folgende Eigenschaften hat:

1) f hat genau eine Nullstelle im Intervall $[a,b]$

2) $f(a) \leq 0 \leq f(b)$

oder $f(a) \geq 0 \geq f(b)$

gemäß 1) kann Gleichheit einmal = gelten!



Speziell sollen die Nullstellen von den Funktionen

Sinus in $[-1,+1]$ und Cosinus in $[-1,+1]$

berechnet werden. Wir benötigen eine MODULA-2 - Beschreibung einer Abbildung der folgenden Form:

$$null : \{f \mid f: X \to \mathbb{R}\} \times \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$$

Für einen Zustand $(f\ a\ b)$ mit den Eigenschaften 1) und 2) (für die gesuchten Objekte) wird ein Zustand gebildet, in dem null diejenige $x \in [a,b]$ mit $f(x)=0$ bezeichnet.

Programmgerüst:

```
MODULE versuch;
  :
TYPE StetigeFunktion = PROCEDURE(REAL):REAL;          geeigneter Prozedurtyp
PROCEDURE Sinus(x: REAL): REAL; ... END Sinus;
PROCEDURE Cosinus(x: REAL): REAL; ... END Cosinus;

PROCEDURE null ( f: StetigeFunktion; a,b: REAL ): REAL;
  :                                                    } Vereinbarung
  BEGIN                                                  von null
        ... = ... f(...) ...
  END null;

  :
BEGIN
  :
  ... null (Sinus, -1.0, 1.0) ...                      Aufrufe von null
  ... null (Cosinus, -1.0, 1.0) ...
  :
END versuch.
```

Vorüberlegungen

Sei zunächst einmal f steigend, das soll hier heißen, wie im Bild:

$$f(a) \leq 0 \leq f(b)$$

Wir merken uns, daß f steigend ist vermöge eines Boolschen Bezeichners s, etwa s bezeichne true.

Für den Mittelpunkt $x = \frac{a+b}{2}$ sind zwei Fälle möglich:

Fall 1: $f(x) < 0 \Leftrightarrow (f(x) < 0) = s$ $\Rightarrow$ Nullstelle im Intervall $[x,b]$ und f ist nämlich bzg $[x,b]$ ebenfalls steigend

Fall 2: $f(x) \geq 0 \Leftrightarrow (f(x) < 0) \neq s$ $\Rightarrow$ Nullstelle im Intervall $[a,x]$ und f ist nämlich bzg $[a,x]$ ebenfalls steigend

die Überlegung paßt sich mit dem neuen Intervall wiederholen

Analog für f fallend, d.h. $f(a) \geq 0 \geq f(b)$; in diesem Fall bezeichne s false:

Fall 1': $f(x) < 0 \Leftrightarrow (f(x) < 0) \neq s$ $\Rightarrow$ Nullstelle im Intervall $[a,x]$ und f ist fallend bzg. $[a,x]$

Fall 2': $f(x) \geq 0 \Leftrightarrow (f(x) < 0) = s$ $\Rightarrow$ Nullstelle im Intervall $[x,b]$ und f ist fallend bzg. $[x,b]$

die Überlegung paßt sich mit dem neuen Intervall wiederholen

Es gilt sich:

$(f(x) < 0) = s \Rightarrow$ Nullstelle im Intervall $[x, b]$

$(f(x) < 0) \neq s \Rightarrow$ Nullstelle im Intervall $[a, x]$

Wir können also folgende <u>Invarianten</u> beobachten, wobei $a < b$ die jeweils beobachteten Intervallgrenzen sind:

$J(a,b) :\Leftrightarrow \underbrace{f \text{ steigend}}_{f(a) \leq 0 \leq f(b)} \wedge f$ hat genau eine Nullstelle im Intervall $[a,b]$

$J^*(a,b) :\Leftrightarrow \underbrace{f \text{ fallend}}_{f(a) \geq 0 \geq f(b)} \wedge f$ hat genau eine Nullstelle im Intervall $[a,b]$

offensichtlich gilt für $x := (a+b)/2.0$:

(* $J(a,b)$ *) IF $(f(x) < 0) = s$ THEN $a := x$ ELSE $b := x$ END (* $J(a,b)$ *)

(* $J^*(a,b)$ *) IF $(f(x) < 0) = s$ THEN $a := x$ ELSE $b := x$ END (* $J^*(a,b)$ *)

Als nächstes müssen wir untersuchen, wie man die Fälle J (f steigend) und J^* (f fallend) unterscheiden soll. Dafür brauchen wir $J(a,b)$ bzw. $J^*(a,b)$ nur für die <u>Anfangswerte</u> von a und b zu untersuchen, da das ja Invarianten sind:

$$J(a,b) \iff \left[f(a) < 0 \leq f(b) \;\vee\; f(a) = 0 < f(b) \right] \wedge$$ f hat genau eine Nullstelle in $[a,b]$

$$J^*(a,b) \iff \left[\underbrace{f(a) > 0 \geq f(b)} \;\vee\; \underbrace{f(a) = 0 > f(b)} \right] \wedge$$ f hat genau eine Nullstelle in $[a,b]$

für $f(a) \neq 0$ sind hierdurch die Fälle zu unterscheiden:

$s := f(a) < 0$

Spezialfall, durch vorgezogene Abfrage:

IF $f(a) = 0$ THEN null := a ELSE ... END

abfangen

Schließlich müssen wir festlegen, wann die Wiederholung abgebrochen werden soll:

Abbruch falls das Intervall $[a,b]$, in dem die Nullstelle liegt hinreichend klein geworden ist, etwa $b - a < \varepsilon$.

Folgendes Programm ist im wesentlichen übertragen aus Jensen/Wirth, S. 79.

```
MODULE versuch;

FROM InOut IMPORT WriteLn, WriteString;
FROM RealInOut IMPORT ReadReal, WriteReal;

CONST epsilon = 1.0E-8;

PROCEDURE Sinus(x: REAL): REAL; ... END Sinus;

PROCEDURE Cosinus(x: REAL): REAL; ... END Cosinus;

TYPE SteligeFunktion = PROCEDURE (REAL): REAL;

PROCEDURE null (F: SteligeFunktion; a,b: REAL): REAL;
   (* a<b ; es gelten Eigenschaften 1) und 2) ; f stetig *)
   VAR x,z: REAL;
       s: BOOLEAN;
   BEGIN IF f(a)=0.0 THEN RETURN a
         ELSE s := f(a) < 0.0;
              REPEAT x := (a+b)/2.0;
                     z := f(x);
                     IF (z<0.0)=s THEN a := x
                     ELSE              b := x
                     END
              UNTIL b-a < epsilon;
              RETURN x
         END
   END null;
VAR x,y: REAL;
BEGIN
   ReadReal(x); ReadReal(y); WriteReal(x), WriteReal(y);
   WriteReal( null (Sinus, x, y)); WriteLn;
   ReadReal(x); ReadReal(y); WriteReal(x), WriteReal(y);
   WriteReal( null (Cosinus, x, y)); WriteLn
END versuch.
```

Hoffentlich, siehe S. 21.8

Hier kommt der formale Parameter F im Rumpf vor

z.B. Daten
-1.0 +1.0
+1.0 +2.0

Der Test f(a) = 0 ist problematisch, da wir die exakte Gleichheit zweier Zahlen in Gleitkomma-Darstellung testen. Falls f selbst nicht exakt berechnet wird, können hier Fehler auftreten. Eine sichere Variante der MODULA-2 - Funktionsprozedur ergibt sich z.B. wie folgt:

```
PROCEDURE null ( f: stetige Funktion; a,b : REAL ) : REAL;
 (* a<b , es gelten Eigenschaften 1) und 2), f stetig *)
VAR x,z : REAL; s : BOOLEAN;
BEGIN
  IF f(a) * f(b) < 0.0  (* f(a)<0<f(b) oder f(a)>0>f(b) *)
  THEN  s := f(a) < 0.0;
        REPEAT x := (a+b) / 2.0;
               z := f(x);
               IF (z<0.0) = s THEN a:=x ELSE b:=x END
        UNTIL b-a < epsilon;
        RETURN x
  ELSE  (* Fehlerfall *)
        WriteLn; WriteString (" möglicherweise fehlerhaft wegen Nullstelle am Rand ");
        IF ABS(f(a)) < ABS(f(b))
        THEN RETURN a ELSE RETURN b END
  END
END null;
```

Standardfunktion zur Berechnung des Betrages (Absolutwert)

Verwendet man Parameter mit Prozedurtyp, so werden die in §19, "Lokalteil", behandelten Regeln über die Möglichkeiten von Prozeduraufrufen und damit zusammenhängende Fragestellungen noch komplizierter. MODULA-2 verlangt deshalb eine Vereinbarung, daß

Werbezweisungen und Parameterübergaben nur für unmittelbar im Modul vereinbarte Prozeduren erlaubt sind (also auch nicht für Standardprozeduren).

Das Problem soll anhand eines Beispiels aufgezeigt werden:

```
MODULE beispiel;
FROM InOut IMPORT WriteLn;
TYPE Arbeiter = PROCEDURE;

    PROCEDURE a1 (u: Arbeiter);
    BEGIN
        u
    END a;

    PROCEDURE b2;
        PROCEDURE c3;
        BEGIN
            WriteLn ("hallo")
        END c;

    BEGIN
        a (c)
    END b;

BEGIN
    b
END beispiel.
```

BEISPIEL

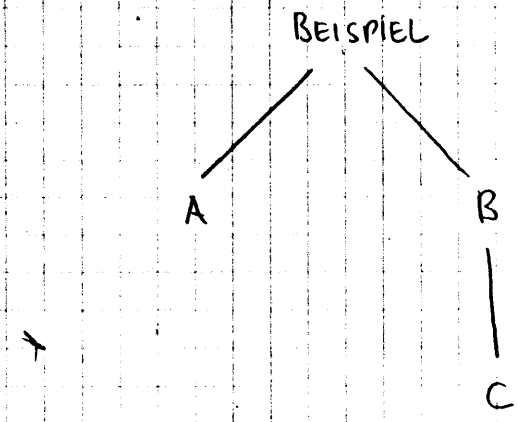
A

B

C

21.10

Vereinbarungsblöcke:



Gültigkeitsbereiche: [a]$_1$ hat Gültigkeitsbereich BEISPIEL.
[b]$_2$ " " BEISPIEL
[c]$_3$ " " B

insbesondere gilt damit: $A \neq B$, d.h.

A ist nicht im Gültigkeitsbereich der Bezeichner-Vereinbarung für C enthalten

[c]$_3$

B

aber:

A enthält im Rumpf eine Anweisung, die "einen Aufruf von C bewirkt".

dann die Ausführung des Programms bewirkt folgende Kette von Prozeduraufrufen:

mit aktuellem Parameter c für a

beispiel → b → a → c

28. Module

Im Zusammenhang mit Prozeduren haben wir für Variable zwei Konzepte kennengelernt:

- Gültigkeitsbereich der Bezeichner-Vereinbarung (statisch)
- Lebenszeit (dynamisch)

Diese Konzepte sind eng gekoppelt mit der Vereinbarungsstruktur der Prozedurtexte.

Das Modul-Konzept erlaubt nun diese enge Kopplung zu durchbrechen, insbesondere um

- weitergehende und leichter wählbare Abschirmungen
- bessere Zerlegungen in unabhängigere Teile
- Trennung der Schnittstellen-Beschreibung von der Vereinbarung

zu ermöglichen.

Zunächst ein Beispiel (aus R. Gleaves, p. 5)

```
MODULE Zufallszahlen;
```

leitet Vereinbarung eines (lokalen) Moduls ein

```
IMPORT Anfangswert;
```

innerhalb des Moduls sind neben lokalen Bezeichnern nur ausdrücklich "importierte" Bezeichner gültig

```
EXPORT Zufall;
```

lokale Bezeichner können der Umgebung sichtbar gemacht werden, indem sie "exportiert" werden, genauer zum Export angeboten werden

lokale Vereinbarungen des Moduls Zufallszahl

```
CONST Teiler = 7415;
      Schritt = 25543;
VAR   Kern : INTEGER;
```

lokale Variable, mit der abgeschirmt der zuletzt erzeugte Zufallswert gemerkt wird; einzig möglicher Zugriff zu dieser Variablen über die lokale, aber exportierte Prozedur Zufall

```
PROCEDURE Zufall() : INTEGER;
```

lokale Prozedur, exportiert in Umgebung

```
BEGIN Kern := (Kern + Schritt) MOD Teiler;
      RETURN Kern
END Zufall;
```

Initialisierungs-Block des Moduls Zufallszahlen

```
BEGIN
    Kern := Anfangswert
END Zufallszahlen;
```

diese Initialisierung wird ausgeführt, wenn die erste umschließende Prozedur aufgerufen wird

ein weiteres Beispiel (aus N. Wirth, p. 82/83) zeigt, wie die Trennung der der Schnittstellen-Beschreibung von der Verwirklichung ermöglicht wird:

```
DEFINITION MODULE Buffer;
(* Buffer ist ein FIFO-Speicher *)

    VAR nichtleer, nichtvoll : BOOLEAN;

    PROCEDURE put (x: CARDINAL);
        (* legt den Wert x im Puffer ab *)

    PROCEDURE get (VAR x: CARDINAL);
        (* holt Wert x aus dem Puffer *)
END Buffer.
```

leitet die Schnittstellen-Beschreibung ein; alle im folgenden vereinbarten Bezeichner werden exportiert!

getrennt davon kann nun festgelegt (oder auch später geändert) werden, wie put und get genau arbeiten, z.B.

```
IMPLEMENTATION MODULE Buffer;
    CONST N = 100;
    VAR ein, aus : [0..N-1];
            n : [0..N];
        speicher : ARRAY [0..N-1] OF CARDINAL;

    PROCEDURE put (x: CARDINAL);
    BEGIN IF n<N THEN
            speicher[ein]:=x; ein:=(ein+1) MOD N;
            n:=n+1; nichtvoll := n<N; nichtleer := TRUE
            END
    END put;
```

vollständig abgeschirmte Größen, aber die ausschließlich über die exportierten Prozeduren zugriffen werden kann

```
PROCEDURE get( VAR x: CARDINAL);
BEGIN  IF n>0 THEN
          x:= speicher[aus]; aus:=(aus+1) MOD N;
          n:= n-1; nichtLeer := n>0; nichtVoll := TRUE
       END
END get;

BEGIN   n:=0; ein:=0; aus:=0
        nichtLeer := FALSE; nichtVoll := TRUE
END Puffer.
```

Ein FIFO-Speicher verwaltet Werte nach dem Grundsatz:

first in - first out (übliche Warteschlange)

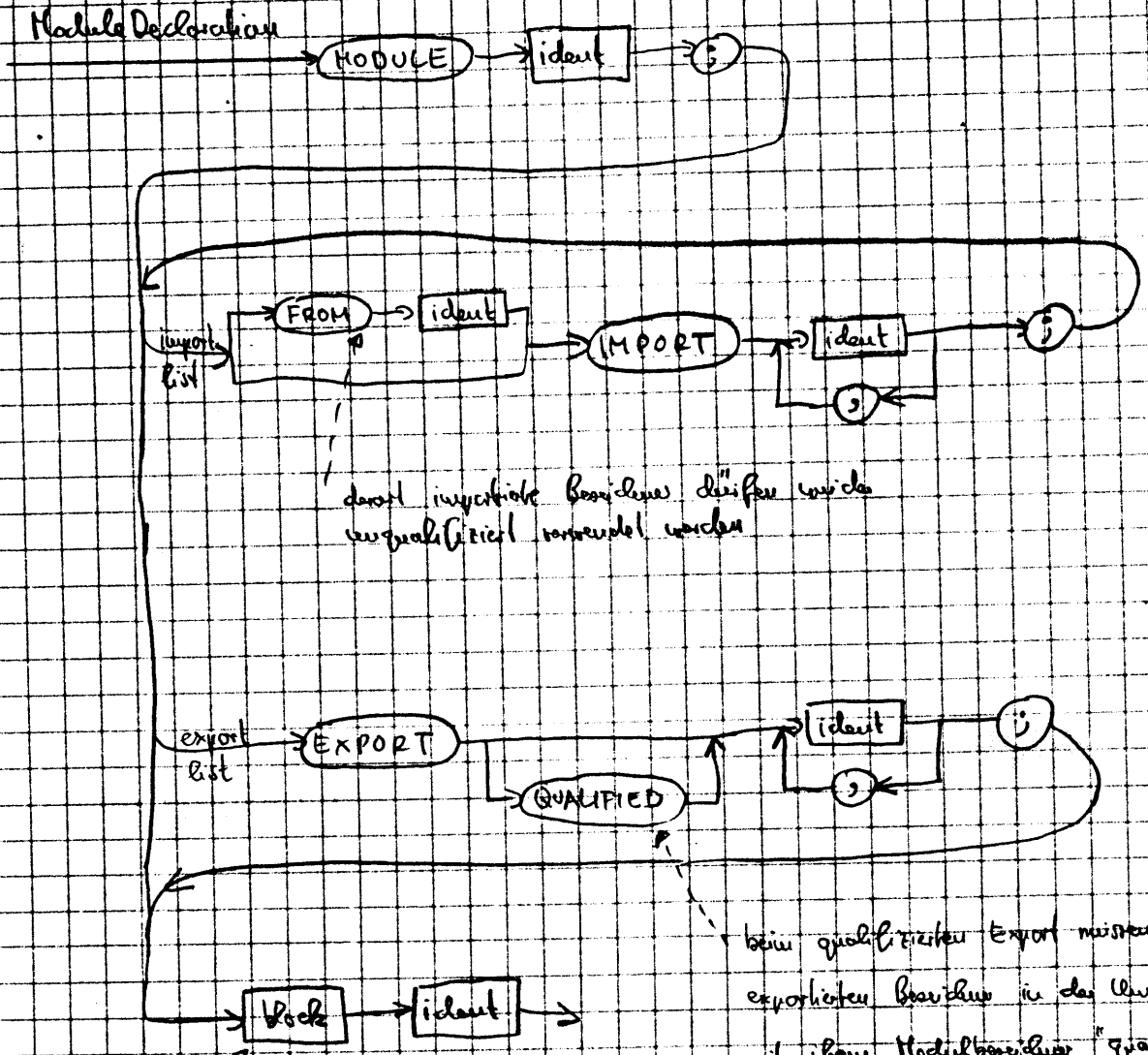


ein bezeichnet erstes freies Feld (Ende der Warteschlange)

aus bezeichnet den Kopf der Schlange

(Anders als am "Bedienerschalter" bewegt sich nicht die "Schlange", sondern der "Bereich aus" kommt zum Kopf!)

Man kann FIFO-Speicher auch anders verwirklichen; dann ändert sich nur der Implementationsteil des Moduls.



damit importierte Bezeichner direkt verwendet werden

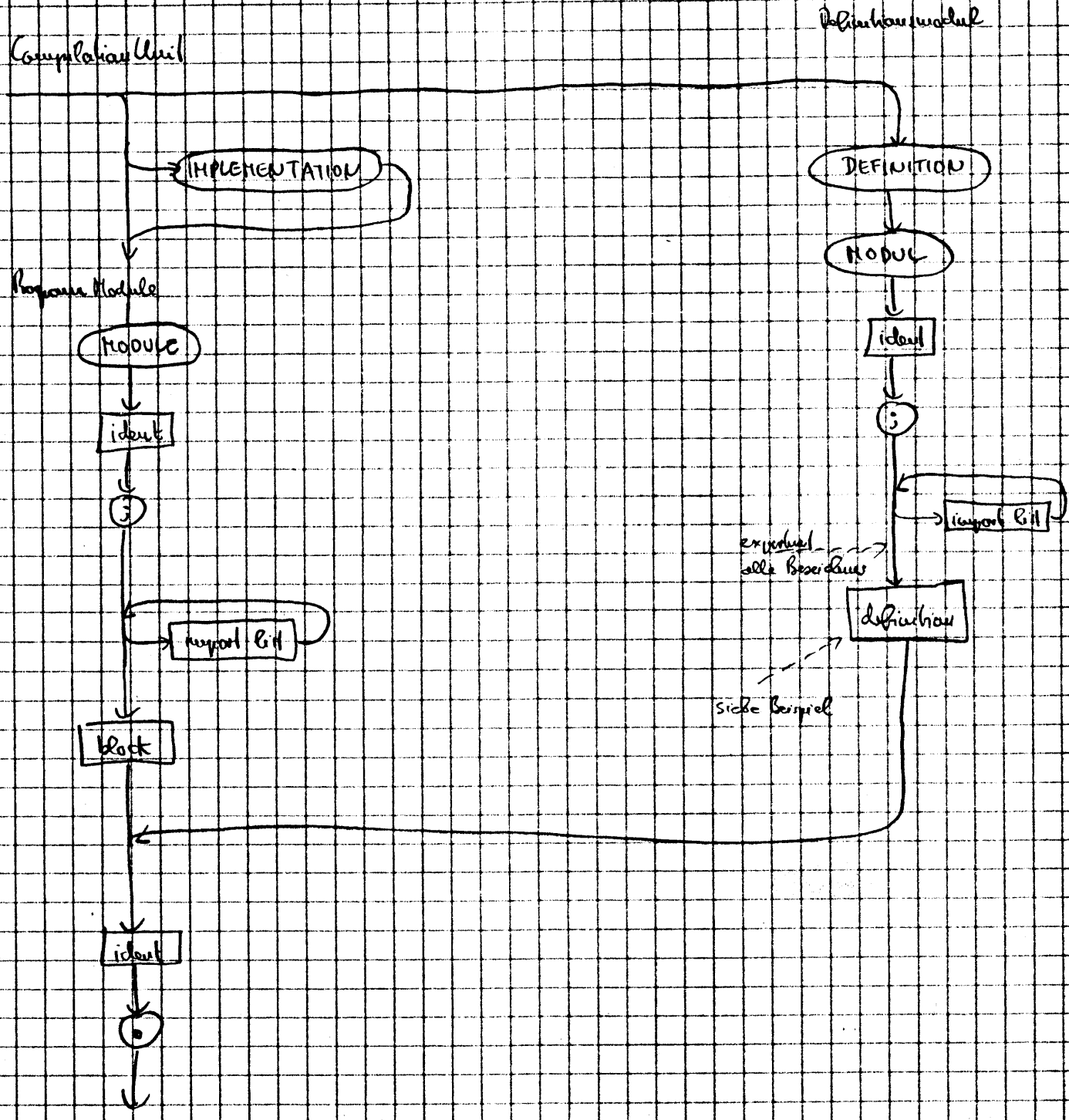
bei qualifiziertem Export müssen die exportierten Bezeichner in der Umgebung mit ihrem Modulbezeichner "qualifiziert" werden, d.h. in der Form

Modulbezeichner . Bezeichner

benutzt werden

Modulbezeichner wird ausgeführt beim Aufruf der exportierenden Prozedur vor deren erster Anweisung

Für große Programme ist es nützlich, deren Teile voneinander unabhängig übersetzen zu können:



Gültigkeitsbereiche und Lebensdauer lokaler Objekte:

Lebensdauer: entsprechend dem Aufruf der engsten umschließenden Prozedur

Gültigkeitsbereich: innerhalb des Moduls

falls exportiert auch außerhalb

Nichtlokale Objekte sind nur sichtbar, falls sie importiert wurden.

Typische Verwendungen von Moduln

1) Modul stellt eine Menge von Prozeduren zur Verfügung
(z.B. zur Umwandlung einer Objektdarstellung in eine andere)

2) Modul schirmt eine Menge von Daten ab, zu denen der Zugriff ausschließlich über exportierte Prozeduren möglich ist.
Bsp: Zufallszahlen

3) Modul stellt Datentyp zur Verfügung, wobei die Einzelheiten der Implementierung abgeschirmt werden.
Bsp: Buffer

Einige grundsätzliche Regeln:

- Modul-Geschlossenheit: ein Modul beschreibt eine (möglichst) abgeschlossene Teilaufgabe

- Einfache Schnittstelle: möglichst wenig importieren / exportieren
noch Möglichkeit: keine Variablen exportieren, importierte Variablen nur "lesen"

(Weitere Vertiefung ⟹ Programmierpraktikum)

22. Aufwandsanalyse am Beispiel des Suchens bei sequentieller Speicherung

Aufgabe: In einem Feld a, in dem ganzzahlige Werte "aufsteigend sortiert" beschrieben werden, soll die Stelle eines vorgegebenen Wertes (such) gesucht werden.

Erfassen und Verdeutlichen:

Argumente:	Feld	a: ARRAY index OF INTEGER	
	untere Grenze	u: index	
	obere Grenze	o: index	
	Suchargument	such: INTEGER	
Ergebnisse:	Mitteilung, ob Suche erfolgreich	gefunden: BOOLEAN	
	Stelle des Suchargumentes	stelle: index	

Zustandsüberführung:

$$(a \mid u \mid o \mid such) \longmapsto (a \mid u \mid o \mid such \parallel gefunden \mid stelle)$$

mit

u Index für a
o Index für a

a sortiert in u..o,

d.h. $a[u] \leq a[succ(u)] \leq .. \leq a[o]$

mit

gefunden $\Leftrightarrow$ ex i mit $u \leq i \leq o$ und $a[i] = such$

$$stelle = \begin{cases} \text{ein obiges } i & \text{falls gefunden} \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Suchverfahren:

1. Binäres Suchen, das dem Wahlspruch "teile und siege" folgt.

Bsp

Argument:

a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	such	...
1	5	8	8	10	13	20	13	

berechne die mittlere Stelle mitte (hier speziell: $\frac{1}{2}(1+7) = 4$)

mitte

teste ob: $a[4] < such$: neuer Suchbereich $a[5], .., a[7]$ (mitte+1 bis o)

$a[4] = such$: Suchargument gefunden

$a[4] > such$: neuer Suchbereich $a[1], .., a[3]$ (u bis mitte−1)

(hier speziell: $a[4] < such$, $8 < 13$)

dann wiederholen mit neuem Suchbereich.

Abbruchbedingungen:

erfolgreiche Suche: $a[mitte] = such$

erfolglose Suche: neuer Suchbereich leer, d.h. $o < u$

Suche muß stets abbrechen, weil in jeder Wiederholung der Suchbereich echt kleiner wird.

```
:
CONST  unten = 1;
       oben = 31;
TYPE   index = [unten .. oben];
       feld = ARRAY index OF INTEGER;
:
```

nur aus Effizienzgründen, um das Kopieren des Feldes zu vermeiden

```
PROCEDURE binarsuch ( VAR a : feld ; u: index; o: index ;      such: INTEGER;
```

Argumente

```
                      VAR gefunden : BOOLEAN;
                      VAR stelle   : index                    );
```

Ergebnisse

```
  VAR mitte : index;
  BEGIN
    IF u <= o
    THEN    mitte := (u+o) DIV 2;
            IF a[mitte] = such
               THEN    stelle := mitte ; gefunden := TRUE   (* Suche erfolgreich *)
            ELSIF a[mitte] < such
               THEN binarsuch ( a, mitte+1, o, such, gefunden, stelle )
               ELSE binarsuch ( a, u, mitte-1, such, gefunden, stelle )
            END
    ELSE (* o < u , d.h. Suche erfolglos *)
       WriteLn; WriteString( "- Suche erfolglos" ); gefunden := FALSE
    END

  END binarsuch.
```

eine entsprechende nichtrekursive Prozedur:

```
PROCEDURE binarsuch ( VAR a : feld ; u : index ; o : index ; such : INTEGER ;
                      VAR gefunden : BOOLEAN ;
                      VAR stelle : index )

  VAR mitte : index ;

  BEGIN gefunden := FALSE ;

    WHILE (u <= o) AND NOT gefunden DO
      mitte := (u + o) DIV 2 ;
      IF a[mitte] = such
        THEN stelle := mitte ; gefunden := true
      ELSIF a[mitte] < such
        THEN u := mitte + 1
      ELSE   o := mitte - 1
      END
    END ;
    IF NOT gefunden THEN WriteLn ; WriteString (" Suche erfolglos ") END

END binarsuch ;
```

(nur aus Effizienzgründen)

Diesen Suchalgorithmus wollen wir nun genauer untersuchen und dabei insbesondere folgende (beim systematischen Programmieren stets auftretende) Fragen klären:

- auf welchen Voraussetzungen beruht er?
- welchen allgemeinen Ideen folgt er?
- welchen Berechnungsaufwand erfordert er?

Dazu wollen wir diesen Algorithmus nicht isoliert betrachten, sondern gleich eine ganze Klasse von Algorithmen.

Suchschema für Suchen bei sequentieller Speicherung:

Argumente:

1) sequentiell gespeicherte Werte,
   beschrieben durch ein Feld a: ARRAY index OF werte,
   wobei index als skalarer Datentyp linear geordnet ist und sequentiell durchlaufen werden kann (vermöge INC)

2) Suchargument,
   beschrieben durch such: werte

Ergebnisse

1) Mitteilung, ob Suche erfolgreich,
   gefunden: BOOLEAN

2) Stelle des Sucharguments:,
   stelle: index

Verfahren

gefunden := FALSE ;

WHILE "weitere Suche notwendig" and NOT gefunden DO
"Suche noch nicht erfolgreich"

(* Auswahl *) wähle eine (neue) Stelle i aus ;

(* Vergleich *) IF a[i] = such THEN gefunden := TRUE; stelle := i

[] a[i] ≠ such THEN "bestimme neuen Suchbereich" END

END

Aus diesem Schema wird ein Suchalgorithmus, wenn man folgende Punkte genau festlegt:

- "weitere Suche notwendig"
- "wähle eine (neue) Stelle i aus"
- ["bestimme neuen Suchbereich"] muß nicht immer explizit erfolgen!

Diese Festlegungen stellen zugleich Anforderungen an

- die Zugriffsmöglichkeiten im Feld a
- Eigenschaften der Wertemenge werte
- Art der Speicherung der Werte im Feld a

Für jeden konkreten Suchalgorithmus $\mathfrak{A}$ kann man den <u>Berechnungsaufwand</u> bestimmen, den wir messen wollen durch

die Anzahl der benötigten Stellenauswahlen
(gleich Anzahl der Durchläufe der Wiederholungsanweisung)

Genauer:

$\mathfrak{A}$ Suchalgorithmus
werte zugrunde liegende Wertemenge
$werte_0$ tatsächlich vorhandene Werte im Feld a
m Anzahl der Elemente von index
("Länge des Feldes")

such Suchargument

$c_m(\mathfrak{A}, such)$:= bei Durchführung von $\mathfrak{A}$ benötigter Aufwand zur Suche von such im Feld a

$M_m^+(\mathfrak{A})$:= bei <u>erfolgreicher</u> Durchführung von $\mathfrak{A}$ <u>maximal</u> benötigter Aufwand

= Maximum $\{ c_m(\mathfrak{A}, such) \mid such \in werte_0 \}$

$M_m^-(\mathfrak{A})$:= bei <u>erfolgloser</u> Durchführung von $\mathfrak{A}$ <u>maximal</u> benötigter Aufwand

= Maximum $\{ c_m(\mathfrak{A}, such) \mid such \in werte \setminus werte_0 \}$

$M_m(\mathfrak{A})$ = bei Durchführung von $\mathfrak{A}$ maximal benötigter Aufwand

$= \text{Maximum} \{ M_m^+(\alpha), M_m^-(\alpha) \}$

$E_m^+(\alpha)$:= bei erfolgreicher Durchführung von α durchschnittlich benötigter Aufwand

$= \text{Erwartungswert} \{ c_m(\alpha, such) \mid such \in werte_0 \}$

$E_m^-(\alpha)$:= bei erfolgloser Durchführung von α durchschnittlich benötigter Aufwand

$= \text{Erwartungswert} \{ c_m(\alpha, such) \mid such \in werte - werte_0 \}$

Der bei Durchführung von α schlechthin - erfolgreich oder erfolglos - durchschnittlich benötigte Aufwand $E_m(\alpha)$ läßt sich als gewichtetes Mittel von $E_m^+(\alpha)$ und $E_m^-(\alpha)$ errechnen:

$$E_m(\alpha) = p^+ \cdot E_m^+(\alpha) + p^- \cdot E_m^-(\alpha) \quad , \text{wobei}$$

p^+ := Wahrscheinlichkeit der Suche nach einem Wert $such \in werte_0$

$p^- := 1 - p^+ =$ Wahrscheinlichkeit der Suche nach einem Wert $such \in werte - werte_0$

Algorithmus A_1 : sequentielles Suchen

A_1 durchsucht das Feld a sequentiell entsprechend der Ordnung auf index, wobei unten bzw. oben das kleinste bzw. größte Element von index sei.

zu "wähle (neue) Stelle i aus" :

$$i := \begin{cases} unten & \text{im ersten Durchlauf} \\ INC^i & \text{sonst} \end{cases}$$

zu "weitere Suche notwendig" : $i \leq oben$

Anforderungen : index linear geordnet, INC definiert

Aufwand :

$$c_m(A_1, such) = \begin{cases} i & \text{falls } a[INC^{i-1}(unten)] = such \\ m & \text{falls } such \in werte \setminus werte_0 \end{cases}$$

$$M_m(A_1) = m$$

Um den durchschnittlichen Aufwand zu ermitteln, muß man statistische Annahmen machen; z.B.

$werte_0$ enthält m Elemente und für $such \in werte_0$ sei $p(such)$ = Wahrscheinlichkeit, daß nach such gesucht wird,

$$\sum_{such \in werte_0} p(such) = 1$$

$$E_m^+(\mathfrak{A}_1) = \sum_{\text{such} \in \text{werte}_0} p(\text{such}) \cdot c_m(\mathfrak{A}_1, \text{such})$$

Bei <u>Gleichverteilung der Suchwahrscheinlichkeiten</u>, d.h. falls $p(\text{such}) = \frac{1}{m}$ für alle such $\in$ werte$_0$, folgt

$$E_m^+(\mathfrak{A}_1) = \sum_{\text{such} \in \text{werte}_0} p(\text{such}) \cdot c_m(\mathfrak{A}_1, \text{such})$$

jedes $i = 1, \dots, m$ kommt einmal vor!

$$= \sum_{i=1}^{m} \frac{1}{m} \, i$$

$$= \frac{1}{m} \left[\frac{m}{2}(1+m) \right] = \frac{m+1}{2} \approx \frac{m}{2}$$

Falls <u>keine Gleichverteilung</u> vorliegt, kann man versuchen, $E_m^+(\mathfrak{A}_1)$ <u>durch die Art der Speicherung zu minimisieren</u>.

Die Werte aus werte$_0$ seien so abgespeichert, daß für $u \in$ werte$_0$ gilt:

$$\underset{\substack{\uparrow \\ \text{location}}}{\text{loc}}(u) := \text{dasjenige } i \text{ mit } a[\text{INC}^{i-1}(\text{unten})] = u$$

$E_m^+(\mathfrak{A}_1)$ wird <u>minimal</u>, wenn die Werte gemäß absteigender Wahrscheinlichkeit gespeichert werden, d.h. wenn für die <u>Speicherfunktion</u> loc und alle $u, v \in$ werte$_0$ gilt:

$$(*) \qquad p(u) > p(v) \Rightarrow \text{loc}(u) < \text{loc}(v)$$

Begründung

Wir zeigen, daß sich die Speicherfunktion loc, für die (*) nicht gilt, "verbessern" läßt:

sei $p(u) > p(v)$, aber $loc(u) > loc(v)$

u, v leisten folgenden Beitrag zu $E_m^+(\alpha_1)$:

$$b := p(u) \cdot loc(u) + p(v) \cdot loc(v)$$

Vertauscht man u und v, so liefern sie den Beitrag

$$\overline{b} := p(u) \cdot loc(v) + p(v) \cdot loc(u)$$

Es folgt:

$$b - \overline{b} = p(u) \cdot [loc(u) - loc(v)] + p(v) \cdot [loc(v) - loc(u)]$$

$$= \underbrace{[p(u) - p(v)]}_{> 0 \text{ wegen } p(u) > p(v)} \cdot \underbrace{[loc(u) - loc(v)]}_{> 0 \text{ wegen } loc(u) > loc(v)}$$

$$> 0.$$

Rest: beim Vertauschen von u und v "verbessert man den Beitrag von u und v", während offensichtlich die Beiträge aller anderen Werte unverändert bleiben, d.h.

die Vertauschung von u und v "verbessert" $E_m^+(\alpha_1)$.

Algorithmus A_2: sequentielles Suchen bei sortierten Werten

Anforderungen: 1) index linear geordnet, INC definiert

2) auf der Wertemenge ist eine lineare Ordnung $\preceq$ definiert, $\preceq$ ist "entscheidbar"

3) Werte sind entsprechend der Ordnung gespeichert:

für alle $u, v \in werte_0$ gilt

$$u \preceq v \iff loc(u) \leq loc(v)$$

zu "wähle (neue) Stelle i aus": wie oben, d.h.

$$i := \begin{cases} unten & \text{im ersten Durchlauf} \\ INC(i) & \text{sonst} \end{cases}$$

zu "weitere Suche notwendig":

$$i < oben \quad \text{und} \quad \underbrace{a[i] < such}$$

d.h. Suche kann als erfolglos abgebrochen werden, sobald $a[i] > such$: wegen der sortierten Speicherung kann such nicht mehr "hinter der Stelle i" vorkommen

Aufwand:

$$c_m(A_2, such) = \begin{cases} i & \text{falls } such \in werte_0 \text{ mit } loc(such) = i \\ n & \text{falls } such \in werte - werte_0 \text{ und } n = Minimum\{j \mid a[j] > such\} \text{ existiert} \\ u & \text{sonst} \end{cases}$$

bei erfolgreicher Suche: keine Verbesserung gegenüber $\mathfrak{A}_1$

bei erfolgloser Suche:

nimmt man an, daß für such $\in$ werte - werte$_0$ such mit Wahrscheinlichkeit $\frac{1}{m+1}$ kleiner als a[unten], bzw. zwischen a[i] und a[succ(i)] liegt, bzw. größer als a[oben] ist,

so ergibt sich

$$E_m^-(\mathfrak{A}_2) = \sum_{i=1}^{m} \frac{1}{m+1} \cdot \underset{\substack{\uparrow \\ \text{Minimum } \{j \mid a[j] > \text{such}\}}}{i} + \frac{1}{m+1} \cdot m$$

$$= \frac{1}{m+1} \left[\frac{m}{2}(m+1) \right] + \frac{m}{m+1}$$

$$= \frac{m}{2} + \frac{m}{m+1} \approx \frac{m}{2} + 1$$

Die Verbesserung von $\mathfrak{A}_2$ bzgl. $\mathfrak{A}_1$ ergibt sich daraus, daß man bei erfolgloser Suche den sequentiellen Durchlauf im allgemeinen schon weit vor dem letzten Feldelement abbrechen kann.

Eine weitere Verbesserung ergibt sich, wenn man den sequentiellen Durchlauf im allgemeinen erst weit hinter dem ersten Feldelement beginnt:

Algorithmus $\mathfrak{A}_3$: index-sequentielles Suchen

Anforderungen: 1) index linear geordnet;
INC auf index definiert;

neue Anforderung $\left\{ \begin{array}{l} \text{index enthält (direkt) adressierbare Stellen} \\ \underbrace{j_1 < j_2 < \ldots < j_k}_{\text{"Index"}} = \text{oben} \end{array} \right.$, wobei zwischen j_i und j_{i+1} jeweils noch weitere Stellen liegen

2) auf der Wertemenge ist eine lineare Ordnung $\preceq$ definiert, $\preceq$ ist "entscheidbar"

3) Werte sind entsprechend der Ordnung gespeichert

zu "wähle (neue) Stelle i aus":

Idee: suche zunächst sequentiell "in großen Schritten" im Index;

falls dort Suche erfolglos wegen

$$\left(a[j_{\ell-1}]\right) < \text{such} < a[j_\ell] ,$$

so suche sequentiell am Abschnitt

$$[INC(j_{\ell-1}) \,..\, DEC(j_\ell)]$$

Annahmen: $INC(j_0) := \text{unten}$

$\text{unten} < j_1$

$INC(j_\ell) < j_{\ell+1}$

$$i := \begin{cases} j_\ell & \text{im } \ell\text{-ten Durchlauf der Indexsuche} \\ INC(j_{\ell-1}) & \text{falls } such < a[j_\ell] \\ & \{ \text{Indexsuche erfolglos beendet, Übergang zum sequentiellen Suchen im ASchnitt.} \\ INC(i) & \text{beim sequentiellen Suchen im ASchnitt.} \end{cases}$$

zu "weitere Suche notwendig":

während Suche im Index $[j_1 < \ldots < j_b]$:

$\ell < k$ und $a[j_\ell] < such$: weiter mit Indexsuche

$such < a[j_\ell]$: Übergang zur ASchnittsuche

während Suche im ASchnitt $[INC(j_{\ell-1}) .. DEC(j_\ell)]$:

$i < pred(j_\ell)$ und $a[i] < such$

Aufwand

a) bei erfolgreicher Suche, sucht befindet sich im ASchnitt $[INC(j_{\ell-1}) .. j_\ell]$:

$$c_m(\alpha_3, such) = \begin{cases} \ell & \text{falls } a[j_\ell] = such \\ \ell+1 & \text{falls } a[INC^r(j_{\ell-1})] = such \end{cases}$$

Zur Vereinfachung seien die Indexstellen gleichmäßig verteilt, etwa $j_i := i \cdot a$ für $i = 1, \dots, k$, wobei unten $= 1$ und oben $= j_k = k \cdot a$ sei.

$$M_m^+(\alpha_3) = k + (a-1)$$

↑ den gesamten letzten Abschnitt bezeichnen

↑ alle Indexstellen bezeichnen

Bei Gleichverteilung der Suchwahrscheinlichkeiten ergibt sich

$$E_m^+(\alpha_3) = \frac{1}{m} \sum_{\ell=1}^{k} \left[\ell + (\ell+1) + \dots + (\ell + (a-1)) \right]$$

↑ nach ℓ-ter Indexstelle Übergang zum sequentiellen Suchen im zugehörigen Abschnitt

$$= \frac{1}{m} \sum_{\ell=1}^{k} \frac{a}{2} \underbrace{\left[\ell + (\ell + (a-1)) \right]}_{2\ell + (a-1)}$$

$$= \frac{1}{m} \sum_{\ell=1}^{k} \left[a \cdot \ell + \frac{a(a-1)}{2} \right] = \frac{1}{m} \left[a \cdot \sum_{\ell=1}^{k} \ell + k \cdot \frac{a(a-1)}{2} \right]$$

$$= \frac{1}{m} \left[\overbrace{\frac{a \cdot k \cdot (k+1)}{2}}^{=m} + \overbrace{\frac{k \cdot a \, (a-1)}{2}}^{m} \right]$$

$$= \underbrace{\frac{k+1}{2}}_{\text{durchschnittliche Suchzeit im Index}} + \underbrace{\frac{a-1}{2}}_{\text{durchschnittliche Suchzeit in einem Abschnitt}}$$

b) bei erfolgloser Suche:

$$c_m(\alpha_3, \text{such}) = \begin{cases} k & \text{falls } a[j_k^{\text{oben}}] \prec \text{such} \\ 2 & \text{falls such} \prec a[\text{unten}] \\ \ell+r & \text{falls } a[\text{succ}^{r-1}(j_{\ell-1})] \prec \text{such} \\ & \quad \prec a[\text{succ}^{r}(j_{\ell-1})] \\ & \text{für } \ell = 1,\dots,k;\ r = 1,\dots,a-1 \text{ mit} \\ & (\ell,r) \neq (1,1) \\ \ell+(a-1) & \text{sonst, d.h.} \\ & a[\underbrace{\text{INC}^{a-1}(j_{\ell-1})}_{\text{DEC}(j_\ell)}] \prec \text{such} \prec a[j_\ell] \end{cases}$$

$\Rightarrow$ Folgt:

$$M_m^-(\alpha_3) = k+(a-1)$$

Bei Gleichverteilung der Suchwahrscheinlichkeiten ergibt sich

$$E_m^-(\alpha_3) = \frac{1}{m+1}\Bigg[\, k + \underbrace{\sum_{\ell=1}^{k}\sum_{r=1}^{a-1}(\ell+r)}_{(a-1)\cdot\ell+\frac{a-1}{2}\cdot a} + \underbrace{\sum_{\ell=1}^{k}\ell+(a-1)}_{k\cdot\left[\frac{k+1}{2}+(a-1)\right]} \Bigg]$$

$$\underbrace{k\cdot(a-1)\left[\frac{k+1}{2}+\frac{a}{2}\right]}_{(k\cdot a-k)}$$

$$\underbrace{\qquad\qquad}_{=m}$$

$$= \frac{1}{m+1}\left[k + m\left[\frac{k+1}{2} + \frac{a}{2}\right] - k\left[\frac{k+1}{2} + \frac{a}{2}\right] + k\cdot\frac{k+1}{2} + \underbrace{k\cdot a}_{=m} - k \right]$$

$$= \frac{1}{m+1}\left[m\left[\frac{k+1}{2} + \frac{a+1}{2}\right]\right]$$

$$\approx \frac{k+1}{2} + \frac{a+1}{2}$$

Schließlich wollen wir untersuchen, <u>für welche Indexgröße k der Suchaufwand</u> (für die erfolgreiche Suche), d.h.

$$E_m^+(\alpha_s) = \frac{k+1}{2} + \frac{a-1}{2} \overset{a=\frac{m}{k}}{=} \frac{1}{2}\left[k + \frac{m}{k}\right]$$

<u>minimal</u> wird.

Statt der diskreten Minimierungsaufgabe behandeln wir die entsprechende kontinuierliche und den Standardmethoden der Analysis, d.h. gesucht ist $k \in \mathbb{R}$ mit $\frac{1}{2}\left[k + \frac{m}{k}\right] \overset{!}{=}$ Minimum:

$$0 \overset{!}{=} \frac{d}{dk}\left(\frac{1}{2}\left[k + \frac{m}{k}\right]\right) = \frac{1}{2}\left[1 - \frac{m}{k^2}\right]$$

$$\Rightarrow \quad k = \sqrt{m}$$

Man kann dann beweisen, daß für $\sqrt{m}$ tatsächlich ein Minimum angenommen wird.

Man kann nun erwarten, daß man die Indexgröße k am günstigsten möglichst nahe bei $\sqrt{m}$ wählt.

Ist m Quadratzahl, so folgt für $k = \sqrt{m}$

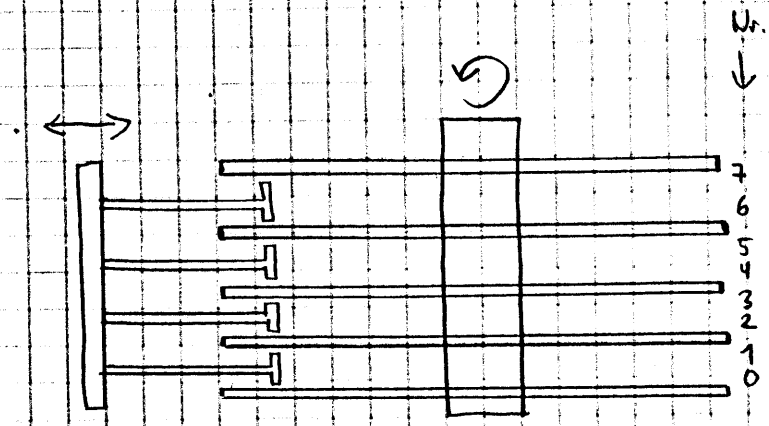
$$E_m^+(\alpha_3) = \frac{1}{2}\left[k + \frac{m}{k}\right] = \sqrt{m}$$

$$E_m^-(\alpha_3) \approx \frac{1}{2}\left[k + \frac{m}{k} + 2\right] = \sqrt{m} + 1$$

<u>Bemerkung</u>: Index-sequentielles Suchen erfordert außer der Sortierung der Werte auch die Möglichkeit, die Indexstellen <u>direkt zu adressieren</u>.

Die verlangte Adressierungsart entspricht in etwa der <u>halb-direkten</u> des Magnetplatten-Speichers, bei dem nur die Spursektoren adressierbar sind. Allerdings wird man dann von der rein sequentiellen Speicherung abgehen. Stattdessen hält man im allgemeinen den Index getrennt im schnellen Hauptspeicher, und nur die einzelnen Bereiche werden jeweils sequentiell auf der Magnetplatte abgelegt.
Zu jedem Indexeintrag j_ℓ muß nun die Anfangsadresse des zugehörigen sequentiellen Suchbereichs $[INC(j_{\ell-1}) .. DEC(j_\ell)]$, bzw. auch $[INC(j_{\ell-1}) .. j_\ell]$ hinzugefügt werden.
Bei dieser Organisation muß nur <u>maximal ein</u> Spursektor adressiert werden; <u>bei Magnetplatten ist die Anzahl der Spursektor-Adressierungen das wichtigste Maß für den Berechnungsaufwand</u>.

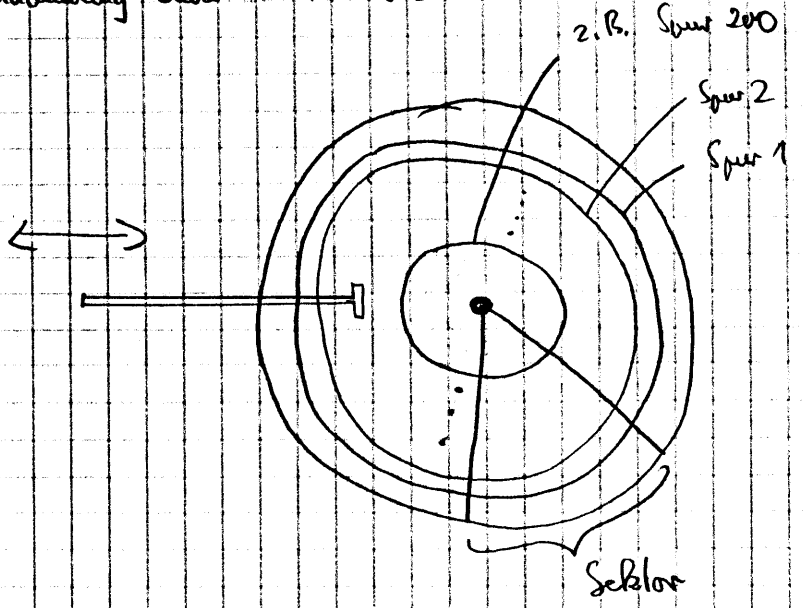
Schematischer Aufbau eines Magnetplatten-Speichers:



(synchron) verschiebbare Lese-Schreibköpfe:

rotierender Plattenstapel, eine Umdrehung ca. 20 msec.

Positionierung etwa 5-50 msec



Adressierung: < Oberflächen-Nr., Spur-Nr., Sektor-Nr. >

Algorithmus A_4: binäres Suchen

siehe Seite 22.1 - 22.4.

Anforderungen:

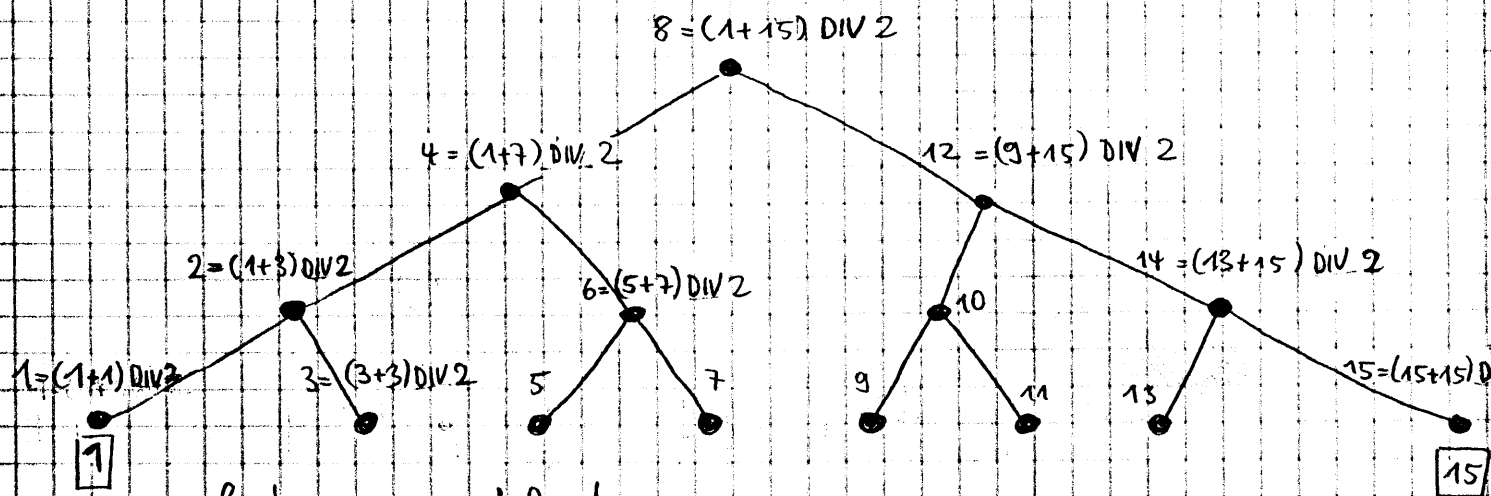
1) index linear geordnet,
   erlaubt Indexrechnungen der Art (u1+o1) DIV 2
   alle Stellen direkt adressierbar

   [MODULA-2 - Felder mit einem Unterbyp von INTEGER als Indextyp sind geeignet]

2) auf Wertemenge ist eine lineare Ordnung $\leq$ definiert,
   $\leq$ ist "entscheidbar"

3) Werte sind entsprechend der Ordnung geordnet

Aufwand: Zur Vereinfachung sei $u = 2^n - 1$
unten = 1, oben = $2^n - 1$, z.B. oben $15 = 2^4 - 1$



Location		Aufwand			
8	: 1	Aufwand	;	$2^{4-1} = 2^3$	teilt 8
4, 12	: 2	Aufwand	;	$2^{4-2} = 2^2$	teilt 4, 12
2, 6, 10, 14	: 3	Aufwand	;	$2^{4-3} = 2^1$	teilt 2, 6, 10, 14
1, 3, 5, 7, 9, 11, 13, 15	: 4	Aufwand	;	$2^{4-4} = 2^0$	teilt 1, 3, 5, 7, 9, 11, 13, 15

a) bei erfolgreicher Suche:

$$c(\alpha_i, \text{such}) = \text{Minimum}\{ l \mid 2^{n-l} \text{ teilt } i \text{ mit } \text{loc}(\text{such}) = i \}$$

$$M_m^+(\alpha_i) = n = \log_2(m+1)$$

Bei Gleichverteilung der Suchwahrscheinlichkeiten ergibt sich:

mit	1	Auswahl	ist	1	Wert erreichbar
mit	2	Auswahlen	sind	2	Werte erreichbar
mit	3	Auswahlen	sind	4	Werte erreichbar
	⋮				
mit	l	Auswahlen	sind	2^{l-1}	Werte erreichbar

Es folgt (mit $n = \log_2(m+1)$):

$$E_m^+(\alpha_i) = \frac{1}{m} \underbrace{\sum_{l=1}^{n} l \cdot 2^{l-1}}_{= (n-1)2^n + 1 \text{ (durch Induktion beweisen!)}}$$

$$= \frac{1}{m} \left[2^n (n-1) + 1 \right]$$

$$= \frac{1}{m} \left[(m+1)(\log_2(m+1) - 1) + 1 \right]$$

$$= \frac{m+1}{m} \left[\log_2(m+1) - 1 \right] + \frac{1}{m}$$

$$\approx \log_2(m+1) - 1$$

Resultat: durchschnittlicher Aufwand nur um etwa 1 kleiner als maximaler Aufwand.

b) erfolglose Suche

$$c_m(A_1, \text{suche}) = n = \log_2(m+1)$$

allgemeiner Trick des binären Suchens

- "teile und siege"
- gewinne Zeit in nur halb so große Aufgabe, dann Wiederholen; beachte die Invariante:

$$a[u1] \leq \text{sich} \leq a[o1]$$

allgemeiner Ansatz zur Bestimmung des Berechnungsaufwand:

Sei $c(m)$ der (zu bestimmende) Berechnungsaufwand für Aufgabe der Größe m.
[hier: Suchen im Feld mit m Elementen]

Gilt dann (grob gerechnet) für die (zu bestimmende) Funktion c:

dann ist eine Funktionalgleichung für c (*)

$$\begin{cases} c(m) \cong 1 + c\left(\frac{m}{2}\right) & \text{für } m \geq 1 \\ \qquad\qquad \uparrow \text{konstant} \\ c(1) = 1 \end{cases}$$

so folgt: $c(m) \sim \log_2(m) + 1$

Rechtfertigung:

. Wir zeigen, daß für $n = 2^l$ gilt: $\log_2(n)+1$ erfüllt (*)

Induktion über l:

$l = 0$ | d.h. $n = 2^0 = 1$:

$$\log_2(1)+1 = 0+1 = 1 \quad \text{erfüllt (*)}$$

$l+1$ |

$$\log_2(2^{l+1})+1 = (l+1)+1$$
$$= 1 + \left[\log_2(2^l)+1\right]$$

Algorithmus	Anforderungen an		(ungefährer) Aufwand = # Anweisungen				Bemerkung
	Zugriffsmöglichkeiten	Werteumfang und Speicherung	erfolgreich		erfolglos		
			max.	durchschn.	max	durchschn.	
$\mathcal{A}_1$: sequentielles Suchen	vermöge INC	— = entscheidbar	n	$\frac{n}{2}$ [1]	n	n	[1] bei Berücksichtigung der W. reflexibar
$\mathcal{A}_2$: sequentielles Suchen bei sortierten Daten	vermöge INC	sortiert $=, \leq$ entscheidbar	n	$\frac{n}{2}$	n	$\frac{n}{2}$	
$\mathcal{A}_3$: index-sequentielles Suchen	- vermöge INC in den Abschnitten - direkt für den Index (falls direkt)	sortiert $=, \leq$ entscheidbar	$2 \cdot \sqrt{n}$	$\sqrt{n}$	$2 \cdot \sqrt{n}$	$\sqrt{n}$	für optimale Indexgröße $k = \sqrt{n}$
$\mathcal{A}_4$: binäres Suchen	direkt auf Indexzahlungen	sortiert $=, \leq$ entscheidbar	$\log_2(n)$	$\log_2(n)$	$\log_2(n)$	$\log_2(n)$	

Schließlich zeigen wir, daß "binäres Suchen" der optimale Suchalgorithmus gemäß des Suchschemas ist, d.h. basierend auf Vergleichen von sich mit ausgewählten Werten $a[i]$.

Genauer:

Sei $\mathcal{A}$ ein Suchalgorithmus gemäß dem Suchschema, für dessen Auswahlen i_ℓ [$i_\ell \hat{=}$ Wahl von i im ℓ-ten Durchlauf] gilt:

$i_1 := c_m$, d.h. 1. Auswahl für alle Werte und alle Suchprogramme gleich

$i_{\ell+1} := f_m(\underbrace{\text{such} < a[i_1], \ldots, \text{such} < a[i_\ell]}_{\ell \text{ Boolsche Werte}})$, d.h. $(\ell+1)$-te Auswahl hängt von den bei vorangegangenen Vergleichen gelieferten Boolschen Werten ab

Dann gilt $E_m^+(\mathcal{A}) \geq E_m^+(\mathcal{A}_4)$.

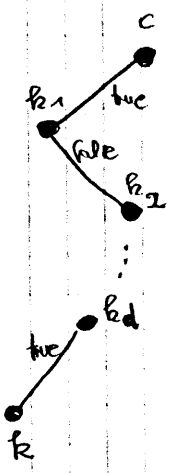
Begründung: $\mathcal{A}_4$ ist ein solcher Suchalgorithmus.

Die Arbeitsweise durch $\mathcal{A}_4$ haben wir auf S. 22.21 für festes m durch einen binären Baum dargestellt.

Die Arbeitsweise jedes beliebigen Suchalgorithmus $\mathcal{A}$ läßt sich aufgrund der Voraussetzungen für festes m ebenfalls durch einen binären Baum darstellen:

1) Wurzel: c_m

2) Die Wurzel sei mit Knoten k verbunden durch Weg $c, k_1, \ldots, k_d, k$



betrachte die nächste Auswahl:

$$f(\underbrace{such\_a[c], \ldots, such\_a[k_d]}_{\text{"Vorgeschichte der Suche"}}, \underbrace{such\_a[k]}_{\text{nur zwei Werte möglich: TRUE, FALSE}})$$

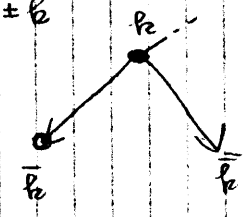
"Vorgeschichte der Suche", noch Einfluss von k fest!

d.h. es liegt folgende Form vor:

$$f(\text{Vorgeschichte}, x) = \begin{cases} \overline{k} & \text{falls } x = \text{TRUE} \\ \overline{\overline{k}} & \text{falls } x = \text{FALSE} \end{cases}$$

also kann sich der Baum weiter konstruieren:

Fall 1: $\overline{k} \neq \overline{\overline{k}}$



Fall 2: $\overline{k} = \overline{\overline{k}}$



(d.h. im Fall 2 keine Verzweigung; maximale Verzweigung ist 2, im Fall 1)

Jetzt m, α können wir also einen binären Suchbaum $B_m(\alpha)$ zeichnen.

$$m \cdot E_m^+(\alpha) = \sum_{\substack{\text{alle Wege } w \\ \text{von Wurzel} \\ \text{zu einem} \\ \text{Knoten in } B_m(\alpha)}} \underbrace{[\text{Länge}(w)+1]}_{\#\text{Knoten auf dem Weg}} =: \underbrace{WL(B_m(\alpha))}_{\text{Wglänge}}$$

Es bleibt zu zeigen:

$$WL(\underbrace{B_m(\alpha_4)}_{\substack{\text{für } m = 2^n - 1 \\ \text{ist das der} \\ \underline{\text{vollständige binäre Baum}} \\ \text{der Tiefe } n \\ (\text{mit } m \text{ Knoten})}}) \leq WL(B) \quad \text{f. alle binären Bäume mit } m \text{ Knoten}$$

d.h. es bleibt eine Behauptung über binäre Bäume zu zeigen, die wir hier nicht weiter verfolgen wollen.

Natürlich ist binäres Suchen nicht optimal schlechthin. Bei Suchverfahren, die "mehr Information" verarbeiten können, kann man schneller suchen:

Bsp

im Feld a: ARRAY [1..100] OF INTEGER
seien die Zahlen 1, 2, ., 99, 100 gespeichert.

Suchen ist hier trivial:

```
IF (1 <= such ) AND ( such <= 100)
   THEN      i := such ; gefunden := TRUE
   ELSE gefunden := FALSE
END;
```

d.h. das Suchargument selbst enthält "die Information über die Speicherstelle".

Suchen (und Sortieren) sind klassische Probleme der Programmierung!

- grundlegendes Buch: D. Knuth, The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley, 1973

weitere Vertiefung: ⟹ Grundzüge der Informatik II (Datenstrukturen und Algorithmen)

Aufwandsanalyse:

- Festlegung des Maßes: (abstrakte) Zeit, Speicherbedarf, Rechenzugriffe, ...
- für vorgegebenes Problem einen guten Algorithmus finden:
  - Aufwand gemäß Maß bestimmen (ergibt obere Schranke bzg. des Problems)
  - Optimalität zeigen
    i. allg. sehr schwierig, (nichttrivale) untere Schranken (d.h. Aussagen der Form: "alle Algorithmen benötigen mehr als ...") sind sehr schwierig zu beweisen; eine Reihe von seit Jahren ungelösten widrigen Fragestellungen der Informatik betrachten die Frage nach solchen unteren Schranken

weitere Vertiefungen:

⟹ Vertiefung Datenstrukturen (weitere Beispiele)

⟹ Vertiefung Berechenbarkeit (Unentscheidbarkeits-Beweise in Schwerentscheidbarkeits-Beweise überführen)

⟹ Komplexitätstheorie

D. Datentypen

23. Allgemeines und ein Rückgriff (: Felder)

Erinnerung: Datentyp – Wertemenge(n) und (zugehörige) Operationen

bisher im wesentlichen nur "skalare Datentypen":

- die Werte werden als "atomar", "unteilbar" aufgefaßt (auf der Ebene von MODULA-2)
- als Operationen jeweils
  - Vergleichsoperationen
  - Ordnungsoperationen
  - "algebraische" Operationen

nunmehr auch strukturierte Datentypen:

- Werte werden "strukturiert aufgebaut" aus Elementen von einfacheren, zugrunde liegenden Datentypen.
- Den "strukturierten Datentypen" von MODULA-2 entsprechen "strukturierte Anweisungen" in MODULA-2 zu ihrer Bearbeitung

Ein (strukturierter) Datentyp ist häufig durch folgende Angaben bestimmt:

für Werte:

Beschreibung der Wertemenge
- statisch; etwa durch Vereinbarungen
- dynamisch; etwa dadurch, daß man angibt, wie man die Elemente (dynamisch) mit Hilfe der Operationen erzeugt.

Für Operationen:

- Beschreibung der erzeugenden Operationen (bei dynamischer Beschreibung der Werteräume).
- Beschreibung des "Zugriffs" auf Bestandteile, d.h. der zerlegenden Operationen
- Beschreibung der möglichen Änderungen
- Beschreibung der vom zugrunde liegenden Datentyp induzierten Funktionen

Man beachte den wechselseitigen Bezug von Werten und Operationen: (bei dynamischer Beschreibung der Werteräume) wird Werteräume durch Operationen beschrieben, und Operationen setzen Werteräume voraus!

Häufig muß man unterscheiden zwischen

- Datentyp als solchem,
- der direkten Darstellung des Datentyps mit den elementaren Sprachmitteln von MODULA-2,
- der indirekten Darstellung des Datentyps mit den elementaren Sprachmitteln von MODULA-2,

d.h. manche Datentypen werden von MODULA-2 direkt unterstützt (durch ARRAY, RECORD, SET); andere müssen indirekt dargestellt werden (insbesondere mit Hilfe von Zeigern und Feldern)

ein größeres Beispiel (S. 23.3 - 23.9 mit einigen Unterbrechungen)

ganzzahlige Vektoren und Indizes aus $I = \{1, 2, .., 100\}$.

strukturelle Beschreibung der Werkzeuge:

$$\mathbb{Z}^I = \{ \mu \mid \mu : I \to \mathbb{Z} \}$$

z.B. $(3, 7, .., 2)$ wird aufgefasst als eine Funktion $\mu : I \to \mathbb{Z}$ mit
$\mu(1) = 3, \mu(2) = 7, .., \mu(100) = 2$.

Operationen:

1. zeigende Operation, Projektionen:

$$p_i : \mathbb{Z}^I \to \mathbb{Z} \quad \text{mit} \quad p_i(\mu) = \mu(i)$$

2. Änderungsoperation

$$\text{ändere} : \mathbb{Z}^I \times I \times \mathbb{Z} \to \mathbb{Z}^I \quad \text{mit}$$

$$\underbrace{\text{ändere}(\mu, i, e)}_{\in \mathbb{Z}^I}(j) = \begin{cases} \mu(j) & \text{falls } i \neq j \\ e & \text{falls } i = j \end{cases}$$

3. induzierte Operationen

$$\text{gleich} : \mathbb{Z}^I \times \mathbb{Z}^I \to \mathbb{B} \quad \{\text{FALSE}, \text{TRUE}\}$$
$$\text{gleich}(\mu, \nu) \Leftrightarrow \text{f.alle } i \in I : \mu(i) = \nu(i)$$

$$\text{gleichungleich} : \mathbb{Z}^I \times \mathbb{Z}^I \to \mathbb{B}$$
$$\text{gleichungleich}(\mu, \nu) \Leftrightarrow \text{f.alle } i \in I : \mu(i) \neq \nu(i)$$

$$\text{add} : \mathbb{Z}^I \times \mathbb{Z}^I \to \mathbb{Z}^I \qquad \text{add}(\mu, \nu)(i) = \mu(i) + \nu(i)$$

$$\text{mult} : \mathbb{Z} \times \mathbb{Z}^I \to \mathbb{Z}^I \qquad \text{mult}(e, \mu)(i) = e \cdot \mu(i)$$

$$\text{produkt} : \mathbb{Z}^I \times \mathbb{Z}^I \to \mathbb{Z} \qquad \text{produkt}(\mu, \nu) = \sum_{i \in I} \mu(i) \cdot \nu(i)$$

4. zu erzeugende Operationen:

. $\mathbb{Z}^I$ läßt sich z.B. erzeugen aus den speziellen Vektoren der Form $(0,..,1,..0)$:

$$einheit_i : \longrightarrow \mathbb{Z}^I \qquad einheit_i(j) = \begin{cases} 1 & falls\ i=j \\ 0 & falls\ i \neq j \end{cases}$$

↑
"0-stellige Funktion"
bezeichnet "Konstante"

Bei diesem Beispiel wird nicht die "Struktur", d.h. hier das Gerüst des n-Tupels (bei "festem Gerüst") erzeugt, sondern der Erzeugungsprozeß von irgendwelchen Datentyp induziert!

Darstellung mit den Sprachmitteln von MODULA-2

zugrunde liegende Datentypen:

$\mathbb{Z}$ durch INTEGER mit entsprechenden Operationen
$\mathbb{B}$ durch BOOLEAN " " "
I durch [1..100] " " "

Wertemenge: CONST n = 100;
TYPE vektor = ARRAY [1..n] OF INTEGER;

Operationen:

1) sei a ein Bezeichner für Vektoren,
i ein Bezeichner für Indizes,

etwa vereinbart vermöge VAR a: vektor;
i: [1..n];

zu 1): Projektion p_i wird beschrieben durch den Ausdruck a[i]

zu 2. Änderungsoperation:

Änderung wird beschrieben durch die Anweisung der Wertzuweisung

$$a[i] := e$$

Man beachte, daß dabei der Vorstellung zugrunde liegt, daß das gesamte Feld neu bestimmt wird; die Änderung in den Komponenten ungleich i sind dabei trivial.

Diese Vorstellung ist auch nützlich, wenn man die axiomatische Semantik der Wertzuweisung von $a[i] := e$ festlegen will, weil die unbedachte Übertragung des Wertzuweisungsaxioms nicht das Gewünschte liefert:

Bsp

Was ist $wp(\underbrace{a[i] := 3}_{x := e}, \underbrace{a[i] = a[j]}_{Q(a[i])})$?

von der Form

Unbedachte Anwendung des Wertzuweisungs-Axioms aus § 11 würde ergeben:

$$Q\left[{}^{a[i]}/_{e}\right] \Leftrightarrow 3 = a[j]$$

Aber: Für $i = j$ ist diese Bedingung nicht notwendig!

Was ist passiert? Wir müssen anknüpfen an die Vorstellung, daß Bezeichner Repräsentanten für Speicherplätze sind, vgl. § 10, § 19.

einfachster Fall: - (ein Bezeichner repräsentiert genau einen Speicherplatz;
- (als Zeichenketten, d.h. syntaktisch) verschiedene Bezeichner repräsentieren verschiedene Speicherplätze!

Dieser einfachste Fall liegt dem Wertzuweisungsaxiom aus §11 zugrunde: zwischen Bezeichnern und Speicherplätzen gibt es eine eineindeutige Zuordnung

schwierige Fälle:

1) ein Bezeichner repräsentiert mehrere Speicherplätze, siehe §19.

2) mehrere Bezeichner repräsentieren den gleichen Speicherplatz:

wir kennen bislang zwei solche Situationen:
- nach dem Aufruf einer Prozedur repräsentiert ein Formaler Ergebnis- Parameter den gleichen Speicherplatz wie der aktuelle Parameter (siehe auch §27)
- Felder eröffnen die Möglichkeit von "errechneten" Bezeichnern, wie folgt:

Wir fassen Ausdrücke der Form $a[e]$ als "errechnete Bezeichner" auf: gilt $e_1 = e_2$, so repräsentieren

$a[e_1]$ und $a[e_2]$ den gleichen Speicherplatz,

d.h. insbesondere, daß eine Wertzuweisung der Form

$$a[e_1] := f$$

auch für $a[e_2]$ "sichtbar wird".

Das Wertzuweisungsp.-Axiom für Felder lautet korrekt:

$$wp(a[i] = e, Q(a)) \Leftrightarrow Q\left[{}^{a}/_{\text{ändere}(a,i,e)}\right] \quad (\wedge\ e \text{ definiert})$$

↑ bezeichnet das gesamte Feld

↑ bezeichnet das gesamte veränderte Feld

Bsp

$$wp(a[i] := 3 \;,\; a(i) = a(j))$$

$$\overset{\text{Axiom}}{\Leftrightarrow} \quad \underbrace{\text{ändere}(a, i, 3)(i)} = \text{ändere}(a, i, 3)(j)$$

$$\Leftrightarrow \quad 3 = \begin{cases} a(j) & \text{falls } i \neq j \\ 3 & \text{falls } i = j \end{cases}$$

$$\Leftrightarrow \quad (i = j \wedge 3 = 3) \vee (i \neq j \wedge a(j) = 3)$$

$$\Leftrightarrow \quad i = j \vee a(j) = 3$$

zu 3). induzierte Operationen:

In MODULA-2 sind standardmäßig für Felder keine Vergleichs- und Ordnungsoperationen definiert. Wir können diese durch selbstgeschriebene Funktionen nachbilden:

um Kopieren zu vermeiden

```
PROCEDURE vektorgleich( VAR a,b: vektor ) : BOOLEAN;
  (* benutzt die Feldgröße n als globalen Bezeichner *)
  VAR i : CARDINAL; (*[1..n], n+1 *)
      test : BOOLEAN;
  BEGIN test:=TRUE; i:=1;
      WHILE test AND (i <= n) DO
          IF a[i]=b[i] THEN i:=i+1
                       ELSE test:= FALSE; END
      END;
      RETURN test
END vektorgleich;
```

```
PROCEDURE vektorkleinergleich (VAR a,b: vektor) : BOOLEAN;
  (* benutzt die Feldgröße n als globalen Bezeichner *)
VAR i : CARDINAL ; (* [1..n], n+1 *)   test : BOOLEAN ;
BEGIN test := TRUE ; i := 1;
      WHILE test AND (i <= n) DO
         IF a[i] <= b[i] THEN i := i+1 ELSE test := FALSE END;
      END;
      RETURN test
END vektorkleinergleich;
```

Die Operationen add und mult stellen uns vor ein neues Problem, nämlich daß in MODULA-2 Funktionsprozeduren als Ergebnisse nur skalare Werte (oder Zeiger, siehe § 27) liefern dürfen. Arbeitet man ohne Zeiger, so verbleibt nur die etwas unbefriedigende Lösung, daß man Prozeduren wie folgt schreibt:

aus Effizienzgründen

```
PROCEDURE vektoradd ( (* Argumente *) VAR a,b: vektor ;
                      (* Ergebnis *) VAR vektorsumme : vektor );
```

(Argument-) Ergebnis-Parameter

```
  (* benutzt Feldgröße n als globalen Bezeichner *)
VAR i : [1..n] ;
BEGIN
   FOR i := 1 TO n DO vektorsumme [i] := a[i] + b[i] END
END vektoradd;
```

analog:

```
PROCEDURE vektormult (e: INTEGER ; VAR a : vektor ; VAR vektorvielfaches : vektor);
VAR i : [1..n] ;
BEGIN FOR i := 1 TO n DO vektorvielfaches [i] := e * a[i] END
END vektormult;
```

Das Produkt kann man wie ab Funktionsprozedur definieren

```
PROCEDURE vektorprodukt( VAR a,b : vektor) : INTEGER;
  (* benutzt Feldgröße n als globalen Bezeichner *)
  VAR zwischensumme : INTEGER;
      i : [1..n];
  BEGIN zwischensumme := 0;
        FOR i := 1 TO n DO zwischensumme := zwischensumme + a[i] * b[i] END;
        RETURN zwischensumme
  END vektorprodukt;
```

Bemerkungen

Das Wünschenswerte läßt sich in MODULA-2 manchmal nur mit einigen Umwegen darstellen, und solche Hilfsdarstellungen müssen dann diszipliniert verwendet werden.

zwei offene Wünsche:

- Funktionen, die auch Felder als Werte liefern.
- "Schnelle" Argument-Parameter, die nicht als Ergebnis-Parameter verwendet werden können.

folgende Wünsche sind dagegen von MODULA-2 zum Teil unterstützt:

- Prozeduren, [illegible] die als Parameter Felder schablonenhaft erlauben, wobei Länge des Feldes (aber nicht auch Typ der Feldelemente?) nicht fixiert sind: siehe open array parameter.
- Absicherung des Datentyps, die sicherstellt, daß man Variablen ausschließlich durch die angebotenen Operationen manipulieren kann: vermöge des Modulkonzepts erreichbar.

- Teilweise Trennung des Kopfes vom Rumpf der Prozeduren um eine bessere Modularisierung von Programmen zu erreichen:

der "Anwender" des Datentyps braucht nur zu kennen:

- Kopf der Prozeduren
- die Spezifikation der beschriebenen Funktion bzw. Zustandsüberführung

der Anwender braucht bzw. soll bzw. darf nicht kennen:

- die gewählte Verwirklichung der Spezifikation, d.h. den Rumpf:

kann über das Modulkonzept verwirklicht werden.

MODULA-2 unterstützt <u>direkt</u> die Bildung von Wertemengen (vom strukturierten Datentypen), die durch die folgenden drei grundlegenden Arten statischer Beschreibung gebildet werden:

1) kartesische Produkte $\bigtimes_{k\in I} A_k \cong A_{k_1} \times \ldots \times A_{k_n} \cong \{\mu \mid \mu: I \to \bigcup_{k\in I} A_k \text{ und } \mu(k) \in A_k\}$

2) disjunkte Vereinigungen $\bigcup\!\!\!\bullet_{k\in I} A_k \cong A_{k_1} \cup \ldots \cup A_{k_n} \cong \{(k,x) \mid k \in I \text{ und } x \in A_k\}$

3) Potenzmengen $\mathcal{P}A = \{X \mid X \subset A\} \cong \{\mu \mid \mu: A \to \{\text{false}, \text{true}\}\}$

wobei A_k bzw A und $I = \{k_1, \ldots, k_n\}$ Mengen von zugrundeliegenden Datentypen sind.

Dies wird in § 24-26 behandelt.

MODULA-2 unterstützt die Bildung von Wertemengen (vom strukturierten Datentypen), die dynamisch durch erzeugende Operationen beschrieben werden, vermöge Zeigerstrukturen. Dies wird in § 27 behandelt.

<u>nur indirekt</u>

24 Kartesische Produkte

1. Homogene Produkte mit (endlichem) skalarem Index : Felder

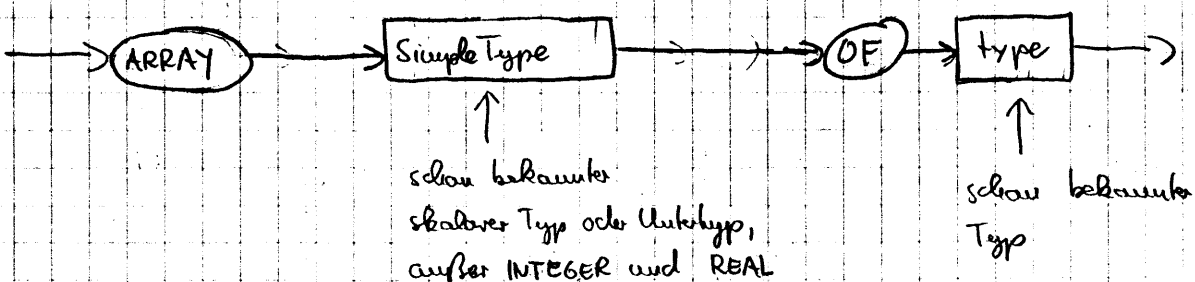
Ist das kartesische Produkt $\bigtimes_{k \in I} A_k$

- homogen : alle A_k gleich, etwa gleich einem Grundtyp A
- mit skalarem Index : I Wertemenge eines endlichen skalaren Typs oder Untertyps, etwa $I = (k_1 < \ldots < k_n)$

so kann es, wie schon aus §16 bekannt, in MODULA-2 beschrieben werden durch z.B:

```
TYPE  grundtyp = ... ;
      index    = (k1, .., kn);
      produkt  = ARRAY index OF grundtyp;
```

allgemein:

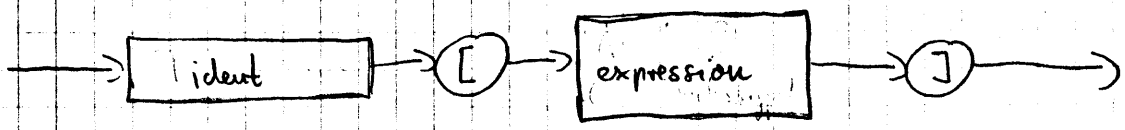


Durch eine Vereinbarung der Form

VAR feld: produkt ;

werden neue "errechnete Bezeichner", nämlich insbesondere feld[k_1], ..., feld[k_n], für Werte vom Typ grundtyp vereinbart.

In Ausdrücken, vgl. Def. von Faktor, dürfen als "Designator" dann "errechnete Bezeichner" der folgenden Form vorkommen (sofern die bezeichneten Werte vom passenden Typ sind):



Anstelle der "Konstanten" $k_1, \ldots, k_n$ dürfen also auch Ausdrücke stehen!

Der Wert eines Ausdrucks feld[e] wird wie folgt ermittelt:

- werte den Ausdruck e aus
- liefert e einen Wert vom Indextyp, etwa k_i,

  so wird der durch feld[k_i] bezeichnete Wert geliefert

[Achtung: Der Compiler kann prüfen, ob e formal einen Wert vom Indextyp liefert (besser: liefern kann);

ob bei einer Ausführung tatsächlich ein Wert vom Indextyp geliefert wird, wird von manchen Laufzeitsystemen geprüft, aber grundsätzlich muß der Programmierer dafür Sorge tragen!]

eine typische Fehlersituation:

```
VAR a: ARRAY [1..n] OF INTEGER;
    i: CARDINAL;
 test: BOOLEAN;
  :
test := ... ;
  :

i := 0;
WHILE test AND i<n DO
   ... i := i+1; ....
END;
a[i] := ... ;
```

↑ falls zu Beginn der WHILE-Schleife test den Wert FALSE bezeichnet, so bezeichnet i den Wert 0, aber a[0] ist nicht verwendbar.

(als aktuelle Argument-Parameter dürfen sie in ihrer Eigenschaft als Ausdruck vorkommen)

"Errechnete Bezeichner" dürfen auch vorkommen
- auf der linken Seite einer Wertzuweisung
- als aktuelle (Argument-Ergebnis-) Parameter für Prozeduraufrufe;

Der Grundtyp einer Feldvereinbarung kann wieder ein Feld sein, z.B.

ARRAY index2 OF ARRAY index1 OF grundtyp

Errechnete Bezeichner haben dann die Form:

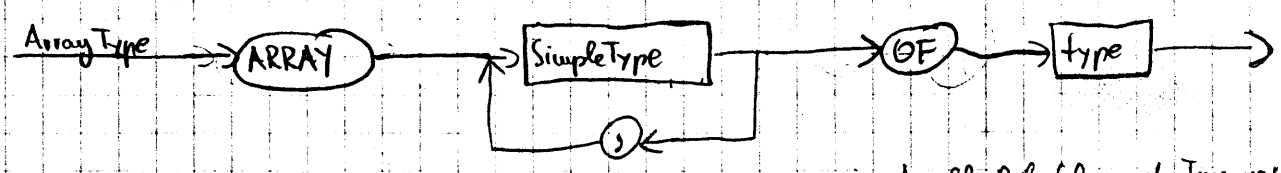
$$\underbrace{a[k_{2i}]}_{\text{bezeichnet ein Element vom Typ ARRAY index1 OF grundtyp}}[k_{1j}]$$

Die Vereinbarung mehrdimensionaler Felder kürzt man ab durch

ARRAY index2, index1 OF grundtyp ,

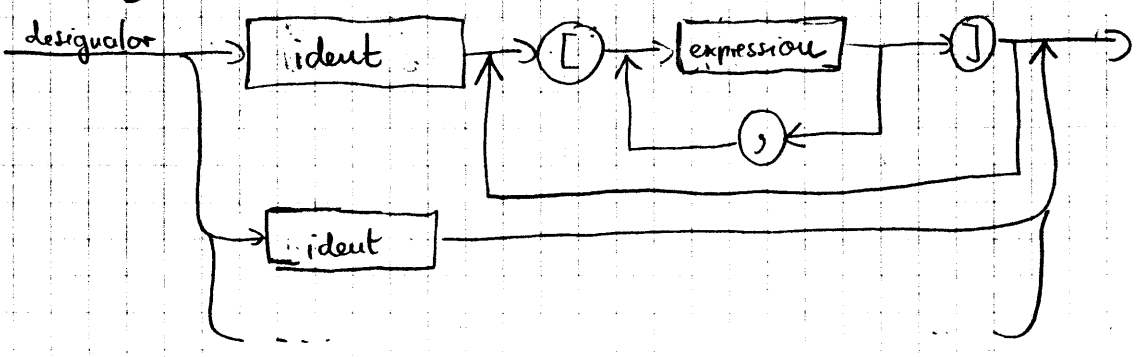
errechnete Bezeichner haben bei dieser Abkürzung die Form $a[k_{2i}, k_{1j}]$

allgemein



(vgl. Seite 9.14; noch vorläufig)

in Anzahl, Reihenfolge und Typ passend zur Vereinbarung



Die beschränkte Wiederholung mit FOR, vgl. 12.5, ist die dem Feldaufbau entsprechende strukturierte Anweisung:

```
FOR i := k1 TO kn DO .... feld[i] ... END;
```

Man beachte: – Die Länge eines Feldes wird durch Vereinbarung festgelegt und kann nicht mehr geändert werden.

– Die Anzahl der Wiederholungen der beschränkten Wiederholung wird zu Beginn ihrer Ausführung festgelegt und kann nicht mehr geändert werden.

Felder über dem Standardtyp Zeichen dienen zur Darstellung von Zeichenreihen oder Worten:

Standardtyp Zeichen: skalarer Datentyp, durch CHAR bezeichnet; Standardbezeichnung für Zeichen x ist "x".

strukturierte Datentypen der Form: Zeichenfolge der (festen) Länge n+1 ;

Bezeichner für einen solchen Typ werden vereinbart mit

ARRAY [0..n] OF CHAR

Standardbezeichner für Zeichenfolgen der Länge n+1, etwa für $x_0 \dots x_n$, sind

$$\underbrace{\text{"}x_0 \dots x_n\text{"}}_{\text{genau } n \text{ Zeichen}} \quad \text{oder} \quad \text{'}x_0 \dots x_n\text{'}$$

Bsp:

```
CONST namenslaenge = 10;
TYPE namensindex = [0..namenslaenge];
     name = ARRAY namensindex OF CHAR;

VAR teilnehmer : name;
  ⋮

teilnehmer := "biskup"
```

hier dürfen höchstens 11 Zeichen stehen, fehlende werden mit Blanks aufgefüllt

2. (möglicherweise) inhomogene Produkte mit Bezeichnungen als Index: Verbunde

Ist das kartesische Produkt $\underset{k \in I}{\times} A_k$

- (möglicherweise) inhomogen, d.h. die A_k können verschiedene Mengen sein
- mit Menge von Bezeichnungen als Indexmenge, etwa $\{bez_1, .., bez_n\}$,

so kann es in MODULA-2 beschrieben werden durch einen Verbund.

Beispiele 1) TYPE datum = RECORD
tag : [1..31];
monat : [1..12];
jahr : [1..2100]
END

beschreibt

$\{1,..,31\}_{tag} \times \{1,..,12\}_{monat} \times \{1,..,2100\}_{jahr}$

mit Bezeichnungen {tag, monat, jahr}.

2) TYPE spielzeug = RECORD

art : (ball, puppe, spiel, buch);

preis : REAL;

spaß : (viel, etwas, wenig, garnicht);

kaputt, verloren : BOOLEAN

END;

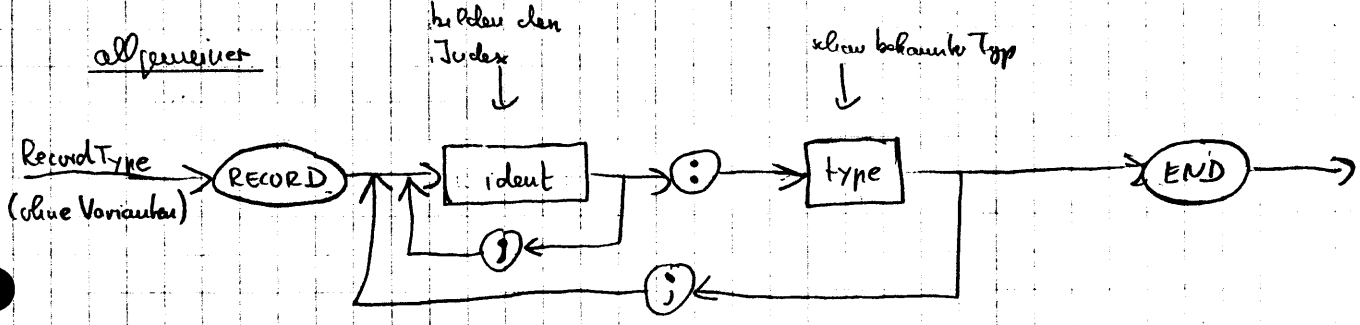
beschreibt

$\{ball, puppe, spiel, buch\}_{art} \times \mathbb{R}_{preis} \times \{viel, etwas, wenig, garnicht\}_{spaß} \times \{FALSE, TRUE\}_{kaputt} \times \{FALSE, TRUE\}_{verloren}$

(mit $\mathbb{R}$: endliche Teilmenge von $\mathbb{R}$)

mit Bezeichnungen {art, preis, spaß, kaputt, verloren}

allgemeiner



etwa verwendet gemäß folgendem Schema:

TYPE produkt1 = RECORD bez_1 : T_1;

bez_n : T_n

END;

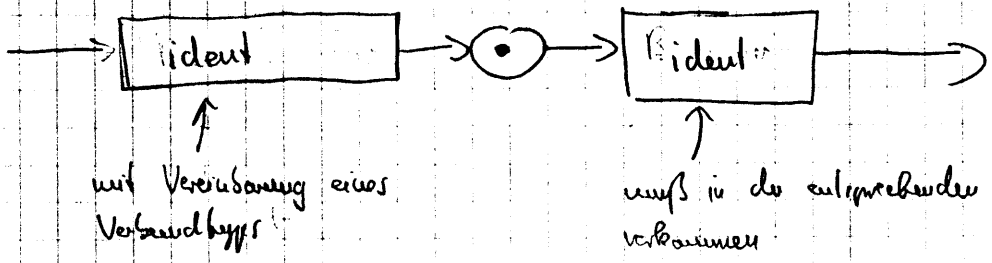
Durch eine Vereinbarung der Form

VAR verbund : produkt1;

werden neue "zusammengesetzte Bezeichner", insbesondere nämlich

verbund.bez_1, ..., verbund.bez_n für die zugehörigen Werte vereinbart.

In Ausdrücken, vgl. Def. von Faktor, dürfen als Designator dann "zusammengesetzte Bezeichner" der folgenden Form vorkommen (sofern die bezeichneten Werte vom passenden Typ sind):



"Zusammengesetzte Bezeichner" dürfen auch vorkommen
- auf der linken Seite einer Wertzuweisung
- als aktuelle (Argument- Ergebnis-) Parameter für Prozedur- bzw. Funktionsaufrufe.

Felder und Verbunde können geschachtelt auftreten, so kann es

Felder von Feldern bzw. Verbunde oder

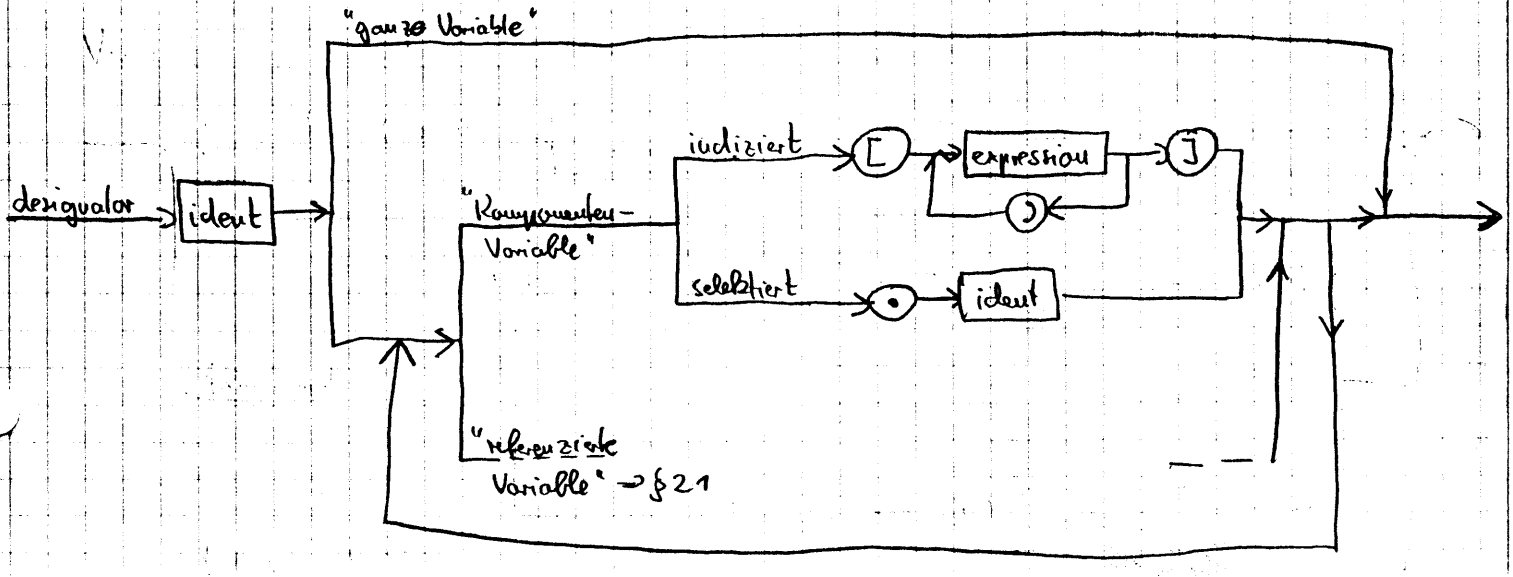
Verbunde mit Feldern bzw. Verbunde geben.

(als aktuelle Argument-Parameter dürfen sie in ihrer Eigenschaft als Ausdruck vorkommen)

Die Komponenten werden ebenfalls durch Schachtelungen der Zugriffs-Operationen bezeichnet: für Felder jeweils in der Form f[k]

für Verbunde jeweils in der Form v.k

Damit können wir jetzt Designator etwas ausführlicher definieren:



(Abgesehen vom Typ ARRAY [0..n] OF CHAR) gibt es in MODULA-2 keine Standardbezeichner für die Elemente des durch ein Feld bzw. Verbund beschriebenen Wertevorrat, d.h.

für VAR d: datum

darf man leider nicht schreiben d := (11, 11, 2000)

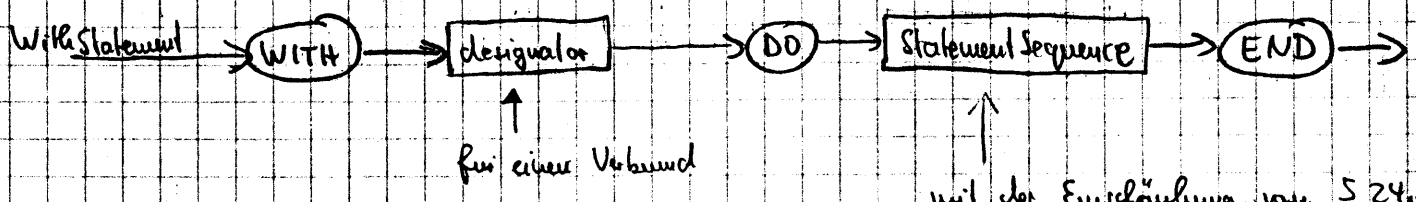
Stattdessen muß man schreiben

```
d.tag   := 11;
d.monat := 11;
d.jahr  := 2000;
```

Das kann man wie folgt abkürzen:

```
WITH d DO
    tag   := 11;
    monat := 11;
    jahr  := 2000
END;
```

<u>allgemein</u>



S 24.11

24.11

Anweisungsmächtigkeit

WITH x DO S END , wobei x eine Variable für einen Verbund mit Bezeichnungen $\{bez_1, .., bez_n\}$ als Index sei, ist gemäß Beschreibung der unten stehenden Einschränkung gleichbedeutend mit

$$S\left[{}^{bez_1}/_{x.bez_1}, .., {}^{bez_n}/_{x.bez_n}\right],$$

d.h. jedes Vorkommen eines Bezeichners bez_i aus dem Index wird durch $x.bez_i$ ersetzt.

2) Allgemein sollten innerhalb der Anweisungsfolge S nur Wertzuweisungen an die Komponenten des durch x angegebenen Verbundes erfolgen.

Einschränkung:

1) In WITH x DO S END darf S keine Wertzuweisung enthalten, die eine in x vorkommende Variable verändert; z.B folgendes ist nicht erlaubt:

```
TYPE  V = RECORD ... END;
VAR   a : ARRAY [1..3] OF V;
      i : [1..3];
      :
WITH a[i] DO ... i:=i+1; ... END;
```

Die WITH-Anweisung ist die den Verbunden entsprechende "strukturierte Anweisung"; wegen der Inhomogenität der Verbund-Komponenten hat diese Anweisung nicht die Eigenschaften einer Wiederholungs-Anweisung. Sie dient im wesentlichen der "Strukturierung" im Sinne von "Lesbarkeit", neben möglicher Optimierungen der Komponenten-Zugriffe durch den Compiler. Die Einschränkung rührt daher, daß in WITH x DO S END für eine Komponenten-Variable x (etwa der Form f[k] bzw. v.k], die Indizes bzw. Selektoren (hier: k) nur einmal zu Beginn der Ausführung bestimmt werden. Ferner können sich durch die Ersetzung hez/x.hez. unbeabsichtigte Effekte bei Namenskonflikten ergeben.

25. Disjunkte Vereinigungen: variante Verbunde

Bsp Personen sollen durch folgende Angaben beschrieben werden:

- Name
- Stand:
- falls ledig:
- falls verheiratet oder verwitwet: Datum der Hochzeit
- falls geschieden: Datum der letzten Scheidung
  - Anzahl geschiedener Ehen

Indexmenge für disjunkte Vereinigung:
Wertemenge eines (endlichen) skalaren Typs bzw. Unterbyps

in MODULA-2

```
TYPE   text = ARRAY [0..9] OF CHAR;

       datum = RECORD   tag   :[1..31];
                        monat :[1..12];
                        jahr  :[1..2100]
                END;

       familienstand = (ledig, verheiratet, verwitwet, geschieden);

       person = RECORD
                    name : text;
                    CASE stand : familienstand OF
                        ledig:
                      | verheirat, verwitwet: hochzeit: datum
                      | geschieden: scheidung : datum;
                                    anzahlehen :[1..10]
                    END
                END;
```

Hierdurch wird die disjunkte Vereinigung

$$P_{ledig} \cup P_{verheiratet} \cup P_{verwitwet} \cup P_{geschieden}$$

mit

$$P_{ledig} = \left(\sum^{10}\right)_{name} \times \{ledig\}_{stand}$$

$$P_{verheiratet} = \left(\sum^{10}\right)_{name} \times \{verheiratet\}_{stand} \times \left(\{1,..,31\}_{tag} \times \{1,..,12\}_{monat} \times \{1,..,2100\}_{jahr}\right)_{hochzeit}$$

$$P_{verwitwet} = \left(\sum^{10}\right)_{name} \times \{verwitwet\}_{stand} \times \left(\{1,..,31\}_{tag} \times \{1,..,12\}_{monat} \times \{1,..,2100\}_{jahr}\right)_{witwe}$$

$$P_{geschieden} = \left(\sum^{10}\right)_{name} \times \{geschieden\}_{stand} \times \left(\{1,..,31\}_{tag} \times \{1,..,12\}_{monat} \times \{1,..,2100\}_{jahr}\right)_{scheidung} \times \{1,..,10\}_{anzahl\ ehen}$$

beschreiben.

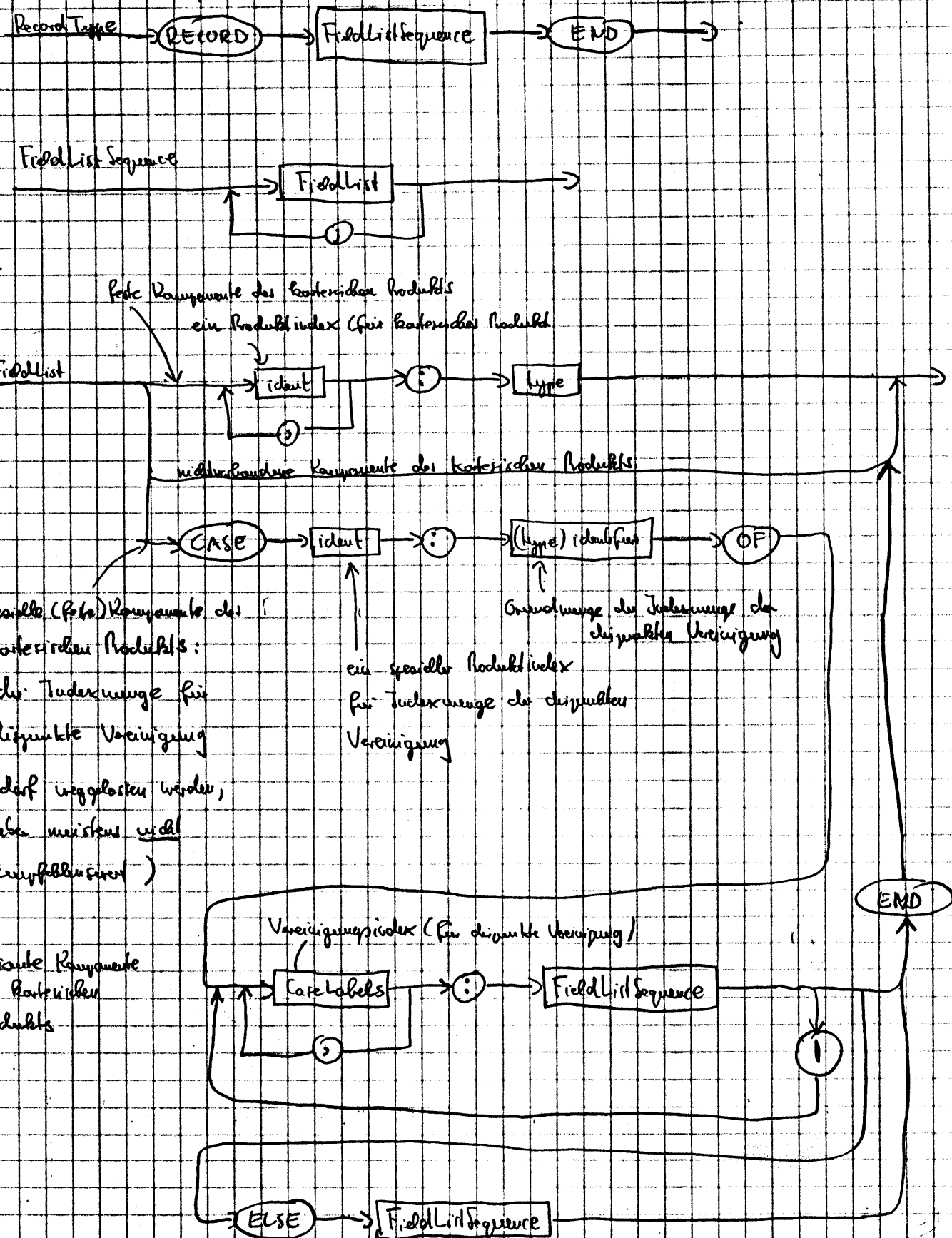
Beim Zugriff auf die einzelnen Komponenten muß man beachten, daß jede Disjunkt verschieden aufgebaut ist. Man muß also jeweils sicherstellen, daß die angesprochenen Komponenten auch tatsächlich vorhanden sind. Durch die Verwendung von disjunkten Vereinigungen kann die ansonsten strenge Typisierung von Bezeichnern durchbrochen werden; hierbei ist äußerste Sorgfalt erforderlich!

Die Selektionsanweisung, § 12.2.3, ist die dem "Variantenteil" (disjunkte Vereinigung) entsprechende "strukturierte Anweisung": durch formale Entsprechung von Vereinbarung und Anweisung kann das Vorhandensein der angesprochenen Komponenten leicht überprüft werden.

Bsp: Drucke die Angaben einer Person aus einer Kartei aus:

```
  :
VAR Kartei : ARRAY [1..n] OF person;
  :
WITH Kartei[i] DO
   WriteString(name);
   CASE stand OF
      ledig : WriteString("ledig") |
      verheiratet : WriteString("verheiratet am");
                    WriteCard(hochzeit.tag); WriteCard(hochzeit.monat);
                    WriteCard(hochzeit.jahr) |
      verwitwet : WriteString("verwitwet am");
                  WriteCard(hochzeit.tag); WriteCard(hochzeit.monat);
                  WriteCard(hochzeit.jahr) |
      geschieden : WriteString("geschieden am");
                   WriteCard(scheidung.tag); WriteCard(scheidung.monat);
                   WriteCard(scheidung.jahr);
                   WriteCard(anzahlkinder)
   END;
   WriteLn
END;
  :
```

Disjunkte Vereinigungen können in MODULA-2 allgemein über variante Verbunde beschrieben werden. Die volle Syntax für Verbunde kennt nunmehr:



Der durch einen Verbundtyp der Form

```
RECORD
    CASE bez : t OF
        k1 : komponliste1 |
        k2 : komponliste2 |
        :
        km : komponlistem
    END
END
```

beschriebene Menge hat die Form

$$\times\ T(t) \times \bigsqcup_{j=1}^{m} T(\text{komponliste}_j),$$

($T(t)$: Indexmenge für disjunkte Vereinigung; $\bigsqcup_{j=1}^{m} T(\text{komponliste}_j)$: variante Komponente)

wobei T die "Semantikfunktion" sei, die den Typbezeichnern jeweils die bezeichneten Wertemengen zuordnet.

26 Potenzmengen

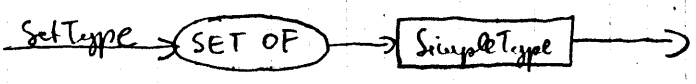
Sei eine Grundmenge ($A = \{a_1, \ldots, a_n\}$) beschränkter als endlicher skalarer Datentyp bzw. Untertyp

Dann ist folgender strukturierter Datentyp verfügbar:

Wertemenge:

$$\mathcal{P}A = \{ \emptyset, \{a_1\}, \ldots, \{a_n\}, \{a_1, a_2\}, \ldots, \{a_{n-1}, a_n\}, \ldots, \{a_1, \ldots, a_n\} \}$$

diese Wertemenge wird vereinbart mit



standardmäßig definiert BITSET = SET OF [0.. W-1] (W: Wortlänge des Rechners)

Bsp

```
TYPE  Kleinzahlen = [1..20];
      menge       = SET OF Kleinzahlen;
```

Operationen

1) Gleichheit $= : \mathcal{P}A \times \mathcal{P}A \longrightarrow \{FALSE, TRUE\}$

Ungleichheit $\# : \mathcal{P}A \times \mathcal{P}A \longrightarrow \{FALSE, TRUE\}$

Mengeninklusion $<= : \mathcal{P}A \times \mathcal{P}A \longrightarrow \{FALSE, TRUE\}$

$x \leq y$: gdw f. alle $a : a \in x \Rightarrow a \in y$

$>= : \mathcal{P}A \times \mathcal{P}A \longrightarrow \{FALSE, TRUE\}$

Elementrow IN : $A \times \mathcal{P}A \longrightarrow \{FALSE, TRUE\}$

a IN x : gdw $a \in x$

---

*) v. noch Implementierung gelten mehr oder weniger starke Einschränkungen bezüglich der Anzahl der Elemente der Grundmenge;

2) Vereinigung $+ : \mathbb{P}A \times \mathbb{P}A \to \mathbb{P}A$

$x + y$ liefert $\{a \mid a \in x \text{ oder } a \in y\}$

Durchschnitt $* : \mathbb{P}A \times \mathbb{P}A \to \mathbb{P}A$

$x * y$ liefert $\{a \mid a \in x \text{ und } a \in y\}$

Differenz $- : \mathbb{P}A \times \mathbb{P}A \to \mathbb{P}A$

$x - y$ liefert $\{a \mid a \in x \text{ und } a \notin y\}$

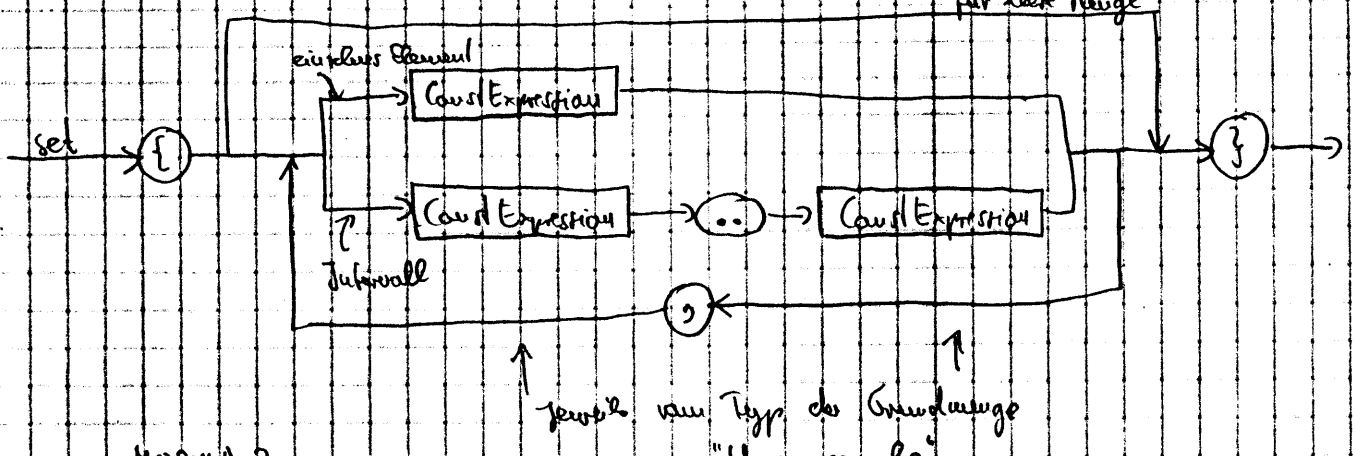
symmetrische Differenz $/ : \mathbb{P}A \times \mathbb{P}A \to \mathbb{P}A$

x / y liefert $\{a \mid (a \in x \text{ und } a \notin y) \text{ oder } (a \notin x \text{ und } a \in y)\}$

Achtung: in 1) und 2) sind die Operationen jeweils nur für Argumente des gleichen Potenzmenge definiert!

3) Mengenbildner $\{\ \}$: Menge der Elementtypen $\to \mathbb{P}A$

gemäß folgender Syntax:



MODULA-2

Bsp: $\{\ \}$ liefert $\varnothing$ (leere Menge)

$\{1..5\}$ liefert $\{1,2,3,4,5\}$

$\{5..4\}$ liefert $\varnothing$

$\{1,3,10..15\}$ liefert $\{1,3,10,11,12,13,14,15\}$

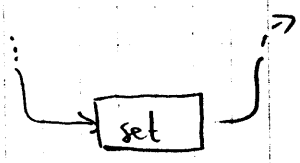
4) Standardprozeduren, um (variabel) gesuchte Werte in eine Menge einzufügen, bzw aus ihr zu entfernen

INCL(s,u)	Prozedur	s := s + {u}	(include)
EXCL(s,u)	Prozedur	s := s − {u}	(exclude)

Alle Operationszeichen können in geeigneter Weise zur Bildung von Ausdrücken verwendet werden; die Syntaxdiagramme von S. 9.13/14 müssen dafür wie folgt ergänzt werden:

in Ausdruck : IN als zusätzlicher "Vergleichsoperator"

in Faktor : zusätzlich




```
TYPE   kleinezahlen = [1..20];
       menge        = SET OF kleinezahlen;

       i, j : kleinezahlen;
       x, y : menge;

   :

       x := (y + {1, 3, 10..15}) * x ;

       IF  i IN x THEN ... ELSE ... END;
```

in Ausdrücken der Form "a IN x" müssen a und x "verträglich" sein, d.h. ist x vom Typ mit Wertemenge PA, so muß a vom Typ mit Wertemenge A sein.

§27 Zeiger

Zunächst drei Vereinbarungen:

1) Bislang konnten die Elemente der Wertemengen von Datentypen im wesentlich nur von fester Größe sein:

skalare Werte;
n-Tupel (Elemente eines kartesischen Produkts) von jeweils fester Länge und Art;
Mengen, jeweils als Teilmenge einer festen Grundmenge;
variante n-Tupel, wobei zwar einerseits die völlig feste Formatierung aufgehoben wird, aber andererseits die Möglichkeiten der Variation von Länge und Art fest vorgegeben ist.

Für die problemangepaßte Beschreibung von Funktionen bzw. Zustandsüberführungen benötigt man jedoch häufig Wertemengen, deren Elemente

- zwar einerseits vom Aufbau her (strukturell) gleichartig sind,
- aber andererseits in ihrer Größe nicht fest beschränkt sind.

Beispiel aus dieser Vorlesung: Bäume (in mehreren Zusammenhängen)

2) Bezeichner für Werte kann man als "Variable" ansehen:

der Bezeichner repräsentiert dann einen Speicherplatz;
der Speicherplatz enthält den bezeichneten Wert

Variable (Speicherplatz) — Wert

↑ "Gefäß" ↑ "Inhalt"

Variable werden in MODULA-2 lokal beim Aufruf einer Prozedur gebunden und leben jeweils bis zur Beendigung der Ausführung.

Bezüglich eines (einzelnen) Prozeduraufrufs ist damit die Anzahl der verfügbaren Variablen fest vorgegeben (nämlich entsprechend der Parameter, lokalen (und ggf. globalen) Bezeichner)

Für die problemangepaßte Beschreibung von Funktionen bzw. Zustandsüberführungen benötigt man jedoch häufig eine (zumindest theoretisch) unbeschränkte Anzahl von Variablen.

Beispiel aus der Vorlesung: die Knoten von Bäumen (in vielfacher Zusammenhängen)

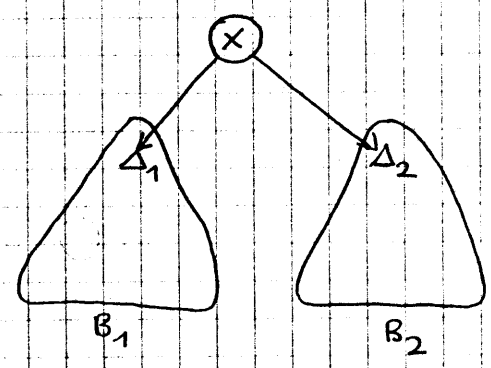
3) Die Wertemenge B eines strukturierten Datentyps wird häufig dynamisch aus den Elementen der Wertemenge W eines zugrunde liegenden (einfacheren) Datentyps aufgebaut mit Hilfe der Operationen des strukturierten Datentyps (vgl. 23.1).

Beispiel aus der Vorlesung:

Bäume werden aus "Knoten", die insbesondere Werte des zugrunde liegenden Datentyps enthalten können, und dann rekursiv aus "schon vorhandenen Bäumen" etwa wie folgt aufgebaut:

(bewerteter, knotenmarkierter, geordneter) Baum über W:

- $\square$ sei der "leere" Baum mit Wurzel $\square$ und innerer Knotenmenge $ikn(\square) = \emptyset$.
- ist $x \in W$ und sind B_1 und B_2 Bäume mit Wurzeln Δ_1 bzw. Δ_2 und inneren Knotenmengen $ikn(B_1)$ bzw. $ikn(B_2)$, so sei

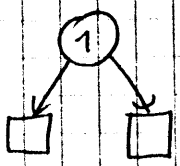


ein Baum mit Wurzel $\otimes$, linkem Teilbaum B_1, rechtem Teilbaum B_2 und innerer Knotenmenge $\{\otimes\} \cup ikn(B_1) \cup ikn(B_2)$

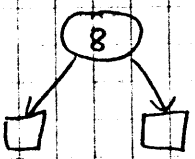
Beispiele: mit $W = \{0, .., 9, +, -, *, /\}$

1) □

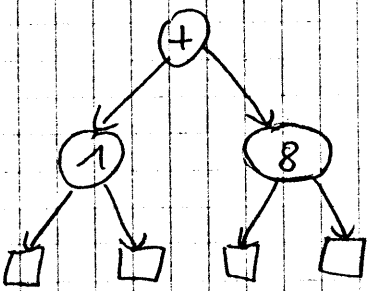
2)



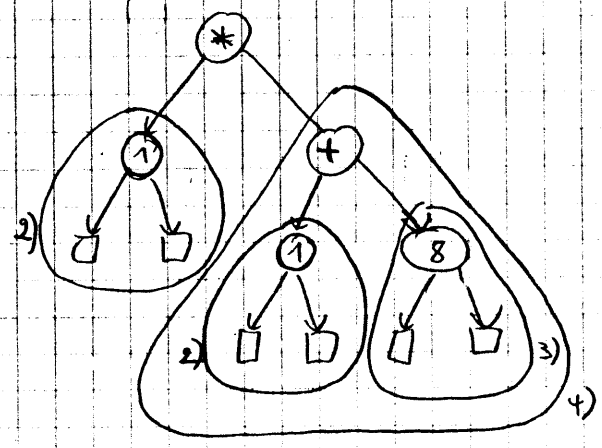
3)



4)



5)



Bemerkung: diese "anschauliche" Definition bedarf noch einiger Verdeutlichungen, z.B.:

- was sind die "Pfeile"?
- wie werden die inneren Knotenmengen der Teilbäume "disjunkt gemacht"?
- wie unterscheidet man "links" und "rechts"?

Zunächst wollen wir solche Bäume als einen <u>abstrakten Datentyp</u> mit dynamischer Beschreibung der Wertemenge behandeln (vgl. § 23):

<u>Wertemenge</u>: <u>Bäume über W</u>, d.h. die Menge der oben beschriebenen Gebilde

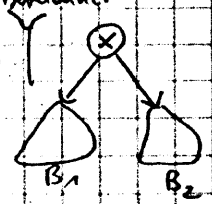
<u>Operationen</u>: (syntaktische Beschreibung durch Angabe der Signaturen und der geforderten Gleichheiten)

- erzeugende Operationen:

leer: $\longrightarrow$ Bäume mit leer bezeichnet □

↑ nullstellige Funktion, Konstante
Verfährt stets den gleichen Wert

construct: Bäume × W × Bäume $\longrightarrow$ Bäume mit construct (B_1, x, B_2) bezeichnet



- Vergleichsoperationen:

isLeer: Bäume $\longrightarrow$ {FALSE, TRUE} mit

isLeer (leer) = TRUE

isLeer (construct (B_1, x, B_2)) = FALSE

<u>beabsichtigte Interpretation</u>

<u>geforderte Gleichheiten</u>

gleich : Bäume × Bäume → {FALSE, TRUE} mit

gefordarte Gleichheiten:

gleich (leer , leer) = TRUE

gleich (leer, construct (B_1, x, B_2)) = FALSE

gleich (construct (B_1, x, B_2) , leer) = FALSE

gleich (construct (B_1, x, B_2) , construct ($\tilde{B}_1$, $\tilde{x}$, $\tilde{B}_2$)) =

⇒ gleich (B_1, $\tilde{B}_1$) AND gleich_W (x, $\tilde{x}$) AND gleich (B_2, $\tilde{B}_2$)

↑ das sei eine ab bekannt vorausgesetzte Funktion auf W, die die "Gleichheit" von Elementen aus W testet

- zerlegende Operationen (jeweils nicht spezifiziert für leer)

linksteil : Bäume ↛ Bäume mit

gleich (linksteil (construct (B_1, x, B_2)) , B_1) = TRUE

rechtsteil : Bäume ↛ Bäume mit

gleich (rechtsteil (construct (B_1, x, B_2)) , B_2) = TRUE

wert : Bäume ↛ W mit

gleich_W (wert (construct (B_1, x, B_2)) , x) = TRUE

- Änderungsoperationen (jeweils nicht spezifiziert für leer)

entferne links : Bäume ↛ Bäume mit

gleich (entferne links (B) , construct (leer, wert (B), rechtsteil (B))) = TRUE

entferne rechts : Bäume ↛ Bäume mit

gleich (entferne rechts (B) , construct (linksteil (B), wert (B), leer)) = TRUE

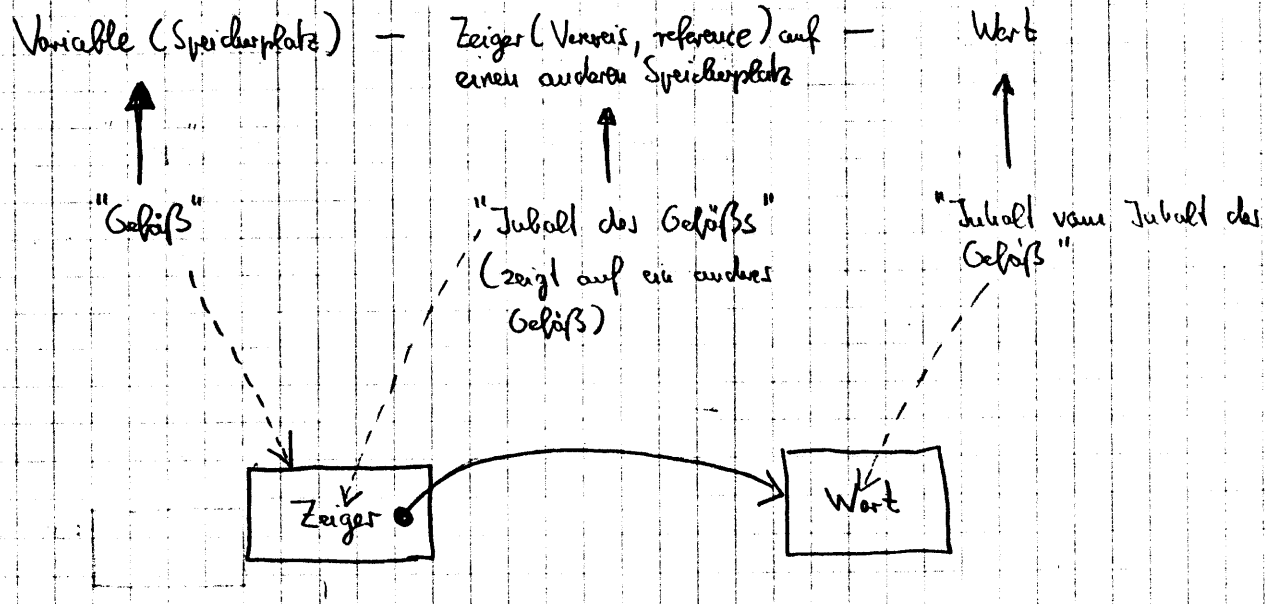
ändere wert : Bäume × W ↛ Bäume mit

gleich (ändere wert (B, y) , construct (linksteil (B), y , rechtsteil (B))) = TRUE

Wir beschäftigen uns mit der Darstellung von Bäumen, d.h. insbesondere mit der Verwirklichung der Definition von Bäumen, geeignete Sprachmittel in MODULA-2.

Für die in den Vorüberlegungen genannten Notwendigkeiten bietet MODULA-2 die Konzepte der Zeiger und dynamischen Variablen an:

anschauliche Einführung von Zeigern:

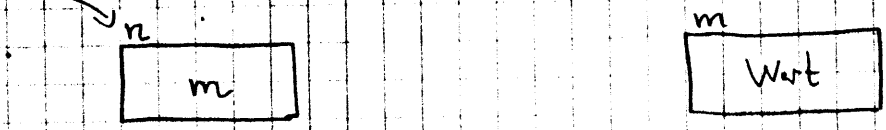


auf der Ebene von MODULA-2

Für einen Designator (Variable, Repräsentation eines Speicherplatzes) kann man vereinbaren, daß die bezeichneten Objekte interne, systemkontrollierte (vom Benutzer abgeschirmte) Bezeichner für Werte eines ausgezeichneten Typs sind. Solche internen, systemkontrollierten Bezeichner nennt man Zeiger.

Bezeichner — interner, systemkontrollierter Bezeichner — Wert
(Zeiger)

auf einer maschinennahen Ebene:



Auf der Ebene der maschinennahen Sprachen ist es im übrigen ganz selbstverständlich, daß man den Inhalt eines Speicherplatzes je nach Zusammenhang als

- einen (ganzzahligen) Wert oder
- als Adresse eines Speicherplatz

interpretiert.

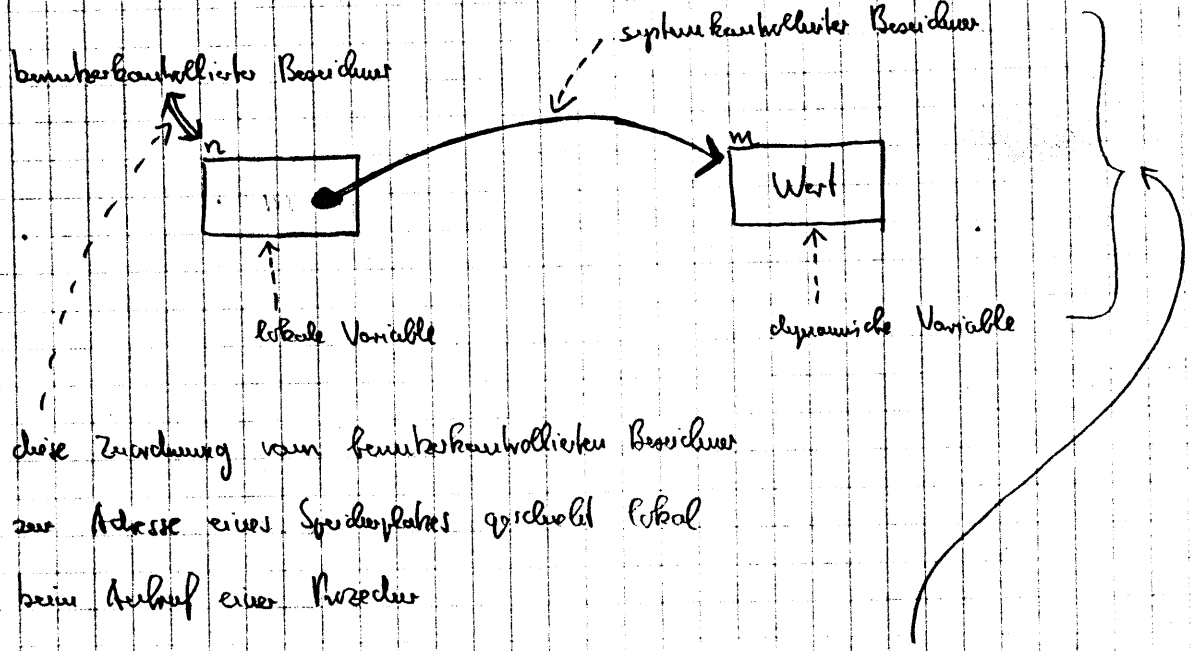
In den höheren Ebenen wurde der Programmierer zunächst von dieser Wahlmöglichkeit abgeschnitten, da (ähnlich wie der Sprunganweisung) ihr Gebrauch Recht zu undurchschaubaren Situationen führt.

Mit Zeigern wird ein halb-disziplinierter Sonderfall wieder zur Verfügung gestellt (so wie schwachtypierte Anweisungen disziplinierte Sonderfälle der Sprunganweisung enthalten).

<u>anschauliche Einführung von dynamischen Variablen</u>:

benannter kontrollierter Bezeichner: repräsentiert <u>lokale Variable</u>, wird beim Aufruf einer Prozedur entsprechend der Deklaration gebunden.

septum kontrollierter Bezeichner: repräsentiert <u>dynamische Variable</u>, wird durch eine besondere Anweisung, NEW( ), gebunden.



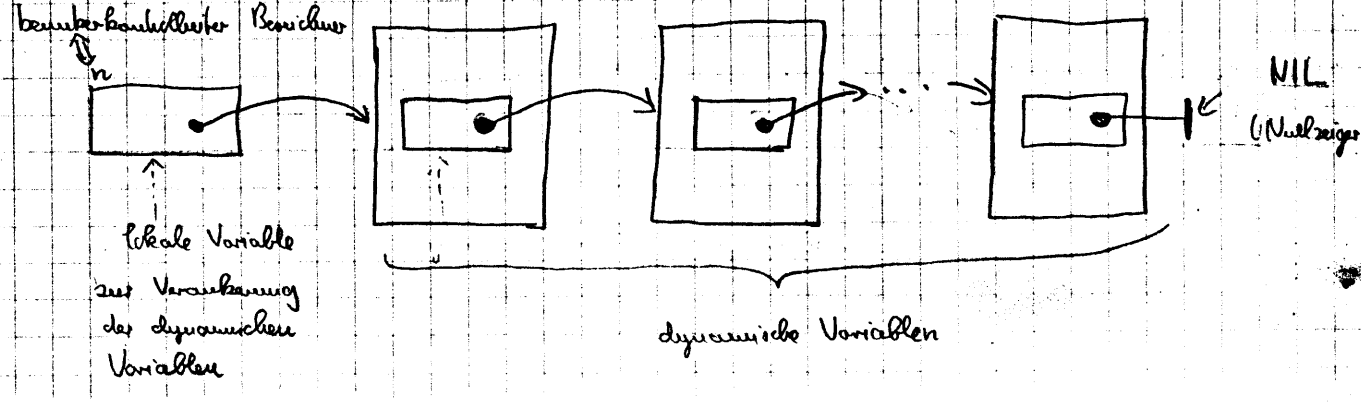
diese Zuordnung vom benutzerbezeichneten Bezeichner zur Adresse eines Speicherplatzes geschieht lokal beim Aufruf einer Prozedur

In MODULA-2 können eine dynamische Variable und die zugehörige systembezeichnete Bezeichner (Zeiger) nur gemeinsam erzeugt werden:

die Anweisung NEW(p)

- bindet eine neue dynamische Variable
- liefert als Wert den zugehörigen Zeiger
- bezeichnet diesen Zeiger durch den aktuellen Parameter p

Man kann auch solche durch Zeiger realisierbaren Verkettungen über mehrere Stufen vornehmen:

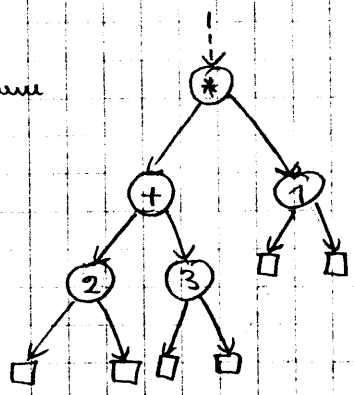


Beispiel Darstellung von Bäumen in MODULA-2 durch Zeiger und dynamische Variablen.

```
TYPE    grundwerte = CHAR;
        baum = POINTER TO knoten;

        knoten = RECORD
                   links : baum;
                   inhalt : grundwerte;
                   rechts : baum
                 END;
```

die hier vereinbarte Objektmenge ist die Menge der Zeiger auf Werte vom Typ knoten, einschließlich des Nullzeigers

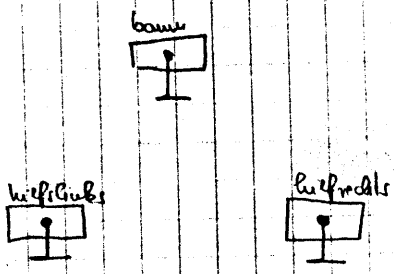
Der Baum



kann dann z.B. wie folgt erzeugt werden:

(Bemerkung: allgemein erzeugt man Bäume beliebiger Größe durch Rekursion, um sich die jeweils "noch weiter zu bearbeitenden Knoten" über die dynamische Kette und die jeweiligen lokalen Bezeichner "merken" zu können.)

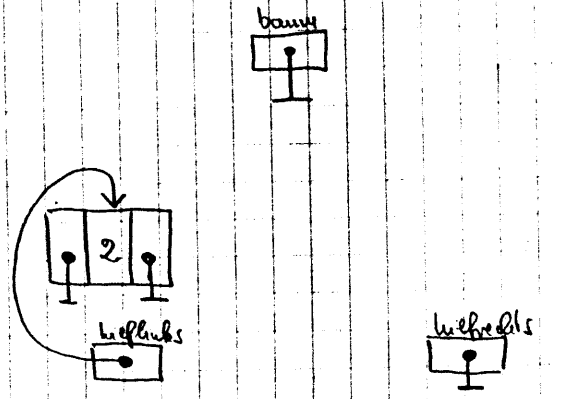
VAR baum, hilflinks, hilfrechts : kante



(Achtung: die Initialisierungen mit den Nullzeigern erfolgt nicht automatisch in MODULA-2)

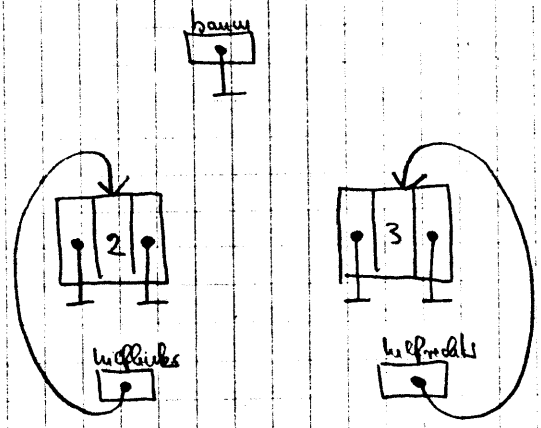
entspricht: construct(leer, 2, leer)

```
NEW (hilflinks);
hilflinks↑. links := NIL;
hilflinks↑. inhalt := "2";
hilflinks↑. rechts := NIL;
```



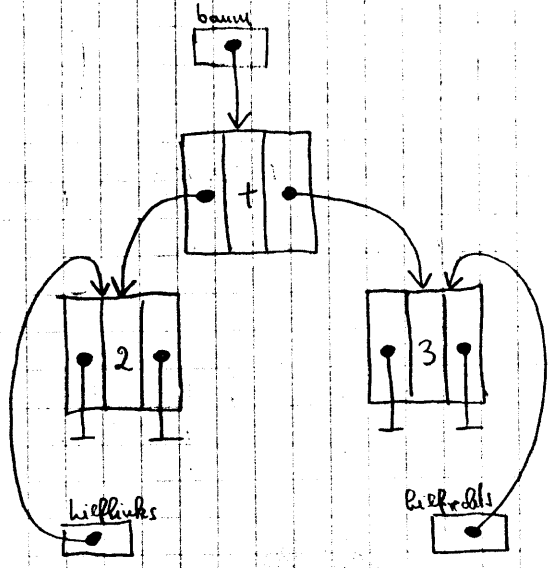
entspricht: construct(leer, 3, leer)

```
NEW (hilfrechts);
WITH hilfrechts↑ DO
        links := NIL;
        inhalt := "3";
        rechts := NIL
END;
```

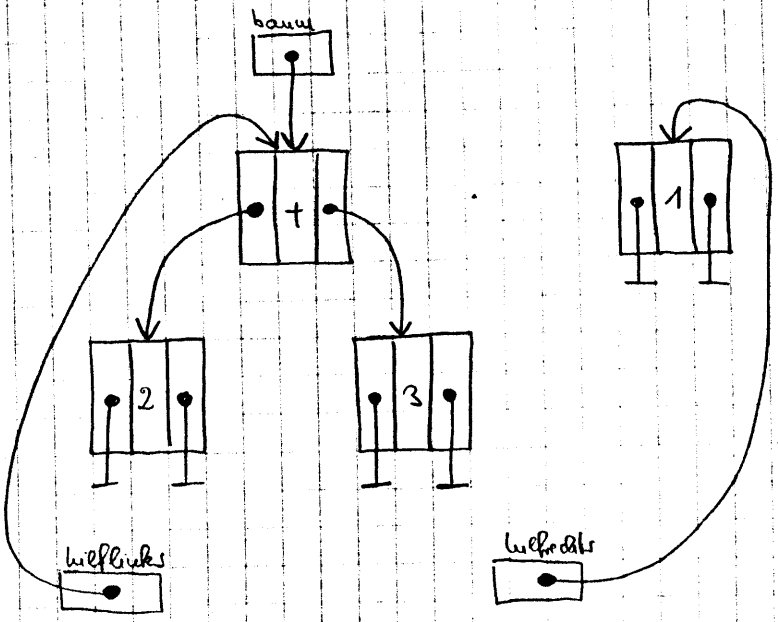


entspricht: construct(hilflinks, +, hilfrechts)

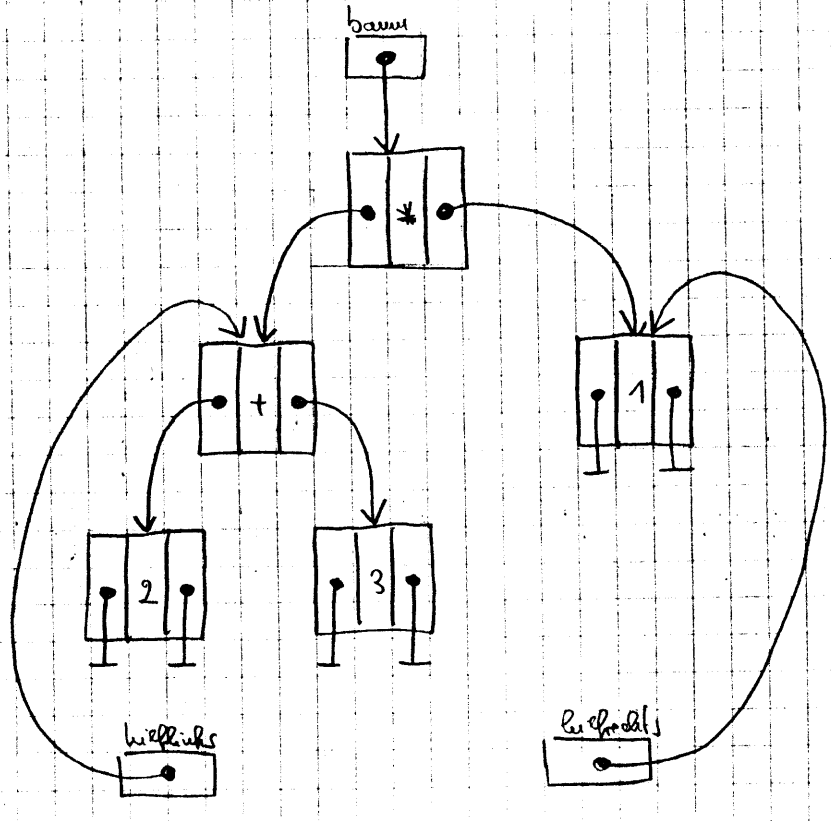
```
NEW (baum);
WITH baum↑ DO
        links := hilflinks;
        inhalt := "+";
        rechts := hilfrechts
END;
```



```
hilfLinks := baum;
NEW(hilfRechts);
WITH hilfRechts↑ DO
        links := NIL;
        inhalt := "1";
        rechts := NIL
END;
```




```
NEW(baum);
WITH baum↑ DO
        links := hilfLinks;
        inhalt := "*";
        rechts := hilfRechts
END
```



Die genauen Regeln von MODULA-2 für Zeigertypen und dynamische Variable sind wie folgt gegeben:

Datentyp T-Zeiger:

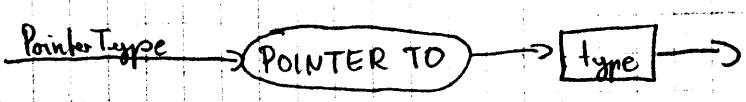
Sei T ein Typ:

Wertemenge: Menge der Zeiger (interne, systemverwaltete Bezeichner) auf Werte des Typs T

sowie

der (nicht typgebundene) Nullzeiger, dessen Standard-bezeichnung NIL ist

diese Wertemenge wird vereinbart mit:



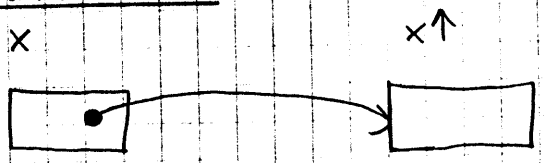
Operationen: als einzige Operationen stehen zur Verfügung:

Gleichheit `=` : POINTER TO T × POINTER TO T → {FALSE, TRUE}

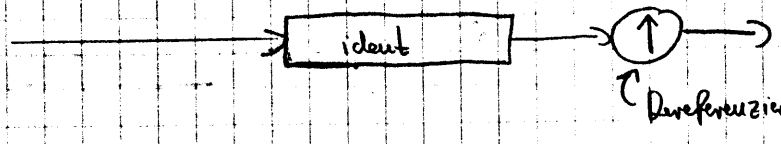
Ungleichheit `#` : POINTER TO T × POINTER TO T → {FALSE, TRUE}

Variable für den Datentyp T-Zeiger nennt man auch T-Zeigervariable.
Ist x eine T-Zeigervariable, die nicht den Nullzeiger bezeichnet,

so steht $x\uparrow$ für diejenige dynamische Variable (für den Datentyp T), auf die der durch x bezeichnete Zeiger verweist; $x\uparrow$ nennt man eine referenzierte Variable:



Wir können die Definition von Variablen (S. 24.9) ergänzen:



In Ausdrücken, vgl. Def. von Faktor, dürfen als Variable auch referenzierte Variable vorkommen. Der Wert von Ausdruck $x\uparrow$ wird wie folgt ermittelt:

- dereferenziere x, d.h. ermittle den durch x bezeichneten Zeiger;
- falls dies der Nullzeiger ist, so ist der Wert von $x\uparrow$ undefiniert;
  andernfalls bezeichnet x einen internen, systemverwalteten Bezeichner, der wiederum einen Wert a bezeichnet: dann ist a der Wert des Ausdrucks $x\uparrow$.

Referenzierte Variable dürfen auch vorkommen

- auf der linken Seite einer Wertzuweisung
  (durch $x\uparrow := a$ wird ein interner, systemverwalteter Bezeichner, d.h. der durch x bezeichnete Zeiger, zur Bezeichnung von a verwendet)
- als aktuelle (Argument-Ergebnis-) Parameter für Prozedur- bzw. Funktionsaufrufe
  (als aktuelle Argument-Parameter dürfen sie in ihrer Eigenschaft als Ausdruck vorkommen)

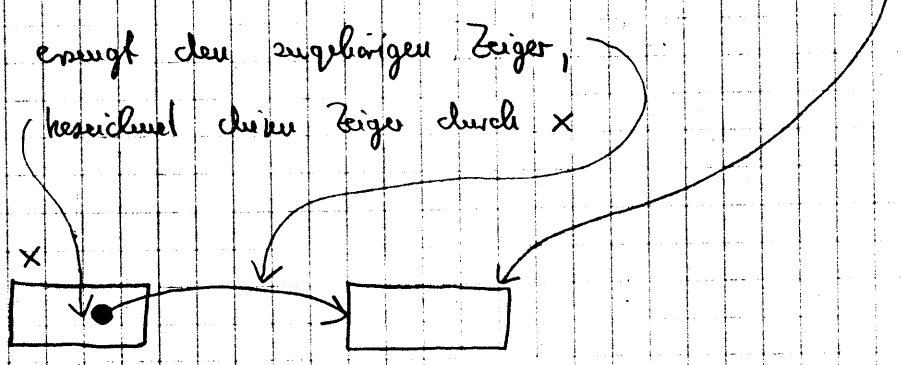
In allen Fällen müssen referenzierte Variable vom passenden Typ sein.

Zur gemeinsamen Erzeugung einer dynamischen Variablen und des zugehörigen internen, systemverwalteten Bezeichners (Zeiger) dient die (in Form eines Aufrufs einer Standardprozedur vorliegende) Anweisung

NEW(x):

der Aufruf NEW(x), wobei x eine T-Zeigervariable sei,

- stellt neuen Speicherplatz für eine neue dynamische Variable vom Typ T bereit,
- erzeugt den zugehörigen Zeiger,
- bezeichnet diesen Zeiger durch x



Bemerkung: Während Zahlen, Wahrheitswerte, Zeichen und Zeichenfolgen als "von vornherein vorhanden" gedacht werden, müssen Zeiger erst vom Programm "erzeugt" werden. Anschließend kann man mit Zeigern nur auf eine einzige Art operieren: man kann sie weitergeben mit Hilfe der Wertzuweisung.

Speicherplatz für dynamische Variablen kann wieder freigegeben werden durch den Aufruf DISPOSE(x):

der Speicherplatz von x↑ wird nicht mehr benötigt.

Achtung: Falls für eine andere Variable y (die gleiche Variable ist) x↑ = y↑, so ist die Variable y anschließend unbekannt!

Benutzt man y, so erscheint eine völlig Unverständliches!

Für variante Verbunde gibt es noch eine Sonderform von NEW bzw. DISPOSE.

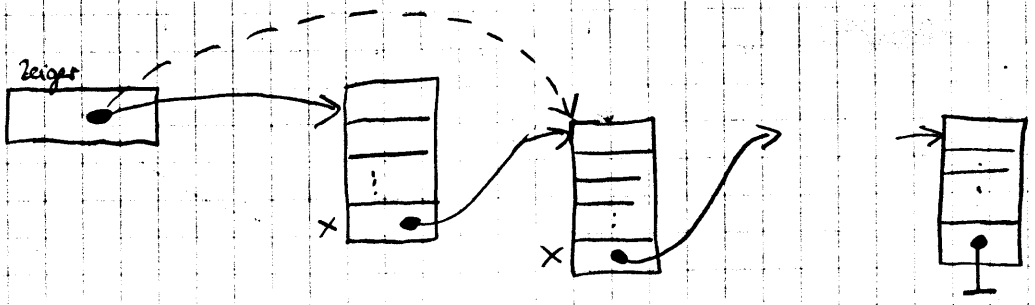
Achtung: NEW und DISPOSE verlangen manchmal geeignete Typanweisungen!
z.B. (siehe Wirth, p. 75) FROM Storage IMPORT ALLOCATE, DEALLOCATE;

Baut man sich "strukturierte Datentypen" auf Hilfe von Zeigern und dynamischen Variablen auf, so ist die entsprechende "strukturierte Anweisung" häufig

- die Wiederholungsanweisung mit WHILE, nach dem Muster: WHILE zeiger # NIL DO ... zeiger := zeiger↑.x ... END;
- der Aufruf einer rekursiven Prozedur, nach dem Muster:

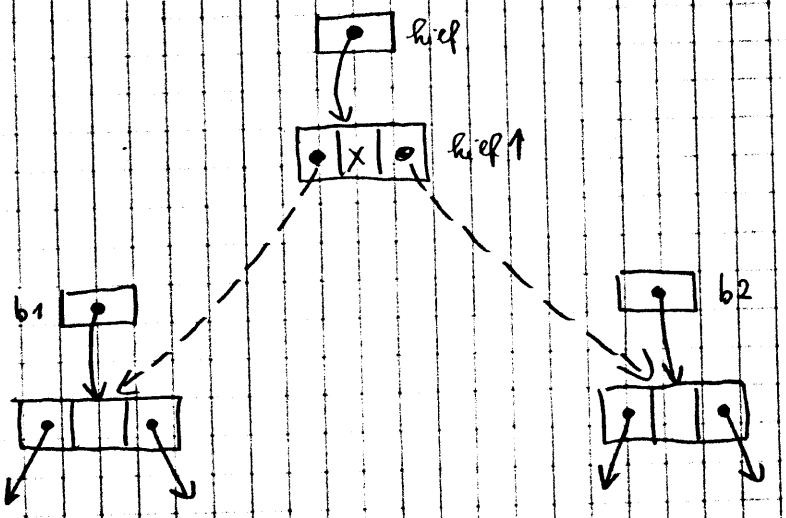
```
PROCEDURE arbeiten (zeiger: POINTER TO T; ...);
  :
  BEGIN IF zeiger # NIL THEN ... arbeiten(zeiger↑.x, ...)
  END arbeiten;
```

NEW(p) ~ ALLOCATE(p, TSIZE(T))
DISPOSE(p) ~ DEALLOCATE(p, TSIZE(T))
mit VAR p: POINTER TO T



Beispiel: "Bäume", nur Struktur spezifiziert

```
(* leer wird durch NIL dargestellt *)
TYPE   grundwerte = CHAR;
       baum       = POINTER TO knoten;
       knoten     = RECORD  links : baum;
                            inhalt : grundwerte;
                            rechts : baum
                    END;

PROCEDURE construct (b1: baum; x: grundwerte; b2: baum): baum;
VAR hilf: baum;
BEGIN NEW (hilf); WITH hilf↑ DO
                  links := b1; inhalt := x; rechts := b2 END;
      RETURN hilf
END construct
```



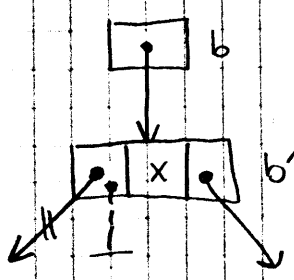
```
PROCEDURE istLeer (b: Baum): BOOLEAN;
BEGIN RETURN b = NIL
END istLeer;

PROCEDURE gleich (b1, b2: Baum): BOOLEAN;
BEGIN
  IF (b1=NIL) AND (b2=NIL)
     THEN RETURN TRUE
  ELSIF (b1=NIL) OR (b2=NIL)    (* aber (b1 # NIL) OR (b2 # NIL) *)
     THEN RETURN FALSE
  ELSE  (* (b1 # NIL) AND (b2 # NIL) *)
        RETURN   gleich (b1↑.links, b2↑.links)
             AND (b1↑.inhalt = b2↑.inhalt)
             AND gleich (b1↑.rechts, b2↑.rechts)
  END
END gleich;

PROCEDURE linkerAst (b: Baum): Baum;
BEGIN IF b # NIL THEN RETURN b↑.links
                 ELSE WriteLn; WriteString ("Fehler: linkerAst von NIL");
                      RETURN NIL
      END
END linkerAst;
```

analog wie linkerteil

```
PROCEDURE rechterteil (b: kante) : kante ;
BEGIN  IF  b # NIL  THEN  RETURN  b↑. rechts
                    ELSE  WriteLn; WriteString ("Fehler: rechterteil nicht definiert");
                          RETURN NIL
       END
END rechterteil ;
```

```
PROCEDURE wert (b: kante) : grundwerte ;
BEGIN  IF  b # NIL  THEN  RETURN  b↑. inhalt
                    ELSE  WriteLn; WriteString ("Fehler: wert nicht definiert");
                          RETURN "?"
       END
END wert ;
```

```
PROCEDURE entfernelinks (b: kante) : kante ;
BEGIN  IF  b # NIL
           THEN  b↑. links := NIL ; RETURN b
       ELSE      WriteLn; WriteString ("Fehler: entfernelinks nicht definiert");
                 RETURN NIL
       END
END entfernelinks ;
```



(definiert, falls b # NIL)

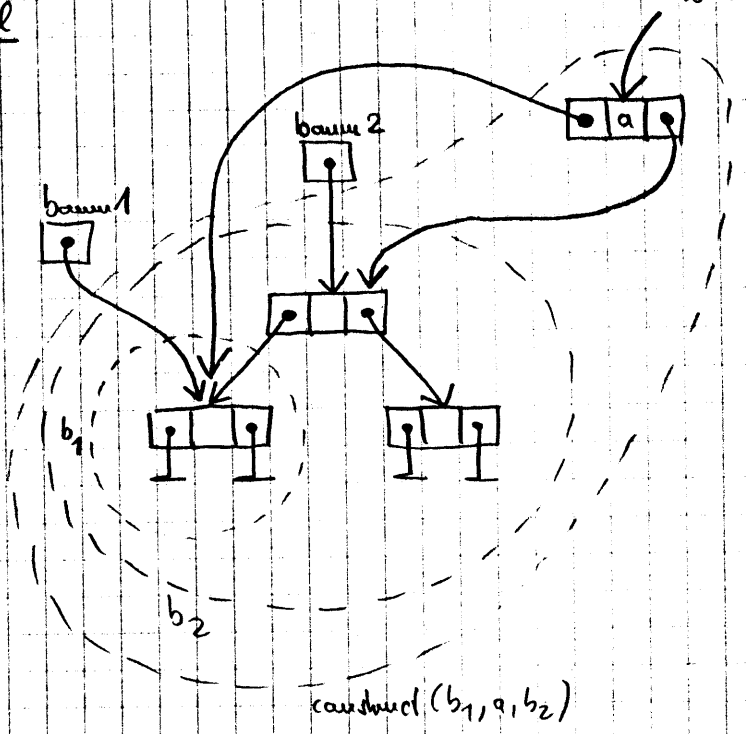
PROCEDURE entfernerechts ....
PROCEDURE änderewert ....
} jeweils analog!

Achtung

1) Diese Implementierung stellt nicht sicher, daß die Teilbäume eines Baumes "disjunkt" sind!

Beispiel

construct(baum1, "a", baum2)



construct(b_1, a, b_2)

Das bedeutet: mit Hilfe von leer und construct können wir tatsächlich erzeugen: die Menge der bewerteten, binären, gerichteten, azyklischen Graphen!

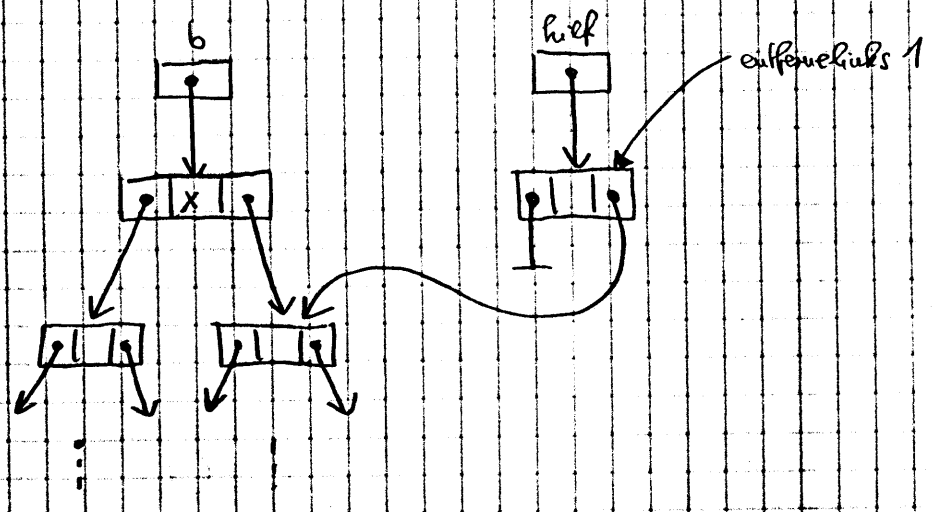
Das ist eine typische Schwierigkeit, die bei der Spezifikation von Datentypen auftaucht:

1) unsere Axiome erfüllen alle Identitäten der Spezifikation.

2) Diese Identitäten erzwingen aber nicht schon alle "gewünschten Eigenschaften", d.h. hier die Disjunktheit der Teilbäume!

2) Ein Aufruf von entfernelinks, entfernerechts oder anderewert ändert jeweils den Baum, auf den der aktuelle Parameter zeigt. Falls das nicht erwünscht ist, so kann man z.B. folgende Axiome verwenden:

```
PROCEDURE  entfernelinks1 (b: kante) : kante;
VAR  hilf : kante;
BEGIN
  IF  b # NIL
  THEN   NEW (hilf);
         WITH  hilf↑ DO    links := NIL;
                           inhalt := b↑. inhalt;
                           rechts := b↑. rechts
                       END
         RETURN  hilf
  ELSE   WriteLn; WriteString ("Fehler: entfernelinks1 nicht definiert");
         RETURN  NIL
  END
END  entfernelinks1;
```



Will man nun den Datentyp "Bäume" (oder genauer bewertete, binäre, gerichtete, azyklische Graphen) anderen Benutzern verfügbar machen, so kapselt man ihn am besten mit Hilfe des Modul-Konstrukts ab, etwa so:

opaque (dunkel, undurchsichtig): wird erst im Implementationsteil definiert.

```
DEFINITION MODULE Baeume;
TYPE  gewichte = CHAR;        (* Knotenbewertungen *)
      Baum;                   (* steht für Bäume *)

PROCEDURE construct (b1: Baum; x: gewichte; b2: Baum): Baum;
   (*...          *)
PROCEDURE istLeer (b: Baum): BOOLEAN;
   (*...          *)
   .
   .
   .
END Baeume.
```

hier wird der opaque-Typ erst genau (und absichtlich) definiert!

```
IMPLEMENTATION MODULE Baeume;
TYPE  Baum = POINTER TO Knoten
      Knoten = RECORD  links: Baum;
                       inhalt: gewichte;
                       rechts: Baum
                END;
PROCEDURE construct (b1: Baum; x: gewichte; b2: Baum): Baum;
VAR hilf: Baum;
   .
   .
END construct;
   .
   .
   .
END Baeume.
```

Aufgabe: Durchlaufe einen Baum in postorder-Reihenfolge nach der

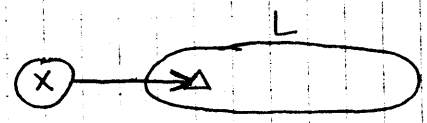
Regel: – Durchlaufe linken Teilbaum
– Durchlaufe rechten Teilbaum
– besuche Wurzel

```
MODULE Baumverarbeiten;
FROM Baeume IMPORT isleer, linksteil, rechtsteil, inhalt, baue;
FROM InOut IMPORT Write, WriteLn;
;
VAR anker : baum;
;
PROCEDURE postorder (p: baum);
  BEGIN IF NOT isleer (p)
        THEN postorder (linksteil (p));
             postorder (rechtsteil (p));
             Write (wert (p))
        END
  END postorder;
;
postorder (anker); WriteLn;
;
END Baumverarbeiten.
```

als weiteres Beispiel behandeln wir Listen:

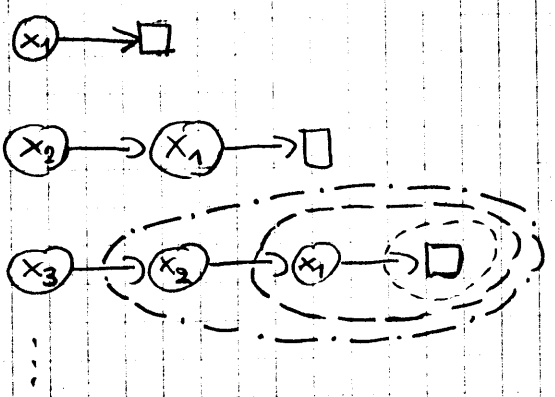
Wertemenge: – $\square$ sei "leere" Liste mit Kopf $\square$ und innerer Knotenmenge $ikn(\square) = \emptyset$

– ist $x \in W$ und ist L eine Liste mit Kopf Δ und innerer Knotenmenge $ikn(L)$, so sei



eine Liste mit Kopf $\otimes$ und innerer Knotenmenge $\{\otimes\} \cup ikn(L)$

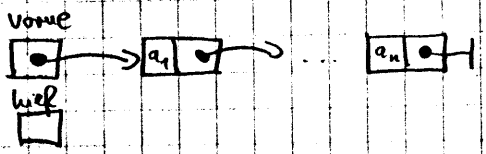
Bsp $\square$



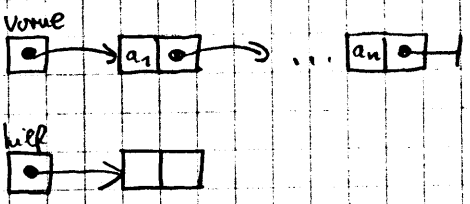
Operationen: (zum Beispiel)

1) füge $\otimes$ als neuen Kopf vorne an
2) ändere alle inneren Knoten $\otimes$ in Ψ
3) drucke Liste.

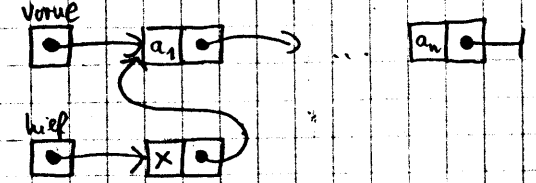
```
MODULE listen;
FROM InOut IMPORT Read, Write, WriteLn, Done;
TYPE  grundwerte = CHAR;
      kante      = POINTER TO knoten;
      knoten     = RECORD inhalt : grundwerte;
                          nachfolger : kante
                   END;
VAR   zeichen : grundwerte;
      listenanker : kante;

PROCEDURE einfügen (x: grundwerte; VAR vorne : kante);
VAR hilf: kante;
```



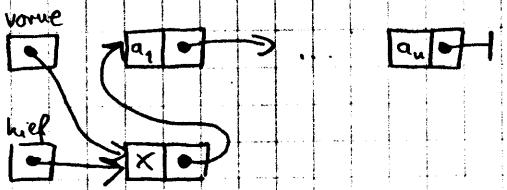

```
BEGIN NEW(hilf);
```




```
      hilf↑.inhalt := x;
      hilf↑.nachfolger := vorne;
```

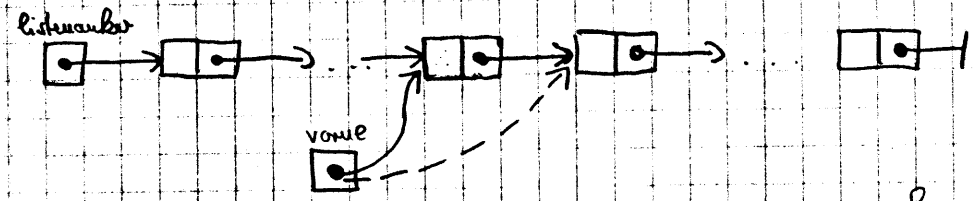



```
      vorne := hilf
```




```
END einfügen;
```

```
PROCEDURE aendern(alt, neu: grundwerte ; vorne : kante );
BEGIN   WHILE   vorne # NIL   DO
                IF  vorne ↑. inhalt = alt THEN vorne ↑. inhalt := neu ;
                vorne := vorne ↑. nachfolger
        END
END aendern ;
```



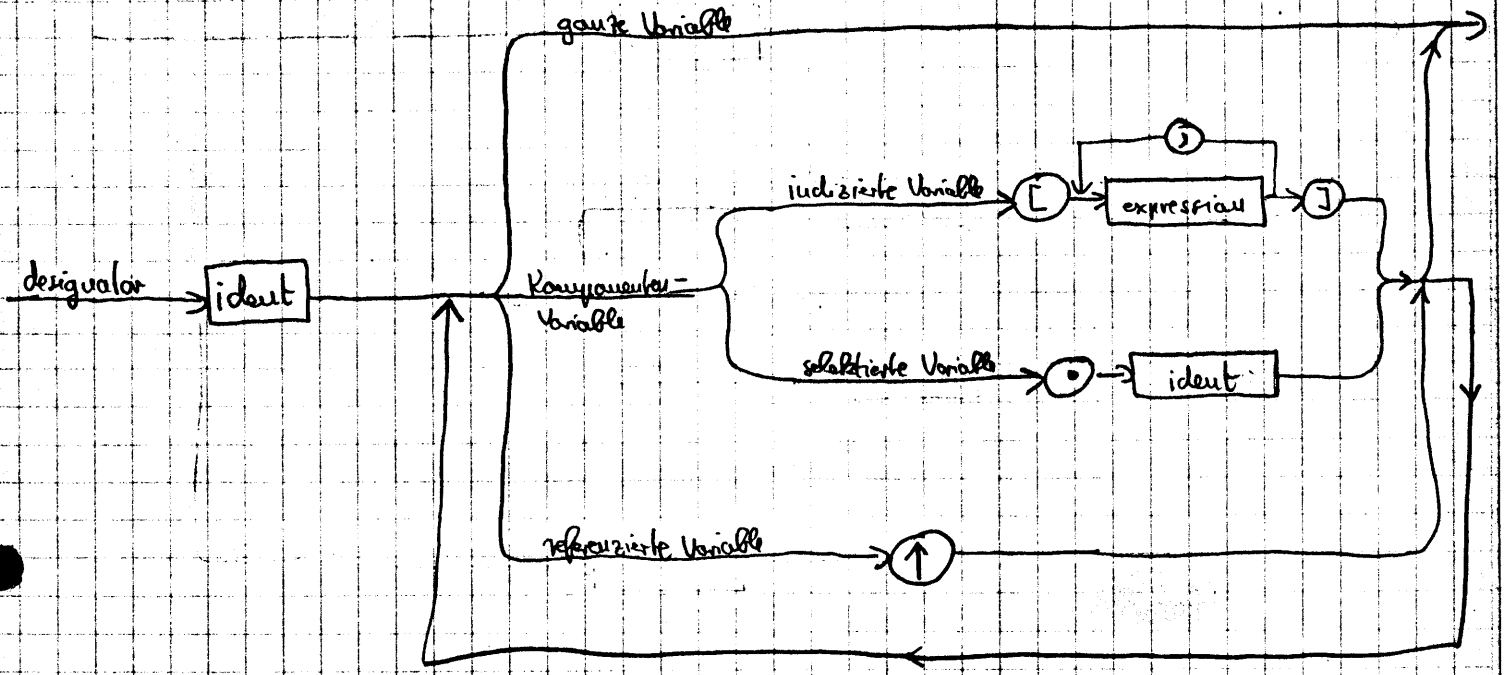
Bem:
aendern und drucken können natürlich auch rekursiv definiert werden entsprechend dem Ansatz
- bearbeite Kopf
- bearbeite Rest

```
PROCEDURE   drucken (vorne : kante);
BEGIN   WHILE   vorne # NIL   DO
                Write ( vorne ↑. inhalt );
                vorne := vorne ↑. nachfolger
        END
END drucken ;

BEGIN
  listenanker := NIL ; Read ( zeichen );
  WHILE   Done   AND   ( zeichen # "␣" )   DO
      einfügen ( zeichen, listenanker );
      Read ( zeichen )
  END ;
  ändern ( "a" , "z" , listenanker );
  drucken ( listenanker );
  WriteLn
END listen.
```

28 Variable

Im Laufe der Vorlesung haben wir öfter von "Variablen" bzw. "Designatoren" gesprochen. In diesem Kapitel soll noch einmal einiges davon zusammengefaßt werden:

1) MODULA-2 - Syntax:



Die Anwendung

- des Komponenten-Zugriffs vermöge { Indizes / eines Selektors }, d.h. durch [,] bzw. .
- des Dereferenzierungs-Operators, d.h. von ↑

setzt voraus, daß die entsprechende Variable von passendem Typ ist.

2) Der Typ einer Variablen wird festgelegt bei ihrer Erzeugung:

eine lokale Variable wird erzeugt durch einen Prozeduraufruf:
für alle lokalen Bezeichner (der Prozedur) wird eine Variable erzeugt, deren Typ durch die entsprechende Typangabe festgelegt ist;

eine dynamische Variable wird erzeugt durch einen Aufruf von NEW(x);
ihr Typ ist festgelegt durch den Typ von x: ist x vom Typ POINTER TO T, so ist die erzeugte dynamische Variable vom Typ T.

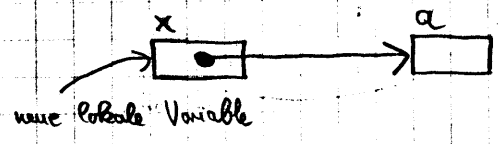
3) Variable werden maschinenmäßig als Repräsentanten von Speicherplatz gedeutet.

4) Variable, als Objekte aufgefaßt, werden bezeichnet: (auch formale Argument-Parameter)

bei lokalen Variablen: 1) durch den entsprechenden lokalen Bezeichner im Zusammenhang mit der dynamischen Kette,
2) durch formale Argument-Ergebnis-Parameter gemäß folgendem Muster:

```
PROCEDURE p ( VAR x : T, ... ); ... END p;
```

Die Wirkung eines Aufrufs der Form p(a,...) kann dann (implementierungsmäßig) gedeutet werden durch:

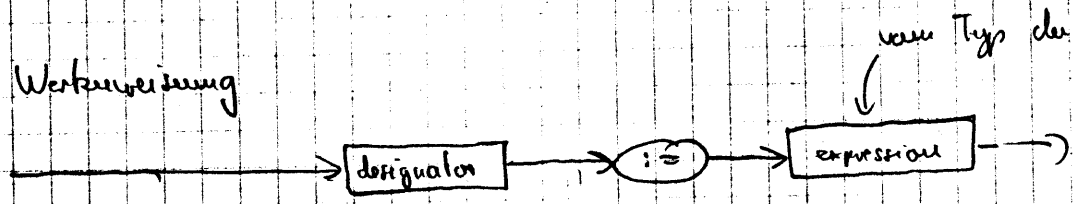


d.h. für die Variable a wird ein interner, systemverwalteter Bezeichner (Zeiger) erzeugt und x zugewiesen; jedes Vorkommen von x im Rumpf wird dann als x↑ gedeutet!

bei <u>dynamischen Variablen</u>: durch den bei ihrer Erzeugung gleichzeitig mit erzeugten, internen, systemkontrollierten Bezeichner (Zeiger).

5) Variable können in MODULA-2 auf folgende Weisen verwendet werden:

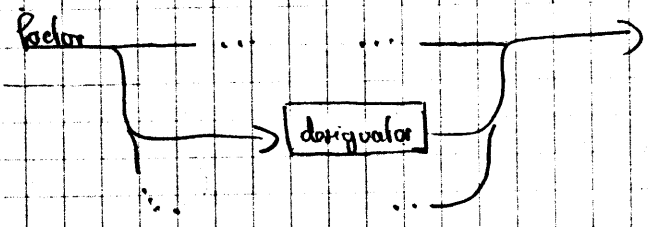
a) zur vom Programmierer ausdrücklich vorgenommenen Bezeichnung von "Ergebniswerten" und deren zwischenzeitlicher Speicherung:



Variable, als Speicherplatz gedacht, können einen Wert "aufnehmen".

b) zum Wiederauffinden von Werten:

im Faktor (als Bestandteil von Ausdrücken):



Variable, als Speicherplatz gedacht, liefern als Wert ihren "Inhalt".

6) Variable werden <u>maschinenfern</u> als Bezeichner für Objekte <u>gedeutet</u>:

der Programmierer in MODULA-2 muß selbst sicherstellen, daß tatsächlich Objekte bezeichnet werden, nämlich durch <u>geeignete Initialisierungen</u>
(in MODULA-2 werden nur durch formale Parameter bezeichnete Variable vom System initialisiert)

7) Bei einem Aufruf einer Prozedur sollen die aktuellen Parameter für Argument- Ergebnis- Parameter <u>paarweise verschiedene Variable</u> sein.
Analog müssen in der (in MODULA-2 nicht vorliegenden) kollektiven Wertzuweisung der Form $(v_1, .., v_n) := (e_1, .., e_n)$ auf der linken Seiten <u>paarweise verschiedene Variable</u> stehen.

<u>Vorsicht:</u> Variable sind nicht immer schon verschieden, wenn ihre Bezeichnungen (textuell) verschieden sind!

<u>Bsp:</u> 1) Sei x ein Feldbezeichner:
x[i] und x[j] bezeichnen die gleiche Variable, falls i und j den gleichen Indexwert bezeichnen

2) Seien x und y "T-Zeigervariablen":
nach einer Wertzuweisung x := y bezeichnen
x↑ und y↑ die gleiche Variable.

3) Sei PROCEDURE p(VAR x: , ...) vereinbart und erfolgt ein Aufruf p(a, ..), wobei a als globale Variable im Rumpf von p "sichtbar" sei.

Dann bezeichnen während der Ausführung a und x die gleiche Variable.

Die beiden letzten Beispiele verdeutlichen insbesondere eine Gefahr beim Umgang mit Zeigern.

29. Nachwort: die Grenzen des Programmierens

1) durch Theorie gefundene Grenzen

- §5: Es gibt Funktionen $f: X^* \to X^*$, die grundsätzlich nicht berechenbar (durch eine Maschine) sind; d.h. jeder Versuch, "sie zu programmieren" muß scheitern

- (⇒ Berechenbarkeits- und Komplexitätstheorie):
  Es gibt Funktionen $f: X^* \to X^*$, die zwar grundsätzlich berechenbar sind, aber für die alle denkbaren Programme einen Aufwand an Zeit und Speicherplatz erfordern, der außerhalb der (physikalischen) Möglichkeiten unserer Welt liegt.

  genauer: sei $g: \mathbb{N} \to \mathbb{N}$ eine vorgegebene (schnellwachsende) Aufwands-Schrankenfunktion

  es gibt $f: X^* \to X^*$ mit der Eigenschaft:

  f: alle Programme P mit $\sigma(P) = f$ gilt

  $$\text{Aufwand}(P, x) \geq g(\text{Länge}(x)) \quad \begin{cases} \text{für fast alle } x \in X^* \\ \text{für unendlich viele } x \in X^* \end{cases}$$

mögliche Auswege:

- für "interessante Spezialfälle", d.h. $Y \subsetneq X^*$, schnellere Programme möglich
- statt f eine "geeignete Abänderung" $\tilde{f}$, die leichter zu berechnen ist, verwenden

2) durch das Gebot der Beherrschbarkeit gezogene Grenzen

Erinnerung an §2: Wie kann man überprüfen, daß ein Programm eine gegebene Klasse von Zustandsüberführungen beschreibt?

eine Antwort: systematisches Programmieren, Verifikation

aber: gegeben: wünschenswert, aber häufig wirklichkeitsfern für sehr große Programmsysteme

überprüfen: nur bedingt tatsächlich durchführbar

manchmal eine andere Antwort:

Beschränktheit: nichtbeherrschbare Programme sind nicht zu verantworten und dürfen deshalb nicht eingesetzt werden.

3) durch Gesetz verordnete Grenzen

- Der "Lebensbereich" Programmieren (Datenverarbeitung) ist Anfang nur vereinzelt gesetzlich geregelt; der Gesetzgebung eilt hinter der stürmischen technischen Entwicklung her.

wichtigste rechtliche Regelungen:

- Bundesdatenschutzgesetz
- sog. bereichsspezifische Regelungen, z.B. X. Sozialgesetzbuch
- Verfassungsgerichtsurteil über die Volkszählung

alle diese Regelungen haben (vor allem) den Schutz des Einzelnen vor dem Mißbrauch staatlicher oder wirtschaftlicher Macht zum Ziel.

4) selbst auferlegte Grenzen

Hinter den Forderungen der Beherrschbarkeit und der Gesetzestreue steht ein ethischer Anspruch, mit dem sich Techniker / Wissenschaftler stets auseinandersetzen müssen:

negative Formulierung:

Der Mensch darf nicht alles verwirklichen, was technisch machbar ist.

positive Formulierung:

Der Mensch soll das verwirklichen, was ihm (der Menschheit) ein würdiges Leben ermöglicht.