1.3.2.5 Folgenbildung (vgl. 1.1.3.4 und 1.1.4.6)

Zu jeder Menge M kann man die Menge M* der endlichen Folgen (auch Menge der Wörter über M genannt) bilden:
$M^* = \{a_1 a_2 \ldots a_n \mid n \geq 0 \text{ und } a_i \in M \text{ für } i = 1, 2, \ldots, n\}$.
M möge zum Datentyp T gehören. Wir definieren den Datentyp seq of T durch folgende Festlegungen:
Zugrunde liegende Menge: M*
Nullstellige Operationen: Alle Elemente von M*. Man schreibt diese Wörter auf, indem man sie entweder in Anführungsstriche setzt und die Elemente von M durch Zwischenräume trennt oder indem man sie als Vektoren $(a_{i_1}, a_{i_2}, \ldots, a_{i_n})$ notiert. Das leere Wort erhält hierbei die Darstellung ε oder ( ). Manchmal schreibt man auch null oder nil für das leere Wort.
Wir werden meist die Vektorschreibweise verwenden.

Einstellige Operationen

Für empty, square, removefirst, removelast: M* → M* gilt:
empty(u) = ε für alle u ∈ M*,
square$((a_{i_1}, a_{i_2}, \ldots, a_{i_n})) = (a_{i_1}, a_{i_2}, \ldots, a_{i_n}, a_{i_1}, a_{i_2}, \ldots, a_{i_n})$,
removefirst$((a_{i_1}, a_{i_2}, \ldots, a_{i_n})) = (a_{i_2}, \ldots, a_{i_n})$,
removelast$((a_{i_1}, a_{i_2}, \ldots, a_{i_n})) = (a_{i_1}, a_{i_2}, \ldots, a_{i_{n-1}})$.
Für das leere Wort sind removefirst und removelast undefiniert.

first, last: M* → M sind definiert durch
first$((a_{i_1}, a_{i_2}, \ldots, a_{i_n})) = a_{i_1}$, sofern n > 0; first(( )) ist undefiniert,
last$((a_{i_1}, a_{i_2}, \ldots, a_{i_n})) = a_{i_n}$, sofern n > 0; last(( )) ist undefiniert.

Statt "first" schreibt man meist "head", statt removefirst "tail".

Einstellige Operationen (Fortsetzung)

in: M → M* mit in(b) = (b), für alle b∈ M (Einbettung).

length: M* → $\mathbb{N}_0$ ist definiert durch
length$((a_{i_1}, a_{i_2}, \ldots, a_{i_n})) = n$; speziell gilt length(( )) = 0.
Für jedes a ∈ M ist $\#_a$: M* → $\mathbb{N}_0$ die Anzahlfunktion für das Element a, d.h., $\#_a(u)$ = Anzahl, wie oft a in u vorkommt.
Üblicherweise definiert man $\#_a$ rekursiv:
$\#_a(( )) = 0$ und für alle u∈M* und alle b∈M:
$\#_a(ub) = \#_a(u) + 1$, falls a = b ist,
$\#_a(ub) = \#_a(u)$, falls a ≠ b ist,

isempty: M* → $\mathbb{B}$ ist definiert durch
isempty(u) = true genau dann, wenn u das leere Wort ist.

Zweistellige Operationen

conc: M* × M* → M* (Konkatenation) ist definiert durch
conc$((a_{i_1}, a_{i_2}, \ldots, a_{i_n}), (b_{j_1}, b_{j_2}, \ldots, b_{j_m})) = (a_{i_1}, \ldots, a_{i_n}, b_{j_1}, \ldots, b_{j_m})$
append: M* × M → M* (Anhängen eines Elements) ist definiert durch append$((a_{i_1}, a_{i_2}, \ldots, a_{i_n}), b) = (a_{i_1}, \ldots, a_{i_n}, b)$.

Die Vergleichsoperationen =, ≠: M* × M* → $\mathbb{B}$ sind wie üblich definiert. Falls M eine geordnete Menge ist, so kann man auch die anderen Vergleichsoperationen <, ≤, > und ≥ verwenden.
(Man achte aber genau darauf, wie die Ordnung von M auf M* fortgesetzt wurde! Beispiel: lexikografisch oder längenlexikografisch.)

Beziehungen, Gesetzmäßigkeiten

Beispiele für Gesetzmäßigkeiten sind (stets für alle u∈ M*):
removelast(append(u, b)) = u für alle b∈ M.
conc(u, u) = square(u).

$$\text{length}(u) = \sum_{a \in M} \#_a(u).$$

not isempty(append(u,b)) für alle b∈ M.

length(conc(u, v)) = length(u) + length(v) für alle v∈ M*.

first(in(b)) = last(in(b)) = b für alle b∈ M.

Die Datenstruktur "seq of T" in der Sprache Ada

Folgen werden in Ada durch "Listen" dargestellt, die mittels Zeigern (also mit ACCESS - Datentypen) realisiert werden.
Für die Operationen müssen geeignete Prozeduren und Funktionen geschrieben werden, sofern nicht eine vordefinierte Klasse für die "Listenverarbeitung" existiert.
Ein Spezialfall sind Folgen über dem Latin-1-Alphabet. In Ada gibt es hierfür den vordefinierten Datentyp "string".

1.3.2.6 Potenzmengen

Zu jeder Menge M kann man die Menge 2^M ihrer Teilmengen konstruieren: $2^M = \{ M' \mid M' \subseteq M\}$. Wegen $\emptyset \in 2^M$ und $M \in 2^M$ ist die Potenzmenge einer Menge niemals leer.

M möge zum Datentyp T gehören. Wir definieren den Datentyp set of T durch folgende Festlegungen:

Zugrunde liegende Menge: 2^M

Nullstellige Operationen: Wichtige "Konstanten" sind die leere Menge $\emptyset$ und die gesamte Menge M. Grundsätzlich sollte man jede endliche Teilmenge von M notieren können.

Eine Teilmenge M' schreibt man auf, indem man ihre Elemente auflistet oder indem man einen |M|-stelligen Booleschen Vektor b': array T of Boolean benutzt mit: $b'(m) = true \Leftrightarrow m \in M'$.

(Andere Darstellungsmöglichkeiten lernen wir noch kennen.)

Einstellige Operationen

compl: $2^M \to 2^M$ (Komplement einer Teilmenge) mit $compl(M') = \{m \in M \mid m \notin M'\} \in 2^M$.

isempty: $2^M \to \mathbb{B}$ mit $isempty(M') = true \Leftrightarrow M' = \emptyset$

Zweistellige Operationen

Durchschnitt $\cap$, Vereinigung $\cup$, Differenz $\setminus$: $2^M \times 2^M \to 2^M$ sind wie üblich definiert, vgl. Einschub in 1.1.2.

Auch die Vergleichsoperationen und die Elementbeziehung $=, \neq, \subset, \subseteq, \supset, \supseteq, \not\subset$: $2^M \times 2^M \to \mathbb{B}$ und $\in, \notin : M \times 2^M \to \mathbb{B}$ sind zweistellige Operationen.

Die Datenstruktur "set of T" ist in der Sprache Ada nicht direkt vorgesehen. Man muss sie mit anderen Strukturen simulieren.

1.3.2.7 Abbildungen (vgl. 1.1.3.4 und 1.1.4.6)

N^M oder mit $Abb(M,N) = \{f \mid f: M \to N\} = N^M$.

Wir müssen unterscheiden zwischen der Menge der totalen N^M Abbildungen von M nach N und der Menge aller Abbildungen (also auch der partiellen Abbildungen, die nicht auf dem gesamten Vorbereich M definiert sind sondern nur auf einer Teilmenge). In der Programmierung geht man meist von der Menge aller Abbildungen aus. Um anzugeben, dass die Funktionen auch partiell sein können, fügt man oft ein "p" (wie "partiell") an: $Abb_p(M,N) = \{f \mid f: M \to N\}$.

M möge zum Datentyp T, N zum Datentyp U gehören. Wir definieren den Datentyp mapping (T,U) durch folgende Festlegungen:

Zugrunde liegende Menge: $Abb_p(M,N)$

Nullstellige Operationen: Dies sind feste Abbildungen z.B. der Sinus sin: $\mathbb{R} \to \mathbb{R}$ oder der größte gemeinsame Teiler ggT: $\mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ oder die überall undefinierte Funktion $f_\perp$: $M \to N$ mit: $f_\perp(m)$ ist undefiniert für alle $m \in M$. (Man realisiert $f_\perp$ leicht durch eine unendliche Schleife.)

Einstellige Operationen: Dies sind Abbildungen der Art $F: Abb_p(M,N) \to Abb_p(M,N)$.

Im Falle M= N ist ein wichtiges Beispiel hierfür "$^{-1}$", also die Bildung der Inversen, das heißt, wenn $f: M \to M$ eine injektive Abbildung ist, so ist $f^{-1}: M \to M$ die Abbildung: $f^{-1}(b) = a$, falls $f(a) = b$, und undefiniert sonst.

Einstellige Operationen (Hinweis)

Ein Beispiel für einstellige Funktionsabbildungen sind "Ableitung" und "Stammfunktion", d.h., wenn $M = N = \mathbb{R}$, dann sind das unbestimmte Integral und die abgeleitete Funktion partielle Abbildungen auf der Menge der reellwertigen Funktionen, also z.B.:

$\int$: $Abb_p(\mathbb{R},\mathbb{R}) \to Abb_p(\mathbb{R},\mathbb{R})$.

Partiell ist das Integral, weil nicht zu jeder reellwertigen Abbildung eine Stammfunktion existiert (in der Regel verlangt man Stetigkeit als Voraussetzung). Ebenso lässt sich nicht jede reellwertige Funktion differenzieren.

Zweistellige Operationen:

Hier sind vor allem zu nennen die Hintereinanderausführung

H: $Abb_p(M,M) \times Abb_p(M,M) \to Abb_p(M,M)$

mit $H(f,g) = fg$ und die Iteration

I: $Abb_p(M,M) \times \mathbb{N}_0 \to Abb_p(M,M)$

mit $I(f,n) = f^n$ (vgl. Folie 146 in Abschnitt 1.1.3).

Realisierung in Programmiersprachen: In der Sprache ALGOL 68 war dieser Datentyp vorgesehen, er lässt sich jedoch schwer implementieren. In funktionalen Sprachen (LISP usw.) lassen sich dagegen Abbildungen gut darstellen und manipulieren.

In Ada ist nur die "nullstellige Operation" vorgesehen, d.h., man kann einzelne Funktionen definieren. Allerdings kann man dies durch das generic-Konzept etwas ausweiten.

Generelle Bemerkung: *(Durchdenken Sie dies selber.)*

Alle Operationen, die auf M oder auf N definiert sind, kann man auf Funktionen ausdehnen. Wenn beispielsweise auf N eine Operation $\circ$: $N \times N \to N$ definiert ist, so kann man diese Operation fortsetzen zu

$\bullet$: $Abb_p(M,N) \times Abb_p(M,N) \to Abb_p(M,N)$

durch $\bullet(f,g) = f \circ g$. Genauer: für alle $a \in M$ setze

$(\bullet(f,g))(a) = f(a) \circ g(a)$.

Beispiele für solche Operationen $\circ$ sind Addition, Subtraktion, Multiplikation, Maximum und Minimum usw.

Man kann auch Vergleiche von M auf $Abb_p(M,M)$ übertragen, z.B.: $f \leq g \Leftrightarrow$ für alle $a \in M$ gilt $f(a) \leq g(a)$. Hierbei geht eine lineare Ordnung jedoch in der Regel nur in eine partielle Ordnung über.

1.3.2.8 Menge der Namen

In jedem Programm gibt es noch eine spezielle Menge, nämlich die Menge aller Objekte, die von diesem Programm definiert oder erzeugt werden. *Jedes Objekt erhält in der Programmierung einen Namen*, auch wenn dem Objekt im Programmtext kein Name explizit zugeordnet wurde. Z.B. erhält der Datentyp "array (1..n) of integer" in der Variablendeklaration "X, Y: array (1..n) of integer" zweimal einen Namen, auch wenn dies nicht im Programmtext geschieht, vgl. 1.3.2.3.b.

In einem Programm ist es entscheidend, *dass alle Namen unterschieden werden können*, da man sonst die Objekte nicht auseinander halten könnte. Der Name besteht hierbei aus einem "Identifikator" (identifier) und einer Information über seine Umgebung (in verschiedenen Umgebungen (z.B. Programmen) können natürlich gleiche Identifikatoren verwendet werden).

Zugrunde liegende Menge: Menge der Namen eines Programms

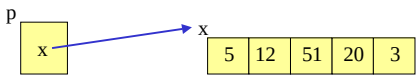
Nullstellige Operation: nil oder null ("kein Name")

Einstellige Operation: Zugriff auf das, was dieser Name repräsentiert. Man bezeichnet diese Operation gern durch

ref p oder p↑.

Zweistellige Operationen:
Gleichheit oder Ungleichheit von Namen.

Veranschaulichung über Pfeile (Zeiger, "pointer", Verweise):



Die Menge der Namen hat eine große Bedeutung. Wenn man beispielsweise einen Vektor aufsummiert:

```
s:= 0.0; for i in x'Range loop s := s + x(i) end loop;
```

und das Gleiche nun für zwei weitere Vektoren y und z tun möchte, so müsste man die Anweisung nochmals hinschreiben:

```
t:= 0.0; for i in y'Range loop t := t + y(i) end loop;
u:= 0.0; for i in z'Range loop u := u + z(i) end loop;
```

Gäbe es nun Variablen p und q vom Datentyp "Menge der Namen", so *könnte* man schreiben

```
p: (x,y,z); q: (s,t,u);
for p in (x..z) loop
    if p=x then q:=s; else q:=succ(q); end if;
    q↑:= 0.0; for i in p↑'Range loop q↑ := q↑ + p↑(i) end loop;
end loop;
```


Diese Schreibweise ist in der imperativen Programmierung ungebräulich. (Dort löst man das besagte Problem über Prozeduren; diese besitzen Parameter, die in gewisser Weise vom Datentyp "Name" sind.)

Dennoch ist das genannte Konzept in den meisten Programmier-sprachen als "Zeigerkonzept" vorhanden.

In Ada kann man nicht auf Namen, aber auf Datentypen verweisen. Hat man beispielsweise den Datentyp

```
type vektor is array (1..5) of integer;
```

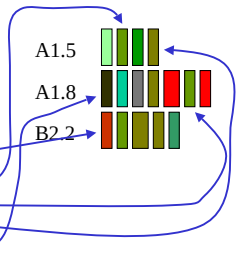
deklariert, so kann man durch

```
type ref_vektor is access vektor;
```

auf Objekte verweisen, die vom Typ vektor sind. (Siehe später ACCESS-Typen).

Beispiel: Bibliothek

Standort der Bücher:



Karteikarten oder Computereinträge

...
Kaiser, Albert, "Über die ...", B2.2
Kaiser, Karl, "Die helle ...", A1.5
Kaiser, Wolfgang, "Das ...", A1.8
Kaisers, Emma, "Über die ...", A1.5
Kaisser, Fritz, "Das alte ...", A1.8
...

Karteikarten sind vom Datentyp "Name" oder "Adresse".