Gliederung des Kapitels 1.4

1.4/1.5 Programmierung (und die Sprache Ada 95)

~~1.4.1 Blöcke, Deklarationen und Ausnahmen~~
~~1.4.2 Prozeduren und Funktionen~~
~~1.4.3 Moduln~~
~~1.4.4 Polymorphie~~
~~1.4.5 Vererbung~~
1.4.6 Abstrakte und konkrete Datentypen
1.4.7 Objekte
1.4.8 Grundprinzipien, Paradigmen der Programmierung

1.4.6 Abstrakte und konkrete Datentypen

Wir haben bereits mehrere konkrete Datentypen kennen gelernt: integer, real, Stack_fuer_Zeichen, array[1..N] of natural usw. Ein konkreter Datentyp besteht aus Datenstrukturen und darauf definierten Operationen einschließlich der erforderlichen Implementierung.

Wer Probleme zu lösen hat, interessiert sich erst in zweiter Linie um die Typen und deren detaillierte Implementierung. Zunächst stellt man Forderungen an die Lösungen, meist in Form von Eigenschaften. Von einem Stack erwartet man, dass er die Eigenschaft

"Was als letztes hingelegt wird, kommt als erstes wieder heraus"

d.h. die Gesetzmäßigkeit pop(push(a,s)) = s erfüllt.

1.4.6.1: Dies führt zum "abstrakten Datentyp":

Ein abstrakter Datentyp wird durch Bezeichnungen für Mengen, durch Bezeichnungen von Abbildungen (einschl. ihrer Stelligkeiten) und durch die zu erfüllenden Gesetze charakterisiert.

Er abstrahiert somit von den konkreten Darstellungen (Datentypen, Implementierung von Unterprogrammen und Moduln) und gibt nur die Gesetzmäßigkeiten an, denen die beteiligten Mengen genügen müssen.

Mathematisch gesehen ist ein abstrakter Datentyp somit eine (heterogene) Algebra.

Schema zur Beschreibung eines abstrakten Datentyps:

```
structure  < Name >  is
(<Liste generischer Elemente und benötigter Einheiten>)
modes  <nicht-leere Liste von Bezeichnern>
functions < Liste von Bezeichnern mit "Stelligkeiten" >
laws  < Liste von Gesetzmäßigkeiten
        in Form von logischen Aussagen >
end structure
```

Man kann auch andere Schlüsselwörter verwenden, z.B.: sorts oder sets anstelle von modes, mappings, operators oder procedures anstelle von functions, axioms, equations oder rules anstelle von laws. Die Auflistungen können durch Komma oder Semikolon getrennt sein und die Abbildungen post- oder preorder dargestellt werden.

In diesem Schema nennt man den Teil

modes <nicht-leere Liste von Bezeichnern>;
functions < Liste von Bezeichnern mit "Stelligkeiten" >

die Signatur des Datentyps. Diese entspricht der Spezifikation bei Moduln, legt aber die Datentypen der beteiligten Mengen noch nicht fest.

Manchmal bezeichnet man auch die Signatur alleine schon als abstrakten Datentyp.

Man beachte, dass die Konstanten, die verwendet werden sollen, als nullstellige Funktionen dargestellt werden.

Wir betrachten als Beispiel die Wahrheitswerte "WHW1":

1.4.6.2: Beispiel Wahrheitswerte 1: Wir nehmen, wir benötigen von den Wahrheitswerten nur folgende Eigenschaften:

```
structure WHW1 is
modes  B            -- dies steht für Menge der Wahrheitswerte
functions  wahr () B;
           not (B) B;
           and, or, impl (B, B) B;
           ifthenelsefi (B, B, B) B
laws       ASS: and (and (x,y), z) = and (x, and (y,z));
           KOMM: and (x, y) = and (y, x);
           NEG: not (not (x)) = x;
           IMP: impl (x, y) = or (not (x), y);
           IF1: ifthenelsefi (wahr, x, y) = x;
           IF2: ifthenelsefi (not (wahr), x, y) = y
end structure
```

Ordnet man den Sorten Mengen und den Funktionen Abbildungen entsprechend der vorgegebenen Stelligkeiten zwischen diesen Mengen zu und sind dann alle Gesetze erfüllt, so nennt man diese Mengen mit ihren Abbildungen (also diese "Algebra") ein Modell des abstrakten Datentyps.

Man nennt diese Mengen mit ihren Abbildungen auch einen zugehörigen konkreten Datentyp oder eine Konkretisierung oder eine Ausprägung oder eine Instanz.

Zu diesem abstrakten Datentyp gibt es meist mehrere Modelle, also mehrere konkrete Mengen mit Abbildungen, die alle Gesetze erfüllen. Zum Beispiel:

Modell 1: Betrachte den abstrakten Datentyp WHW1.
Für die Sorte B wählen wir die Menge $\mathbb{B} = \{false, true\}$,
wahr werde durch true und not(wahr) durch false dargestellt und die Funktionen not, and, or, impl und ifthenelsefi entsprechen den Funktionen Negation, Konjunktion, Disjunktion, Implikation und Alternative auf $\mathbb{B}$.
Dann sind alle Gesetze erfüllt, wie man leicht nachprüft.

Doch dieses Modell ist nicht das einzige. Vielmehr ist der abstrakte Datentyp WHW1 *polymorph,* d.h., er hat viele Modelle. Ein weiteres Modell ist:

Modell 2: Betrachte erneut den abstrakten Datentyp WHW1.
Für die Sorte B wählen wir die einelementige Menge $E = \{0\}$,
wahr werde durch 0 dargestellt und die Funktionen not, and, or, impl und ifthenelsefi liefern für alle Argumente den Wert 0.
Dann sind ebenfalls alle Gesetze erfüllt, z.B.:

Für das Gesetz
impl (x, y) = or (not (x), y);
setzen wir alle möglichen Werte ein (hier ist die 0 möglich):
0 = impl (0,0) = or (not (x), y) = or (0,0) = 0.

Es gibt zu WHW1 sogar unendlich viele verschiedene Modelle. Ein weiteres ist:

Modell 3: Betrachte erneut den abstrakten Datentyp WHW1.
Für die Sorte B wählen wir die ganzen Zahlen $\mathbb{Z}$,
wahr werde durch die Konstante 1 dargestellt und die Funktionen werden repräsentiert durch:
not(z) = - z (das Negative einer ganzen Zahl),
and (a,b) = a+b (Addition),
or (a,b) = a+b (Addition)
impl(a,b) = b - a (Subtraktion der ersten Zahl von der zweiten)
ifthenelsefi(1,b,c) = b und ifthenelsefi(a,b,c) = c für $a \neq 1$.
Dann sind ebenfalls alle Gesetze erfüllt, z.B.:
impl (x, y) = y - x = (-x) + y = or (not (x), y).

Kann man den abstrakten Datentyp auch "monomorph" machen, also so formulieren, dass es im Wesentlichen nur noch *ein* Modell zu ihm gibt? Versuchen wir es mit dem Beispiel Wahrheitswerte 2 (WHW2):


```
structure WHW2 is
modes  B
functions  wahr () B;     falsch () B;          -- 5 Funktionen
           not (B) B;     and (B, B) B;
           or (B, B) B
laws       ZWEI: wahr ≠ falsch;                 -- ≥ 2 Elemente
           A1: and (wahr,x) = x;                -- A1 bis A3
           A2: and (x,wahr) = x;                -- legen and fest
           A3: and (falsch,falsch) = falsch;
           OR1: or (falsch,x) = x;              -- OR1 bis OR3
           OR2: or (x,falsch) = x;              -- legen or fest
           OR3: or (wahr,wahr) = wahr;
           N1: not(falsch) = wahr;              -- N1 und N2
           N2: not(wahr) = falsch               -- legen not fest
end structure
```


Setze nun die Menge $\mathbb{B} = \{false, true\}$ für die Sorte B,
wahr werde durch true und falsch durch false dargestellt und die Funktionen not, and und or entsprechen den Funktionen Negation, Konjunktion und Disjunktion auf $\mathbb{B}$.
Dann sind alle Gesetze erfüllt.

Da die Funktionen not, and und or in den Gesetzen komplett wie in einer Funktionstabelle festgelegt sind, gibt es im Wesentlichen auch kein zweites hiervon verschiedenes Modell. Aber es gibt natürlich Erweiterungen (oder Einbettungen in größere Bereiche).

Setze hierfür die Menge der ganzen Zahlen **Z** für die Sorte B und wahr werde durch 1 und falsch durch 0 dargestellt.
Für die Funktionen wähle man:

not (x) = 1 - x

and (x, y) = x · y

or (x,y) = 1 - ((1-x) · (1-y))

Hier wurde IB in die Menge **Z** geeignet eingebettet.

Um solche erweiterten Modelle auszuschließen, kann man zwei Wege beschreiten:

a) Man fügt ein Gesetz der Form "Es gibt höchstens zwei Elemente" hinzu. In unserem Beispiel:
x∈B ⇒ (x = wahr ∨ x = falsch).

b) Oder man erlaubt generell nur "erzeugbare Modelle", d.h., in den Modellen sind nur solche Mengen erlaubt, die sich aus den angegebenen Konstanten mit Hilfe der angegebenen Abbildungen erzeugen lassen.
In unserem Beispiel sind dies alle Werte, die man aus "wahr" und "falsch" mittels not, and und or erzeugen kann; man sieht, dass hierdurch keine neuen Werte hinzukommen, so dass damit nur das zweielementige Modell zugelassen ist.

1.4.6.3: Beispiel "längenbeschränkter Keller" (vgl. Duden Inf.)

Bei einem Keller (Stack) interessiert den Benutzer nicht die Implementierung, sondern nur "LIFO" = last-in-first-out, also die kellerartige Speicherungstechnik. Diese lässt sich durch Gleichungen beschreiben (d: Element, x: Stack):

top (push (d, x)) = d,
pop (push(d, x)) = x.

Dies haben wir bereits unter den Moduln betrachtet. Wir modifizieren den dortigen Ansatz leicht, indem wir den "längenbeschränkten Keller" einführen, der nur bis zu "max" Elemente aufnehmen kann. "max" und die Sorte Δ der einzufügenden Elemente übergeben wir als Parameter; der Keller erhält die Sorte lbsΔ. Weiterhin mixen wir hier die Postorder- und die Preorder-Notation.


```
structure LBK is (mode Δ, natural max: max > 0,
                  based on boolean, based on natural)
modes  lbsΔ
functions  () lbsΔ empty;
           (lbsΔ) boolean isempty, isfull;
           (lbsΔ) Δ top;
           (lbsΔ) lbsΔ pop;
           (Δ, lbsΔ) lbsΔ push;
           (lbsΔ) natural length
laws       LEER: (x = empty) ⇔ isempty(x);
           VOLL: (length(x) = max) ⇔ isfull(x);
           LIFO: not isfull(x) ⇒ pop (push(d,x)) = x;
           KON: not isempty(x) ⇒ push (top(x), pop(x)) = x;
           OBEN: not isfull(x) ⇒ top (push(d,x)) = d;
           ANZ: not isfull(x) ⇒ length (push(d,x)) = length(x)+1;
           ANZ0: length (empty) = 0
end structure
```


Diese Darstellung ist möglicherweise etwas gewöhnungsbedürftig, doch lässt sie sich rasch in unsere gewohnte Repräsentation übersetzen.
(In der Praxis werden Notationen für jede Anwendung vorab festgelegt, so dass Sie lernen müssen, sich zunächst mit den Darstellungen vertraut zu machen.)

Wie bei Moduln können wir nun einen konkreten Datentyp angeben, der die Programmierung der Abbildungen und ggf. Initialisierungen enthält. Dies kann man ebenfalls wie einen abstrakten Datentyp formulieren, wobei man ein Schlüsselwort (hier: implementation; bei Ada-Moduln hieß dies "body") verwendet. Wir wählen hier die Realisierung mit Hilfe eines Feldes.


```
structure LBK is (mode Δ, natural max: max > 0,
                  based on boolean, based on natural)
modes  lbsΔ
functions  () lbsΔ empty;
           (lbsΔ) boolean isempty, isfull;
           (lbsΔ) Δ top;
           (lbsΔ) lbsΔ pop;
           (Δ, lbsΔ) lbsΔ push;
           (lbsΔ) natural length
< wer möchte, kann hier auch alle laws nochmals angeben >
implementation (*Implementierungsteil, hier: Ada-ähnlich*)
S: array (1 .. max) of Δ;
t: 0 .. max;
< Programmstücke für die Operationen, nächste Folie >
end structure;
```

Vorschlag für den Implementierungsteil, Ada-ähnlich formuliert:

```
implementation
S: array (1 .. max) of Δ;
t: 0 .. max;
procedure empty is begin t:= 0; end;
function isempty return boolean is  begin return (t = 0); end;
function isfull return boolean is  begin return (t = max); end;
function top return Δ is
        begin if isempty then „Fehlerabbruch“;
        else top:= S(t); end if; end;
procedure pop is
        begin if isempty then „Fehlerabbruch“;
        else t:= 1-1; end if; end;
procedure push (d: in Δ) is
        begin if isfull then „Fehlerabbruch“;
        else t:= 1+1; S(t):= d; end if; end;
function length return integer is  begin return t; end;
begin empty; end;                              -- Initialisierung
```


Hier wurde die Sorte lbsΔ zwar in der Signatur aufgeführt, aber in der Implementierung stillschweigend durch

```
array (1 .. max) of Δ
```

ersetzt. In einer Umsetzung in eine konkrete Programmiersprache muss dieser Zusammenhang natürlich deutlich gemacht werden.

Statt mit einem Feld hätte man den Datentyp LBK auch durch eine Liste realisieren können, wie dies im Kapitel 1.4.3 im Detail ausgeführt wurde (siehe auch Folie 155).

In der Praxis muss auch der Zugriff auf die Sorten geklärt werden (in Ada: private, limited private). Weiterhin kann man auch abstrakte Datentypen als "abstract" (im Ada-Sinne = man darf keine konkreten Datentypen hiervon erzeugen) kennzeichnen. Weitere Beschränkungen sind in der Praxis üblich; sie hängen meist von der jeweiligen Anwendung ab.