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 Generizität
1.4.5 Vererbung
1.4.6 Abstrakte und konkrete Datentypen
1.4.7 Objekte
1.4.8 Grundprinzipien, Paradigmen der Programmierung

1.4.2 Prozeduren und Funktionen

Hat man einmal einen Algorithmus geschrieben, der eine Aufgabe löst (d.h., eine Abbildung realisiert), so kann man ihn immer wieder benutzen, indem man ihn an der jeweiligen Stelle, wo man ihn braucht, hineinkopiert. Statt ihn zu kopieren, kann man dem Algorithmus als Abkürzung einen Namen geben und diesen Namen an die jeweilige Stelle schreiben. In der Regel benötigt der Algorithmus noch Eingabe- und Ausgabewerte, die man als ihm als "Parameter" mitgibt.
Einen solchen in sich abgeschlossenen Algorithmus nennt man "Unterprogramm" oder "Prozedur" (engl.: subroutine und procedure); wird der Algorithmus nur verwendet, um Werte zu berechnen, so spricht man von einer Funktion (engl. function).

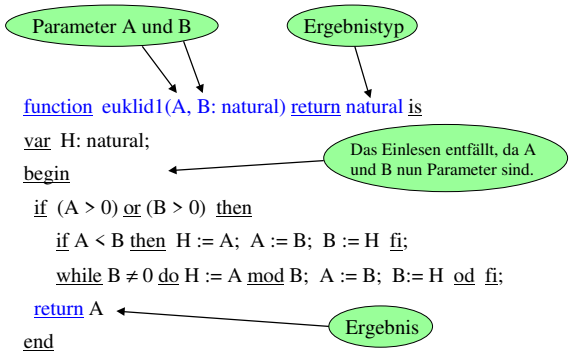
Wir beginnen mit den **Funktionen**.

Beispiel 1.4.2.1: ggT (siehe Beispiel 1.1.2.3)

Das dortige Programm lautete:

```
program euklid1 is
var A, B, H: natural;
begin read (A); read (B);
  if (A > 0) or (B > 0) then
     if A < B then H := A; A := B; B := H fi;
     while B ≠ 0 do H := A mod B; A := B; B:= H od fi;
  write (A)
end
```

Die Veränderlichen ("Parameter") sind die einzulesenden Variablen A und B. Das Ergebnis ist eine natürliche Zahl (write (A)). Wir schreiben diesen Algorithmus wie folgt in eine Funktion von $\mathbb{N}_0 \times \mathbb{N}_0$ nach $\mathbb{N}_0$ um:



(Änderungen gegenüber dem Programm sind hier blau notiert.
In Ada anders: Dort sind A und B keine Variablen, sondern Konstanten!)

Man definiert die Funktion euklid1 in irgendeinem Deklarationsteil. Damit ist zugleich ihr Sichtbarkeitsbereich festgelegt. Innerhalb dieses Sichtbarkeitsbereichs kann euklid1 in allen ganzzahligen Ausdrücken verwendet werden, wobei die aktuellen Werte natürliche Zahlen sein müssen, z.B. (K sei vom Typ natural; I, J, M: integer):

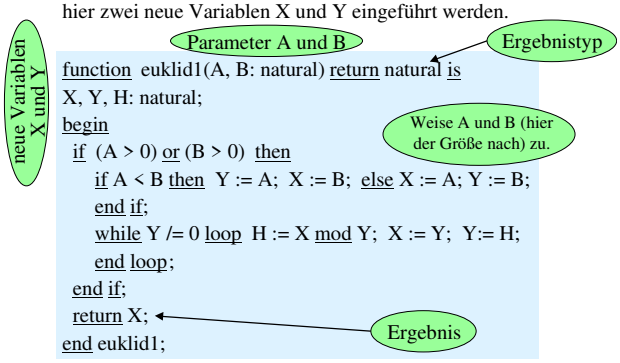
```
...
M := (7 + euklid1(K, 720)) * (I + J);
...
```

Eine Funktion der Form

```
        function ... return T is ....
```

kann also wie jeder Operand vom Typ T in Ausdrücken verwendet werden.

Darstellung in Ada: In Ada werden in Funktionen die Parameter stets als Konstanten behandelt. Daher müssen hier zwei neue Variablen X und Y eingeführt werden.



Haben Sie den Fehler in diesem Ada-Programm entdeckt?

Im Falle, dass A=0 und B=0 die aktuellen Parameter sind, trifft man auf return X, ohne dass X einen Wert erhalten hat. (Der Ada-Compiler gibt eine entsprechende Warnung aus!)

Ein Testlauf mit unserem Ada-Compiler ergab, dass irgendeine Zahl ohne Kommentar beim Aufruf euklid1(0,0) als Ergebnis zurückgegeben und das Programm nicht etwa abgebrochen wird. Dies mag bei anderen Ada-Compilern anderes sein.

Auf jeden Fall müssen wir X initialisieren, z.B. indem wir X:=0; als erste Anweisung in die Funktion einfügen.

Ende Beispiel 1.4.2.1 ■

1.4.2.2. *Eine Funktionsdeklaration hat in Ada die Form*

function <Name der Funktion> <Liste von formalen Parametern>
return <Ergebnisdatentyp> is
<Deklarationsteil> ;
begin <Folge von Anweisungen> end;

Falls es keinen Parameter gibt, so ist
<Liste von formalen Parameter>
leer, anderenfalls werden die Parameter mit ihren Datentypen aufgelistet (meist: je Datentyp getrennt durch Semikolon, die einzelnen Parameter für jeden Datentyp getrennt durch Komma).

Alle Berechnungsfolgen in der <Folge der Anweisungen> müssen auf eine "Ergebnis-Anweisung" return <Ausdruck> oder auf eine Ausnahmebehandlung führen.

Funktionen werden (nur) in Ausdrücken verwendet. Hierbei spricht man von einem "Funktionsaufruf".

Dieser "Funktionsaufruf" hat die Form

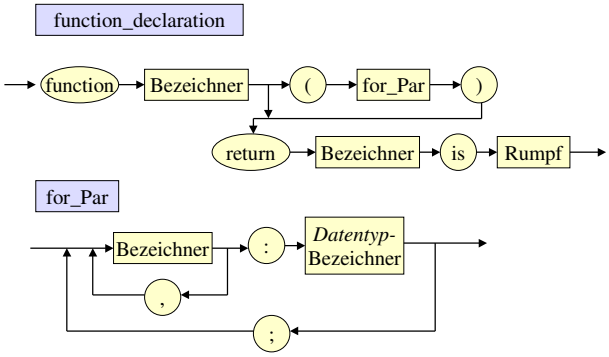
<Name der Funktion> <Liste von aktuellen Parametern>

wobei es genau so viele aktuelle wie formale Parameter geben muss und jeder aktuelle Parameter den gleichen Datentyp wie sein zugehöriger formaler Parameter haben muss.

Die aktuellen Parameter sind in der Regel Ausdrücke.

In obigem Beispiel könnte ein Funktionsaufruf in einer Wertzuweisung lauten: Z := (I*J)/euklid1(I+J, I-J) + abs(I);

[Hier können diverse Fehler auftreten (z.B. bei I-J < 0 oder bei I=J=0), die im Programm abgefangen werden sollten.]



Der Rumpf ist im Wesentlichen ein Block, aber ohne das Wort "declare" und ohne die Ausnahmebehandlung, siehe später EBNF für subprogram.

Besonderheiten der Funktionsdeklaration in Ada:

In Ada können die formalen Parameter nicht wie Variablen verwendet werden. Vielmehr sind sie Konstanten, denen anfangs die Werte ihrer zugehörigen aktuellen Parameter zugewiesen werden.

Die Werte der formalen Parameter dürfen in Ada-Funktionen also nicht verändert werden. Dadurch sollen Fehlerquellen, die durch die gedankenlose Verwendung von Parametern entstehen können, vermieden werden.

Weiterhin müssen die Datentypen "benannt" sein, d.h., sie müssen einen Namen tragen (sie dürfen also nicht "anonym" sein wie etwa array (1..5) of integer. Vielmehr muss man erst
type fuenftupel is array (1..5) of integer deklarieren und kann dann den formalen Parameter ... X: fuenftupel; ... einführen.

In der Deklaration dürfen für formale Parameter und für das Ergebnis unbegrenzte Typen stehen, die erst zur Laufzeit mit den konkreten Grenzen versehen werden. Hierdurch kann man Funktionen sehr universell verwenden, z.B:, wenn Felder das Ergebnis sind. Beispiel:

Beispiel

```
type Vektor is array (integer range <>) of float;
Bereichsunterschied: exception;
function Skalarprodukt (X, Y: Vektor) return float is
SP: float := 0.0;
begin
    if (X'First /= Y'First) or (X'Last /= Y'Last)
        then raise Bereichsunterschied;
    else
        for J in X'Range loop
            SP := SP + X(J) * Y(J); end loop;
        return SP;
    end if;
end Skalarprodukt;
```

1.4.2.3 Bedeutung eines Funktionsaufrufs $f(\alpha_1, ...,\alpha_k)$

Wenn f eine (zuvor deklarierte) Funktion mit k formalen Parametern ist und f in der Wertzuweisung für eine Variable X

X := ... $f(\alpha_1, ...,\alpha_k)$ ...

steht, so wird die Berechnung des Ausdrucks "... $f(\alpha_1,...,\alpha_k)$ ..." unterbrochen, sobald man auf den Operanden f stößt. Zunächst wird geprüft, ob f hier ein sichtbarer Name ist, dann werden die Ausdrücke $\alpha_1, ..., \alpha_k$ (dies sind die k aktuellen Parameter) in irgendeiner Reihenfolge ausgewertet, diese Werte werden den zugehörigen formalen Parametern von f zugewiesen und der Funktionsrumpf von f wird hiermit ausgerechnet, wobei man ein Resultat b erhält. Wenn b den Ergebnistyp der Funktion besitzt, so wird $f(\alpha_1, ...,\alpha_k)$ durch diesen Wert ersetzt und der Ausdruck "... $f(\alpha_1, ...,\alpha_k)$ ..." wird weiter ausgerechnet.

1.4.2.4 Rekursion

Im Inneren einer Funktion ist der Name der Funktion sichtbar. Man kann daher dort auch die Funktion selbst wiederum verwenden. Die (direkte oder indirekte) Verwendung einer Funktion in ihrem eigenen Rumpf nennt man Rekursion.

1.4.2.5 Erstes Standardbeispiel: Die Fakultätsfunktion

$$n! = \begin{cases} 1 & \text{für } n= 0; \\ n\cdot(n-1)! & \text{für } n>0 \end{cases}$$

```
function fak(n: natural) return natural is
begin  if n=0 then return 1 else return n*fak(n-1) fi  end fak
```

Vorteil: Diese rekursive Formulierung übernimmt direkt die Definition und ist daher fehlerfrei.

Die Formulierung in Ada ist fast identisch:

```
function fak(n: natural) return natural is
begin  if n=0 then return 1; else return n*fak(n-1); end if;
end fak;
```

Man kann die Fakultät natürlich auch iterativ (= mit Hilfe von Schleifen) berechnen, indem man $1\cdot2\cdot3\cdot...\cdot n$ bildet:

```
function fak2(n: natural) return natural is
F: natural :=1;
begin
   for J in 1..n loop F := F*J; end loop;
   return F;
end fak;
```


1.4.2.6 Zweites Standardbeispiel: ggT (in Ada formuliert)

```
function ggT(A, B: natural) return natural is
begin
      if  (A=0) and (B=0)  then  raise Constraint_Error;
      elsif  B=0  then  return A;
      else  return ggT(B, A mod B);
      end if;
end ggT;
```

Beachten Sie: Für a < b gilt stets a mod b = a, so dass der erste Aufruf ggT(a,b) zum Aufruf ggT(b,a) führt. Daher kann man die Abfrage, ob A<B ist, weglassen.

1.4.2.7 Drittes Standardbeispiel: Ullman's Funktion (in Ada)

```
function U(A: positive) return natural is
begin
      if A=1 then return 0;
      elsif A mod 2 = 0 then return g(A/2) + 1;
      else return g(3*A+1) + 1;
      end if;
end U;
```

Bei dieser Funktion ist noch nicht für alle natürlichen Zahlen z bewiesen, dass U(z) definiert ist (d.h., dass die Funktion U total ist); man vermutet dies aber. Rechnen Sie z.B. die Werte U(3), U(7), U(27), U(703), U(2463), U(159487), U(360361) aus.

1.4.2.8 Viertes Standardbeispiel: Gerade-Ungerade (in Ada)

```
function ungerade(A: natural) return Boolean;

function gerade (A: natural) return Boolean is
begin
      if A = 0 then return true;
      else return ungerade(A-1); end if;
end gerade;

function ungerade (A: natural) return Boolean is
begin
      if A = 0 then return false;
      else return gerade(A-1); end if;
end gerade;
```

1.4.2.9: Bezeichnungen

Funktionsspezifikation: Bezeichnung für die Angabe von Name, Urbild- und Wertebereich einer Funktion (oder Prozedur). In der Regel fügt man noch die Bezeichner für die formalen Parameter hinzu.
Eine Funktion h: $\mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{B}$ hat also z.B. die Spezifikation
function h (X, Y: natural) return Boolean;

Funktionsdeklaration: Spezifikation zusammen mit dem Programmstück, welches die Funktion realisiert.

Die Funktionsdeklaration ohne die Spezifikation bezeichnet man meist als Rumpf oder Implementation.
In Ada etwas anders: Ada bezeichnet die gesamte Funktionsdeklaration als Rumpf (= "body", siehe Syntax subprogram).

Obiges Beispiel: Spezifikation vorab angeben, damit die Funktion "gerade" die Funktion "ungerade" verwenden kann.

Funktionsspezifikation

```
function ungerade (A: natural) return Boolean;
```

zugehörige Funktionsdeklaration

```
function ungerade (A: natural) return Boolean is
begin
    if A = 0 then return false;
    else return gerade(A-1); end if;
end gerade;
```


Prozeduren: Ebenso wie man die Berechnung eines Wertes zu einer Funktion zusammenfassen und in Ausdrücken verwenden kann, darf man eine Folge von Deklarationen und Anweisungen zu einer Programmeinheit, genannt "Prozedur" oder "Unterprogramm" (engl.: procedure, subprogram, subroutine) unter einem Namen einschließlich der formalen Parameter zusammenfassen. Diesen Namen mit aktuellen Parametern kann man dann wie eine (elementare) Anweisung im Sichtbarkeitsbereich des Namens benutzen (Prozeduraufruf, "call").

Spezifikation für Unterprogramme, der "(<Parameterteil>)" darf fehlen:
procedure <Name> (<Parameterteil>);

Die Prozedurdeklaration beginnt mit dieser Spezifikation. Danach folgt der Rumpf bestehend aus dem Deklarationsteil und den Anweisungen.

Beispiel: Austauschen zweier Inhalte

```
program P is
var A, B, C, D, H: float;
begin   ... if A<B then H:=A; A:=B; B:=H fi;  ...
        ... H:=C; C:=D; D:=H;  ...
end
```

Herausziehen des Vertauschens als Unterprogramm:

```
procedure vertausche (X, Y: float) is
var Z: float;
begin Z:=X; X:=Y; Y:=Z end
```

Das abgewandelte Programm lautet dann:

Beispiel:

```
program PP is
var A, B, C, D, H: float;
procedure vertausche (X, Y: float) is
var Z: float;
begin Z:=X; X:=Y; Y:=Z end;
begin   ... if A<B then vertausche(A, B) fi;  ...
        ... vertausche(C, D);  ...
end
```

?

Ob dies korrekt ist, hängt von der *Übergabe der Parameter* ab! Falls X und Y als Konstanten (wie bei Ada-Funktionen) aufgefasst werden, dann ist PP kein äquivalentes Programm zu P.

1.4.2.11 Parameterübergaben

Den Mechanismus, wie die aktuellen Parameter den formalen Parameter beim Prozeduraufruf ("call") zugewiesen werden, bezeichnet man als Parameterübergabe.
Die drei wichtigsten Mechanismen sind:

call by value: Nur die Werte werden übergeben; die formalen Parameter sind lokale Variablen der Prozedur.

call by reference: Ein Verweis auf die aktuelle Variable wird übergeben; die formalen Parameter sind Zeigervariablen.

call by name: Der formale Parameter wird textuell durch den aktuellen Parameter ersetzt (wobei keine Namen im aktuellen Parameter hierdurch lokal werden dürfen).

Generell gilt für jede Verwendung von Unterprogrammen:

Globale Variablen dürfen bei der Parameterübergabe und den anschließenden Ersetzungsmechanismen nicht zu lokalen Variablen werden!

Dann liegt nämlich fast immer ein Fehler vor.

Das System muss in solchen Fällen die lokalen Variablen vor der Ausführung des Prozeduraufrufs umbenennen und auf diese Weise den "Namenskonflikt" beseitigen.

Beispiele finden Sie auf den Folien 71, 85/86.

Um diese Übergabemechanismen genau zu verstehen, müssen wir die *Bedeutung des Prozeduraufrufs* exakt beschreiben. Dies geschieht durch die

Kopierregel 1.4.2.12:

Gegeben sei eine Prozedur
procedure P($pü_1$ X_1: T_1; $pü_2$ X_2: T_2; ...; $pü_n$ X_n: T_n) is PRUMPF;

Hierbei gibt $pü_i \in$ {value, reference, name} die Art der Parameterübergabe für den i-ten formalen Parameter an. PRUMPF ist der Rumpf der Prozedur P.

Ein Prozeduraufruf $P(\alpha_1, \alpha_2, ..., \alpha_n)$ wird dann wie folgt ausgeführt (der aktuelle Parameter α_i ist ein Ausdruck vom Datentyp T_i des zugehörigen formalen Parameters):

Der Prozeduraufruf $P(\alpha_1, \alpha_2, ..., \alpha_n)$ wird durch folgenden Block BP ersetzt:

BP:
declare X_1: Typ_1; X_2: Typ_2; ...; X_n: Typ_n;
begin
$X_1 := \alpha_1$; $X_2 := \alpha_2$; ...; $X_n := \alpha_n$;
modifizierterPRUMPF
end;

Hierbei ist
$Typ_i = T_i$, falls $pü_i$ = value,
Typ_i = access T_i , falls $pü_i$ = reference,
X_i: Typ_i; und $X_i := \alpha_i$; entfallen, falls $pü_i$ = name ist.

Die Anweisung modifizierterPRUMPF ist stets ein Block. Er entsteht aus PRUMPF, indem

- die reference-Parameter "dereferenziert" werden,
- die Namenskonflikte, die durch globale Variable in den Prozedurrümpfen beim Kopieren entstehen würden, durch Umbenennung beseitigt werden,
- die Namenskonflikte, die bei der name-Übergabe durch die aktuellen Parameter entstehen würden, durch Umbenennung beseitigt und anschließend die formalen name-Parameter durch den Text des zugehörigen aktuellen Parameters ersetzt werden.

Genauer:

1. Jeder formale Parameter X_j mit $pü_j$ = reference wird durch deref X_j ersetzt (deref = folge einmal dem Verweis).

2. Jeder formale Parameter und jeder lokale Name in PRUMPF, der gleich einem Namen ist, der in irgendeinem α_i mit $pü_i$ = name auftritt, durchgehend mit einem neuen Namen bezeichnet wird und

3. danach alle X_i mit $pü_i$ = name durch α_i (textuell) ersetzt werden.

(4. Auf die globalen Variablen der Prozedurrümpfe gehen wir nicht näher ein, siehe Compilerbauvorlesungen, vgl. Folie 71.)

Dann wird dieser Block BP ausgeführt. Sobald BP beendet ist, wird er wieder durch den Prozeduraufruf $P(\alpha_1, \alpha_2, ..., \alpha_n)$ ersetzt und das Programm setzt die Berechnung mit der danach folgenden Anweisung fort.

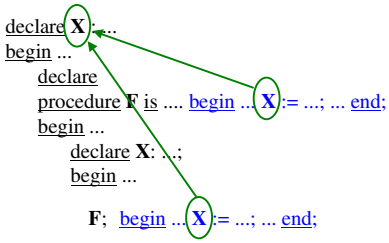
Diese (etwas modifizierte) Kopie des Prozedurrumpfs

BP:
declare X_1: Typ_1; X_2: Typ_2; ...; X_n: Typ_n;
begin
$X_1 := \alpha_1$; $X_2 := \alpha_2$; ...; $X_n := \alpha_n$;
modifizierterPRUMPF
end;

bezeichnet man als Inkarnation (oder konkrete Ausprägung) der Prozedur.

Hinweis zum Sonderfall 4.: Globale Variable der Prozedur müssen "mitgenommen" werden. Beispiel:

```
declare X: ...
begin ...
    declare
    procedure F is .... begin ... X := ...; ... end;
    begin ...
        declare X: ...;
        begin ...
            F; ...
        end;
    ...
    end;                                    Betrachte nun
    ...                                     den Aufruf  F
end;
```

Bereits zugeordnete globale Variablen dürfen durch die Kopierregel nicht zu "lokaleren" Variablen werden !! Daher müssen Namen umbenannt werden. Wir formulieren dies hier nicht weiter aus.

Ende Beschreibung der Kopierregel ■

Beispiel (siehe oben): Wir schreiben C statt Z, um den Namenskonflikt später zu erläutern. Wir beginnen mit call by value.

```
program PP is
var A, B, C, D, H: float;

procedure vertausche (value X: float; value Y: float) is
var C: float;
begin C:=X; X:=Y; Y:=C end;

begin   ...
        ... vertausche(C, D); ...
end
```

Ersetze nun vertausche(C, D) wie oben angegeben.

vertausche(C, D); ⟷

```
declare X: float; Y: float;
begin
  X := C; Y := D;
  declare C: float;
  begin C:=X; X:=Y; Y:=C end
end;
```

Ausgeführt wird also:

```
program PP is
...
begin     ...
    declare X: float; Y: float;
    begin  X := C; Y := D;
        declare C: float; begin C:=X; X:=Y; Y:=C end
    end;  ...
end
```


Diese Ausführung bewirkt keine Veränderungen für die Variablen C und D. Zwar werden die Werte von C und D in den Variablen X und Y vertauscht, aber diese beiden Variablen sterben nach Abarbeitung des Blocks

```
declare X: ...
end;
```

und die Inhalte der im äußeren Block deklarierten Variablen C und D bleiben unverändert.

Es darf also keine call-by-value-Übergabe erfolgen, wenn man den Inhalt zweier Variablen vertauschen will!

Wir probieren nun die call-by-reference-Übergabe aus.

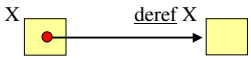

```
program PP is
var A, B, C, D, H: float;

procedure
vertausche (reference X: float; reference Y: float) is
var C: float;
begin C:=X; X:=Y; Y:=C end;

begin   ...
        ... vertausche(C, D); ...
end
```

Ersetze nun vertausche(C, D)
mittels call-by-reference.

vertausche(C, D);

```
declare  X: access float;
         Y: access float;
begin
   X := C; Y := D;
   declare C: float;
   begin C:=deref X;
      deref X:=deref Y;
      deref Y:=C
   end
end;
```

deref ist ein Operator für Zeigertypen. Er bedeutet: Folge dem Verweis einen Schritt weit (deref X ist das Objekt, auf das X verweist):



Ersetze nun vertausche(C, D), dann wird folgendes Programmstück ausgeführt:

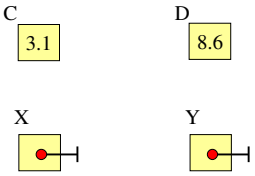
```
program PP is
...
begin      ...
  declare X: access float; Y: access float;
  begin  X := C; Y := D;
    declare C: float;
    begin C:=deref X; deref X:=deref Y; deref Y:=C end
  end;
 ...
end
```

Wir gehen dieses Programmstück schrittweise durch.

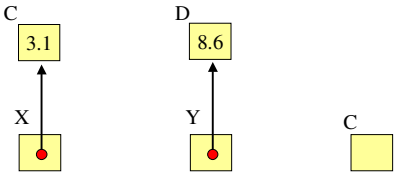
declare X: access float; Y: access float;

"Programmzeiger"

C habe den Wert 3.1 und D den Wert 8.6

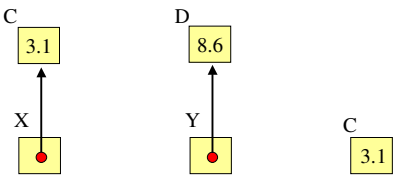


declare X: access float; Y: access float;
begin X := C; Y := D;
  declare C: float;

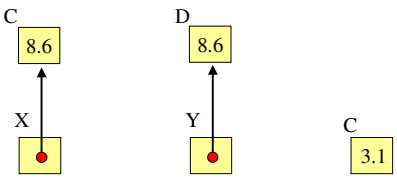


C ist lokale Variable des inneren Blocks.
Es entsteht hier kein Namenskonflikt.

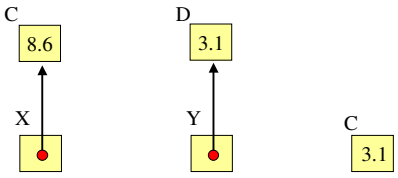
declare X: access float; Y: access float;
begin X := C; Y := D;
  declare C: float;
  begin C:=deref X; deref X:=deref Y; deref Y:=C end



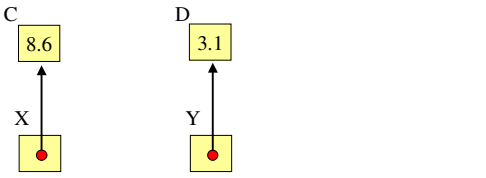
declare X: access float; Y: access float;
begin X := C; Y := D;
  declare C: float;
  begin C:=deref X; deref X:=deref Y; deref Y:=C end



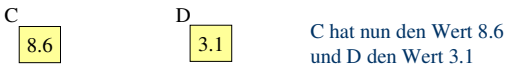
```
declare X: access float; Y: access float;
begin  X := C; Y := D;
    declare C: float;
    begin C:=deref X; deref X:=deref Y; deref Y:=C end
```




```
declare X: access float; Y: access float;
begin  X := C; Y := D;
    declare C: float;
    begin C:=deref X; deref X:=deref Y; deref Y:=C end
end;
```




```
declare X: access float; Y: access float;
begin  X := C; Y := D;
    declare C: float;
    begin C:=deref X; deref X:=deref Y; deref Y:=C end
end;
```



C hat nun den Wert 8.6
und D den Wert 3.1

call by reference leistet also das Gewünschte.

Wir untersuchen nun noch call by name.


```
program PP is
var A, B, C, D, H: float;

procedure
vertausche (name X: float; name Y: float) is
var C: float;
begin C:=X; X:=Y; Y:=C end;

begin   ...
        ... vertausche(C, D); ...
end
```

Ersetze vertausche(C, D) mittels call-by-name.

Aus begin C:=X; X:=Y; Y:=C end;
wird also begin C:=C; C:=D; D:=C end;

Hier liegt nun aber ein Namenskonflikt vor: Die global definierte Variable C würde durch die textuelle Ersetzung zur lokalen Variablen C:

```
var C: float; begin C:=C; C:=D; D:=C end;
```

Daher muss die innere Variable C umbenannt werden.
Aus var C: float; begin C:=X; X:=Y; Y:=C end;
wird also var C1: float; begin C1:=X; X:=Y; Y:=C1 end;

und erst danach erfolgt die textuelle Ersetzung, d.h.,
aus var C: float; begin C:=X; X:=Y; Y:=C end;
wird var C1: float; begin C1:=C; C:=D; D:=C1 end;

vertausche(C, D); ⟷

```
declare ;
begin
   declare C1: float;
   begin C1:=C; C:=D; D:=C1 end
end;
```

Es ist klar, dass dieses Programmstück die korrekte Vertauschung durchführt.
call-by-name arbeitet in unserem Beispiel also auch richtig.

Ende des Beispiels ■

Die exakte Bedeutung des Prozeduraufrufs versteht man vor allem durch Beispiele, Beispiele, Beispiele. Rechnen Sie daher viele Beispiele durch. Konstruieren Sie sich unangenehme Fälle und analysieren Sie diese, indem Sie die Kopierregel anwenden.

Üblicherweise dürfen alle Möglichkeiten, die für Prozeduren gelten, auch für Funktionen benutzt werden. D.h., auch für Funktionen sind die Parameterübergabemechanismen zu verwenden. Aber:

In Ada ist das alles etwas anders!

Einige Beispiele zum Üben finden Sie auf den folgenden Folien.

Übungsbeispiel Neu1:

```
program Neu1 is
var X, Y: natural;
function W(value A: natural) return natural is
begin
      if A<=1 then return 0  else return A + W(A-1)  fi
end W;
begin
  read (X, Y);
  for I :=1 to Y do X:=X+W(X) od;
  write (X)
end Neu1;
```

Berechnen Sie die Ausgabe zu den Eingaben 1, 2 per Hand bzw. mit dem Rechner für 6, 8 bzw. 9, 12 bzw. 10, 18.

Übungsbeispiel Neu2:

```
program Neu2 is
var A,J,X: integer;
function Q(value A: integer) return integer is
  begin  return A + X  end Q;
begin
  read (A,X);
  for J:=1 to A do X:=X+Q(A) od;
  write (X)
end Neu1;
```

Berechnen Sie die Ausgabe zu den Eingaben 1, 2 per Hand bzw. mit dem Rechner für 6, 8 bzw. 9, 12 bzw. 10, 18. Was erhält man, wenn man value durch reference oder name ersetzt?

Übungsbeispiel Neu3:

```
program Neu3 is
var A,J,X: integer;
procedure R(value A: integer; reference B: integer) is
var X: integer;
  begin  X := B+A; B := X+A; A := A+B  end R;
begin
  read (A,X);
  for J := 1 to A do R(X,A) od;
  write (A,X)
end Neu3;
```

Berechnen Sie die Ausgabe zu den Eingaben 1, 2 per Hand bzw. mit dem Rechner für 6, 8 bzw. 10, 18. Ersetzen Sie reference durch name. Welche Ausgaben erhalten Sie dann?

Übungsbeispiel Neu4:

```
program Neu4 is
type vektor is array (1..4) of integer;
var A,I,J: integer; Y: vektor := (1, 2, 3, 4);
procedure S(name A: integer; reference B: vektor) is
  begin  J:=J+1; A:=A+1; B[A] := B[J] + A end R;
begin  A:=2; J:=1; S(J,Y); S(Y[J],Y);
  for I:= 1 to 4 do write (Y[I]) od
end Neu4;
```

Berechnen Sie die Ausgabe zu den Eingaben 1, 2 per Hand bzw. mit dem Rechner für 6, 8 bzw. 10, 18. Ersetzen Sie reference durch name. Welche Ausgaben erhalten Sie dann?

1.4.2.12 Parameterübergaben in Ada:

Die formalen Parameter werden als lokale Konstanten oder Variablen des entsprechenden Typs in der Prozedurinkarnation aufgefasst. Es gibt in Ada95 drei mögliche Arten von für Parameter (in Ada "mode" genannt; für Funktionen ist nur "in" erlaubt, weshalb man diese Angabe dort auch weglassen kann; wird kein mode angegeben, so wird "in" eingefügt):

in	Der formale Parameter ist eine Konstante, die mit dem Wert des aktuellen Parameters initialisiert wird.
in out	Der formale Parameter ist eine Variable, die mit dem Wert des aktuellen Parameters initialisiert wird. Sie kann auf den Inhalt des aktuellen Parameters lesend und schreibend zu greifen.
out	Wie "in out", aber ohne Initialisierung.

Die in-Parameter heißen "Eingangsparameter", die out- bzw. in-out-Parameter heißen "Ausgangsparameter".

Eingangsparameter werden wie Konstanten behandelt, deren Wert in der Prozedur nicht verändert werden darf. Jeder zugehörige aktuelle Parameter kann ein Ausdruck des Typs des formalen Parameters sein.

Ausgangsparameter sind Variablen, die auf jeden Fall am Ende der Prozedur ihren Wert an den zugehörigen aktuellen Parameter übergeben (dies kann auch zwischendurch geschehen, muss aber nicht; implementierungsabhängig). Der zugehörige aktuelle Parameter muss eine Variable sein, die während des Prozeduraufrufs nicht durch eine andere Variable ersetzt werden kann.

Die Übergabe kann durch Kopieren oder durch Verweis erfolgen (zum Teil ist dies implementierungsabhängig).

Prozeduren verändern in der Regel Inhalte von Variablen (oder sie erzeugen Ausdrucke). Diese Veränderungen erfolgen können nur geschehen, wenn entweder in-out- oder out-Parameter vorliegen oder globalen Variablen Werte zugewiesen werden.

Hinweis 1: Die Veränderung von globalen Variablen in einer Funktion oder Prozedur bezeichnet man allgemein als "Seiteneffekt". Solche Effekte sind oft Anlass für Fehler, die nur schwer aufzuspüren sind. Das Programmieren mit Seiteneffekten sollte daher unbedingt vermieden werden (schlechter Programmierstil)!

Hinweis 2: Die Übergabe mit Ausgangsparametern kann durch Kopieren oder durch Verweis erfolgen (zum Teil ist dies implementierungsabhängig). Man muss so programmieren, dass bei beiden denkbaren Möglichkeiten das gleiche Resultat entsteht (sofern kein Fehlerabbruch erfolgt).

Standardbeispiel für einen einfachen Seiteneffekt:

```
program ... is
var A: integer :=1;
function erhöhe return integer is
  begin A:=A+1; return A end erhöhe;
begin
  write (A+erhöhe(A))
end;
```

In diesem Beispiel kann 3 oder 4 ausgegeben werden, je nachdem, ob die Addition "A+erhöhe(A)" zuerst den ersten oder den zweiten Operanden auswertet.

Auf weitere Beispiele, insbesondere Rekursion und zusammengesetzte Datentypen, kommen wir im Laufe der Vorlesung bzw. im Programmierkurs häufiger zurück.

1.4.2.13: Es folgt die Syntax für Prozeduren und Funktionen und weitere Programmbestandteile *für Ada95* an (beachte: In Ada heißt die Unterprogrammdeklaration "Rumpf" = "body"):

```
subprogram_body ::=
    subprogram_specification 'is'
    declarative_part
    'begin'
    handled_sequence_of_statements
    'end' [designator];
subprogram_specification ::=
    'procedure' defining_program_unit_name  parameter_profile
    | 'function' defining_designator  parameter_and_result_profile
```



```
parameter_profile ::= [formal_part]

parameter_and_result_profile ::=
        [formal_part] 'return' subtype_mark

formal_part ::=
      "("parameter_specification {";" parameter_specification}")"

parameter_specification ::=
        defining_identifier_list ":"  mode  subtype_mark
                [":=" default_expression]      |
        defining_identifier_list ":" access_definition
                [":=" default_expression]

subtype_mark ::= subtype_name

mode ::= ['in'] | 'in' 'out' | 'out'

access_definition ::= 'access' subtype_mark
```



```
defining_program_unit_name ::=
        [parent_unit_name "." ] defining_identifier

parent_unit_name ::= name

defining_designator ::=
        defining_program_unit_name |
        defining_operator_symbol

defining_operator_symbol ::= operator_symbol

operator_symbol ::= string_literal

string_literal ::= "{string_element}"

string_element ::= "" | non_quotation_mark_graphic_character
```

Ein string_element ist also entweder ein Paar von Anführungszeichen ("") oder ein Tastaturzeichen verschieden vom Anführungszeichen.

1.4.2.14 Operatoren

Spezielle Funktionen sind die Operatoren, die in der Regel mit besonderen Symbolen bezeichnet werden. Z.B.:

Addition natürlicher Zahlen $+ : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$

Addition ganzer Zahlen $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

Addition reeller Zahlen $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$

Addition von Vektoren $+ : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$

In Ada ist es erlaubt, auch Operatoren (als Funktionen) zu vereinbaren und das Operatorsymbol dann im Programm zu verwenden.

Es sei
type Vektor is array (1..N) of float;
Dann kann man deklarieren:


```
function "+" (X, Y: Vektor) return Vektor is
Summe: Vektor;
begin  for J in Y'Range loop Summe (J) := X(J) + Y(J);  end loop;
       return Summe;
end "+";
```

Operatoren werden wie Funktionen deklariert, allerdings muss das Operatorzeichen in Anführungsstriche eingeklammert sein. Einstellige Operatoren werden in Präfix-, zweistellige in Infix-Notation in Ausdrücken benutzt:

```
A, B, C: Vektor; ....
A := B + C; ...
```

In Ada sind nicht beliebige Zeichen für Operatoren zu gelassen, sondern nur die Symbole, die als Operatoren in Ada erlaubt sind, also nur: +, -, *, /, **, &, =, /=, <, <=, >, >=, abs, and, mod, not, or, rem, xor. Diese können durch eine Funktionsdeklaration mit einer zusätzlichen Bedeutung belegt werden.

1.4.2.15 Überladen

Wie in der Informatik bzw. Mathematik üblich hat der Operator "+" mehrere Bedeutungen je nachdem, in welchem Kontext wir ihn einsetzen. Diese Zuordnung verschiedener Bedeutungen bezeichnet man in Ada als "überladen" (englisch: *overloading*).

Das Überladen eines Operators, aber auch eines Funktions- oder eines Unterprogramm-Namens ist in Ada erlaubt, sofern sich die diversen Deklarationen
- in der Reihenfolge der Parametertypen,
- in mindestens einem Parametertyp oder
- im Ergebnistyp

unterscheiden. Dann kann man nämlich den passenden Operator bzw. Unterprogramm-Namen eindeutig auffinden.

Beispiel: Erlaubt sind im gleichen Deklarationsteil:

```
function "+" (X, Y: Vektor) return Vektor is
Summe: Vektor;
begin  for J in Y'Range loop Summe (J) := X(J) + Y(J); end loop;
       return Summe;
end "+";
function "+" (X: float; Y: Vektor) return Vektor is
Summe: Vektor;
begin  for J in Y'Range loop Summe (J) := X + Y(J); end loop;
       return Summe;
end "+";
function "+" (X, Y: Vektor) return float is
Summe: float := 0.0;
begin  for J in Y'Range
              loop  Summe := Summe + X(J) + Y(J); end loop;
       return Summe;
end "+";
```


Es seien:
A, B, C: float; D, E, F: Vektor;
Mit den obigen drei Operatoren sind im weiteren Verlauf die folgenden vier Zuweisungen erlaubt, die fünfte dagegen nicht.

```
C := A+B;          -- vordefinierte float-Addition
A := (D+E) + A;    -- links: dritter Operator und dann
                   -- rechtes "+" = float-Addition
F := A+D;          -- zweiter Operator
E := F+D;          -- erster Operator
D := (E+B) + F;    -- nicht definiert, da "Vektor + float"
                   -- nicht deklariert wurde.

D := (B + E) + F;  -- Dies ist dagegen zulässig: erst den
                   -- zweiten, dann den ersten Operator.
```


Beispiel aus 1.4.2.2: Das Skalarprodukt schreibt man meist als

```
function "*" (X, Y: Vektor) return float is
SP: float := 0.0;
begin  for J in X'Range loop
            SP := SP + X(J) * Y(J);  end loop;
       return SP;
end "*";
```

Beachten Sie: Die Operatorsymbole bzw. Unterprogramm-Namen unterliegen der bereits festgelegten Lebensdauer und der Sichtbarkeit. Siehe nächste Folie; die beiden Deklarationen für "*" sind innerhalb des gleichen Deklarationsteil nicht erlaubt.

äußerer Block

```
declare  Z: float;
function "*" (X, Y: Vektor) return float is
SP: float := 0.0 ;
begin  for J in Y'Range loop  SP := SP + X(J) * Y(J) ;  end loop ;
       return SP ;
end "*"; ...
begin ...
    Z := A*B;  ...
    declare                                          innerer Block
    function "*" (X, Y: Vektor) return float is
    S: float := 0.0 ;
    begin for J in Y'Range loop  S := S + X(J) + Y(J) ;  end loop ;
        return S ;
    end "*"; ...
    begin  ...
        Z := A*B;  ...
    end; ...
    ...
end;
```

Im inneren Block wird der Operator "*" des äußeren Blocks umdefiniert.

Warnung: Das Überladen kann zu schwer erkennbaren Fehlern führen, insbesondere wenn hierbei ein Operator oder ein Unterprogrammname in geschachtelten Blöcken mehrfach umdefiniert wird.

Man gebe sich in solchen Fällen einige Regeln, etwa:

- Einheitliche Stelligkeit von Operatorsymbolen, z.B.: Deklariere ein Operatorsymbol, das üblichweise zwei Operanden hat, nicht zu einem dreistelligen Operator um.
- Benenne die einzelnen Operatordeklarationen im Kommentarteil eindeutig und notiere im Kommentarteil hinter den Zuweisungen, welcher Operator hier gemeint ist.