# Abschnitt 5

## Funktionen, Unterprogramme

Funktionen

Rekursion, Instanzen/Inkarnationen

Operationen

Prozeduren

Formale und aktuelle Parameter

Überladen



## 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**.

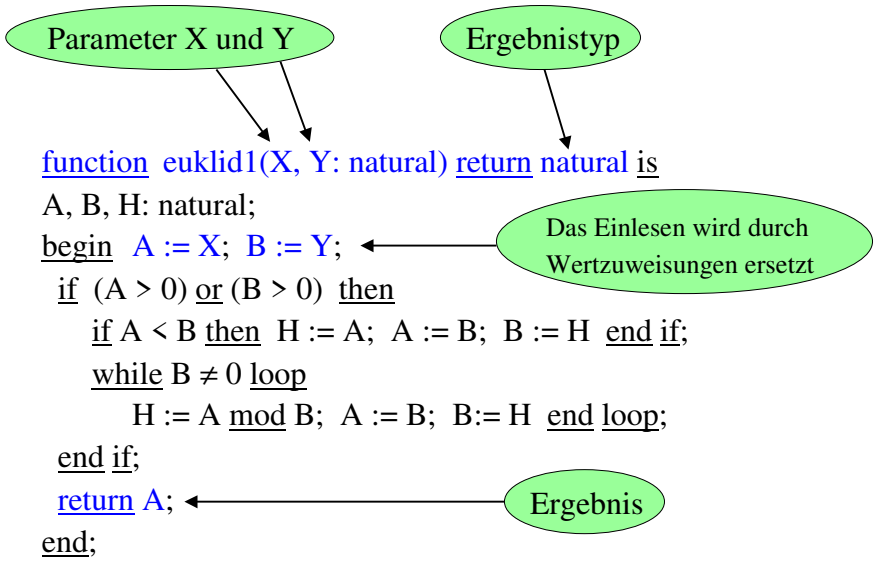
*Standardbeispiel ist der größte gemeinsame Teiler ggT*

Das Verfahren lautet als Programm bekanntlich:

```
program euklid1 is
A, B, H: natural;
begin  get (A); get (B);
 if  (A > 0) or (B > 0)  then
    if A < B then  H := A;  A := B;  B := H  end if;
    while B ≠ 0 loop
        H := A mod B;  A := B;  B:= H  end loop;
 end if;
 put (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.)

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.



*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 entfällt
(<Liste von formalen Parameter>)
anderenfalls werden die Parameter mit ihren Datentypen aufgelistet (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".

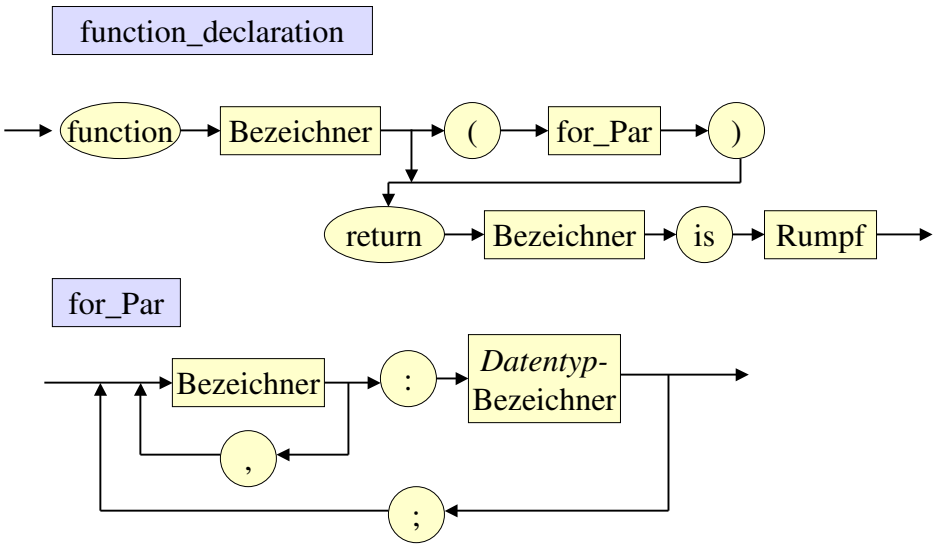
Dieser "Funktionsaufruf" hat die Form

<Name der Funktion> (<Liste von aktuellen Parametern>)

wobei es genau so viele aktuelle wie formale Parameter geben und jeder aktuelle Parameter den gleichen Datentyp wie sein zugehöriger formaler Parameter haben muss. [Die Zuordnung kann auch wie üblich mittels "=>" erfolgen.]

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);
Hierbei wird durch den Datentyp der formalen Parameter sichergestellt, dass die Funktion euklid1 nur für I+J ≥ 0 und I-J ≥ 0 ausgeführt wird (sonst: Constraint_Error).






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-Funktionen 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 einen 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;
```

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

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

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

Stößt man auf den Namen "f", so wird die Berechnung des Ausdrucks "... $f(\alpha_1,...,\alpha_k)$ ..." unterbrochen. 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 b ersetzt und der Ausdruck auf der rechten Seite der Wertzuweisung wird weiter ausgerechnet.



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.

*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); end if;
end fak;
```

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

Man kann die Fakultät natürlich auch iterativ (= mit Hilfe von Schleifen) berechnen, indem man $1 \cdot 2 \cdot 3 \cdot ... \cdot n$ 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 fak2;
```



*Zweites Standardbeispiel: ggT*

```
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 Aufruf ggT(a,b) zum Aufruf ggT(b,a) führt. Daher kann man die Abfrage, ob A<B ist, weglassen.

*Drittes Standardbeispiel: Ullman's Funktion*

```
function U(A: positive) return natural is
begin
    if A=1 then return 0;
    elsif A mod 2 = 0 then return U(A/2) + 1;
    else return U(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), aber man vermutet dies. Rechnen Sie z.B. die Werte U(3), U(7), U(27), U(703), U(2463), U(159487), U(360361) aus.



*Viertes Standardbeispiel: Gerade-Ungerade*

```
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;
```

Übliche 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 (in Ada wird dies jedoch als "body" bezeichnet): Spezifikation zusammen mit dem Programmstück, welches die Funktion realisiert.

Die Funktionsdeklaration ohne die Spezifikation bezeichnet man meist als Rumpf oder Implementation (engl. body).



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
A, B, C, D, H: float;
begin   ...  if A<B then H:=A; A:=B; B:=H; end if;  ...
        ...  H:=C; C:=D; D:=H;  ...
end;
```

Herausziehen des Vertauschens als Unterprogramm:

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

Das abgewandelte Programm lautet dann:

*Beispiel:*

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

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

begin   ... if A<B then vertausche(A, B) end if; ...
        ... 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.



Parameterübergaben allgemein laut Grundvorlesung

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 Variable sind zulässig. Sie dürfen bei der Parameterübergabe und den anschließenden Ersetzungsmechanismen jedoch 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 in späteren Folien.

Weiterhin beziehen sich globale Variable stets auf die "statische Umgebung", d.h. auf die Stelle, an der das Unterprogramm deklariert wird.



Um diese Übergabemechanismen genau zu verstehen, muss die *Bedeutung des Unterprogrammaufrufs* exakt beschrieben werden. Dies geschieht durch die Kopierregel, siehe Grundvorlesung: Man kopiert an die Stelle den Rumpf des Unterprogramms ein und modifiziert diesen entsprechend der jeweiligen Parameterübergabe.

Die (modifizierte) Kopie des Prozedurrumpfs bezeichnet man als Inkarnation (oder Instanz) des Unterprgramms.

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 formalen 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 zugreifen. |
| 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 der Auswertung 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 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
A: integer :=1;
function erhöhe return integer is
  begin A:=A+1; return A; end erhöhe;
begin
  put (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.

Weitere Beispiele, insbesondere Rekursion und zusammengesetzte Datentypen, werden Sie im Laufe des Programmierkurses im Sommersemester häufiger finden

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

Einige Beispiele zum Üben finden Sie auf den folgenden Folien.



Übungsbeispiel Neu1:

```
program Neu1 is
X, Y: natural;
function W(A: natural) return natural is
begin
  if A<=1 then return 0; else return A + W(A-1); end if;
end W;
begin
  get (X); get(Y);
  for I in 1..Y loop X:=X+W(X); end loop;
  put (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
A, J, X: integer;
function Q(A: integer) return integer is
  begin  return A + X;  end Q;
begin
  get(A); get(X);
  for J in 1..A loop X:=X+Q(A); end loop;
  put (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 Neu3:

```
program Neu3 is
A, J, X: integer;
procedure R(A: in integer; B: in out integer) is
X: integer;
  begin  X := B+A; B := X+A;  end R;
begin
  get(A); get(X);
  for J in 1..A loop R(X, A); end loop;
  put(A); put(X);
end Neu3;
```

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

Übungsbeispiel Neu4:

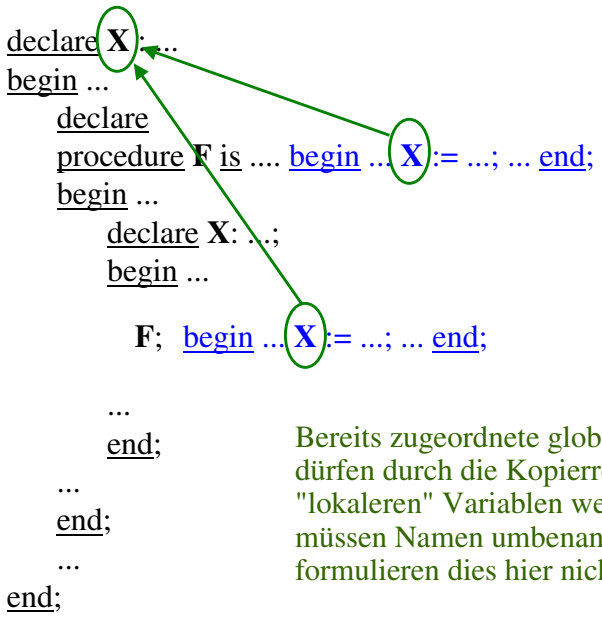
```
program Neu4 is
type vektor is array (1..4) of integer;
A, I, J: integer; Y: vektor := (1, 2, 3, 4);
procedure S(A: in out integer; B: in out vektor) is
  begin  J:=J+1; A:=A+1; B[A] := B[J] + A; end R;
begin  A:=2; J:=1; S(J,Y); S(Y[J],Y);
  for I:= 1 to 4 loop put (Y[I]); end loop;
end Neu4;
```

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



Hinweis zum Fall auf Folie 23 unten: 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.



Konkretes Beispiel zu diesem Fall:

```
procedure BeispielA is
X : integer;
function F (J: Integer) return Integer is
   begin X:=X+1; return J+2; end;

begin
   X := 0;
   declare Q, X: Integer := 1;
   begin  X := F(Q); X := X+1;
   end;
   X := X + F(X);
   put (X);
end;
```

Berechnen Sie die Ausgabe dieses Programms.

Die Syntax für Prozeduren und Funktionen und weitere Programmbestandteile für Ada95 (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.



*Beispielprozedur GD für den Graphdurchlauf:*

```
...
Anfang, p: Nextknoten;
procedure GD (u: in out NextKnoten) is
v: NextKnoten; e: NextKante;
begin if not u.besucht then
      u.besucht := true; < "bearbeite den Knoten u" >;
      e := u.EIK;
      while e /= null loop  < "bearbeite die Kante e" >;
            GD(e.EKn); e:=e.NKa; end loop;
end GD;
begin  ... < "baue den Graphen auf" >; ... ;
      p := Anfang;
      while p /= null loop p.besucht:=false; p := p.NKn; end loop;
      p := Anfang;
      while p /= null loop GD(p); p := p.NKn; end loop;  ...
end;
```

Operatoren

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

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

Addition ganzer Zahlen $+ : \mathbf{Z} \times \mathbf{Z} \rightarrow \mathbf{Z}$

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

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

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.

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 verschiedener 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:* 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; dort sind die beiden Deklarationen für "*" 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 üblicherweise 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.