8. Funktionen

Idee: Wenn ein Verfahren ausformuliert ist, möchte man es als "elementare Handlung" überall verwenden können. Hierzu benötigt man einen Namen und die genaue Angabe zur Übergabe von Werten ("Parameter").

Der Name muss als Funktion deklariert werden.

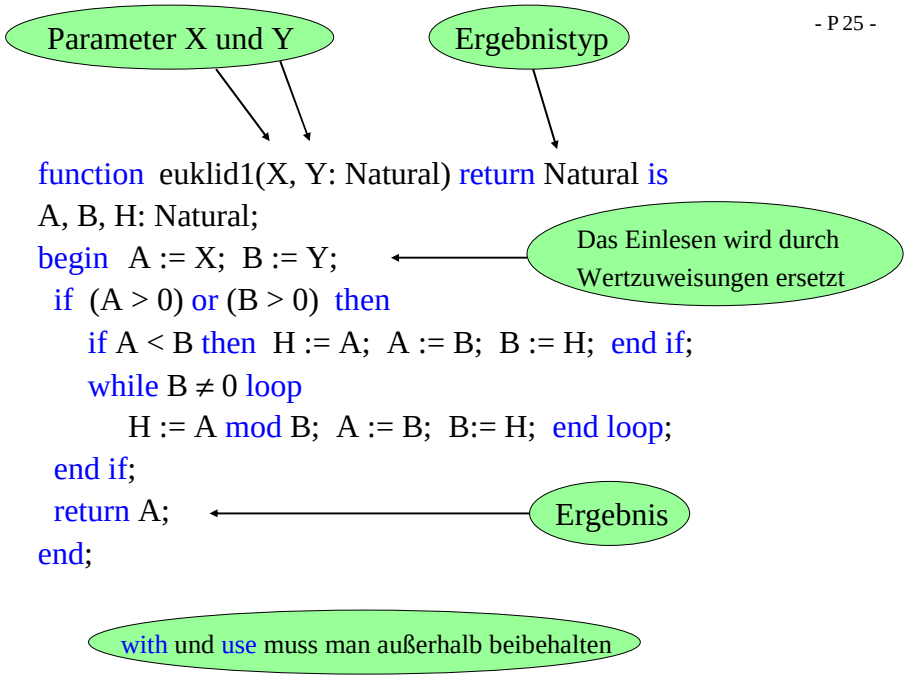
Die Funktion muss die Quell- und Zielbereiche der zugehörigen realisierten Abbildung genau bezeichnen. Dies geschieht durch eine Liste von formalen Parametern mit ihren Typen und die Angabe des Ergebnistyps. Die formalen Parameter sind bei Funktionen stets Konstanten.

Funktionen können in Ada nur in Ausdrücken benutzt werden.

Standardbeispiel ggT. Euklidischer Algorithmus:

```
with Ada.Integer_Text_IO; use Ada.Integer.Text_IO;
procedure 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 (Put (A)). Wir schreiben diesen Algorithmus wie folgt in eine Funktion von $\mathbb{N}_0 \times \mathbb{N}_0$ nach $\mathbb{N}_0$ um:



Man definiert die Funktion euklid1 im Deklarationsteil. Damit ist festgelegt, wo der Name bekannt ist (Sichtbarkeitsbereich). 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

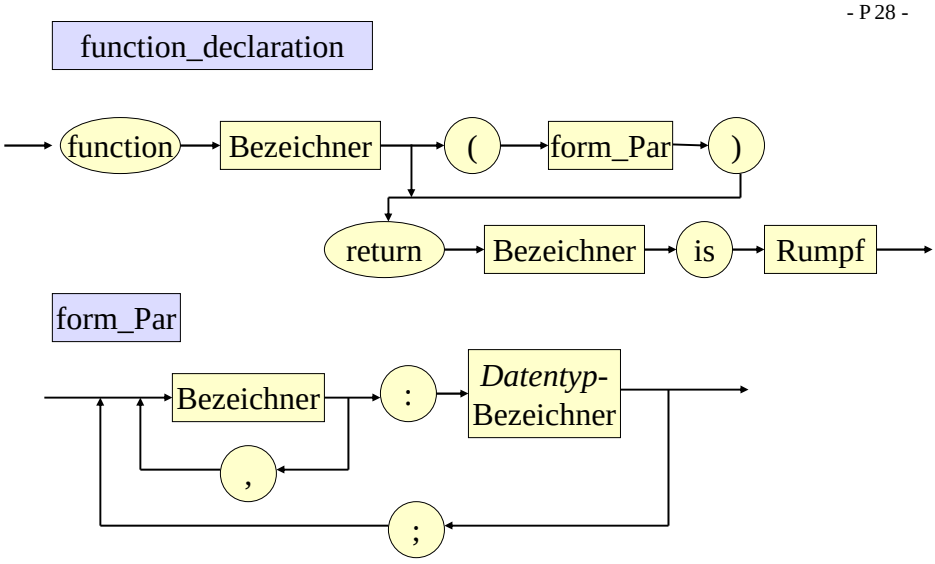
function <Name der Funktion> (<Liste von formalen Parametern>)
return <Ergebnisdatentyp> is
<Deklarationsteil> ;
begin <Folge von Anweisungen> end;

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).

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 mittels "=>" erfolgen.]
Aktuelle Parameter sind meist Ausdrücke: Z := (I*J)/euklid1(I+J, I-J) + I;



Der Rumpf ist im Wesentlichen eine Folge von Anweisungen eingeschlossen in begin ... end.

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.
Man muss also 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

```
type Vektor is array (Integer range <>) of Float;    . . .

function Skalarprodukt (X, Y: Vektor) return Float is
SP: Float := 0.0;
begin
    if (X'First /= Y'First) or (X'Last /= Y'Last)
        then  Put("Fehler. Bereiche stimmen nicht überein.");
    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 bekannt ("sichtbar") ist, dann werden die Ausdrücke α_1, ..., α_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 bekannt (man sagt auch "sichtbar"). Man kann daher dort die Funktion selbst 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.

Man kann die Fakultät natürlich auch iterativ (= mit Hilfe von Schleifen) berechnen, indem man $1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ 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  Put("Fehler";
      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: Ulam-Collatz-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 Ungerade;
```

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 als function-"body" bezeichnet): Spezifikation zusammen mit dem Programmstück, welches die Funktion realisiert.

Die Funktionsdeklaration ohne die Spezifikation bezeichnet man meist als Rumpf oder Implementation.

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 Ungerade;
```

9. Prozeduren:

Man darf 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 (eventuell leeren) Deklarationsteil und den Anweisungen.

Beispiel: Austauschen zweier Inhalte

```
procedure 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: in out Float) is
Z: Float;
begin Z:=X; X:=Y; Y:=Z; end;
```

Das abgewandelte Programm lautet dann:

Beispiel:

```
procedure PP is
A, B, C, D, H: Float;

  procedure Vertausche (X, Y: in out 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 in Ada:

Die formalen Parameter werden als lokale Konstanten oder Variablen des entsprechenden Typs 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.

Hinweis: Variablen, die außerhalb der Prozedur deklariert und in der Prozedur bekannt sind, heißen globale Variable. Solche Variablen, die erst im Inneren der Prozedur deklariert werden, heißen lokale Variable. Globale Variable sind zugelassen und können zur Übergabe von Werten genutzt werden. Aber: Vorsicht! Denn dies ist eine Quelle für undurchschaubare Fehler.

- P 43 -

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 daher 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 auch dies implementierungsabhängig).

- P 44 -

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:

```
procedure ... 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 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
```

- P 47 -

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

- P 48 -

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.