1.4 Unterprogramme

Strukturierung von Programmen durch

- Unterprogramme
- Module
- Klassen

... u.a.

Unterprogramm *(subprogram):*

- *benanntes Programmfragment,*
- kann mit Bezugnahme auf den Namen *aufgerufen* werden, d.h. Programmausführung wird dort fortgesetzt;
- nach Abarbeitung *Rücksprung* zur *Aufrufstelle;*
- *Parametrisierung* möglich.

= imperative Variante der applikativen **Funktion**

Vereinbarung eines Unterprogramms z.B. so:

```
void print(float x) {.....}
```

Parameter Programmfragment
(parameter)

Benutzung durch **Aufruf** in anderem Teil des Programms, z.B. so:

```
...;
print(pi);
...;
```

Aufrufstelle

! Großer Variantenreichtum bei der Terminologie -
teilweise nur Synonyme, teilweise mit semantischen Unterschieden !

Alternative Bezeichnungen:

Unterprogramm *(subprogram)* - eher bei Assembler-Sprachen, ursprünglich auch bei *Fortran*

Prozedur *(procedure)* - klassische Bezeichnung (*Algol* und Sprachen der Algol-Familie ... bis Modula)

Funktion *(function)* - unglückliche Bezeichnung, z.B. bei *C,* etwas treffender auch bei *Pascal*

Routine *(routine)* - so z.B. bei *Eiffel*

Operation *(operation)* - bei anderen objektorientierten Sprachen

Methode *(method)* - bei wieder anderen, z.B. *Java*

1.4.1 Prozedurvereinbarung

(procedure declaration)

= vollständiger Text der Prozedur,
an bestimmten Stellen im Programm möglich

Java: Prozedur wird *im Inneren einer Klasse* vereinbart
und auch als **Methode** *(method)* bezeichnet

Bemerkung: Ein Java-Programm besteht aus einer oder
mehreren *Klassen* (➔1.5)

```
class Test {

static int minsquare(int n)
                {...}

static int ggT(int a, int b)
                {...}

static int round(float x)
                {...}

.....
}
```

MethodDeclaration:	*MethodHeader MethodBody*
MethodHeader:	*{ Modifier } ResultType MethodDeclarator*
ResultType:	*Type* | `void`
MethodDeclarator:	*Identifier* `(` *[FormalArguments]* `)`
FormalArguments:	*FormalArgument* { `,` *FormalArgument* }
FormalArgument:	*Type Identifier*
MethodBody:	`;` | *Block*
Modifier:	`static` | . . . . .

MethodHeader z.B.

```
int  ggT(int a, int b)
void writeMultiple(char c, short n)
long getTime()
```

Prozedurkopf (Methodenkopf, *method header*) enthält

❶ **Name** der Prozedur, z.b. `ggT` ,

❷ **Formale Parameter** der Prozedur mit Typ und Namen,
z.B. `int a, int b` ,
in der Syntax als *formale Argumente (arguments)* bezeichnet,

❸ **Ergebnistyp** *(result type)*, z.B. `int` ;
eine Prozedur, deren Aufruf einen Ergebniswert liefert,
heißt auch **Funktionsprozedur** *(function procedure)*,
sonst - in Java durch Schlüsselwort `void` („leer") bezeichnet-
eigentliche Prozedur *(proper procedure)*.

Gültigkeitsbereich eines Methodennamens ist die ganze Klasse

(und evtl. auch andere Klassen)

Formale Argumente:

- *Gültigkeitsbereich* ist der ganze Methodenrumpf
- Verwendbar wie *lokale Variable*, z.B.

```
long fact(int n) { // factorial function
     long result = 1;
     while(n > 1) result *= n--;
     return result;   }
```

- Vorsicht vor Namenskollision mit lokalen Variablen!

Prozedurrumpf (Methodenrumpf, *method body)*

ist *leer* (`;`) oder besteht aus einem *Block* `{.....}`

Im Methodenrumpf kann man neben den

- *Parameternamen* und den anderen
- *lokalen Namen* auch
- **nichtlokale Namen** *(non-local names)* (1.3.2 ←)

benutzen, z.B. den Namen einer in der gleichen Klasse vereinbarten Methode (zwecks Aufrufs dieser Methode),

z.B.:

```
class Combinatorics {

// factorial
static long fact(int n) { // factorial function
     long result = 1;
     while(n > 1) result *= n--;
     return result;   }

// binomial coefficient „n over k“
static long binom(int n, int k) {
     return fact(n)/fact(k)/fact(n-k);}

}
```

Beendigung der Ausführung einer Prozedur - zur Erinnerung (1.3.5←):

JumpStatement: *BreakStatement*
ContinueStatement
GotoStatement
ReturnStatement

. . . . .

ReturnStatement: `return` *[Expression]* `;`

Rücksprung *(return statement)*

Ende und **Beendigung** einer Prozedur:

Statisches Ende = Blockende `}`

Dynamische Beendigung:

- entweder beim statischen Ende
- oder vorzeitig durch Rücksprung mit einem (von evtl. mehreren)

  `return;` *bei eigentlicher Prozedur* bzw.

  `return` *[Expression] ;* *bei Funktionsprozedur:*
  - Prozedur *muss* so beendigt werden*,
  - der Wert des *Expression* ist der Ergebniswert,
  - sein Typ muss mit dem im Kopf angegebenen Ergebnistyp *verträglich* sein.

* sonst statischer Fehler!

1.4.2 Prozeduraufruf

(procedure call, method invocation, ...)

ist entweder **Anweisung** - falls eigentliche Prozedur -
z.B. `print(pi)`

oder **Ausdruck** - falls Funktionsprozedur -
z.B. `ggt(60,x)`

! **In Java** ist ein **Methodenaufruf** *(method invocation)* Bestandteil

entweder einer Anweisung mit Semikolon, z.B.

```
print(pi);  ggt(60,x);
```
(!)

oder eines Ausdrucks, z.B.

```
1 + ggt(60,x)
```

Methodenaufruf in Java	verwendbar in Ausdruck	verwendbar für Anweisung
mit Ergebniswert	JA	JA
`void`	NEIN	JA

```
int i = 1 + ggt(60,x);      erlaubt
ggt(60,x);                  erlaubt !
1 + ggt(60,x);              verboten

print(pi);                  erlaubt
float f = print(pi);        verboten
```

Erweiterung der Syntax aus 1.2, 1.3:

```
Primary:                .....
                        MethodInvocation

ExpressionStatement:    StatementExpression ;

StatementExpression:    .....
                        MethodInvocation


MethodInvocation:       MethodIdentifier ( [ ActualArguments ] )

ActualArguments:        Expression { , Expression }
```

+ *Kontextbedingung:*

  Methodenaufruf kann als *Primary* nur dann verwendet werden, wenn die Methode nicht `void` ist.

+ *Typkorrektheit* betr. Argumente und Ergebnis:

  die **tatsächlichen Argumente** *(actual arguments)* müssen typverträglich mit den formalen Argumenten sein;

  der Ergebnistyp muss zum Aufrufkontext passen.

Semantik des Methodenaufrufs:

1. **Argumentübergabe:**
   wirkt wie *Mehrfachzuweisung* (1.3.1 ←)
   der tatsächlichen Argumente an die formalen Argumente.

2. **Sprung** an den Beginn des Methodenrumpfs,
   Ausführung des Methodenrumpfs.

3. **Beendigung** des Methodenrumpfs, gegebenenfalls bei `return;`
   falls `return` *Expression*, Berechnung des Ergebniswerts
   durch Auswertung des *Expression*.

4. **Rücksprung** an die Aufrufstelle und dort Fortsetzung der
   Ausführung bzw. Auswertung.

1.4.3 Rekursion

ist in fast allen imperativen Sprachen möglich

Beispiel 1: größter gemeinsamer Teiler

```
int gcd(int a, int b) {
    return b==0 ? a : gcd(b, a%b);
    }
```

Beispiel 2: Türme von Hanoi

```
void hanoi(int size, char src, char dst, char aux) {
     if(size>0) {
        hanoi(size-1, src, aux, dst);
        System.out.print(src); System.out.println(dst);
        hanoi(size-1, aux, dst, src); }
     }
```

Verschränkte Rekursion

```
class C {

static void some(...) {
    .....
    if(...) other(...);
    .....
    }
static void other(...) {
    .....
    if(...) some(...);
    .....
    }
}
```

(Beispiele später)

1.4.4 Parameterübergabe

(parameter passing, parameter mechanisms)

Beachte: Die Argumentübergabe bei Java ist nur eine von mehreren möglichen Varianten der Parameterübergabe.
Diese Übergabeart wird auch als

Wertübergabe *(pass-by-value, call-by-value)*

bezeichnet. Ein so übergebener Parameter heißt auch

Wertparameter *(value parameter)* oder **Argument**.

Varianten der Parameter/Übergabe:

Wertparameter (value parameter)	Wertübergabe (pass-by-value, call-by-value)
Ergebnisparameter (result parameter)	Ergebnisübergabe (pass-by-result, call-by-result)
Wert/Ergebnisparameter (value/result parameter)	Wert/Ergebnisübergabe (pass-by-value/result, call-by-value/result)
Variablenparameter (variable parameter)	Variablenübergabe (pass-by-reference, call-by-reference)
Namensparameter (name parameter)	Namensübergabe (pass-by-name, call-by-name)

u.a.

1.4.3.1 Ergebnisparameter

FormalParameter:	`OUT` *Type Identifier*
ActualParameter:	*Variable*

+ *Typkorrektheit:* Formaler Parameter muss typverträglich mit dem tatsächlichen Parameter sein.

Semantik:

Beim Rücksprung erfolgt eine Mehrfachzuweisung der Werte der formalen Ergebnisparameter an die tatsächlichen Parameter - d.h. Semantik ist „spiegelbildlich zur Wertparameter-Semantik".

Ergebnisparameter gibt es z.B. in **Ada** (1979).

Beispiel 1:

```
void intdiv(int dividend, int divisor,
            OUT int quotient, OUT int remainder) {
     quotient  = dividend / divisor;
     remainder = dividend % divisor;
}

...; intdiv(a+3, 4711, quotient, rest); ...

     intdiv(a+3, 4711, 0, 0);
```

ist falsch

Beispiel 2: Ein/Ausgabe

```
... read(nextchar); write(nextchar); ...
```

in Java nicht möglich

1.4.3.2 Wert/Ergebnisparameter

FormalParameter:	`IN OUT` *Type Identifier*
ActualParameter:	*Variable*

+ *Typkorrektheit:* Formaler Parameter und tatsächlicher Parameter müssen gleichen Typ haben

Semantik:

Kombination von Wertübergabe und Ergebnisübergabe.

Wert/Ergebnisparameter gibt es z.B. in **Ada**.

Beispiel:

```
void incr(IN OUT int variable, int value){
     variable += value;
}
```

```
...; incr(x, y+z); ...
```

```
     incr(x+y, z);
```

ist falsch

1.4.3.3 Variablenparameter

(pass-by-reference)

FormalParameter:	`VAR` *Type Identifier*
ActualParameter:	*Variable*

\+ *Typkorrektheit:* Formaler Parameter und tatsächlicher Parameter müssen gleichen Typ haben

Semantik:

Formaler Parameter ist *Synonym (!)* für tatsächlichen Parameter.

➔ Über Variablenparameter können *Effekte* bewirkt werden !

Variablenparameter gibt es z.B. in *Pascal, Modula, C#* (`ref`, `out`)

Beispiel 1:

```
void incr(VAR int variable, int value){
     variable += value;
}

...; incr(x, y+z); ...

     incr(x+y, z);
```

ist falsch

Beispiel 2:

```
void incr2(VAR int a, VAR int b) {
      a += b;
      a += b;
}
```

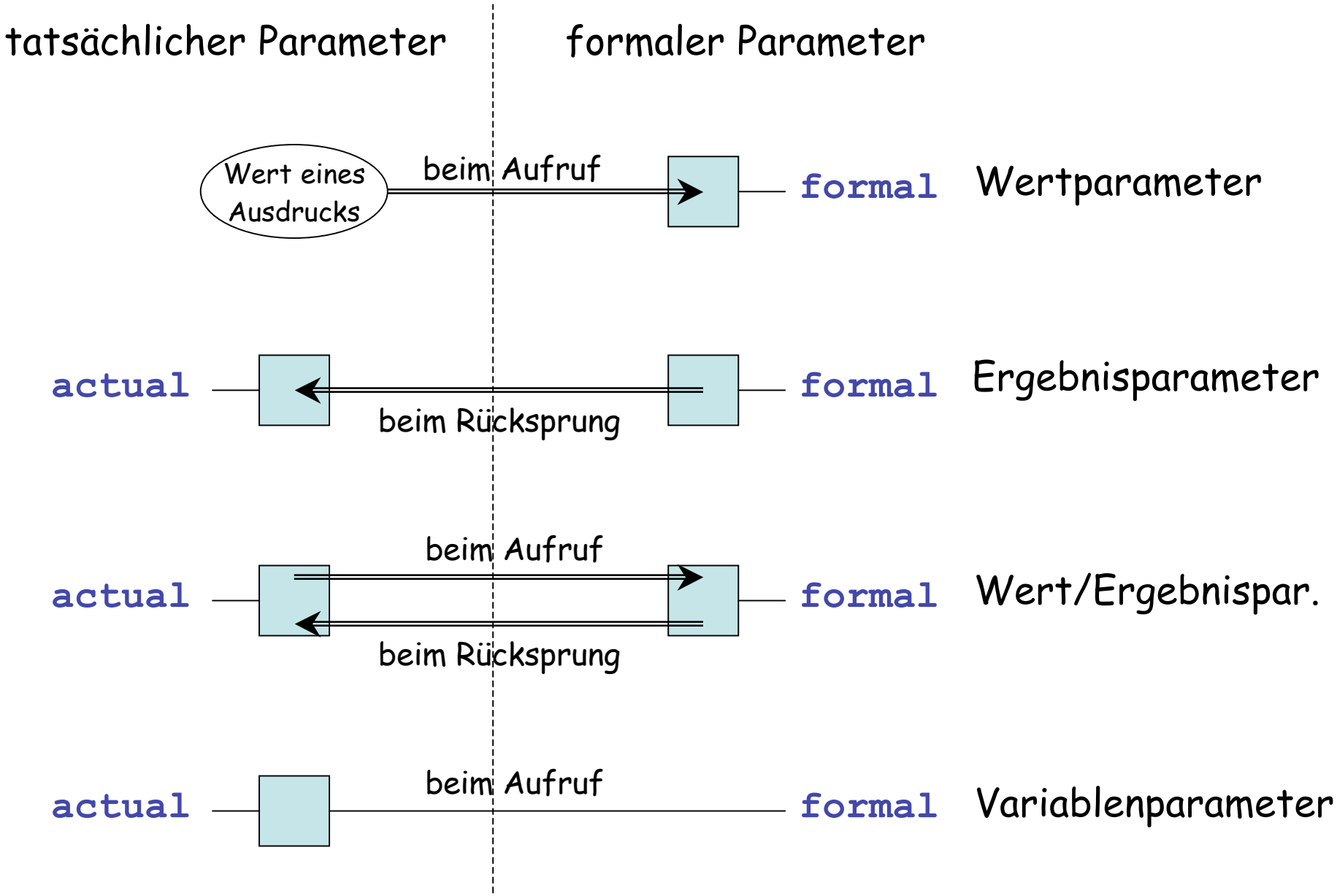
```
                incr2(x, y+z);
```
ist falsch

```
x = y = 1; incr2(x,y);
```
{ x = 3 , y = 1 }

```
x = y = 1; incr2(x,x);
```
{ x = 4 , y = 1 } !

Achtung: mit **IN OUT** statt **VAR** haben wir

```
x = y = 1; incr2(x,x);
```
{ x = 3 , y = 1 } !


tatsächlicher Parameter
formaler Parameter
Wert eines Ausdrucks
beim Aufruf
formal
Wertparameter
actual
formal
Ergebnisparameter
beim Rücksprung
beim Aufruf
actual
formal
Wert/Ergebnispar.
beim Rücksprung
beim Aufruf
actual
formal
Variablenparameter

1.4.3.4 Namensparameter

(call-by-name)

FormalParameter:	**NAME** *Type Identifier*
ActualParameter:	*Expression*

+ *Typkorrektheit:* Formaler Parameter und tatsächlicher Parameter müssen gleichen Typ haben

+ *Kontextbedingung:* Wenn der Prozedurrumpf eine *Zuweisung* an den formalen Parameter enthält, muss der tatsächliche Parameter eine Variable sein.

Semantik:

Die aufgerufene Prozedur verhält sich so, als wäre in ihrem Rumpf überall der formale Parameter durch den tatsächlichen Parameter *textuell ersetzt (!)* - evtl. mit geeigneter Umbenennung der dort auftretenden Variablen zwecks Vermeidung von Namenskollisionen.

Namensparameter gibt es z.B. in *Algol 60* und in *Simula*. Sie sind erheblich ineffizienter als Variablenparameter oder Ergebnisparameter - die fast das Gleiche leisten (nämlich Effekte über Parameter zu bewirken).

Beispiel:

```
int If(boolean Cond, NAME int Then, NAME int Else){
    return Cond ? Then : Else ;
}

... result = If(x==0, 1, y/x); ...
```

Prozedurrumpf wirkt wie `return 0 ? 1 : y/x ;`

d.h. `y/x` *wird nicht ausgewertet !*

Merke:

- In ihren Namensparametern ist eine Prozedur **nicht strikt**.
- Das obige `If` ist in Java *nicht formulierbar!*

Vergleich mit funktionalen Sprachen:

imperativ	applikativ
call-by-value	*eager evaluation* *(innermost reduction,* *applicative-order reduction)*
call-by-name	*lazy evaluation* (Haskell) *(outermost reduction,* *normal-order reduction)*

1.4.5 Prozedurtypen

Zur Erinnerung:

In der funktionalen Programmierung haben nicht nur Daten, sondern auch *Funktionen einen Typ - ihre Signatur* - und können als Argumente übergeben werden.

Analogon in der imperativen Programmierung:

Prozedurtyp *(procedure type)* oder **Signatur** *(signature)*

Beispiel: `int m(float a, int b)`

hat die Signatur `int (float, int)`

(Haskell: `float -> int -> int`)

☛ Variable oder Parameter mit Prozedurtyp gibt es **in Java nicht** (sehr wohl aber in *Modula*, *C/C++*, *C#* u.a.).

Syntax in *Modula* (vereinfacht) :

ProcedureType:	`PROCEDURE` *[FormalTypes]*
FormalTypes:	`(` *[[* `VAR` *] Type { ,* *[* `VAR` *] Type }]* `)` *[* `:` *Type]*
Type:	*Identifier*

Beispiele für Definition und Anwendung von Prozedurtypen:

```
TYPE function  = PROCEDURE(REAL): REAL;

TYPE integrate = PROCEDURE(REAL,REAL,function): REAL;


PROCEDURE twice(function f, REAL arg): REAL;
BEGIN RETURN f(f(arg)) END;

PROCEDURE square(REAL arg): REAL;
BEGIN RETURN arg*arg END;

VAR s: function;
... s := square; x := s(x); x := twice(s,x) ...
```

Beachte:

Auch für einen Parameter eines Prozedurtyps sind prinzipiell alle Übergabemechanismen realisierbar.