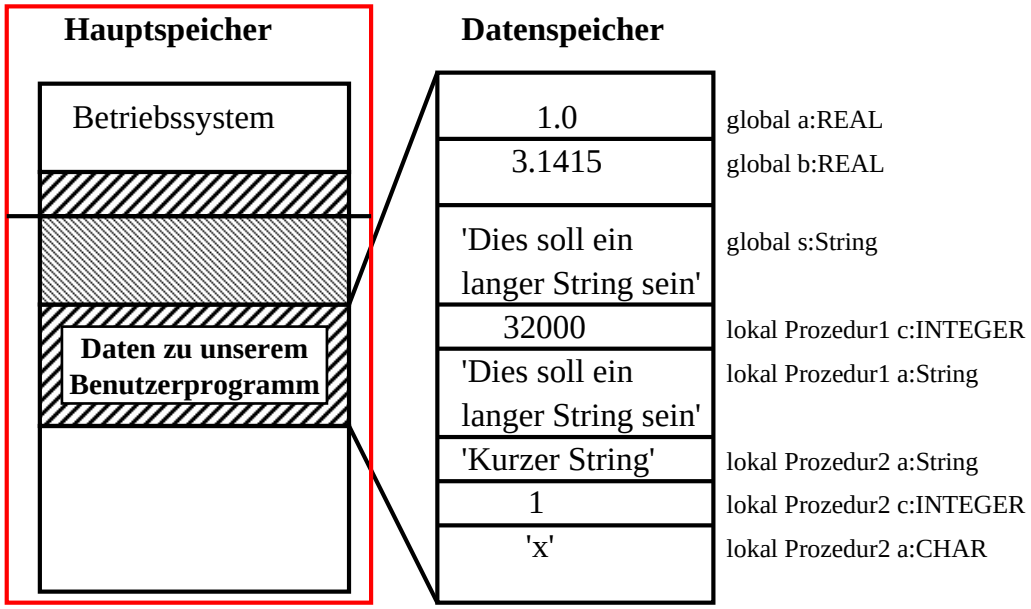
Rückblick Speicherverwaltung

Im Datenbereich des entsprechenden Programms werden die globalen und lokalen Variablen gespeichert.



Compiler

Funktionsweise eines Compilers am Beispiel von Modula-2

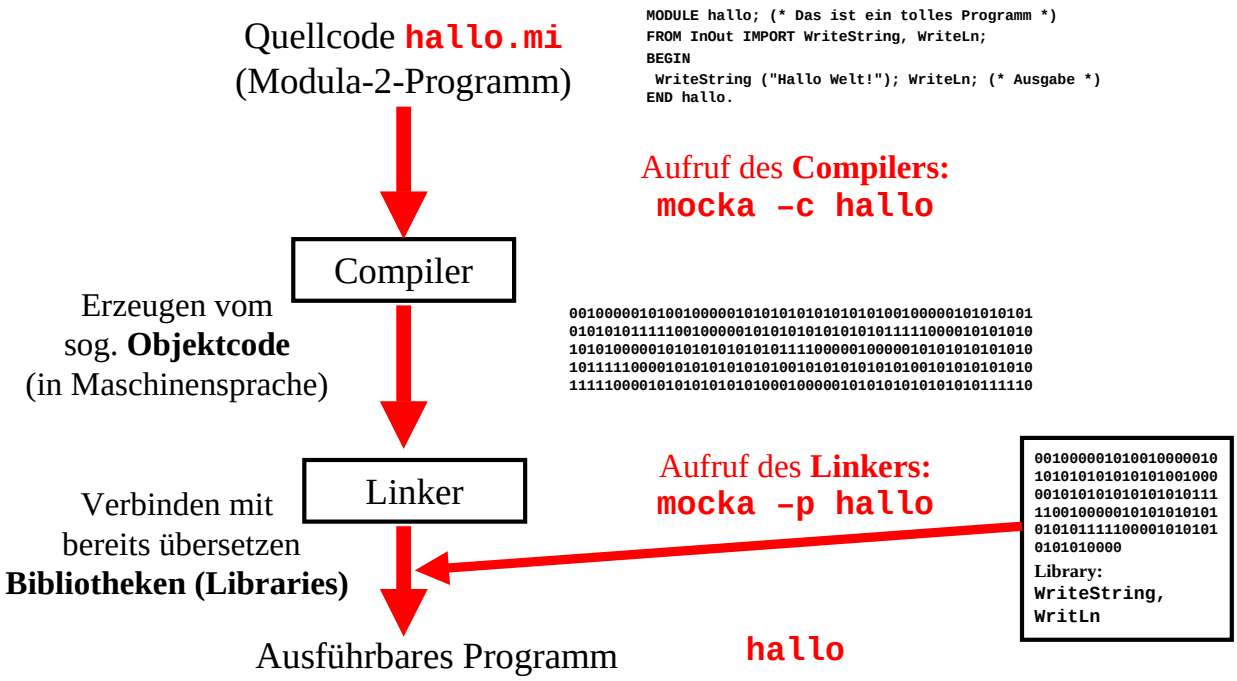
- Gegeben sei folgendes Modula-2-Programm:

```
MODULE hallo;
(* Das ist ein tolles Programm *)
FROM InOut IMPORT WriteString, WriteLn;
BEGIN
  WriteString ("Hallo Welt!"); (* Ausgabe *)
  WriteLn;
END hallo.
```

- Programmtext (**Quellcode**) unter dem Namen **hallo.mi** speichern
  Die Endung "**mi**" gilt zur Kennzeichnung als Modula-2-Code.

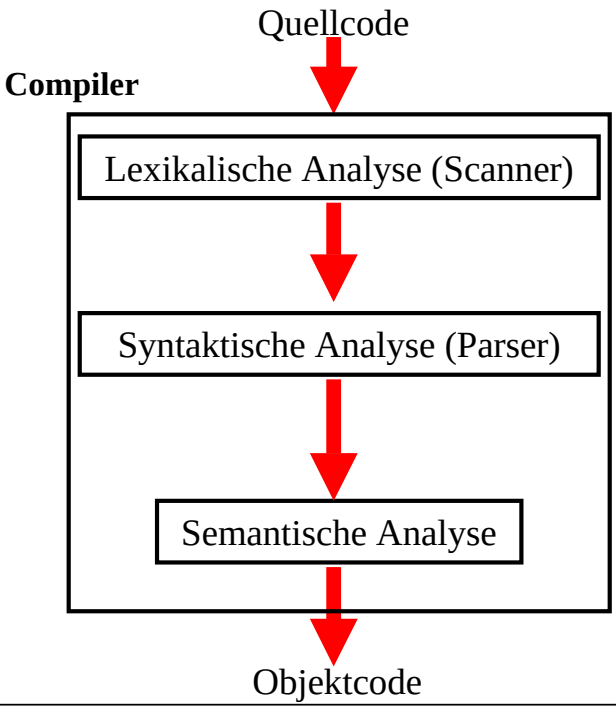
Compiler

Funktionsweise eines Compilers am Beispiel von Modula-2



Compiler

Detaillierter Ablauf während des Übersetzens:



Erkennen von **lexikalischen Quelltextbestandteilen**, z.B. Schlüsselwörtern, Bezeichnern...

Überprüfung auf **syntaktische Korrektheit** (z.B. anhand von Syntaxdiagrammen)

Überprüfen der **Semantik** (sind alle Bezeichner definiert, Test der Signaturen etc.)

Die Parameterübergabe beim Prozeduraufruf

Beim **Prozeduraufruf** werden die Argumente durch **aktuelle Parameter** ersetzt.

Man unterscheidet hauptsächlich zwei Arten der Übergabe:

- **Call by value**
- **Call by reference**

Call by value

Erforderliche Syntax: `Proc(x:INTEGER)`

Ablaufbeschreibung beim Prozeduraufruf:

- Von der übergebenen Variablen wird eine **Kopie** angelegt.
- Die übergebene Variable behält ihren alten Wert.
- Die Kopie ist nur **während** der Abarbeitung der Prozedur gültig.
- Die Kopie kann innerhalb der Prozedur verändert werden.
- Nach Verlassen der Prozedur wird die Kopie aus dem Speicher entfernt.

Parameterübergabe

Beispiel: Call by value

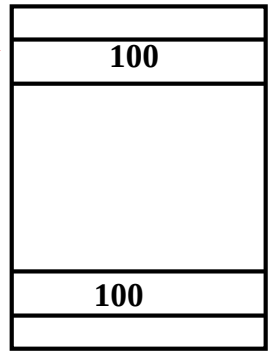
Gegeben sei eine Prozedur `Proc(x:INTEGER);`

Innerhalb des Hauptprogramms steht ein Aufruf: `Proc(b);`

Datenspeicher der Applikation

Im Hauptprogramm wird die Variable `b:=100` definiert und an eine Speicherstelle geschrieben.

Beim Prozeduraufruf `Proc(b)` wird eine Kopie von b an eine (andere) Stelle im Speicher geschrieben.



Parameterübergabe

Call by reference

Erforderliche Syntax: `Proc(VAR x:INTEGER)`

Ablaufbeschreibung beim Prozeduraufruf:

- Es wird ein **Verweis (Referenz)** auf die übergeben Variable angelegt.
- Der Wert der übergebenen Variable kann **direkt** von der Prozedur verändert werden, die übergeben Variable **ändert sich mit**.
- Alle Änderungen während der Prozedur bleiben auch nach deren Beendigung **erhalten**.
- **Vorteil:** Es wird kein zusätzlicher Speicherplatz benötigt.

Parameterübergabe

Beispiel: Call by reference

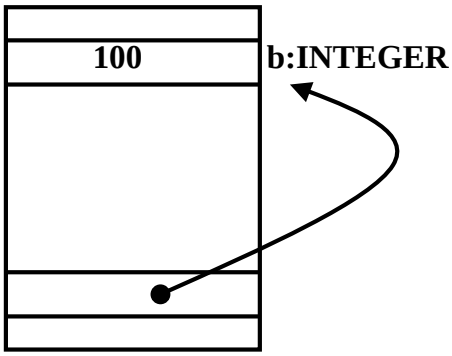
Gegeben sei eine Prozedur **`Proc(VAR x:INTEGER);`**

Innerhalb des Hauptprogramms steht ein Aufruf: **`Proc(b);`**

Datenspeicher der Applikation

Im Hauptprogramm wird die Variable **`b:=100`** definiert und an in eine Speicherstelle geschrieben.

Beim Prozeduraufruf **`Proc(b)`** wird (an einer anderen Stelle im Speicher) ein **Verweis auf die Adresse von b** erstellt.



Parameterübergabe

Beispiel: "Call by value" vs. "Call by reference"

```
MODULE CallByValue;
VAR a,b: INTEGER;
:
PROCEDURE Test1(x:INTEGER):INTEGER;
BEGIN
   x:=x+1;
   RETURN x;
END Test1;
:
BEGIN
   a:=3;
   b:= Test1(a);
   WriteInt(a);  (*a=3*)
   WriteInt(b);  (*b=4*)
END CallByValue.
```

```
MODULE CallByReference;
 VAR a,b:INTEGER;
 :
PROCEDURE Test2(VAR x:INTEGER):INTEGER;
  BEGIN
     x:=x+1;
     RETURN x;
  END Test2;
  :
 BEGIN
     a:=3;
     b:=Test2(a);
     WriteInt(a);     (*a=4*)
     WriteInt(b);     (*b=4*)
 END CallByReference.
```

Parameterübergabe

Hinweise zu Call by value:

- "Call by value" verhindert die Änderung von Variablen, die an anderer Stelle ebenfalls verwendet werden.
- Benutze "Call by value" wenn eine Änderung der globalen Parameter durch die Prozedur nicht erwünscht ist. (➔ Seiteneffekt)
- **Vorsicht:** "Call by value" ist für komplexe Strukturen sehr aufwendig.

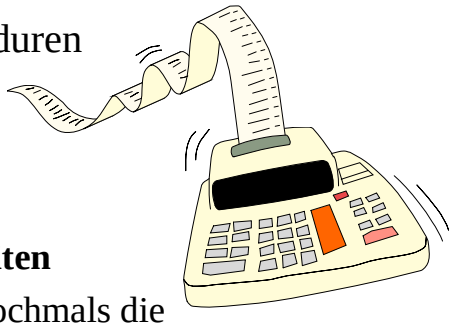
Prozeduren (Zusammenfassung)

Vorteile der Modularisierung durch Prozeduren und Funktionen:

1) **Wiederverwendung** in anderen Programmteilen möglich.
   - einmaliger Implementierungsteil
   - beliebig viele Aufrufe
2) **Lesbarkeit** der Programme wird besser:
   - Trennung der Implementierung vom Aufruf
3) **Modifikationen** werden erleichtert.
   - Änderung nur im Implementierungsteil der Prozedur/Funktion
   - Ausnahme: Änderungen der Signatur
4) **Fehlersuche** wird erleichtert.
   - Jede Prozedur/Funktion kann **lokal verifiziert** werden.

Prozeduren - Modularität

Ein zusammenfassendes Beispiel zu Prozeduren



Realisierung des Taschenrechners

Zur Veranschaulichung der **Einsatzmöglichkeiten** von **Prozeduren,** ist auf den nächsten Seiten nochmals die Implementierung des Taschenrechners **zusammengefasst**.

Dazu wird der Taschenrechner

- mit den **Funktionen** +, - und * **ohne** Prozeduren,
- mit den **Funktionen** +, - und * **mit** Prozeduren und
- erweitert um die **Potenzrechnung**

angegeben.

Prozeduren - Modularität

Beispiel: Programm Taschenrechner (**ohne** Prozeduren):

```
MODULE Taschenrechner;
FROM InOut IMPORT Read, ReadInt, WriteInt,
                  WriteString, WriteLn;
VAR x,y,erg    :INTEGER;
    op         :CHAR;
BEGIN
   WriteString('Hallo, ich bin ein Taschenrechner.'); WriteLn;
   WriteString('Eingabe x: '); ReadInt(x); WriteLn;
   WriteString('Eingabe y: '); ReadInt(y); WriteLn;
   WriteString('Welche Operation (+,-,*)? '); Read(op);
   CASE op OF
       '+' : erg:= x+y
    |  '-' : erg:= x-y
    |  '*' : erg:= x*y
    ELSE WriteString('Falsche Eingabe');
   END;
   WriteString('Ergebnis: '); WriteInt(erg); WriteLn;
END Taschenrechner.
```

Prozeduren - Modularität

Beispiel: Programm Taschenrechner (**mit** Prozeduren):

```
MODULE Taschenrechner;
FROM InOut IMPORT Read, ReadInt, WriteInt, WriteString, WriteLn;
VAR x,y,erg     : INTEGER;  op : CHAR;
PROCEDURE Add(a,b:INTEGER):INTEGER;(...)
PROCEDURE Sub(a,b:INTEGER):INTEGER;(...)(* siehe nächste Seite*)
PROCEDURE Mul(a,b:INTEGER):INTEGER;(...)
BEGIN
   WriteString('Hallo, ich bin ein Taschenrechner.'); WriteLn;
   WriteString('Eingabe x: '); ReadInt(x); WriteLn;
   WriteString('Eingabe y: '); ReadInt(y); WriteLn;
   WriteString('Welche Operation (+,-,*)? '); Read(op); WriteLn;
   CASE op OF
       '+' : erg:= Add(x,y)
    |  '-' : erg:= Sub(x,y)
    |  '*' : erg:= Mul(x,y)
    ELSE WriteString('Falsche Eingabe'); WriteLn;
   END;
   WriteString('Ergebnis: '); WriteInt(erg); WriteLn;
END Taschenrechner.
```


Prozeduren - Modularität

Beispiel: Programm Taschenrechner (Prozeduren):

```
PROCEDURE Add(a,b:INTEGER):INTEGER;
VAR res: INTEGER;
BEGIN
     res:=a+b;
     RETURN res;
END Add.
PROCEDURE Sub(a,b:INTEGER):INTEGER;
VAR res: INTEGER;
BEGIN
     res:=a-b;
     RETURN res;
END Sub.
PROCEDURE Mul(a,b:INTEGER):INTEGER;
VAR res: INTEGER;
BEGIN
     res:=a*b;
     RETURN res;
END Mul.
```

Prozeduren - Modularität

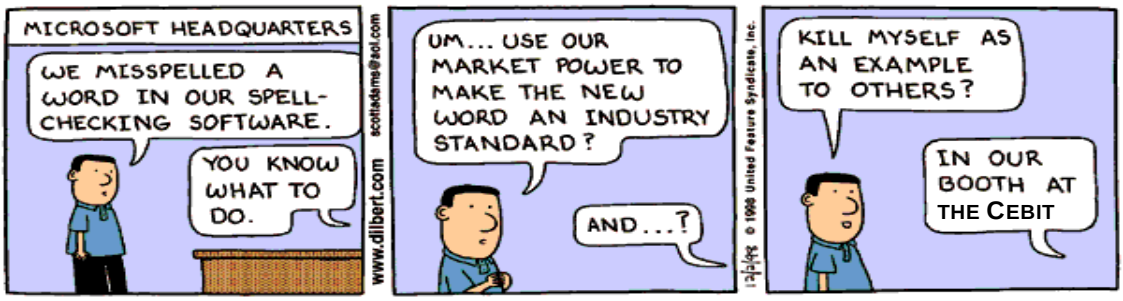
Beispiel: Programm Taschenrechner (Erweiterung um **Potenzrechnung**):

```
MODULE Taschenrechner;
FROM InOut IMPORT Read, ReadInt, WriteString, WriteLn, WriteInt;
VAR x,y,erg    : INTEGER;   op : CHAR;
PROCEDURE Add(a,b:INTEGER):INTEGER;  (...)
PROCEDURE Sub(a,b:INTEGER):INTEGER;  (...)
PROCEDURE Mul(a,b:INTEGER):INTEGER;  (...)
PROCEDURE Pot(a,b:INTEGER):INTEGER;  (...)
BEGIN
   ...   (* Eingabe von x und y *)
   WriteString('Welche Operation (+,-,*,^)? '); Read(op);
   CASE op OF
       '+' : erg:= Add(x,y)
    |  '-' : erg:= Sub(x,y)
    |  '*' : erg:= Mul(x,y)
    |  '^' : erg:= Pot(x,y)
    ELSE WriteString('Falsche Eingabe'); WriteLn;
   END;
   WriteString('Ergebnis: '); WriteInt(erg); WriteLn;
END Taschenrechner.
```


Prozeduren - Modularität

Beispiel: Programm Taschenrechner (**Prozedur Pot**):

```
PROCEDURE Pot(basis,exp:INTEGER):INTEGER;
VAR i,res: INTEGER;
BEGIN
   res:=1;
   FOR i:=1 TO exp DO
        res:=res*basis;
   END;
   RETURN res;
END Pot.
```



Rekursionen

In den meisten Programmiersprachen (wie auch MODULA-2) ist es möglich, dass eine Prozedur **sich selbst** aufruft.

Solch eine Selbst-Reaktivierung wird auch **Rekursion** genannt.

- Bei jedem Rekursionsschritt wird eine neue **Inkarnation** (Instanz) der jeweiligen Prozedur erzeugt.
- Viele Algorithmen in der Informatik und Mathematik sind selbstbezüglich (rekursiv) definiert. In solchen Fällen bietet sich eine rekursive Implementierung/Realisierung an.

Beispiel: Berechnung der **Fakultät n!** einer natürlichen Zahl n

Die **mathematische Fakultät** ist rekursiv definiert durch

$$0! = 1$$

$$1! = 1$$

$$n! = n \cdot (n-1)! \quad \text{(für alle } n \in \mathbb{N} \text{ mit } n>1\text{)}$$

Rekursionen

Jede Rekursion muss irgendwann **terminieren**.

- Deswegen muss in jeder rekursiven Prozedur vor dem jeweils nächsten **Rekursionsschritt** überprüft werden, ob die **Terminierungsbedingung** erfüllt ist.
  (z.B. mit einer IF-Anweisung)
- Ansonsten, kommt es irgendwann zu einem Programmabbruch durch einen **Speicherüberlauf** (**stack overflow**),
  da jeder Rekursionsschritt neuen Speicherplatz benötigt.
- Zur Übersetzungszeit kann normalerweise **nicht** überprüft werden, ob alle Rekursionen ordnungsgemäß terminieren.
- Die maximale **Rekursionstiefe** (depth of recursion) ist also von der Größe des zur Verfügung stehenden Speichers abhängig.

Rekursionen

Beispiel: Berechnung der **Fakultät** n! einer natürlichen Zahl n

```
PROCEDURE Fakultaet(n:CARDINAL):CARDINAL;
BEGIN
       IF n<=1 THEN RETURN 1
        ELSE RETURN n * Fakultaet(n-1);
       END;
END Fakultaet.
```

Terminierungs-bedingung

Beispielaufruf:

```
x:=Fakultaet(3);
```

Die Prozedur ruft solange sich selbst auf, bis die Abbruchbedingung `n<=1` erreicht ist.

Endergebnis: `x:=6;`

Rekursionen

Jede **Rekursion** kann auch als **Iteration** dargestellt werden.

Beispiel: Iterative Darstellung der Prozedur Fakultaet.

```
PROCEDURE Fakultaet(n:CARDINAL):CARDINAL;
VAR res:CARDINAL;
BEGIN
   res := 1;
   WHILE n >= 1 DO
        res := n * res;
        n:= n-1;
   END;
   RETURN res;
END Fakultaet.
```

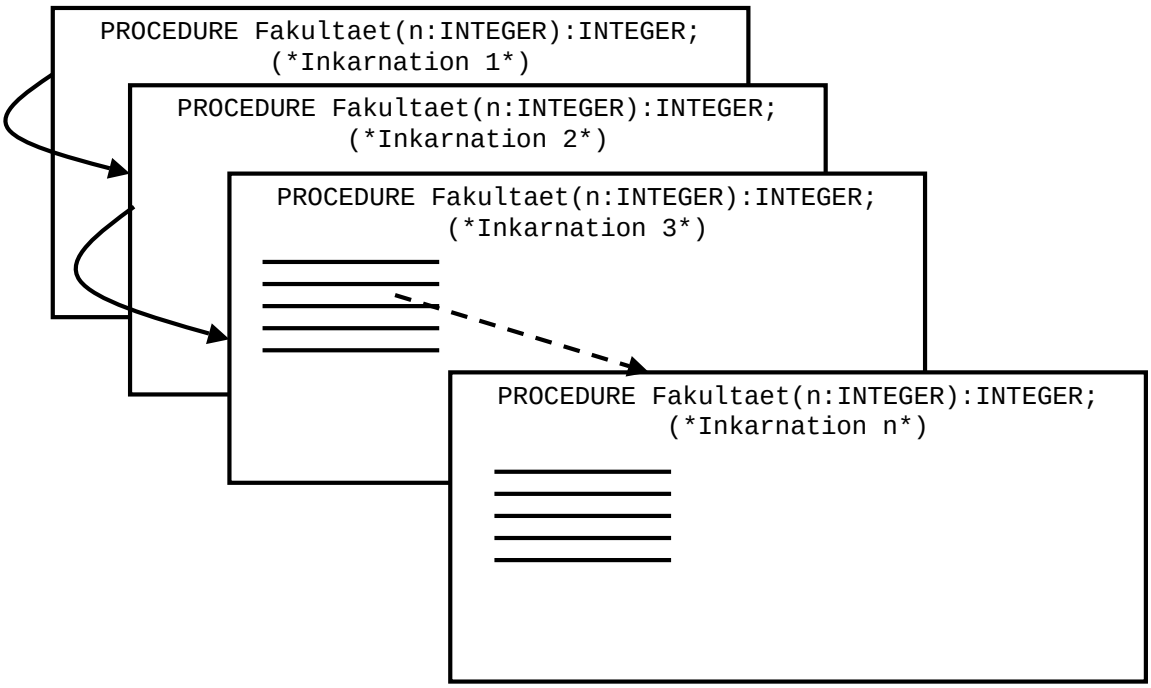
Rekursionen

Frage: Ist die rekursive oder die iterative Lösung besser?

- Eine iterative Prozedur wird das Ergebnis normalerweise **effizienter** berechnen als die entsprechende rekursive Lösung, da Schleifenkonstrukte **weniger Speicherplatz und weniger Zeit** als Rekursionen benötigen.
- Allerdings ist es oft **sehr schwierig** für eine gegebene rekursive Formulierung eine iterative Lösung zu finden. Das iterative Programm selbst ist dann auch meistens **viel umfangreicher und sehr unübersichtlich**.

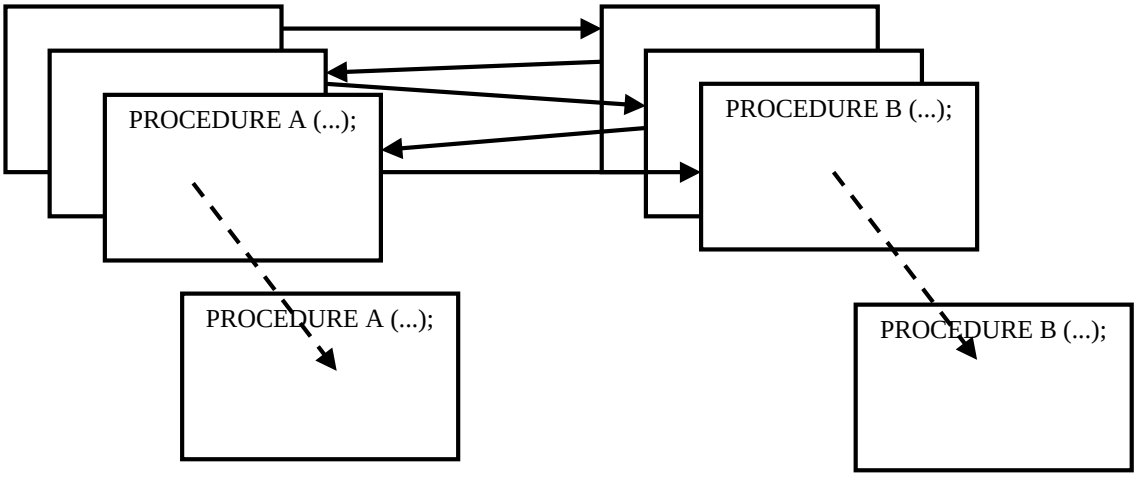
Direkte Rekursion

Bei der **direkten Rekursion** ruft die Prozedur **sich selbst** auf:



Indirekte Rekursion

Bei der **Indirekten Rekursion** wird das ursprüngliche Programm wieder von einer anderen Prozedur (hier B) aufgerufen.



Strukturierte Datentypen

Bis jetzt kennen wir nur **einfache Datentypen** (**`INTEGER`**...).

Wie stellt man **strukturierte Datentypen** in Modula-2 dar?

Beispiel: Übliche Strukturen aus der Mathematik

- Vektoren, Felder (array) $<a_1,a_2,a_3,.....,a_n>$
- Mengen (set) {a, b, c, ... , z }
- Point (x,y)

  etc.