Was ist Programmieren?

Grundlegende Konzepte

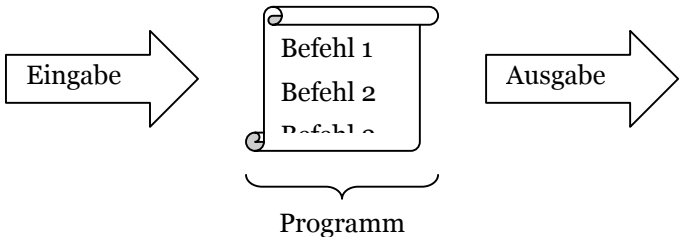
Was, wie und wann programmiert man?

Bis heute benutzt man Computer zur Automatisierung und schnelleren Abwicklung von Vorgängen, die zuvor von Menschen erledigt wurden. Oder aber die Computer unterstützen den Menschen bei seiner Arbeit mit zusätzlichen Informationen, die errechnet werden.

Alle Computer bestehen lediglich aus Hardware, welche rechnen kann, auf dem Bildschirm Graphiken anzeigen kann und Informationen vom Menschen über die Tastatur und Maus entgegennehmen kann. Alleine ist ein Computer eigentlich noch nutzlos; erst mit der Software, den Programmen gibt man dem Computer Anweisungen wie er sich in welchem Fall verhalten soll.

Im Grunde genommen ist ein Programm nichts anderes als eine für den Computer verständliche endliche Abfolge von Befehlen. Diese Abfolge von Befehlen wird in einer Programmiersprache verfasst.

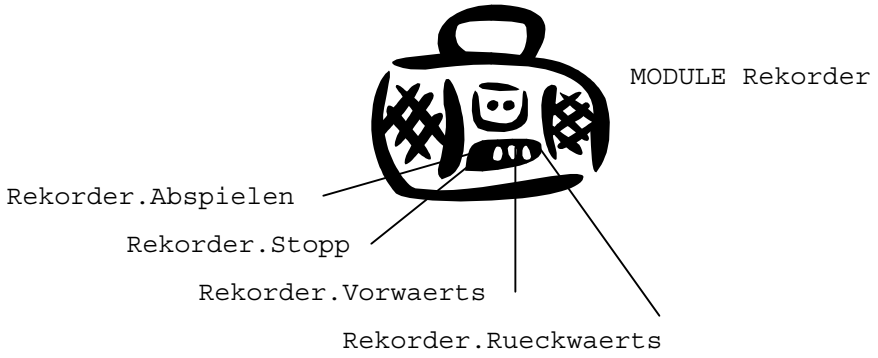
Ein Programm kann später, wenn es aufgerufen wurde, Eingaben des Benutzers entgegennehmen und daraus errechnete Resultate ausgeben. Eigentlich das Gleiche wie eine mathematische Funktion *f(Eingabe) = Ausgabe*.



Ein Mensch, der für ein gegebenes Problem einen Lösungsweg kennt, kann also auf einem Computer ein Programm schreiben, damit dieser dann für ein eingegebenes Problem eine Lösung errechnet und diese zum Beispiel auf dem Bildschirm ausgibt.

Module & Prozeduren: Wie funktionnieren Programme in Oberon?

Oberon ist eine Programmiersprache, mit der Programme verfasst werden können. In Oberon heissen diese einzelnen Programme Module. Jedes Modul enthält dabei einzelne Prozeduren. Zum Beispiel könnte ein Kassettengerät ein Modul sein und seine einzelnen Tastenfunktionen in den Prozeduren beschrieben sein:



Hier ist der Rekorder das Modul und Play, Stopp, Vorwaerts und Rueckwaerts sind Prozeduren, welche in diesem Modul untergebracht sind.

Die Syntax eines Moduls wird mit folgender EBNF definiert:

```
module = MODULE ident ";"  [ImportList]
DeclarationSequence [BEGIN StatementSequence] END
ident ".".

ImportList = IMPORT import {"," import} ";".

import = ident [":=" ident].
```

Das sieht jetzt ein bisschen kompliziert aus, aber eigentlich ist es ganz einfach. Hier das Modul unseres Rekorders:

```
MODULE Rekorder;
      PROCEDURE Abspielen*();
      BEGIN
            (* hier ist eine Abfolge von Befehlen *)
      END Abspielen;
      PROCEDURE Stopp*();
      BEGIN
            (* hier ist eine Abfolge von Befehlen *)
      END Stopp;
```

```
        PROCEDURE Vorwaerts*();
        BEGIN
              (* hier ist eine Abfolge von Befehlen *)
        END Vorwaerts;
        PROCEDURE Rueckwaerts*();
        BEGIN
              (* hier ist eine Abfolge von Befehlen *)
        END Rueckwaerts;
BEGIN
END Rekorder.
```

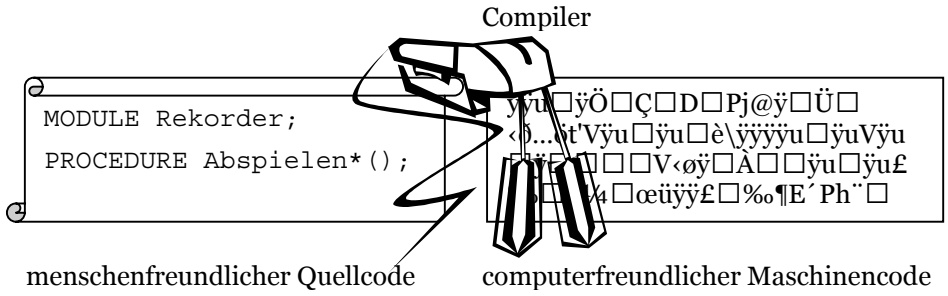
Prozeduren werden in Oberon mit folgender Syntax aufgerufen:

```
Modulname.Prozedurname;
```

Compiler: Weshalb muss ich eine genaue Grammatik befolgen?

Eine Programmiersprache definiert immer Grammatiken, welche dem Programmierer vorschreiben, in welcher Form er die gewünschten Module, Prozeduren und Befehle schreiben muss. Dies ist erforderlich damit der Computer danach sich ganz auf den Sinn des Programmes konzentrieren kann und somit auch das Programm sinngemäss ausführen kann.

Dazwischen liegt aber noch ein Schritt. Denn selbst diese scheinbar starre und maschinelle Programmiersprache ist für den Computer noch zu kompliziert, als dass er es schnell und richtig interpretieren könnte. Damit der Computer das Programm ausführen kann, muss es noch zuerst von einem Programm, dem Compiler kompiliert werden. Dieser Compiler macht aus dem menschenfreundlichen Quellcode einen maschinenfreundlichen Maschinencode.



Der Compiler überprüft die Syntax des Moduls, der darin enthaltenen Prozeduren und Befehlen. Stimmt das geschriebene mit der EBNF von Oberon überein, dann überprüft der Compiler, ob alle Typen stimmen. Das heisst, ob alle Zuweisungen legal sind. Man kann beispielsweise keine Zahlen in Boolsche Werte speichern und keine Buchstaben in Kommazahlen. Wenn alles korrekt war, dann übersetzt der Compiler das Modul in Maschinensprache.

Variablen & Typen: Wo speichere ich Zwischenresultate?

Meistens wenn man programmiert, möchte man ein vorheriges Resultat weiter bearbeiten und verändern. Dazu benötigt man Variablen. Variablen sind vergleichbar mit jenen in der mathematischen Algebra. In der Informatik verwendet man aber meist statt den nichts aussagenden Variablennamen wie zum Beispiel `x`, `y`, oder `i` aussagekräftigere Namen wie `Summe`, `Produkt` oder `Anzahl`.

In Oberon müssen die Variablen, welche man in einer Prozedur benutzen will, zuerst als solche definieren, so dass der Compiler weiss, dass es sich dabei um Variablen handelt. In Programmiersprachen haben Variablen im Gegensatz zur Mathematik einen Typ. Ein Typ sagt, welches Format von Daten eine Variable aufnimmen beziehungsweise enthalten kann. Folgende Basistypen gibt es in Oberon:

- `BOOLEAN`
  diese Variable kann entweder den Wert TRUE oder FALSE
- `CHAR`
  alle Buchstaben (Characters) in erweiterten ASCII Format
- `SHORTINT`, `INTEGER`, `LONGINT`
  alle ganzen Zahlen ohne Komma in einem bestimmten Bereich (In `SHORTINT` kann man nur kleine kurze Zahlen speichern, in `LONGINT` wesentlich längere)
- `REAL`, `LONGREAL`
  alle Zahlen mit Komma in verschiedenen Bereichen (In REAL sind es kleinere Zahlen, in `LONGREAL` sehr grosse Zahlen)
- `SET`
  ein `SET` enthält Werte zwischen 0 und 256

Wie oben erwähnt, müssen die Variablen, bevor sie in einer Prozedur verwendet werden können, deklariert werden. Die Syntax wird in EBNF folgendermassen definiert:

```
VariableDeclaration = IdentList ":" type.
```

Ein Beispiel für eine Prozedur, welche eine Variable benötigt, die aussagt, ob im Rekorder eine Kassette eingelegt ist oder nicht, würde folgendermassen aussehen:

```
PROCEDURE Abspielen*();
VAR istKassetteEingelegt: BOOLEAN;
VAR wurdeAbspieltasteGedrueckt: BOOLEAN;
VAR andereVariable1, andereVariable2: INTEGER;
BEGIN
      wurdeAbspieltasteGedrueckt := TRUE;
END Abspielen;
```

Im Wesentlichen kommen die Deklarationen immer zwischen dem `PROCEDURE` und dem `BEGIN` beziehungsweise bei Modulen zwischen dem `MODULE` und dem `BEGIN`.

Wie man oben sieht, kann man auch mehrere Variablen des gleichen Typs mit einem Komma trennen, so dass man nicht jedes mal den Typ erneut schreiben muss.

Wenn man den Wert einer Variablen haben möchte, um etwas damit zu berechnen, baut man den Variablennamen einfach statt den festen Werten in die Operation ein, im folgenden Beispiel die Addition. Wenn man den Wert eines Resultates, den man aus einer Operation erhält, in eine Variable speichern möchte, tut man dies mit einer Zuweisung. Das Zeichen für die Zuweisung ist `:= .` Was links steht erhält den Wert von dem was rechts steht:

```
Resultat1 := 5 + 6; (* Resultat1 hat jetzt Wert 11 *)
Resultat2 := andereVariable2 + andereVariable2;
(* Resultat2 hat jetzt den Wert der Summe von
andereVariable2 und andereVariable2 *)
```

Konstanten: wie speichere ich fixe Werte?

Gelegentlich möchte man im Programmcode einen festen Wert verwenden, welchen man einmal definiert. Zum Beispiel ist im Programmcode von Microsoft Word nicht jedes Mal hingeschrieben „`Microsoft Word 1.0`“, wenn das irgendwo steht, wahrscheinlich wird dann eine Konstante verwendet, wie zum Beispiel `ProduktName`. Wechselt nämlich einmal die Versionsnummer, so müsste man den Quellcode nach diesem „`Microsoft Word 1.0`“ durchsuchen, was umständlich und fehleranfällig wäre.

Konstanten sind also fixe Variablen, die nicht mehr verändert werden können. Für Konstantendeklarationen verwendet man folgende Syntax:

```
ConstantDeclaration = identdef "=" ConstExpression.
ConstExpression = expression.
```

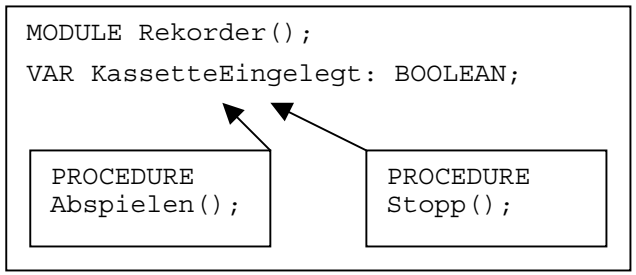
Oder im Rekorderbeispiel:

```
MODULE Rekorder;
CONST hersteller = "Sony";
BEGIN
      PROCEDURE Abspielen*();
      BEGIN
            (* hier ist eine Abfolge...
```

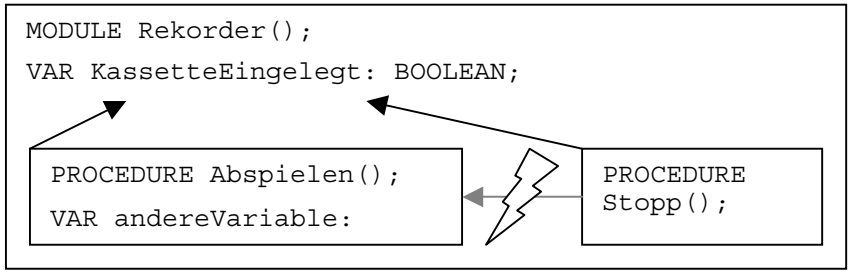
Scope: Von wo aus kann ich auf welche Variablen zugreifen?

Wie bei der Erklärung der Variablen kurz darauf hingedeutet wurde, können Variablen für Module oder auch nur für Prozeduren deklariert werden. Der Unterschied besteht im Bereich, von wo aus auf die bestimmte Variable zugegriffen werden kann, dem sogenannten Scope (Gültigkeitsbereich).

Deklariert man eine Variable im Modul (dann ist das `VAR` zwischen dem `MODULE` und dem `BEGIN`), dann können alle im Modul enthaltenen Prozeduren darauf zugreifen, das heisst deren Werte lesen und verändern - es ist dann eine globale Variable:

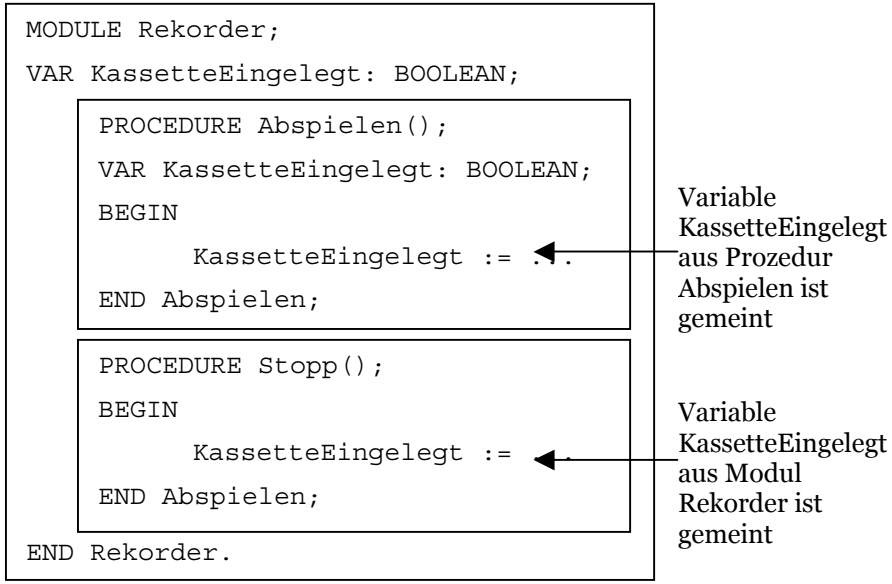


Hingegen, wenn die Variable in einer Prozedur deklariert wird, kann nur die Prozedur, aber nicht die andern Prozeduren des gleichen Moduls darauf zugreifen. Sie ist dann lokal.



Variablen gelten also immer nur dort, wo sie definiert wurden, oder aber dann in einer untergeordneten Prozedur, nicht aber in „Geschwisterprozeduren“, welche nicht untergeordnet sind.

Es kann aber jetzt zum Beispiel der Fall sein, dass eine Prozedur eine Variable mit dem selben Namen hat wie das Modul. Welche Variable ist dann gemeint?



Es ist also immer die zuletzt deklarierte Variable die momentan gültige.

Arrays: Wie kann ich eine Reihe von Variablen machen?

Oft möchte man eine Reihe von Werte speichern und verwenden. Zum Beispiel eine Messreihe, oder die letzten zehn Siege des Fussballclub Basel. Dafür benötigen wir eine neue Art von Variablen. Es handelt sich dabei um den Array. Ein Array kann eine feste Anzahl von Werten in Variablen eines bestimmten Typs aufnehmen. Zum Beispiel möchte ich für jeden Wochentag speichern, ob ich Vorlesungen habe oder nicht; für diesen Zweck nehme ich sieben BOOLEAN Variablen:

```
VAR WochentagVorlesungsfrei: ARRAY 7 OF BOOLEAN;
```

Auf die einzelnen der sieben Variablen kann ich folgendermassen zugreifen:

```
WochentagVorlesungsfrei[1] := TRUE;
```

Wobei die Zahl zwischen den eckigen Klammern bedeutet, das wievielte Element des Arrays gemeint ist. Die Nummerierung fängt immer bei 0 an! 1 würde jetzt also in unserem Beispiel Dienstag bedeuten. Der letzte Tag der Woche ist der Sonntag, welcher die Nummer 6 hat, und nicht die 7, weil eben die Nummerierung bei 0 anfängt.

Records: Wie kann ich Variablen gruppieren und so neue Variablentypen machen?

Records sind Definitionen von mehreren Typen zusammen, welche dann wieder einen neuen Typ ergeben. Zum Beispiel könnte man einen Personen-Typ erstellen, der dann wie folgt aussieht:

```
TYPE
 Person = RECORD
      vorname: ARRAY 50 OF CHAR;
      nachname: ARRAY 50 OF CHAR;
      passnummer: INTEGER
END;
```

In diesem Beispiel hat also die Person sozusagen drei Eigenschaften. Man kann jetzt neue Variablen mit dem Typ Person erstellen und danach auf die Eigenschaften zugreifen:

```
VAR p: Person;
p.vorname := "Hans"; p.nachname := "Meiser";
```

Diese Records kann man jetzt noch erweitern, indem man weitere Eigenschaften hinzufügt:

```
TYPE
 Student = RECORD(Person)
      immatrikuliert: BOOLEAN;
      semester: INTEGER;
END;
```

Nun hat der Typ Student 5 Eigenschaften: `vorname`, `nachname`, `passnummer`, `immatrikuliert` und `semester`. Stellen wir uns noch einen weiteren Typ vor:

```
TYPE
 Arbeiter = RECORD(Person)
      lohn: INTEGER;
END;
```

Was passiert jetzt, wenn wir Folgendes tun:

```
VAR p: Person; a: Arbeiter; s: Student;
p := a;
p.semester := 5;
```

Der Person p wurde ein Arbeiter a zugewiesen. Das ist möglich, weil der Arbeiter von der Person vererbt wurde. Aber wir wissen jetzt nicht, ob p eine Person, ein Arbeiter oder gar ein Student ist. Es gäbe jetzt zum Beispiel einen Fehler, wenn wir versuchen das Semester des Arbeiters zu setzen, denn der Arbeiter hat kein Semester!

Genau für dieses Problem gibt es den sogenannten Type Guard:

```
p(Student).semester := 5;
```

Dies garantiert uns, dass zuerst getestet wird, ob es sich um einen Studenten handelt, bevor da einfach ein Semester gesetzt wird. Ein kontrollierter Fehler tritt auf, wenn der Type Guard sagt, dass es sich nicht um einen Studenten handelt.

Wenn wir wissen wollen, ob eine bestimmte Variable den gewünschten Typ hat, können wir das mit dem IS testen:

```
IF ( p IS Student ) THEN ...
```

Hier wird geprüft, ob p ein Student ist, und erst dann etwas ausgeführt.

Anweisungen: Wie kann ich den Programmfluss kontrollieren und verändern?

Wie am Anfang dieses Skriptes erklärt, stehen in Oberon Anweisungen zur Verfügung (Kapitel Anweisungen in Oberon).

IF, LOOP, WHILE und REPEAT wurden erklärt. Zusätzlich gibt es eine inoffizielle, undokumentierte, aber sehr nützliche Anweisung, nämlich die FOR-Schleife. Sie wird oft verwendet, um einen Array zu durchlaufen. Bei der FOR-Anweisung wird der Variablen jeder ganzzahlige Wert zwischen dem unteren und oberen Rand zugewiesen und die Schlaufe jeweils ausgeführt.

Eine traumhafte Konstellation für jeden Studenten wäre der Stundenplan, welche folgende FOR-Schleife produzieren würde:

```
FOR i := 0 TO 6 DO
      WochentagVorlesungsfrei[i] := TRUE;
END;
```

Es wird also zuerst der Code zwischen `DO` und `END` mit i = 0 ausgeführt, dann mit i = 1, danach mit i = 2 und so weiter.

Parameter: Wie kann ich einer Prozedur meine Eingaben übergeben?

Wie wir einige Untertitel zuvor gesehen haben, kann man Prozeduren deklarieren, welche Teile der Arbeit für einen erledigen. Nun ist es aber relativ unnütz, wenn die Prozedur nichts von meiner Eingabe weiss, sondern immer nur das Gleiche ausführt. Ich möchte also einer Prozedur Werte übergeben, und dies kann ich über die Parameter machen.

In den vorherigen Beispielen haben wir immer nur Prozeduren ohne Parameter gesehen. Diese Prozeduren haben einfach eine Klammer auf und dann gerade wieder eine schliessende Klammer:

```
PROCEDURE Abspielen*();
```

Parameter sind Variablen, welche man zwischen die beiden Klammern nach dem Namen der aufzurufenden Prozedur setzt. Zum Beispiel wäre es nützlich, die Prozedur Abspielen weiss, welches Lied ich abspielen möchte:

```
PROCEDURE Abspielen*(Liedtitel : ARRAY OF CHAR);
```

Ich kann jetzt also mit folgendem Aufruf zum Beispiel ein gewünschtes Lied abspielen:

```
Rekorder.Abspielen("Neunundneunzig Luftballons");
```

Es fragt sich dann nur noch, was die Prozedur abspielt, wenn eine andere Kassette eingelegt ist, als die von Nena.

Die Parameter werden also folgendermassen deklariert: zuerst der Variablenname, danach ein Doppelpunkt und am Schluss der Typ. Mehrere Parameter kann man mit einem Strichpunkt voneinander trennen:

```
PROCEDURE Abspielen*(Liedtitel : ARRAY OF CHAR;
Lautstaerke : INTEGER);
```

In der Prozedur selbst kann man auf die Variablen mit dem angegebenen Namen zugreifen. Zum Beispiel:

```
PROCEDURE Abspielen*(Liedtitel : ARRAY OF CHAR;
Lautstaerke : INTEGER);
BEGIN
      IF (Lautstaerke > 9) THEN
            (* der wird bestimmt schwerhoerig! *)
      END;
END Abspielen;
```

Wichtig dabei ist: die Übergabeparameter übergeben lediglich den Wert, aber nicht die Variable. Das heisst, wenn man zum Beispiel oben in der Prozedur `Abspielen` die Variable `Liedtitel` ändert, die aufrufende Prozedur von dem nichts bemerkt:

Aufrufende Prozedur:

```
VAR dasLied : ARRAY 32 OF CHAR;
BEGIN
      dasLied := "Blood on the Dancefloor";
      Abspielen(dasLied, 5);
      (* jetzt ist dasLied immer noch gleich *)
      (* "Blood on the Dancefloor" *)
END;
```

Aufgerufene Prozedur:

```
PROCEDURE Abspielen*(Liedtitel : ARRAY OF CHAR;
Lautstaerke : INTEGER);
BEGIN
    Liedtitel := "You're my hearth, you're my soul";
END Abspielen;
```

VAR Parameter: Gibt es Parameter, welche die ganze Variable übergeben?

Im vorherigen Abschnitt wurde das Phänomen gezeigt, dass eine veränderte Parametervariable sich nicht auf die aufrufende Variable auswirkt. Das heisst, dass diese Parametervariable lokal gilt und demnach zuvor kopiert werden musste (zusätzlicher Aufwand!).

Wenn man aber möchte, dass sich die Veränderung direkt auf die aufrufende Variable auswirkt, dann kann man das explizit mit dem Wort `VAR` tun:

```
PROCEDURE AbspielenNeu(VAR Liedtitel : ARRAY OF
CHAR);
```

In diesem Fall wird beim Aufruf der Prozedur `Abspielen` der erste Parameter direkt übergeben, und Veränderungen in der Prozedur wirken sich auf die aufrufende Variable aus.

Achtung: dieses `VAR` ist nicht gleichbedeutend mit dem der Variablendeklaration. Es hat hier eine komplett andere Bedeutung!

Aufrufende Prozedur:

```
PROCEDURE Aufrufer*();
VAR einLied : ARRAY 32 OF CHAR;
BEGIN
      einLied := "Blood on the Dancefloor";
      AbspielenNeu(einLied);
      (* die Variable einLied ist jetzt verändert *)
      (* worden: You're my hearth, you're my soul *)
END Aufrufer;
```

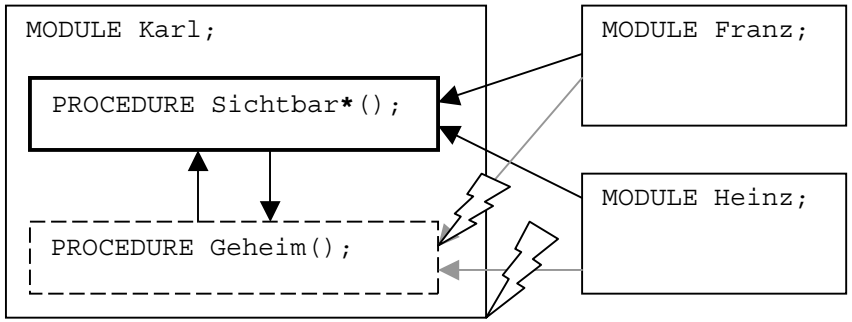
Aufgerufene Prozedur:

```
PROCEDURE AbspielenNeu(VAR Liedtitel : ARRAY OF
CHAR);
BEGIN
    Liedtitel := "You're my hearth, you're my soul";
END AbspielenNeu;
```

Also immer, wenn ich entweder Speicherplatz sparen möchte, oder die aufrufende Variable bewusst verändern möchte, verwende ich VAR Parameter, aber wirklich nur dann. Im Zweifelsfall immer kein `VAR` verwenden!

Stern: Was bedeutet der Stern hinter dem Prozedurnamen?

Bei der Deklaration einer Prozedur wurde bisher immer ein Prozedurname gefolgt von einem Stern geschrieben. Aber was der Stern bedeutet, wurde noch nicht erklärt. Der Stern bedeutet, dass die Prozedur von aussen sichtbar ist. Andere Module und deren Prozeduren dürfen ausschliesslich auf sichtbar Prozeduren zugreifen. Prozeduren ohne Stern sind nur für das eigene Modul und dessen Prozeduren gedacht:



Rückgabewerte: Wie bekomme ich das Resultat einer Prozedur zurück?

Oft will man von einer Prozedur Werte zurück erhalten. Zum Beispiel würde folgende Prozedur, wenn sie denn auch funktionieren würde, nicht viel bringen, weil wir das Resultat nie erfahren würden:

```
PROCEDURE BerechneSinnDesLebens();
```

Viel besser wäre, wenn uns die Funktion zum Beispiel eine Zahl zurückgeben würde, mit dem Resultat, das die Prozedur nach ihrer Berechnung hatte. Also für unser obiges Beispiel:

```
PROCEDURE BerechneSinnDesLebens() : INTEGER;
```

Die Prozedur muss dann ein RETURN enthalten. Zum Beispiel:

```
RETURN(43);
```

Der Prozeduraufruf sieht dann folgendermassen aus:

```
eineZahl := BerechneSinnDesLebens();
```

Nachdem die Prozedur ausgeführt wurde, hat eineZahl den Wert 43.

Rekursion: Kann sich eine Prozedur selbst aufrufen und was bringt das?

Manchmal kann es nützlich sein, dass sich eine Prozedur selbst aufruft. Zum Nehmen wir an, wir haben eine Prozedur, die auf dem Bildschirm Zeichen ausgibt:

```
PROCEDURE DruckeZeichen(String: ARRAY OF CHAR);
BEGIN
      (* drucke die Zeichen *)
      IF fehler_passiert THEN
            DruckeZeichen("Fehler ist passiert!");
      END;
END DruckeZeichen;
```

Hier führt sich die Prozedur DruckeZeichen, wenn ein Fehler passiert nochmals aus, um auf dem Bildschirm anzuzeigen, dass sich ein Fehler ereignet hat.

Dieser Selbstaufruf wird Rekursion genannt. Bei der Rekursion wird die gleiche Funktion erneut aufgerufen. Aber alle Variablen, die in der Prozedur deklariert sind, werden neu angelegt. Das heisst, wenn eine Prozedur die Variable a hat und in dieser der Wert 5 gespeichert ist und sie sich dann selbst aufruft (eben eine Rekursion macht), dass diese neue aufgerufene Prozedur dann eine neue Variable a bekommt, welche **nicht** den Wert 5 hat. Also werden alle Variablen zwischengespeichert. Das folgende Beispiel verwendet Pseudocode: