# Einführung in die Informatik 2

Prof. Dr. H. Seidl, N. Hartmann, R. Vogler
WS 2017/18

**Übungsblatt 7**
Abgabefrist: 10.12.2017

---

**Allgemeine Informationen**

Ausführliche Informationen zu den Übungs- und Hausaufgaben, sowie allgemein zum Übungsbetrieb erhalten Sie auf der Vorlesungswebsite und in Moodle.

Wirken Sie durch ihr Feedback in der wöchentlichen Moodle-Umfrage an der Verbesserung der Veranstaltung mit. Schauen Sie auf Piazza[1] vorbei, wo Sie mit Ihren Kommilitonen diskutieren oder Fragen zu den Aufgaben stellen und beantworten können. Bei Fragen oder Problemen bezüglich Vorlesung, Übungsbetrieb, Hausaufgaben oder Klausur, kommen Sie in unsere Sprechstunde, die wir jeden Freitag, 15-17 Uhr in 02.07.034 anbieten, oder wenden Sie sich direkt an die Übungsleitung (`info2@in.tum.de`).

**Abgabe von OCaml-Hausaufgaben**

Die Vorlage für die OCaml-Hausaufgaben steht auf Moodle bereit. Sie müssen nur die Datei `ha7.ml` bearbeiten. Die Datei `ha7.mli` enthält die Typ-Signaturen, die ihre Implementierung einhalten soll. Kompilieren Sie Ihre Abgabe lokal mit `ocamlbuild ha7.native` um zu überprüfen, dass diese frei von Syntax- und Typfehlern ist und sich an die gegebene Signatur hält. Nicht kompilierende Abgaben geben keine Punkte.

Zur Abgabe melden Sie sich mit Ihrer TUM-Kennung auf `https://vmnipkow3.in.tum.de/ocaml` an und laden Ihre `ha7.ml`-Datei (nicht umbenennen!) hoch. Die Ergebnisse der Tests sind sichtbar sobald die Abgabe abgearbeitet wurde. Die endgültige Bewertung erfolgt aber erst nach der Abgabefrist.

Alle Funktionen müssen selbst definiert werden. Die automatisch geladenen Definitionen aus Pervasives[2] sind verfügbar (außer `open_*`), andere Module nicht. Ebenso sind die imperativen Features von OCaml auf dem Server nicht verfügbar sondern ergeben Syntaxfehler.

**Lernziele**

In den Übungen dieser Wochen lernen Sie:

- Die `fold` und `map` Funktionen zur Implementierung diverser Listenoperationen einzusetzen.
- Die Daten einer Anwendung mit Hilfe von Summentypen zu strukturieren.
- Anonyme Funktionen zur Darstellung von Abbildungen einzusetzen.
- Funktionen zur Laufzeit eines Programms zu erzeugen und zu verändern.

---

[1] `piazza.com/tum.de/ll2017/in0003/home`
[2] `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html`

### Aufgabe 7.1 (P) `unit`, Seiteneffekte, sequentielle Ausführung [Oase]

In OCaml können Funktionen Seiteneffekte beinhalten ohne dass sich dies in der Signatur widerspiegelt.

```
# let succ x = x+1;;
val succ : int -> int = <fun>
# let succ x = print_int x; x+1;;
val succ : int -> int = <fun>
```

*Bemerkung:* Die `;;` sind nur im Toplevel notwendig um die Eingabe abzuschließen. In Ihren .ml-Dateien brauchen Sie diese nicht einfügen.

Für Bibliotheksfunktionen gilt die Konvention, dass diese den Typ `unit` liefern, wenn sie auf Seiteneffekten basieren (z.B. `print_int : int -> unit`). Das Ziel dieser Funktionen ist der Seiteneffekt - `unit` hat nur einen Wert, und enthält daher keine nützliche Information.

```
type unit = ()
```

Der Wert soll ein leeres Tupel darstellen, daher die besondere Syntax. Allerdings hätte man sich genauso auf einen anderen Typ einigen können (z.B. `type side = Side`).

Da das Ergebnis immer `()` ist, kann man mehrere solcher Funktion z.B. wie folgt hintereinander ausführen:

```
let a,b = "1","2" in
let () = print_endline a in (* exhaustive pattern *)
let _ = print_endline b in (* _ matches everything  *)
a
```

In OCaml gibt es `;` um diese Sequenzen etwas kompakter schreiben zu können:

```
let a,b = "1","2" in
print_endline a;
print_endline b;
a
```

Den Typ von `;` kann man sich als `unit -> 'a -> 'a` vorstellen. Allerdings ist dies ein spezieller Operator, den wir nicht selbst definieren können, da Applikation zwar linksassoziativ ist, Argumente aber von rechts nach links ausgewertet werden.

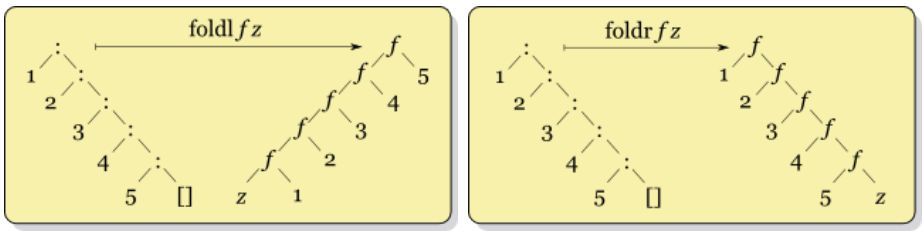
### Aufgabe 7.2 (P) Funktionen höherer Ordnung auf Listen [Steppe]

Listen sind in OCaml, bis auf den syntaktischen Zucker (`[1;2] = 1::2::[]`), auch nur ein normaler Summentyp:

```
type 'a list = [] | (::) of 'a * 'a list
```

Das Gegenstück zu Schleifen in imperativen Sprachen, ist die Rekursion in funktionalen Sprachen. Nachdem Sie einige rekursive Funktionen auf Listen definiert haben, stellen Sie vielleicht fest, dass diese ein Schema einhalten, welches wir abstrahieren können. In OCaml gibt es für Listen z.B. `fold_left` und `fold_right` welche eine Liste von links bzw. rechts falten. Beide erwarten eine Funktion $f$, welche jeweils ein Element der Liste und den Akkumulator erhält und einen neuen Wert dafür liefert, und den Initialwert $z$. Wir können den Unterschied bzgl. der Anwendung von $f$ wie folgt visualisieren[3] (Haskell, in OCaml `::` statt `:`):

[3] `https://en.wikipedia.org/wiki/Fold_(higher-order_function)`




In OCaml erwartet `fold_right` die Liste vor $z$:

```
List.fold_left  : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

*Bemerkung:* Typvariablen werden alphabetisch vergeben; beide Funktionen liefern den Typ des Akkumulators.

Geben Sie Implementierungen für

1. `fold_left` wie oben beschrieben.
2. `fold_right` wie oben beschrieben.
3. `sum : int list -> int` definiert durch `fold_left` oder `fold_right`.
4. `map : ('a -> 'b) -> 'a list -> 'b list` definiert durch `fold_left` oder `fold_right`.
5. `map_rev : ('a -> 'b) -> 'a list -> 'b list` definiert durch `fold_left` oder `fold_right`.

Bestimmen Sie für Ihre Implementierungen zudem ob diese endrekursiv sind bzw. ob diese nur endrekursive Funktionen aufrufen und stellen Sie, falls möglich, endrekursive Versionen her.

### Aufgabe 7.3 (P) Bäume falten [Steppe]

Gegeben sei folgende Typ-Definition eines binären Baumes:

```
type 'a tree = Node of 'a * 'a tree * 'a tree | Leaf
```

1. Implementieren Sie die Funktion

   ```
   val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b tree -> 'a
   ```

   die alle Werte per in-order Traversal[4] faltet.
2. Implementieren Sie die Funktion

   ```
   val sum : int tree -> int
   ```

   welche die Summe aller Werte des Baumes berechnet mithilfe von `fold_left`.
3. Implementieren Sie die Funktion

   ```
   val to_list : 'a tree -> 'a list
   ```

   die eine Liste mit allen Werten des Baums erstellt mithilfe von `fold_left`. Die Elemente der Liste sollen dabei dem in-order Traversal des Baums entsprechen.

[4] https://en.wikipedia.org/wiki/Tree_traversal

**Aufgabe 7.4 (P) Ausdrücke auswerten** [Steppe]

Um arithmetische Ausdrücke über `int`-Konstanten und -Variablen darzustellen, definieren wir die folgenden Typen:

```
type const = int
type var = string
type unary_op = Neg
type binary_op = Add | Sub | Mul | Div
type expr = Const of const
          | Var of var
          | Unary of unary_op * expr
          | Binary of expr * binary_op * expr
```

Weiterhin definieren wir einen Zustand (State)

```
type state = var -> const
```

welcher Variablen auf deren Werte abbildet.

Bearbeiten Sie diese Aufgaben:

1. Stellen Sie die folgenden Ausdrücke mit Hilfe der gegebenen Typen dar:
   (a) `3 + x`
   (b) `y * -x + 5`
   (c) `(x - 7) * y / 2`
2. Implementieren Sie die Funktion `eval_expr : state -> expr -> const`, die den Wert eines arithmetischen Ausdrucks (2. Argument) berechnet, wobei die Werte der Variablen durch den übergebenen Zustand (1. Argument) gegeben sind.
3. Nutzen Sie Ihre Funktion `eval_expr` um die in 1. definierten Ausdrücke auszuwerten. Verwenden Sie dazu die Variablenbelegung $\{x \mapsto 5, y \mapsto -1\}$.

**Aufgabe 7.5 (H) Meine eigene Programmiersprache** [6 Punkte, Steppe]

Mit Hilfe der in Aufgabe 7.4 definierten Typen, werden nun Bedingungen (Conditions) definiert:

```
type comp_op = Eq | Neq | Le | Leq
type cond = True
          | False
          | Comp of expr * comp_op * expr
          | Not of cond
          | And of cond * cond
          | Or of cond * cond
```

Eine Bedingung kann dabei wahr (`True`) oder falsch (`False`), ein Vergleich auf $=, \neq, <$ oder $\leq$ von zwei arithmetischen Ausdrücken, die Negation (`Not`) einer Bedingung oder eine Konjunktion (`And`) bzw. Disjunktion (`Or`) von zwei Bedingungen sein.

Damit definieren wir nun eine imperative Programmiersprache mit den folgenden Anweisungen (Statements):

```
type stmt = Assign of var * expr
          | Print of var
          | IfThenElse cond * stmt list * stmt list
          | While cond * stmt list
```

Hierbei weist eine `Assign`-Anweisung den Wert des Ausdrucks der Variablen zu, `Print` gibt den Wert der Variablen auf der Konsole aus und bei einer `IfThenElse`-Verzweigung werden die Anweisungen der ersten (bzw. zweiten) Liste ausgeführt, wenn die Bedingung zu *true* (bzw. *false*) auswertet. Die `While`-Schleife wiederholt die Anweisungen solange die Bedingung erfüllt ist. Gehen Sie davon aus, dass alle Variablennamen eindeutig und im ganzen Programm sichtbar/gültig sind, es gibt also kein Shadowing[5].

Ein Programm dieser Sprache lässt sich nun als

```
type prog = stmt list
```

definieren.

Bearbeiten Sie die folgenden Teilaufgaben:

1. Stellen Sie das folgende Programm mittels der gegebenen Typen dar:

```
x = 4;
if(arg < 0 || z != 0) {
  y = -arg;
  x = x * 3;
} else {
  y = 3 * arg;
}
z = x * y + 10;
print(z);
ret = z;
```

   Binden Sie den Wert an den Namen `prog1` (siehe *Ha7.ml*).

[5] `https://en.wikipedia.org/wiki/Variable_shadowing`

2. Implementieren Sie die Funktion `eval_cond : state -> cond -> bool`, die den Wert einer Bedingung (2. Argument) unter der gegebenen Variablenbelegung (1. Argument) berechnet.
3. Implementieren Sie die Funktion `eval_stmt : state -> stmt -> state`, die eine Anweisung (2. Argument) auf einem gegebenen Programmzustand (1. Argument) ausführt und den resultierenden Zustand des Programms zurückgibt.
4. Implementieren Sie die Funktion `eval_prog : const -> prog -> const`, welche ein Programm (2. Argument) ausführt. Dazu wird zuerst die Variable `arg` auf den übergebenen Wert (1. Argument) initialisiert. Alle übrigen Variablen werden mit 0 vorinitialisiert. Nachdem die Funktion das Programm ausgeführt hat, liefert sie den Wert zurück, der am Ende des Programms in der Variable `ret` steht.

*Hinweis: Um Ihre Implementierungen zu testen stehen einige Hilfsfunktionen zur Verfügung.*

**Aufgabe 7.6 (H) Jetzt wird's noch besser!** [14 Punkte, Steppe]

Der Optimierer ist der größte Teil moderner Compiler. Die Aufgabe des Optimierers ist es, das vom Programmierer geschriebene Programm in ein anderes Programm zu transformieren, das zwar das gleiche Ergebnis berechnet, aber schneller ausgeführt werden kann. Dabei gibt es eine Vielzahl sehr unterschiedlicher Optimierungen. Vereinfachte Versionen von zwei der wichtigsten Optimierungen werden Sie im Folgenden für die in Aufgabe 7.5 definierte Sprache implementieren:

1. Partial Evaluation[6]: Implementieren Sie die Funktion `pe_expr_opt : expr -> expr`, die einen arithmetischen Ausdruck partiell evaluiert, das heißt, Sie müssen alle Berechnungen, die unabhängig von den Variablenwerten durchgeführt werden können, durchführen. Das sind:
   - Teilausdrücke, die nur aus Konstanten bestehen, müssen so weit wie möglich berechnet werden, z.B. $3 * (5 - 2)$ muss zu 9 vereinfacht werden.
   - Binäre Operationen mit neutralem Element müssen vereinfacht werden; z.B. müssen $a + 0$, $a - 0$, $a * 1$ und $a/1$ zu $a$ vereinfacht werden (ebenso wie die symmetrischen Gegenstücke).
   - Binäre Operationen mit absorbierendem Element müssen vereinfacht werden; z.B. müssen $a * 0$ und $0/a$ zu 0 vereinfacht werden.
   - Wird ein Ausdruck von sich selbst subtrahiert oder durch sich selbst dividiert, muss dies vereinfacht werden, z.B. $(a+2)/(a+2)$ muss zu 1 vereinfacht werden. Wir definieren hier, dass dies auch für Division durch 0 gilt, also $0/0 = 1$.

   Es ist möglich, dass nach einer Vereinfachung weitere Umforumungen möglich werden, z.B. kann $x*(t-t)$ erst zu $x*0$ und dann zu 0 umgeformt werden. Ihre Funktion muss diese Umformungen deshalb solange durchführen, bis keine Vereinfachung mehr möglich ist. Implementieren Sie anschließend die Funktion `pe_cond_opt : cond -> cond`, die eine Bedingung soweit vereinfacht wie nur möglich. Dazu müssen:
   - Alle enthaltenen arithmetischen Ausdrücke mit `pe_expr_opt` partielle evaluiert werden und sofern beide vereinfachten Ausdrücke nur noch Konstanten sind, muss die Bedingung direkt zu `True` oder `False` ausgewertet werden.
   - `Not (True)` und `Not (False)` zu `False` bzw. `True` vereinfacht werden.
   - `And` wird zu `False` und `Or` zu `True` vereinfacht, wenn eine der beiden Bedingungen `False` bzw. `True` ist.

[6] `https://en.wikipedia.org/wiki/Partial_evaluation`

- Konjunktion oder Disjunktion einer Bedingung `c` mit neutralem Element muss direkt zu `c` vereinfacht werden.

Auch `pe_cond_opt` muss solange vereinfachen, bis keine der genannten Vereinfachungen mehr möglich ist. Schreiben Sie dann die Funktion `pe_opt : prog -> prog`, die alle Ausdrücke und Bedingungen eines Programms mit Hilfe von `pe_expr_opt` bzw. `pe_cond_opt` partiell evaluiert.

2. Dead Code Elimination[7]: Implementieren Sie die Funktion `dce_opt : prog -> prog`, die toten - das heißt unbenutzten - Code[8] entfernt. Führen Sie dazu folgende Transformationen durch:
   - Kommt eine Variable auf der linken Seite einer Zuweisung vor und von dieser Variablen wird aber nirgendwo im gesamten Programm gelesen, so muss die Zuweisung entfernt werden. Zuweisungen zur Variablen `ret` dürfen niemals entfernt werden.
   - Ist die Bedingung einer `IfThenElse`-Anweisung `True` (oder. `False`), so kann direkt der *then* (bzw. *else*) Zweig ausgeführt werden. Der andere Zweig, sowie die `IfThenElse` Anweisung müssen dann entfernt werden.
   - Ist die Bedingung einer `While`-Schleife `False`, so muss diese vollständig entfernt werden.
3. Implementieren Sie die Funktion `opt : prog -> prog`, die das gegebene Programm vollständig optimiert. Dazu müssen die beiden obigen Optimierungen solange (abwechselnd) durchgeführt werden, bis keine Optimierung mehr möglich ist.

*Hinweis: Um Ihre Implementierungen zu testen stehen einige Hilfsfunktionen zur Verfügung.*

[7] `https://en.wikipedia.org/wiki/Dead_code_elimination`
[8] `https://de.wikipedia.org/wiki/Toter_Code`