

Helmut Seidl

Informatik 2

TU München

Sommersemester 2007

# Inhaltsverzeichnis

<b>0</b>	<b>Allgemeines</b>	<b>4</b>
<b>1</b>	<b>Korrektheit von Programmen</b>	<b>5</b>
1.1	Verifikation von Programmen . . . . .	7
1.2	Korrektheit . . . . .	14
1.3	Optimierung . . . . .	18
1.4	Terminierung . . . . .	21
1.5	Modulare Verification und Prozeduren . . . . .	26
1.6	Prozeduren mit lokalen Variablen . . . . .	34
<b>2</b>	<b>Grundlagen</b>	<b>36</b>
2.1	Die Interpreter-Umgebung . . . . .	36
2.2	Ausdrücke . . . . .	37
2.3	Wert-Definitionen . . . . .	38
2.4	Komplexere Datenstrukturen . . . . .	39
2.5	Listen . . . . .	41
2.6	Definitionen von Funktionen . . . . .	42
2.7	Benutzerdefinierte Typen . . . . .	45
<b>3</b>	<b>Funktionen – näher betrachtet</b>	<b>49</b>
3.1	Rekursionsarten . . . . .	50
3.2	Funktionen höherer Ordnung . . . . .	52
3.3	Funktionen als Daten . . . . .	54
3.4	Einige Listen-Funktionen . . . . .	54
3.5	Polymorphe Funktionen . . . . .	55
3.6	Polymorphe Datentypen . . . . .	57
3.7	Anwendung: Queues . . . . .	58
3.8	Namenlose Funktionen . . . . .	60
<b>4</b>	<b>Praktische Features in Ocaml</b>	<b>61</b>
4.1	Ausnahmen (Exceptions) . . . . .	61
4.2	Imperative Features im Ocaml . . . . .	64
4.3	Textuelle Ein- und Ausgabe . . . . .	68
<b>5</b>	<b>Anwendung:</b>	
	<b>Grundlegende Graph-Algorithmen</b>	<b>69</b>
5.1	Gerichtete Graphen . . . . .	70
5.2	Erreichbarkeit und DFS . . . . .	70
5.3	Topologisches Sortieren . . . . .	72

<b>6</b>	<b>Formale Methoden für Ocaml</b>	<b>76</b>
6.1	MiniOcaml . . . . .	76
6.2	Eine Semantik für MiniOcaml . . . . .	78
6.3	Beweise für MiniOcaml-Programme . . . . .	83
<b>7</b>	<b>Das Modulsystem von OCAML</b>	<b>90</b>
7.1	Module oder Strukturen . . . . .	91
7.2	Modul-Typen oder Signaturen . . . . .	94
7.3	Information Hiding . . . . .	96
7.4	Funktoren . . . . .	97
7.5	Getrennte Übersetzung . . . . .	99

# 0 Allgemeines

## Inhalt dieser Vorlesung:

- Korrektheit von Programmen;
- Funktionales Programmieren mit OCaml :-)

# 1 Korrektheit von Programmen

- Programmierer machen Fehler :-)
- Programmierfehler können **teuer** sein, z.B. wenn eine Rakete explodiert, ein firmenwichtiges System für **Stunden** ausfällt ...
- In einigen Systemen dürfen **keine** Fehler vorkommen, z.B. Steuerungssoftware für Flugzeuge, Signalanlagen für Züge, Airbags in Autos ...

## Problem:

Wie können wir sicherstellen, dass ein Programm das **richtige** tut?

## Ansätze:

- Sorgfältiges Vorgehen bei der Software-Entwicklung;
- Systematisches Testen
  - ⇒ formales Vorgehensmodell (**Software Engineering**)
- Beweis der Korrektheit
  - ⇒ **Verifikation**

**Hilfsmittel:** Zusicherungen

## Beispiel:

```
public class GGT extends MiniJava {
    public static void main (String[] args) {
        int x, y, a, b;
        a = read(); b = read();
        x = a; y = b;
        while (x != y)
            if (x > y) x = x - y;
            else      y = y - x;

        assert(x != y);

        write(x);
    } // Ende der Definition von main();
} // Ende der Definition der Klasse GGT;
```

## Kommentare:

- Die statische Methode `assert()` erwartet ein Boolesches Argument.
- Bei normaler Programm-Ausführung wird jeder Aufruf `assert(e)` ignoriert :-)
- Starten wir **Java** mit der Option: `-ea` (**enable assertions**), werden die `assert`-Aufrufe ausgewertet:
  - ⇒ Liefert ein Argument-Ausdruck `true`, fährt die Programm-Ausführung fort.
  - ⇒ Liefert ein Argument-Ausdruck `false`, wird ein **Fehler** `AssertionError` geworfen.

## Achtung:

Der Laufzeit-Test soll eine **Eigenschaft** des Programm-Zustands bei Erreichen eines Programm-Punkts überprüfen.

Der Test sollte **keineswegs** den Programm-Zustand verändern !!!

Sonst zeigt das beobachtete System ein anderes Verhalten als das unbeobachtete ???

## Tipp:

Um Eigenschaften komplizierterer Datenstrukturen zu überprüfen, empfiehlt es sich, getrennt **Inspector**-Klassen anzulegen, deren Objekte eine Datenstruktur **störungsfrei** besichtigen können :-)

## Problem:

- Es gibt i.a. sehr viele Programm-Ausführungen :-)
- Einhalten der Zusicherungen kann das **Java**-Laufzeit-System immer nur für eine Programm-Ausführung überprüfen :-)



Wir benötigen eine generelle Methode, um das Einhalten einer Zusicherung zu **garantieren** ...

## 1.1 Verifikation von Programmen



Robert W Floyd, Stanford U. (1936 – 2001)

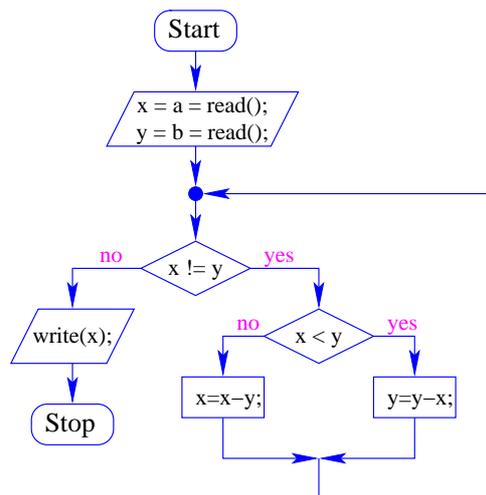
### Vereinfachung:

Wir betrachten erst mal nur **MiniJava** ;-)

### Idee:

- Wir schreiben eine Zusicherung an **jeden** Programmpunkt :-)
- Wir überprüfen **lokal**, dass die Zusicherungen von den einzelnen Anweisungen im Programm eingehalten werden.

### Unser Beispiel:



## Diskussion:

- Die Programmpunkte entsprechen den **Kanten** im Kontrollfluss-Diagramm :-)
- Wir benötigen eine Zusicherung pro Kante ...

## Hintergrund:

$d \mid x$  gilt genau dann wenn  $x = d \cdot z$  für eine ganze Zahl  $z$ .

Für ganze Zahlen  $x, y$  sei  $ggT(x, y) = 0$ , falls  $x = y = 0$  und andernfalls die größte ganze Zahl  $d$ , die  $x$  und  $y$  teilt.

Dann gelten unter anderem die folgenden Gesetze:

$$\begin{aligned} ggT(x, 0) &= |x| \\ ggT(x, x) &= |x| \\ ggT(x, y) &= ggT(x, y - x) \\ ggT(x, y) &= ggT(x - y, y) \end{aligned}$$

## Idee für das Beispiel:

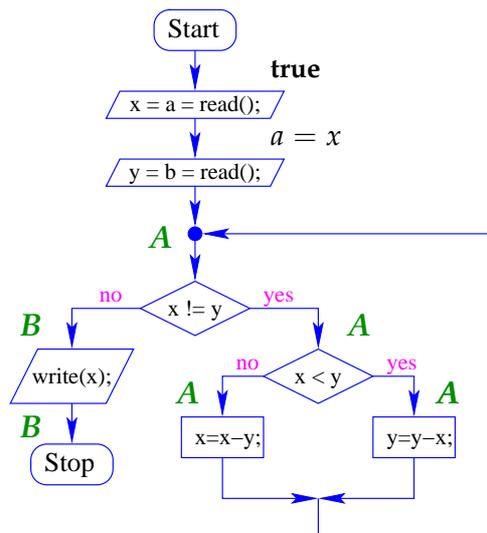
- Am Anfang gilt nix :-)
- Nach `a=read(); x=a;` gilt  $a = x$  :-)
- Vor Betreten und während der Schleife soll gelten:

$$A \equiv ggT(a, b) = ggT(x, y)$$

- Am Programm-Ende soll gelten:

$$B \equiv A \wedge x = y$$

## Unser Beispiel:



## Frage:

Wie überprüfen wir, dass Zusicherungen lokal zusammen passen?

## Teilproblem 1: Zuweisungen

Betrachte z.B. die Zuweisung:  $x = y+z;$

Damit **nach** der Zuweisung gilt:  $x > 0$ , // **Nachbedingung**

muss **vor** der Zuweisung gelten:  $y + z > 0$ . // **Vorbedingung**

## Allgemeines Prinzip:

- Jede Anweisung transformiert eine Nachbedingung  $B$  in eine **minimale** Anforderung, die **vor** Ausführung erfüllt sein muss, damit  $B$  **nach** der Ausführung gilt :-)
- Im Falle einer Zuweisung  $x = e;$  ist diese **schwächste Vorbedingung** (engl.: **weakest precondition**) gegeben durch

$$\mathbf{WP}[x = e;] (B) \equiv B[e/x]$$

Das heißt: wir **substituieren** einfach in  $B$  überall  $x$  durch  $e$  !!!

- Eine beliebige Vorbedingung  $A$  für eine Anweisung  $s$  ist **gültig**, sofern

$$A \Rightarrow \mathbf{WP}[s] (B)$$

//  $A$  **impliziert** die schwächste Vorbedingung für  $B$ .

## Beispiel:

Zuweisung:  $x = x-y;$   
Nachbedingung:  $x > 0$   
schwächste Vorbedingung:  $x - y > 0$   
stärkere Vorbedingung:  $x - y > 2$   
noch stärkere Vorbedingung:  $x - y = 3$  :-)

## ... im GGT-Programm (1):

Zuweisung:  $x = x-y;$   
Nachbedingung:  $A$   
schwächste Vorbedingung:

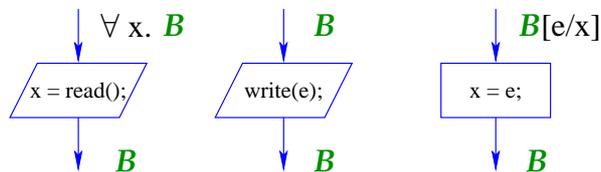
$$\begin{aligned} A[x - y/x] &\equiv \text{ggT}(a, b) = \text{ggT}(x - y, y) \\ &\equiv \text{ggT}(a, b) = \text{ggT}(x, y) \\ &\equiv A \end{aligned}$$

... im GGT-Programm (2):

Zuweisung:  $y = y - x;$   
 Nachbedingung:  $A$   
 schwächste Vorbedingung:

$$\begin{aligned} A[y - x/y] &\equiv \text{ggT}(a, b) = \text{ggT}(x, y - x) \\ &\equiv \text{ggT}(a, b) = \text{ggT}(x, y) \\ &\equiv A \end{aligned}$$

Zusammenstellung:



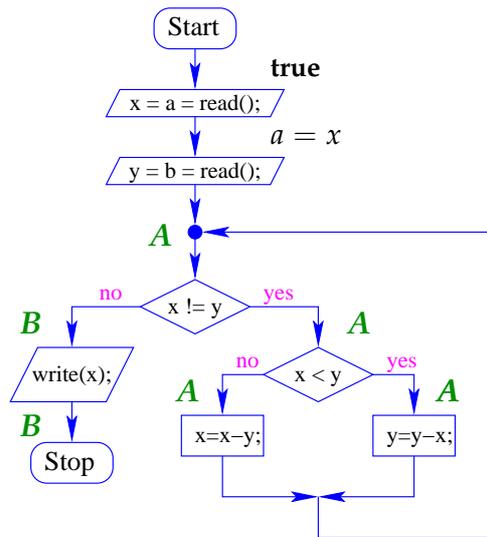
$$\begin{aligned} \text{WP}[\text{;}](B) &\equiv B \\ \text{WP}[x = e;](B) &\equiv B[e/x] \\ \text{WP}[x = \text{read}();](B) &\equiv \forall x. B \\ \text{WP}[\text{write}(e);](B) &\equiv B \end{aligned}$$

Diskussion:

- Die Zusammenstellung liefert für alle Aktionen jeweils die **schwächsten** Vorbedingungen für eine Nachbedingung  $B$ .
- Eine Ausgabe-Anweisung ändert keine Variablen. Deshalb ist da die schwächste Vorbedingung  $B$  selbst ;-)
- Eine Eingabe-Anweisung  $x = \text{read}();$  ändert die Variable  $x$  auf unvorhersehbare Weise.

Damit nach der Eingabe  $B$  gelten kann, muss  $B$  **vor** der Eingabe für jedes mögliche  $x$  gelten ;-)

Orientierung:

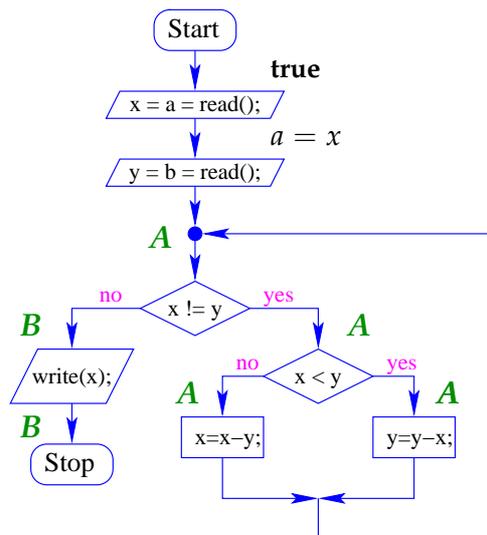


Für die Anweisungen:  $b = \text{read}(); y = b;$  berechnen wir:

$$\begin{aligned} \text{WP}[y = b;] (A) &\equiv A[b/y] \\ &\equiv \text{ggT}(a, b) = \text{ggT}(x, b) \end{aligned}$$

$$\begin{aligned} \text{WP}[b = \text{read}();] (\text{ggT}(a, b) = \text{ggT}(x, b)) \\ &\equiv \forall b. \text{ggT}(a, b) = \text{ggT}(x, b) \\ &\Leftarrow a = x \quad \text{: -) } \end{aligned}$$

Orientierung:

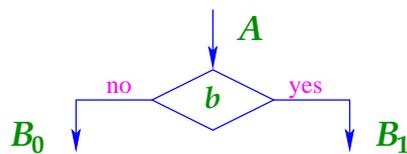


Für die Anweisungen:  $a = \text{read}(); x = a;$  berechnen wir:

$$\begin{aligned} \text{WP}[[x = a;] (a = x) &\equiv a = a \\ &\equiv \mathbf{true} \end{aligned}$$

$$\begin{aligned} \text{WP}[[a = \text{read}();] (\mathbf{true}) &\equiv \forall a. \mathbf{true} \\ &\equiv \mathbf{true} \quad \text{:)} \end{aligned}$$

## Teilproblem 2: Verzweigungen



Es sollte gelten:

- $A \wedge \neg b \Rightarrow B_0$  und
- $A \wedge b \Rightarrow B_1$ .

Das ist der Fall, falls  $A$  die **schwächste Vorbedingung** der Verzweigung:

$$\text{WP}[[b] (B_0, B_1) \equiv ((\neg b) \Rightarrow B_0) \wedge (b \Rightarrow B_1)$$

impliziert :-)

Die schwächste Vorbedingung können wir umschreiben in:

$$\begin{aligned} \text{WP}[[b] (B_0, B_1) &\equiv (b \vee B_0) \wedge (\neg b \vee B_1) \\ &\equiv (\neg b \wedge B_0) \vee (b \wedge B_1) \vee (B_0 \wedge B_1) \\ &\equiv (\neg b \wedge B_0) \vee (b \wedge B_1) \end{aligned}$$

### Beispiel:

$$B_0 \equiv x > y \wedge y > 0 \qquad B_1 \equiv x > 0 \wedge y > x$$

Sei  $b$  die Bedingung  $y > x$ .

Dann ist die schwächste Vorbedingung:

$$\begin{aligned} &(x > y \wedge y > 0) \vee (x > 0 \wedge y > x) \\ &\equiv x > 0 \wedge y > 0 \wedge x \neq y \end{aligned}$$

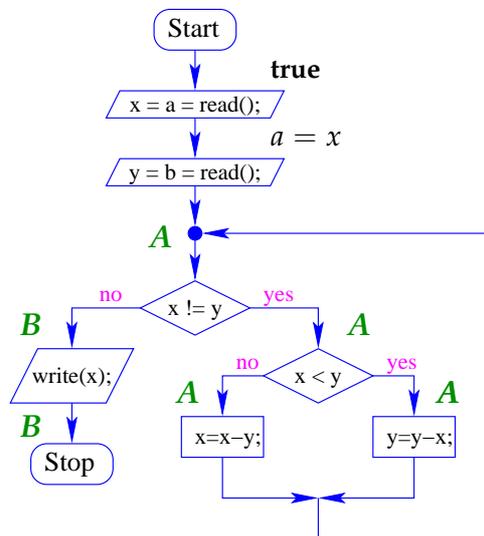
... im GGT-Beispiel:

$$\begin{aligned}
 b &\equiv y > x \\
 \neg b \wedge A &\equiv x \geq y \wedge \text{ggT}(a, b) = \text{ggT}(x, y) \\
 b \wedge A &\equiv y > x \wedge \text{ggT}(a, b) = \text{ggT}(x, y)
 \end{aligned}$$

⇒ Die schwächste Vorbedingung ist:

$$\begin{aligned}
 \text{ggT}(a, b) &= \text{ggT}(x, y) \\
 \dots \text{ also genau } &A \text{ :-)}
 \end{aligned}$$

Orientierung:



Analog argumentieren wir für die Zusicherung vor der Schleife:

$$\begin{aligned}
 b &\equiv y \neq x \\
 \neg b \wedge B &\equiv B \\
 b \wedge A &\equiv A \wedge x \neq y
 \end{aligned}$$

⇒  $A \equiv (A \wedge x = y) \vee (A \wedge x \neq y)$  ist die schwächste Vorbedingung für die Verzweigung :-)

## Zusammenfassung der Methode:

- Annotiere jeden Programmpunkt mit einer Zusicherung.
- Überprüfe für jede Anweisung  $s$  zwischen zwei Zusicherungen  $A$  und  $B$ , dass  $A$  die schwächste Vorbedingung von  $s$  für  $B$  impliziert, d.h.:

$$A \Rightarrow \mathbf{WP}[[s]](B)$$

- Überprüfe entsprechend für jede Verzweigung mit Bedingung  $b$ , ob die Zusicherung  $A$  vor der Verzweigung die schwächste Vorbedingung für die Nachbedingungen  $B_0$  und  $B_1$  der Verzweigung impliziert, d.h.

$$A \Rightarrow \mathbf{WP}[[b]](B_0, B_1)$$

Solche Annotierungen nennen wir **lokal konsistent**.

## 1.2 Korrektheit

### Fragen:

- Welche Programm-Eigenschaften können wir mithilfe lokal konsistenter Annotierungen garantieren ?
- Wie können wir nachweisen, dass unser Verfahren **keine falschen Ergebnisse** liefert ??

### Erinnerung (1):

- In **MiniJava** können wir ein Zustand  $\sigma$  aus einer **Variablen-Belegung**, d.h. einer Abbildung der Programm-Variablen auf ganze Zahlen (ihren Werten), z.B.:

$$\sigma = \{x \mapsto 5, y \mapsto -42\}$$

- Ein Zustand  $\sigma$  **erfüllt** eine Zusicherung  $A$ , falls

$$A[\sigma(x)/x]_{x \in A}$$

// wir substituieren jede Variable in  $A$  durch ihren Wert in  $\sigma$   
eine **wahre** Aussage ist, d.h. äquivalent **true**.

**Wir schreiben:**  $\sigma \models A$ .

### Beispiel:

$$\begin{aligned}
\sigma &= \{x \mapsto 5, y \mapsto 2\} \\
A &\equiv (x > y) \\
A[5/x, 2/y] &\equiv (5 > 2) \\
&\equiv \mathbf{true}
\end{aligned}$$

$$\begin{aligned}
\sigma &= \{x \mapsto 5, y \mapsto 12\} \\
A &\equiv (x > y) \\
A[5/x, 12/y] &\equiv (5 > 12) \\
&\equiv \mathbf{false}
\end{aligned}$$

### Triviale Eigenschaften:

$$\begin{aligned}
\sigma &\models \mathbf{true} && \text{für jedes } \sigma \\
\sigma &\models \mathbf{false} && \text{für kein } \sigma \\
\sigma &\models A_1 \text{ und } \sigma \models A_2 && \text{ist äquivalent zu} \\
&&& \sigma \models A_1 \wedge A_2 \\
\sigma &\models A_1 \text{ oder } \sigma \models A_2 && \text{ist äquivalent zu} \\
&&& \sigma \models A_1 \vee A_2
\end{aligned}$$

### Erinnerung (2):

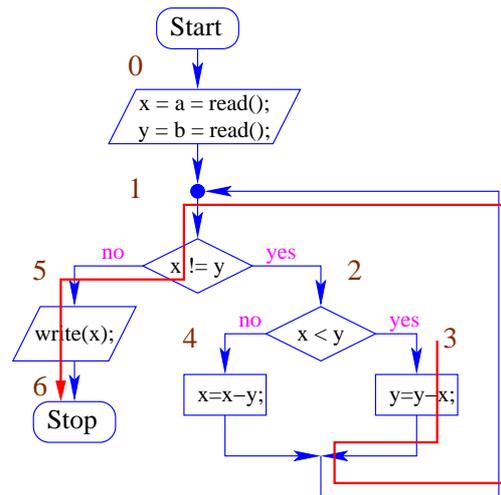
- Eine Programmausführung  $\pi$  durchläuft einen **Pfad** im Kontrollfluss-Graphen :-)
- Sie beginnt in einem Programmpunkt  $u_0$  in einem Anfangszustand  $\sigma_0$ . und führt in einen Programmpunkt  $u_m$  und einen Endzustand  $\sigma_m$ .
- Jeder Schritt der Programm-Ausführung führt eine Aktion aus und ändert Programmpunkt und Zustand :-)

$\implies$  Wir können  $\pi$  als Folge darstellen:

$$(u_0, \sigma_0) s_1 (u_1, \sigma_1) \dots s_m (u_m, \sigma_m)$$

wobei die  $s_i$  Elemente des Kontrollfluss-Graphen sind, d.h. Anweisungen oder Bedingungen ...

### Beispiel:



Nehmen wir an, wir starten in Punkt 3 mit  $\{x \mapsto 6, y \mapsto 12\}$ .

Dann ergibt sich die **Programmausführung**:

$$\begin{aligned} \pi &= (3, \{x \mapsto 6, y \mapsto 12\}) \quad y = y - x; \\ &\quad (1, \{x \mapsto 6, y \mapsto 6\}) \quad !(x \neq y) \\ &\quad (5, \{x \mapsto 6, y \mapsto 6\}) \quad \text{write}(x); \\ &\quad (6, \{x \mapsto 6, y \mapsto 6\}) \end{aligned}$$

**Satz:**

Sei  $p$  ein MiniJava-Programm, Sei  $\pi$  eine Programmausführung, die im Programmpunkt  $u$  startet und zum Programmpunkt  $v$  führt.

**Annahmen:**

- Die Programmpunkte von  $p$  seien lokal konsistent mit Zusicherungen annotiert.
- Der Programmpunkt  $u$  sei mit  $A$  annotiert.
- Der Programmpunkt  $v$  sei mit  $B$  annotiert.

**Dann gilt:**

Erfüllt der Anfangszustand von  $\pi$  die Zusicherung  $A$ , dann erfüllt der Endzustand die Zusicherung  $B$ .

**Bemerkungen:**

- Ist der Startpunkt des Programms mit **true** annotiert, dann erfüllt jede Programmausführung, die den Programmpunkt  $v$  erreicht, die Zusicherung an  $v$  :-)

- Zum Nachweis, dass eine Zusicherung  $A$  an einem Programmpunkt  $v$  gilt, benötigen wir eine lokal konsistente Annotierung mit zwei Eigenschaften:
  - (1) der Startpunkt ist mit **true** annotiert;
  - (2) Die Zusicherung an  $v$  impliziert  $A$  :-)
- Unser Verfahren gibt (vorerst) keine Garantie, dass  $v$  überhaupt erreicht wird !!!
- Falls ein Programmpunkt  $v$  mit der Zusicherung **false** annotiert werden kann, kann  $v$  nie erreicht werden :-))

### Beweis:

Sei  $\pi = (u_0, \sigma_0) s_1(u_1, \sigma_1) \dots s_m(u_m, \sigma_m)$

Gelte:  $\sigma_0 \models A$ .

Wir müssen zeigen:  $\sigma_m \models B$ .

### Idee:

Induktion nach der Länge  $m$  der Programmausführung :-)

### Fazit:

- Das Verfahren nach Floyd ermöglicht uns zu beweisen, dass eine Zusicherung  $B$  bei Erreichen eines Programmpunkts stets (bzw. unter geeigneten Zusatzannahmen :-)) gilt ...
- Zur Durchführung benötigen wir:
  - Zusicherung **true** am Startpunkt.
  - Zusicherungen an jedem weiteren Programmpunkt :-)
  - Nachweis, dass die Zusicherungen lokal konsistent sind  
 $\implies$  Logik, automatisches Beweisen

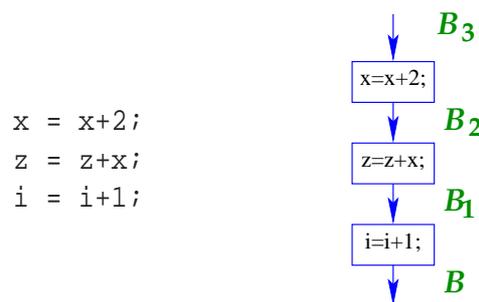
## 1.3 Optimierung

**Ziel:** Verringerung der benötigten Zusicherungen

**Beobachtung:**

Hat das Programm **keine Schleifen**, können wir für jeden Programmpunkt eine hinreichende Vorbedingung **ausrechnen !!!**

**Beispiel:**



**Beispiel (Fort.):**

Sei  $B \equiv z = i^2 \wedge x = 2i - 1$

Dann rechnen wir:

$$\begin{aligned}
 B_1 &\equiv \mathbf{WP}[[i = i+1;]](B) &\equiv z = (i+1)^2 \wedge x = 2(i+1) - 1 \\
 & &\equiv z = (i+1)^2 \wedge x = 2i + 1 \\
 B_2 &\equiv \mathbf{WP}[[z = z+x;]](B_1) &\equiv z + x = (i+1)^2 \wedge x = 2i + 1 \\
 & &\equiv z = i^2 \wedge x = 2i + 1 \\
 B_3 &\equiv \mathbf{WP}[[x = x+2;]](B_2) &\equiv z = i^2 \wedge x + 2 = 2i + 1 \\
 & &\equiv z = i^2 \wedge x = 2i - 1 \\
 & &\equiv B \quad ;-)
 \end{aligned}$$

**Idee:**

- Für jede Schleife wähle **einen** Programmpunkt aus.

**Sinnvolle Auswahlen:**

- Vor der Bedingung;
- Am Beginn des Rumpfs;
- Am Ende des Rumpfs ...

- Stelle für jeden gewählten Punkt eine Zusicherung bereit  
⇒ **Schleifen-Invariante**
- Für alle übrigen Programmpunkte bestimmen wir Zusicherungen mithilfe **WP[... ]()**  
:-)

**Beispiel:**

```

int a, i, x, z;
a = read();
i = 0;
x = -1;
z = 0;
while (i != a) {
    x = x+2;
    z = z+x;
    i = i+1;
}

```

```

assert(z==a*a);

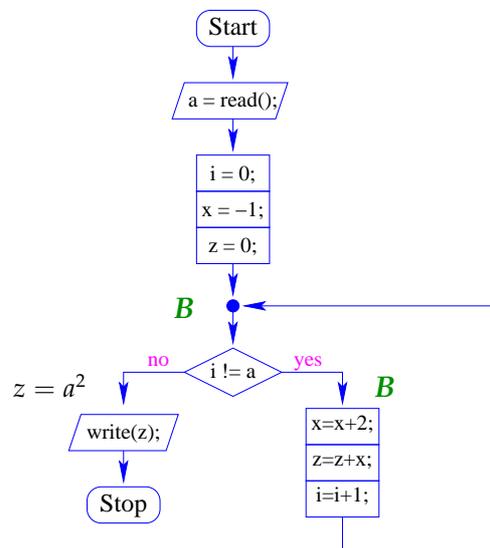
```

```

write(z);

```

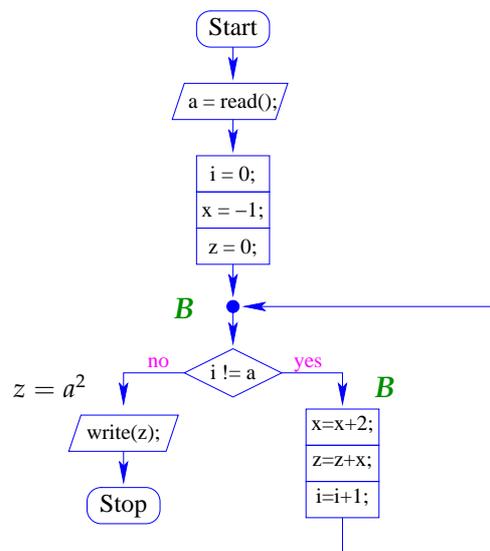
**Beispiel:**



Wir überprüfen:

$$\begin{aligned}
 \mathbf{WP}[i \neq a](z = a^2, B) & \\
 \equiv (i = a \wedge z = a^2) \vee (i \neq a \wedge B) & \\
 \equiv (i = a \wedge z = a^2) \vee (i \neq a \wedge z = i^2 \wedge x = 2i - 1) & \\
 \leftarrow (i \neq a \wedge z = i^2 \wedge x = 2i - 1) \vee (i = a \wedge z = i^2 \wedge x = 2i - 1) & \\
 \equiv z = i^2 \wedge x = 2i - 1 \equiv B & \quad \text{: -)}
 \end{aligned}$$

Orientierung:



Wir überprüfen:

$$\begin{aligned}
 \mathbf{WP}[z = 0;](B) & \equiv 0 = i^2 \wedge x = 2i - 1 \\
 & \equiv i = 0 \wedge x = -1 \\
 \mathbf{WP}[x = -1;](i = 0 \wedge x = -1) & \equiv i = 0 \\
 \mathbf{WP}[i = 0;](i = 0) & \equiv \mathbf{true} \\
 \mathbf{WP}[a = \text{read}();](\mathbf{true}) & \equiv \mathbf{true} \quad \text{: -)}
 \end{aligned}$$

## 1.4 Terminierung

### Problem:

- Mit unserer Beweistechnik können wir nur beweisen, dass eine Eigenschaft gilt wann immer wir einen Programmpunkt erreichen !!!
- Wie können wir aber garantieren, dass das Programm immer terminiert ?
- Wie können wir eine Bedingung finden, unter der das Programm immer terminiert ??
  
- Das ggT-Programm terminiert nur für Eingaben  $a, b$  mit:  $a > 0$  und  $b > 0$ .
- Das Quadrier-Programm terminiert nur für Eingaben  $a \geq 0$ .
- `while (true) ;` terminiert nie.
- Programme ohne Schleifen terminieren immer :-)

Lässt sich dieses Beispiel verallgemeinern ??

## Exkurs: Aussagenlogik

**Aussagen:** “Alle Menschen sind sterblich”,  
“Sokrates ist ein Mensch”, “Sokrates ist sterblich”

$\forall x. \text{Mensch}(x) \Rightarrow \text{sterblich}(x)$   
 $\text{Mensch}(\text{Sokrates}), \text{sterblich}(\text{Sokrates})$

**Tautologien:**  $A \vee \neg A$   
 $\forall x \in \mathbb{Z}. x < 0 \vee x = 0 \vee x > 0$

**Gesetze:**  $\neg\neg A \equiv A$   
 $A \wedge A \equiv A$   
 $\neg(A \vee B) \equiv \neg A \wedge \neg B$   
 $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$   
 $A \Rightarrow B \equiv \neg A \vee B$

## Beispiel:

```
int i, j, t;
t = 0;
i = read();
while (i>0) {
    j = read();
    while (j>0) { t = t+1; j = j-1; }
    i = i-1;
}
write(t);
```

- Die gelesene Zahl i (falls positiv) gibt an, wie oft eine Zahl j eingelesen wird.
- Die Gesamtlaufzeit ist (im wesentlichen :-)) die Summe der positiven für j gelesenen Werte  
 $\implies$  **das Programm terminiert immer !!!**

Programme nur mit for-Schleifen der Form:

```
for (i=n; i>0; i--) {...}
// im Rumpf wird i nicht modifiziert
... terminieren ebenfalls immer :-))
```

## Frage:

Wie können wir aus dieser Beobachtung eine Methode machen, die auf beliebige Schleifen anwendbar ist ?

## Idee:

- Weise nach, dass jede Scheife nur endlich oft durchlaufen wird ...
- Finde für jede Schleife eine Kenngröße  $r$ , die zwei Eigenschaften hat:
  - (1) Wenn immer der Rumpf betreten wird, ist  $r > 0$ ;
  - (2) Bei jedem Schleifen-Durchlauf wird  $r$  kleiner :-)
- Transformiere das Programm so, dass es neben der normalen Programmausführung zusätzlich die Kenngrößen  $r$  mitberechnet.
- Verifiziere, dass (1) und (2) gelten :-)

### Beispiel: Sicheres ggT-Programm

```

int a, b, x, y;
a = read(); b = read();
if (a < 0) x = -a; else x = a;
if (b < 0) y = -b; else y = b;
if (x == 0) write(y);
else if (y == 0) write(x);
    else {
        while (x != y)
            if (y > x) y = y-x;
            else      x = x-y;
        write(x);
    }

```

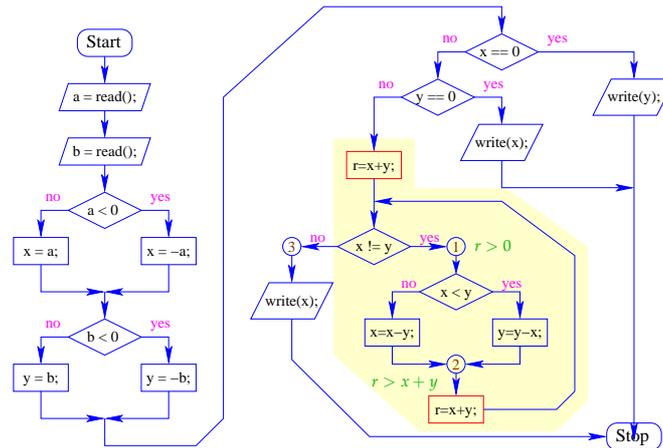
Wir wählen:  $r = x + y$

### Transformation:

```

int a, b, x, y, r;
a = read(); b = read();
if (a < 0) x = -a; else x = a;
if (b < 0) y = -b; else y = b;
if (x == 0) write(y);
else if (y == 0) write(x);
    else { r = x+y;
        while (x != y) {
            if (y > x) y = y-x;
            else      x = x-y;
            r = x+y; }
        write(x);
    }

```



An den Programmpunkten 1, 2 und 3 machen wir die Zusicherungen:

- (1)  $A \equiv x \neq y \wedge x > 0 \wedge y > 0 \wedge r = x + y$
- (2)  $B \equiv x > 0 \wedge y > 0 \wedge r > x + y$
- (3) **true**

Dann gilt:

$$A \Rightarrow r > 0 \quad \text{und} \quad B \Rightarrow r > x + y$$

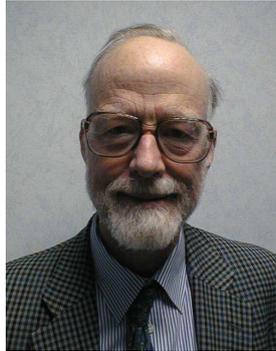
Wir überprüfen:

$$\begin{aligned} \text{WP}[x \neq y](\text{true}, A) &\equiv x = y \vee A \\ &\Leftarrow x > 0 \wedge y > 0 \wedge r = x + y \\ &\equiv C \\ \text{WP}[r = x+y](C) &\equiv x > 0 \wedge y > 0 \\ &\Leftarrow B \\ \text{WP}[x = x-y](B) &\equiv x > y \wedge y > 0 \wedge r > x \\ \text{WP}[y = y-x](B) &\equiv x > 0 \wedge y > x \wedge r > y \\ \text{WP}[y > x](\dots, \dots) &\equiv (x > y \wedge y > 0 \wedge r > x) \vee \\ &\quad (x > 0 \wedge y > x \wedge r > y) \\ &\Leftarrow x \neq y \wedge x > 0 \wedge y > 0 \wedge r = x + y \\ &\equiv A \end{aligned}$$

Orientierung:



## 1.5 Modulare Verification und Prozeduren



Tony Hoare, Microsoft Research, Cambridge

### Idee:

- Modularisiere den Korrektheitsbeweis so, dass Teilbeweise für wiederkehrende Aufgaben wiederverwendet werden können.
- Betrachte Aussagen der Form:

$$\{A\} \quad p \quad \{B\}$$

... das heißt:

Gilt **vor** der Ausführung des Programmstücks  $p$  Eigenschaft  $A$  und terminiert die Programm-Ausführung, dann gilt **nach** der Ausführung von  $p$  Eigenschaft  $B$ .

$A$  : Vorbedingung

$B$  : Nachbedingung

### Beispiele:

$$\{x > y\} \quad z = x - y; \quad \{z > 0\}$$

$$\{\mathbf{true}\} \quad \text{if } (x < 0) \ x = -x; \quad \{x \geq 0\}$$

$$\{x > 7\} \quad \text{while } (x \neq 0) \ x = x - 1; \quad \{x = 0\}$$

$$\{\mathbf{true}\} \quad \text{while } (\mathbf{true}); \quad \{\mathbf{false}\}$$

Modulare Verifikation können wir benutzen, um die Korrektheit auch von Programmen mit Funktionen nachzuweisen :-)

## Vereinfachung:

Wir betrachten nur

- **Prozeduren**, d.h. statische Methoden ohne Rückgabewerte;
- nur **globale Variablen**, d.h. alle Variablen sind ebenfalls `static`.  
// werden wir später verallgemeinern :-)

## Beispiel:

```
int a, b, x, y;          void mm() {
                          if (a>b) {
void main () {           x = a;
                          y = b;
    a = read();          } else {
    b = read();          y = a;
    mm();                x = b;
    write (x-y);        }
                          }
}
```

## Kommentar:

- Der Einfachheit halber haben wir alle Vorkommen von `static` gestrichen :-)
- Die Prozedur-Definitionen sind nicht rekursiv.
- Das Programm liest zwei Zahlen ein.
- Die Prozedur `minmax` speichert die größere in `x`, die kleinere in `y` ab.
- Die Differenz von `x` und `y` wird ausgegeben.
- Wir wollen zeigen, dass gilt:

$$\{a \geq b\} \text{ mm()}; \{a = x\}$$

## Vorgehen:

- Für jede Prozedur `f()` stellen wir ein Tripel bereit:

$$\{A\} f(); \{B\}$$

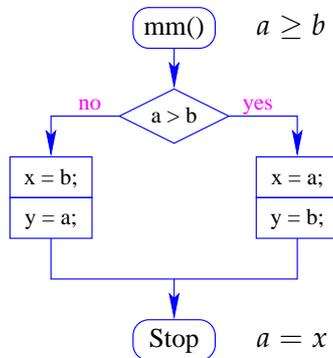
- Unter dieser **globalen Hypothese**  $H$  verifizieren wir, dass sich für jede Prozedurdefinition `void f() { ss }` zeigen lässt:

$$\{A\} \text{ ss } \{B\}$$

- Wann immer im Programm ein Prozeduraufruf vorkommt, benutzen wir dabei die Tripel aus  $H \dots$

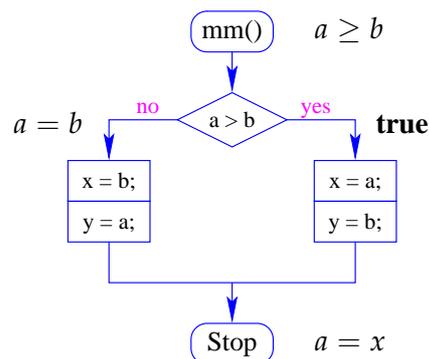
... im Beispiel:

Wir überprüfen:



... im Beispiel:

Wir überprüfen:



Diskussion:

- Die Methode funktioniert auch, wenn die Prozedur einen Rückgabewert hat: den können wir mit einer globalen Variable return simulieren, in die das jeweilige Ergebnis geschrieben wird :-)
- Es ist dagegen nicht offensichtlich, wie die Vor- und Nachbedingung für Prozeduraufrufe gewählt werden soll, wenn eine Funktion an mehreren Stellen aufgerufen wird ...
- Noch schwieriger wird es, wenn eine Prozedur rekursiv ist: dann hat sie potentiell unbeschränkt viele verschiedene Aufrufe !?

Beispiel:

```

int x, m0, m1, t;          void f() {
                            x = x-1;
void main () {             if (x>1) f();
    x = read();            t = m1;
    m0 = 1; m1 = 1;        m1 = m0+m1;
    if (x > 1) f();        m0 = t;
    write (m1);           }
}

```

### Kommentar:

- Das Programm liest eine Zahl ein.
- Ist diese Zahl höchstens 1, liefert das Programm 1 ...
- Andernfalls berechnet das Programm die **Fibonacci-Funktion fib** :-)
- Nach einem Aufruf von  $f$  enthalten die Variablen  $m_0$  und  $m_1$  jeweils die Werte  $\text{fib}(i-1)$  und  $\text{fib}(i)$  ...

### Problem:

- Wir müssen in der Logik den  $i$ -ten vom  $(i+1)$ -ten Aufruf zu unterscheiden können :-)
- Das ist **einfacher**, wenn wir **logische Hilfsvariablen**  $\underline{l} = l_1, \dots, l_n$  zur Verfügung haben, in denen wir (ausgewählte) Werte **vor** dem Aufruf retten können ...

### Im Beispiel:

$\{A\} f(); \{B\}$  wobei

$$A \equiv x = l \wedge x > 1 \wedge m_0 = m_1 = 1$$

$$B \equiv l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1}$$

### Allgemeines Vorgehen:

- Wieder starten wir mit einer **globalen Hypothese**  $H$ , die für jeden Aufruf  $f();$  eine Beschreibung bereitstellt:

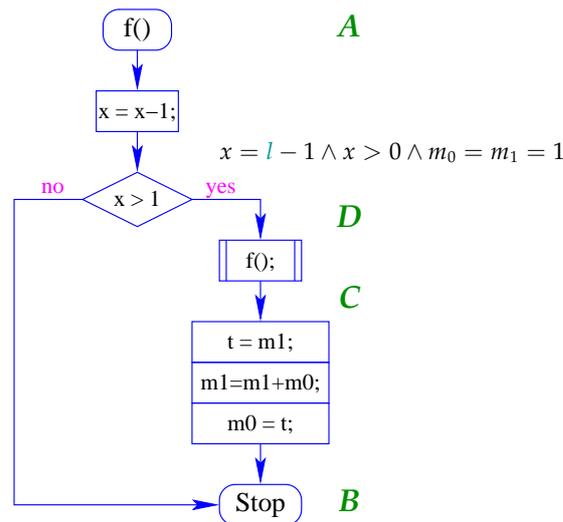
$\{A\} f(); \{B\}$

// sowohl  $A$  wie  $B$  können  $l_i$  enthalten :-)

- Unter dieser **globalen Hypothese**  $H$  verifizieren wir, dass für jede Funktionsdefinition  $\text{void } f() \{ \text{ss} \}$  gilt:

$\{A\} \text{ss } \{B\}$

### ... im Beispiel:



- Wir starten von der Zusicherung für den Endpunkt:

$$B \equiv l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1}$$

- Die Zusicherung  $C$  ermitteln wir mithilfe von  $\mathbf{WP}[\dots]$  und **Abschwächung** ...

$$\begin{aligned} \mathbf{WP}[\text{t=m1; m1=m1+m0; m0=t;}](B) & \\ \equiv l - 1 > 0 \wedge m_1 + m_0 \leq 2^l \wedge m_1 \leq 2^{l-1} & \\ \Leftarrow l - 1 > 1 \wedge m_1 \leq 2^{l-1} \wedge m_0 \leq 2^{l-2} & \\ \equiv C & \end{aligned}$$

### Frage:

Wie nutzen wir unsere **globale Hypothese**, um einen konkreten Prozeduraufruf zu behandeln ???

### Idee:

- Die Aussage  $\{A\} f(); \{B\}$  repräsentiert eine **Wertetabelle** für  $f()$  :-)
- Diese Wertetabelle können wir logisch repräsentieren als die Implikation:

$$\forall \underline{l}. (A[\underline{h}/\underline{x}] \Rightarrow B)$$

//  $\underline{h}$  steht für eine Folge von **Hilfsvariablen**

Die Werte der Variablen  $\underline{x}$  vor dem Aufruf stehen in den **Hilfsvariablen** :-)

## Beispiele:

Funktion:       void double () { x = 2\*x; }  
Spezifikation:   {x = l} double(); {x = 2l}  
Tabelle:          $\forall l. (h = l) \Rightarrow (x = 2l)$   
                   $\equiv (x = 2h)$

Für unsere Fibonacci-Funktion berechnen wir:

$$\begin{aligned} \forall l. (h > 1 \wedge h = l \wedge h_0 = h_1 = 1) &\Rightarrow \\ &l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1} \\ \equiv (h > 1 \wedge h_0 = h_1 = 1) &\Rightarrow m_1 \leq 2^h \wedge m_0 \leq 2^{h-1} \end{aligned}$$

Ein anderes Paar  $(A_1, B_1)$  von Zusicherungen liefert ein gültiges Tripel  $\{A_1\} f(); \{B_1\}$ , falls wir zeigen können:

$$\frac{\forall l. A[h/x] \Rightarrow B \quad A_1[h/x]}{B_1}$$

Ein anderes Paar  $(A_1, B_1)$  von Zusicherungen liefert ein gültiges Tripel  $\{A_1\} f(); \{B_1\}$ , falls wir zeigen können:

$$\frac{\forall l. A[h/x] \Rightarrow B \quad A_1[h/x]}{B_1}$$

**Beispiel:**       double()

$$\begin{aligned} A &\equiv x = l & B &\equiv x = 2l \\ A_1 &\equiv x \geq 3 & B_1 &\equiv x \geq 6 \end{aligned}$$

Ein anderes Paar  $(A_1, B_1)$  von Zusicherungen liefert ein gültiges Tripel  $\{A_1\} f(); \{B_1\}$ , falls wir zeigen können:

$$\frac{\forall l. A[h/x] \Rightarrow B \quad A_1[h/x]}{B_1}$$

**Beispiel:**       double()

$$\begin{aligned} A &\equiv x = l & B &\equiv x = 2l \\ A_1 &\equiv x \geq 3 & B_1 &\equiv x \geq 6 \end{aligned}$$

Wir überprüfen:

$$\frac{x = 2h \quad h \geq 3}{x \geq 6}$$

:-)

### Bemerkungen:

Gültige Paare  $(A_1, B_1)$  erhalten wir z.B.,

- indem wir die logischen Variablen **substituieren**:

$$\frac{\{x = l\} \text{ double}(); \{x = 2l\}}{\{x = l - 1\} \text{ double}(); \{x = 2(l - 1)\}}$$

- indem wir eine Bedingung C an die logischen Variablen hinzufügen:

$$\frac{\{x = l\} \text{ double}(); \{x = 2l\}}{\{x = l \wedge l > 0\} \text{ double}(); \{x = 2l \wedge l > 0\}}$$

### Bemerkungen (Forts.):

Gültige Paare  $(A_1, B_1)$  erhalten wir z.B. auch,

- indem wir die Vorbedingung **verstärken** bzw. die Nachbedingung **abschwächen**:

$$\frac{\{x = l\} \text{ double}(); \{x = 2l\}}{\{x > 0 \wedge x = l\} \text{ double}(); \{x = 2l\}}$$

$$\frac{\{x = l\} \text{ double}(); \{x = 2l\}}{\{x = l\} \text{ double}(); \{x = 2l \vee x = -1\}}$$

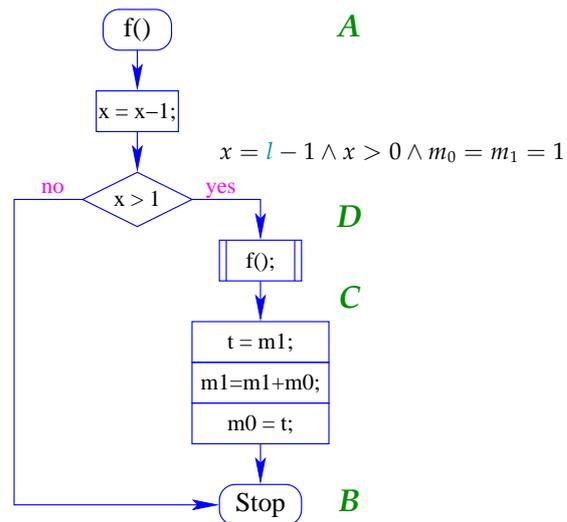
### Anwendung auf Fibonacci:

Wir wollen beweisen:  $\{D\} \text{ f}(); \{C\}$

$$\begin{aligned} A &\equiv x > 1 \wedge l = x \wedge m_0 = m_1 = 1 \\ A[(l-1)/l] &\equiv x > 1 \wedge l-1 = x \wedge m_0 = m_1 = 1 \\ &\equiv D \end{aligned}$$

$$\begin{aligned} B &\equiv l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1} \\ B[(l-1)/l] &\equiv l-1 > 1 \wedge m_1 \leq 2^{l-1} \wedge m_0 \leq 2^{l-2} \\ &\equiv C \quad \quad \quad \text{:-) } \end{aligned}$$

### Orientierung:



Für die bedingte Verzweigung verifizieren wir:

$$\mathbf{WP}_{[x>1]}(B, D) \equiv (x \leq 1 \wedge l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1}) \vee (x > 1 \wedge x = l - 1 \wedge m_1 = m_0 = 1)$$

$$\Leftarrow x > 0 \wedge x = l - 1 \wedge m_0 = m_1 = 1$$

:-))

## 1.6 Prozeduren mit lokalen Variablen

- Prozeduren  $f()$  modifizieren globale Variablen.
- Die Werte der lokalen Variablen des Aufrufers **vor** und **nach** dem Aufruf sind unverändert :-)

### Beispiel:

```
{int y= 17; double(); write(y);}
```

Vor und nach dem Aufruf von `double()` gilt:  $y = 17$  :-)

- Der Erhaltung der lokalen Variablen tragen wir **automatisch** Rechnung, wenn wir bei der Aufstellung der globalen Hypothese beachten:
  - Die Vor- und Nachbedingungen:  $\{A\}, \{B\}$  für Prozeduren sprechen nur über globale Variablen !
  - Die  $h$  werden nur für die **globalen** Variablen eingesetzt !!
- Als neuen Spezialfall der Adaption erhalten wir:

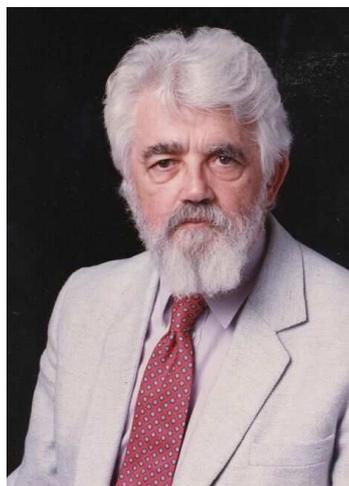
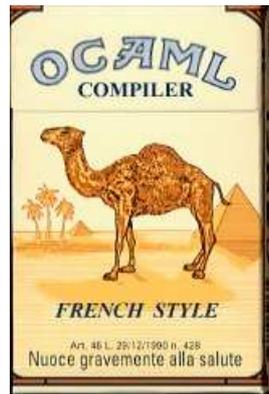
$$\frac{\{A\} \ f(); \ \{B\}}{\{A \wedge C\} \ f(); \ \{B \wedge C\}}$$

falls  $C$  nur über logische Variablen oder lokale Variablen des Aufrufers spricht :-)

### Abschluss:

- Jedes weitere Sprachkonstrukt erfordert neue Methoden zur Verifikation :-)
- Wie behandelt man dynamische Datenstrukturen, Objekte, Klassen, Vererbung ?
- Wie geht man mit **Nebenläufigkeit**, **Reaktivität** um ??
- Erlauben die vorgestellten Methoden **alles** zu beweisen  $\implies$  **Vollständigkeit** ?
- Wie weit lässt sich Verifikation automatisieren ?
- Wieviel Hilfe muss die **Programmiererin** und/oder die **Verifiziererin** geben ?

# Funktionale Programmierung



John McCarthy, Stanford



Robin Milner, Edinburgh



Xavier Leroy, INRIA, Paris

## 2 Grundlagen

- Interpreter-Umgebung
- Ausdrücke
- Wert-Definitionen
- Komplexere Datenstrukturen
- Listen
- Definitionen (Forts.)
- Benutzer-definierte Datentypen

### 2.1 Die Interpreter-Umgebung

Der Interpreter wird mit `ocaml` aufgerufen...

```
seidl@linux:~> ocaml
Objective Caml version 3.09.3
#
```

Definitionen von Variablen, Funktionen, ... können direkt eingegeben werden :-)  
Alternativ kann man sie aus einer Datei einlesen:

```
# #use "Hallo.ml";;
```

## 2.2 Ausdrücke

```
# 3+4;;  
- : int = 7  
# 3+  
  4;;  
- : int = 7  
#
```

- Bei `#` wartet der Interpreter auf Eingabe.
- Das `;;` bewirkt Auswertung der bisherigen Eingabe.
- Das Ergebnis wird berechnet und mit seinem Typ ausgegeben.

**Vorteil:** Das Testen von einzelnen Funktionen kann stattfinden, ohne jedesmal neu zu übersetzen :-)

### Vordefinierte Konstanten und Operatoren:

Typ	Konstanten: Beispiele	Operatoren
int	0 3 -7	+ - * / mod
float	-3.0 7.0	+. -. *. /. not    && ^
bool	true false	
string	"hallo"	
char	'a' 'b'	

Typ	Vergleichsoperatoren
int	= <> < <= >= >
float	= <> < <= >= >
bool	= <> < <= >= >
string	= <> < <= >= >
char	= <> < <= >= >

```
# -3.0/.4.0;;
- : float = -0.75
# "So"^" "^"geht"^" "^"das";;
- : string = "So geht das"
# 1>2 || not (2.0<1.0);;
- : bool = true
```

## 2.3 Wert-Definitionen

Mit `let` kann man eine **Variable** mit einem Wert belegen.  
Die Variable behält diesen Wert **für immer** :-)

```
# let seven = 3+4;;
val seven : int = 7
# seven;;
- : int = 7
```

**Achtung:** Variablen-Namen werden **klein** geschrieben !!!

Eine erneute Definition für `seven` weist **nicht** `seven` einen neuen Wert zu, sondern erzeugt eine **neue** Variable mit Namen `seven`.

```
# let seven = 42;;
val seven : int = 42
# seven;;
- : int = 42
# let seven = "seven";;
val seven : string = "seven"
```

Die alte Definition wurde **unsichtbar** (ist aber trotzdem noch vorhanden :-)  
Offenbar kann die neue Variable auch einen **anderen Typ** haben :-)

## 2.4 Komplexere Datenstrukturen

- Paare:

```
# (3,4);;
- : int * int = (3, 4)
# (1=2,"hallo");;
- : bool * string = (false, "hallo")
```

- Tupel:

```
# (2,3,4,5);;
- : int * int * int * int = (2, 3, 4, 5)
# ("hallo",true,3.14159);;
-: string * bool * float = ("hallo", true, 3.14159)
```

### Simultane Definition von Variablen:

```
# let (x,y) = (3,4.0);;
val x : int = 3
val y : float = 4.

# let (3,y) = (3,4.0);;
val y : float = 4.0
```

## Records:            Beispiel:

```
# type person = {vor:string; nach:string; alter:int};;
type person = { vor : string; nach : string; alter : int; }
# let paul = { vor="Paul"; nach="Meier"; alter=24 };;
val paul : person = {vor = "Paul"; nach = "Meier"; alter = 24}
# let hans = { nach="kohl"; alter=23; vor="hans"};;
val hans : person = {vor = "hans"; nach = "kohl"; alter = 23}
# let hans1 = {alter=23; nach="kohl"; vor="hans"}
val hans1 : person = {vor = "hans"; nach = "kohl"; alter = 23}
# hans=hans1;;
- : bool = true
```

## Bemerkung:

- ... Records sind Tupel mit benannten Komponenten, deren Reihenfolge irrelevant ist :-)
- ... Als neuer Typ muss ein Record vor seiner Benutzung mit einer `type`-Deklaration eingeführt werden.
- ... Typ-Namen und Record-Komponenten werden `klein` geschrieben :-)

## Zugriff auf Record-Komponenten

### ... per Komponenten-Selektion:

```
# paul.vor;;
- : string = "Paul"
```

### ... mit Pattern Matching:

```
# let {vor=x;nach=y;alter=z} = paul;;
val x : string = "Paul"
val y : string = "Meier"
val z : int = 24
```

### ... und wenn einen nicht alles interessiert:

```
# let {vor=x} = paul;;
val x : string = "Paul"
```

## Fallunterscheidung: `match` und `if`

```
match n
with 0 -> "Null"
     | 1 -> "Eins"
     | _ -> "Soweit kann ich nicht zaehlen!"

match e
with true  -> e1
     | false -> e2
```

Das zweite Beispiel kann auch so geschrieben werden (`-:`

```
if e then e1 else e2
```

### Vorsicht bei redundanten und unvollständigen Matches!

```
# let n = 7;;
val n : int = 7
# match n with 0 -> "null";;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
Exception: Match_failure ("", 5, -13).
# match n
  with 0 -> "null"
       | 0 -> "eins"
       | _ -> "Soweit kann ich nicht zaehlen!";;
Warning: this match case is unused.
- : string = "Soweit kann ich nicht zaehlen!"
```

## 2.5 Listen

Listen werden mithilfe von `[]` und `::` konstruiert.

**Kurzschreibweise:** `[42; 0; 16]`

```
# let mt = [];;
val mt : 'a list = []
# let l1 = 1::mt;;
val l1 : int list = [1]
# let l = [1;2;3];;
val l : int list = [1; 2; 3]
# let l = 1::2::3::[];;
val l : int list = [1; 2; 3]
```

## Achtung:

Alle Elemente müssen den gleichen Typ haben:

```
# 1.0::1::[];;
```

This expression has type int but is here used with type float

`tau list` beschreibt Listen mit Elementen vom Typ `tau :-)`

Der Typ `'a` ist eine **Typ-Variable**:

`[]` bezeichnet eine leere Liste für beliebige Element-Typen `:-))`

## Pattern Matching auf Listen:

```
# match l
  with []      -> -1
       | x::xs -> x;;
-: int = 1
```

## 2.6 Definitionen von Funktionen

```
# let double x = 2*x;;
val double : int -> int = <fun>
# (double 3, double (double 1));;
- : int * int = (6,4)
```

- Nach dem Funktions-Namen kommen die Parameter.
- Der Funktionsname ist damit auch nur eine Variable, deren Wert eine Funktion ist `:-)`
- Alternativ können wir eine Variable einführen, deren Wert direkt eine Funktion beschreibt ...

```
# let double = fun x -> 2*x;;
val double : int -> int = <fun>
```

- Diese Funktionsdefinition beginnt mit `fun`, gefolgt von den formalen Parametern.
- Nach `->` kommt die Berechnungsvorschrift.
- Die linken Variablen dürfen rechts benutzt werden `:-)`

## Achtung:

Funktionen sehen die Werte der Variablen, die zu ihrem **Definitionszeitpunkt** sichtbar sind:

```

# let faktor = 2;;
val faktor : int = 2
# let double x = faktor*x;;
val double : int -> int = <fun>
# let faktor = 4;;
val faktor : int = 4
# double 3;;
- : int = 6

```

## Achtung:

Eine Funktion ist ein Wert:

```

# double;;
- : int -> int = <fun>

```

## Rekursive Funktionen:

Eine Funktion ist **rekursiv**, wenn sie sich selbst aufruft.

```

# let rec fac n = if n<2 then 1 else n * fac (n-1);;
val fac : int -> int = <fun>
# let rec fib = fun x -> if x <= 1 then 1
                        else fib (x-1) + fib (x-2);;
val fib : int -> int = <fun>

```

Dazu stellt **Ocaml** das Schlüsselwort `rec` bereit :-)

Rufen mehrere Funktionen sich gegenseitig auf, heißen sie **verschränkt rekursiv**.

```

# let rec even n = if n=0 then "even" else odd (n-1)
                  and odd n = if n=0 then "odd" else even (n-1);;
val even : int -> string = <fun>
val odd : int -> string = <fun>

```

Wir kombinieren ihre Definitionen mit dem Schlüsselwort `and` :-)

## Definition durch Fall-Unterscheidung:

```

# let rec len = fun l -> match l
                        with [] -> 0
                             | x::xs -> 1 + len xs;;
val len : 'a list -> int = <fun>
# len [1;2;3];;
- : int = 3

```

... kann kürzer geschrieben werden als:

```
# let rec len = function [] -> 0
                    | x::xs -> 1 + len xs;;
val len : 'a list -> int = <fun>
# len [1;2;3];;
- : int = 3
```

## Fall-Unterscheidung bei mehreren Argumenten:

```
# let rec app l y = match l
                   with [] -> y
                    | x::xs -> x :: app xs y;;
val app : 'a list -> 'a list -> 'a list = <fun>
# app [1;2] [3;4];;
- : int list = [1; 2; 3; 4]
```

... kann auch geschrieben werden als:

```
# let rec app = function [] -> fun y -> y
                  | x::xs -> fun y -> x::app xs y;;
val app : 'a list -> 'a list -> 'a list = <fun>
# app [1;2] [3;4];;
- : int list = [1; 2; 3; 4]
```

## Lokale Definitionen

Definitionen können mit `let` lokal eingeführt werden:

```
# let    x = 5
  in let sq = x*x
  in    sq+sq;;
- : int = 50
# let facit n = let rec
                iter m yet = if m>n then yet
                             else iter (m+1) (m*yet)
                in iter 2 1;;
val facit : int -> int = <fun>
```

## 2.7 Benutzerdefinierte Typen

**Beispiel:** Spielkarten

Wie kann man die Farbe und den Wert einer Karte spezifizieren?

**1. Idee:** Benutze Paare von Strings und Zahlen, z.B.

```
("Karo",10)  ≡  Karo Zehn
("Kreuz",12) ≡  Kreuz Bube
("Pik",1)   ≡  Pik As
```

**Nachteile:**

- Beim Test auf eine Farbe muss immer ein String-Vergleich stattfinden  
→ ineffizient!
- Darstellung des Buben als 12 ist nicht intuitiv  
→ unleserliches Programm!
- Welche Karte repräsentiert das Paar ("Kaor",1)?  
(Tippfehler werden vom Compiler nicht bemerkt)

**Besser:** Aufzählungstypen von **Ocaml**.

**Beispiel:** Spielkarten

**2. Idee:** Aufzählungstypen

```
# type farbe = Karo | Herz | Pik | Kreuz;;
type farbe = Karo | Herz | Pik | Kreuz
# type wert = Neun | Bube | Dame | Koenig | Zehn | As;;
type wert = Neun | Bube | Dame | Koenig | Zehn | As
# Kreuz;;
- : farbe = Kreuz
# let pik_bube = (Pik,Bube);;
val pik_bube : farbe * wert = (Pik, Bube)
```

**Vorteile:**

- Darstellung ist intuitiv.
- Tippfehler werden erkannt:

```
# (Kaor,As);;
Unbound constructor Kaor
```

→ Die interne Repräsentation ist **effizient** :-)

### Bemerkungen:

- Durch `type` wird ein **neuer Typ** definiert.
- Die Alternativen heißen **Konstruktoren** und werden durch `|` getrennt.
- Jeder Konstruktor wird groß geschrieben und ist **eindeutig** einem Typ zugeordnet.

### Aufzählungstypen (cont.)

Konstruktoren können verglichen werden:

```
# Kreuz < Karo;;  
- : bool = false;;  
# Kreuz > Karo;;  
- : bool = true;;
```

Pattern Matching auf Konstruktoren:

```
# let istTrumpf = function  
    (Karo,_)    -> true  
  | (_,Bube)   -> true  
  | (_,Dame)   -> true  
  | (Herz,Zehn) -> true  
  | (_,_)      -> false;;  
  
val istTrumpf : farbe * wert -> bool = <fun>
```

Damit ergibt sich z.B.:

```
# istTrumpf (Karo,As);;  
- : bool = true  
# istTrumpf (Pik,Koenig);;  
- : bool = false
```

### Eine andere nützliche Funktion:

```
# let string_of_farbe = function  
    Karo  -> "Karo"  
  | Herz -> "Herz"  
  | Pik  -> "Pik"  
  | Kreuz -> "Kreuz";;  
  
val string_of_farbe : farbe -> string = <fun>
```

## Beachte:

Die Funktion `string_of_farbe` wählt für eine Farbe in **konstanter Zeit** den zugehörigen String aus (der Compiler benutzt – hoffentlich – **Sprungtabellen** :-)

Jetzt kann **Ocaml** schon fast Karten spielen:

```
# let sticht = function
    ((Herz,Zehn),_)      -> true
  | (_,(Herz,Zehn))     -> false
  | ((f1,Dame),(f2,Dame)) -> f1 > f2
  | (_,Dame),_         -> true
  | (_,_,Dame)         -> false
  | ((f1,Bube),(f2,Bube)) -> f1 > f2
  | (_,Bube),_         -> true
  | (_,_,Bube)         -> false
  | ((Karo,w1),(Karo,w2)) -> w1 > w2
  | ((Karo,_),_)       -> true
  | (_,(Karo,_))       -> false
  | ((f1,w1),(f2,w2))  -> if f1=f2 then w1 > w2
                          else false;;

...
# let nimm (karte2,karte1) =
    if sticht (karte2,karte1) then karte2 else karte1;;

# let stich (karte1,karte2,karte3,karte4) =
    nimm (karte4, nimm (karte3, nimm (karte2,karte1))));;

# stich ((Herz,Koenig),(Karo,As), (Herz,Bube),(Herz,As));;
- : farbe * wert = (Herz, Bube)
# stich((Herz,Koenig),(Pik,As), (Herz,Koenig),(Herz,As));;
- : farbe * wert = (Herz, As)
```

## Summentypen

Summentypen sind eine Verallgemeinerung von Aufzählungstypen, bei denen die Konstruk-toren **Argumente** haben.

## Beispiel: Hexadezimalzahlen

```
type hex = Digit of int | Letter of char;;
let char2dez c = if c >= 'A' && c <= 'F'
  then (Char.code c)-55
  else if c >= 'a' && c <= 'f'
  then (Char.code c)-87
  else -1;;
let hex2dez = function
  Digit n -> n
  | Letter c -> char2dez c;;
```

**Char** ist ein **Modul**, der Funktionalität für **char** sammelt :-)

Ein Konstruktor, der mit `type t = Con of <typ> | ...`

definiert wurde, hat die Funktionalität `Con : <typ> -> t` — muss aber stets **angewandt** vorkommen ...

```
# Digit;;
The constructor Digit expects 1 argument(s),
but is here applied to 0 argument(s)
# let a = Letter 'a';;
val a : hex = Letter 'a'
# Letter 1;;
This expression has type int but is here used with type char
# hex2dez a;;
- : int = 10
```

**Datentypen können auch rekursiv sein:**

```
type folge = Ende | Dann of (int * folge)

# Dann (1, Dann (2, Ende));;
- : folge = Dann (1, Dann (2, Ende))
```

Beachte die Ähnlichkeit zu Listen :-)

**Rekursive Datentypen führen wieder zu rekursiven Funktionen:**

```
# let rec n_tes = function
  (_,Ende) -> -1
  | (0,Dann (x,_)) -> x
  | (n,Dann (_, rest)) -> n_tes (n-1,rest);;
val n_tes : int * folge -> int = <fun>

# n_tes (4, Dann (1, Dann (2, Ende)));;
- : int = -1
# n_tes (2, Dann (1, Dann(2, Dann (5, Dann (17, Ende)))));;
- : int = 5
```

## Anderes Beispiel:

```
# let rec runter = function
    0 -> Ende
    | n -> Dann (n, runter (n-1));;
val runter : int -> folge = <fun>

# runter 4;;
- : folge = Dann (4, Dann (3, Dann (2, Dann (1, Ende))));;
# runter -1;;
Stack overflow during evaluation (looping recursion?).
```

## Der Options-Datentyp

Ein eingebauter Datentyp in **Ocaml** ist `option` mit den zwei Konstruktoren `None` und `Some`.

```
# None;;
- : 'a option = None
# Some 10;
- : int option = Some 10
```

Er wird häufig benutzt, wenn eine Funktion nicht für alle Eingaben eine Lösung berechnet:

```
# let rec n_tes = function
    (n,Ende) -> None
    | (0, Dann (x,_)) -> Some x
    | (n, Dann (_,rest)) -> n_tes (n-1,rest);;
val n_tes : int * folge -> int option = <fun>

# n_tes (4,Dann (1, Dann (2, Ende))));;
- : int option = None
# n_tes (2, Dann (1, Dann (2, Dann (5, Dann (17, Ende))))));;
- : int option = Some 5
```

## 3 Funktionen – näher betrachtet

- Rekursionsarten
- Funktionen höherer Ordnung
  - Currying

→ Partielle Anwendung

- Polymorphe Funktionen
- Polymorphe Datentypen
- Namenlose Funktionen

### 3.1 Rekursionsarten

Je nach Art der rekursiven Aufrufe unterscheiden wir folgende Arten von Rekursion:

1. **End-Rekursion:** Jeder rekursive Aufruf liefert gleichzeitig den Rückgabewert.

```
let rec fac1 = function
  (1,acc) -> acc
  | (n,acc) -> fac1 (n-1,n*acc);;
```

```
let rec loop x = if x<2 then x
                 else if x mod 2 = 0 then loop (x/2)
                 else loop (3*x+1);;
```

2. **Repetitive Rekursion:** Es gibt in jedem Zweig höchstens einen rekursiven Aufruf.

```
let rec fac = function
  1 -> 1
  | n -> n * fac (n-1);;
```

3. **Allgemeine Rekursion:**

```
let rec fib = function
  0 -> 1
  | 1 -> 1
  | n -> fib (n-1) + fib (n-2);;
```

```
let rec f n = if n<2 then n
              else f (f (n/2));;
```

Das **beste Speicherverhalten** hat die End-Rekursion. Es wird zur allgemeinen Rekursion hin immer schlechter.

Der typische Fall von repetitiver Rekursion hat die folgende Form:

```
let rec f x = if t(x) then v else e(x,f(g(x)))
```

**Beispiel:** Fakultät

```
let rec fac x = if x=1 then 1 else x * fac (x-1)
```

Hier ist  $t(x) \equiv x=1$ ,  $v \equiv 1$ ,  $g(x) \equiv x-1$  und  $e(x,y) \equiv x*y$ .  
Wie wir gesehen haben, gibt es auch eine end-rekursive Variante:

```
let rec fac1 (x,a) = if x=1 then a else fac1 (x-1,x*a)
```

Im allgemeinen Fall kann man  $f$  ersetzen durch:

```
let rec f1 (x,a) = if t(x) then a else f1 (g(x),e(x,a))
```

Dann ist  $f1(x,v)$  gleich  $f\ x$  — aber nur dann, wenn  $e$  eine **assoziative, kommutative** Funktion ist:

```
n*(...*(2*1)) = 1*(2*(...*n))
```

Der zusätzliche Parameter  $a$  heißt auch **Akkumulator**.

**Vorsicht bei Listenfunktionen:**

```
let rec f x      = if x=0 then [] else x :: f (x-1)
let rec f1 (x,a) = if x=0 then a  else f1 (x-1,x::a)
```

Der Listenkonstruktor  $::$  ist weder kommutativ noch assoziativ. Deshalb berechnen  $f\ x$  und  $f1\ (x,[])$  nicht den gleichen Wert:

```
# f 5;;
- : int list = [5; 4; 3; 2; 1]
# f1 (5,[]);;
- : int list = [1; 2; 3; 4; 5]
```

Selbst allgemeine Rekursion kann manchmal linearisiert werden:

```
let rec fib = function
  | 0 -> 1
  | 1 -> 1
  | n -> fib (n-1) + fib (n-2);;

let fib1 n =
  let rec iter x (m0,m1) =
    if x<=1 then m1 else iter (x-1) (m1,m0+m1)
  in iter n (1,1)
```

Das lässt sich aber nicht **so einfach** verallgemeinern :-)

## 3.2 Funktionen höherer Ordnung

Betrachte die beiden Funktionen

```
let f (a,b) = a+b+1;;  
let g a b   = a+b+1;;
```

Auf den ersten Blick scheinen sich  $f$  und  $g$  nur in der Schreibweise zu unterscheiden. Aber sie haben einen **anderen Typ**:

```
# f;;  
- : int * int -> int = <fun>  
# g;;  
- : int -> int -> int = <fun>
```

- Die Funktion  $f$  hat ein Argument, welches aus dem **Paar**  $(a,b)$  besteht. Der Rückgabewert ist  $a+b+1$ .
- $g$  hat ein Argument  $a$  vom Typ `int`. Das Ergebnis einer Anwendung auf  $a$  ist **wieder eine Funktion**, welche, angewendet auf ein weiteres Argument  $b$ , das Ergebnis  $a+b+1$  liefert:

```
# f (3,5);;  
- : int = 9  
# let g1 = g 3;;  
val g1 : int -> int = <fun>  
# g1 5;;  
- : int = 9
```



Haskell B. Curry, 1900–1982

Das Prinzip heißt nach seinem Erfinder Haskell B. Curry **Currying**.

- g heißt Funktion **höherer Ordnung**, weil ihr Ergebnis wieder eine Funktion ist.
- Die Anwendung von g auf ein Argument heißt auch **partiell**, weil das Ergebnis nicht vollständig ausgewertet ist, sondern eine weitere Eingabe erfordert.

Das Argument einer Funktion kann auch wieder selbst eine Funktion sein:

```
# let apply f a b = f (a,b);;
val apply ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
...
...
# let plus (x,y) = x+y;;
val plus : int * int -> int = <fun>
# apply plus;;
- : int -> int -> int = <fun>
# let plus2 = apply plus 2;;
val plus2 : int -> int = <fun>
# let plus3 = apply plus 3;;
val plus3 : int -> int = <fun>
# plus2 (plus3 4);;
- : int = 9
```

### 3.3 Funktionen als Daten

Funktionen sind **Daten** und können daher in Datenstrukturen vorkommen:

```
# ((+), plus3) ;
- : (int -> int -> int) * (int -> int) = (<fun>, <fun>);;
# let rec plus_list = function
    [] -> []
  | x::xs -> (+) x :: plus_list xs;;
val plus_list : int list -> (int -> int) list = <fun>
# let l = plus_list [1;2;3];;
val l : (int -> int) list = [<fun>; <fun>; <fun>]

// (+) : int -> int -> int ist die Funktion zum Operator +
```

...

```
# let do_add n =
    let rec add_list = function
        [] -> []
      | f::fs -> f n :: add_list fs
    in add_list ;;
val do_add : 'a -> ('a -> 'b) list -> 'b list = <fun>
# do_add 5 l;;
- : int list = [6;7;8]

# let rec sum = function
    [] -> 0
  | f::fs -> f (sum fs);;
val sum : (int -> int) list -> int = <fun>
# sum l;;
- : int = 6
```

### 3.4 Einige Listen-Funktionen

```
let rec map f = function
    [] -> []
  | x::xs -> f x :: map f xs

let rec fold_left f a = function
    [] -> a
```

```

    | x::xs -> fold_left f (f a x) xs

let rec fold_right f = function
    [] -> fun b -> b
    | x::xs -> fun b -> f x (fold_right f xs b)

let rec find_opt f = function
    [] -> None
    | x::xs -> if f x then Some x
                else find_opt f xs

```

### Beachte:

- Diese Funktionen abstrahieren von dem Verhalten der Funktion  $f$ . Sie spezifizieren das Rekursionsverhalten gemäß der Listenstruktur, unabhängig von den Elementen der Liste.
- Daher heißen solche Funktionen **Rekursions-Schemata** oder (Listen-)**Funktionale**.
- Listen-Funktionale sind unabhängig vom Typ der Listenelemente. (Diesen muss nur die Funktion  $f$  kennen :-)
- Funktionen, die gleich strukturierte Eingaben verschiedenen Typs verarbeiten können, heißen **polymorph**.

## 3.5 Polymorphe Funktionen

Das **Ocaml**-System inferiert folgende Typen für diese Funktionale:

```

map : ('a -> 'b) -> 'a list -> 'b list
fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
find_opt : ('a -> bool) -> 'a list -> 'a option

```

- $'a$  und  $'b$  sind **Typvariablen**. Sie können durch jeden Typ ersetzt (**instanziiert**) werden (aber an jedem Vorkommen durch den gleichen Typ).
- Durch partielle Anwendung auf eine Funktion können die Typvariablen instanziiert werden:

```

# Char.chr;;
val : int -> char = <fun>
# map Char.chr;;
- : int list -> char list = <fun>

# fold_left (+);;
val it : int -> int list -> int = <fun>

```

- Wenn man einem Funktional eine polymorphe Funktion als Argument gibt, ist das Ergebnis wieder polymorph:

```

# let cons_r xs x = x::xs;;
val cons_r : 'a list -> 'a -> 'a list = <fun>
# let rev l = fold_left cons_r [] l;;
val rev : 'a list -> 'a list = <fun>
# rev [1;2;3];;
- : int list = [3; 2; 1]
# rev [true;false;false];;
- : bool list = [false; false; true]

```

### Ein paar der einfachsten polymorphen Funktionen:

```

let compose f g x = f (g x)
let twice f x = f (f x)
let iter f g x = if g x then x else iter f g (f x);;

val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
val iter : ('a -> 'a) -> ('a -> bool) -> 'a -> 'a = <fun>

# compose neg neg;;
- : bool -> bool = <fun>
# compose neg neg true;;
- : bool = true;;
# compose Char.chr plus2 65;;
- : char = 'C'

```

## 3.6 Polymorphe Datentypen

Man sich auch selbst polymorphe Datentypen definieren:

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

- `tree` heißt **Typkonstruktor**, weil er aus einem anderen Typ (seinem Parameter `'a`) einen neuen Typ erzeugt.
- Auf der rechten Seite dürfen nur die Typvariablen vorkommen, die auf der linken Seite als Argument für den Typkonstruktor stehen.
- Die Anwendung der Konstruktoren auf Daten instanziiert die Typvariable(n):

```
# Leaf 1;;
- : int tree = Leaf 1
# Node (Leaf ('a',true), Leaf ('b',false));;
- : (char * bool) tree = Node (Leaf ('a', true),
                              Leaf ('b', false))
```

Funktionen auf polymorphen Datentypen sind typischerweise wieder polymorph ...

```
let rec size = function
  Leaf _      -> 1
| Node(t,t') -> size t + size t'

let rec flatten = function
  Leaf x      -> [x]
| Node(t,t') -> flatten t @ flatten t'

let flatten1 t = let rec doit = function
  (Leaf x, xs) -> x :: xs
| (Node(t,t'), xs) -> let xs = doit (t',xs)
                      in doit (t,xs)
  in doit (t,[])
...

...
val size : 'a tree -> int = <fun>
val flatten : 'a tree -> 'a list = <fun>
val flatten1 : 'a tree -> 'a list = <fun>

# let t = Node(Node(Leaf 1,Leaf 5),Leaf 3);;
val t : int tree = Node (Node (Leaf 1, Leaf 5), Leaf 3)
```

```

# size t;;
- : int = 3
# flatten t;;
val : int list = [1;5;3]
# flatten1 t;;
val : int list = [1;5;3]

```

## 3.7 Anwendung: Queues

### Gesucht:

Datenstruktur 'a queue, die die folgenden Operationen unterstützt:

```

enqueue : 'a -> 'a queue -> 'a queue
dequeue : 'a queue -> 'a option * 'a queue
is_empty : 'a queue -> bool
queue_of_list : 'a list -> 'a queue
list_of_queue : 'a queue -> 'a list

```

### 1. Idee:

- Repräsentiere die Schlange als eine Liste:

```
type 'a queue = 'a list
```

Die Funktionen `is_empty`, `queue_of_list`, `list_of_queue` sind dann trivial :-)

- Entnehmen heißt Zugreifen auf das erste Element der Liste:

```

let dequeue = function
  [] -> (None, [])
  | x::xs -> (Some x, xs)

```

- Einfügen bedeutet hinten anhängen:

```
let enqueue x xs = xs @ [x]
```

### Diskussion:

- Der Operator `@` konkateniert zwei Listen.

- Die Implementierung ist sehr einfach :-)
- Entnehmen ist sehr billig :-)
- Einfügen dagegen kostet so viele rekursive Aufrufe von @ wie die Schlange lang ist :-(
- Geht das nicht besser ??

## 2. Idee:

- Repräsentiere die Schlange als **zwei** Listen !!!

```

type 'a queue = Queue of 'a list * 'a list
let is_empty = function
    Queue ([],[]) -> true
    | _          -> false
let queue_of_list list = Queue (list,[])
let list_of_queue = function
    Queue (first,[]) -> first
    | Queue (first,last) ->
        first @ List.rev last

```

- Die zweite Liste repräsentiert das **Ende** der Liste und ist deshalb in **umgedrehter Anordnung ...**

## 2. Idee (Fortsetzung):

- Einfügen erfolgt deshalb in der zweiten Liste:

```

let enqueue x (Queue (first,last)) =
    Queue (first, x::last)

```

- Entnahme bezieht sich dagegen auf die erste Liste :-)  
Ist diese aber leer, wird auf die zweite zugegriffen ...

```

let dequeue = function
    Queue ([],last) -> (match List.rev last
        with [] -> (None, Queue ([],[]))
        | x::xs -> (Some x, Queue (xs,[])))
    | Queue (x::xs,last) -> (Some x, Queue (xs,last))

```

## Diskussion:

- Jetzt ist Einfügen billig :-)

- Entnehmen dagegen kann so teuer sein, wie die Anzahl der Elemente in der zweiten Liste :-)
- Gerechnet aber auf jede Einfügung, fallen nur **konstante** Zusatzkosten an !!!

⇒ **amortisierte Kostenanalyse**

### 3.8 Namenlose Funktionen

Wie wir gesehen haben, sind Funktionen **Daten**. Daten, z.B. [1;2;3] können verwendet werden, ohne ihnen einen Namen zu geben. Das geht auch für Funktionen:

```
# fun x y z -> x+y+z;;
- : int -> int -> int -> int = <fun>
```

- **fun** leitet eine **Abstraktion** ein.  
Der Name kommt aus dem **λ-Kalkül**.
- **->** hat die Funktion von **=** in Funktionsdefinitionen.
- **Rekursive** Funktionen können so nicht definiert werden, denn ohne Namen kann eine Funktion nicht in ihrem Rumpf vorkommen :-)



Alonzo Church, 1903–1995

- Um Pattern Matching zu benutzen, kann man **match ... with** für das entsprechende Argument einsetzen.
- Bei einem einzigen Argument bietet sich **function** an ...

```
# function None -> 0
      | Some x -> x*x+1;;
- : int option -> int = <fun>
```

Namenlose Funktionen werden verwendet, wenn sie nur **einmal** im Programm vorkommen. Oft sind sie **Argument für Funktionale**:

```
# map (fun x -> x*x) [1;2;3];;
- : int list = [1; 4; 9]
```

Oft werden sie auch benutzt, um eine Funktion **als Ergebnis** zurückzuliefern:

```
# let make_undefined () = fun x -> None;;
val make_undefined : unit -> 'a -> 'b option = <fun>
# let def_one (x,y) = fun x' -> if x=x' then Some y
                           else None;;
val def_one : 'a * 'b -> 'a -> 'b option = <fun>
```

## 4 Praktische Features in Ocaml

- Ausnahmen
- Imperative Konstrukte
  - modifizierbare Record-Komponenten
  - Referenzen
  - Sequenzen
  - Arrays und Strings
  - Ein- und Ausgabe

### 4.1 Ausnahmen (Exceptions)

Bei einem Laufzeit-Fehler, z.B. Division durch Null, erzeugt das **Ocaml**-System eine **exception** (Ausnahme):

```
# 1 / 0;;
Exception: Division_by_zero.
# List.tl (List.tl [1]);;
Exception: Failure "tl".
# Char.chr 300;;
Exception: Invalid_argument "Char.chr".
```

Hier werden die Ausnahmen `Division_by_zero`, `Failure "tl"` bzw. `Invalid_argument "Char.chr"` erzeugt.

Ein anderer Grund für eine Ausnahme ist ein **unvollständiger Match**:

```
# match 1+1 with 0 -> "null";;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
Exception: Match_failure ("", 2, -9).
```

In diesem Fall wird die Exception `Match_failure ("", 2, -9)` erzeugt **:-)**  
**Vordefinierte Konstruktoren für Exceptions:**

<code>Division_by_zero</code>	Division durch Null
<code>Invalid_argument of string</code>	falsche Benutzung
<code>Failure of string</code>	allgemeiner Fehler
<code>Match_failure of string * int * int</code>	unvollständiger Match
<code>Not_found</code>	nicht gefunden <b>:-)</b>
<code>Out_of_memory</code>	Speicher voll
<code>End_of_file</code>	Datei zu Ende
<code>Exit</code>	für die Benutzerin ...

Eine Exception ist ein **First Class Citizen**, d.h. ein Wert eines Datentyps `exn ...`

```
# Division_by_zero;;
- : exn = Division_by_zero
# Failure "Kompletter Quatsch!";;
- : exn = Failure "Kompletter Quatsch!"
```

Eigene Exceptions werden definiert, indem der Datentyp `exn` **erweitert** wird ...

```
# exception Hell;;
exception Hell
# Hell;;
- : exn = Hell

# Division_by_zero;;
- : exn = Division_by_zero
# Failure "Kompletter Quatsch!";;
- : exn = Failure "Kompletter Quatsch!"
```

Eigene Exceptions werden definiert, indem der Datentyp `exn` erweitert wird ...

```
# exception Hell of string;;
exception Hell of string
# Hell "damn!";;
- : exn = Hell "damn!"
```

## Ausnahmebehandlung:

Wie in `Java` können Exceptions ausgelöst und behandelt werden:

```
# let teile (n,m) = try Some (n / m)
                  with Division_by_zero -> None;;

# teile (10,3);;
- : int option = Some 3
# teile (10,0);;
- : int option = None
```

So kann man z.B. die member-Funktion neu definieren:

```
let rec member x l = try if x = List.hd l then true
                       else member x (List.tl l)
                    with Failure _ -> false

# member 2 [1;2;3];;
- : bool = true
# member 4 [1;2;3];;
- : bool = false
```

Das Schlüsselwort `with` leitet ein Pattern Matching auf dem Ausnahme-Datentyp `exn` ein:

```
try <exp>
with <pat1> -> <exp1> | ... | <patN> -> <expN>
```

⇒ Man kann mehrere Exceptions gleichzeitig abfangen :-)

Der Programmierer kann selbst Exceptions auslösen.

Das geht mit dem Schlüsselwort `raise ...`

```
# 1 + (2/0);;
Exception: Division_by_zero.
# 1 + raise Division_by_zero;;
Exception: Division_by_zero.
```

Eine Exception ist ein Fehlerwert, der jeden Ausdruck ersetzen kann.

Bei Behandlung wird sie durch einen anderen Ausdruck (vom richtigen Typ) ersetzt — oder durch eine andere Exception **;-)**

Exception Handling kann nach jedem beliebigen Teilausdruck, auch geschachtelt, stattfinden:

```
# let f (x,y) = x / (y-1);;
# let g (x,y) = try let n = try f (x,y)
                  with Division_by_zero ->
                    raise (Failure "Division by zero")
                  in string_of_int (n*n)
  with Failure str -> "Error: "^str;;

# g (6,1);;
- : string = "Error: Division by zero"
# g (6,3);;
- : string = "9"
```

## 4.2 Imperative Features im Ocaml

Gelegentlich möchte man Werte **destruktiv** verändern **;-)**

Dazu benötigen wir neue Konzepte ...

### Modifizierbare Record-Komponenten:

- Records fassen benannte Werte zusammen **;-)**
- Einzelne Komponenten können als **modifizierbar** deklariert werden ...

```
# type cell = {owner: string; mutable value: int};;
type cell = { owner : string; mutable value : int; }

...
# let x = {owner="me"; value=1};;
val x : cell = {owner = "me"; value = 1}
# x.value;;
- : int = 1
# x.value <- 2;;
- : unit = ()
```

```
# x.value;;
- : int = 2
```

- Modifizierbare Komponenten werden mit `mutable` gekennzeichnet.
- Die Initialisierung erfolgt wie bei einer normalen Komponente.
- Der Ausdruck `x.value <- 2` hat den Wert `()`, aber modifiziert die Komponente `value` als **Seiteneffekt !!!**

## Spezialfall: Referenzen

Eine Referenz `tau ref` auf einen Typ `tau` ist ein Record mit der einzigen Komponente `mutable contents: tau`:

```
# let ref_hallo = {contents = "Hallo!"};;
val ref_hallo : string ref = {contents = "Hallo!"}
# let ref_1 = ref 1;;
val ref_1 : int ref = {contents = 1}
```

Deshalb kann man auf den Wert einer Referenz mit Selektion zugreifen:

```
# ref_hallo.contents;;
- : string = "Hallo!"
# ref_1.contents;;
- : int = 1
```

Eine andere Möglichkeit ist der **Dereferenzierungs-Operator** `!`:

```
# !ref_hallo;;
- : string = "Hallo!"
```

Der Wert, auf den eine Referenz zeigt, kann mit `<-` oder mit `:=` verändert werden:

```
# ref_1.contents <- 2;;
- : unit = ()
# ref_1 := 3;;
- : unit = ()
```

## Gleichheit von Referenzen

Das Setzen von `ref_1` mittels `:=` erfolgt als **Seiteneffekt** und hat keinen Wert, d.h. ergibt `()`.

```
# (:=);;
- : 'a ref -> 'a -> unit = <fun>
```

Zwei Referenzen sind **gleich**, wenn sie auf **den gleichen** Wert zeigen:

```
# let x = ref 1
  let y = ref 1;;
val x : int ref = {contents = 1}
val y : int ref = {contents = 1}
# x = y;;
- : bool = true
```

## Sequenzen

Bei Updates kommt es nur auf den Seiteneffekt an **:-)**

Bei Seiteneffekten kommt es auf die Reihenfolge an **:-)**

Mehrere solche Aktionen kann man mit dem **Sequenz-Operator** **;** hintereinander ausführen:

```
# ref_1 := 1; ref_1 := !ref_1 + 1; ref_1;;
- : int ref = {contents = 2}
```

In **Ocaml** kann man sogar **Schleifen** programmieren ...

```
# let x = ref 0;;
val x : int ref = {contents = 1}
# while !x < 10 do x := !x+1 done;;
- : unit = ()
# x;;
- : int ref = contents = 10
```

Ein wichtiges Listenfunktional ist **List.iter**:

```
# let rec iter f = function
  [] -> ()
  | x::xs -> f x; iter f xs;;

val iter : ('a -> unit) -> 'a list -> unit = <fun>
```

## Arrays und Strings

Ein Array ist ein Record fester Länge, auf dessen modifizierbare Elemente mithilfe ihres Index in **konstanter Zeit** zugegriffen wird:

```
# let arr = [|1;3;5;7|];;
val arr : int array = [|1; 3; 5; 7|]
# arr.(2);;
- : int = 5
```

Zugriff außerhalb der Array-Grenzen löst eine Exception aus:

```
# arr.(4);;
Invalid_argument "index out of bounds"
```

Ein Array kann aus einer Liste oder als **Wertetabelle** einer Funktion erzeugt werden ...

```
# Array.of_list [1;2;3];;
- : int array = [|1; 2; 3|]
# Array.init 6 (fun x -> x*x);;
- : int array = [|0; 1; 4; 9; 16; 25|]
```

... und wieder zurück in eine Liste transformiert werden:

```
Array.fold_right (fun x xs -> x::xs)
                 [|0; 1; 4; 9; 16; 25|] [];;
- : int list = [0; 1; 4; 9; 16; 25]
```

Modifizierung der Array-Einträge funktioniert analog der Modifizierung von Record-Komponenten:

```
# arr.(1) <- 4;;
- : unit = ()
# arr;;
- : int array = [|1; 4; 5; 7|]
# arr.(5) <- 0;;
Exception: Invalid_argument "index out of bounds".
```

Ähnlich kann man auch Strings manipulieren :-)

```

# let str = "Hallo";;
val str : string = "Hallo"
# str.[2];;
- : char = 'l'
# str.[2] <- 'r';;
- : unit = ()
# str;;
- : string = "Harlo"

```

Für Arrays und Strings gibt es übrigens auch die Funktionen `length` und `concat` (und weitere :-).

### 4.3 Textuelle Ein- und Ausgabe

- Selbstverständlich kann man in **Ocaml** auf den Standard-Output schreiben:

```

# print_string "Hello World!\n";;
Hello World!
- : unit = ()

```

- Analog gibt es eine Funktion: `read_line : unit -> string ...`

```

# read_line ();;
Hello World!
- : "Hello World!"

```

Um aus einer **Datei zu lesen**, muss man diese zum Lesen **öffnen ...**

```

# let infile = open_in "test";;
val infile : in_channel = <abstr>
# input_line infile;;
- : "Die einzige Zeile der Datei ...";;
# input_line infile;;
Exception: End_of_file

```

Gibt es keine weitere Zeile, wird die Exception `End_of_file` geworfen :-)  
Benötigt man einen Kanal nicht mehr, sollte man ihn geregelt **schließen ...**

```

# close_in infile;;
- : unit = ()

```

## Weitere nützliche Funktionen:

```
stdin          : in_channel
input_char     : in_channel -> char
input_channel_length : in_channel -> int
input          : in_channel -> string -> int -> int -> int
```

- `input_channel_length` liefert die Gesamtlänge der Datei.
- `input_chan buf p n` liest aus einem Kanal `chan n` Zeichen und schreibt sie ab Position `p` in den String `buf :-)`

Die [Ausgabe in Dateien](#) erfolgt ganz analog ...

```
# let outfile = open_out "test";;
val outfile : out_channel = <abstr>
# output_string outfile "Hello ";;
- : unit = ()
# output_string outfile "World!\n";;
- : unit = ()
...
```

Die einzeln geschriebenen Wörter sind mit Sicherheit in der Datei erst zu finden, wenn der Kanal geregelt [geschlossen wurde](#) ...

```
# close_out outfile;;
- : unit = ()
```

## 5 Anwendung: Grundlegende Graph-Algorithmen

- Gerichtete Graphen
- Erreichbarkeit und DFS
- Topologische Sortierung

## 5.1 Gerichtete Graphen

### Beobachtung:

- Viele Probleme lassen sich mit Hilfe **gerichteter Graphen** repräsentieren ...

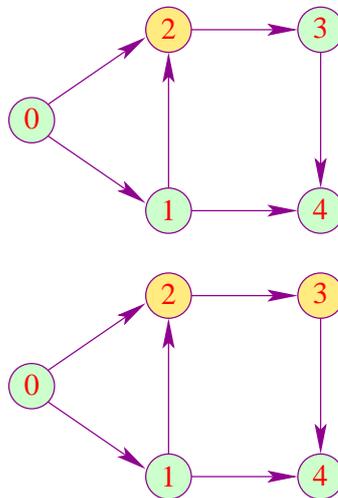
	Knoten	Kanten
Betrieblicher Prozess	Bearbeitungsschritt	Abfolge
Software	Zustand	Änderung
Systemanalyse	Komponente	Einfluss

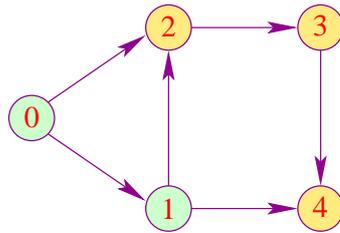
- Oft sind die Kanten mit Zusatz-Informationen **beschriftet** :-)

## 5.2 Erreichbarkeit und DFS

### Einfaches Problem:

- Gegeben ein Knoten.
- Welche anderen Knoten im Graphen kann man erreichen?





## Eine Datenstruktur für Graphen:

- Wir nehmen an, die Knoten sind fortlaufend mit 0, 1, ... durch-nummeriert.
- Für jeden Knoten gibt es eine Liste ausgehender Kanten.
- Damit wir schnellen Zugriff haben, speichern wir die Listen in einem Array ab ...

```
# type node      = int
# type 'a graph = ('a * node) list array

# let example = [| [(((),1); (((),2))]; [(((),2); (((),4))];
                 [(((),3))]; [(((),4))]; [] |] ;;
```

## Idee:

- Wir verwalten eine Datenstruktur `reachable : bool array`.
- Am Anfang haben alle Einträge `reachable.(x)` den Wert `false`.
- Besuchen wir einen Knoten `x`, der noch nicht erreicht wurde, setzen wir `reachable.(x) <- true` und besuchen alle Nachbarn von `x` :-)
- Kommen wir zu einem Knoten, der bereits besucht wurde, tun wir nix :-))

```
# let fold_li f a arr = let n = Array.length arr
                       in let rec doit i a = if i = -1 then a
                                               else doit (i-1) (f i a arr.(i))
                       in doit (n-1) a;;
```

```
# let reach edges x =
  let n = Array.length edges
  in let reachable = Array.make n false
  in let extract_result () = fold_li
    (fun i acc b -> if b then i::acc
                     else acc) [] reachable
  ...
```

## Kommentar:

- `fold_li : (int -> 'a -> 'b -> 'a) -> 'a -> 'b array -> 'a` erlaubt, über ein Array zu iterieren unter Berücksichtigung des Index wie der Werte für den Index :-)
- `Array.make : int -> 'a -> 'a array` legt ein neues initialisiertes Array an.
- Die entscheidende Berechnung erfolgt in der rekursiven Funktion `dfs ...`

```

...
in let rec dfs x = if not reachable.(x) then (
    reachable.(x) <- true;
    List.iter (fun (_,y) -> dfs y)
              edges.(x)
            )
in dfs x;
   extract_result ();;

```

- Die Technik heißt **Depth First Search** oder **Tiefensuche**, weil die Nachfolger eines Knotens **vor** den Nachbarn besucht werden :-)

### Kommentar (Forts.):

Der Test am Anfang von `dfs` liefert für jeden Knoten nur **einmal** `true`.

⇒⇒⇒ Jede Kante wird maximal **einmal** behandelt.

⇒⇒⇒ Der Gesamtaufwand ist proportional zu  $n + m$

//  $n$  == Anzahl der Knoten

//  $m$  == Anzahl der Kanten

## 5.3 Topologisches Sortieren

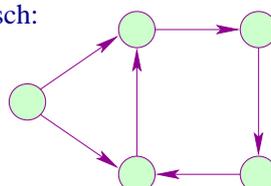
Ein **Pfad** oder **Weg** der Länge  $n$  in einem gerichteten Graphen ist eine Folge von Kanten:

$$(v_0, \_ , v_1)(v_1, \_ , v_2) \dots (v_{n-1}, \_ , v_n)$$

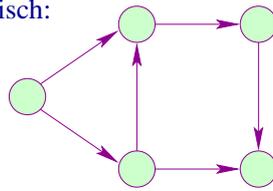
Ein Pfad der Länge  $n > 0$  von  $v$  nach  $v$  heißt **Kreis**.

Ein gerichteter Graph ohne Kreise heißt **azyklisch ...**

zyklisch:



azyklisch:

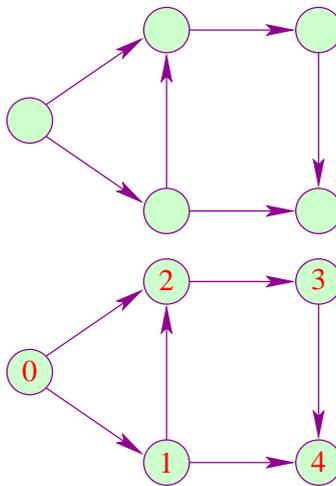


Aufgabe:

**Gegeben:** ein gerichteter Graph.

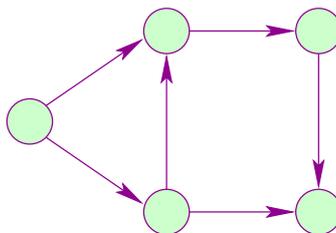
**Frage:** Ist der Graph azyklisch?

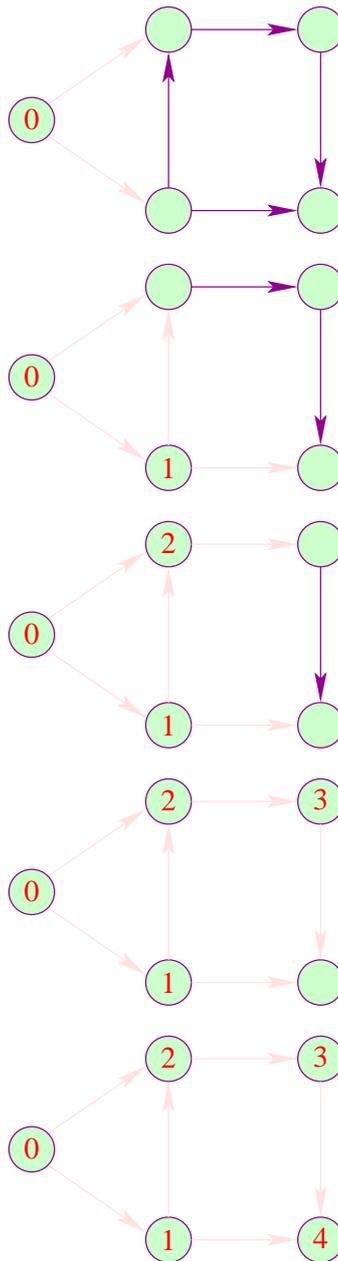
**Wenn ja:** Finde eine **lineare Anordnung** der Knoten!



Idee:

- Suche einen Knoten **ohne eingehende Kanten**.
- Besitzt jeder Knoten eingehende Kanten, muss der Graph einen Kreis enthalten ;-)
- Gibt es einen Knoten ohne eingehende Kanten, können wir diesen als kleinstes Element auswählen :-)
- Entferne den Knoten mit allen seinen **ausgehenden Kanten** !
- Löse das verbleibende Problem ...





### Implementierung:

- Für jeden Knoten  $x$  benutzen wir einen **Zähler**  $\text{count}(x)$ , der die Anzahl der eingehenden Kanten zählt.
- Haben wir einen Knoten, dessen Zähler 0 ist, nummerieren wir ihn :-)
- Dann durchlaufen wir seine ausgehenden Kanten.
- Für jeden Nachbarn dekrementieren wir den Zähler und fahren rekursiv fort.

⇒ wieder **Depth-First-Search**

```

# let top_sort edges =
    let n = Array.length edges
    in let count = Array.make n 0
    in let inc y = count.(y) <- count.(y) + 1
    in let dec y = count.(y) <- count.(y) - 1
    in for x=0 to n-1 do
        List.iter (fun (_,y) -> inc y) edges.(x)
        done ;
        let value = Array.make n (-1)
        ...

    in let next = ref 0
    in let number y = value.(y) <- !next;
        next := !next + 1
    in let rec dfs x = if count.(x) = 0 then (
        number x;
        List.iter (fun (_,y) ->
            dec y;
            dfs y)
            edges.(x))

    in for x=0 to n-1 do
        if value.(x) = -1 then dfs x
    done; value;;

```

## Diskussion:

- `for y=... to ... do ... done` ist eine vereinfachte `while`-Schleife :-)
- Für die Initialisierung des Arrays `count` iterieren wir einmal über alle Knoten und für jeden Knoten einmal über alle ausgehenden Kanten.

⇒ Aufwand: proportional  $n + m$

- Die Hauptschleife behandelt jeden Knoten maximal einmal.
- `dfs` wird maximal einmal für jeden Knoten und jede Kante aufgerufen und behandelt jede Kante maximal einmal.

⇒ Aufwand: proportional  $n + m$

⇒ Der Gesamtaufwand ist linear :-)

## 6 Formale Methoden für Ocaml

Frage:

Wie können wir uns versichern, dass ein Ocaml-Programm das macht, was es tun soll ???

Wir benötigen:

- eine formale Semantik;
- Techniken, um Aussagen über Programme zu beweisen ...

### 6.1 MiniOcaml

Um uns das Leben leicht zu machen, betrachten wir nur einen kleinen Ausschnitt aus Ocaml.

Wir erlauben ...

- nur die Basistypen `int`, `bool` sowie Tupel und Listen;
- rekursive Funktionsdefinitionen nur auf dem Top-Level :-)

Wir verbieten ...

- veränderbare Datenstrukturen;
- Ein- und Ausgabe;
- lokale rekursive Funktionen :-)

Dieses Fragment von Ocaml nennen wir MiniOcaml.

Ausdrücke in MiniOcaml lassen sich durch die folgende Grammatik beschreiben:

$$\begin{aligned} E ::= & \text{const} \mid \text{name} \mid \text{op}_1 E \mid E_1 \text{op}_2 E_2 \mid \\ & (E_1, \dots, E_k) \mid \text{let name} = E_1 \text{ in } E_0 \mid \\ & \text{match } E \text{ with } P_1 \rightarrow E_1 \mid \dots \mid P_k \rightarrow E_k \mid \\ & \text{fun name} \rightarrow E \mid E E_1 \end{aligned}$$
$$P ::= \text{const} \mid \text{name} \mid (P_1, \dots, P_k) \mid P_1 :: P_2$$

## Abkürzung:

$$\text{fun } x_1 \rightarrow \dots \text{fun } x_k \rightarrow e \equiv \text{fun } x_1 \dots x_k \rightarrow e$$

## Achtung:

- Die Menge der **erlaubten** Ausdrücke muss weiter eingeschränkt werden auf diejenigen, die **typkorrekt** sind, d.h. für die der **Ocaml**-Compiler einen Typ herleiten kann ...

(1, [true; false]) **typkorrekt**

(1 [true; false]) **nicht typkorrekt**

([1; true], false) **nicht typkorrekt**

- Wir verzichten auf `if ... then ... else ...`, da diese durch `match ... with true -> ... | false -> ...` simuliert werden können :-)
- Wir hätten auch auf `let ... in ...` verzichten können (wie?)

Ein **Programm** besteht dann aus einer Folge wechselseitig rekursiver globaler Definitionen von Variablen  $f_1, \dots, f_m$ :

```
let rec f1 = E1
    and f2 = E2
    ...
    and fm = Em
```

## 6.2 Eine Semantik für MiniOcaml

### Frage:

Zu welchem **Wert** wertet sich ein Ausdruck  $E$  aus ??

Ein **Wert** ist ein Ausdruck, der nicht weiter ausgerechnet werden kann :-)

Die Menge der Werte lässt sich ebenfalls mit einer Grammatik beschreiben:

$$V ::= \text{const} \mid \text{fun name}_1 \dots \text{name}_k \rightarrow E \mid (V_1, \dots, V_k) \mid [] \mid V_1 :: V_2$$

### Ein MiniOcaml-Programm ...

```
let rec comp = fun f g x -> f (g x)
    and map = fun f list -> match list
    with [] -> []
    | x::xs -> f x :: map f xs
```

## Ein MiniOcaml-Programm ...

```
let rec comp = fun f g x -> f (g x)
    and map = fun f list -> match list
        with [] -> []
             | x::xs -> f x :: map f xs
```

## Beispiele für Werte ...

```
1
(1, [true; false])
fun x -> 1 + 1
[fun x -> x+1; fun x -> x+2; fun x -> x+3]
```

## Idee:

- Wir definieren eine Relation:  $e \Rightarrow v$  zwischen Ausdrücken und ihren Werten  $= \Rightarrow$  **Big-Step operationelle Semantik**.
- Diese Relation definieren wir mit Hilfe von Axiomen und Regeln, die sich an der **Struktur** von  $e$  orientieren **:-)**
- Offenbar gilt stets:  $v \Rightarrow v$  für jeden Wert  $v$  **:-))**

## Tupel:

$$\frac{e_1 \Rightarrow v_1 \quad \dots \quad e_k \Rightarrow v_k}{(e_1, \dots, e_k) \Rightarrow (v_1, \dots, v_k)}$$

## Listen:

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 :: e_2 \Rightarrow v_1 :: v_2}$$

## Globale Definitionen:

$$\frac{f = e \quad e \Rightarrow v}{f \Rightarrow v}$$

## Lokale Definitionen:

$$\frac{e_1 \Rightarrow v_1 \quad e_0[v_1/x] \Rightarrow v_0}{\text{let } x = e_1 \text{ in } e_0 \Rightarrow v_0}$$

## Funktionsaufrufe:

$$\frac{e \Rightarrow \text{fun } x \rightarrow e_0 \quad e_1 \Rightarrow v_1 \quad e_0[v_1/x] \Rightarrow v_0}{e \ e_1 \Rightarrow v_0}$$

Durch mehrfache Anwendung der Regel für Funktionsaufrufe können wir zusätzlich eine Regel für Funktionen mit **mehreren** Argumenten ableiten:

$$\frac{e \Rightarrow \text{fun } x_1 \dots x_k \rightarrow e_0 \quad e_1 \Rightarrow v_1 \dots e_k \Rightarrow v_k \quad e_0[v_1/x_1, \dots, v_k/x_k] \Rightarrow v_0}{e \ e_1 \dots e_k \Rightarrow v_0}$$

Diese abgeleitete Regel macht Beweise etwas weniger umständlich :-)

## Pattern Matching:

$$\frac{e_0 \Rightarrow v' \equiv p_i[v_1/x_1, \dots, v_k/x_k] \quad e_i[v_1/x_1, \dots, v_k/x_k] \Rightarrow v}{\text{match } e_0 \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_m \rightarrow e_m \Rightarrow v}$$

— sofern  $v'$  auf keines der Muster  $p_1, \dots, p_{i-1}$  passt :-)

## Eingebaute Operatoren:

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad v_1 \text{ op } v_2 = v}{e_1 \text{ op } e_2 \Rightarrow v}$$

Die unären Operatoren behandeln wir analog :-)

## Beispiel:

```
let f = fun x -> x+1
let s = fun y -> y*y
```

$$\frac{\frac{f = \text{fun } x \rightarrow x+1 \quad 16+1 = 17}{f \Rightarrow \text{fun } x \rightarrow x+1 \quad 16+1 \Rightarrow 17} \quad \frac{s = \text{fun } y \rightarrow y*y \quad 2*2 = 4}{s \Rightarrow \text{fun } y \rightarrow y*y \quad 2*2 \Rightarrow 4}}{f \ 16 \Rightarrow 17 \quad s \ 2 \Rightarrow 4 \quad 17+4 = 21}$$

$$f \ 16 + s \ 2 \Rightarrow 21$$

// Benutzungen von  $v \Rightarrow v$  haben wir i.a. weggelassen :-)

## Beispiel:

```
let rec app = fun x y -> match x
  with [] -> y
       | h::t -> h :: app t y
```

**Behauptung:**  $\text{app } (1::[]) (2::[]) \Rightarrow 1::2::[]$

**Beweis:**

$$\frac{\frac{\frac{\text{app} = \text{fun } x \ y \rightarrow \dots}{\text{app} \Rightarrow \text{fun } x \ y \rightarrow \dots} \quad \frac{\frac{2::[] \Rightarrow 2::[]}{\text{match } [] \dots \Rightarrow 2::[]}}{\text{app } [] (2::[]) \Rightarrow 2::[]}}{1 :: \text{app } [] (2::[]) \Rightarrow 1::2::[]}}{\frac{\frac{\text{app} = \text{fun } x \ y \rightarrow \dots}{\text{app} \Rightarrow \text{fun } x \ y \rightarrow \dots} \quad \frac{\text{match } 1::[] \dots \Rightarrow 1::2::[]}{\text{app } (1::[]) (2::[]) \Rightarrow 1::2::[]}}{\text{app } (1::[]) (2::[]) \Rightarrow 1::2::[]}}$$

// Benutzungen von  $v \Rightarrow v$  haben wir i.a. weggelassen :-)

## Diskussion:

- Die **Big-Step operationelle Semantik** ist nicht sehr gut geeignet, um Schritt für Schritt nahezu vollziehen, was ein **MiniOcaml**-Programm macht :-)
- Wir können damit aber sehr gut nachweisen, dass die Auswertung eine Funktion für bestimmte Argumentwerte stets terminiert:

Dazu muss nur nachgewiesen werden, dass es jeweils einen Wert gibt, zu dem die entsprechende Funktionsanwendung ausgewertet werden kann ...

## Beispiel-Behauptung:

$\text{app } l_1 l_2$  terminiert für alle Listen-Werte  $l_1, l_2$ .

## Beweis:

Induktion nach der Länge  $n$  der Liste  $l_1$ .

**$n = 0$  :** D.h.  $l_1 = []$ . Dann gilt:

$$\frac{\frac{\text{app} = \text{fun } x \ y \rightarrow \dots}{\text{app} \Rightarrow \text{fun } x \ y \rightarrow \dots} \quad \text{match } [] \text{ with } [] \rightarrow l_2 \mid \dots \Rightarrow l_2}{\text{app } [] l_2 \Rightarrow l_2}$$

:-)

$n > 0$ : D.h.  $l_1 = h :: t$ .

Insbesondere nehmen wir an, dass die Behauptung bereits für alle kürzeren Listen gilt. Deshalb haben wir:

$$\text{app } t \ l_2 \Rightarrow l$$

für ein geeignetes  $l$ . Wir schließen:

$$\frac{\frac{\text{app } = \text{ fun } x \ y \ -> \ \dots}{\text{app } \Rightarrow \text{ fun } x \ y \ -> \ \dots} \quad \frac{\frac{\text{app } t \ l_2 \Rightarrow l}{h :: \text{app } t \ l_2 \Rightarrow h :: l}}{\text{match } h :: t \ \text{with } \dots \Rightarrow h :: l}}{\text{app } (h :: t) \ l_2 \Rightarrow h :: l}$$

:-)

### Diskussion (Forts.):

- Wir können mit der Bigstep-Semantik auch überprüfen, dass **optimierende Transformationen** korrekt sind :-)
- Schließlich können wir sie benutzen, um die Korrektheit von Aussagen über funktionale Programme zu beweisen !
- Die Big-Step operationelle Semantik legt dabei nahe, Ausdrücke als **Beschreibungen** von Werten aufzufassen.
- Ausdrücke, die sich zu den **gleichen** Werten auswerten, sollten deshalb austauschbar sein ...

### Achtung:

- **Gleichheit** zwischen Werten kann in **MiniOcaml** nur getestet werden, wenn diese keine Funktionen enthalten !!
- Solche Werte nennen wir **vergleichbar**. Sie haben die Form:

$$C ::= \text{const} \mid (C_1, \dots, C_k) \mid [] \mid C_1 :: C_2$$

- Offenbar ist ein **MiniOcaml**-Wert genau dann vergleichbar, wenn sein **Typ** funktionsfrei, d.h. einer der folgenden Typen ist:

$$c ::= \text{bool} \mid \text{int} \mid \text{unit} \mid c_1 * \dots * c_k \mid c \ \text{list}$$

:-)

Für Ausdrücke  $e_1, e_2, e$  mit funktionsfreien Typen können wir **Schlussregeln** angeben ...

## Substitutionslemma:

$$\frac{e_1 \Rightarrow v' \quad e_2 \Rightarrow v' \quad e[e_1/x] \Rightarrow v}{e[e_2/x] \Rightarrow v}$$

### Beachte:

$$e_1 = e_2 \Rightarrow \text{true} \quad \Leftrightarrow \quad e_1 \Rightarrow v' \wedge e_2 \Rightarrow v' \quad \text{für ein } v' \quad \text{:})$$

### Wir folgern:

$$\frac{e_1 = e_2 \Rightarrow \text{true} \quad e[e_1/x] \text{ terminiert}}{e[e_1/x] = e[e_2/x] \Rightarrow \text{true}}$$

### Diskussion:

- Das Lemma besagt damit, dass wir in **jedem Kontext** alle Vorkommen eines Ausdrucks  $e_1$  durch einen Ausdruck  $e_2$  ersetzen können, sofern  $e_1$  und  $e_2$  die selben Werte liefern :-)
- Das Lemma lässt sich mit Induktion über die Tiefe der benötigten Herleitungen zeigen (was wir uns sparen :-))
- Der Austausch von als gleich erwiesenen Ausdrücken ist die Grundlage unserer Methode zum Nachweis der **Äquivalenz** von Ausdrücken ...

## 6.3 Beweise für MiniOcaml-Programme

### Beispiel 1:

```
let rec app = fun x -> fun y -> match x
  with [] -> y
       | x::xs -> x :: app xs y
```

Wir wollen nachweisen:

- (1)  $\text{app } x \ [] = x$  für alle Listen  $x$ .
- (2)  $\text{app } x \ (\text{app } y \ z) = \text{app } (\text{app } x \ y) \ z$   
für alle Listen  $x, y, z$ .

**Idee:** Induktion nach der Länge  $n$  von  $x$

$n = 0$ : Dann gilt:  $x = []$

Wir schließen:

$$\begin{aligned} \text{app } x \ [] &= \text{app } [] \ [] \\ &= [] \\ &= x \quad \text{:)} \end{aligned}$$

$n > 0$ : Dann gilt:  $x = h::t$  wobei  $t$  Länge  $n - 1$  hat.

Wir schließen:

$$\begin{aligned} \text{app } x \ [] &= \text{app } (h::t) \ [] \\ &= h :: \text{app } t \ [] \\ &= h :: t \quad \text{nach Induktionsannahme} \\ &= x \quad \text{:))} \end{aligned}$$

Analog gehen wir für die Aussage (2) vor ...

$n = 0$ : Dann gilt:  $x = []$

Wir schließen:

$$\begin{aligned} \text{app } x \ (\text{app } y \ z) &= \text{app } [] \ (\text{app } y \ z) \\ &= \text{app } y \ z \\ &= \text{app } (\text{app } [] \ y) \ z \\ &= \text{app } (\text{app } x \ y) \ z \quad \text{:)} \end{aligned}$$

$n > 0$ : Dann gilt:  $x = h::t$  wobei  $t$  Länge  $n - 1$  hat.

Wir schließen:

$$\begin{aligned} \text{app } x \ (\text{app } y \ z) &= \text{app } (h::t) \ (\text{app } y \ z) \\ &= h :: \text{app } t \ (\text{app } y \ z) \\ &= h :: \text{app } (\text{app } t \ y) \ z \quad \text{nach Induktionsannahme} \\ &= \text{app } (h :: \text{app } t \ y) \ z \\ &= \text{app } (\text{app } (h::t) \ y) \ z \\ &= \text{app } (\text{app } x \ y) \ z \quad \text{:))} \end{aligned}$$

Diskussion:

- Bei den Gleichheitsumformungen haben wir einfache Zwischenschritte weggelassen :-)
- Eine Aussage:  $\text{exp1} = \text{exp2}$  steht für:  $\text{exp1} = \text{exp2} \Rightarrow \text{true}$  und schließt damit die Terminierung der Auswertungen von  $\text{exp1}, \text{exp2}$  ein.

- Zur Korrektheit unserer Induktionsbeweise benötigen wir, dass sämtliche vorkommenden Funktionsaufrufe **terminieren**.
- Im Beispiel reicht es zu zeigen, dass für alle  $x, y$  ein  $v$  existiert mit:

$$\text{app } x \ y \Rightarrow v$$

... das haben wir aber bereits bewiesen, natürlich ebenfalls mit **Induktion** ;-)

### Beispiel 2:

```
let rec rev = fun x -> match x
  with [] -> []
       | x::xs -> app (rev xs) [x]
let rec rev1 = fun x -> fun y -> match x
  with [] -> y
       | x::xs -> rev1 xs (x:y)
```

### Behauptung:

$\text{rev } x = \text{rev1 } x \ []$  für alle Listen  $x$ .

### Allgemeiner:

$\text{app } (\text{rev } x) \ y = \text{rev1 } x \ y$  für alle Listen  $x, y$ .

**Beweis:** Induktion nach der Länge  $n$  von  $x$

$n = 0$ : Dann gilt:  $x = []$

Wir schließen:

$$\begin{aligned} \text{app } (\text{rev } x) \ y &= \text{app } (\text{rev } []) \ y \\ &= \text{app } [] \ y \\ &= y \\ &= \text{rev1 } [] \ y \\ &= \text{rev1 } x \ y \quad \text{:)} \end{aligned}$$

$n > 0$ : Dann gilt:  $x = h::t$  wobei  $t$  Länge  $n - 1$  hat.

Wir schließen:

```
app (rev x) y = app (rev (h::t)) y
              = app (app (rev t) [h]) y
              = app (rev t) (app [h] y) wegen Beispiel 1
              = app (rev t) (h::y)
              = revl t (h::y) nach Induktionsvoraussetzung
              = revl (h::t) y
              = revl x y :-))
```

### Diskussion:

- Wieder haben wir implizit die Terminierung der Funktionsaufrufe von `app`, `rev` und `revl` angenommen :-)
- Deren Terminierung können wir jedoch leicht mittels Induktion nach der Tiefe des ersten Arguments nachweisen.
- Die Behauptung:  
folgt aus: 
$$\begin{aligned} \text{rev } x &= \text{revl } x [] \\ \text{app } (\text{rev } x) \ y &= \text{revl } x \ y \end{aligned}$$

indem wir:  $y = []$  setzen und Aussage (1) aus **Beispiel 1** benutzen :-)

### Beispiel 3:

```
let rec sorted = fun x -> match x
  with x1::x2::xs -> (match x1 <= x2
    with true -> sorted (x2::xs)
      | false -> false)
  | _ -> true

and merge = fun x -> fun y -> match (x,y)
  with ([],y) -> y
  | (x,[]) -> x
  | (x1::xs,y1::ys) -> (match x1 <= y1
    with true -> x1 :: merge xs y
      | false -> y1 :: merge x ys
```

## Behauptung:

$\text{sorted } x \wedge \text{sorted } y \rightarrow \text{sorted } (\text{merge } x \ y)$   
für alle Listen  $x, y$ .

**Beweis:** Induktion über die **Summe**  $n$  der Längen von  $x, y$  :-)

Gelte  $\text{sorted } x \wedge \text{sorted } y$ .

**$n = 0$ :** Dann gilt:  $x = [] = y$

Wir schließen:

$\text{sorted } (\text{merge } x \ y) = \text{sorted } (\text{merge } [] \ [])$   
 $= \text{sorted } []$   
 $= \text{true} \quad \text{:})$

**$n > 0$ :**

**Fall 1:**  $x = []$ .

Wir schließen:

$\text{sorted } (\text{merge } x \ y) = \text{sorted } (\text{merge } [] \ y)$   
 $= \text{sorted } y$   
 $= \text{true} \quad \text{:})$

**Fall 2:**  $y = []$  analog :-)

**Fall 3:**  $x = x1 :: xs \wedge y = y1 :: ys \wedge x1 \leq y1$ .

Wir schließen:

$\text{sorted } (\text{merge } x \ y) = \text{sorted } (\text{merge } (x1 :: xs) (y1 :: ys))$   
 $= \text{sorted } (x1 :: \text{merge } xs \ y)$   
 $= \dots$

**Fall 3.1:**  $xs = []$

Wir schließen:

$\dots = \text{sorted } (x1 :: \text{merge } [] \ y)$   
 $= \text{sorted } (x1 :: y)$   
 $= \text{sorted } y$   
 $= \text{true} \quad \text{:})$

**Fall 3.2:**  $xs = x2 :: xs' \wedge x2 \leq y1$ .

Insbesondere gilt:  $x1 \leq x2 \wedge \text{sorted } xs$ .

Wir schließen:

```
... = sorted (x1 :: merge (x2::xs') y)
    = sorted (x1 :: x2 :: merge xs' y)
    = sorted (x2 :: merge xs' y)
    = sorted (merge xs y)
    = true nach Induktionsannahme :-)
```

**Fall 3.3:**  $xs = x2 :: xs' \wedge x2 > y1$ .

Insbesondere gilt:  $x1 \leq y1 < x2 \wedge \text{sorted } xs$ .

Wir schließen:

```
... = sorted (x1 :: merge (x2::xs') (y1::ys))
    = sorted (x1 :: y1 :: merge xs ys)
    = sorted (y1 :: merge xs ys)
    = sorted (merge xs y)
    = true nach Induktionsannahme :-)
```

**Fall 4:**  $x = x1 :: xs \wedge y = y1 :: ys \wedge x1 > y1$ .

Wir schließen:

```
sorted (merge x y) = sorted (merge (x1::xs) (y1::ys))
                  = sorted (y1 :: merge x ys)
                  = ...
```

**Fall 4.1:**  $ys = []$

Wir schließen:

```
... = sorted (y1 :: merge x [])
    = sorted (y1 :: x)
    = sorted x
    = true :-)
```

**Fall 4.2:**  $ys = y2 :: ys' \wedge x1 > y2$ .

Insbesondere gilt:  $y1 \leq y2 \wedge \text{sorted } ys$ .

Wir schließen:

```

... = sorted (y1 :: merge x (y2::ys'))
    = sorted (y1 :: y2 :: merge x ys')
    = sorted (y2 :: merge x ys')
    = sorted (merge x ys)
    = true nach Induktionsannahme :-)

```

**Fall 4.3:**  $ys = y2::ys' \wedge x1 \leq y2$ .

Insbesondere gilt:  $y1 < x1 \leq y2 \wedge \text{sorted } ys$ .

Wir schließen:

```

... = sorted (y1 :: merge (x1::xs) (y2::ys'))
    = sorted (y1 :: x1 :: merge xs ys)
    = sorted (x1 :: merge xs ys)
    = sorted (merge x ys)
    = true nach Induktionsannahme :-))

```

### Diskussion:

- Wieder steht der Beweis unter dem Vorbehalt, dass alle Aufrufe der Funktionen `sorted` und `merge` terminieren :-)
- Als zusätzliche Technik benötigten wir **Fallunterscheidungen** über die verschiedenen Möglichkeiten für Argumente in den Aufrufen :-)
- Die Fallunterscheidungen machten den Beweis länglich :-(  
// Der Fall  $n = 0$  ist tatsächlich überflüssig,  
// da er in den Fällen 1 und 2 enthalten ist :-)

### Ausblick:

- In Programmoptimierungen möchten wir gelegentlich **Funktionen** austauschen, z.B.

$$\text{comp (map f) (map g) = map (comp f g)}$$

- Offenbar stehen rechts und links des **Gleichheitszeichens** Funktionen, deren Gleichheit **Ocaml** nicht überprüfen kann

⇒⇒⇒

Die Logik benötigt einen **stärkeren** Gleichheitsbegriff :-)

- Wir **erweitern** die **Ocaml**-Gleichheit  $=$  so dass gilt:

□  $x = y$  falls  $x$  und  $y$  nicht terminieren :-)

□  $f = g$  falls  $f\ x = g\ x$  für alle  $x$   
 $\implies$  extensionale Gleichheit

Wir haben:

- Seien der Typ von  $e_1, e_2$  **funktionsfrei**. Dann gilt:

$$\frac{e_1 = e_2 \quad e_1 \text{ terminiert}}{e_1 = e_2 \Rightarrow \text{true}}$$

- Außerdem gibt es eine stark vereinfachte **Substitutionsregel**:

$$\frac{e_1 = e_2}{e[e_1/x] = e[e_2/x]}$$

// Die Annahmen über Terminierung entfallen :-)

$\implies$  Die Beweistechnik vereinfacht sich :-))

## 7 Das Modulsystem von OCAML

- Strukturen
- Signaturen
- Information Hiding
- Funktoren
- Getrennte Übersetzung

### 7.1 Module oder Strukturen

Zur Strukturierung großer Programmsysteme bietet **Ocaml Module** oder **Strukturen** an:

```
module Pairs =
  struct
    type 'a pair = 'a * 'a
    let pair (a,b) = (a,b)
    let first (a,b) = a
    let second (a,b) = b
  end
```

Auf diese Eingabe antwortet der Compiler mit dem Typ der Struktur, einer **Signatur**:

```
module Pairs :  
  sig  
    type 'a pair = 'a * 'a  
    val pair : 'a * 'b -> 'a * 'b  
    val first : 'a * 'b -> 'a  
    val second : 'a * 'b -> 'b  
  end
```

Die Definitionen innerhalb der Struktur sind außerhalb **nicht sichtbar**:

```
# first;;  
Unbound value first
```

## Zugriff auf Komponenten einer Struktur:

Über den Namen greift man auf die Komponenten einer Struktur zu:

```
# Pairs.first;;  
- : 'a * 'b -> 'a = <fun>
```

So kann man z.B. **mehrere Funktionen** gleichen Namens definieren:

```
# module Triples = struct  
  type 'a triple = Triple of 'a * 'a * 'a  
  let first (Triple (a,_,_)) = a  
  let second (Triple (_,b,_)) = b  
  let third (Triple (_,_,c)) = c  
end;;  
...  
...  
module Triples :  
sig  
  type 'a triple = Triple of 'a * 'a * 'a  
  val first : 'a triple -> 'a  
  val second : 'a triple -> 'a  
  val third : 'a triple -> 'a  
end  
# Triples.first;;  
- : 'a Triples.triple -> 'a = <fun>
```

... oder **mehrere Implementierungen** der gleichen Funktion:

```
# module Pairs2 =  
  struct  
    type 'a pair = bool -> 'a  
    let pair (a,b) = fun x -> if x then a else b  
    let first ab = ab true  
    let second ab = ab false  
  end;;
```

## Öffnen von Strukturen

Um nicht immer den Strukturnamen verwenden zu müssen, kann man **alle** Definitionen einer Struktur auf einmal sichtbar machen:

```
# open Pairs2;;
# pair;;
- : 'a * 'a -> bool -> 'a = <fun>
# pair (4,3) true;;
- : int = 4
```

Sollen die Definitionen des anderen Moduls **Bestandteil** des gegenwärtigen Moduls sein, dann macht man sie mit **include** verfügbar ...

```
# module A = struct let x = 1 end;;
module A : sig val x : int end
# module B = struct
  open A
  let y = 2
end;;
module B : sig val y : int end
# module C = struct
  include A
  include B
end;;
module C : sig val x : int val y : int end
```

## Geschachtelte Strukturen

Strukturen können selbst wieder Strukturen enthalten:

```
module Quads = struct
  module Pairs = struct
    type 'a pair = 'a * 'a
    let pair (a,b) = (a,b)
    let first (a,_) = a
    let second (_,b) = b
  end
  type 'a quad = 'a Pairs.pair Pairs.pair
  let quad (a,b,c,d) =
    Pairs.pair (Pairs.pair (a,b), Pairs.pair (c,d))
  ...
end
```

```

...
let first q = Pairs.first (Pairs.first q)
let second q = Pairs.second (Pairs.first q)
let third q = Pairs.first (Pairs.second q)
let fourth q = Pairs.second (Pairs.second q)
end

# Quads.quad (1,2,3,4);;
- : (int * int) * (int * int) = ((1,2),(3,4))
# Quads.Pairs.first;;
- : 'a * 'b -> 'a = <fun>

```

## 7.2 Modul-Typen oder Signaturen

Mithilfe von **Signaturen** kann man einschränken, was eine Struktur nach außen exportiert.

### Beispiel:

```

module Count =
  struct
    let count = ref 0
    let setCounter n = count := n
    let getCounter () = !count
    let incCounter () = count := !count+1
  end

```

Der Zähler ist nach außen sichtbar, zugreifbar und veränderbar:

```

# !Count.count;;
- : int = 0
# Count.count := 42;;
- : unit = ()

```

Will man, daß nur die Funktionen der Struktur auf ihn zugreifen können, benutzt man einen Modul-Typ oder **Signatur**:

```

module type Count =
  sig
    val setCounter : int -> unit
    val incCounter : unit -> unit
    val getCounter : unit -> int
  end

```

Die Signatur enthält den Zähler selbst nicht :-)

Mit dieser Signatur wird die Schnittstelle der Struktur eingeschränkt:

```
# module SafeCount : Count = Count;;
module SafeCount : Count
# SafeCount.count;;
Unbound value Safecount.count
```

Die Signatur bestimmt, welche Definitionen exportiert werden. Das geht auch direkt bei der Definition:

```
module Count1 : Count =
  struct
    val count = ref 0
    fun setCounter n = count := n
    fun getCounter () = !count
    fun incCounter () = count := !count+1
  end

# !Count1.count;;
Unbound value Count1.count
```

## Signaturen und Typen

Die in der Signatur angegebenen Typen müssen **Instanzen** der für die exportierten Definitionen inferierten Typen sein :-)

Dadurch werden deren Typen spezialisiert:

```
module type A1 = sig
  val f : 'a -> 'b -> 'b
end
module type A2 = sig
  val f : int -> char -> int
end
module A = struct
  let f x y = x
end

# module A1 : A1 = A;;
Signature mismatch:
Modules do not match: sig val f : 'a -> 'b -> 'a end
```

is not included in A1

Values do not match:

```
val f : 'a -> 'b -> 'a
is not included in
val f : 'a -> 'b -> 'b
# module A2 : A2 = A;;
module A2 : A2
# A2.f;;
- : int -> char -> int = <fun>
```

## 7.3 Information Hiding

Aus Gründen der Modularität möchte man oft verhindern, dass die Struktur exportierter Typen einer Struktur von außen sichtbar ist.

### Beispiel:

```
module ListQueue = struct
  type 'a queue = 'a list
  let empty_queue () = []
  let is_empty = function
    [] -> true | _ -> false
  let enqueue xs y = xs @ [y]
  let dequeue (x::xs) = (x,xs)
end
```

Mit einer Signatur kann man die Implementierung einer Queue verstecken:

```
module type Queue = sig
  type 'a queue
  val empty_queue : unit -> 'a queue
  val is_empty : 'a queue -> bool
  val enqueue : 'a queue -> 'a -> 'a queue
  val dequeue : 'a queue -> 'a * 'a queue
end

# module Queue : Queue = ListQueue;;
module Queue : Queue
# open Queue;;
# is_empty [];;
This expression has type 'a list but is here used with type
'b queue = 'b Queue.queue
```



Das Einschränken per Signatur genügt, um die **wahre Natur** des Typs `queue` zu verschleiern :-)

Soll der Datentyp mit seinen Konstruktoren dagegen exportiert werden, **wiederholen** wir seine Definition in der Signatur:

```
module type Queue =
sig
  type 'a queue = Queue of ('a list * 'a list)
  val empty_queue : unit -> 'a queue
  val is_empty : 'a queue -> bool
  val enqueue : 'a -> 'a queue -> 'a queue
  val dequeue : 'a queue -> 'a option * 'a queue
end
```

## 7.4 Funktoren

Da in **Ocaml** fast alles höherer Ordnung ist, wundert es nicht, dass es auch Strukturen höherer Ordnung gibt: die **Funktoren**.

- Ein Funktor bekommt als Parameter eine Folge von Strukturen;
- der Rumpf eines Funktors ist eine Struktur, in der die Argumente des Funktors verwendet werden können;
- das Ergebnis ist eine neue Struktur, die abhängig von den Parametern definiert ist.

Wir legen zunächst per Signatur die Eingabe und Ausgabe des Funktors fest:

```
module type Enum = sig
  type enum
  val null : enum
  val incr : enum -> enum
  val int_of_enum : enum -> int
end
module type Counter = sig
  type counter
  val incCounter : unit -> unit
  val getCounter : unit -> counter
```

```

    val int_of_counter : counter -> int
  end
  ...

  ...
# module GenCounter (Enum : Enum) : Counter =
  struct
    open Enum
    type counter = enum
    let count = ref null
    let incCounter() = count := incr(!count)
    let getCounter() = !count
    let int_of_counter x = int_of_enum x
  end;;
module GenCounter : functor (Enum : Enum) -> Counter

```

Jetzt können wir den Funktor auf eine Struktur **anwenden** und erhalten eine neue Struktur ...

```

module PlusEnum = struct
  type enum = int
  let null = 0
  let incr x = x+1
  let int_of_enum x = x
end

# module PlusCounter = GenCounter (PlusEnum);;
module PlusCounter : Counter

# PlusCounter.incCounter ();
PlusCounter.incCounter ();
PlusCounter.int_of_counter (PlusCounter.getCounter ());;
- : int = 2

```

Eine erneute Anwendung des Funktors erzeugt eine **neue Struktur**:

```

module MalEnum = struct
  type enum = int
  let null = 1
  let incr x = 2*x
  let int_of_enum x = x
end

# module MalCounter = GenCounter (MalEnum);;
module MalCounter : Counter

```

```

# MalCounter.incCounter();
  MalCounter.incCounter();
  MalCounter.int_of_counter (MalCounter.getCounter());;
- : int = 4

```

Mit einem Funktor kann man sich auch **mehrere Instanzen** des gleichen Moduls erzeugen:

```

# module CountApples = GenCounter (PlusEnum);;
module CountApples : Counter
# module CountPears = GenCounter (PlusEnum);;
module CountPears : Counter
...

...
# CountApples.incCounter();;
- : unit = ()
# CountApples.incCounter();;
- : unit = ()
# CountPears.incCounter ();;
- : unit = ()
# CountApples.int_of_counter (CountApples.getCounter());;
- : int = 2
# CountPears.int_of_counter (CountPears.getCounter());;
- : int = 1

```

## 7.5 Getrennte Übersetzung

- Eigentlich möchte man **Ocaml**-Programme nicht immer in der interaktiven Umgebung starten **:-)**
- Dazu gibt es u.a. den Compiler **ocamlc ...**  

```
> ocamlc Test.ml
```

interpretiert den Inhalt der Datei **Test.ml** als Folge von Definitionen einer Struktur **Test**.
- Als Ergebnis der Übersetzung liefert **ocamlc** die Dateien:

<b>Test.cmo</b>	Bytecode für die Struktur
<b>Test.cmi</b>	Bytecode für das Interface
<b>a.out</b>	lauffähiges Programm

- Gibt es eine Datei `Test.mli` wird diese als Definition der Signatur für `Test` aufgefasst. Dann rufen wir auf:  

```
> ocamlc Test.mli Test.ml
```
- Benutzt eine Struktur `A` eine Struktur `B`, dann sollte diese mit übersetzt werden:  

```
> ocamlc B.mli B.ml A.mli A.ml
```
- Möchte man auf die Neuübersetzung von `B` verzichten, kann man `ocamlc` auch die vor-übersetzte Datei mitgeben:  

```
> ocamlc B.cmo A.mli A.ml
```
- Zur praktischen Verwaltung von benötigten Neuübersetzungen nach Änderungen von Dateien bietet `Linux` das Kommando `make` an. Das Protokoll der auszuführenden Aktionen steht dann in einer Datei `Makefile` :-)