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 main () {
   x = read();
   m0 = 1; m1 = 1;
   if (x > 1) f();
   write (m1);
}
```

```
void f() {
   x = x-1;
   if (x>1) f();
   t = m1;
   m1 = m0+m1;
   m0 = t;
}
```

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 `m0` und `m1` jeweils die Werte $\mathsf{fib}(i-1)$ und $\mathsf{fib}(i)$ ...

Problem:

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

Im Beispiel:

$$\{A\}\ \texttt{f();}\ \{B\} \qquad \text{wobei}$$

$$\begin{aligned} A &\equiv x = l \wedge x > 1 \wedge m_0 = m_1 = 1 \\ B &\equiv l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1} \end{aligned}$$

Allgemeines Vorgehen:

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

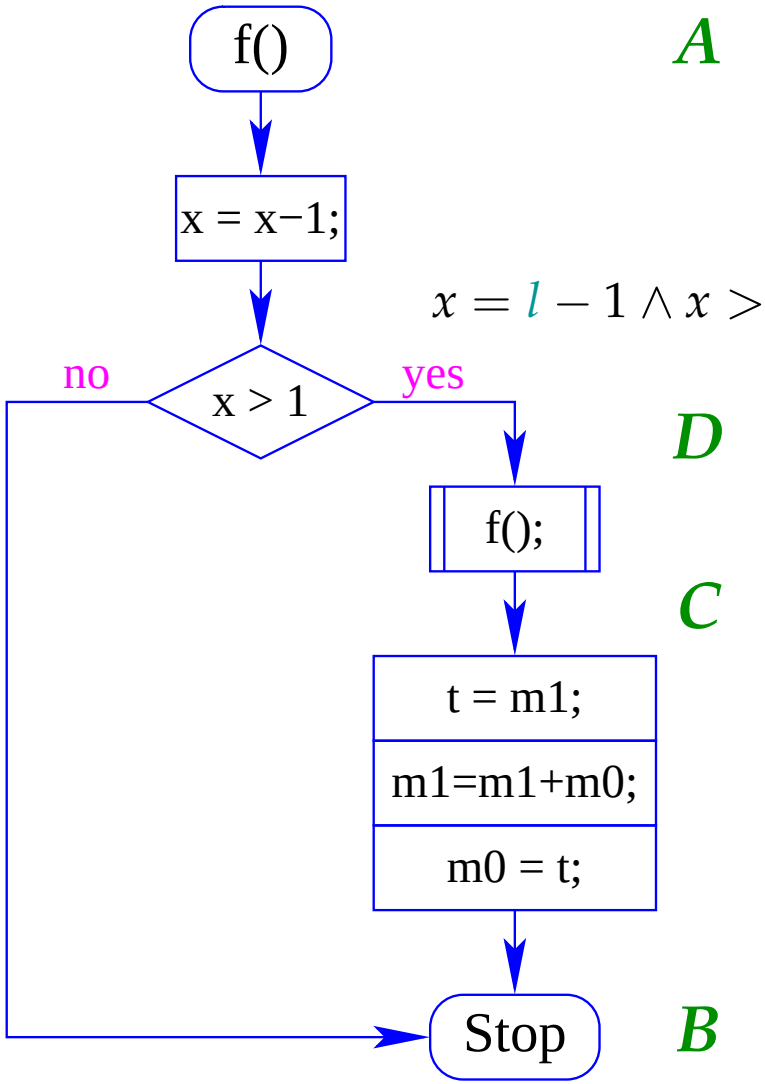
$$\{A\}\ \texttt{f();}\ \{B\}$$

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

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

$$\{A\}\ \texttt{ss}\ \{B\}$$

... im Beispiel:


f()
A
x = x−1;
$x = l - 1 \wedge x > 0 \wedge m_0 = m_1 = 1$
no
x > 1
yes
D
f();
C
t = m1;
m1=m1+m0;
m0 = t;
Stop
B

- 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 $\mathbf{WP}[\![\ldots]\!]$ und Abschwächung ...

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

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:
$$\forall l.(h = l) \Rightarrow (x = 2l)$$
$$\equiv \quad (x = 2h)$$

Für unsere Fibonacci-Funktion berechnen wir:

$$\forall\, l.\ \ (h > 1 \wedge h = l \wedge h_0 = h_1 = 1) \ \ \Rightarrow$$
$$l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1}$$

$$\equiv \quad (h > 1 \wedge h_0 = h_1 = 1) \ \Rightarrow \ m_1 \leq 2^h \wedge m_0 \leq 2^{h-1}$$

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

$$\frac{\forall\, \underline{l}.\;\; A[\underline{h}/\underline{x}] \Rightarrow B \qquad\qquad A_1[\underline{h}/\underline{x}]}{B_1}$$

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

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

Beispiel: `double()`

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

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

$$\frac{\forall\, \underline{l}.\;\; A[\underline{h}/\underline{x}] \Rightarrow B \qquad\qquad A_1[\underline{h}/\underline{x}]}{B_1}$$

Beispiel: `double()`

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

Wir überprüfen:

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

:-)

Bemerkungen:

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

- indem wir die logischen Variablen substituieren:

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

Bemerkungen:

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

- indem wir die logischen Variablen substituieren:

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

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

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

Bemerkungen (Forts.):

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

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

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

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

Anwendung auf Fibonacci:

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

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

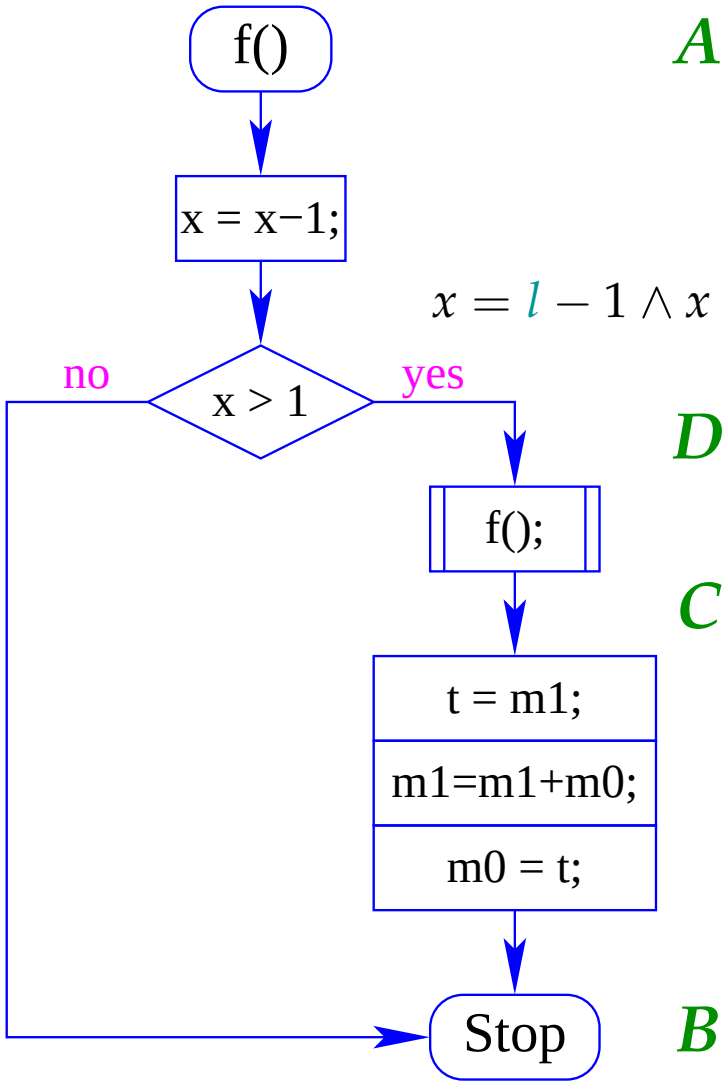
Anwendung auf Fibonacci:

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

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

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

Orientierung:



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

Für die bedingte Verzweigung verifizieren wir:

$$\begin{aligned}
\mathbf{WP}[\![\texttt{x>1}]\!]\,(B, D) \quad &\equiv \quad (x \leq 1 \wedge l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1}) \;\vee \\
&\qquad (x > 1 \wedge x = l - 1 \wedge m_1 = m_0 = 1) \\
&\Leftarrow \quad x > 0 \wedge x = l - 1 \wedge m_0 = m_1 = 1
\end{aligned}$$

:-))

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:

  $\rightarrow$ Die Vor- und Nachbedingungen: $\{A\}, \{B\}$ für Prozeduren sprechen nur über globale Variablen !

  $\rightarrow$ Die $\underline{h}$ werden nur für die globalen Variablen eingesetzt !!

- Der Erhaltung der lokalen Variablen tragen wir automatisch Rechnung, wenn wir bei der Aufstellung der globalen Hypothese beachten:

  $\rightarrow$ Die Vor- und Nachbedingungen: $\{A\}, \{B\}$ für Prozeduren sprechen nur über globale Variablen !

  $\rightarrow$ Die $\underline{h}$ werden nur für die globalen Variablen eingesetzt !!

- Als neuen Spezialfall der Adaption erhalten wir:

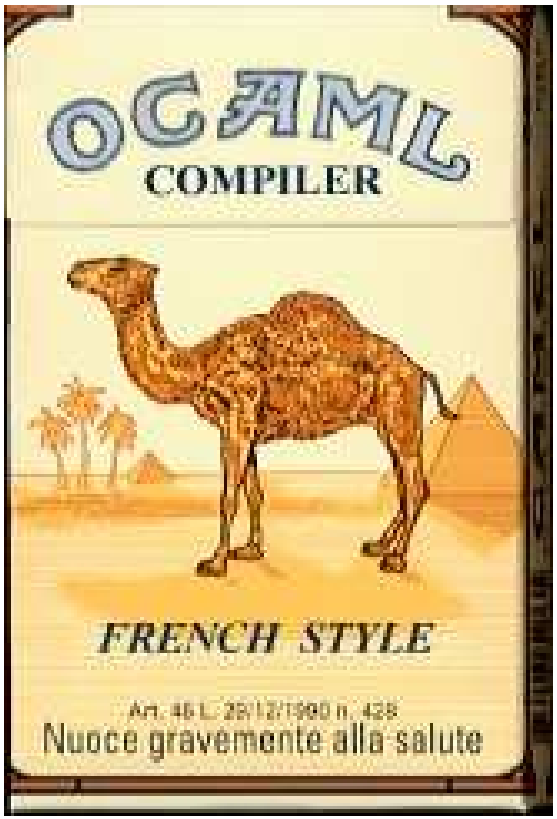
$$\frac{\{A\}\ \texttt{f();}\ \{B\}}{\{A \wedge C\}\ \texttt{f();}\ \{B \wedge C\}}$$

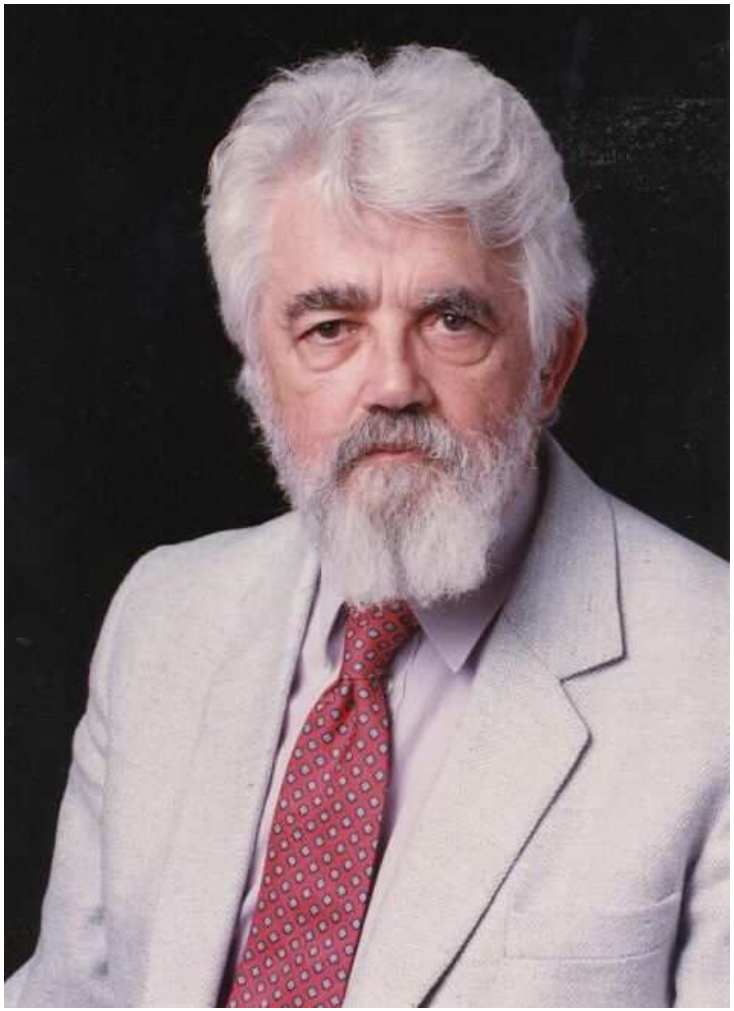
  falls C nur über logische Variablen oder lokale Variablen des Aufrufers spricht :-)

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 $\Longrightarrow$ Vollständigkeit ?
- Wie weit lässt sich Verifikation automatisieren ?
- Wieviel Hilfe muss die Programmiererin und/oder die Verifiziererin geben ?

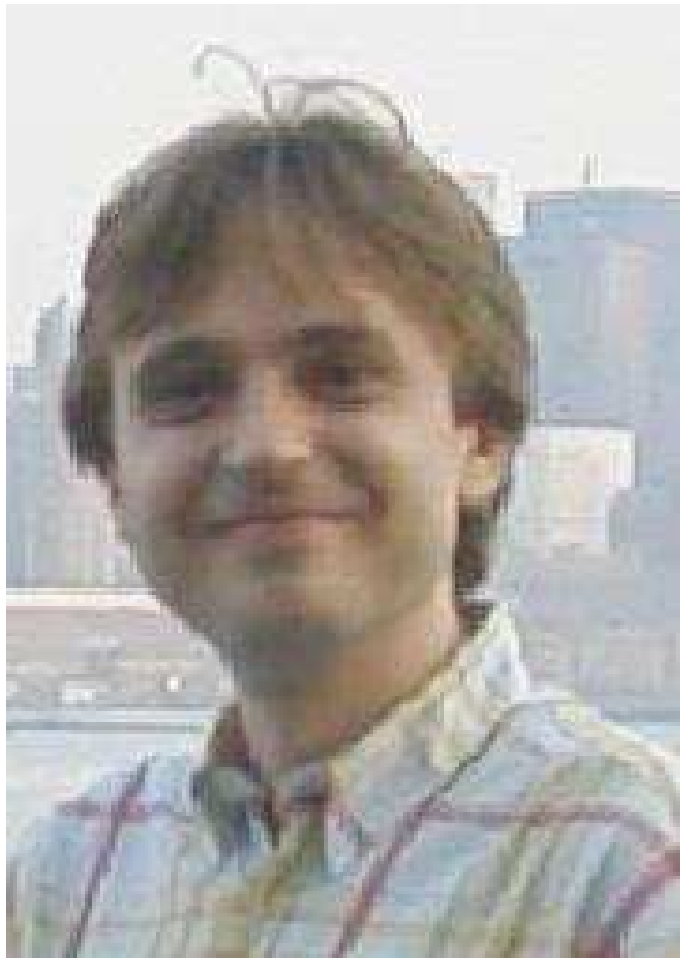
Funktionale Programmierung





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`	`+. -. *. /.`
bool	`true false`	`not || &&`
string	`"hallo"`	`^`
char	`'a' 'b'`	

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

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
```