K01

Einführung – Teil II

1. Programmiersprache
2. Softwareprozessor bzw. Übersetzer
   - Compiler versus Interpreter
3. Typ, Typsystem, Typumwandlungen
4. Bezeichner, Gültigkeitsbereich
5. Anweisung, Ausdruck

Was ist eine Programmiersprache?

Informell:

Eine Programmiersprache ist eine künstliche Sprache zur Notation von Programmen.[1]

Etwas genauer:

Eine Programmiersprache ist eine Menge von Wörtern über einem endlichen Alphabet. Sie dient der Umsetzung von Algorithmen in eine maschinenverarbeitbare Form.[2]

[1] ISO/IEC 2382-1:1993 – Information technology – Vocabulary – Part 1: Fundamental terms

[2] nach R. Dumke

Definition Programmiersprache

Formell:

Sei $G = (V, \Sigma, R, \sigma)$ eine (i.d.R. kontextfreie) Grammatik, bestehend aus

- V, eine endliche Menge der Nichtterminale (Variablen),
- Σ, eine endliche Menge – das Alphabet (Terminale) – so dass $V \cap \Sigma = \varnothing$,
- R, die Menge der Ableitungsregeln (Produktionsregeln), sowie
- $\sigma \in V$, dem Startsymbol.

Eine Programmiersprache ist ein Tripel $L_{\text{prog}} = (L(G), S, I)$ wobei

- $L(G) = \{w \in \Sigma^* \mid \sigma \Rightarrow_G^* w\}$ die Sprache ist, die durch die Grammatik G generiert wird (die Menge der ableitbaren Wörter über G),
- S die Menge der Interpretationsregeln ist – Semantik – und
- $I : R \rightarrow S$ eine berechenbare Funktion ist – Implementierung.

nach R. Dumke

Softwareprozessor bzw. Übersetzer

- Informell: Ein Softwareprozessor/Übersetzer verarbeitet ein gegebenes Programm in einer wohldefinierten Art und Weise indem er es:
  - transformiert in ein i.d.R. äquivalentes Programm, meist einer anderen Sprache.
- Oberbegriff für:
  - Präprozessor, Postprozessor, Compiler, Generator, Disassembler, ...

Formell:
Seien L_0, L_1 und L_2 formale Sprachen; ($L_0 = L_1$, $L_0 = L_2$ bzw. $L_1 = L_2$ möglich). Dann ist ein Softwareprozessor ein Tripel

$$p = (L_0, L_1, L_2)$$

welcher ein gegebenes L_0-Programm in ein L_1-Programm transformiert und welcher selbst in L_2 geschrieben ist. M.a.W.: Abbildung von L_0 auf L_1, implementiert durch L_2.

Beispiel – Compiler

Sei L_0 die Programmiersprache C++ (“höhere” Sprache mit einer mächtigeren und abstrakteren Befehlsmenge), L_1 die Intel x86 Maschinensprache („einfache“ Sprache), und sei L_2 die Programmiersprache C (Implementierungssprache).

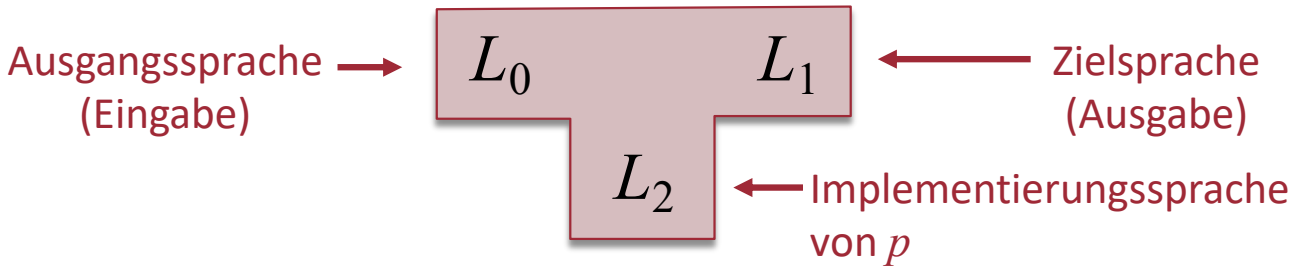
Dann ist ein C++-Compiler für x86 Prozessoren der selbst in C geschrieben ist ein Software-Prozessor

$$p = (L_0, L_1, L_2) \ .$$

Grafische Darstellung

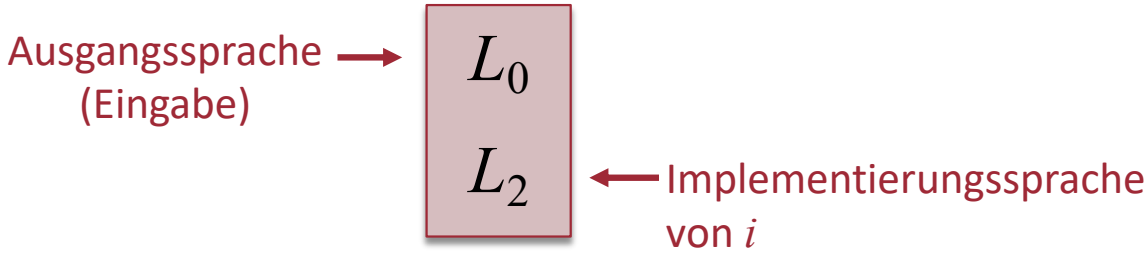
McKeeman-Diagramm (auch Tombstone-Diagramm)

$$p = (L_0, L_1, L_2)$$



Bei einem Interpreter i gibt es die Zielsprache L_1 nicht, deshalb:

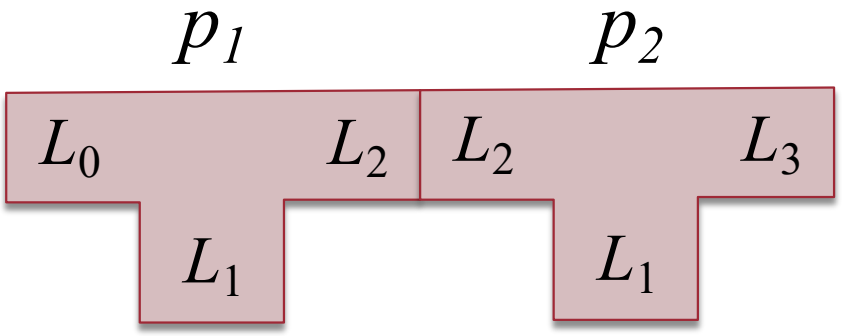
$$i = (L_0, L_2)$$



Verknüpfung von S'prozessoren

1. Komposition

$$p_2 \circ p_1 = p$$



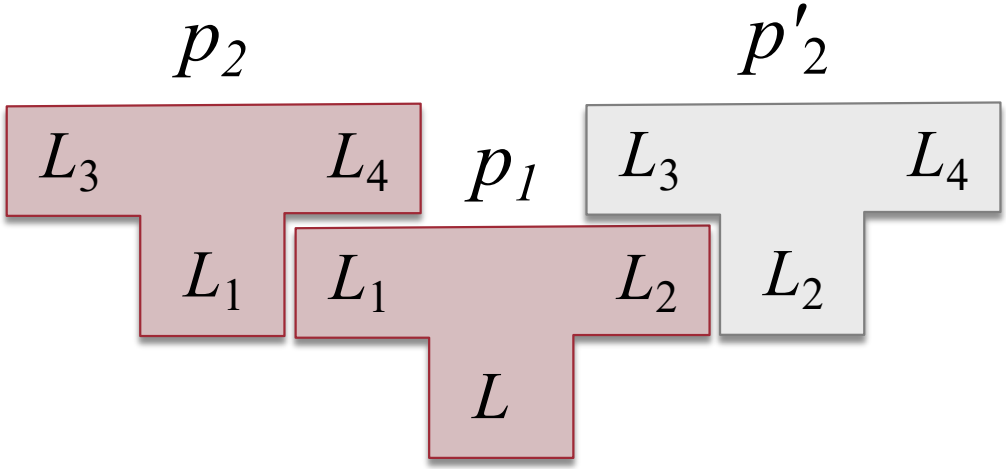
p_1 und p_2 werden hintereinander ausgeführt und damit zu $p = (L_0, L_3, L_1)$ zusammengefasst

Beispiel: Java (L_0) → Bytecode (L_2) → Maschinencode (L_3)
p_1 Bytecode Compiler, p_2 Just-in-time Compiler

Verknüpfung von S'prozessoren

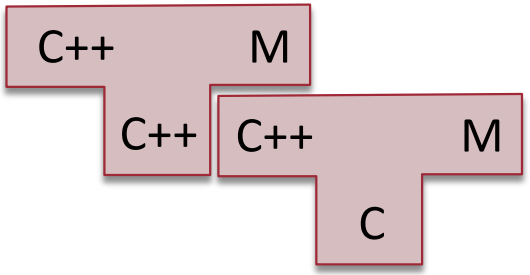
2. Applikation

$$p_1(p_2) = p_2'$$



p_1 wird auf p_2 in der Weise angewandt, dass er die Implementierungssprache von L_1 auf L_2 ändert $p'_2 = (L_3, L_4, L_2)$

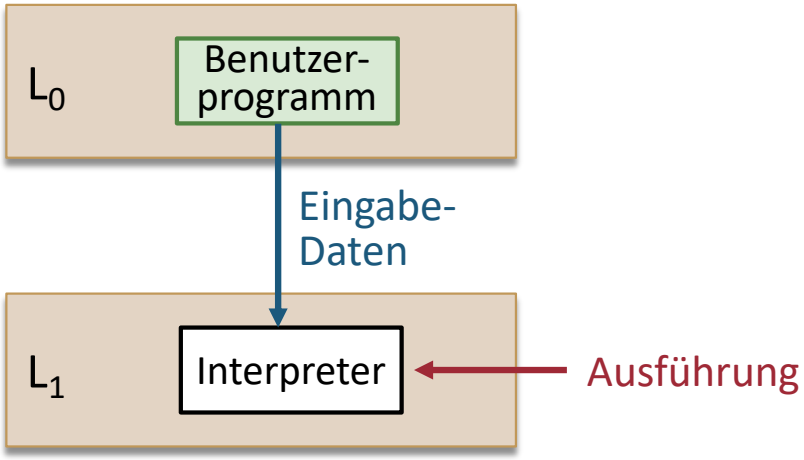
Beispiel: Bootstrapping eines C++ Compiler der in C++ geschrieben ist durch einen C++ Compiler der in C geschrieben ist.



Interpreter versus Compiler

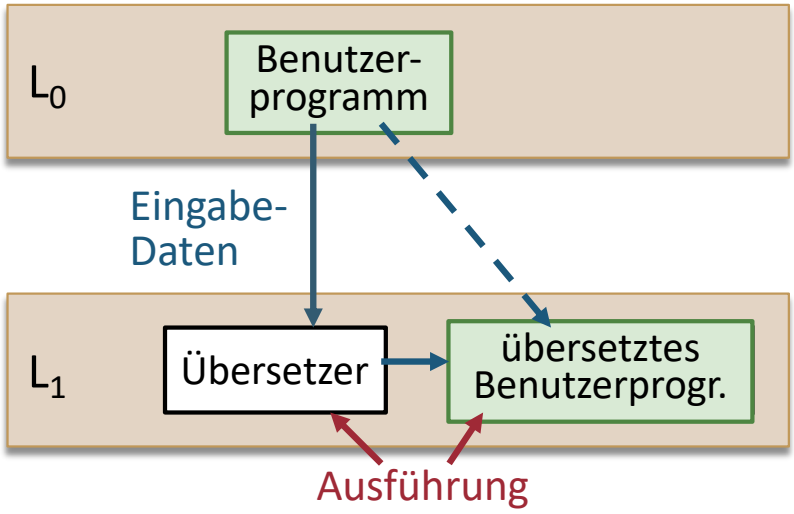
Interpretation:

Es wird ein (ständig laufendes) L_1–Programm (Interpreter) vorgegeben. Dieses Programm nimmt das L_0–Programm als Eingabe, überführt jeden L_0–Befehl in eine Folge von L_1–Befehlen und führt diese sofort aus.



Übersetzung:

Das komplette L_0–Programm wird in eine Folge von L_1–Befehlen (L_1–Programm) übersetzt. Dieses Programm kann dann direkt ausgeführt werden. Das ursprüngliche L_0–Programm wird nicht mehr benötigt.



Typsystem und Typ (i)

- Objekte, die mit einem Programm manipuliert werden, unterteilt man intuitiv in disjunkte Mengen.

Beispiele:

Daten: (ganze) Zahlen, Zeichenketten, Datum, Zeit, E-Mail, ...

Datenstrukturen: Listen, Matrizen, Graphen, ...

Funktionen: int -> int , ...

X_{int} $X_{\text{List<int>}}$ $X_{\text{2d-matrix<int>}}$ $X_{\text{function int -> int}}$

$5 \in X_{\text{int}}$ $[5,2] \in X_{\text{List<int>}}$ $[[5,1],[1,2]] \in X_{\text{2d-matrix<int>}}$

Typsystem und Typ (ii)

- Objekte verschiedener Mengen haben unterschiedliche Eigenschaften (aber innerhalb einer Menge dieselben), und demzufolge sind für Objekte verschiedener Mengen unterschiedliche Operationen sinnvoll.

Beispiele:

Zahlen: Addition, Subtraktion, ...

Listen: Konstruktor, Konkatenation, (indizierte) Selektion, ...

Funktionen: Aufruf, Komposition $f(g(x))$

Typsystem und Typ (iii)

Definition *Typ*:

Ein Typ ist ein 2-Tupel $\mathbf{T} = (X, O)$, wobei X eine Menge[1] von gleichartigen Objekten $x \in X$, und O eine Menge von Operationen $o \in O$ ist, die auf X definiert sind.[2]

Beispiele:

$$\mathbf{T}_{\text{boolean}} = (\ \{\text{true}, \text{false}\},\ \{\&\&, ||, !\}\ )$$

$$\mathbf{T}_{\text{32-bit unsigned int}} = (\ \{0, 1, 2, \ldots, 2^{32}\text{-}1\},\ \{+, -, *, /\}\ )$$

[1] X kann theoretisch unendlich sein; praktisch ist X aber immer endlich, da heutige Maschinen endliche Ressourcen haben (z.B. Speicher). Leere Menge für X, O u.U. sinnvoll.
[2] Jede Operation $o \in O$ ist i.d.R. total, kann aber auch partiell sein. Stelligkeit von Operationen meist unär oder binär. Definitionsbereich D einer Operation $o \in O$ allgemein: $D \subseteq X_1 \times \ldots \times X_n$ mit $n \geq 0$. Falls $n \geq 1$ dann $\exists X_i$: $X_i = X$ $(0 \leq i \leq n)$ und alle anderen X_j $(i \neq j)$ nicht notwendigerweise gleich X.

Typsystem und Typ (iv)

Definition *Typsystem*:

Ein Typsystem ist ein 2-Tupel $\mathbf{S} = (A, R)$, wobei A eine Funktion ist, die einem Term t einer Programmiersprache einen Typ $\mathbf{T}$ zuordnet – t ist mit $\mathbf{T}$ assoziiert –, und R eine Menge von Regeln ist, mit denen man das Vorhandensein bzw. Nichtvorhandensein von Typfehlern zwischen zwei Termen t, u beweisen bzw. zusichern kann.

Typ versus Wert *engl. value*

- Die *Repräsentation* eines Objektes $x \in X$ eines Typs $\mathbf{T}$ wird Wert genannt.
- Der Wert einer Variable ist somit letztlich eine Abbildung auf die Repräsentation in der Umgebung (z.B. dem Arbeits-speicher) und wird meist durch die Hardware realisiert.

- Bei Sprachen mit veränderlichen Variablen (wie z.B. C, C++), d.h. deren Wert (jederzeit) durch Zuweisung verändert werden kann, wird zusätzlich zwischen L-values und R-values unterschieden.
  --> Dazu später genauer.

Wesentliche Typklassen

- Datentypen
- Zusammengesetzte Typen (Datenstrukturen)
- Zeigertypen
- Funktionstypen

... u.a.

- Primitive Datentypen (auch elementar genannt) sind integraler Bestandteil einer Programmiersprache und i.d.R. gegeben durch die Maschine (CPU) für die eine Sprache implementiert ist, d.h. für die ein Compiler/Interpreter existiert.
- Benutzerdefinierte Typen werden durch Ableitung oder Konstruktion aus primitiven Typen gebildet und sind, wie der Name andeutet, nicht integraler Bestandteil einer Programmiersprache. Die Ableitungs- bzw. Konstruktionsarten sind i.d.R. durch die jeweilige Programmiersprache vorgegeben und endlich. Nichtsdestotrotz lassen sich i.d.R. unendlich viele benutzerdefinierte Typen, die u.U. aber strukturell gleich sind oder identische Wertebereiche habe, definieren.

Aufgaben von Typsystemen (i)

1. Erkennen von Typverletzungen – nicht erlaubte oder nicht sinnvolle „Dinge“ verhindern, z.B.:
   - Anwendung einer Operation, die auf einem Datentyp nicht definiert ist.
   - Falsche Argumentanzahl und/oder -typ bei Funktions-/Methodenaufruf.

→ Erkennen erlaubter Werte

→ Erkennen erlaubter Operationen

→ Verhinderung der (unbeabsichtigten) Nutzung einer Operation, die für einen Typ nicht definiert ist.

→ Eindeutige Auswahl einer Operation aus Menge von möglichen Operationen; insbesondere wenn Operationsnamen überladen sind.

Aufgaben von Typsystemen (ii)

2. Definition der Semantik (und Implementierung) von Typumwandlungen.

Es existieren im Wesentlichen drei Arten von Typumwandlungen:

1. Up cast — Promotion / Generalisierung
2. Down cast — Demotion / Spezialisierung
3. Type coercion — Allgemein eine Konvertierung zwischen möglicherweise disjunkten Typen.

Typumwandlung können auf zwei Arten implementiert werden:

1. Materiell — Repräsentation eines anderen Typs wird (temporär im Speicher oder einem Register) erzeugt.
2. Virtuell — Gegebene Repräsentation wird lediglich als Repräsentation eines andern Typs reinterpretiert, ohne dass dabei neue Repräsentation erzeugt wird.

Typumwandlung: *cast*

Wird für Datentypen verwendet. Beteiligt sind zwei Typen:

Supertyp $\mathbf{T}_{\text{sup}} = (X_{\text{sup}}, O_{\text{sup}})$

Subtyp $\mathbf{T}_{\text{sub}} = (X_{\text{sub}}, O_{\text{sub}})$ so dass $X_{\text{sub}} \subseteq X_{\text{sup}}$

1. Demotion: down cast $x_{\text{sup}} \rightarrow x_{\text{sub}}$ explizit anzufordern
2. Promotion: up cast $x_{\text{sub}} \rightarrow x_{\text{sup}}$ meist implizit

Beispiele:

Java: explicit down cast

```
int x = 5; long y = 5_000_000_000l;
int z = x + (int) y;
```

Java: implicit up cast

```
int x = 5; long y = 5_000_000_000l;
long z = x + y;
```

Implizite Umwandlung: *type coercion*

Wie bei Casts sind auch hier zwei Typen $\mathbf{T}_1$, $\mathbf{T}_2$ beteiligt. Im Allgemeinen wird aber nicht $X_1 \subseteq X_2$ gefordert; d.h. type coercion kann insbesondere auch für $X_1 \cap X_2 = \emptyset$ definiert sein.

Meist automatisch/implizit vom Compiler/Interpreter durchgeführt.

Beispiele:

== Operator in JavaScript

```
(true == 1)   // → true
(false == "") // → true
(false == []) // → true

// crazy things in JavaScript
(true  == {}) // → false
(false == {}) // → false
if ({}) { /* executed */ }
if ([]) { /* executed */ }
```

C, C++ u.a.

```
5 + 2.2 // =7.5; 5 coerced to float

int x = -1, y = 5;
if (x = y) { /* executed */ }
```

JavaScript

```
"2" + 2 // ="22"; 2 coerced to string
```

Typsysteme

- Im weiteren Verlauf der Vorlesung werden wir weitere Begriffe vertiefen:

Typisierung ...

- ... nominativ versus strukturell — Typgleichheit
- ... statisch versus dynamisch — Typüberprüfung
- ... schwach versus stark — Typsicherheit
- ... explizit versus implizit — Typbestimmung

Nichtorthogonale Dimensionen, d.h. es bestehen Zusammenhänge. Z.B. ist schwache Typisierung in einigen dynamischen Sprachen einhergehend mit einem implizitem dynamischen Typ. Starke Typisierung in vielen Sprachen einhergehend mit einer expliziten statischen Typdeklaration.

Bezeichner (Symbol) *engl. identifier*

- Ein Bezeichner ist der **Name** eines Programmelementes; z.B.:
  - Variable / Zeigervariable / Referenz
  - Funktion / Methode / Prozedur
  - Struktur / Klasse / Aufzählung
  - Alias
  - Atom / Prädikat
- In systemnahen bzw. imperativen Sprachen ist ein Bezeichner meist fest mit einem Speicherbereich assoziiert, insbesondere ist dies für Variablen der Fall.

Bezeichner (ii)

- Meist unterliegen Bezeichner bestimmten lexikalischen Regeln (sind also nicht beliebig wählbar):
  - Längenbegrenzung
  - Schlüsselwörter nicht (überall) erlaubt, z.B.

    ~~`do if for while switch class struct`~~ `...`
  - Alphabet erlaubter (Start-)Zeichen, z.B.

    `a..z` **`0..9`** **`üöäéè`** **`_-%$@&`**
- Beachte: Viele Sprachen bieten die Möglichkeit für anonyme Elemente, die also nicht benannt sind, und nur aus ihrem Kontext heraus identifizierbar sind.

Namensauflösung *engl. name resolution*

- Eindeutige Abbildung von Bezeichnern/Namen auf Programmelemente (z.B. Speicherbereich, Speicheradresse).
  - Realisiert/implementiert durch Compiler, Linker oder Interpreter.

- Assemblersprachen
  - Bezeichner werden durch eine simple Lookup-Tabelle auf Programmelemente, abgebildet
- Höhere Sprachen
  - Mehr oder weniger komplexe Mechanismen/Regeln.

Beispiel: Namensauflösung in C++

- C++ gilt als eine der komplexesten Sprachen hinsichtlich der Regeln zur Namensauflösung.
- In C++ ist die Namensauflösung beeinflusst durch:
  - *Namespaces* – identische Bezeichner können je nach Kontext unterschiedlichen Programmelementen zugeordnet sein.
  - *Lexical Scope* – Programmabschnitt.
  - *Visibility* – Regeln, die innerhalb Vererbungshierarchien zum Tragen kommen.
  - *Overloading* – Regeln, die gestatten, dass unterschiedliche Elemente (insbesondere Funktionen) über identische Bezeichner angesprochen werden können.

Gültigkeitsbereich *engl. scope*

- Programmabschnitt*, in dem ein Bezeichner nutzbar bzw. sichtbar ist.
  - Nicht notwendigerweise zusammenhängend!
  - Bei Variablen: nicht notwendigerweise identisch mit Lebensdauer, d.h. Zeitraum in dem für Variable Speicher reserviert ist! (z.B. **`static`** in C, C++, Java)

Gültigkeitsbereich ≤ Lebensdauer

Sichtbarkeit unterliegt präzise definierten Regeln.

* Gemeint sind damit Abschnitte des Quelltextes, nicht zeitliche Abschnitte zur Laufzeit.

Gültigkeitsbereich (ii)

- Es existieren im Wesentlichen zwei unterschiedliche Regeltypen, die die Sichtbarkeit definieren:

1. Lexikalischer Gültigkeitsbereich *engl. lexical scope*
   - Der Programmkontext (d.h. der umgebende Programmtext) bestimmt den Gültigkeitsbereich.
   - Gültigkeit unabhängig vom Aufruf-Stack.
   - Heutzutage am verbreitetsten.

→ Kann vor der Laufzeit (also statisch) bestimmt werden.

Gültigkeitsbereich (iii)

2. Dynamischer Gültigkeitsbereich *engl. dynamic scope*
   - Der Programmablauf und der Programmzustand zur Laufzeit (d.h. die Programmausführungsgeschichte bis zum momentanen Zeitpunkt) bestimmen den Gültigkeitsbereich.
   - Gültigkeit gekoppelt an Aufruf-Stack; d.h. sie leitet sich direkt von den Regeln ab, wann auf dem Stack ein neuer Aufruf-Frame erzeugt wird und wann dieser wieder gelöscht wird. (Lässt sich dadurch leichter implementieren als static scope, wo im Wesentlichen der Compiler verantwortlich ist).

→ Kann im Allgemeinen nur zur Laufzeit (also dynamisch) bestimmt werden.

Gültigkeitsbereich - Beispiel (iv)

JavaScript – lexical scope

```
var x=1;
function g() { console.log(x); x=2; }
function f() { var x=3; g(); }
f();              // Ausgabe 1, oder 3?
console.log(x); // Ausgabe 1, oder 2?
```

g() gibt globale Variable **x** aus, da in erster Zeile definiert.

Ausgabe auf Konsole:

```
1
2
```

Bash – dynamic scope

```
x=1
function g() { echo $x; x=2; }
function f() { local x=3; g; }
f       # Ausgabe 1, oder 3?
echo $x # Ausgabe 1, oder 2?
```

g() gibt **f()**'s lokales **x** aus und modifiziert dieses.

Ausgabe auf Konsole:

```
3
1
```

Anweisung *engl. statement*

- Befehl mit wohldefinierter Semantik
  - Entweder direkt ausführbar (Maschinenbefehl), oder
  - Übersetzung in eine Menge von Maschinenbefehlen.
  - Charakteristisch für imperativen Sprachen

- Wesentliche Anweisungsarten:
  - Deklaration `int x;`
  - Zuweisung `x = 10;`
  - Kontrollstruktur
    ```
    if (x == true) { ... }
    else { ... }
    while (x > 0) { ... }
    ```
  - Sprunganweisung **`goto`**, **`return`**, **`break`**, **`()`**

Ausdruck *engl. expression*

- Liefert bei Auswertung einen Wert (Ergebnis)
  - Auswertung gemäss:
    - math. Kalkühl/Theorie (z.B. sukzessive Termersetzung)
    - Auswertungsstrategie (z.B. *lazy* oder *eager*; dazu später mehr)
  - Charakteristisch für deklarative Sprachen
  - Manche Anweisungen sind auch Ausdrücke, z.B. Zuweisung, Prozeduraufruf

- Wesentliche Ausdrucksarten:
  - Arithmetische `3*3` `x+(*y)` `f.g x`
  - Aussagenlogische `a||b` `4==3` `!c`
  - Prädikatenlogische `birthplace(J.S.Bach,Eisenach)`

If you really care about software then you have to care about hardware.

Steve Wozniak, so oder so ähnlich