

Programmiersprachen  
Vorlesung 4 vom 30.10.23  
Variablen und Speichermodelle

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

# Wo sind wir?

- ▶ Einführung
- ▶ Einfache Ausdrücke und Werte
- ▶ Einfache Typen, Anweisungen und Seiteneffekte
- ▶ Variablen und Speichermodelle
- ▶ Aggregierende Typen
- ▶ Ausnahmen und Fehlerbehandlung
- ▶ Prozeduren und Funktionen
- ▶ Fortgeschrittene Typsysteme
- ▶ Datenabstraktion
- ▶ Programmierparadigmen
- ▶ Skriptsprachen
- ▶ Ab Woche 11 (2024): Studentische Vorträge.

# Variablen, Namen, Deklarationen

- ▶ Was genau macht eine **Variablendeklaration**?

```
int x;
```

```
...
```

# Variablen, Namen, Deklarationen

- ▶ Was genau macht eine **Variablendeklaration**?

```
int x;  
  
...
```

- 1 Sie führt den **Namen** (Bezeichner) `x` ein.
- 2 Sie reserviert Platz im **Speicher**.

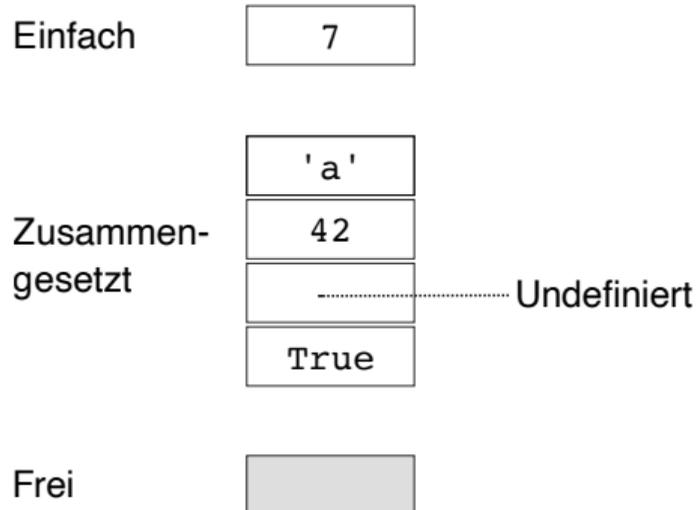
- ▶ Vergleiche:

```
let x= 37 in ...
```

- ▶ Hier wird **nur** der **Name** eingeführt.

# Ein Einfaches Speichermodell

- ▶ Der Speicher hat eine Menge von **Speicherzellen** mit einer eindeutigen **Adresse**
- ▶ Speicherzellen haben einen **Status**:
  - ▶ **Belegt** (allocated) oder **frei** (unallocated)
  - ▶ Belegte Speicherzellen haben einen **Inhalt**, entweder ein **Wert** oder **undefiniert**.
- ▶ Zusammengesetzte Werte belegen mehrere Speicherzellen (“composite variables”)
- ▶ Abstraktion über Wortbreite etc.



# Einfache Variablen

- ▶ Unterschied: Name `n` als **Adresse** der Variable vs. `n` als **Wert** der Speicherzelle mit dieser Adresse
- ▶ Unterschied nach Kontext:
  - ▶ Links der Zuweisung (“L-Wert”) vs. rechts der Zuweisung (“R-Wert”)

```
x = x + 1
```

- ▶ In funktionalen Sprachen sind Variablen **unveränderlich**
  - ▶ Es gibt nur lokale **Namen**
- ▶ Keine Zuweisung, keine L-Werte
- ▶ **Verwirrend**: Rust nutzt `let x = 37`; zur Deklaration von **Variablen**.

```
let x = 37 in x + 1
```

# Lebenszyklus einer Variablen

- ▶ Generell haben Variablen einen **Lebenszyklus**: Allokation, Nutzung, Deallokation
  - ▶ Bei der **Allokation** wird Platz im Speicher reserviert
  - ▶ Bei der **Deallokation** wird der Speicher wieder freigegeben
- ▶ Klassifikation von Variablen nach der Lebensdauer:
  - ▶ **Global** oder statisch — ganze Laufzeit des Programmes
  - ▶ **Lokal** oder automatisch — innerhalb eines **Blocks**
  - ▶ **Heap** — beliebig, aber höchstens bis Programmende
  - ▶ **Persistent** — länger als das Programm (e.g. Dateien, Datenbanken)

# Block

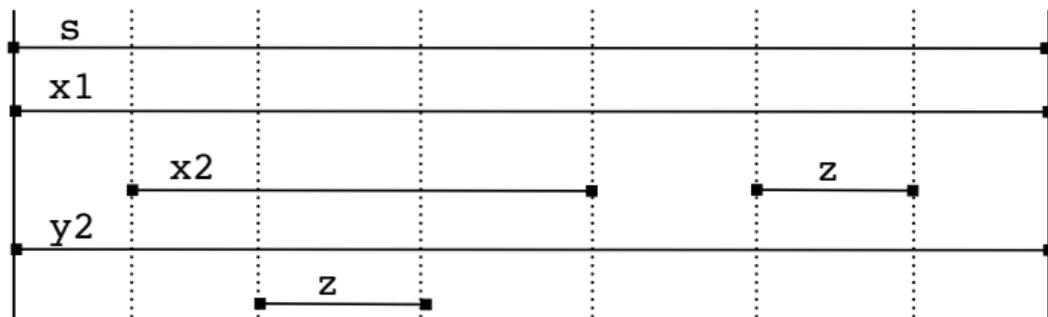
- ▶ Ein **Block** ist ein Programmabschnitt zusammen mit **lokalen Deklarationen**
- ▶ Blöcke dienen zur
  - ▶ **Gruppierung** von Anweisung
  - ▶ **Verkapselung** (durch lokale Deklarationen)
- ▶ Fast alle Programmiersprachen haben **verschachtelte Blöcke**
- ▶ Blöcke bestimmen die Lebensdauer und Sichtbarkeit der lokalen Variablen
- ▶ NB: Lebensdauer  $\neq$  Sichtbarkeit

# Lebensdauer — Beispiel

```
char s[] = "Foo";  
void main()  
{ int x1;  
  ... P(); ... Q(); ...  
}
```

```
void P()  
{ int *x2; static float y2;  
  ... Q(); ...  
}  
void Q()  
{ float z;  
  ...  
}
```

start call P call Q return Q return P call Q return Q stop



# Bindung und Scope

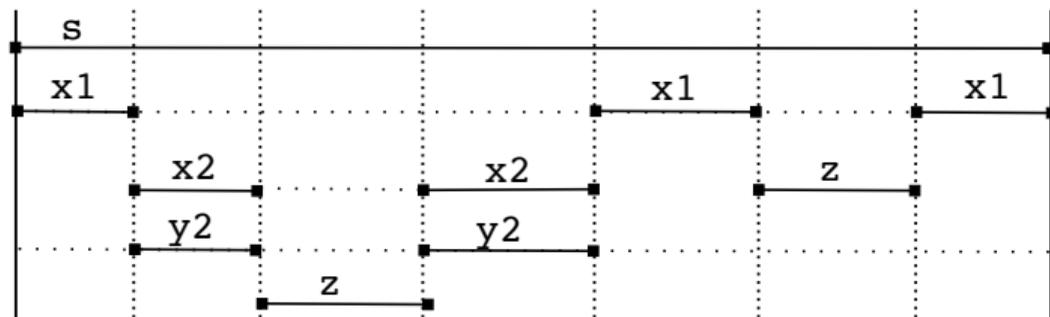
- ▶ Eine **Bindung** assoziiert lexikalische Bezeichner mit einem semantischen Wert
  - ▶ Abstrakt: symbolische Bezeichner der ausführenden abstrakten Maschine
  - ▶ Konkret: Speicheradresse
- ▶ Eine **Umgebung** ist eine Menge von Bindungen
- ▶ Der **Scope** eines Bezeichners ist sein Gültigkeitsbereich oder Sichtbarkeitsbereich
- ▶ Unterscheidung:
  - ▶ Statischer (oder lexikalischer) Scope — Gültigkeitsbereich wird zur Übersetzungszeit festgelegt
  - ▶ Dynamischer Scope — Gültigkeitsbereich wird während der Laufzeit festgelegt

# Sichtbarkeit ist nicht Lebensdauer — Beispiel

```
char s[] = "Foo";  
void main()  
{ int x1;  
  ... P(); ... Q(); ...  
}
```

```
void P()  
{ int *x2; static float y2;  
  ... Q(); ...  
}  
void Q()  
{ float z;  
  ...  
}
```

start call P call Q return Q return P call Q return Q stop



# Static vs. Dynamic Scope

Ein Beispielprogramm (fiktive Syntax):

```
s= 2

def foo(x):
    print(s*x)

def baz(y):
    foo(y)

def bah(z):
    s= 4 # Implicitly declared as local,
    foo(z)

bah(5)
baz(5)
```

## ▶ **Statisch**

- ▶ C, Java, Python, Haskell:
- ▶ Ausgabe 10, 10
- ▶ Python hat "late binding"
- ▶ Alternative Ausgabe: 20, 20  
(dann ist `s` in `bah` global)

## ▶ **Dynamisch**

- ▶ Perl, shell:
- ▶ Ausgabe 20, 10

# Deklarationen

- ▶ Deklarationen führen eine Bindung ein.
- ▶ **Komposition** von Deklarationen:
  - ▶ Sequential
  - ▶ Rekursiv
  - ▶ Kollateral
- ▶ Beispiel Standard ML: kann alles

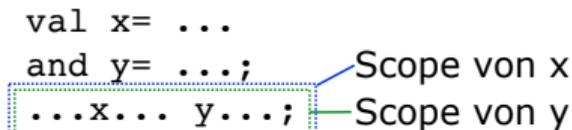
**Übung 2.1:** Wie werden Deklarationen in C, Java, Python, Haskell gehandhabt?

```
val x= ...;  
val y= ...;  
...x... y...;
```



Sequentielle Deklarationen

```
val x= ...  
and y= ...;  
...x... y...;
```



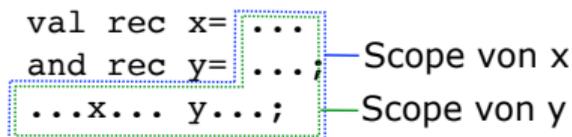
Kollaterale Deklarationen

```
val rec x= ...  
... x ...
```



Rekursive Deklaration

```
val rec x= ...  
and rec y= ...;  
...x... y...;
```



Rekursive, kollaterale Deklarationen

# Speicherverwaltung

- ▶ Der Speicher wird meist unterteilt in einen **Stack** und einen **Heap**
- ▶ Der Stack verwaltet lokale Variablen:
  - ▶ Für jeden Aufruf einer Funktion ein **Stack Frame**
  - ▶ Wird am Ende der Funktion wieder entfernt
- ▶ Der Heap verwaltet Heap-Variablen
  - ▶ Allokation manuell (C, `malloc`) oder durch Konstruktor (`new`)
  - ▶ Deallokation manuell (C, `free`) oder durch **Garbage collector**
- ▶ Garbage-Collection Algorithmen:
  - ▶ reference counting, mark&sweep, copy
- ▶ Problemquellen:
  - ▶ Dangling pointers, memory leaks

## Sonderfall: Ownership in Rust

- ▶ Rust hat ein neuartiges Konzept — **Ownership**:
  - ▶ Variable hat **einen** Besitzer, der die Lebenszeit bestimmt:

```
let s1 = String::from("hello");  
let s2 = s1;    // Does not work!
```

- ▶ Variablen können **verliehen** werden:

```
let s1 = String::from("hello");  
let len = calculate_length(&s1);
```

- ▶ Verleihte Variablen dürfen nicht **verändert** werden.
- ▶ Es gibt immer **einen** Besitzer, oder **beliebig viele** unveränderliche Referenzen.
- ▶ Referenzen sind immer **gültig**:

```
fn dangle() -> &String {  
    let s = String::from("hello");  
    &s  
}
```

# Namen und Substitution

# Lokale Namen

$$\begin{aligned} e ::= & \mathbb{Z} \mid i \mid true \mid false \\ & \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\ & \mid e_1 == e_2 \mid e_1 < e_2 \\ & \mid !e \mid e_1 \&\& e_2 \mid e_1 \parallel e_2 \\ & \mid i := e \\ & \mid \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \end{aligned}$$

- ▶ **let**  $x = e$  **in**  $f$  führt den lokalen Namen  $x$  in  $f$  ein, und setzt ihn auf den Wert  $e$ .
- ▶  $x$  ist **nicht veränderlich** (keine **Variable**)!

# Semantik lokaler Namen

- ▶ **let**  $x = e_1$  **in**  $e_2$  ist wie  $e_2$  in dem  $x$  durch  $e_1$  ersetzt ist.
- ▶ Problem: Schachtelung lokaler Namen:

```
let x= 5 in
  let y= 2*x in
    (let x= 3 in x+y)+ x
```

# Substitution

- ▶ Definition  $e_1[e_2/x]$ :

$$v[e/x] = v$$

$$y[e/x] = \begin{cases} e & x = y \\ y & \text{otherwise} \end{cases}$$

$$(e_1 + e_2)[e/x] = (e_1[e/x]) + (e_2[e/x])$$

$$(x := e_1)[e_2/y] = (x := (e_1[e_2/y]))$$

$$(\mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2)[e_3/y] = ???$$

# Freie Variablen

$$FV(v) = \emptyset$$

$$FV(x) = \{x\}$$

$$FV(e_1 + e_2) = FV(e_1) \cup FV(e_2)$$

...

$$FV(i := e) = FV(e)$$

$$FV(\mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2) = FV(e_1) \cup (FV(e_2) \setminus \{x\})$$

**Übung 3.2:** Berechne die freien Variablen in  
 $(\mathbf{let} \ z = x + 3 * y \ \mathbf{in} \ z + x) + (\mathbf{let} \ y = 5 * x \ \mathbf{in} \ y + x)$ .

# Substitution

- ▶ Damit Definition der Substitution:

$$(\text{let } x = e_1 \text{ in } e_2)[e_3/y] = \begin{cases} \text{let } x = e_1[e_3/y] \text{ in } e_2 & x = y \\ \text{let } x = e_1[e_3/y] \text{ in } e_2[e_3/y] & x \neq y, x \notin \text{FV}(e_3) \\ \text{let } z = e_1[e_3/y] \text{ in } (e_2[z/x])[e_3/y] & x \neq y, x \in \text{FV}(e_3), \\ & z \notin \text{FV}(e_3) \cup \text{FV}(e_2) \end{cases}$$

- ▶  $z$  ist eine **frische** Variable.

**Frage:** Berechne  $(\text{let } x = x + y \text{ in } x + 5 == 7 - y)[\text{let } x = x + 3 \text{ in } 2 * x / y]$ .

**Übung 3.3:** Sei

$e \stackrel{\text{def}}{=} \text{let } x = x + y \text{ in } 4 * ((\text{let } y = x + y \text{ in } y + 5) + (\text{let } x = 3 * x * y \text{ in } y + x))$ .

Berechne  $e[3 * x + y / y]$ .

# Auswertung

- ▶ Zwei Möglichkeiten:

$$\frac{\langle e_2[e_1/x], \sigma \rangle \rightarrow_{Exp} \langle v, \sigma' \rangle}{\langle \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2, \sigma \rangle \rightarrow_{Exp} \langle v, \sigma' \rangle} \quad (1)$$

$$\frac{\langle e_1, \sigma \rangle \rightarrow_{Exp} \langle v_1, \sigma' \rangle \quad \langle e_2[v_1/x], \sigma' \rangle \rightarrow_{Exp} \langle v_2, \sigma'' \rangle}{\langle \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2, \sigma \rangle \rightarrow_{Exp} \langle v_2, \sigma'' \rangle} \quad (2)$$

- ▶ (1) ist nicht-strikt (“call-by-name”)
- ▶ (2) ist strikt (“call-by-value”)

# Variablen und Speichermodelle

# Variablen und Speichermodelle

- ▶ Speichermodell  $\Sigma = \mathbf{Loc} \rightarrow \mathbf{V}$  mit  $\mathbf{Loc} = \mathbb{N}$
- ▶ Trennung von Bezeichner **Idt** und Adressen **Loc**
- ▶ Speicherzellen können frei oder belegt (uninitialisiert oder mit Wert) sein
- ▶ Daher  $\mathbf{V} = \mathbb{Z}_{\perp} = \mathbb{Z} \uplus \{\perp\}$

# Erweiterung der Sprache

- ▶ Neues Kommando:

$$c ::= e \mid \mathbf{if} (e) \mathbf{then} c_1 \mathbf{else} c_2 \mid \mathbf{while} (e) c \mid c_1; c_2 \mid \mathbf{nil} \mid \mathbf{new} x : t \mathbf{in} c$$

- ▶ Sequential, nicht kollateral
- ▶ Typisierung:

$$\frac{\Gamma[x \mapsto t] \vdash c : \mathbf{unit}}{\Gamma \vdash \mathbf{new} x : t \mathbf{in} c : \mathbf{unit}}$$

- ▶ Erweiterung der Semantik:

$$\Gamma = \mathbf{Idt} \rightarrow \mathbf{Loc}$$
$$\sigma = \mathbf{Loc} \rightarrow V$$
$$\Gamma \vdash \langle e, \sigma \rangle \rightarrow_{Exp} \langle v, \sigma' \rangle$$
$$\Gamma \vdash \langle c, \sigma \rangle \rightarrow_{Stmt} \sigma'$$

$\Gamma$  ist eine **Umgebung** (statisch, nur zur Übersetzungszeit)

# Semantik: alte Regeln

$$\frac{\Gamma \vdash \langle c_1, \sigma \rangle \rightarrow_{Stmnt} \sigma' \quad \Gamma \vdash \langle c_2, \sigma' \rangle \rightarrow_{Stmnt} \sigma''}{\Gamma \vdash \langle c_1; c_2, \sigma \rangle \rightarrow_{Stmnt} \sigma''}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \rightarrow_{Exp} \langle true, \sigma' \rangle \quad \Gamma \vdash \langle c_1, \sigma' \rangle \rightarrow_{Stmnt} \sigma''}{\Gamma \vdash \langle \mathbf{if} (b) \mathbf{then} c_1 \mathbf{else} c_2, \sigma \rangle \rightarrow_{Stmnt} \sigma''}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \rightarrow_{Exp} \langle false, \sigma' \rangle \quad \Gamma \vdash \langle c_2, \sigma' \rangle \rightarrow_{Stmnt} \sigma''}{\Gamma \vdash \langle \mathbf{if} (b) \mathbf{then} c_1 \mathbf{else} c_2, \sigma \rangle \rightarrow_{Stmnt} \sigma''}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \rightarrow_{Exp} \langle false, \sigma' \rangle}{\Gamma \vdash \langle \mathbf{while} (b) c, \sigma \rangle \rightarrow_{Stmnt} \sigma'}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \rightarrow_{Exp} \langle true, \sigma' \rangle \quad \Gamma \vdash \langle c, \sigma' \rangle \rightarrow_{Stmnt} \sigma'' \quad \Gamma \vdash \langle \mathbf{while} (b) c, \sigma'' \rangle \rightarrow_{Stmnt} \sigma'''}{\Gamma \vdash \langle \mathbf{while} (b) c, \sigma \rangle \rightarrow_{Stmnt} \sigma'''}$$

## Semantik: neue Regeln

$$\frac{x \in \mathbf{ldt}, x \in \text{dom}(\Gamma), \sigma(\Gamma(x)) = v}{\Gamma \vdash \langle x, \sigma \rangle \rightarrow_{Exp} \langle v, \sigma \rangle}$$

$$\frac{\Gamma \vdash \langle e, \sigma \rangle \rightarrow_{Exp} \langle n, \sigma' \rangle \quad n \in \mathbb{Z}}{\Gamma \vdash \langle x := e, \sigma \rangle \rightarrow_{Stmt} \sigma'[\Gamma(x) \mapsto n]}$$

$$\frac{\Gamma[x \mapsto l] \vdash \langle c, \sigma[l \mapsto \perp] \rangle \rightarrow_{Stmt} \sigma' \quad l \notin \text{dom}(\sigma)}{\Gamma \vdash \langle \mathbf{new} \ x : t \ \mathbf{in} \ c, \sigma \rangle \rightarrow_{Stmt} \sigma' \setminus l}$$

# Beispiel

Auswertung mit  $\Gamma = \langle z \mapsto 0 \rangle, \sigma = \langle 0 \mapsto \perp \rangle$

```
new x: int in
  x := 0;
  new y: int in
    y := 5;
    x := y + 3;
    new x: int = 0 in
      z := x + y;
    y := x;
```

# Beispiel

Auswertung mit  $\Gamma = \langle z \mapsto 0 \rangle, \sigma = \langle 0 \mapsto \perp \rangle$

```
new x: int in
  x := 0;
  new y: int in
    y := 5;
    x := y + 3;
  new x: int = 0 in
    z := x + y;
  y := x;
```

```
new x: int = 0 in
  new y: int = 5 in
    x := y + 3;
  new x: int = 0; in
    z := x + y;
  y := x;
```

# Zusammenfassung

- ▶ Namen vs. (veränderliche) Variablen, L-Werte vs. R-Werte
- ▶ Variablen haben einen **Lebenszyklus**
  - ▶ Global/statisch, lokal/automatisch, Heap
- ▶ Lebenszeit  $\neq$  Sichtbarkeit
- ▶ Scope: Statisch vs. Dynamisch
- ▶ Deklarationen: sequentiell, kollateral, rekursiv
- ▶ Speicherverwaltung: Stack und Heap, Garbage Collection vs. manuell