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;

...
```

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

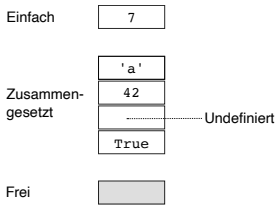
- Vergleiche:

```
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**

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

  - Keine Zuweisung, keine L-Werte
  - **Verwirrend**: Rust nutzt `let x = 37u32;` zur Deklaration von **Variablen**.

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)

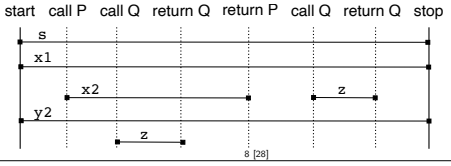
Block

- 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;
  ...
}
```



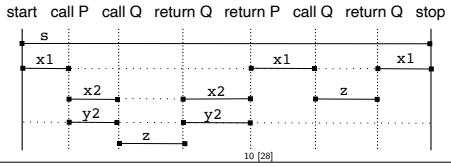
Bindung und Scope

- Eine **Bindung** assoziert 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;
  ...
}
```



Static vs. Dynamic Scope

Ein Beispielprogramm (fiktive Syntax):

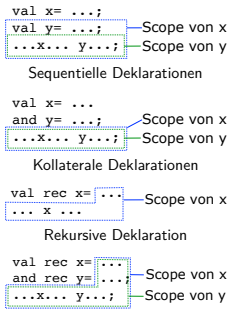
```
s= 2

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?



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 eine 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);
```

- Verliehene 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

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

- **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 $e_1[e_2/x]$:

$$v[e/x] = v$$
$$y[e/x] = \begin{cases} e & x = y \\ y & \text{otherwise} \end{cases}$$
$$(e_1 + e_2)[e/x] = (e_1[e/x]) + (e_2[e/x])$$
$$(x := e_1)[e_2/y] = (x := (e_1[e_2/y]))$$
$$(\textbf{let } x = e_1 \textbf{ in } e_2)[e_3/y] = \textbf{???}$$

Freie Variablen

$$\begin{aligned} \mathrm{FV}(v) &= \emptyset \\ \mathrm{FV}(x) &= \{x\} \\ \mathrm{FV}(e_1 + e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\ &\dots \\ \mathrm{FV}(i := e) &= \mathrm{FV}(e) \\ \mathrm{FV}(\textbf{let } x = e_1 \textbf{ in } e_2) &= \mathrm{FV}(e_1) \cup (\mathrm{FV}(e_2) \setminus \{x\}) \end{aligned}$$

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

Substitution

- Damit Definition der Substitution:

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

- z ist eine **frische** Variable.

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

Übung 3.3: Sei
$e \stackrel{def}{=} \textbf{let } x = x + y \textbf{ in } 4 * ((\textbf{let } y = x + y \textbf{ in } y + 5) + (\textbf{let } x = 3 * x * y \textbf{ in } y + x))$.
Berechne $e[3 * x + y/y]$.

Auswertung

- Zwei Möglichkeiten:

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

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

- (1) ist nicht-strikt ("call-by-name")
- (2) ist strikt ("call-by-value")

Variablen und Speichermodelle

Variablen und Speichermodelle

- Speichermodell $\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V}$ 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:

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

- Sequential, nicht kollateral
- Typisierung:

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

- Erweiterung der Semantik:

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

Γ ist eine **Umgebung** (statisch, nur zur Übersetzungszeit)

Semantik: alte Regeln

$$\frac{\Gamma \vdash \langle c_1, \sigma\rangle \to_{Stmt} \sigma' \qquad \Gamma \vdash \langle c_2, \sigma'\rangle \to_{Stmt} \sigma''}{\Gamma \vdash \langle c_1; c_2, \sigma\rangle \to_{Stmt} \sigma''}$$

$$\frac{\Gamma \vdash \langle b, \sigma\rangle \to_{Exp} \langle \mathit{true}, \sigma'\rangle \qquad \Gamma \vdash \langle c_1, \sigma'\rangle \to_{Stmt} \sigma''}{\Gamma \vdash \langle \textbf{if } (b) \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to_{Stmt} \sigma''}$$

$$\frac{\Gamma \vdash \langle b, \sigma\rangle \to_{Exp} \langle \mathit{false}, \sigma'\rangle \qquad \Gamma \vdash \langle c_2, \sigma'\rangle \to_{Stmt} \sigma''}{\Gamma \vdash \langle \textbf{if } (b) \textbf{ then } c_1 \textbf{ else } c_2, \sigma''\rangle \to_{Stmt} s}$$

$$\frac{\Gamma \vdash \langle b, \sigma\rangle \to_{Exp} \langle \mathit{false}, \sigma'\rangle}{\Gamma \vdash \langle \textbf{while } (b)\ c, \sigma'\rangle \to_{Stmt} \sigma'}$$

$$\frac{\Gamma \vdash \langle b, \sigma\rangle \to_{Exp} \langle \mathit{true}, \sigma'\rangle \qquad \Gamma \vdash \langle c, \sigma'\rangle \to_{Stmt} \sigma'' \qquad \Gamma \vdash \langle \textbf{while } (b)\ c, \sigma''\rangle \to_{Stmt} \sigma'''}{\Gamma \vdash \langle \textbf{while } (b)\ c, \sigma\rangle \to_{Stmt} \sigma'''}$$

Semantik: neue Regeln

$$\frac{x \in \textbf{Idt}, x \in \mathrm{dom}(\Gamma), \sigma(\Gamma(x)) = v}{\Gamma \vdash \langle x, \sigma\rangle \to_{Exp} \langle v, \sigma\rangle}$$

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

$$\frac{\Gamma[x \mapsto l] \vdash \langle c, \sigma[l \mapsto \bot]\rangle \to_{Stmt} \sigma' \qquad l \notin \mathrm{dom}(\sigma)}{\Gamma \vdash \langle \textbf{new } x : t \textbf{ in } c, \sigma\rangle \to_{Stmt} \sigma' \setminus l}$$

Beispiel

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

```
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