Programmiersprachen

Vorlesung 3 vom 01.11.21

Anweisungen, Variablen und Speicher

Christoph Lüth

Universität Bremen

Wintersemester 2021/22

Wo sind wir?

- Einführung
- Werte und Typen
- Anweisungen, Variablen und Zustand
- Kontrollabstraktion
- Datenabstraktion
- Fortgeschrittene Typsysteme
- Nebenläufigkeit
- Objektorientierung
- Skriptsprachen
- Beispielsprache II
- Ab Woche 11: Studentische Vorträge.

Ausdrücke und Anweisungen

Fundamentale Ausdrücke

- Literale
- Konstruktoren
- Selektoren
- Funktionsaufrufe
  - Bedingte Ausdrücke
  - Iterative Ausdrücke
  - Variablen

Literale

- Denotieren Werte der **primitiven** Typen
- Ganze Zahlen, Fließkommazahlen, Hexadezimal- und Oktalzahlen
  - Haskell hat **überladene** Literale
- Zeichenketten
  - Notation für nicht-druckende Zeichen: `\n`, `\t` etc.
  - Lange (zeilenübergreifende) Strings, e.g.

```
"""Ein
ganz langer
String."""
```

Konstruktoren

- Konstruieren Werte **zusammengesetzter** Typen
- Tupel werden als `(3, True, "Foo")` konstruiert
  - Außer in C
- Arrays: meist nur bei der Initialisierung:

```
int a[] = {3, 7, 9};
```

- Dictionaries in Python:

```
d = { 3 : "Three", 5 : "Five", 7 : "Seven" }
```

- Konstruktoren in C: Speicherallokation
- Konstruktoren in Java, Python: Objektinitialisierung

Selektoren

- Zugriff auf Komponenten zusammengesetzter Typen
- Rechtsinvers zum Konstruktor
- Für Tupel meist **nicht** verdefiniert
- Feldselektion in C, Java, Python (`x.foo`)
- Optional definiert in Haskell (`data X = C { sel :: ...}`)
- Array access in C und Java
  - Überladen in Python

Funktionsaufrufe

- Vordefinierte Funktionen:
  - arithmetische Operationen
  - Relationen
  - Boolsche Operationen
- Fallunterscheidung (als Ausdruck)

```
x == y ? "Gleich" : "Ungleich"
```

- Iteration
  - Haskell und Python kennen Listenkomprehension

```
[ str(x) for x in range(3,27,3) ]
```

  - Syntaktischer Zucker für `map`, `filter`, `concat`.
- C: explizite Referenzierung/De-Referenzierung (`&`, `*`)
- Methodenaufrufe und selbstdefinierte Funktionen → **später**

Striktheit

- Eine Funktion ist **strikt** (in einem Argument), wenn das Ergebnis undefiniert ist, sobald das Argument undefiniert ist.
- Striktheit erlaubt es, Argumente **vor** dem Aufruf auszuwerten.
- Die meisten Sprachen sind strikt (C, Java, Python), **aber**:
  - Fallunterscheidung ist **nie** strikt.
  - Logische Konjunktion (`&&`) und Disjunktion (`||`) nicht-strikt im zweiten Argument
- Haskell ist (natürlich) nicht-strikt

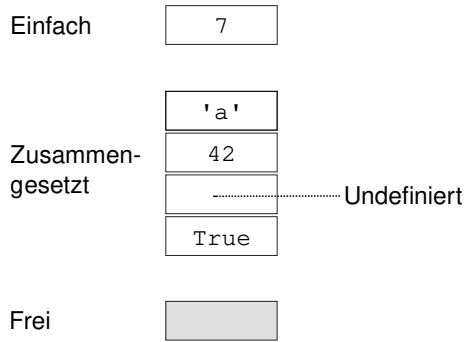
Einfache Anweisungen:

- Kernsprache:
  - Zuweisung
  - Sequenzierung und leere Anweisung
  - Fallunterscheidung
  - Iteration
    - `while`, `repeat`, Rekursion
  - Turing-mächtig
- Sprünge: `goto` etc. — considered harmful
- Manche Sprachen unterscheiden Ausdrücke und Anweisungen nicht
  - In C sind Zuweisungen Ausdrücke
  - In Haskell ist alles ein Ausdruck

Variablen und Speicher

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 entweder einen **Inhalt**, entweder ein **Wert** oder **undefiniert**.
- Zusammengesetzte Werte belegen mehrere Speicherzellen (“composite variables”)
- Abstraktion über Wortbreite etc.



Einfache Variablen

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

```
x= x+1
```

- In funktionalen Sprachen sind Variablen **unveränderlich**
  - Unterschied entfällt, Optimierungspotenzial

Zusammengesetzte Variablen

- Zusammengesetzte Variablen belegen einen **Block**
- Bei Feldern zusammenhängend
- Bei Tupeln nicht notwendigerweise
- Speicherlayout und alignment
  - Nur für systemnahe Programmiersprachen (e.g. C)
- Totales und selektives update

```
struct date { int y, m, d; } d1, d2;
d1.m= 11; // selektiv
d2= d1;   // total
```

- C erlaubt **Speicherarithmetik**: `a[i] == *(a+i)`

Felder

- Statische Felder haben **feste**, **unveränderliche** Länge (C, Java)
- Bei **dynamischen** Felder kann die Länge verändert werden (Haskell, Ada, `Vector` in Java)
- Bei **flexiblen** Feldern ist die Länge variabel (aber fest)

```
double a1[] = {2.0, 3.0, 5.0};

static void prtVec(double [] v) {
   for (int i= 0; i< v.length; i++)
      System.out.println(v[i]+" ")
}
```

Copy Semantics vs Reference Semantics

- Was passiert bei einer Zuweisung `x= e`, wenn `x` einen zusammengesetzten Typ hat?
- **Copy semantics**: `x` enthält danach eine **Kopie** von `e`, alle Komponenten von `e` werden in die Komponenten von `x` kopiert
- **Reference semantics**: `x` ist eine **Referenz** auf `e`

Copy Semantics vs Reference Semantics

- Was passiert bei einer Zuweisung `x= e`, wenn `x` einen zusammengesetzten Typ hat?
- **Copy semantics**: `x` enthält danach eine **Kopie** von `e`, alle Komponenten von `e` werden in die Komponenten von `x` kopiert
- **Reference semantics**: `x` ist eine **Referenz** auf `e`
- C kopiert (Referenzen sind in der Sprache **explizit**)
- Java und Python referenzieren (alles ist eine Referenz, Kopie explizit über `clone`, `copy`, `deepcopy`)
- Haskell referenziert, aber Werte sind **unveränderlich**

Verwandt damit: Gleichheit

- Identität vs. strukturelle Gleichheit
- Identität: Referenz auf das gleiche Objekt im **Speicher**
- Strukturelle Gleichheit: gleicher “Inhalt”
  - Java: == für Identität (der Referenzen), `equals` für strukturelle Gleichheit
  - Python: `is` für Identität (der Referenzen), == für strukturelle Gleichheit
  - C: == auf Referenzen für Identität, == auf zusammengesetzten Typen für strukturelle Gleichheit
  - Haskell: **nur** strukturelle Gleichheit (==, Typklasse `Eq`)

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)

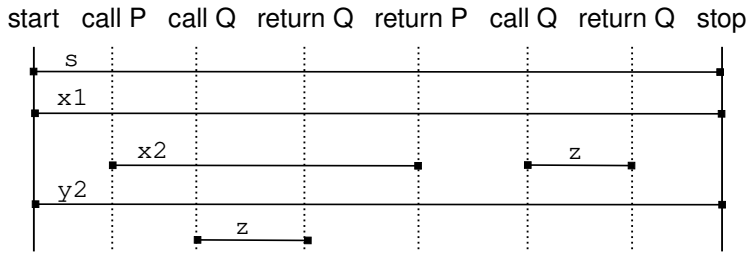
Block

- 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

Beispiel

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

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



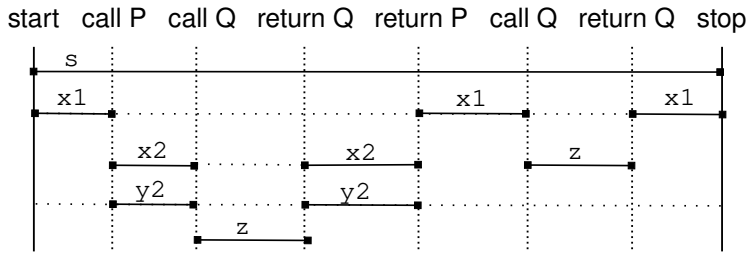
Bindung und Scope

- 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

```
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):

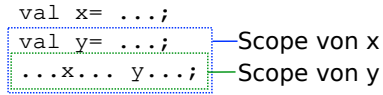
```
s= 2

int foo(x)
  return s*x;

void baz(y)
  print(foo(y))

void bah(y)
  local s= 4
  print (foo(y))

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`

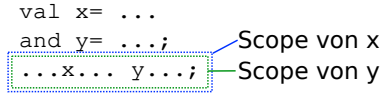
Deklarationen

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

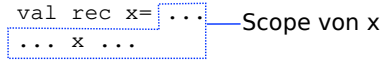
Frage 3.1: Wie werden Deklarationen in C, Java, Python, Haskell gehandhabt?



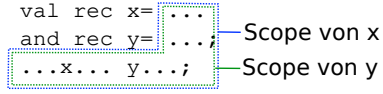
Sequentielle Deklarationen



Kollaterale Deklarationen



Rekursive Deklaration



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

Zusammenfassung

Zusammenfassung

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