Programmiersprachen
Vorlesung 1 vom 21.10.21
Einführung

Christoph Lüth

Universität Bremen

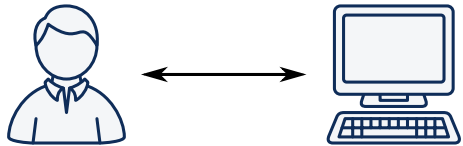
Wintersemester 2021/22

Einführung

Worum geht es?

- Es gibt über 700 Programmiersprachen (sagt Google)
- Wie kriegen wir da Ordnung rein?
- Zugrundeliegende Prinzipien
  - Was haben alle Programmiersprachen gemein?
  - Wo gibt es Unterschiede?
  - **Taxonomie** der Programmiersprachen
- Neue Programmiersprachen lernen (neue, alte, merkwürdige)

Warum Programmiersprachen?



- Wollen Programme in **verständlicher** Notation aufschreiben
- Maschine soll sich dem Menschen anpassen (nicht umgekehrt)
- Programme müssen **maschinenlesbar** und **auführbar** bleiben
- **Modellbildung** und **Abstraktion**

Konzept der Veranstaltung

- Vorlesung:
  - Montag um 10:00, online
    - Zoom: `https://uni-bremen.zoom.us/j/93767566115?pwd=QThKSlR0ckVLdUpGc3BqMm5wNTN6Zz09`
  - Dazu Übungsblatt
- Übungen:
  - Werden bis Donnerstag bearbeitet
  - Lösungen werden am Donnerstag 8-10 in der Übung vorgestellt
  - Dazu "Musterlösung" vom Veranstalter
  - Werden **nicht korrigiert**
- Referate:
  - Ab Woche 10/11
  - Studierende stellen je **eine** neue Sprache vor

Scheinbedingungen

- Referat über eine neue Sprache
- Mündliche Prüfung am Ende

Grundlagen

Was ist eine Programmiersprache?

1. Definierte, maschinenlesbare **Syntax**
2. Mathematisch, informell oder pragmatisch definierte **Semantik**
3. Die Sprache muss **ausführbar** und **Turing-mächtig** sein

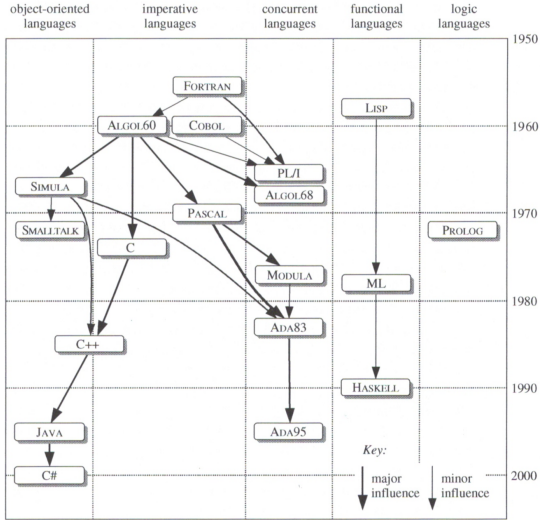
Arten von Programmiersprachen

- Programmiersprachen sind immer **Abstraktionen** über einem Berechnungsmodell.
- **Imperativ**: Zustandsübergänge auf einem Speicher (Turing-Maschine)
  - Abstraktion durch Datentypen
  - Abstraktion durch Verkapselung
- **Funktional**: Rekursive Funktionen (Auswertung von Ausdrücken)
- Sonstige: logische, domänenspezifisch

Scope dieser Veranstaltung — was machen wir nicht?

- Aspekte der Ausführung: wie **implementieren** wir eine Programmiersprache
  - Compilerbau, Übersetzer, abstrakte Maschinen, ...
- Aspekte der Syntax
  - Grammatiken und formale Sprachen, Parsergeneratoren, Lexer, ...
- Formale Semantik
  - Mathematische Beschreibung der Semantik, semantische Rahmenwerke, Typentheorie

Historisches: Stammbaum einiger Programmiersprachen



Welche Sprachen betrachten wir?

- Laufende Beispiele:
  - C
  - Java
  - Python
  - Haskell
- Weitere in den Referaten

Liste weiterer Sprachen

- Systemnah: Rust
- Logische Programmierung: Prolog, Oz
- Dynamisch: JavaScript
- Nebenläufig/Reaktiv: Erlang, Golang
- Abhängige Typen: Idris, Agda, Liquid X (Dependent types)
- Prozedural: Julia, Kotlin, Swift
- Skriptsprachen: Lua, Tcl, sh/bash
- Funktional[1]: SML, OCAML, Elm, Clojure, LISP, Scala
- Stack-basiert: Forth
- Historisch: COBOL, Algol-68, APL, Ada, Smalltalk
- Datenflusssprachen: Id, Lucid, Lustre
- DSLs: R, SQL, Postscript, TeX, Verilog/VHDL, SystemC, SpinalHDL

[1] Optional

Struktur der Veranstaltung

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

Literatur und Basis

- David A. Watt: **Programming Language Design Concepts**, John Wiley & Sons, 2004.
- Maurizio Gabbrielli, Simone Martini: **Programming Languages: Principles and Paradigms**. Springer, 2010.
- Robert W. Sebesta: **Concepts of Programming Languages**. Pearson Education, 2016.

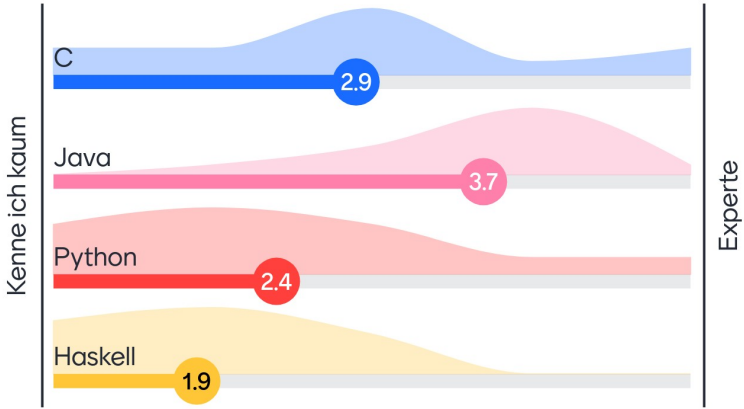
Vorkenntnisse

Online-Umfrage: `https://www.menti.com/veswebppdp`

- Vorkenntnise in folgende Sprachen:
  - C
  - Java
  - Python
  - Haskell
- Welche weiteren Sprachen kennt ihr?

Vorkenntnisse

Mentimeter



12

Welche anderen Sprachen kennt ihr?

Mentimeter

12

Zum Abschluss

Eine einfache Funktion

In den Programmiersprachen C, Haskell, Java, Python:
Schreibe eine Funktion (Methode), welche als Argument eine Zeichenkette bekommt, und zählt, wie oft die Zeichen `x`, `y` und `z` (egal, ob groß oder klein) auftreten.

Zum Abschluss

Eine einfache Funktion

In den Programmiersprachen C, Haskell, Java, Python:
Schreibe eine Funktion (Methode), welche als Argument eine Zeichenkette bekommt, und zählt, wie oft die Zeichen `x`, `y` und `z` (egal, ob groß oder klein) auftreten.

Nächster Termin

- Montag, 25.10.2020 um 10 ct auf Zoom.

Programmiersprachen

Vorlesung 2 vom 25.10.21

Werte und Typen

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

- Viele Sprachen unterscheiden **Ausdrücke** und **Anweisungen**
  - Ausdrücke bezeichnen die zu manipulierenden **Werte**
  - Anweisungen beschreiben **Kontrollfluß**
  - **Imperatives** Konstrukt: Programm ist abstrakte Sicht auf Speichermanipulation
- Funktionale Sprachen, logische Sprachen uvm. haben diese Unterscheidung **nicht**.

Werte

- Was sind Werte?
  "Any entity that can be manipulated by a program."
  - **Primitive** Werte:
    Booleans, ganze Zahlen, Fließkommazahlen, Adressen (Pointer, Referenzen)
  - **Zusammengesetzte** Werte (composite values):
    Strukturen, Felder, Listen, Objekte, ...
- Semantisch gesehen:
  - Abstraktionen über dem Speicherinhalt
  - Nichtreduzierbare Ausdrücke

Typen

Typen

- Was sind Typen?
  - "A set of values." (Mengen von Werten)
  - Durch die Operationen darauf charakterisiert. z.B. $\{13, \textit{Monday}, \textit{true}\}$ ist kein Typ.
    — kann man drüber streiten
- Arten von Typen:
  - **Primitive** Typen (primitive types)
  - **Zusammengesetzte** Typen (composite types)
- Typüberprüfung und Typsysteme

Vorgegebene Primitive Typen

- Anwendungsabhängig: COBOL vs. FORTRAN; BCPL vs. C vs. TEX
- Ganze Zahlen:
  - Java: `int`
  - C: `char`, `short`, `int`, `long`, `long long`, etc.
  - Haskell: `Int`, `Natural`, `Integer`
- Booleans:
  - In C **kein** expliziter Typ (`bool_t` ist `int`)
- Fließkommazahlen: `float` und `double`

Frage 2.1: Welche Wortbreite haben ganze Zahlen in C, Java, Python, Haskell?

Selbstdefinierte Primitive Typen

- C, Java, Haskell erlauben **Aufzählungen:**

```
enum colour {red, green, blue}
```

- In C nur syntaktischer Zucker für `int`, kein separater Typ.
- In Java: Klasse mit konstanter Anzahl von Objekten (s. hier)
- In Haskell: algebraischer Typ mit ausschließlich konstanten Konstruktoren.

```
data Colour = Red | Green | Blue
```

- Ada erlaubt Untermengen von `int`:

```
type Cent is range 0 .. 99;
```

Zusammengesetzte Typen

- Cartesische Produkte (Tupel und Strukturen)
- Endliche Abbildungen (Arrays, Dictionaries)
- Disjunkte Vereinigung (algebraische Typen, discriminated records, Objekte)
- Rekursive Typen

Cartesische Produkte

- Für zwei Typen A und B sind **Tupel** der Typ $A \times B$ mit folgenden **Eigenschaften:**

1. Es gibt es zwei Funktionen $\pi_1 : A \times B \to A, \pi_2 : A \times B \to B$.
2. Für zwei Funktionen $f : C \to A$, $g : C \to B$ gibt es **genau eine** Funktion $\langle f, g \rangle : C \to A \times B$ so dass $\pi_1 \cdot \langle f, g \rangle = f$ und $\pi_2 \cdot \langle f, g \rangle = g$

- Verallgemeinerung auf n Komponenten:

$$A_1, \ldots, A_n \quad \text{und} \quad \prod_{i=1,\ldots,n} A_i$$

- Sonderfall: Tupel der Länge 0

$$T = 1$$

  - Unit-Typ, in vielen Sprachen nur implizit (Java, C).

Cartesische Produkte

- Fast alle Programmiersprachen haben cartesische Produkte
- Entweder direkt als Tupel (Haskell, Scala) oder
- `struct` sind cartesische Produkte mit **benannten** Projektionen

```
struct pair {
   int fst;
   int snd;
}
```

- Tupel sind “Listen fester Länge”

Frage 2.2: Wieso gibt es dann keine Funktion, um diese Listen aneinanderzuhängen?

Tupel und Funktionen

- Ungenaue Notation bei Funktionsaufruf:

Haskell:

```
f1 :: Int → Int → Int
f2 :: (Int, Int) → Int
```

`f1` hat zwei Argumente, `f2` eines (ein Tupel)

C:

```
int f(int x, int y)
```

`f` hat **zwei** Argumente, nicht eins.

- Es gilt $A \times B \to C \cong A \to B \to C$ ("Currying")
- Wird nicht von allen Programmiersprachen unterstützt
- $A \to B \to C$ ist eine Funktion höherer Ordnung

Endliche Abbildungen

- Eine endliche Abbildung ist eine Funktion $A \to B$ mit A endlich.
- Dargestellt als linkseindeutige und rechtstotale Relation

$$A \to B \subseteq A \times B$$

- Wenn $A = 0, \ldots, n$ für $n \in \mathbb{N}$, dann ist $A \to B$ ein **Feld** (Array)
  - Felder können effizient als zusammenhängende Speicherbereiche implementiert werden
  - Indizierung ($a[i]$) sehr billig $O(1)$
  - Manchmal Indizierung auch ab 1 oder von $n \ldots m$
  - Mehrdimensionale Felder durch Felder von Feldern (C, Java) oder Indizierung mit Tupeln (Haskell)
- Wenn A eine Menge von **Bezeichnern**, dann ist $A \to B$ ein **Dictionary** (Python)
  - Im Unterschied zu `structs` sind Dictionaries in Python zur Laufzeit definierbar und erweiterbar

Disjunkte Vereinigung

Für zwei Typen A und B ist die **disjunkte Vereinigung** der Typ $A + B$ mit folgenden **Eigenschaften:**

1. Es gibt zwei Funktionen $in_1 : A \to A + B$ und $in_2(b) : B \to A + B$.
2. Für zwei Funktionen $f : A \to C$, $g : B \to C$ gibt es **genau eine** Funktion $[f, g] : A + B \to C$ so dass $[f, g] \cdot in_1 = f$ und $[f, g] \cdot in_2 = g$.

Verallgemeinerung auf n Komponenten:

$$A_1, \ldots, A_n \quad \text{und} \quad \coprod_{i=1,\ldots,n} A_i$$

- Disjunkte Vereinigungen werden sehr **heterogen** gehandhabt.
- Sonderfall: leere Vereinigung — der **leere** Typ

$$T = \emptyset$$

  - `void` in C und Java, `Nothing` in Scala; in Haskell nicht vordefiniert und nutzlos

Frage 2.3: Warum nutzlos? Warum ist ‘void‘ in C eigentlich semantisch falsch?

Disjunkte Vereinigung in C

- Der Union-Typ vereinigt alle Komponenten an der gleichen Adresse:

```
union { int x; double y; } u;
```

- Extrem fehleranfällig
- Keine **disjunkte** Vereinigung
- Zur Unterscheidung (discriminated records in Ada):

```
enum u_tag { u_a, u_b };
struct { enum u_tag tag; union { int a; double b; } cont; } u;

switch (u.tag) {
  case u_b: printf("Double: %f\n", u.cont.y); break
}
```

Disjunkte Vereinigung in Java

- Sehr indirekt durch Subtyping:

```
class A {
  C f() { ... };
}
class B {
  C g() { ... };
}
```

```
abstract class AB {
   private class InlA extends AB
     private A a;
     C fg() { a.f(); }
   }
   private class InrB extends AB
     private B b;
     C fg() { b.g(); }
   }
   C fg();
}
```

Disjunkte Vereinigung in Haskell

- Algebraische Datentypen (mit Fallunterscheidung):

```
data A
f :: A → C

data B
g :: B → C

data AB = Inl A | Inr B

fg :: AB → C
fg ab = case ab of Inl a → f a; Inr b → g b
```

Rekursive Typen

- Rekursive Typen sind durch **Gleichungen** definiert:

$$T = F(T)$$

- Beispiele:
  - Listen: $L(A) = 1 + A \times L(A)$
  - Binäre Bäume: $T(A) = 1 + T(A) \times A \times T(A)$
  - Variadische Bäume: $R(A) = A \times L(R(A))$

Frage 2.4: Wieso definieren diese Gleichungen den Typ?

Rekursive Typen in C

- C erlaubt **keine** direkt rekursiven Typen
  - Umweg über Zeiger und “incomplete types”:

```
typedef struct list_el {
    void *head;
    struct list_el *tail;
    } *list;

typedef struct tree_el {
  struct tree_el *le; void *node; struct tree_el *ri;
  } *tree;
```

- Der `NULL`-Pointer übernimmt die Rolle des Unit-Typen.
- Polymorphie durch `void *`.

Rekursive Typen in Java

Rekursive Typen durch rekursive Klassen:

```
class List {
  public Object head;
  public List tail;
  }
```

```
class Tree {
  public Tree left;
  public Object node;
  public Tree right;
}
```

- In Java ist alles[1] **implizit** eine Referenz (Pointer), daher eigentlich ähnlich C einschließlich `null` für den Unit-Typ.
- Polymorphie durch `Object`, mehr dazu später.

[1] Bis auf primitive Typen.

Rekursive Typen in Haskell

- Hier können wir die Domänengleichungen direkt abschreiben:

```
data List a = Null | Cons a (List a)

data Tree a = Node { le :: Tree a, node :: a, ri :: Tree a}

data NTree a = Node a (List (Ntree a))
```

- Listen sind mit syntaktischem Zucker vordefiniert.

Rekursive Typen in Python

- Listen sind vordefiniert (Typ `list`)
- Definition von rekursiven Typen nicht direkt möglich, nur als Klasse.
- Binäre Bäume:

```
class Tree:
  def __init__(self, node):
    self.left = None
    self.right = None
    self.node = node
```

```
class NTree:
  def __init__(self, node):
    self.node= node
    self.children= []
```

- `__init__` ist der Konstruktor, der Typ selbst wird dynamisch definiert.

Zeichenketten

- Zeichenketten (Strings) spielen **meist** eine Sonderrolle
- Konzeptionell: Array oder Liste von Zeichenketten, e.g. C und Haskell:

```c
char txt1[] = "Foo";
char txt2[4] = {'F', 'o', 'o', 0};
```

```haskell
type String = [Char]
```

- Aus Effizienzgründen: **Unveränderliche** Strings
  - Java und Python
  - `bytestring` in Haskell

Typsysteme

Typsysteme

- Was gibt es für Typen?
- Wie überprüfen wir Typen?
  - Statisch vs. dynamisch
- Gleichheit von Typen
- Anforderung an ein Typsystem:
  - Entscheidbarkeit
  - Effizienz

Typüberprüfung

- **Statische** Typsysteme:
  - Typen werden zur Compile-Zeit geprüft.
  - Zur Laufzeit werden keine Typinformationen benötigt (“type erasure”)
  - Sicher und effizient, aber unflexibel
  - Bsp: C, Haskell, Java
- **Dynamische** Typsysteme
  - Typen werden zur Laufzeit geprüft.
  - Flexibel, aber fehleranfällig.
  - Bsp: Python, Java (dynamische Bindung von Methoden)

Typäquivalenz

- Grundsätzliche Frage: wann ist $T_1 \cong T_2$? (Wichtig für die Typüberprüfung)
- **Nominal**: wenn sie den gleichen Namen haben.
- **Strukturell**: wenn sie die gleiche Struktur haben.

```
typedef struct a {int a;
                  double d;
                 } t1;
void f(t1 x) {
  printf("Double is %f\n", x.d);
}
```

```
typedef struct b {int a;
                  double d;
                 } t2;
void g(t2 y) {
  f(y);
}
```

Typäquivalenz konkret

- C, Haskell, Java nutzen nominale Typäquivalenz
  - `typedef` in C, `type` in Haskell sind **Typsynonyme**, definiert keinen neuen Typ
- Python nutzt (schwächere Form der) strukturellen Typäquivalenz
  - “Duck typing”

Implizit vs. explizit

- Explizit oder manifeste Typsysteme: alle Typen werden **explizit** angegeben
  - Compiler muss nur prüfen
  - Beispiel: Java
- Implizite Typsysteme: Typen werden abgeleitet
  - Compiler muss Typ inferieren
  - Beispiel: Haskell, Hindley-Milner-Typsystem
  - Problem: Entscheidbarkeit, bspw. mit Subtyping unentscheidbar

Frage 2.5: Ist Python implizit oder explizit?

Zusammenfassung

Zusammenfassung

- Werte sind ...
- Typen sind **Mengen von Werten**
- Primitive Typen: Zahlen, Zeichen
- Zusammengesetzte Typen:
  - Tupel, endliche Abbildungen, disjunkte Vereinigung
  - Rekursive Typen
- Typsysteme:
  - Statisch vs. dynamisch
  - Nominal vs. strukturell
  - Implizit vs. explizit

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

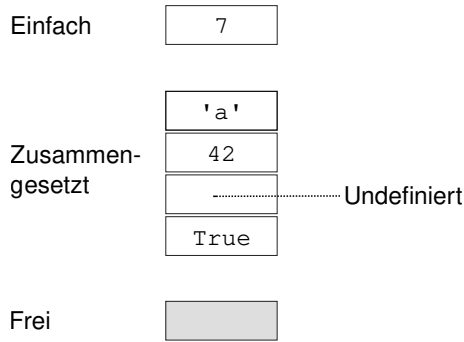
Einfache Anweisungen:

- 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)

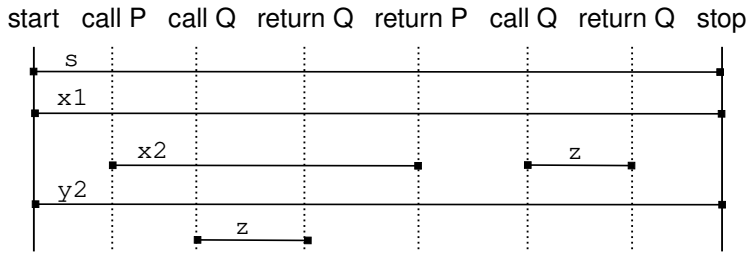
Block

- 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;
  ...
}
```



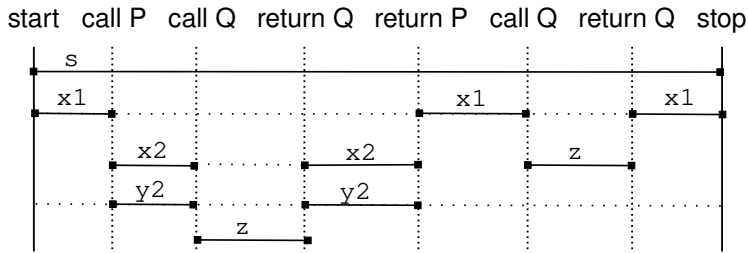
Bindung und Scope

- 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):

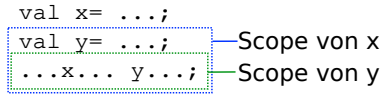
```
s= 2

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`

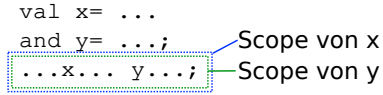
Deklarationen

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

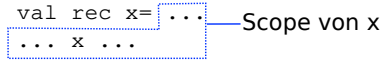
Frage 3.1: Wie werden Deklarationen in C, Java, Python, Haskell gehandhabt?



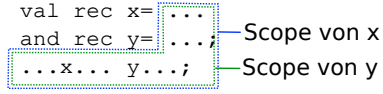
Sequentielle Deklarationen



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

Programmiersprachen

Vorlesung 4 vom 08.11.21

Kontrollabstraktion

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.

Abstraktion

- Definition Wikipedia:

> Das Wort Abstraktion[a] bezeichnet meist den [...] Denkprozess des erforderlichen Weglassens von Einzelheiten und des Überführens auf etwas Allgemeineres oder Einfacheres. Daneben gibt es spezifische sowie unspezifische Verwendungen des Begriffes in bestimmten Einzelwissenschaften und einzelnen Theorien, Thesen sowie Behauptungen.
>
> [a] lat. *abstractus* "abgezogen", Partizip Perfekt Passiv von abs-trahere "abziehen", "trennen"

- Weiter: "In der Mathematik [...] werden Abstrakta meist mit Äquivalenzklassen identifiziert."
- Im Lambda-Kalkül ist Abstraktion $\lambda x.\ t$ — die Einführung des Funktionsparameters

Abstraktion

- Kontrollabstraktion (heute):
  - Jenseits der direkten Auswertung
  - Prozeduren und Parameter
  - Sprünge
- Datenabstraktion (nächstes Mal):
  - Abstrakte Datentypen
  - Verkapselung und Objekte
  - Module
  - Packages

Prozeduren

Prozeduren und Funktionen

- **Prozeduren** sind benannte, parameterisierte **Blöcke**
  - Meist ohne Rückgabewert
- **Funktionen** sind Prozeduren mit Rückgabewert
  - **Reine** Funktionen (pure functions): referentiell transparent, ohne Seiteneffekt
  - Meist **mit** Seiteneffekten
- Viele Programmiersprachen unterscheiden das nicht
- Funktionsdefinition hat **(formale) Parameter**, beim Aufruf **Parameterwerte** (Argumente)

```
int f(x) { return x*10; }   // ‘x‘ ist formaler Parameter

... f(29+2) ... // ‘29+2‘ ist Parameterwert
```

Parameterübergabe (Parameter Passing)

- Aus **konzeptioneller** Sicht gibt es drei Arten von Parametern:
  - **Eingabeparameter** — erlaubt Kommunikation vom Aufrufer an die Funktion
  - **Ausgabeparameter** — erlaubt Kommunikation von der Funktion an den Aufrufer
  - **Ein/Ausgabeparameter** — erlaubt bidirektionale Kommunikation

Parameterübergabe (Parameter Passing)

- Aus **konzeptioneller** Sicht gibt es drei Arten von Parametern:
  - **Eingabeparameter** — erlaubt Kommunikation vom Aufrufer an die Funktion
  - **Ausgabeparameter** — erlaubt Kommunikation von der Funktion an den Aufrufer
  - **Ein/Ausgabeparameter** — erlaubt bidirektionale Kommunikation
- Beispiel (Ada; Eingabeparater `v`, `w`, Ausgabeparameter `sum`)

```
type Vector is array (1 .. n) of Float;

procedure add (v, w: in Vector; sum: out Vector) is
  begin for in 1 .. n loop
    sum(i) := v(i) + w(i);
  end loop
```

Parameterübergabe (Parameter Passing)

- Aus **konzeptioneller** Sicht gibt es drei Arten von Parametern:
  - **Eingabeparameter** — erlaubt Kommunikation vom Aufrufer an die Funktion
  - **Ausgabeparameter** — erlaubt Kommunikation von der Funktion an den Aufrufer
  - **Ein/Ausgabeparameter** — erlaubt bidirektionale Kommunikation
- Beispiel (Ada; Eingabeparater `v`, `w`, Ausgabeparameter `sum`)

```
type Vector is array (1 .. n) of Float;

procedure add (v, w: in Vector; sum: out Vector) is
  begin for in 1 .. n loop
    sum(i) := v(i) + w(i);
  end loop
```

- Aus **operationaler** Sicht gibt verschiedene Arten der **Parameterübergabe**

Call by Value

- **Parameterwert** ist **beliebiger** Ausdruck (R-Wert)
- **Funktionsaufruf**:
  - Parameter wird zu v ausgewertet
  - Formaler Parameter wird lokale Variable im Funktionsrumpf, mit v initialisert
  - Funktionsrumpf wird ausgeführt
- Für **Eingabeparameter**
- Klare Semantik (kein Effekt auf Aufrufer)
- Effizient für "kleine" v, ineffizient für große Datenstrukturen (Felder etc.)
- Wertet eventuell zu viel aus

Call by Reference (Call by Variable)

- Parameterwert muss ein L-Wert sein
- Funktionsaufruf:
  - Umgebung des Funktionsrumpfes wird erweitert
  - Formaler Parameter wird zu L-Wert aufgelöst (aliasing)
  - Funktionsrumpf wird ausgeführt
- Für **Ausgabeparameter** und **Ein/Ausgabeparameter**
- Funktion kann Parameterwert verändern
- Effizient aber fehleranfällig (wegen Aliasing)

```
void foo(reference int x)
{ x= x+1; }

char V[10];
i= 2;
V[2]= 5;
foo(V[i]);
// V[2] == 6
```

Call By Name

- Parameterwert ist beliebiger Ausdruck `e` (R-Wert)
- Aufruf von Funktion `f(x)` ist semantisch äquivalent zur Ausführung des Rumpfes, in dem alle `x` durch `e` ersetzt werden.
- Semantisch sauber, aber subtil
- Stammt von ALGOL-60, wird heute nur noch wenig benutzt

```
int x= 0;
int foo(name int y)
{
  int x= 2;
  return x+y;
}
...
int a= foo(x+1);
  // = {int x= 2; x+ x+ 1} ???
```

Jensen's Device

- Call-by-Name erlaubt **Metaprogrammierung** (Macros)
- Beispiel: Jensen's Device

```
int sum(name int exp; name int i; int fr; int to)
{
  int acc= 0;
  for (i= fr; i<= to; i++) acc= acc+ exp;
  return acc;
}

int x= ...;
int y= sum(2*x*x- 2*x+ 1, x, 1, 10)
```

- Berechnet

$$y = \sum_{x=1}^{10} 2x^x - 2x + 1$$

Variationen

- Call by constant:
  - Wenn Funktionsrumpf den formalen Parameter nicht modifiziert kann call-by-value durch call-by-reference implementiert werden.
- Call by need (Haskell):
  - Ähnlich call-by-name, Parameterwert wird **nur** ausgewertet, wenn er benutzt wird
- Call by value mit Zeigern (C, Java, Python)
  - Wenn Werte Zeiger (Referenzen) sind kann der Aufruf Seiteneffekte haben
  - Parameter vom Typ Pointer (C) oder `Object` (Java, Python) sind Ein/Ausgabe-Parameter

Parameterübergabe in C

C-Standard (C99, §6.5.2.2)

1. The expression that denotes the called function[77] shall have type pointer to function returning void or returning an object type other than an array type.
4. An argument may be an expression of any object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.[a]

[a] A function may change the values of its parameters, but these changes cannot affect the values of the arguments. On the other hand, it is possible to pass a pointer to an object, and the function may change the value of the object pointed to. A parameter declared to have array or function type is adjusted to have a pointer type as described in 6.9.1.

Parameterübergabe und Auswertungsstrategie

- Auswertungsstrategien in funktionalen Sprachen:
  - Innermost-first
  - Outermost-first
- Innermost-first $\sim$ call-by-value, eager evaluation
- Outermost-first $\sim$ call-by-need, lazy evaluation
- Outermost-first: nicht-strikt

```
f 7 undefined  ⇝  14
```

Beispiel:

```
f x y = x+ x
```

Auswertung innermost:

```
    f (f 7 3) (f 5 9)
⇝   f (7+7) (5+ 5)
⇝   f 14 20
⇝   14+14  ⇝  28
```

Auswertung outermost:

```
    f (f 7 3) (f 5 9)
⇝   f 7 3+ f 7 3
⇝   (7+ 7)+ (7+ 7)
⇝   14+14  ⇝  28
```

Funktionen höherer Ordnung

Funktionen Höherer Ordnung

- Funktionen höherer Ordnung sind Funktionen $A \to B$ mit A oder B eine Funktion.
- Funktion als Argument, Beispiel (Python):

```
map(str, [1, 18, true, "foo"])
```

- Funktion als Resultat, Beispiel `3 <=` (vom Typ `Int-> Bool`) in (Haskell):

```
filter (3 ≤) [0,7,1,8,2,9,-2]
```

- Dabei hilfreich: "anonyme" Funktionen (Lambda-Ausdrücke), Beispiel (Python):

```
filter (lambda x: 3<= x, [0,7,1,8,2,9,-2])
```

- Komplikationen: Scoping
- Python und besonders Haskell unterstützen Funktionen höher Ordnung

Funktionen höherer Ordnung in C

- Auch C unterstützt Funktionen höherer Ordnung — durch Zeiger
- Beispiel:

```
typedef struct list_t {
     void          *elem;
     struct list_t *next;
     } list_t;
extern list_t *filter(int f(void *x), list_t *l);
extern list_t *map(void *f(void *x), list_t *l);
```

- Problem: Speicherverwaltung, Typsystem nicht expressiv genug
- Wird genutzt für Sprungtabellen, Signalhandler, Callbacks.

Exceptions

Ausnahmen (Exceptions)

- Motivation:
  - Fehlerbehandlung in geschachtelten Funktionen
  - Ohne Sprünge nur umständliche Fallunterscheidungen

Ausnahmen (Exceptions)

- Motivation:
  - Fehlerbehandlung in geschachtelten Funktionen
  - Ohne Sprünge nur umständliche Fallunterscheidungen
  - Nicht alle Fehlermöglichkeiten **können** im Vorfeld ausgeschlossen werden
  - Klassisches Beispiel: Dateizugriff

```
if os.path.exists(filenm):
  # someone deletes filenm
  fd= open(filenm) # FEHLER!
```

```
int main() {
  fd= open("data");
  ... P(fd); ...;
  close(fd);
  }

void P(int fd) { ... Q(fd); ... }

void Q(int fd) { ... R(fd); ... }

void R(int fd) {
  if ((x= read(fd, 1024))== -1)
    // Fehler!
  ...
  }
```

Ausnahmen

- Konzeptionell: Ausnahmen sind **Erweiterung des Definitionsbereichs**:

$$\texttt{f} : A \to B \texttt{ throws } C \;=\; \texttt{f} : A \to B + C$$

$$B[f(x)] \texttt{ catch } e \Rightarrow E \;=\; \begin{cases} B[b] & f(x) = b \\ E & f(x) = e \end{cases}$$

- Unterstützung von Ausnahmen durch eine Programmiersprache benötigt:
  - Ausnahmen **deklarieren** — meist eigener Typ (SML) oder Klasse (Java, Haskell)
  - Ausnahmen **auslösen** (`throw`, `raise`)
  - Ausnahmen **fangen** (`catch`)

Ausnahmen in Java

- Ausnahmen sind Objekte der Klassen `Throwable`, `Exception`
- Geworfene Ausnahmen müssen **deklariert** werden (`throws ...`)
  - Warum? Static Scoping (siehe Beispiel)
- Schema:

```
try
  block
catch (exception_type e)
  block
catch (exception_type e)
  block
finally
  block
```

Ausnahmen in Haskell

- Ausnahmen sind Instanzen der Typklasse `Exception` (Modul `Control.Exception`)
  - Situation in Haskell98 anders
- Werfen und fangen:

```
throw :: Exception e ⇒ e → a
catch :: Exception e ⇒ IO a → (e → IO a) → IO a
```

- Exceptions können überall geworfen werden, aber nur als Aktion (`IO`) gefangen werden.
  - Warum? Bricht referentielle Transparenz
- Durch Nicht-Striktheit (verzögerte Auswertung) werden Ausnahmen später geworfen als man denkt (siehe Beispiel)

Ausnahmen in C

- C hat **keine** Exceptions
- Alternativen:
  - `goto`
  - `setjmp` and `longjmp`
  - `switch`, siehe Duff's Device

Duff's Device:

```
void send(short *to, short *from, i
{
  int n = (count+ 7)/8;
  switch (count % 8) {
   case 0: do {
              *to = *from++;
   case 7:    *to = *from++;
   case 6:    *to = *from++;
   case 5:    *to = *from++;
   case 4:    *to = *from++;
   case 3:    *to = *from++;
   case 2:    *to = *from++;
   case 1:    *to = *from++;
           } while (--n > 0);
} }
```

Quelle: Wikipedia

Programmieren mit Ausnahmen

Ausnahmen sollten **unerwartete** und **seltene** Situationen modellieren.

1. "Ask forgiveness, not permission"
   - Bessere Robustheit
2. "Let it fail"

3. Nur Ausnahmen fangen, die auch behandelt werden
   - Unbehandelte Fehler werden nur schlimmer
4. Ausnahmen können auch der Effizienz dienen.

Zusammenfassung

- Abstraktion ist die Kunst des Weglassens unerheblicher Details
- Heute: Kontrollabstraktion
- Prozeduren — parametrisierte Blöcke
  - Parameter: In, Out, In/Out
  - Parameterübergabemechanismen: call-by-value, call-by-reference, call-by-name
- Ausnahmen: reglementierte Sprünge
  - In Java, Python, Haskell recht ähnlich
  - In C nicht vorhanden
  - Sollten **unerwartete** und **seltene** Situationen behandeln

Programmiersprachen
Vorlesung 5 vom 15.11.21
Datenabstraktion

Christoph Lüth

Universität Bremen

Wintersemester 2021/22

Organisatorisches

- Erste Vergabe der Referatsthemen am Donnerstag (18.11.2021)
- Bei Interesse gerne vorher Mail an Veranstalter
- Es gilt im Zweifel first come, first serve.
- List der Sprachen ist **nicht exklusiv** — nehme gerne weitere Vorschläge entgegen.

Liste möglicher Sprachen

- Systemnah: Rust
- Logische Programmierung: Prolog, Oz
- Dynamisch: JavaScript
- Nebenläufig/Reaktiv: Erlang, Golang
- Abhängige Typen: Idris, Agfa/Agda, Liquid X (Dependent types)
- Prozedural: Julia, Kotlin, Swift
- Skriptsprachen: Lua, Tcl, sh/bash
- Funktional: SML/OCaml, Elm, Clojure, LISP, Scala
- Stack-basiert: Forth
- Historisch: COBOL, Algol-68, APL, Ada, Smalltalk
- Datenflusssprachen: Id, Lucid, Lustre
- DSLs: R, SQL, Postscript, TeX, Verilog/VHDL, SystemC, SpinalHDL

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.

Abstraktion

- Kontrollabstraktion (done):
  - Jenseits der direkten Auswertung
  - Prozeduren und Parameter
  - Sprünge
- Datenabstraktion (heute):
  - Abstrakte Datentypen
  - Verkapselung und Objekte
  - Module
  - Packages

Wozu Datenabstraktion?

- Kontrollabstraktion hilft uns, Programme **verständlich** zu machen.
- Datenabstraktion hilft uns, **große** Programme verständlich zu machen.
- Indem wir existierende Daten und Funktionen (Methoden) zu neuen Datentypen zusammenfassen erlauben wir Abstraktion in der Sprache.
  - Abstraktion = "geordnetes Weglassen", hier: von Implementationsdetails.
  - Triviales Beispiel: `Int` als `Bool`, ist `0` jetzt `True` oder `False`?

Abstrakte Datentypen

Abstrakter Datentyp (ADT)

Ein ADT besteht aus besteht aus einem (oder mehreren) **Typen** und **Operationen** darauf, mit folgenden Eigenschaften:

1. Werte des Typen können nur über die Operationen **erzeugt** werden
2. Eigenschaften von Werten des Typen werden nur über die Operationen **beobachtet**.
3. Die Einhaltung von **Invarianten** über dem Typ kann garantiert werden

- Damit eine Programmiersprache ADTs unterstützt, müssen wir die **Sichtbarkeit** einschränken können (*information hiding*).
- **Repräsentationsunabhängigkeit**: Eigenschaften sollten von konkreter Implementation unabhängig sein.

Beispiel: Stack

Ein Stack (von ganzen Zahlen) hat mehrere Operationen:

- den leeren Stack (**empty**),
- eine Zahl auf den Stack schieben (**push**),
- die oberste Zahl vom Stack nehmen (**top** und **pop**),
- und einen Test, ob der Stack leer ist (**isEmpty**).

mit folgenden Eigenschaften:

1. der leere Stack ist leer, und nur der;
2. das oberste Element das Stacks ist das letzte darauf geschobene;
3. wenn ich von einem Stack, auf den ich ein Element geschoben habe, das oberste Element herunternehme, ändert sich nichts.

Stack: Implementation

- Beispiel: Stack in Python
  - Stack als Liste
  - Stack als Feld
- Unveränderliche Stacks (immutable): `push` und `pop` liefern **neuen** Stack
- Veränderliche Stacks (mutable): `push` und `pop` haben Seiteneffekte
- Problem: Python schränkt Sichtbarkeit bedingt ein
  - Keine Trennung zwischen Interface und Implementation
  - Private Felder **syntaktisch** gekennzeichnet (`__name`), Zugriff trotzdem möglich (als `_Class__name`)
  - Python Design-Prinzip: "Wir sind alle erwachsen."
- Beispiel: Stack in C
  - Interface `stack.h` syntaktisch getrennt

Stacks in anderen Sprachen

- In Haskell:
  - Wir können Sichtbarkeit einschränken, aber syntaktisch nicht trennen
  - Nur funktionale Lösung
- In Java:
  - **Interfaces** als separates Konstrukt

Spezifikation

- Wollen Stacks **mathematisch** beschreiben
  - Möglichst unzweideutig
  - Können daraus Tests generieren
- Hier:

$$\text{top}(\text{push}(s, x)) = x \qquad \text{isEmpty}(\text{empty})$$

$$\text{pop}(\text{push}(s, x)) = s \qquad \neg(\text{isEmpty}(\text{push}(s, x)))$$

- Gleichungen gelten nur für **unveränderliche** (zustandsfreie) Stacks
- Was bedeutet Gleichheit?
  - Gleichheit für `int` — bekannt
  - Gleichheit für `stack` — nicht gleich, nur **beobachtbar** gleich

Sprachmittel zur Unterstützung von Datenabstraktion

- Sichtbarkeitseinschränkungen aka. Module
- Syntax zur Beschreibung von Schnittstellen
- Trennung der Schnittstelle von der Implementation

Module

- Ein **Modul** ist die Zusammenfassung mehrerer Definition zu einer Einheit
  - Oft mit Verkapselung — Modul hat definierte **Schnittstelle**
  - Module dienen oft auch zur **getrennten Übersetzung** (aber nicht notwendigerweise)
- Module werden deklariert, definiert und benutzt (importiert).
  - Trennt die Sprache das?
  - Wie erfolgt die Benutzung?
  - Wie verhalten sich Module zu Quelldateien?
  - Wie wird importiert — qualifiziert oder unqualifiziert, immer alles, mit Umbenennung?
- Qualifizierter Import: Bezeichner `f` aus Modul `M` wird zu `M.f`.

Module in Python

- Module sind Quelldateien
- Keine Interface-Definition
- Kaum Sichtbarkeitseinschränkungen
- Keine Datenabstraktion
- Import: mit Namen, qualifiziert/unqualifiziert, alles oder selektiv, Umbenennung möglich

Module in Haskell

- Jede Quelldatei ist ein Modul
- Interface: Sichtbarkeit von Bezeichnern kann eingeschränkt werden
- Aber:
  - algebraische Datentypen und Konstruktoren bleiben erkennbar
  - Typsynonyme sind nicht abstrakt
  - Klasseninstanzen werden immer exportiert
- Datenabstraktion möglich (manchmal umständlich)
- Interface keine separate Datei
- Import: mit Namen, qualifiziert/unqualifiziert, alles oder selektiv, Umbenennung möglich

Module in Java

- Module sind **Klassen** (nicht an Quelldatei gebunden)
- Klassen verkapseln interne Repräsentation, Einschränkung der Sichtbarkeit (`public`, `private`, `protected`)
  - Aber Konstruktoren bleiben erkennbar
- Datenabstraktion möglich (durch Reflektion zu durchbrechen?)
- Interfaces sind separates Konstrukt
- Import: nur ganze Klassen, keine Umbenennung, impliziter Import möglich

Module in C

- Module sind Quelldateien (“translation units”)
- Sichtbarkeitseinschränkungen für Bezeichner (`static`, `extern`)
  - Local and global linkage
- Interfaces sind **per Konvention** separate Dateien (`.h`)
  - Konvention wird durch den Präprozessor ermöglicht
- Datenabstraktion möglich (durch Zeigeroperationen zu durchbrechen)
- Import: immer alles, nur unqualifziert, keine Umbenennung

Packages

- Packages schränken die Sichtbarkeit von Modulen ein.
- Existiert in Java, Python.
- Haskell kennt nur hierarchische Module.
- Wenn Module Quelldateien entsprechen, sind Packages Verzeichnisse.

Sprachneutral Interfaces: IDL

- IDL ist die `Interface Definition Language` der OMG
- Definition der Schnittstelle von Komponenten in CORBA
  - Nicht mehr ganz aktuell
- Syntax an C angelehnt
- Compiler erzeugt aus IDL “Rumpf” in entsprechender Programmiersprache
- Alle Funktionsaufrufe gehen über einen “Broker”

Fallbeispiel: Standard ML

- Standard ML (SML) ist eine strikte, nicht-reine funktionale Sprache
  - Typsystem wie Haskell
  - Referenzen und Ein/Ausgabe eingebaut
- Caml und OCaml sind französische Varianten, F# ist OCaml für .Net
- Freie Implementationen, e.g. SML/NJ: `http://www.smlnj.org/`

Module in SML

- Interfaces sind separates Konzept (`signature`)
- Module (`structure`)
- Parametrisierte Module (`functor`)
- Beispiele:
  - Ordnung und Sortieren (`sort.sml`)
  - Finite Maps (`maps.sml`)
- Vorteile:
  - Sehr flexibel, deklarative Beschreibung der Systemstruktur
  - Kann unübersichtlich werden (insbesondere sharing constraints)

Zusammenfassung

- Datenabstraktion durch **abstrakte Datentypen**:
  - Typ mit Operationen darüber
  - Zugriff nur über definierte Schnittstelle
  - Repräsentationsunabhängigkeit
- Module: **Verkapselung** durch Einschränkung der Sichtbarkeit
  - Was wird verkapselt: Sichtbarkeit der Bezeichner, Typrepräsentation, Konstruktoren?
  - Sind Interfaces explizit oder implizit?
  - Modul = Quelldatei?
  - Wie wird importiert?
  - Qualifizierte Bezeichner `M.f`
- Packages: Sammlungen von Modulen
- Fallbeispiel: Module in SML

Programmiersprachen
Vorlesung 6 vom 22.11.21
Fortgeschrittene Typsysteme

Christoph Lüth

Universität Bremen

Wintersemester 2021/22

Wo sind wir?

- Einführung
- Werte und Typen
- Anweisungen, Variablen und Zustand
- Kontrollabstraktion
- Datenabstraktion
- Fortgeschrittene Typsysteme
- Nebenläufigkeit
- Programmierparadigmen
- Eine Beispielsprache: Tcl und Scriptsprache
- Beispielsprache II
- Ab Woche 11: Studentische Vorträge.

Polymorphie

Polymorphie

- Von griechisch $\pi o \lambda \nu \zeta$ (viel), $\mu o \rho \phi \eta$ (Gestalt)
- Ganz allgemein: Funktionen (Methoden), die auf **mehr als einem** Typ anwendbar sind.
- Im speziellen:
  - In **objektorientierten Sprachen**: jede Methode ist auch auf allen Untertypen anwendbar.
  - **Parametrische Polymorphie**: nach Hindley-Milner, uniform auf **allen** Typen definiert.
  - **Ad-Hoc Polymorphie**: Überladene Funktionen, auf **einigen** Typen spezifisch definiert.

Polymorphie in Java und Python

- Methode `f` eine Klasse kann auf allen Untertypen angewandt werden.
- Klasse des Objektes bestimmt konkrete Methode (**dynamische Bindung**)

```
class C:
  def f(self):
    print("Foo.")
  def g(self):
    print("Baz.")

class D(C):
  def f(self):
    print("Wibble.")
```

Parametrische Polymorphie:

- Parametrische Polymorphie: Abstraktion über **Typen**
- Sowohl für Funktionen (Methoden) als auch für Datentypen (Klassen)
- Beispiel: Listen
  - Haben für alle Typen die gleiche Struktur
  - Viele Funktionen auf Listen sind vom Inhalt der Listen unabhängig.
- Typisierung nach dem Hindley-Milner-Damas-Algorithmus

Parametrische Polymorphie in Java: Generics

- Beispiel: Listen

```
class List<T> {
  public T elem;
  public List<T> next;

  public List(T el, List<T> tl) {
    this.elem= el;
    this.next= tl;
  }
```

- Benutzung umständlich weil Java keine Typen inferiert:

```
List<Integer> l1= new List<>(1, new List<>(2, null));
```

Parametrische Polymorphie in Haskell

- Typ-Parameter, an Funktionen oder Typen:

```
data List a = Cons a (List a) | Null

map :: (a → b) → List a → List b
map f Null = Null
map f (Cons a l) = Cons (f a) (map f l)
```

- Haskell leitet Typen ab, vergleicht dann (ggf.) mit deklarierten Typen
- Elegante Benutzung

Parametrische Polymorphie in C

- Der Typ `void *` ist mit `t *` für alle Typen `t` kompatibel.
  - C-Standard (C-90), 6.3.2.3 Pointers:
    A pointer to void may be converted to or from a pointer to any incomplete or object type.
- Manuelle Typannotation nötig — Typinformation geht verloren (bspw. für `map` und `filter`)
- Funktionen höherer Ordnung durch Zeiger auf Funktionen.
- Vergleiche Beispiel.

Theoretische Aspekte:

- Parametrische Polymorphie ist **entscheidbar** (Hindley-Milner-Damas).
  - Mit exponentiellem Aufwand.
  - In der Praxis unerheblich.
- Wird schnell unentscheidbar:
  - Konstruktorklassen
  - Rank-2 Polymorphie
  - Subtyping

Ad-Hoc-Polymorphie und Überladen

- **Ad-Hoc-Polymorphie** ist ein Aspekt von **Überladung:** ein Bezeichner, mehrere Funktionen.
  - Beispiel für Ad-hoc-Polymorphie: Addition + auf verschiedenen numerischen Typen.
  - Beispiel für Überladen: – unär für Negation, binäre für Subtraktion
- **Java** erlaubt Überladen, aber keine Ad-Hoc-Polymorphie
  - Gleicher Bezeichner mit mehreren, aber **unterschiedlichen** Signaturen
- **C** hat überladene numerische Operatoren (+, –, ∗) und erlaubt sonst **kein** Überladen
- **Python** erlaubt kein Überladen (hat aber default-Parameter u.ä.)

Ad-Hoc-Polymorphie in Haskell

- Haskell hat Typklassen:

```
class Eq a where
  (==) :: a → a → Bool
class Eq a ⇒ Num a where
  (+) :: a → a → a

instance Num Int where
  a+ b=...
```

- Ansonsten kein Überladen
- Typklassen für Datentypen (Konstrukturklassen)

```
class Monad m where
  (>>=)  :: m a → (a → m b) → m b
  return :: a → m a
```

Subtyping

Untertypen (Subtyping)

- Semantisch ist S ein **Untertyp** von T gdw.

$$S \subseteq T$$

  - Wo immer T gefordert ist, kann auch S benutzt werden.
- Beispiel numerische Typen

$$\mathbb{N} \subseteq \mathbb{Z} \subset \mathbb{R} \tag{1}$$

  - Siehe Typkonversionen in C: `unsigned int`, `int`, `double` sowie verschieden Wortbreiten
- Verallgemeinert: S ist Untertyp von T wenn es eine **Einbettung** gibt:

$$\iota : S \hookrightarrow T \quad \iota \text{ injektiv}$$

- Einbettung: **explizite** Konversion
- Beispiel numerische Typen in Haskell

Record Subtyping

- Vererbung erzeugt semantisch gesehen keine Subtypen

```
class Point {
    double x;
    double y;
}
```

```
class ColouredPoint
      extends Point {
    Colour col;
}
```

- `Point` ist $\mathbb{R} \times \mathbb{R}$, `ColouredPoint` ist $\mathbb{R} \times \mathbb{R} \times \mathit{Colour}$
- Wir können aus jedem `ColouredPoint` einen `Point` machen
  - Allerdings nicht injektiv
  - Solange wir **nur** auf die **Felder** zugreifen
- Deshalb können wir `ColouredPoint` als Untertyp von `Point` **definieren** ("record subtyping" oder "structural subtyping")

Inheritance is not Subtyping

- Record Subtyping funktioniert nur bei Argumenten:

```
void move(Point p) {
  this.x += p.x;
  this.y += p.y;
}
```

- Kann auch auf `ColouredPoints` angewandt werden.

```
static Point
  fromPolar(double phi, double r) {
  return new Point(r*Math.cos(phi),
                   r* Math.sin(phi));
}
Point move1(Point p) {
  return new Point(this.x+ p.x,
                   this.y+ p.y);
}
```

- Welche Farbe soll der Ergebnistyp haben?

- Wenn $A \subseteq A'$, dann ist $A \to B \subseteq A' \to B$, aber $B \to A \not\subseteq B \to A'$

Subtyping und Parametrische Polymorphie (Generics)

- Frage: sind Typkonstruktoren F **monoton**

$$A \subseteq B \Longrightarrow F(A) \subseteq F(B)?$$

- Ganz allgemein gilt:

$$\begin{array}{rcl} A \subseteq A' & \Longrightarrow & A \to B \subseteq A' \to B \\ & & A \times B \subseteq A' \times B \\ & & A + B \subseteq A' \times B \\ A \subseteq A' & \Longrightarrow & B \to A' \subseteq B \to A \end{array}$$

- **Polynomiale** Typkonstruktoren sind Datentypen aus Produkt und Koprodukt (vgl. `data` in Haskell ohne Funktionsräume)
- Polynomiale Typkonstruktoren sind monoton.
- Funktionsräume $A \to B$ machen den Unterschied.

Kovarianz und Kontravarianz

- Ein Typkonstruktor $F(X)$ ist **kovariant**, wenn X nur in **Argumentposition** des Funktionsraums $\rightarrow$ auftaucht.
- Ein Typkonstruktor ist $F(X)$ ist kontravariant, wenn X nur in **Resultatposition** des Funktionsraums $\rightarrow$ auftaucht.
- Wenn F **kovariant**, dann ist F monoton: $A \subseteq B \Longrightarrow F(A) \subseteq F(B)$
- Wenn F **kontravariant**, dann ist F anti-monoton: $A \subseteq B \Longrightarrow F(B) \subseteq F(A)$

Subtyping und Generics

- Praktische Auswirkungen — ein klassisches Beispiel:

{

```
class Ref<T> {
  private T curr;
  Ref(T init) { this.curr= init; }

  T get() { return curr; }
  void set(T x) { curr= x; }
  }
```

}

- Consider this:

```
Ref<String> c1 = new Ref<>("foo");
Ref<Object> c2 = c1;   // String ⊆ Object, also Ref<String> ⊆ Ref<Object>
c2.set(1);             // Ändert auch c1
String s = c1.get();   // Liest c1, wo aber jetzt eine Zahl steht... ↯
```

- Problem ist zweite Zeile, `Ref<String>` $\not\subseteq$ `Ref<Object>`

Arrays und Generics

- Lösung: Generics sind **invariant** (Typkonstruktoren nicht monoton)
  - Typ `<? extends T>`, `<? super T>`
- Arrays sind **nicht** invariant:

```
String[] c1 = { "foo" };
Object[] c2 = c1;
c2[0]= 99;
String s = c1[0];
System.out.println(s); // What will happen?
```

Arrays und Generics

- Lösung: Generics sind **invariant** (Typkonstruktoren nicht monoton)
  - Typ `<? extends T>`, `<? super T>`
- Arrays sind **nicht** invariant:

```
String[] c1 = { "foo" };
Object[] c2 = c1;
c2[0]= 99;
String s = c1[0];
System.out.println(s); // What will happen?
```

- Grund: Arrays sind **reifizierbar** — Typ wird zur Laufzeit **nicht** gelöscht.
- Bei Generics wird der Typ zur Laufzeit gelöscht (type erasure) — effizientere Ausführung.

Subtyping, Overloading and Dynamic Binding

- Java kombiniert Subtyping, Overloading und dynamisches Binden.
- Im ersten Argument (Methodenauswahl) wird die Methode **dynamisch** ausgewählt.
- Ansonsten wird der Typ **statisch** bestimmt.
- **Overloading** wird **statisch** aufgelöst.
- Siehe Beispiel.

Dependent Types

- Abhängige Typen vermischen Typen und Terme
- Typen können mit Termen gebildet werden
- Beispiel (fiktive Syntax): `Vec l` sind Listen der Länge `l :: Int`

```
data Vec (l :: Int) a = Null 0 | Cons a (Vec (l-1))

map :: (a→ b) → Vec l a→ Vec l b
(++) :: Vec l a→ Vec m a→ Vec (l+m) a
```

- Nicht mehr entscheidbar.
- Erlaubt Kodierung von **Spezifikation** im Typ.
- Kann **Beweise** zur Übersetzungszeit erfordern.

Zusammenfassung

Zusammenfassung

- Fortgeschrittene Aspekte der Typsysteme
- Arten der Polymorphie
  - Polymorphie über Subtypen
    - Java und Python
  - Parametrische Polymorphie
    - In Haskell, Java (Generics); rudimentär in C (`void *`)
  - Ad-Hoc-Polymorphie und Overloadimg
    - Overloading in Java, Ad-Hoc-Polymorphie in Haskell
- Subtyping
  - Record Subtyping in Java, Python.
  - Kombination mit parametrischer Polymorphie delikat.

Programmiersprachen
Vorlesung 7 vom 29.11.21
Nebenläufigkeit

Christoph Lüth

Universität Bremen

Wintersemester 2021/22

Wo sind wir?

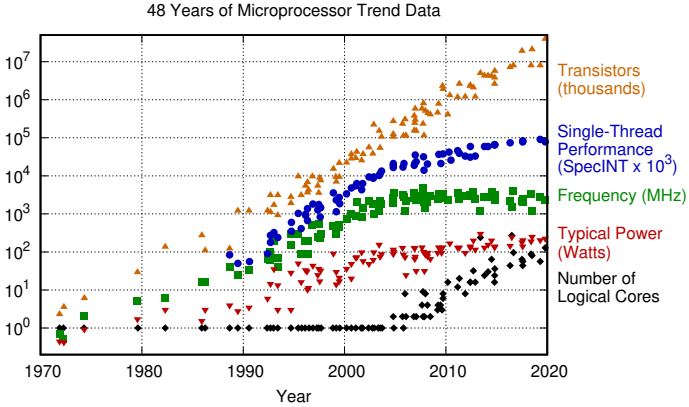
- Einführung
- Werte und Typen
- Anweisungen, Variablen und Zustand
- Kontrollabstraktion
- Datenabstraktion
- Fortgeschrittene Typsysteme
- Nebenläufigkeit
- Objektorientierung
- Skriptsprachen
- Beispielsprache II
- Ab Woche 11: Studentische Vorträge.

Nebenläufigkeit

Warum Nebenläufigkeit?

- Abstraktion:
  - Die Welt **ist** nebenläufig.
  - Viele Anwendungen reagieren
    - auf Eingaben in undefinierter Reihenfolge,
    - zu unvorgesehenen Zeitpunkten.
    - Beispiel: Webanwendungen
  - Architekur der Anwendung muss diese Struktur reflektieren

Warum Nebenläufigkeit?



- Performance:
  - Ausnutzung mehrerer Cores
  - Ausnutzung von **Latenz** (Warten auf externe Ereignisse)

Sequentielle Architektur.

- Die Von-Neumann Architektur:
  - Programme und Daten in einem Speicher
  - Berechnung in Zyklus fetch — compute — store
- Formale Beschreibung:
  - Turing-Maschine
  - Interner Zustand plus Arbeitsspeicher (Band)

Nicht-sequentielle Modelle:

- Nicht alle Berechnungsmodelle sind sequentiell.
- Beispiel: λ-Kalkül, Gleichungsreduktion

```
inc :: Int → Int
inc x = x+1
```

```
mdbl :: Int → Int → Int
mdbl x y = 2* x * y
```

- Im Ausdruck `inc (mdbl (inc 4) (inc 2))` sind mehrere Reduktionen möglich.
  - Innermost-first, outermost-first (s. vorletzte Vorlesung)
  - ... oder sogar parallel (`inc 4` und `inc 2`)

Implementationsaspekte

Prozesse vs. Threads

- Prozesse:
  - Schwergewichtig, Erzeugung teuer
  - Eigener Speicherbereich, eigene Ressourcen
  - Vom Betriebssystem verwaltet
- Threads:
  - Leichtgewichtig, Erzeugung billig
  - Geteilter Speicherbereich und Ressourcen
  - Von Programmierer/Programmiersprache verwaltet
- Uns interessieren hier **Threads**

Arten der Nebenläufigkeit:

- **true concurrency** vs. **interleaved concurrency**
  - true concurrency: Programme laufen **gleichzeitig**, z.B. threads auf einer Multi-Core-CPU
  - interleaved concurrency: Programme laufen auf einer zentralen Berechungseinheit
- **Kooperativ** vs. **präemptiv**
  - Kooperativ: Programme geben explizit Kontrolle ab
  - Präemptiv: Betriebssytem/Scheduler unterbricht Programme, verteilt Kontrolle.

Probleme der Nebenläufigkeit

- Nicht-Determinismus
- **Deadlock**
  - Nichts geht mehr — gegenseitige Blockade
  - Beispiel: Straßenkreuzung, Bewerber aus dem Ausland
- **Starvation**
  - Ein Thread wird “ausgehungert”
  - Beispiel: Auto in Nebenstraße zu Hauptverkehrsstraße

Operationen für Threads

- Erzeugung — neuen Thread starten
  - Erhält auszuführenden Rumpf als Argument
  - Jeder Thread hat eine eindeutige `id`
- Beendigung
  - Beendet **laufenden** Thread
- Ggf. Kontrollübergabe
  - Für kooperative Nebenläufigkeit
  - Meist implizit in I/O-Operationen
- **Synchronisation**
  - Auf andere Threads warten, andere Threads stoppen, Nachrichten senden und empfangen
  - Hier wird es interessant...

Konzepte der Nebenläufigkeit

Konzepte

- Konzepte zur Nebenläufigkeit in Programmiersprachen dienen zwei Zwecken:
  1. Verhinderung von Interferenz
  2. Kommunikation

Definition: Kritischer Abschnitt

Ein **kritischer Abschnitt** (in Bezug auf eine Resource r) ist ein Teil des Programmes, in kein anderer Thread/Prozess auf r zugreifen darf.
In Bezug auf die Resource CPU: in dem kein anderer Thread/Prozess laufen darf.

Kritische Abschnitte I: Spin-Locks.

- Spin-Locks sind ein Beispiel für **Mutexe**
  - Sicherstellung des gegenseitigen Ausschluß (mutual exclusion)
- Spin-Locks sind "busy-waiting loops"
- Benutzung mit `acquire(r)` und `release(r)`:

```
... non-critical code ...
acquire(r);
// CRITICAL SECTION
release(r);
... more non-critical code ...
```

- `r` ist hier der Parameter, der die Resource identifiziert.
  - Ohne `r`: kein anderer Thread darf laufen

Dekker's Algorithmus

- Implementiert `acquire(r)` und `release(r)`
- Setup:
  - Zwei Threads, jeweils `self` und `other`.
- Implementation in vier Versuchen nach Djikstra (1968a)
- Erster Versuch:

```
acquire(r) =
  while turn = other loop null: end loop;

release(r) =
  turn := other;
```

- Stellt Ausschluss sicher
- Problem: Prozesse **müssen** alternieren

Dekker's Algorithmus: Zweiter Versuch

- Zweiter Versuch:
  - Nutzt Array ein `claimed[2]`

```
acquire(r) =
  while claimed[other] loop null; end loop;
  claimed[self]:= true;

release(r) =
  claimed[self]:= false;
```

- Problem: Gegenseitiger Ausschluss nicht garantiert:
  - Thread 1 findet `claimed[2] == false` und will gerade `claimed[1]` auf `true` setzen...
  - ... als Thread 2 `claimed[1] == false` findet und den kritischen Abschnitt betritt ....
  - ... Thread 1 ist wieder dran, setzt `claimed[1]` auf `true` aber zu spät.

Dekker's Algorithmus: Dritter Versuch

- Dritter Versuch:

```
acquired(r) =
  claimed[self]:= true;
  while claimed[other] loop
    claimed[self] := false;
    while  claimed[other] loop null; end loop;
    claimed[self] := true;
  end loop;
```

- Problem: Deadlock bei **gleichzeitiger** Ausführung
  - Kontextwechsel nach jeweils einer Anweisung

Dekker's Algorithmus: Korrekte Fassung

- Korrekte Fassung:

```
acquired(r) =
  claimed[self]:= true;
  while claimed[other] loop
    if turn = other then
      claimed[self] := false;
      while  claimed[other] loop null; end loop;
      claimed[self] := true;
    end if;
  end loop;

release(r) =
  turn := other;
  claimed[self]:= false;
```

- Problem: korrekt, aber schlecht auf n Threads zu verallgemeinern.

Peterson's Algorithmus

- Peterson's Algorithmus: einfacher, verallgemeinert Dekker:

```
acquire(r) =
  claimed[self] := true;
  turn := other;
  while claimed[other] and turn = other
    loop null; end loop;

release(r) =
  claimed[self]:= false;
```

Fazit: Dekker und Peterson

- Algorithmen zeigen, wie kompliziert Nebenläufigkeit ist...
- Spin-Locks verschwenden CPU-Zeit.
- Algorithmen haben Voraussetzungen:
  - Compiler optimiert keine Zugriffe
  - Änderungen werden **sofort** für **alle** Prozessoren sichtbar
- Weiteres: Simpson's Algorithmus

Semaphoren

- Verallgemeinerung von Spin-Locks nach Dijkstra
- Drei Operationen:
  - Initialisierung mit Wert n – gibt an, wieviele Prozesse **maximal** im kritschen Abschnitt sein dürfen.
  - Warten (`wait`, `p`) — Betritt kritischen Abschnitt, kann blockieren wenn gewartet werden muss
  - Freigeben (`signal`, `v`) — Verläßt kritischen Abschnitt, kann anderen Thread freigeben
- Invariante (für Semaphore s):

$$0 \leq s.waits \leq s.signals + s.initial$$

Semaphoren: Beispiel

- Ein geteilter Buffer (e.g. Speicherbereich)
- Gemeinsamer Teil:

```
Semaphore full;
Semaphore empty;
char buf[BUFFERSIZE]; // Actual type irrelevant

sema_init(full, 0);
sema_init(empty, 1);
```

Sender:

```
for (;;) {
  sema_wait(empty);
  write(buf);
  sema_signal(full);
}
```

Empfänger:

```
for (;;) {
  sema_wait(full);
  read(buf);
  sema_signal(empty);
}
```

Kontrollabstraktionen

Events und Messages

- Prozesse (oder Threads) senden **Nachrichten** auf **Kanälen**
- Grundlegende Funktionen:
  - Kanal erstellen (wenn möglich getypt, ggf. mit initialer Kapazität)
  - Auf Kanal Nachricht **senden** (blockiert wenn Kanal voll, oder Fehler)
  - Aus Kanal Nachricht **empfangen** (blockiert wenn Kanal leer)
- Damit fortgeschrittene Funktionen möglich:
  - Testen ob Nachricht empfangen werden kann
  - Aus mehreren Kanälen lesen
- Abstraktion über zugrundeliegenden Transportmechanismus:
  - Shared memory, pipes, Sockets (im Filesystem oder über das Netz),...
- Diverse Implementierungen

Monitore

- Zuerst in Concurrent Pascal und Modula-2 (Niklaus Wirth)
- Monitore kombinieren
  - gegenseitigen Ausschluss mit
  - Kommunikation und
  - Verkapselung.
- Modern: `synchronized` in Java

```
DEFINITION MODULE Processes;
  TYPE SIGNAL;

  PROCEDURE StartProcess(P: PROC; n: INTEGER);
    (* startet einen nebenläufigen Prozeß mit Programm P
       und einem Arbeitsspeicher der Größe n. PROC ist
       ein Standardtyp, definiert durch PROC = PROCEDURE *)

  PROCEDURE SEND(VAR s: SIGNAL);
    (* startet einen auf s wartenden Prozeß wieder *)

  PROCEDURE WAIT(VAR s: SIGNAL);
    (* wartet auf einen anderen Prozeß, der s sendet *)

  PROCEDURE Awaited(s: SIGNAL): BOOLEAN;
    (* Awaited(s) = "mindestens ein Prozeß wartet auf s" *)

  PROCEDURE Init(VAR s: SIGNAL);
    (* zwingende Initialisierung *)
END Processes.
```

Aus: Niklaus Wirth, Programmieren in Modula-2, Springer 1991.

Aktoren

- Grundlegendes Berechnungsmodell nach Hewitt, Bishop, Steiger
- Hier: Kontrollabstraktion für Nebenläufigkeit
- Aktoren verarbeiten Nachrichten

Während ein Aktor eine Nachricht verarbeitet, kann er

- neue Aktoren erzeugen,
- Nachrichten an bekannte Aktor-Referenzen versenden,
- festlegen, wie die nächste Nachricht verarbeitet werden soll.

Ein Aktor darf nicht

- auf einen **globalen** Zustand zugreifen,
- **veränderliche** Nachrichten versenden,
- irgendetwas tun, während er keine Nachricht verarbeitet.

Aktoren: Nachrichten

- Nachrichten sind **unveränderliche** Daten, **reine** Funktionen oder **Futures**
- Die Zustellung von Nachrichten passiert höchstens einmal (Best-effort)
  - Wenn z.B. die Netzwerkverbindung abbricht, wird gewartet, bis der Versand wieder möglich ist
  - Wenn aber z.B. der Computer direkt nach Versand der Nachricht explodiert (oder der Speicher voll läuft), kommt die Nachricht möglicherweise niemals an.
- Über den Zeitpunkt des Empfangs kann keine Aussage getroffen werden (Unbounded indeterminacy)
- Über die Reihenfolge der Empfangenen Nachrichten wird im Aktorenmodell keine Aussage gemacht (In vielen Implementierungen allerdings schon)
- Nachrichtenversand $\neq$ (Queue | Lock | Channel | ...)

Beispiel: Aktorenmodell in Erlang

- Aktorenmodell implementiert in Erlang und Akka (Bücherei für Scala/Java).
- Erlang:
  - Jede Funktion ein Aktor
  - Schwach getypt
  - Nachrichten senden mit `!`
  - Nachrichten empfangen mit `receive`
- Beispiel: `pingpong.erl`

Weitere Abstraktionsmechanismen

- Rendezvous (z.B. CSP)
- Remote Procedure Call (implemetationsnah)
- Futures (Promises)
- Software Transactional Memory (sehr abstrakt)

Implementationen

C

- Sprache selber streng sequentiell, Unterstützung durch externe Büchereien (i.e. nicht Teil des Standards)
- Ausnahme: `volatile`
  - C-Standard (C-90), 6.7.3.5 Type qualifiers:
    An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine [...].
    Furthermore, at every sequence point the value last stored in the object shall agree with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously.

Volatile

- Warum unterstützt `volatile` Nebenläufigkeit?
- Ohne `volatile` könnte der Compiler Optimierungen vornehmen.
- Siehe Peterson's Algorithmus:

```
acquire(int r) {
  claimed[self]= 1;
  turn = other;
  while (claimed[other] && turn == other);
}
```

- Compiler könnte `turn = other` aus Schleifenbedingung entfernen
- `volatile` verhindert diese Optimierung
- Auch in Java (mit der gleichen Bedeutung)

Python

- Sprache selber unterstützt keine Nebenläufigkeit
- Global Interpreter Lock (für CPython)
- Nebenläufigkeit nur durch Büchereien:
  - Threads `threading`
  - Asynchrone IO `asyncio` (Coroutinen)

Java

- Per design nebeneläufig: Threads werden durch JVM unterstützt
- Threads relativ schwergewichtig
- Synchronisation durch Monitore (`synchronized`)
- Beispiel: `threads.java`

Haskell

- **Sequentielles** Haskell: Reduktion eines Ausdrucks
- **Nebenläufiges** Haskell: Reduktion eines Ausdrucks an **mehreren Stellen**
  - `ghc` implementiert Haskell-Threads
  - Zeitscheiben (Default 20ms), Kontextwechsel bei Heapallokation
  - Threaderzeugung und Kontextswitch sind **billig**
- Wenige Basisprimitive, darauf aufbauend Abstraktionen
- Synchronisation mit `MVars`
- Erstes Beispiel: `SimpleConcurrent1.hs`

Synchronisationsmechanismus: MVars

- `MVar a` ist **polymorph** über dem Inhalt
- Entweder **leer** oder **gefüllt** mit Wert vom Typ `a`
- Verhalten beim Lesen und Schreiben:

Zustand vorher:	leer	gefüllt
Lesen	blockiert (bis gefüllt)	danach leer
Schreiben	danach gefüllt	blockiert (bis leer)

- **Aufwecken** blockierter Prozesse **einzeln** in **FIFO**

Zusammenfassung

Zusammenfassung

- Nebenläufigkeit tut not weil:
  - die Welt ist nebenläufig (Abstraktionsaspekt),
  - es geht schneller (Performanceaspekt).
- Nebenläufigkeitsabstraktionen:
  - Spin-Locks
  - Semaphoren
  - Monitore
  - Aktoren
- Unterstützung in Programmiersprachen: dürftig (C, Python), sehr gut (Java, Haskell)

Programmiersprachen

Vorlesung 8 vom 06.12.21

Programmierparadigmen

Christoph Lüth

Universität Bremen

Wintersemester 2021/22

Vortragsthemen

06.01.2022 Do	Jan Walther: Rust	Felix Glinka: LLVM IR
10.01.2022 Mo	Mamia Lehbib: JavaScript	Ole Fischer: TypeScript
13.01.2022 Do	Ove von Stackelberg: Golang	Jan Engel: Swift
17.01.2022 Mo	Fida Fahmadi: Kotlin	Hannes Wehrmann: Scala
20.01.2022 Do	Tarek Soliman: LISP	Jona Dirks: Prolog
24.01.2022 Mo	Jakob Gahde: Ocaml	Nele Matschull: Idris
27.01.2022 Do	Philip Hönnecke: Ada	Leon Ehrardt: Cobol
31.01.2022 Mo	Tobias Stage: ABAP	Larissa Fastenau: VHDL
03.02.2022 Do	Lennard Scheurer: Lua	Johanna Tholen: Elm

Wo sind wir?

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

Programmierparadigmen

Was ist ein Paradigma?

Definition nach dem Duden:

Bedeutungen (2)

1. Beispiel, Muster; Erzählung mit beispielhaftem Charakter

   Gebrauch **bildungssprachlich**

2. Gesamtheit der Formen der Flexion eines Wortes, besonders als Muster für Wörter, die in gleicher Weise flektiert werden

   Gebrauch **Sprachwissenschaft**

Definition nach Merriam-Webster:

Full Definition of *paradigm*

1 : EXAMPLE, PATTERN

especially **:** an outstandingly clear or typical example or archetype

// ... regard science as the *paradigm* of true knowledge.

— G. C. J. Midgley

2 : an example of a conjugation or declension showing a word in all its inflectional forms

3 : a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated

// the Freudian *paradigm* of psychoanalysis

broadly **:** a philosophical or theoretical framework of any kind

Was ist ein Paradigma?

Definition nach dem Duden:

Bedeutungen (2)

1. Beispiel, Muster; Erzählung mit beispielhaftem Charakter

   Gebrauch **bildungssprachlich**

2. Gesamtheit der Formen der Flexion eines Wortes, besonders als Muster für Wörter, die in gleicher Weise flektiert werden

   Gebrauch **Sprachwissenschaft**

Definition nach Merriam-Webster:

Full Definition of *paradigm*

1 : EXAMPLE, PATTERN

especially : an outstandingly clear or typical example or archetype

// ... regard science as the *paradigm* of true knowledge.
— G. C. J. Midgley

2 : an example of a conjugation or declension showing a word in all its inflectional forms

3 : a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated

// the Freudian *paradigm* of psychoanalysis

broadly : a philosophical or theoretical framework of any kind

Was ist ein Programmierparadigma?

Programmierparadigma

Ein Programmierparadigma ist eine **grundlegende Herangehensweise** an die Programmierung, und umfasst:

- ein **Berechnungsmodell** — wie rechne ich?
- ein **Weltmodell** — was sind die Objekte meiner Berechnungen?
- sowie darauf aufbauende Konzepte.

- Programmierparadigmen sind mehr als nur eine bestimmte Programmiersprache.
- Ein Programmierparadigma bedingt auch eine Modellierung und Systemarchitektur.

Bekannte Programmierparadigmen

- Prozedural-Imperativ
- Objektorientiert
- Funktional
- Logisch

Fallstudie

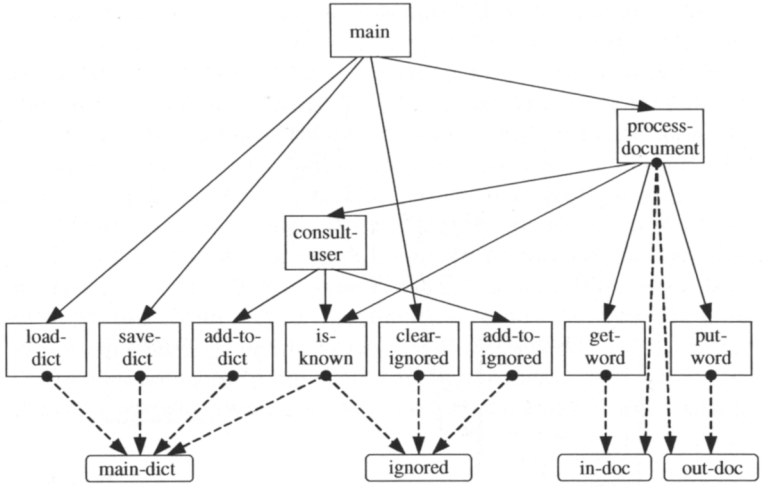
- Ein einfacher Spellchecker:
- Ablauf:
  - Eingabe: Text aus einer Datei
  - Zerlegt Text in Wörter
  - Prüft Wort gegen Wörterbuch
  - Wenn Wort nicht in Wörterbuch: ignorieren oder hinzufügen
  - Ausgabe: korrigierter Text in neuer Datei
- Quelle: Watt, Programming Language Design Concepts.

Imperative Programmierung

Prozedural-Imperativ

- Berechnungsmodell: Zustandsübergänge
- Weltmodell: Abstraktion des Speichers
- Schlüsselkonzepte:
  - Variablen und Befehle
  - Prozeduren und Funktionen
  - Einfache Datentypen
- Prozedural: Abstraktion durch Funktionen und Prozeduren
- Mutter aller Programmierparadigmen
- Programmiersprachen: C (uvm)

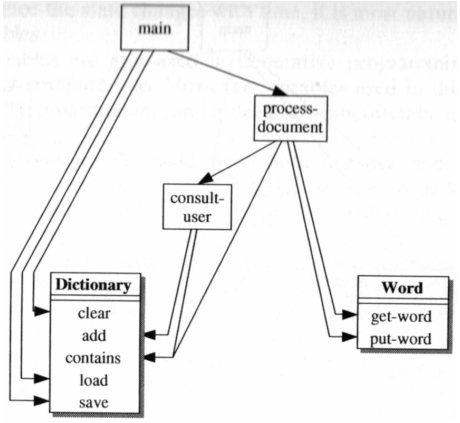
Imperative Systemarchitektur



Fallstudie imperativ

- Ablauf ist imperativ
- Keine Datenabstraktion:
  - Wörterbuch und Ignore-Liste globale Variablen
  - Ein- und Ausgabedatei global
  - Wörter sind `char *`, müssen anders deklariert werden

Prozedurale Systemarchitekure



Fallstudie prozedural

- Datenabstraktionen:
  - Funktionen zu `Dictionary` zusammengefasst
  - Dadurch Vereinfachung: `main_dict` und `ignored` sind beides Dictionaries.
  - Datenabstraktion zu `Word` immer noch unvollständig
- Erste Anflug von Objekt-Orientierung

Objekt-Orientierte Programmierung

Objekt-Orientiertes Paradigma

- Objekte der Berechnung sind **Objekte**
- Berechnung sind (abstrakte) Nachrichten (**Methoden**), welche die Objekte austauschen.
- Im einzelnen:
  - Verkapselung der Objekte als **abstrakte Datentypen**
  - Typisierung der Objekte durch Klassen
  - Strukturierung der Klassen durch **Vererbung**
  - Methodenaufruf durch **dynamische Bindung**
  - **Polymorphie** durch Subtyping

Geschichtliches

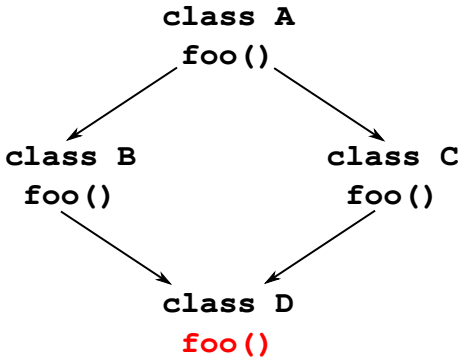
- Erste Sprache: Simula (1960s), Ole-Johann Dahl und Kristen Nygaard
- Simula kannte Objekte, Klassen, Vererbung, Koroutinen und hatte Speicherverwaltung.
- Spätere Sprachen: Smalltalk-80, C++, Eiffel, Java
- Noch spätere Sprachen: Ruby, Python, C#, Scala, OCaml

Vererbung

- Vererbung löst ein Problem mit abstrakten Datentypen:
  - ADTs sind **verkapselt** und könnten gut erweitert und wiederverwendet werden.
  - Aber: Verkapselung verhindert Wiederverwendung
  - Ferner: keine **hierarchischen** Definitionen
- Arten der Vererbung:
  - Einfache Vererbung (jede Klasse hat **eine** Oberklasse)
  - Mehrfachvererbung (jede Klasse hat **mehrere** Oberklassen)

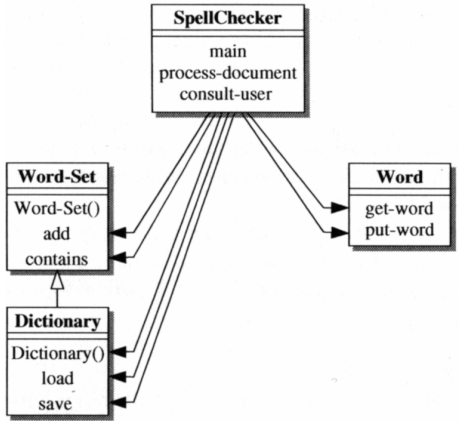
Probleme mit Mehrfachvererbung

1. Das **Diamanten-Problem** (diamond problem):
   - `B` erbt von `A` und überschreibt `foo`
   - `C` erbt von `A` und überschreibt `foo`
   - `D` erbt von `B` und `C` — was ist `foo` in `D`?
2. Implementation



- Sprachen mit Mehrfachvererbung: C++, Eiffel
- Sprachen mit Einfachvererbung: Java, Python, ...
  - Dafür Interfaces und/oder Traits

Die Fallstudie objekt-orientiert



Funktionale Programmierung

Funktionales Programmierparadigma

- Berechnungsmodell: rekursive Funktionen
- Weltmodell: algebraische Datentypen
  - frei definierbar — SML, Haskell
  - fest — LISP
- Schlüsselkonzepte:
  - Sprachen können pur (Haskell) oder gemischt sein (SML, LISP)
  - Getypt (Haskell, SML) oder ungetypt (LISP)
  - Strikt (SML, LISP) oder nicht-strikt (Haskell)

Die Fallstudie funktional

- Erste Annäherung:
  - Modularchitektur ähnlich vorher
  - Implementation ähnlich Java
- Zweite Annäherung: funktionale Modellierung
  - `spellcheck` als `stream processor`

```
consultUser :: String → Dictionaries → IO (String, Dictionarie
dropWord :: String → (String, String)
getWord :: String → Maybe (String, String)
```

Zusammenfassung

Zusammenfassung

- Programmierparadigmen bestehen aus einem Weltmodell und einem Berechnungsmodell
  - Was sind die Objekte meiner Berechnung?
  - Wie berechne ich?
- Beispiele:
  - imperativ und prozedural
  - objekt-orientiert
  - funktional
  - logisch
- Die meisten Programmiersprachen sind **gemischt** und unterstützen mehrere Paradigmen.
  - Aber meistens ein bestimmtes besonders gut.

Programmiersprachen
Vorlesung 9 vom 13.12.21
Skriptsprachen

Christoph Lüth

Universität Bremen

Wintersemester 2021/22

Wo sind wir?

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

Skriptsprachen

Was ist das?

- Kennzeichen von Skriptsprachen:
  1. Komposition komplexer Subsysteme zu einem Gesamtsystem[1];
  2. Hohe Abstraktionsstufe gegenüber der Hardware;
  3. Kleiner Sprachkern;
  4. Schneller Entwicklungszyklus;
  5. Nicht vorranging effizient, meist interpretiert;
  6. Meist auf ein Anwendungsgebiet zugeschnitten;
- Kein *Programmierparadigma*, eher Benutzungspragmatik

[1] A scripting language is one where the main effect of a program is to drive another system, while in a programming language the program itself is the main action. (Quelle)

Warum?

- Die meisten Skriptsprachen entstehen **evolutionär**:
  - Große Anwendungen mit komplexer Funktionalität
  - Anwender muss Ablauf steuern
  - Repetitiv und schematisch — Bedarf der Automatisierung
- Skriptsprachen automatisieren, was der Benutzer händisch macht.

Geschichtliches

- Stapelverabeitung (**batch control**) für Mainframe-Rechner: JCL (IBM) (1960?)
  "JCL ... is, I am convinced, the worst computer programming language ever devised by anybody, anywhere. It was developed under my supervision; there is blame enough to go around among all the supervisory levels." — Fred Brooks.
- Mainframes wurden zu Minicomputern zu PCs ...
  - Benutzerinteraktion durch eine **Shell**
  - Dazu Shell-Sprachen: sh, REXX (ab 1970)
- Später kamen dazu: GUIs, Datenbanken, Office-Anwendungen, Webanwendungen (ab 1980)

Moderne Skriptsprachen

- Allzweckskriptsprachen:
  - Python, Ruby
- “Glue languages”:
  - Tcl, bash
- Spezielle Anwendungsgebiete:
  - PHP, JavaScript (Webanwendungen)
  - Office-Anwendungen (Visual Basic)
  - Textverarbeitung (awk, sed, Perl)
- Eingebettete Sprachen:
  - Lua, Tcl, JavaScript

Schlüsselkonzepte

- Gute Unterstützung von Zeichenketten
  - Leichtgewichtige Parsierung durch **reguläre Ausdrücke**
  - Repräsentation strukturierter Daten durch XML oder JSON
- Integration von Hostanwendungen
  - GUI, Webschnittstelle, ...
  - Machen die "eigentliche" Arbeit
- Dynamisch getypt oder ungetypt
  - Flexibler, schnellerer Entwicklungszyklus

Beispielsprache: Shell

- Die Bourne-Shell und die Bourne-Again-Shell
- Erste Version 1976 für Unix V7
- Reimplementierung als Bourne-Again-Shell (`bash`) für GNU/Linux ab 1999.
- Was macht eine Shell?
  - Behandlung der Benutzereingabe (am Terminal oder auf der Kommandozeile)
  - Starten von Kommandos, Parameterexpansion, Umgebungsvariablen
    - Standardeingabe/-ausgabe
    - Als besondere "Filedeskriptoren"
  - Signalbehandlung (Unterbrechen, Suspendieren)
  - Umleitung der Ein/Ausgabe, Verkettung durch Pipes, Umgebungsvariablen
  - ... und sie kann programmiert werden!

Die Shell-Sprache

- Datentypen: Zeichenketten.
  - Komplett ungetypt (Was braucht man mehr?)
  - Ganze Zahlen sind auch nur Zeichenketten
  - Auswertung von Ausdrücken durch
- Elaborate Unterstützung von Literalen:

```
echo "Hello $World."
echo 'Hello $World.'
```

- Macros (Antiquotation)

```
x=`ls -la`
```

- Syntax: von ALGOL inspiriert

Programmierstrukturen

- Variablen (global, lokal)
- Volle Turingmächtigkeit
- `while`, `if`, `case`
- Ausführungsmodell:
  - Sequentielle Ausführung von Kommandos
  - Parameterexpansion,
- Funktionen:
  - Positionale Parameterübergabe
  - Dynamisches Scoping

Beispiele:

- Fakultät iterativ (`fac1.sh`)
- Fakultät rekursiv (`fac2.sh`)

Beispielsprache: Tcl

Was ist Tcl?

- Tcl ist eine **Skriptsprache**
  - Dynamisch und komplett ungetypt
  - Als **eingebette** und **erweiterbare** “glue language” konzipiert
- Erfolgreich durch
  - Expect
  - Tk, das GUI Toolkit (Tcl/Tk)
- Inzwischen etwas überholt (insbesondere Tk)

Geschichte

- Entstanden 1987, Autor John Ousterhout (damals University of California at Berkeley)
- Erste Version 1990
- 1993 erste Tcl/Tk-Konferenz
- 1994 Ousterhout wechselt zu Sun, gründet Tcl-Abteilung
- 1997 ACM Software Award für Tcl/Tk
- 2012 Objekt-Orientierte Erweiterung

Datentypen

- Strings
- **Keine** numerische Datentypen (aber es gibt ein Kommando dafür)
- Assoziativlisten (`array`, wie in Python) und Listen

Syntax und Semantik

- Tcl-Programme (Skripte) bestehen aus einer Sequenz von **Kommandos** der Form

```
cmd arg1 arg2 arg3 ...
```

- Trennung der Kommandos durch Zeilenumbruch oder Semikolon
- Auswertung in drei Schritten:
  1. Gruppierung der Argumente (durch `{ ... }` oder `\" ... \"`)
  2. Substitution der Argumente
  3. Aufruf des Kommandos

Substitution

- Variablen (brauchen nicht deklariert zu werden)

```
set x 5
puts {x is $x}
puts "x is $x"
```

- Kommandosubstitution: `[ ... ]` (kann geschachtelt werden)

```
set x [string length "foo"]
```

- Backslashes: `\$`, `\n`, `\u001b` ...
- Gruppierung **vor** Substitution

Kommandos

- Variablenzuweisung: `set`
- Auswertung arithmetischer Ausdrücke: `expr`
- Auswertung eines Tcl-Kommandos: `eval`
- Fallunterscheidung und Schleifen:
  - `if`,
  - `while`, `foreach`, `for` `break`, `continue`
  - `if`, `while` erwarten numerische Argumente (`0` ist `false`, wie in C)
- Ausnahmen: `try` und `catch`
- Arraymanipulation: `array`
- Ein/Ausgabe: `gets`, `puts`
- Fallunterscheidung: `switch`
  - Auf Werten oder **regulären Ausdrücken**

Metaprogrammierung durch `eval`

- `eval` ruft den Tcl-Interpreter auf das Argument auf:

```
set cmd {puts {Hello, World!}}
...
eval $cmd
```

- Auswertung der Variablen zur **Ausführungszeit**:

```
set string "Hello, world!"
set cmd {puts $string}
unset string
eval $cmd
```

- Nutzung: **callbacks**, e.g. für GUI-Elemente

Beispiel

- Die Fakultätsfunktion, iterativ:

```
proc Factorial {x} {
  set i 1; set product 1
  while {$i <= $x} {
    set product [expr $product * $i]
    incr i
  }
  return $product
}
```

Beispiel

- Die Fakultätsfunktion, rekursiv:

```
proc Factorial {x} {
  if {$x <= 1} {
    return 1
  } else {
    return [expr $x * [Factorial [expr $x - 1]]]
  }
}
```

Die GUI-Bibliothek Tk

- Integraler Bestandteil der Sprache, und Grund für die Popularität
- Inzwischen etwas out-of-date, aber sehr stabil
- Grundprinzip:
  - GUI läuft asynchron
  - Erzeugt **Events**, an die **Callbacks** gebunden werden
- Eigene Bücherei, hat daher kein (kaum) natives Look&Feel
  - Sieht überall gleich (schlecht) aus

Tk im Beispiel: Hello, World!

- Ein erstes Beispiel:

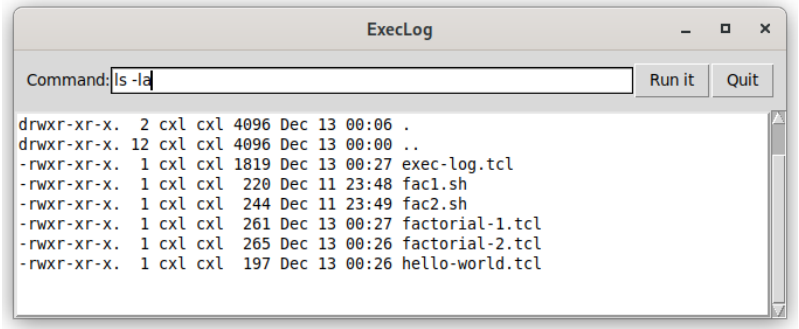
```
button .hello -text Hello \
              -command {puts stdout {Hello, World!}}
pack .hello -padx 20 -pady 10
```

- `button` deklariert Knopf

- `pack` plaziert Knopf im GUI

Ein längeres Beispiel

- `exec-log.tcl`



Steckbrief Tcl

Name	Tcl
Entstehung	Ab 1990 von John Ousterhout entwickelt
Programmierparadigma	Imperativ
Typen	Strings, Arrays, Listen
Typsystem	Ungetypt (schwach dynamisch getypt)
Parameterübergabe	call-by-name
Datenabstraktion	Wenig (lokale/global Variablen)
Anwendungsgebiet	"Glue language"
Besondere Kennzeichen	GUI-Bücherei Tk

Zusammenfassung

- Tcl hat **extrem simples** Programmiermodell, was auf der Auswertung von **Zeichenketten** basiert
- Drei Phasen des Programmablaufs:
  1. Gruppierung
  2. Substitution
  3. Ausführung
- Zu Tcl gehört das GUI-Toolkit Tk (Tcl/Tk)
- Damit schnelle Konstruktion von graphischen Benutzerschnittstellen für kommandozeilenbasierte Tools möglich.

Zusammenfassung

Zusammenfassung

- Skriptsprachen sind kein Programmierparadigma, eher eine Frage der Nutzpragmatik
- Verschiedene Kennzeichen von Skriptsprachen:
  - Kleiner Sprachkern, interpretiert
  - Schneller Entwicklungszyklus
  - Dient meist dazu, andere Programme zusammenzufügen
- Beispielsprachen: sh, Tcl
- **Achtung:** Übung am Donnerstag pandemiebedingt per **Zoom**

Programmiersprachen
Vorlesung 10 vom 20.12.21
Weihnachtsvorlesung

Christoph Lüth

Universität Bremen

Wintersemester 2021/22

Was machen wir heute?

- Geschichtliches
- Eine besondere Sprache

Geschichtliches

Was ist ein Computer?

- Ein Computer:
  1. ist elektronisch und digital;
  2. beherrscht die vier arithmetischen Operationen ($+$, $-$, $\cdot$, $/$);
  3. kann programmiert werden;
  4. erlaubt die Speicherung von Programmen und Daten.
- (2)– (4) garantiert **Turing-Vollständigkeit**.

Die ersten Computer

- Zuse Z3 (Zuse, 1941): sogar mit Fließkommarithmetik, aber nicht elektronisch;
- ENIAC (Mauchly, Eckert, von Neumann, 1946): nicht reprogrammierbar
- EDSAC (Wilkes, 1949): erfüllt alle vier Kriterien
- Thomas Watson, IBM (1943): “Es gibt einen Weltmarkt von 5 Computern”

Generationen von Programmiersprachen:

- Programmiert in **Maschinensprache**
  - Programmiersprachen der ersten Generation — 1 GL
- Symbolische Repräsentation: **Assemblersprachen** (2 GL)
- Hochsprachen: 3GL
- 4GL: Nicht exakt definiert
  - "Programming without the Programmer", CASE, ...
  - Model-driven development, DSLs

Die erste Hochsprachen: FORTRAN

- Hardware war **teuer** als Arbeitskraft — deshalb **Programmeffizienz** wichtig
- FORTRAN (1957): FORmula TRANslator
- Erste Programmiersprache mit symbolischer Notation `a*2+ b`
- Entwickelt für numerische Berechnungen
- Lief auf der IBM 704

John Backus
(1924–2007)

- Turing Award (1977)
- FORTRAN
- Backus-Naur-Form

Die ersten Hochsprachen: COBOL

- COBOL
- Erste Sprache für Business-Anwendungen
- Standardisiert, Design by committee

Grace Hopper
(1906–1992)

- Erfand “Bugs”
- Allgemeinverständliche Programmierung
- Beeinflusste COBOL

Die ersten Hochsprachen: ALGOL

- ALGOL (Algorithmic Language): Ursprung der modernen Programmiersprachen
- Erste Version 1958, 1960 (ALGOL-60), spätere Versionen (ALGOL-68)
- Amerikanisch-Europäische Koproduktion
- Erste Sprachen mit formal definierter Grammatik (BNF), Blöcken, strukturierter Programmierung, call-by-name
- "ALGOL-Syntax":

```
for p := 1 step 1 until n do
  for q := 1 step 1 until m do
    if abs(a[p, q]) > y then
      begin y := abs(a[p, q]);
        i := p; k := q
      end
end Absmax
```

Funktionale Sprachen: LISP

- LISP (LISt Processor):
  die erste funktionale Sprache
- 1960 von John McCarthy am MIT
  entworfen
- Speziell für nicht-numerische Probleme (KI)

- Basiert auf dem Lambda-Kalkül
- Alles ist eine S-Expression

LISP-Maschine:
Symbolics 3640

John McCarthy
(1927–2011)

- Turing Award (1971)
- LISP
- Pionier der KI

Die 1970er Jahre

- C (Dennis Ritchie & Ken Thompson, 1972):
  - Portable Assembler-Sprache
  - Implementationssprache für UNIX
- Pascal (Niklaus Wirth, 1970):
  - Virtuelle Maschine (P-Code)
  - Strukturiert, block-orientiert, stark getypt
- Smalltalk (Alan Kay, 1970):
  - Objektorientiert, GUI integriert
- ML (Robin Milner, 1974):
  - Für den LCF-Beweiser, Hindley-Milner-Polymorphie
  - Standardisiert (1983), mathematisch formal definierte Semantik (1997)
- Prolog (Bob Kowalski, 1974):
  - Logische Programmierung, Ausführung durch **Resolution**

Die 1980er und 1990er Jahre

- C++ (Stroustrup, 1986): objektierientierte Erweiterung von C
- Ada (1983): vom DoD standardisiert, sehr komplexer Standard.
  - Erster Compiler 1986
- Erlang (Armstrong, 1986–92): für verteilte und nebenläufige Applikationen, Fa. Ericsson
- Java (Gosling *et al*, 1990): zuerst “Oak”, für Set-Top-Boxen.
  - Objektorientiert, JVM, Applets — portabel und sicher
- Haskell (1987, 1998): nicht-strikt und funktional,
- Python (van Rossum, 1991): leichtgewichtig, einfach zu nutzen
- JavaScript (Eich, 1995): Skriptsprache für den Browser

Eine besondere Sprache

Brainfuck

- Brainfuck wurde 1993 von Urban Müller erfunden, um den “kleinstmöglichen Compiler für eine Turing-vollständige Sprache” zu schreiben.
- Abgeleitet von P'' (Corrado Boehm, 1964)
- Acht Kommandos, Turing-vollständig
- Kryptisch und von keinem praktischen Nutzen

Ein Beispielprogramm

- Hello, world:

```
++++++++
[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]
>>.>---.+++++++..+++.>>.<-.
<.+++.------.--------.>>+.>++.
```

Die Sprache

- Syntak: acht Kommandozeichen
- Lexikalik: alles andere ist Kommentar
- Ausführungsmodell:
  - Maschine mit Programm und Instruktionszeiger, Datenzeiger und 30.000 Speicherzellen

Kommandos

<	Datenzeiger erhöhen
>	Datenzeiger erniedrigen
+	Aktueller Zellenwert erhöhen
-	Aktueller Zellenwert erniedrigen
.	Aktueller Zellenwert ausgeben
,	Aktueller Zellenwert einlesen
[	Springt hinter das entsprechende] wenn aktueller Zellenwert 0 ist
]	Springt hinter das entsprechende [wenn aktueller Zellenwert nicht 0 ist

- `[P]` ist Iteration von P solange aktueller Zellenwert ungleich 0 ist.

Einfache Programme:

- `,[.,]`

Einfache Programme:

- `,[.,]` Echo
- `[>+<-]`

Einfache Programme:

- `,[.,]` Echo
- `[>+<-]` Addition `p[i+1]= p[i+1]+p[i]`
- `[>-<-]`

Einfache Programme:

- `,[.,]` Echo
- `[>+<-]` Addition `p[i+1]= p[i+1]+p[i]`
- `[>-<-]` Subtraktion `p[i+1]= p[i+1]-p[i]`
- `>[-]<[>+<-]`

Einfache Programme:

- `,[.,]` Echo
- `[>+<-]` Addition `p[i+1]= p[i+1]+p[i]`
- `[>-<-]` Subtraktion `p[i+1]= p[i+1]-p[i]`
- `>[-]<[>+<-]` Verschieben `p[i+1]= p[i]; p[i]= 0`

Größere Programme

- Kopieren:

```
>[-]>[-]<<          Initialisierung
[>+>+<<-]           Verschiebe p[i] nach p[i+1],p[i+2]
>>[<<+>>-]          Verschiebe p[i+2] nach p[i]
<<
```

Zusammenfassung

- Brainfuck ist:
  - Turing-vollständig
  - extrem kompliziert zu benutzen
  - extrem einfach zu implementieren
  - in der Praxis unbrauchbar


Frohe Weihnachten und einen Guten Rutsch!