Einleitung

Was ist ein Paradigma?

- Paradigma aus dem Altgriechischen "Beispiel, Muster", Erzählung mit beispielhaftem Charakter (laut Duden)
- Programmierparadigmen beschreiben grundsätzliche Arten wie Computer-Programme formuliert werden können
- Programmiersprachen können einzelne oder viele Konzepte aufgreifen
  - Keine verbreitete Sprache greift alle behandelten Konzepte auf
  - Betrachtung unterschiedlicher Sprachen
- Warum unterschiedliche Paradigmen? Komplexität von Software schlecht beherrschbar

Was bedeutet das?

- Programmierer schreiben, testen und dokumentieren (nur) zwischen 325 und 750 Codezeilen pro Monat
- Komplexität muss verborgen werden, z.B. durch
  - Kapselung
  - Spezifische Spachkonstrukte, Domain Specific Languages
  - Ausdrucksstärkere Sprachen
- Entwicklung neuer Programmierparadigmen hilft Grenzen (ein wenig) zu verschieben
- Theoretische Rahmenbedingungen (Turing-Mächtigkeit, Satz von Rice) behalten Gültigkeit!

Welche Paradigmen existieren?

- Grundlegend
  - Imperative Algorithmen
  - Applikative Algorithmen
  - Deduktive Algorithmen
- aber Vielzahl weiterer Formen
  - teilweise ergänzend, unterschiedliche Kategorisierung möglich
  - Bsp: prozedural, deklarativ, objekt-orientiert, datenstromorientiert, parallele & verteilte Programmierung...
- Teilweise unterschiedliche Bezeichnungen
  - Applikativ bzw. Funktional
  - Deduktiv bzw. Logisch
- Aktueller Trend: Multiparadigmen-Sprachen
  - Umsetzung unterschiedlichster Paradigmen in einer Sprache
  - Beispiele: Scala, neuere C++-Standards, ...

Objektorientierung und weiterführende Konzepte

Ziele

Einführen von Mechanismen zur Handhabung von komplexeren Code

- Systematisiertes & schnelles Testen
- Inspektion/Veränderung von Code zur Laufzeit
- Zusichern von Bedingungen
- Fehlerbehandlung
- Typsicherheit
- Generische und wiederverwendbare Algorithmen

Unit-Testing

Motivation

- Große Software-Systeme entwickeln sich über lange Zeiträume
- Wie können Änderungen an komplexen Code-Basen beherrscht werden?
- Veränderung über Zeit + Komplexität der Software
  - Änderungen führen möglicherweise zu Auswirkungen, die für Einzelne nicht immer überschaubar sind
  - Software muss nach Änderung von Grund auf durchgetestet werden
- Verbreitetes Vorgehen: zusätzlichen Code schreiben, der eigentlichen Code automatisch "überprüft"
  - Nicht vollständig möglich (z.B. Halteproblem)
  - Eher Heuristik
- Test-Code wird bei Ereignissen oder periodisch ausgeführt
  - Vor Releases, nach Commit in Repository, während der Entwicklung ...

Eigenschaften von Unit-Tests

- Software schlecht als Ganzes testbar → Zergliederung von Software in sinnvolle Einheiten
- Individuelle Tests dieser Einheiten
- Dabei: reproduzierbar & vollautomatisierbar
  - Ziel: Wann immer Änderungen in komplexen Programmen vorgenommen werden, möglichst vollständiger Test, da Programmierer nicht mehr alles überblicken
- Messung der Vollständigkeit der Tests schwierig
- Üblich: Messung von Überdeckung (Coverage) in Bezug auf Anzahl Funktionen, Code-Zeilen oder Verzweigungen
- Gute Praxis: Wenn ein Bug beim Testen oder Live-Betrieb auftritt → Schreiben eines zusätzlichen Tests, um Wiederauftreten zu erkennen

Richtiges Abstraktionsniveau bei Unit Testing

- Um die Tests auszuführen, müssen jeweils entsprechende Hauptprogramme generiert werden ("Test Suites")
- Hauptschwierigkeiten von Unit-Tests:
  - Richtiges Abstraktionsniveau
  - "Herauslösen"von zu testendem Code aus Umgebung
- Zwei wesentliche Möglichkeiten:
  - Individuelles Testen von Klassen:
    - Vernachlässigt Zusammenspiel zwischen Klassen
    - Oft sehr aufwändig, da andere Klassen für Unit-Tests nachgebildet werden müssen (Mocks)
    - Was bei zyklischen Abhängigkeiten?
  - Gemeinsames Testen von Klassen:
    - Erfordert Eingreifen in gekapselte Funktionalitäten
    - Private & Protected Member-Variablen & Methoden!
    - Eigentlich nicht möglich?!

Klasse

```java
public class Multi {
    int mul(int a, int b) {
        return a * b;
    }
}
```

Testklasse

```java
import static org.junit.jupiter.api.Assertions.*;
class MultiTest {
    @org.junit.jupiter.api.Test
    void mul() {
        Multi m = new Multi();
        assertEquals(m.mul(1,2), 2, "should work");
        assertEquals(m.mul(2,0), 1, "explodes");
    }
}
```

Reflections

- Normaler Ablauf: Programm schreiben, compilieren, ausführen
  - Aber was wenn ich ein Programm zur Laufzeit inspizieren oder verändern möchte?
- Unterschiedliche Gründe
  - Testen (um Fehler zu injizieren!)
  - Fehlersuche ("Debugging")
  - Nachladen von Plugins zur Modularisierung von Programmen
  - Serialisierung/Deserialisierung von Code
  - "Patchenßur Laufzeit
  - Erkunden der Ablaufumgebung (z.B. OS-/Shared-Library Version)
- Benötigt die Fähigkeit, im Programm Codestruktur zu analysieren und ggf. zu verändern:
  - Typisch: Abruf Klassenhierarchie, Auflisten von Methoden und Parametern, Austausch von Klassen und Methoden
  - Teil von Java, Python, ...

API verstreut über verschiedene Packages, z.B. java.lang.Class, java.lang.instrument, java.lang.reflect

```
Class cls = "test".getClass();
System.out.println("Die Klasse heisst " + cls.getName());
// Die Klasse heisst java.lang.String
```

```
// import java.lang.reflect.Method;
Method[] methods = cls.getMethods();
for (Method m : methods)
System.out.println(m.getName());
```

Kurz: Programm zur Laufzeit inspizieren oder verändern

```
static class Foo {
    private String h = "Hallo";
    public void greet() { System.out.println(h); }
}
public static void main(String[] args) {
    Foo foo = new Foo();
    foo.greet();
    try {
        Field f = foo.getClass().getDeclaredField("h");
        f.setAccessible(true);
        f.set(foo, "Moin");
    } catch (Exception e) {
    }
    foo.greet();
}
```

Annotationen

- Annotationen erlauben Anmerkungen an Klassen & Methoden
- Beginnen mit @
- Einige wenige vordefinierte z.B. @Override
- Aber auch eigene; u.a. durch Reflections abrufbar
- Häufig genutzt, wenn zusätzlicher Code geladen wird (Java EE)
- Oder um Unit-Tests zu markieren...

Nachteile:

- Geringe Geschwindigkeit weil Zugriff über Programmcode erfolgt
- Kapselung kann umgangen werden

```
class MultiTest {
@org.junit.jupiter.api.Test
void mul() {
  ...
```

Reflektionen über Reflections

- Reflections sind ein sehr mächtiges Werkzeug, aber Einsatz sollte wohldosiert erfolgen
- Nachteile:
  - Geringe Geschwindigkeit weil Zugriff über Programmcode erfolgt
  - Kapselung kann umgangen werden
    - private, protected und final können entfernt werden
    - Aufruf/Veränderung interner Methoden & Auslesen/Veränderung interner Variablen
    - Synchronisation zwischen externen und internen Komponenten bei Weiterentwicklung?
  - Debugging veränderter Programme?
  - Sicherheit?!
- Verwandte Techniken:
  - Monkey Patching (JavaScript-Umfeld)
  - Method Swizzling (Swift/Objective-C-Umfeld)

Assertions/Pre-/Postconditions/Invarianten

- Kann man interne Zustände testen, ohne invasive Techniken wie Reflections?
- Einfache Möglichkeit: An sinnvollen Stellen im Programmcode testen, ob Annahmen/Zusicherungen (Assertions) stimmen...
- Tests, die nie falsch sein sollten
  - Erlauben gezielten Programmabbruch, um Folgefehler zu vermeiden
  - Erlauben gezieltes Beheben von Fehlern
  - Gemeinsames Entwickeln von Annahmen und Code

```
class Stack {
public void push(Object o) {
  ...
  if(empty() == true) // es sollte ein Objekt da sein
    System.exit(-1);
}
...
}
```

Aber: Ausführungsgeschwindigkeit niedriger

- Zeitverlust stark abhängig von Programm/Programmiersprache
- Verbreitetes Vorgehen:
  - Aktivieren der Tests in UnitTests und Debug-Versionen
  - Deaktivieren in Releases
- Wann Assertion hinzufügen? ËigentlichRegel: beim Gedanken ëigentlich müsste hier ... gelten"hinzufügen

- Aktivierung der Tests über Start mit java -ea
- Benötigt spezielle ïfBedingung: assert

```
class Stack {
    public void push(Object o) {
        ...
        assert empty() == false
    }
}
```

Welche braucht man?

- Woran erkennt man beim Programmieren bzw. (erneutem) Lesen von Code, dass man eine Assertion hinzufügen sollte?
- Eine einfache Heuristik - Die ËigentlichRegel:
  - Wenn einem beim Lesen von Programmcode ein Gedanke der Art Ëigentlich müsste an dieser Stelle XY gelten"durch den Kopf geht,
  - dann sofort eine entsprechende Assertion formulieren!

Spezielle Assertions: Pre- & Postconditions

- Methoden/Programmabschnitte testen Bedingung vor und nach Ausführung
- Einige Sprachen bieten spezialisierte Befehle: requires und ensures
- Bei OO-Programmierung sind Vor- und Nachbedingungen nur eingeschränkt sinnvoll
  - Bedingungen oft besser auf Objekt-Ebene → interner Zustand
- An welchen Stellen ist es sinnvoll, Annahmen zu prüfen?
- Einfache Antwort: an so vielen Stellen wie möglich
- Komplexere Antwort: Design by contract, ursprünglich Eiffel
- Methoden/Programmabschnitte testen Bedingung vor und nach Ausführung
- Einige Sprachen bieten spezialisierte Befehle: requires und ensures
- Ziel mancher Sprachen: Formale Aussagen über Korrektheit

```
class Stack {
  public void push(Object o) {
    assert o != null // precondition
    ...
    assert empty() == false // postcondition
  }
...
}
```

Klasseninvarianten

- Bei OO-Programmierung sind Vor- und Nachbedingungen nur eingeschränkt sinnvoll
- Bedingungen oft besser auf Objekt-Ebene → interner Zustand
- Invarianten spezifizieren Prüfbedingungen
- In Java nicht nativ unterstützt
  - Erweiterungen, wie Java Modeling Language
  - Simulation

```
class Stack {
void isValid() {
  for(Object o : _objs) // Achtung: O(n) Aufwand!
    assert o != null
}
public void push(Object o) {
  isValid() // always call invariant
  ...
  isValid() // always call invariant
}
```

Exceptions

- Wie wird mit Fehlern umgegangen?
- gut für Code-Komplexität: Fehlerprüfungen an zentralerer Stelle
  - Abbrechen und Programm-Stack äbbauen"bis (zentralere) Fehlerbehandlung greift
  - Dabei Fehler sinnvoll gruppieren
- Java (und viele mehr): try/catch/throw-Konstrukt
  - throw übergibt ein Objekt vom Typ Throwable an Handler, dabei zwei Unterarten:
  - Error: Sollte nicht abgefangen werden z.B. Fehler im Byte-Code, Fehlgeschlagene Assertions
  - Exceptions: Programm muss Exception fangen oder in Methode vermerken,
    - Checked Exception: Programm muss Exception fangen oder in Methode vermerken
    - Runtime Exceptions: Müssen nicht (aber sollten) explizit behandelt werden, bspw. ArithmeticException oder IndexOutOfBoundsException

```
private void readFile(String f) {
  try {
    Path file = Paths.get("/tmp/file");
    if(Files.exists(file) == false)
      throw new IOException("No such dir");
    array = Files.readAllBytes(file);
  } catch(IOException e) {
    // do something about it
  }
}
```

Checked Exceptions

Deklaration einer überprüften Exception:

```
void dangerousFunction() throws IOException {
  ...
  if(onFire)
    throw IOException("Already burns");
  ...
}
```

Die Deklaration mit "throws IOException"lässt beim build mögliche Fehler durch IOExceptions dieser Funktion zu, diese müssen durch die aufrufende Methode abgefangen werden. Aufrufe ohne try-catch-Block schlagen fehl! Sollte man checked oder unchecked Exceptions verwenden?

- Checked sind potenziell sicherer
- Unchecked machen Methoden lesbarer
- Faustregel unchecked, wenn immer auftreten können (zu wenig Speicher, Division durch 0)

Abfangen mehrerer unterschiedlicher Exceptions

```
try {
  dangerousFunction();
} catch(IOException i) {
  // handle that nasty error
} catch(Exception e) {
  // handle all other exceptions
}
```

Aufräumen nach einem try-catch-Block: Anweisungen im finally-Block werden immer ausgeführt, d.h. auch bei return in try- oder catch-Block (oder fehlerloser Ausführung)

```
try {
    dangerousFunction();
} catch(Exception e) {
    // handle exceptions
    return;
} finally {
    // release locks etc..
}
```

Generizät von Datentypen

(Typ-)Generizität:

- Anwendung einer Implementierung auf verschiedene Datentypen
- Parametrisierung eines Software-Elementes (Methode, Datenstruktur, Klasse, ...) durch einen oder mehrere Typen

Beispiel:

```
int min(int a, int b) {
  return a < b ? a : b;
}
float min(float a, float b) {
  return a < b ? a : b;
}
String min(String a, String b) { // lexikographisch
  return a.compareTo(b) < 0 ? a : b;
}
```

Grenzen von Typsubstitution

Kann ein Objekt einer Oberklasse (eines Typs) durch ein Objekt seiner Unterklasse (Subtyps) ersetzt werden?
Möglicher Ausweg: Klassenhierarchie mit zentraler Basisklasse

```
void sort(Object[] feld) { ... }            //z.B. java.lang.Object
void sort(java.util.Vector feld) { ... }    //alternativ (nutzt intern Object)
```

Möglicher Ausweg 2: Nutzung primitiver Datentypen nicht direkt möglich

```
Object[] feld = new Object[10];          //Object[] != int[]
feld[0] = new Integer(42);
int i = ((Integer) feld[0]).intValue(); //erfordert Wrapper-Klassen wie java.lang.Integer
```

Weiteres Problem: Typsicherheit
Typ-Substituierbarkeit: Kann ein Objekt einer Oberklasse (eines Typs) durch ein Objekt seiner Unterklasse (Subtyps) ersetzt werden?
Beispiel (isSubtyp): short → int → long
Viele Programmiersprachen ersetzen Typen automatisch, d.h. diese wird auch für shorts und ints verwendet

```
long min(long a, long b) {
  return a < b ? a : b;
}
```

Kreis-Ellipse-Problem: Modellierung von Vererbungsbeziehungen

- Ïst ein Kreis eine Ellipse?Oder eine Ellipse ein Kreis?"
- Annahme: Kreis := Ellipse mit Höhe = Breite

```
Circle c = new Circle();
c.skaliereX(2.0);        //skalieren aus Klasse Circle
c.skaliereY(.5);         //is das noch ein Kreis?
```

evtl. Reihenfolge in der Klassenhierarchie tauschen (nutzung von Radius)? Was bedeutet das für Ellipse? Verwandte Probleme: Rechteck-Quadrat, Set-Bag

Ko- und Kontravarianz

Geg.: Ordnung von Datentypen von spezifisch → allgemeiner

- Gleichzeitige Betrachtung einer Klassenhierarchie, die Datentypen verwendet
  - Kovarianz: Erhaltung der Ordnung der Typen
  - Kontravarianz: Umkehrung der Ordnung
  - Invarianz: keines von beiden
- In objektorientierten Programmiersprachen im Allgemeinen
  - Kontravarianz: für Eingabeparameter
  - Kovarianz: für Rückgabewerte und Ausnahmen
  - Invarianz: für Ein- und Ausgabeparameter
- Anwendung für
  - Parameter
  - Rückgabetypen
  - Ausnahmetypen
  - Generische Datenstrukturen

Beispiel: Basierend auf Meyer's SKIER-Szenario

```
class Student {
  String name;
  Student mate;
  void setRoomMate(Student s) { ... }
}
```

Wie überschreibt man in einer Unterklasse Girl oder Boy die Methode ßetRoomMateïn elternfreundlicher Weise? Von Eltern sicher gewollt - Kovarianz:

```
class Boy extends Student {
  void setRoomMate(Boy b) { ... }
}
class Girl extends Student {
  void setRoomMate(Girl g) { ... }
}
```

Was passiert mit folgendem Code?

```
Boy kevin = new Boy("Kevin");
Girl vivian = new Girl("Vivian");
kevin.setRoomMate(vivian);
```

- Verwendet setRoomMate der Basisklasse
- setRoomMate Methoden der abgeleiteten Klassen überladen nur Spezialfälle → gültig
- In C++ und Java keine Einschränkung der Typen zur Compile-Zeit
- Kovarianz so nur in wenigen Sprachen implementiert (z.B. Eiffel über redefine); Überprüfung auch nicht immer statisch!
- Auch bekannt als catcall-Problem (cat = changed availablility type)

Ausweg: Laufzeitüberprüfung

```
class Girl extends Student {
  ...
  public void setRoomMate(Student s) { //student wird aufgerufen! nicht boy oder girl, dadurch koennen die
    methoden der klasse verwendet werden
    if (s instanceof Girl)
      super.setRoomMate(s);
    else
      throw new ParentException("Oh Oh!");
  }
}
```

Nachteil: Nur zur Laufzeit überprüfung

Ko- und Kontravarianz für Rückgabewerte

Kovarianz (gängig):

```
public class KlasseA {
  KlasseA ich() { return this; }
}
public class KlasseB extends KlasseA {
  KlasseB ich() { return this; }
}
```

Kontravarianz macht wenig Sinn und kommt (gängig) nicht vor

Liskovsches Substitutionsprinzip (LSP)

Barbara Liskov, 1988 bzw. 1993, definiert stärkere Form der Subtyp-Relation, berücksichtigt Verhalten:
Wenn es für jedes Objekt o_1 eines Typs S ein Objekt o_2 des Typs T gibt, so dass für alle Programme P, die mit Operationen von T definiert sind, das Verhalten von P unverändert bleibt, wenn o_2 durch o_1 ersetzt wird, dann ist S ein Subtyp von T. Subtyp darf Funktionalität eines Basistyps nur erweitern, aber nicht einschränken.
Beispiel: Kreis-Ellipse → Kreis als Unterklasse schränkt Funktionalität ein und verletzt damit LSP
Typsicherheit

```
String s = GMethod.<String>thisOrThat("Java", "C++");
Integer i = GMethod.<Integer>thisOrThat(new Integer(42), new Integer(23));
```

Generics in Java (Typsicherheit)

Motivation: Parametrisierung von Kollektionen mit Typen

```
LinkedList<String> liste = new LinkedList<String>();
liste.add("Generics");
String s = liste.get(0);
```

auch für Iteratoren nutzbar

```
Iterator<String> iter = liste.iterator();
while(iter.hasNext()) {
  String s = iter.next();
  ...
}
```

oder mit erweiterter for-Schleife

```
for(String s : liste) {
  System.out.println(s);
}
```

Deklaration: Definition mit Typparameter

```
class GMethod {
  static <T> T thisOrThat(T first, T second) {
    return Math.random() > 0.5 ? first : second;
  }
}
```

- T = Typparameter (oder auch Typvariable) wird wie Typ verwendet, stellt jedoch nur einen Platzhalter dar
- wird bei Instanziierung (Parametrisierung) durch konkreten Typ ërsetzt"
- nur Referenzdatentypen (Klassennamen), keine primitiven Datentypen Anwendung:
- explizite Angabe des Typparameters

```
String s = GMethod.<String>thisOrThat("Java", "C++");
Integer>thisOrThat(new Integer(42), new Integer(23));

```

- automatische Typinferenz durch Compiler

```
String s = GMethod.thisOrThat("Java", "C++");
Integer i = GMethod.thisOrThat(new Integer(42), new Integer(23));

```

Eingrenzung von Typparametern

Festlegung einer Mindestfunktionalität der einzusetzenden Klasse, z.B. durch Angabe einer Basisklasse

- Instanziierung von T muss von Comparable abgeleitet werden (hier ein Interface, dass wiederum generisch ist, daher Comparable¡T¿)
- Verletzung wird vom Compiler erkannt

```
static<T extends Comparable<T>> T min(T first, T second) {
  return first.compareTo(second) < 0 ? first : second;
}
```

Angabe des Typparameters bei der Klassendefinition:

```
class GArray<T> {
  T[] data;
  int size = 0;
  public GArray(int capacity) { ... }
  public T get(int idx) { return data[idx]; }
  public void add(T obj) { ... }
}
```

Achtung: new $T[n]$ ist unzulässig! Grund liegt in der Implementierung von Generics
Es gibt zwei Möglichkeiten der internen Umsetzung generischen Codes:

- Code-Spezialisierung: jede neue Instanziierung generiert neuen Code
  - Array¡String¿ → ArrayString, Array¡Integer¿ → ArrayInteger
  - Problem: Codegröße
- Code-Sharing: gemeinsamer Code für alle Instanziierungen
  - Array¡String¿ → Array¡Object¿, Array¡Integer¿ → Array¡Object¿
  - Probleme: keine Unterstützung primitiver Datentypen & keine Anpassung von Algorithmen an Typ

Java: Code-Sharing durch Typlöschung (Type Erasure)
Typen beim Übersetzen geprüft, aber keinen Einfluss auf Code
sichert auch Kompatibilität zu nicht generischem Code (Java-Version ¡ 1.5)
Bsp.: ArrayList vs. ArrayList¡E¿
Beispiel: Reflektion (Metaklassen) zur Erzeugung nutzen; danach Konstruktionsaufruf

```
public GArray(Class<T> clazz, int capacity) {
  data = (T[]) Array.newInstance(clazz, capacity);
}
GArray<String> array = new GArray<String>(String.class, 10);
```

Kovarianz generischer Typen

einfache Felder in Java sind kovariant

```
Object[] feld = new Object[10];
feld[0] = "String";
feld[1] = new Integer(42);
```

Instanziierungen mit unterschiedliche Typen sind jedoch inkompatibel

```
GArray<String> anArray = new GArray<String>();
GArray<Object> anotherArray = (GArray<Object>) anArray;
```

Wildcards

Wildcard "?äls Typparameter und abstrakter Supertyp für alle Instanziierungen

```
GArray<?> aRef;
aRef = new GArray<String>();
aRef = new GArray<Integer>();
```

aber nicht:

```
GArray<?> aRef = new GArray<?>();
```

hilfreich insbesondere für generische Methoden

```
// dieser Methode ist der genaue Typ egal
static void pO(GArray<?> ia) {
  for(Object o : ia) {
    System.out.print(o);
  }
}
// floats wollen wir mit Genauigkeit 2 haben
static void pF(GArray<Float> ia) {
  for(Float f : ia) {
    System.out.printf("%5.2f\n", f);
  }
}
```

Beschränkte Wildcards

- nach üntenïn der Klassenhierarchie → Kovarianz

```
? extends Supertyp
```

- Anwendungsbeispiel: Sortieren eines generischen Feldes erfordert Unterstützung der Comparable-Schnittstelle

```
void sortArray(GArray<? extends Comparable> array) {
  ...
}
```

- nach öbenïn der Klassenhierarchie → Kontravarianz

```
? super Subtyp

```

- Anwendungsbeispiel: Feld mit ganzen Zahlen und Objekten

```
GArray<? super Integer> array;
// Zuweisungskompatibel zu ...
array = new GArray<Number>();
array = new GArray<Object>();
array = new GArray<Integer>();
// aber nur erlaubt:
Object obj = array.get(0);

```

PECS = Producer extends, Consumer super → Producer liest nur sachen, Consumer legt daten/Objekte/... ab

Objektorientierung am Beispiel C++

- Ziel von C++: volle Kontrolle über Speicher & Ausführungsreihenfolgen sowie skalierbarere Projekt-Größe
- Kompiliert zu nativem Maschinencode und erlaubt genauere Aussagen über Speicher-, Cache- und Echtzeitverhalten
- Viele Hochsprachenelemente
- Jedoch kompromissloser Fokus Ausführungsgeschwindigkeit, d.h.
  - Keine automatische Speicherverwaltung
  - Keine Initialisierung von Variablen (im Allgemeinen)
  - Kein Speicherschutz!
  - Dinge, die Zeit kosten, müssen im Allgemeinen erst durch Schlüsselworte aktiviert werden
- C++ ist zu sehr großen Teilen eine Obermenge von C
  - Fügt Objektorientierung hinzu
  - Versucht fehleranfällige Konstrukte zu kapseln
  - Führt (viele) weitere Sprachkonstrukte ein, die Code kompakter werden lassen

”C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.”[Bjarne Stroustrup]
Wichtige Makrobefehle:

```
#include "X.hpp" // Datei X.hpp aus Projekt-Ordner
#include <cstdio> // Datei cstdio aus System-Includes

#ifdef DEBUG // falls Konstante DEBUG definiert ist
std::cout << "Wichtige Debugausgabe" << std::endl;
#endif

#define DEBUG // Konstante setzen
#define VERSION 3.1415 // Konstante auf einen Wert setzen
#define DPRINT(X) std::cout << X << std::endl; // Macro-Fkt.
#undef DEBUG // Konstante löschen, good practice!

#ifndef __linux__ // falls nicht für Linux übersetzt
playMinesweeper();
#endif
```

C++ Klassen

Header Foo.hpp deklariert Struktur und Schnittstelle

```
public:      // Block ohne Zugriffsbeschränkung
Foo();     // Konstruktor
~Foo();    // Destruktor
protected:  // Block von Dingen, auf die auch abgeleitete Klassen zugreifen dürfen
int num;  // Member-Variable
```

Implementierung in getrennter Datei Foo.cpp

```
#include "Foo.hpp"  // Klassen Deklaration einbinden
#include <iostream> // Einbinden von Funktionen der stdlib
Foo::Foo() :        // Implementierung des Konstuktors von Foo
  num(5) {          // Statische Initialisierung von num, Code in Klammern {} kann auch   initialisieren
  std::cout << "c" << std::endl;
}
Foo::~Foo() {
  std::cout << "d" << std::endl;
}
```

- Reine Implementierung auch im Header möglich, aber Trennung von Implementierung und Deklaration erlaubt schnelleres Kompilieren
- Trennung nicht immer möglich (später mehr Details), aber im Allgemeinen zu bevorzugen
- Der scope-Operator :: wird zum Zugriff auf namespaces und zur Beschreibung der Klassenzugehörigkeit von Methoden verwendet
- Initialisierung von Variablen vor Funktionsrumpf etwas ”merkwürdigßu lesen, aber erlaubt schnelle Implementierungen...
  - Syntax: nach Konstruktor : dann jeweils Variable(Wert)
  - Variablen durch , getrennt
  - Wichtig: Reihenfolge der Variablen wie in Deklaration der Klasse!
- Schlüsselworte private, protected und public vergleichbar zu Java, werden aber vor ganze Blöcke geschrieben
  - Kapselung nur auf Ebene von Klassen → Klassen sind immer public
  - protected erlaubt nur der Klasse selber und Unterklassen den Zugriff
- Zugriffe außerhalb der Klassenstruktur können durch friend- Deklaration erlaubt werden (teilweise verrufen!)
- Auch final ähnlich zu Java → Verhindert weiteres Ableiten von Klassen
- Schlüsselwort const markiert Methoden, die Objekte nicht verändern → Erlauben die Übergabe von Nur-Lesen-Referenzen
- Größere Unterschiede zu Java:
  - Klassen können Destruktoren besitzen
    - Werden aufgerufen wenn Objekt zerstört wird
    - Kann bspw. dafür verwendet werden, um von dem Objekt allozierte Speicherbereiche freizugeben (Achtung: anschließend darf auf diese nicht mehr zugegriffen werden - problematisch wenn anderen Objekte diese Speicherbereiche bekannt gegeben wurden!)
    - Destruktor kann Zerstören eines Objekts aber nicht verhindern
    - Methodensignatur Klassenname() - kein Rückgabetyp!
    - Warum gibt es das nicht in Java?
  - Neben dem Standardkonstruktor oder einem expliziten Konstruktor existiert ein Copy-Constructor
    - Methodensignatur Klassenname(const Klassenname& c)
    - Wird aufgerufen wenn Objekt kopiert werden soll
    - Vergleichbar zu Object.clone() in Java
- Überladen von Methoden vergleichbar zu Java
  - Parametertypen (oder const-Markierung) müssen sich unterscheiden!
  - Nur Veränderung des Rückgabewertes nicht ausreichend

```
class Foo {
  public:
    void doMagic(int i);
    void doMagic(std::string s);
};

```

C++ Präprozessor

C/C++-Code kann vor dem Übersetzen durch einen Präprozessor verändert werden

- Alle Präprozessor-Makros beginnen mit #
- (Haupt-)gründe:
  - Importieren anderer Dateien
  - An- und Ausschalten von Features je nach Compile-Optionen
  - Kapselung von Plattform-spezifischem Code

  - Vermeiden von Redundanzen
- Makros sollten vermieden werden
  - Schwierig zu lesen
  - Keine Namespaces
  - Keine Typsicherheit
- Manchmal jedoch einzige Möglichkeit

Beispiele:

```
#include "X.hpp" // Datei X.hpp aus Projekt-Ordner
#include <cstdio> // Datei cstdio aus System-Includes

#ifdef DEBUG // falls Konstante DEBUG definiert ist
std::cout << "Wichtige Debugausgabe" << std::endl;
#endif

#define DEBUG // Konstante setzen
#define VERSION 3.1415 // Konstante auf einen Wert setzen
#define DPRINT(X) std::cout << X << std::endl; // Macro-Fkt.
#undef DEBUG // Konstante löschen, good practice!

#ifndef __linux__ // falls nicht für Linux übersetzt
playMinesweeper();
#endif
```

Include Guards

Eine (oft hässliche) Eigenschaft des #include-Befehls: kein Überprüfen ob eine Datei vorher bereits eingebunden wurde. Problematisches Beispiel:

```
#include "Bar.hpp" //in "Bar.hpp" ist "Foo.hpp" bereits inkludiert worden
#include "Foo.hpp" //Fehler weil kallse Foo bereits deklariert wurde
```

Common Practice: Include-Guards um alle Header-Dateien

```
#ifndef FOO_HPP
#define FOO_HPP
...
#endif
```

Speichermanagement

- Programmspeicher enthält Code und Daten, vom Betriebssystem i.A. auf virtuelle Adressbereiche abgebildet
- Unterschiedliche Varianten von Datenspeicher:
  - Stack hält alle Variablen einer Methode, aller aufrufenden Methoden, Parameter, Rückgabewerte und einige Management-Daten
  - Heap hält Variablen und Objekte, die nicht direkt über Methodenaufrufe übergeben werden
  - Speicher für globale und statische Objekte und Variablen
- Java legt primitive Datentypen im Stack ab und Objekte im Heap
- C++ kann sowohl primitive Datentypen als auch Objekte in Stack und Heap abbilden
- Für den Stack bieten Java und C++ automatisches Speicher-Mgmt.
- Für den Heap bietet nur Java automatisches Speicher-Mgmt.

Eigenschaften des Stack-Speichers

- Variablen/Objekte haben klare Lebensdauer → Werden immer gelöscht wenn Funktion verlassen wird → Man kann Speicher nicht äufheben"
- In der Regel sehr schnell, weil im Prozessor-Cache
- In der Größe begrenzt, z.B. 8MB bei aktuellen Linux-Systemen
- Für flexiblere Speicherung brauchen wir anders organisierten Speicher...

Heap: Keine klare Struktur

- Anlegen: in C++ & Java mit new
- Um angelegten Speicher anzusprechen: Zeiger und Referenzen
  - In Java automatisch Zeiger
  - In C++ Zeiger durch * hinter Typ

```
int main() {
  int* i = new int[3];
  int* j = new int;
  delete [] i;
  delete j;
  return 0;
}
```

- Löschen von Heap-Speicher:
  - Java automatisch
  - In C++ nur manuell
    - durch genau einen Aufruf von delete
    - Programmierer ist dafür verantwortlich, dass danach kein Zeiger auf diesen Speicher mehr benutzt wird
- Warum der Unterschied?
  - Nicht einfach festzustellen, wann letzter Zeiger auf Objekt gelöscht wurde
    - Zeiger können selbst auch im Heap gespeichert sein
    - Zyklische Referenzen!
  - Relativ aufwändiges Scannen, in Java durch regelmäßige Garbage Collection gelöst
    - Führt zu Jitter (Schwankung der Zeitdauer, die bestimmte Programmabschnitte zur Bearbeitung benötigen) & Speicher-Overhead, ...

Beispiele

- Anlegen eines Objects auf dem Heap:

```
    std::string* s = new std::string("wiz!");
    delete s;

```

- Allokation von Feldern:

```
    int* i = new int[29]; // gültige Indicies 0-28
    i[0] = 23;
    delete [] i; // nicht mit delete i; verwechseln!

```

- Zeiger können durch & auf beliebige Variablen ermittelt werden

```
    int i = 0;
    int* j = \&i; // \&-Operator erzeugt Zeiger; j darf nicht gelöscht werden

```

- Zeiger können durch * dereferenziert werden

```
    int i = 0;
    int* j = \&i; // \&-Operator erzeugt Zeiger
    *j = 1; // Zugriff auf Variableninhalt

```

- Zugriff auf Methoden/Member Variablen

```
    std::string* s = new std::string("wiz");
    (*s).push_back('?'); // manuelles Derefenzieren
    s$\rightarrow$push_back('?'); // $\rightarrow$ Operator
    delete s;

```

- C++ übergibt alles als Kopie

```
    void set(std::string s) { s = "foo"; }
    int main() {
    std::string s = "bar";
    set(s);
    std::cout << s; // gibt bar aus
    return 0;
    }

```

- Zeiger können verwendet werden, um schreibend zuzugreifen

```
    void set(std::string* s) { *s = "foo"; }
    int main() {
    std::string s = "bar";
    set(\&s);
    std::cout << s; // gibt foo aus
    return 0;
    }

```

- Zeiger erlauben syntaktisch sehr viele Dinge mit unvorhersehbaren Nebenwirkungen

```
    std::string* magicStr() {
    std::string s("wiz!");
    return \&s; // gibt Speicher auf Stack weiter; Tun Sie das nie!
    }
    int main() {
    std::string* s = magicStr();
    std::cout << *s; // Stack ist bereits überschrieben!
    return 0;
    }

```

Warum wirken sich Speicherfehler so unvorhersehbar aus?

- Speicherfehler entstehen sehr häufig durch Zugriff auf Speicherbereiche nachdem diese freigegeben worden sind
- Ob hierdurch später ein Fehler auftritt, hängt davon ab wie der freigegebene Speicher nach der Freigabe wieder genutzt wird
- Die insgesamte Speichernutzung wird durch die Gesamtheit aller Speicherallokationen und -freigaben beeinflusst
- Das kann dazu führen, dass ein Speicherfehler in Modul X erst lange nach seinem Entstehen Auswirkungen zeigt, nachdem in einem anderen Modul Y eine Änderung eingeführt wurde
- Auch eingebundene dynamische Bibliotheken haben Einfluss
- Das macht es so schwierig, solche Fehler schwierig zu finden

Bessere Alternative: Referenzen

- Zeigen ebenfalls auf Speicher, Compiler stellt aber sicher, dass Speicher gültig ist (wenn man nicht in Zeiger wandelt etc.)!
- Markiert durch Suffix &
- Beispiel:

```
void set(std::string\& s) { s = "foo"; }
  int main() {
  std::string s = "bar";
  set(s);
  std::cout << s; // gibt foo aus
  return 0;
}
```

- Dereferenzierung durch * und → nicht notwendig
- Referenzen sind toll, haben aber eine Einschränkung:

```
std::string& magicStr() {
  std::string s("wiz!");
  return s; //< FEHLER
}
```

```
std::string& magicStr() {
  static std::string s("wiz!");
  return s; // klappt prima
}
```

- Per Referenz übergebene Rückgabewerte müssen im Speicher noch existieren, wenn Methodenaufruf abgeschlossen ist...
  - OK für globale Variablen, Member-Variablen, statische Variablen...
  - Nicht-OK für Speicher der wirklich dynamisch alloziert werden muss
- Allgemein bleiben nur Zeiger und Heap:

```
std::string* magicStr() {
  std::string* s = new std::string("wiz!");
  return s; // klappt prima, aber: aufpassen wann s gelöscht
  // werden kann und vollständig vergessen wurde!
}
```

- Konvertierung von Zeigern zu Referenzen mit ∗-Operator:

```
std::string\& s = *magicStr(); // Konvertieren in Referenz; Delete nicht mehr möglich
std::string s2 = *magicStr(); // Konvertieren in Referenz \& Kopie! Delete nicht direkt möglich
```

- Konvertierung von Referenzen zu Zeigern mit &-Operator:

```
   std::string s("bla");
   std::string* sStar = \&s; // Konvertieren in Zeiger
```

- Abschließende Bemerkungen zum Speicher
  - Niemals Speicher doppelt löschen - Niemals Löschen vergessen!
  - Häufige Praxis: Zeiger auf NULL setzen nach dem Löschen (Aber: gibt es danach wirklich keinen anderen Zeiger mehr?)
  - Nur Speicher löschen, der mit "newällokiert wurde
  - Speicher der mit "newällokiert wurde in jedem möglichen Programmablauf löschen (selbst wenn Exceptions auftreten)...
  - Nie über Feldgrenzen hinweg lesen/schreiben (auch negative Indizes!)
  - Programme ausgiebig testen (dabei Address Sanitizer aktivieren!)
  - Statische Code Analyse nutzen: z.B. http://cppcheck.sourceforge.net
  - malloc/free sind Äquivalente in Sprache C und nicht typsicher!
- Verbreitetes Vorgehen in C++ (Pattern): Resource Acquisition Is Initialization (RAII)
  - Speicher (oder Ressourcen im Allgemeinen) wird nur im Konstruktor einer Klasse reserviert
  - Destruktor gibt Speicher frei
  - Sicheres (Exceptions!), nachvollziehbares Konstrukt
  - Beispiel: (Funktioniert leider noch nicht immer)

```
   class MagicString {
     std::string* s;
     public:
       MagicString() : s(new std::string("wiz!")) {}
       std::string* magicStr() { return s; }
       ~MagicString() { delete s; }
   };
```

Vererbung

- Vermeiden von Mehrfachimplementierungen
- Vermeiden von Dopplung interner Daten
- Vererbung syntaktisch ebenfalls ähnlich zu Java

```
class Foo {
  public:
    int magic() const { return 23; }
    int enchanting() const { return 0xbeef; }
};
class FooBar : public Foo {
  public:
    int magic() const { return 42; }
};
```

- Unterschied zu Java: Methoden "liegen"bei C++ statisch im Speicher
  - D.h. *f.magic*(); ruft statisch magic-Methode in Klasse Foo auf, weil f eine Referenz vom Typ Foo ist
  - Vermeidet Mehrfachimplementierungen, realisiert aber keine einheitliche Schnittstelle!

- Nach Überschreiben einer Methode wollen wir meist, dass genutzte Methode nicht vom Referenztyp abhängt, sondern vom Objekttyp
  - Idee zu jedem Objekt speichern wir Zeiger auf zu nutzende Methoden
  - Tabelle wird *vtable* bezeichnet
  - Markierung von Methoden, für die ein Zeiger vorgehalten wird, mit Schlüsselwort virtual
  - Funktionierendes Beispiel:

```
class Foo {
  public:
    virtual int magic() const { return 23; }
};
class FooBar : public Foo {
  public:
    int magic() const override { return 42; }
};
int r(const Foo\& f) { return f.magic(); }
int main() {
  return r(FooBar()); // yeah gibt 42 zurück!
}
```

  - virtual-Markierung genügt in Oberklasse, alle abgeleiteten Methoden ebenfalls "virtuell"
  - override-Markierung optional, aber hätte vor fehlendem virtual gewarnt!

Definierte Programmierschnittstellen durch Überschreiben von Methoden/abstrakte Methoden; Vermeiden von Dopplung interner Daten. Unterschied zu Java: Methoden "liegen"bei C++ statisch im Speicher

```
class Stromfresser {
  public:
    Stromfresser() {
        std::cerr << "Mjam" << std::endl;
    }
};
class Roboter : virtual public Stromfresser {};
class Staubsauger : virtual public Stromfresser {};
class Roomba : public Staubsauger, public Roboter {};

int main() {
    Roomba q;
    return 0;
}
```

Mit virtual: "Mjam", ohne virtual: "Mjam Mjam"

Mehrfachvererbung

- C++ unterstützt keine Interfaces
- Aber C++ unterstützt Mehrfachvererbung! Pro Interface eine Basisklasse → mit abstrakten Methoden erstellen
- Gute Praxis: Explizites Überschreiben

```
  class NiceFooBar : public Foo, public Bar {
    // erlaube NiceFooBar().magic()
    int magic() const override { return Bar::magic(); }
  };

```

  - Wegen Mehrfachvererbung: kein super::
  - Stattdessen immer NameDerBasisKlasse::
- Aber: Diamond Problem
  - Markieren der Ableitung als virtual behebt das Problem

Komposition statt Vererbung

- Vererbungshierarchien werden trotzdem häufig als zu unflexibel angesehen
- Ein möglicher Ausweg:
  - Klassen flexiblen aus anderen Objekten zusammensetzen
  - Einzelobjekte modellieren Aspekte des Verhaltens des Gesamtobjekts
  - Werden beim Anlegen des Gesamtobjekts übergeben
- Engl.: Prefer composition over inheritance

```
class Automatisierungsmodul {
  public:
    void steuere() = 0;
  };
}
class Roboter : public Automatisierungsmodul{
  public:
    void steuere() { /* call HAL */ }
  };
}
class DumbDevice : public Automatisierungsmodul {
  public:
    void steuere() { /* do nothing */ }
  };
}
class Geraet {
  protected:
    Automatisierungsmodul* _a;
    Geraet(Automatisierungsmodul* a, Saeuberungsmodul* s): _a(a), _s(s) {}
  public:
    void steuere() { _a$\rightarrow$steuere(); }
  };
}
```

Operator-Overloading

- In Java: Unterschied zwischen == und *equals*() bei String-Vergleich
- In C++: ==-Operator für String-Vergleich
- Umsetzung: Hinzufügen einer Methode mit Namen *operatorx* wobei für x unter anderem zulässig:
  $+ - */\%^{\&} | \; ! = <> + = - = * = / = \% = ^{=} \; \& = | = << >> >> = << === ! = <= >= <=> \; \&\& || + + - -, - > *- > ()[]$
- Vereinfacht Nutzung komplexer Datentypen teilweise sehr stark
- Aber: Erfordert Disziplin beim Schreiben von Code
  - Oft erwartet: Freiheit von Exceptions (Wer gibt Speicher frei, wenn eine Zuweisung fehlgeschlagen ist?)
  - Semantik der Operatoren muss selbsterklärend sein
    - Ist der Operator auf einem multiplikativen Ring + oder * ?
    - Was ist, wenn zwei ungleiche Objekte jeweils kleiner als das andere sind?
    - Ist * bei Vektoren das Skalar- oder das Kreuzprodukt (oder etwas ganz anderes)?

```
class MagicString {
    std::string* s;
public:
    MagicString() : s(new std::string("wiz!")) {}
    MagicString(const MagicString\& m) : s(new std::string(*m.s)) {}
    std::string* magicStr() { return s; }
    // Neu: = operator erlaubt Zuweisungen
    MagicString\& operator=(const MagicString\& other) {
        if(this != \&other) {
            // ACHTUNG beide Werte werden dereferenziert...
            // ruft operator= in std::string auf $\rightarrow$ String wird kopiert
            *s = *other.s;
        }
        return *this;
    }
    ~MagicString() { delete s; }
};
```

Templates

- Generische Datentypen werden in C++ mit Templates realsiert
- Häufig ähnlich eingesetzt wie Generics, aber können neben Typen auch Konstanten enthalten
- Zur Compile-Zeit aufgelöst → Deklaration & Implementierung in Header-Dateien
- Einfaches Beispiel (mit Typen, ähnl. zu Generics, primitive Typen ok!):

```
template<typename T> // typename keyword $\rightarrow$ deklariert T als Typ
T max(T a, T b) {
  return (a > b ? a : b);
}
```

```
int i = 10;
int j = 2;
int k = max<int>(j, i); // explizit
int l = max(j, i); // automat. Typinferenz durch Parametertypen
```

- Ein wichtiges Grundkonzept von Templates: Substitution failure is not an error (SFINAE) es → wird solange nach passenden Templates (in lexikogr. Reihenfolge) gesucht bis Parameter passen (sonst Fehler!)
- Sehr häufig verwendetes Konstrukt & mächtiger als es scheint, aber schwer zu beherrschen
  - Wir können alternativ versuchen, durch SFINAE zu verhindern, dass Funktionen doppelt definiert sind
  - Trick: Einführen eines Pseudoparameters, der nicht benutzt wird

```
    template<typename T>
    T quadrieren(T i, typename T::Val pseudoParam = 0) {
      T b(i); b *= i; return b;
    }

```

  - Trick: Einführen eines Hilfstemplates (sogenannter trait): wenn arithmetic¡T¿::Cond definiert ist, muss T = int sein

```
    template<typename T> struct arithmetic {};
    template<> struct arithmetic<int> { using Cond = void*; };

```

  - Definition einer Funktion, die nur für int instanziiert werden kann:

```
    template<typename T>
    T quadrieren(T i, typename arithmetic<T>::Cond = nullptr) {
      return i * i;
    }

```

Container

- Templates werden an vielen Stellen der C++ Standard-Bibliothek verwendet
- Container implementieren alle gängigen Datenstrukturen
- Prominente Beispiele:

```
  template<typename T> class vector; // dynamisches Array
  template<typename T> class list; // doppelt verkette Liste
  template<typename T> class set; // geordnete Menge basiert auf Baum
  template<typename K, typename V> class map; // Assoziatives Array,

```

- Alle Templates sind stark vereinfacht dargestellt, weitere Parameter haben Standardwerte, die z.B. Speicherverhalten regeln

Container Enumerieren

- Je nach Struktur unterschiedlicher Zugriff
- Oft über Iteratoren vom Typ Container::iterator, bspw. vector¡int¿::iterator

```
  std::vector<int> v{ 1, 2, 3 }; // Initialisierung über Liste
  // normale for-Schleife, Beachte: Überladene Operatoren ++ und *
  for(std::vector<int>::iterator i = v.begin(); i != v.end(); ++i) {
    std::cout << *i << std::endl;
  }
  // auto erlaubt Typinferenz $\rightarrow$ Code lesbarer, aber fehleranfälliger
  for(auto i = v.begin(); i != v.end(); ++i) {
    std::cout << *i << std::endl;
  }
  // range loop (nutzt intern Iteratoren), komplexe Datentypen nur mit Ref. & sonst werden Kopie erzeugt!
  for(int i : v) { // hier ohne &, da nur int in v gespeichert
    std::cout << i << std::endl;
  }

```

Container Einfügen

- Unterschiedliche Operationen je nach Container-Typ
- $std :: vector < T >:: push\_back()$ fügt neues Element am Ende ein
  - Allokiert ggf. neuen Speicher
  - Exisitierende Pointer können dadurch invalidiert werden!!!
- $std :: list < T >$ zusätzlich $push\_front()$ fügt Element am Anfang ein
- $std :: set$, $std :: map$, ...
  - $insert()$ fügt Element ein, falls es nicht existiert (Optional mit Hinweis wo ungefähr eingefügt werden soll)
  - $operator[]$ erlaubt Zugriff aber auch Überschreiben alter Elemente
  - $emplace()$ Einfügen, ohne Kopien zu erzeugen (nicht behandelt)

Container Löschen

- Unterschiedliche Operationen je nach Container-Typ
- Allgemein: erase(Container::iterator) (Vorsicht ggf. werden Iterator/Zeiger auf Objekte dadurch ungültig!)
- $std :: vector < T >:: resize()$ löscht implizit letzte Elemente bei Verkleinerung
- $std :: vector < T >:: pop\_back()$ entfernt letztes Element
- $std :: list < T >$ hat zusätzlich $pop\_front()$
- $std :: set$, $std :: map$, ... löschen nur mit $erase()$

Shared Pointer

- Synonym: Smart Pointer
- Ziel: Sichereres Verwenden von Speicher
- Idee: kleine, schlanke Zeiger-Objekte, die Referenzzähler + Zeiger auf komplexere Objekte enthalten, wird letztes Zeiger-Objekt gelöscht, wird auch das komplexe Objekt gelöscht
- Realisierung mit RAII, Templates, Operator-Überladung
- Beispiel, wie shared_ptr sich verhalten sollten

```
using stringP = shared_ptr<std::string>;
stringP hello() { // gibt kopie der referenz zurück
  return stringP(new std::string("Hello!"));
}

int main() {
  stringP x = hello();
  stringP y(x); // Erstellen einer weiteren Referenz
  std::cout << y$\rightarrow$length();
  return 0; // Original-String wird gelöscht wenn letzte Ref. weg
}

template<class T> class shared_ptr { // Vereinfacht!
  T* p; // Zeiger auf eigentliches Objekt
  int* r; // Referenzzähler
public:
  // neue Referenz auf Objekt erzeugen
  shared_ptr(T* t) : p(t), r(new int) { *r = 1; }
  // Referenz durch andere Referenz erzeugen
  shared_ptr(const shared_ptr<T>\& sp) : p(sp.p), r(sp.r) { ++(*r); }
  T* operator$\rightarrow$() const { // benutzen wie einen richtigen Zeiger
    return p;
  }
  ~shared_ptr() {
    if(--(*r) == 0) { // Objekt loeschen, wenn letzte Referenz weg
      delete r;
      delete p;
    }
  }
};
```

```
template<class T> class shared_ptr { // Vereinfacht!
    T* p; // Zeiger auf eigentliches Objekt
    int*r; // Referenzzähler
public:
    // neue Referenz auf Objekt erzeugen
    shared_ptr(T* t) : p(t), r(new int) { *r = 1; }
    // Referenz durch andere Referenz erzeugen
    shared_ptr(const shared_ptr<T>\& sp) : p(sp.p), r(sp.r) { ++(*r); }
    T* operator$\rightarrow$() const { // benutzen wie einen richtigen Zeiger
        return p;
    }
    ~shared_ptr() {
        if(--(*r) == 0) { // Objekt loeschen, wenn letzte Referenz weg
            delete r;
```

```
            delete p;
}}};
```

Vergleich mit Java

- Unterschiede im Aufbau:
  - C++ hat globale Funktionen, also außerhalb von Klassen, wie main
  - #include gibt Dateien mit Klassen- und Funktionsdefinitionen an, die der Compiler einlesen soll
  - Java-Programme werden in packages gegliedert, in C++ gibt es mit modules ein ähnliches Konzept, welches aber (noch) nicht verbreitet ist
  - C++-Programme können (ohne Bezug zu Dateien) in namespaces untergliedert werden, hier std
- Programmargumente:
  - In Java bekommt main ein String-Array übergeben, die Länge kann über .length abgefragt werden
  - C/C++-Programme erhalten ein Array von char* (Details zu Pointern folgen)
  - In C/C++ sind Arrays keine Pseudoobjekte, sondern Speicherbereiche in denen die Daten konsekutiv abgelegt sind → argc wird benötigt die Anzahl an Elementen zu kodieren
- Rückgabewerte:
  - In Java keine Rückgabe in der main-Methode
  - In C++ Rückgabe eines exit code
    - 0 gibt an: Programmausführung erfolgreich
    - Andere Werte geben einen Programm-spezifischen Fehlercode zurück
- Primitive Datentypen:
  - Wie in Java einfache Datentypen, die SZahlenënthalten
  - char, short, int, long sind auf 64-bit Maschinen 8 bit, 16 bit, 32 bit und 64 bit breit (char braucht in Java 16 Bit!)
  - long ist auf 32 bit Maschinen 32 Bit breit, long long [sic!] ist immer 64 Bit
  - bool speichert Boolsche Werte (Breite hängt vom Compiler ab!)
  - Ein unsigned vor Ganzahltypen gibt an, dass keine negativen Zahlen in der Variable gespeichert werden (Beispiel: unsigned int) → Kann größere Zahlen speichern & zu viel Unsinn führen (beim Vergleich mit vorzeichenbehafteten Zahlen)

```
[Hello.java]
package hello; // say that we are part of a package
  public class Hello { // declare a class called Hello:
  // declare the function main that takes an array of Strings:
  public static void main(String args[]) {
    // call the static method, println on class System.out with parameter "Hi Welt!":
    System.out.println("Hi Welt!");
  }
} // end of class Hello
```

```
[Hello.cpp]
// include declarations for I/O library where cout object is specified in namespace std::
#include <iostream>
// declare the function main that takes an int and array of strings and returns an int as the exit code
int main(int argc, char* argv[]) {
  // stream string to cout object flush line with endl
    std::cout << "Hello world!" << std::endl;
    return 0;
} // end of main()
```

Zusammenfassung

- C++ erlaubt sehr detaillierte Kontrolle über Speicher- und Laufzeitverhalten
- Es ist relativ einfach, schwierig zu findende Fehler einzubauen
- Die Sprache ist durch Operator-Overloading, Mehrfachvererbung und Templates sehr mächtig
- Erlaubt hohen Grad an Wiederverwendung
- Anzahl an Code-Zeilen kann reduziert werden
- Code kann völlig unlesbar werden! Viele Features sollten nur eingesetzt werden wenn sich dadurch ein wirklicher Vorteil ergibt!

Objektorientierung am Beispiel Java

- Bekannt
  - Grundlegendes Verständnis von Java
  - Kapselung durch Klassen und Vererbung
- Ziele
  - Verständnis der Probleme bei Vererbung und Typersetzbarkeit in objektorientierten Programmiersprachen
  - Kennenlernen der Grundideen generischer und abstrahierender Konzepte in Objekt-orientierter Programmierung (OOP)
  - Praktische Erfahrungen anhand von Java & C++
- Ausdrucksstärke erhöhen, Komplexität verbergen

Unit-Testing in Java

- De facto Standard: JUnit Framework
- Best Practice für einfachen Einsatz:
  - Java Code in ein oder mehrere Klassen im Ordner src speichern
  - Im Ordner tests jeweils eine Klasse anlegen, die Funktionen einer Implementierungsklasse prüft
  - Konvention: Testklasse einer Klasse Name heißt NameTest
  - Eigentliche Tests werden in Methoden implementiert, die als Tests annotiert sind
  - Typischer Ansatz: für bekannte Werte ausführen und Ergebnis mit Grundwahrheit (erwartetes Verhalten) vergleichen, bspw. mit assertEquals-Funktion
- Viele weitere Features, z.B. Deaktivieren von Tests, Timeouts, GUI Coverage, Mocks

Einführung in Funktionale Programmierung

Sind $v_1, ..., v_n$ Unbestimmte vom Typ $T_1, ..., T_n$ (bool oder int) und ist $t(v_1, ..., v_n)$ ein Term, so heißt $f(v_1, ..., v_n) = t(v_1, ..., v_n)$ eine Funktionsdefinition vom Typ T. T ist dabei der Typ des Terms.
Ein **applikativer Algorithmus** ist eine Menge von Funktionsdefinitionen
$f_1(v_{1,1}, ..., v_{1,n_1}) = t_1(v_{1,1}, ..., v_{1,n_1}, ..., f_m(v_{m,1}, ..., v_{m,n_m}) = t_m(v_{m,1}, ..., v_{m,n_m}))$. Die erste Funktion f_1 wird wie beschrieben ausgewertet und ist die Bedeutung (Semantik) des Algorithmus.
Kategorien der funktionalen Sprachen

- Ordnung der Sprache
  - Erster Ordnung: Funktionen können (nur) definiert und aufgerufen werden
  - Höherer Ordnung:
    - Funktionen können außerdem als Parameter an Funktionen übergeben werden und/oder Ergebnisse von Funktionen sein.
    - Funktionen sind hier auch Werte! – erstklassige Werte;
    - Erstklassig: Es gibt keine Einschränkungen.
    - Umgekehrt: Wert ist eine Funktion ohne Parameter
- Auswertungsstrategie:
  - Strikte Auswertung:
    - Synonyme: strict evaluation, eager evaluation, call by value, applikative Reduktion
    - Die Argumente einer Funktion werden vor Eintritt in die Funktion berechnet (ausgewertet) - wie z.B. in Pascal oder C.
  - Bedarfsauswertung:
    - Synonyme: Lazy evaluation, call by need
    - Funktionsargumente werden unausgewertet an die Funktion übergeben
    - Erst wenn die Funktion (in ihrem Körper) die Argumente benötigt, werden die eingesetzten Argumentausdrücke berechnet, und dann nur einmal.
    - Realisiert SSharing"(im Unterschied zur Normalform-Reduktion - dort werden gleiche Ausdrücke immer wieder erneut berechnet).
- Typisierung:
  - Stark typisiert: Die verbreiteten funktionalen Programmiersprachen sind stark typisiert, d.h. alle Typfehler werden erkannt.

    Statisch typisiert Typprüfung wird zur Übersetzungszeit ausgeführt.
    Dynamisch typisiert Typprüfung wird zur Laufzeit ausgeführt
  - Untypisiert: Reiner Lambda-Kalkül (später)

Applikaive Algorithmen

Grundidee

- Definition zusammengesetzter Funktionen durch Terme: $f(x) = 5x + 1$
- Unbestimmte:
  - $x, y, z, ...$ vom Typ int
  - $q, p, r, ...$ vom Typ bool
- Terme mit Unbestimmten (z.B. Terme vom Typ int: $x, x - 2, 2x + 1, (x + 1)(y - 1)$)
- Terme vom Typ bool $p, p \vee true, (p \vee true) \Rightarrow (q \vee false)$

Sind $v_1, ..., v_n$ Unbestimmte vom Typ $\tau_1, ..., \tau_n$ (bool oder int) und ist $t(v_1, ..., v_n)$ ein Term, so heißt $f(v_1, ..., v_n) = t(v_1, ..., v_n)$ eine Funktionsdefinition vom Typ $\tau.\tau$ ist dabei der Typ des Terms.

- Erweiterung der Definition von Termen
- Neu: Aufrufe definierter Funktionen sind Terme

Ein applikativer Algorithmus ist eine Menge von Funktionsdefinitionen. Die erste Funktion f_1 wird wie beschrieben ausgewertet und ist die Bedeutung (Semantik) des Algorithmus.

Die funktionale Programmiersprache Erlang

- Entwickelt ab der zweiten Hälfte der 1980er Jahre im Ericsson Computer Science Laboratory (CSLab, Schweden)
- Ziel war, eine einfache, effiziente und nicht zu umfangreiche Sprache, die sich gut zur Programmierung robuster, großer und nebenläufiger Anwendungen für den industriellen Einsatz eignet.
- Erste Version einer Erlang-Umgebung entstand 1987 auf der Grundlage von Prolog. Später wurden Bytecode-Übersetzer und abstrakte Maschinen geschaffen.

Arbeiten mit Erlang

- Erlang-Programme werden durch Definition der entsprechenden Funktionen in Modulen erstellt
- Module können in den Erlang-Interpreter geladen und von diesem in Zwischencode übersetzt werden
- Anschließend können Anfragen im Interpreter gestellt werden

Modul "fakultaet.erl":

```
1 -module(fakultaet).
2 -export([fak/1]).
3 fak(0) $\rightarrow$ 1;
4 fak(N) when N > 0 $\rightarrow$ (N) * fak(N-1).
```

Laden in Interpreter mittels: $c(fakultaet)$
Testen der Funktion, z.B. mit: $fakultaet : fak(5)$

Elemente von Erlang Kurz:

- Kommentare: werden mit % eingeleitet und erstrecken sich bis Zeilenende
- Ganzzahlen (Integer): 10, -234, $A, $Char, 16#AB10F
- Gleitkommazahlen (Floats): 17.368, -56.34, 12.34E-10
- Atoms (Atoms): abcdf, start_with_lower_case, 'Blanks are quoted' #Konstanten mit eigenem Namen als Wert
- Tupel (Tupels): 123, bcd, abc, def, 'joe', 234, als #können feste Anzahl von Dingen speichern
- Listen (Lists): [123, xzy], [person, 'Joe', 'Armstrong,person...] #können variable Anzahl von Dingen speichern
- Variablen: Abc, A_long_variable #fangen mit Großbuchstaben an; zur Speicherung von Werten; können nur einmal gebunden werden
- Komplexe Datenstrukturen: beliebige komplexe Datenstrukturen können erzeugt werden (durch einfaches Hinschreiben)
- Case-Ausdrücke:

```
1   case Expression of Pattern1 [when Guard1] -> Expr_seq1; ... end
2
```

Ganzzahlen (Integer):

- 10
- -234
- 16#AB10F
- 2#110111010
- $A
- B#Val erlaubt Zahlendarstellung mit Basis B (mit $B \leq 36$).
- $Char ermöglicht Angabe von Ascii-Zeichen ($A für 65).

Gleitkommazahlen (Floats):

- 17.368
- -56.654
- 12.34E-10.

Atome (Atoms):

- abcef
- start_with_a_lower_case_letter
- "Blanks can be quoted"
- Änything inside quotes"
- Erläuterungen:
  - Atome sind Konstanten, die Ihren eigenen Namen als Wert haben
  - Atome beliebiger Länge sind zulässig
  - Jedes Zeichen ist innerhalb eines Atoms erlaubt
  - Einige Atome sind reservierte Schlüsselwörter und können nur in der von den Sprachentwicklern gewünschen Weise verwendet werden als Funktionsbezeichner, Operatoren, Ausdrücke etc.
  - Reserviert sind: *after and andalso band begin bnot bor bsl bsr bxor case catch cond div end fun if let not of or orelse query receive rem try when xor*

Tupel (Tuples):

- {123, bcd} % Ein Tupel aus Ganzzahl und Atom
- {123, def, abc}
- {person, 'Joe', 'Armstrong'}
- {abc, def, 123, jkl}
- {}
- Erläuterungen:
  - Können eine feste Anzahl von "Dingenßpeichern
  - Tupel beliebiger Größe sind zulässig
  - Kommentare werden in Erlang mit % eingeleitet und erstrecken sich dann bis zum Zeilenende

Listen:

- $[123, xyz]$
- $[123, def, abc]$
- $[\{person,' Joe',' Armstrong'\}, \{person,' Robert',' Virding'\}, \{person,' Mike',' Williams'\}]$
- äbcdefghẅird zu $[97, 98, 99, 100, 101, 102, 103, 104]$
- wird zu []
- Erläuterungen:
  - Listen können eine variable Anzahl von Dingen speichern
  - Die Größe von Listen wird dynamisch bestimmt
  - "...ïst eine Kurzform für die Liste der Ganzzahlen, die die ASCIICodes der Zeichen innerhalb der Anführungszeichen repräsentieren

Variablen:

- Abc
- A_long_variable_name
- AnObjectOrientatedVariableName
- Erläuterungen:
  - Fangen grundsätzlich mit einem Großbuchstaben an
  - Keine "Funny Characters"
  - Variablen werden zu Speicherung von Werten von Datenstrukturen verwendet
  - Variablen können nur einmal gebunden werden!
  - Der Wert einer Variablen kann also nicht mehr verändert werden, nachdem er einmal gesetzt wurde: $N = N + 1$ VERBOTEN!
  - Einzige Ausnahmen: Die anonyme Variable "_"(kein Lesen möglich) und das Löschen einer Variable im Interpreter mit $f(N)$.

Komplexe Datenstrukturen:

- $[\{\{person,' Joe',' Armstrong'\}, \{telephoneNumber, [3, 5, 9, 7]\}, \{shoeSize, 42\}, \{pets, [\{cat, tubby\}, \{cat, tiger\}]\}, \{children, [\{thomas, 5\}, \{claire, 1\}]\}\},$
- Erläuterungen:
  - Beliebig komplexe Strukturen können erzeugt werden
  - Datenstrukturen können durch einfaches Hinschreiben erzeugt werden (keine explizite Speicherbelegung oder -freigabe)
  - Datenstrukturen können gebundene Variablen enthalten

Pattern Matching:

- $A = 10$ erfolgreich, bindet A zu 10
- $B, C, D = 10, foo, bar$ erfolgreich, bindet B zu 10, C zu foo and D zu bar
- $A, A, B = abc, abc, foo$ erfolgreich, bindet A zu abc, B zu foo
- $A, A, B = abc, def, 123$ schlägt fehl ("fails")
- $[A, B, C] = [1, 2, 3]$ erfolgreich, bindet A zu 1, B zu 2, C zu 3
- $[A, B, C, D] = [1, 2, 3]$ schlägt fehl
- $[A, B|C] = [1, 2, 3, 4, 5, 6, 7]$ erfolgreich bindet A zu 1, B zu 2, C zu [3,4,5,6,7]
- $[H|T] = [1, 2, 3, 4]$ erfolgreich, bindet H zu 1, T zu [2,3,4]
- $[H|T] = [abc]$ erfolgreich, bindet H zu abc, T zu []
- $[H|T] = []$ schlägt fehl
- $A, , [B|_], B = abc, 23, [22, x], 22$ erfolgreich, bindet A zu abc, B zu 22
- Erläuterungen:
  - "Pattern Matching", zu Deutsch "Mustervergleich", spielt eine zentrale Rolle bei der Auswahl der "richtigenÄnweisungsfolge für einen konkreten Funktionsaufruf und dem Binden der Variablen für die Funktionsparameter (siehe spätere Erklärungen)
  - Beachte die Verwendung von "_", der anonymen ("don't care") Variable (diese Variable kann beliebig oft gebunden, jedoch nie ausgelesen werden, da ihr Inhalt keine Rolle spielt).
  - Im letzten Beispiel wird die Variable B nur einmal an den Wert 22 gebunden (das klappt, da der letzte Wert genau 22 ist)

Funktionsaufrufe:

- module:func(Arg1, Arg2, ... Argn)
- func(Arg1, Arg2, .. Argn)
- Erläuterungen:
  - Arg1 .. Argn sind beliebige Erlang-Datenstrukturen
  - Die Funktion und die Modulnamen müssen Atome sein (im obigen Beispiel module und func)
  - Eine Funktion darf auch ohne Parameter (Argumente) sein (z.B. date() - gibt das aktuelle Datum zurück)
  - Funktionen werden innerhalb von Modulen definiert
  - Funktionen müssen exportiert werden, bevor sie außerhalb des Moduls, in dem sie definiert sind, verwendet werden
  - Innerhalb ihres Moduls können Funktionen ohne den vorangestellten Modulnamen aufgerufen werden (sonst nur nach einer vorherigen Import-Anweisung)

Modul-Deklaration:

```
-module(demo).
-export([double/1]).
double(X) $\rightarrow$ times(X, 2).
times(X, N) $\rightarrow$ X * N.
```

- Erläuterungen:
- Die Funktion double kann auch außerhalb des Moduls verwendet werden, times ist nur lokal in dem Modul verwendbar
- Die Bezeichnung double/1 deklariert die Funktion double mit einem Argument
- Beachte: double/1 und double/2 bezeichnen zwei unterschiedliche Funktionen

Eingebaute Funktionen (Built In Functions, BIFs)

- date()
- time()
- length([1,2,3,4,5])
- size(a,b,c)
- atom_to_list(an_atom)
- list_to_tuple([1,2,3,4])
- integer_to_list(2234)
- tuple_to_list()
- Erläuterungen:
  - Eingebaute Funktionen sind im Modul erlang deklariert
  - Für Aufgaben, die mit normalen Funktionen nicht oder nur sehr schwierig in Erlang realisiert werden können
  - Verändern das Verhalten des Systems
  - Beschrieben im Erlang BIFs Handbuch

Definition von Funktionen:

```
func(Pattern1, Pattern2, ...) $\rightarrow$
  ... ; % Vor dem ; steht der Rumpf
func(Pattern1, Pattern2, ...) $\rightarrow$
  ... ; % Das ; kündigt weitere Alternativen an
  ... % Beliebig viele Alternativen möglich
func(Pattern1, Pattern2, ...) $\rightarrow$
  ... . % Am Ende muss ein Punkt stehen!
```

Erläuterungen:

- Funktionen werden als Sequenz von Klauseln definiert
- Sequentielles Testen der Klauseln bis das erste Muster erkannt wird (Pattern Matching)
- Das Pattern Matching bindet alle Variablen im Kopf der Klausel
- Variablen sind lokal zu jeder Klausel (automatische Speicherverw.)
- Der entsprechende Anweisungsrumpf wird sequentiell ausgeführt

Was passiert wenn wir $mathstuff : factorial()$ mit einem negativen Argument aufrufen? Der Interpreter reagiert nicht mehr?

- Erste Reaktion: rette das Laufzeitsystem durch Eingabe von CTRL-G
  - User switch command
    1. -→ h
    2. c [nn] - connect to job
    3. i [nn] - interrupt job
    4. k [nn] - kill job
    5. j - list all jobs
    6. s [shell] - start local shell
    7. r [node [shell]] - start remote shell
    8. q - quit erlang
    9. ? — h - this message
    10. -→
  - Liste durch Eingabe von j alle Jobnummern auf
  - Beende den entsprechenden Shell-Job durch k ¡jobnr¿
  - Starte eine neue Shell durch Eingabe von s
  - Liste durch erneute Eingabe von j die neuen Jobnummern auf
  - Verbinde durch Eingabe von c ¡shelljobnr¿ mit neuer Shell
- Zweite Reaktion: Ergänze factorial() um zusätzliche Bedingung:
  - "Beschütze"die Funktion vor Endlosrekursion durch Ergänzung eines sogenannten Wächters (Guards) bei dem entsprechenden Fallmuster (Pattern)
  - Erläuterungen:
    - Der Guard wird durch das Atom when und eine Bedingung vor dem Pfeil → formuliert
    - Vollständig "beschützte"Klauseln können in beliebiger Reihenfolge angeordnet werden
    - Achtung: Ohne Guard führt diese Reihenfolge zu Endlosschleifen
  - Beispiele für Guards:

```
        number(X)        % X is a number
        integer(X)       % X is an integer
        float(X)         % X is a float
        atom(X)          % X is an atom
        tuple(X)         % X is a tuple
        list(X)          % X is a list
        length(X) == 3   % X is a list of length 3
        size(X) == 2     % X is a tuple of size 2.
        X > Y + Z        % X is > Y + Z
        X == Y           % X is equal to Y
        X =:= Y          % X is exactly equal to Y (i.e. 1 == 1.0 succeeds but 1 =:= 1.0 fails)

```

- Alle Variablen in einem Wächter müssen zuvor gebunden werden

Traversieren (Äblaufen") von Listen:

```
average(X) -> sum(X) / len(X).
sum([H|T]) -> H + sum(T); % summiert alle Werte auf
sum([]) -> 0.
len([_|T]) -> 1 + len(T); % Wert des Elements
len([]) -> 0. % interessiert nicht
```

- Die Funktionen sum und len verwenden das gleiche Rekursionsmuster
- Zwei weitere gebräuchliche Rekursionsmuster:

```
  double([H|T]) -> [2*H|double(T)];   % verdoppelt alle
  double([]) -> [].                   % Listenelemente
  member(H, [H|_]) -> true;           % prüft auf
  member(H, [_|T]) -> member(H, T);   % Enthaltensein
  member(_, []) -> false.             % in Liste

```

Listen und Akkumulatoren:

```
  average(X) -> average(X, 0, 0).
  average([H|T], Length, Sum) -> average(T, Length + 1, Sum + H);
  average([], Length, Sum) -> Sum / Length.
```

- Interessant sind an diesem Beispiel:
- Die Liste wird nur einmal traversiert
- Der Speicheraufwand bei der Ausführung ist konstant, da die Funktion ëndrekursivïst (nach Rekursion steht Ergebnis fest)
- Die Variablen Length und Sum spielen die Rolle von Akkumulatoren
- Bemerkung: average([]) ist nicht definiert, da man nicht den Durchschnitt von 0 Werten berechnen kann (führt zu Laufzeitfehler)

Ïdentisch"benannte Funktionen mit unterschiedlicher Parameterzahl:

```
  sum(L) -> sum(L, 0).
  sum([], N) -> N;
  sum([H|T], N) -> sum(T, H+N).
```

- Erläuterungen
- Die Funktion *sum*/1 summiert die Elemente einer als Parameter übergebenen Liste
- Sie verwendet eine Hilfsfunktion, die mit *sum*/2 benannt ist
- Die Hilfsfunktion hätte auch irgendeinen anderen Namen haben können
- Für Erlang sind *sum*/1 und *sum*/2 tatsächlich unterschiedliche Funktionsnamen

Shell-Kommandos:

```
  h()  history . Print the last 20 commands.
  b()  bindings. See all variable bindings.
  f()  forget. Forget all variable bindings.
  f(Var)  forget. Forget the binding of variable X. This can ONLY be used as a command to the shell - NOT in
    the body of a function!
  e(n) evaluate. Evaluate the n:th command in history.
  e(-1) Evaluate the previous command.
```

Erläuterungen: Die Kommandozeile kann wie mit dem Editor Emacs editiert werden (werl.exe unterstützt zusätzlich Historie mit Cursortasten)
Spezielle Funktionen:

```
apply(Func, Args)
apply(Mod, Func, Args) % old style, deprecated
```

- Erläuterungen:
- Wendet die Funktion Func (im Modul Mod bei der zweiten Variante) auf die Argumente an, die in der Liste Args enthalten sind
- Mod und Func müssen Atome sein bzw. Ausdrücke, die zu Atomen evaluiert werden und die eine Funktion bzw. Modul referenzieren
- Jeder Erlang-Ausdruck kann für die Formulierung der an die Funktion zu übergebenden Argumente verwendet werden
- Die Stelligkeit der Funktion ist gleich der Länge der Argumentliste
- Beispiel:

```
  ‘ 1> apply( lists1,min_max,[[4,1,7,3,9,10]]). -> {1, 10}

```

- Bemerkung: Die Funktion *min_max* erhält hier ein Argument

Anonyme Funktionen:

```
Double = fun(X) -> 2*X end.
> Double(4).
> 8
```

- Erläuterung:
- Mittels "fun"können anonyme Funktionen deklariert werden
- Diese können auch einer Variablen (im obigen Beispiel Double) zugewiesen werden
- Interessant wird diese Art der Funktionsdefinition, da anonyme Funktionen auch als Parameter übergeben bzw. als Ergebniswert zurückgegeben werden können
- Die Funktionen, die anonyme Funktionen als Parameter akzeptieren bzw. als Ergebnis zurückgeben nennt man Funktionen höherer Ordnung
- Erlang-Programme werden durch Definition der entsprechenden Funktionen in Modulen erstellt
- Module können in den Erlang-Interpreter geladen und von diesem in Zwischencode übersetzt werden
- Anschließend können Anfragen im Interpreter gestellt werden

Modul fakultaet.erl:

```
-module(fakultaet).
-export([fak/1]).
fak(0) $\rightarrow$ 1;
fak(N) when N > 0 $\rightarrow$ (N) * fak(N-1).
```

Laden in den Interpreter mittels:

```
c(fakultaet).
```

Testen der Funktion, z.B. mit:

```
fakultaet:fak(5).
```

Kalküle

- Minimalistische Programmiersprachen zur Beschreibung von Berechnungen,
- mathematische Objekte, über die Beweise geführt werden können.

In dieser Vorlesung:

- λ-Kalkül (Church, Landin) für sequentielle (funktionale/imperative Sprachen)

Beispiele weiterer Kalküle:

- CSP (Hoare) Communicating Sequential Processes - für nebenläufige Programme mit Nachrichtenaustausch
- π-Kalkül (Milner) für nebenläufige, mobile Programme

Lambda Kalkül

Kalküle sind minimalistische Programmiersprachen zur Beschreibung von Berechnungen, mathematische Objekte, über die Beweise geführt werden können Hier: λ-Kalkül (Church, Landin) für sequentielle (funktionale /imperative Sprachen)
Definition der λ-Terme: Die Klasse $\bigwedge$ der Lambda-Terme ist die kleinste Klasse, welche die folgenden Eigenschaften erfüllt:

- Wenn x eine Variable ist, dann ist $x \in \bigwedge$
- Wenn $M \in \bigwedge$ ist, dann ist $(\lambda xM) \in \bigwedge$ (Abstraktion)
- Wenn $M, N \in \bigwedge$ sind, dann ist $(MN) \in \bigwedge$ (Funktionsanwendung)

Bezeichnung	Notation	Beispiele
Variablen	x	x y
Abstraktion	$\lambda x.t$	$\lambda y.0$ $\lambda f.\lambda x.\lambda y.fyx$
Funktionsanwendung	t_1t_2	$f42$, $(\lambda x.x + 5)7$

λ-Abstraktionen: $\lambda x.t$ bindet die Variable x im Ausdruck t
Die Menge der freien Variablen eines Terms M wird mit $FV(M)$ bezeichnet und ist wie folgt induktiv definiert:

- $FV(x) = x$
- $FV(MN) = FV(M) \cup FV(N)$
- $FV(\lambda x.M) = FV(M) - \{x\}$

Ein Lambda-Term ohne freie Variablen heißt Kombinator

- Identitätsfunktion: $I \equiv \lambda x.x$
- Konstanten-Funktional: $K \equiv \lambda xy.x$
- Fixpunkt-Kombinator: $Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

Das untypisierte Lambdakalkül

- Turing-mächtiges Modell funktionaler Programme
- Auch: Beschreibung sequentieller imperativer Konstrukte

Definition der λ-Terme:
Die Klasse Λ der Lambda-Terme ist die kleinste Klasse, welche die folgenden Eigenschaften erfüllt:

- Wenn x eine Variable ist, dann ist $x \in \Lambda$
- Wenn $M \in \Lambda$ ist, dann ist $(\lambda xM) \in \Lambda$ (Abstraktion)
- Wenn $M, N \in \Lambda$ sind, dann ist $(MN) \in \Lambda$ (Funktionsanwendung)

- Um Klammern zu sparen verwendet man oft eine alternative Notation: λx.M
- Bei mehreren zu bindenden Variablen: $\lambda xyz.M = (\lambda x(\lambda y(\lambda zM)))$

λ-Terme

Bezeichnung	**Notation**	**Beispiele**
Variablen	x	x y
Abstraktion	λx.t	$\lambda y.0$ $\lambda f.\lambda x.\lambda y.fyx$
Funktionsanwendung	t_1t_2	$f42$ $(\lambda x.x + 5)7$

(weitere primitive Operationen nach Bedarf) 17, True, $+$, $\cdot$

	x, y, f	sind korrekte Programmvariablen
Variablenkonvention	x, y, z	sind Meta-Variablen für Programmvariablen
	t, t', t_1, t_2	bezeichnen immer einen Term

Funktionsanwendung ist linksassoziativ und bindet stärker als Abstraktion

$$\lambda x.fxy = \lambda x.((fx)y)$$

Abstraktion ist rechtsassoziativ:

$$\lambda x\lambda y.fxy = (\lambda x.(\lambda .fxy))$$

Strukturelle Induktion

- Aufgrund des "rekursivenÄufbaus der Definition der Klasse Λ der Lamda-Terme, können Aussagen über Lambda-Terme mittels ßtruktureller Induktion"geführt werden:
  - Hierbei folgt der Induktionsbeweis der Struktur der Lambda-Terme, wie er in der Definition vorgegeben wird
- Beispiel: Jeder Term in Λ ist wohl geklammert
  - Induktionsanfang: trivial, da jede Variable ein wohlgeklammerter Lambda-Term ist.
  - Induktionsannahme: M, N sind wohlgeklammerte Lambda-Terme
  - Induktionsschritt: dann sind auch die Terme (MN) und (λxM) wolgeklammert.

Variablenbindung bei Abstraktion

Variablenbindung in Haskell (erlaubt anonyme Lambda-Funktionen):
Anonyme Funktion: $\backslash x- > (\backslash y- > y + 5)(x + 3)$
let-Ausdruck: let $x = 5 in x + y$
Analog bei λ-Abstraktionen: $\lambda x.t$ bindet die Variable x im Ausdruck t
Beispiele:

- $\lambda x.\lambda y.fyx$ bindet x in $\lambda y.fyx$, das selbst x in fyx bindet.
- f ist frei in $\lambda x.\lambda y.fyx$.

Innere Abstraktionen können äußere Variablen verdecken:

$$(\lambda x.\lambda y.\lambda z.f(\lambda x.z + x)(yx))(\lambda y.y + x)$$

g = λ n. **if** isZero n **then** c_1 **else** (times n g (pred n))
G = λ g. λ n. **if** isZero n **then** c_1 **else** (times n g (pred n))
} // Keine Lambda-Terme

G = λ g. λ n. (λ a. a) (isZero n) c_1 (times n (g (sub n c_1)))
Y = λf. (λx. f (x x)) (λx.f (x x))
Fak = Y G

Fak c_2 = Y G c_2 ⇒ ((λx. G (x x)) (λx. G (x x))) c_2
⇒ G ((λx. G (x x)) (λx. G (x x))) c_2
⇒ (isZero c_2) c_1 (times c_2 ((λx. G (x x)) (λx. G (x x)) (pred c_2)))
⇒ times c_2 ((λx. G (x x)) (λx. G (x x)) (pred c_2))
⇒ times c_2 ($\overbrace{(\lambda x.\ G\ (x\ x))\ (\lambda x.\ G\ (x\ x))}^{YG\ \Rightarrow}$) c_1
⇒ times c_2 ((isZero c_1) c_1 (times c_1 ((λx. G (x x)) (λx. G (x x)) (pred c_1))))
⇒ times c_2 (times c_1 ((λx. G (x x)) (λx. G (x x)) (pred c_1)))
⇒ times c_2 (times c_1 ((is_zero c_0) c_1 ...)) ⇒ c_2

Freie und gebundene Variablen

- Die Menge der freien Variablen eines Terms M wird mit FV(M) bezeichnet und ist wie folgt induktiv definiert:
  - $FV(x) = \{x\}$
  - $FV(MN) = FV(M) \cup FV(N)$
  - $FV(\lambda x.M) = FV(M) - \{x\}$
- Übung: Definieren sie analog die Menge der gebundenen Variablen GV(M)
- Ein Lambda-Term ohne freie Variablen heißt Kombinator
- Einige besonders wichtige Kombinatoren haben eigene Namen:
  - Identitätsfunktion: I ≡ λx.x
  - Konstanten-Funktional: $K \equiv \lambda xy.x$
  - Fixpunkt-Kombinator: $Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

Ausführung von λ Termen

Redex	Ein λ-Term der Form $(\lambda x.t_1)t_2$ heißt Redex.
β-Reduktion	entspricht der Ausführung der Funktionanwendung auf einem Redex: $(\lambda x.t_1)t_2 \Rightarrow t_1[x \rightarrow t_2]$
Substitution	$t_1[x \rightarrow t_2]$ erhält man aus dem Term t_1, wenn man alle freien Vorkommen von x durch t_2 ersetzt.
Normalform	Ein Term, der nicht weiter reduziert werden kann, heißt in Normalform

Beispiele:

$$(\lambda x.x)y \Rightarrow x[x \rightarrow y] = y$$

$$(\lambda x.x(\lambda x.x))(yz) \Rightarrow (x(\lambda x.x))[x \rightarrow (yz)] = ((yz)(\lambda x.x)$$

Braucht man primitive Operationen?

Nicht unbedingt - Kodierung mit Funktionen höherer Ordnung
Beispiel: let
let x = t_1 in t_2 wird zu $(\lambda x.t_2)t_1$
Beispiel: let x = g y in f x berechnet f(g y)
$(\lambda x.fx)(gy) \Rightarrow f(gy)$

Äquivalenz

α-Äquivalenz Namen gebundener Variablen

- dienen letztlich nur der Dokumentation
- entscheidend sind die Bindungen

α-Äquivalenz
t_1 und t_2 heißen α-Äquivalent ($t_1 \stackrel{\alpha}{=} t_2$), wenn t_1 in t_2 durch konsistente Umbenennung der λ-gebundenen Variablen überführt werden kann.

Beispiele:

$$\lambda x.x \overset{\alpha}{=} \lambda y.y$$

$$\lambda x.\lambda z.f(\lambda y.zy)x \overset{\alpha}{=} \lambda y.\lambda x.f(\lambda z.xz)y$$

aber

$$\lambda x.\lambda z.f(\lambda y.zy)x \overset{\alpha}{\neq} \lambda x.\lambda z.g(\lambda y.zy)x$$

$$\lambda z.\lambda z.f(\lambda y.zy)z \overset{\alpha}{\neq} \lambda x.\lambda z.f(\lambda y.zy)x$$

η-Äquivalenz Extensionalitäts-Prinzip:

- Zwei Funktionen sind gleich, falls Ergebnis gleich für alle Argumente

η-Äquivalenz
Terme λx.fx und f heißen η-äquivalent (λx.fx $\overset{\eta}{=}$ f), falls x nicht freie Variable von f ist.

Beispiele:

$$\lambda x.\lambda y.fzxy \overset{\eta}{=} \lambda x.fzx$$

$$fz \overset{\eta}{=} \lambda x.fzx$$

$$\lambda x.x \overset{\eta}{=} \lambda x.(\lambda x.x)x$$

aber

$$\lambda x.fxx \overset{\eta}{\neq} fx$$

Kodierung boolscher Werte

Church Booleans

- True wird zu: $C_{true} = \lambda t.\lambda f.t$
- False wird zu: $C_{false} = \lambda t.\lambda f.f$
- If-then-else wird zu: $If = \lambda a.a$

Beispiel

```
1    if True then x else y
```

ergibt: $(\lambda a.a)(\lambda t - \lambda f.t)xy = (\lambda t.\lambda f.t)xy = (\lambda f.x)y \Rightarrow x$

g = λ n. **if** isZero n **then** c_1 **else** (times n g (pred n))
G = λ g. λ n. **if** isZero n **then** c_1 **else** (times n g (pred n))
} // Keine Lambda-Terme

G = λ g. λ n. (λ a. a) (isZero n) c_1 (times n (g (sub n c_1)))
Y = λf. (λx. f (x x)) (λx.f (x x))
Fak = Y G

Fak c_2 = Y G c_2 ⇒ ((λx. G (x x)) (λx. G (x x))) c_2
⇒ G ((λx. G (x x)) (λx. G (x x))) c_2
⇒ (isZero c_2) c_1 (times c_2 ((λx. G (x x)) (λx. G (x x)) (pred c_2)))
⇒ times c_2 ((λx. G (x x)) (λx. G (x x)) (pred c_2))
⇒ times c_2 (YG ⇒ (λx. G (x x)) (λx. G (x x))) c_1
⇒ times c_2 ((isZero c_1) c_1 (times c_1 ((λx. G (x x)) (λx. G (x x)) (pred c_1))))
⇒ times c_2 (times c_1 ((λx. G (x x)) (λx. G (x x)) (pred c_1)))
⇒ times c_2 (times c_1 ((is_zero c_0) c_1 ...)) ⇒ c_2

- if True then x else y ergibt:
  (λa.a)(λt.λf.t) x y ⇒ (λt.λf.t) xy ⇒ (λf.x)y⇒ x
- b_1 && b_2 ist äquivalent zu if b_1 then b_2 else False
  ⇒ b_1 && b_2 wird zu (λa.a) b_1 b_2 C_false
  ⇒ b_1 && b_2 wird zu (λa.a) b_1 b_2 (λt.λf.f)
- True && True ergibt:
  (λa.a)C_true C_true (λt.λf.f)
  ⇒ (λt.λf.t)(λt.λf.t)(λt.λf.f)
  ⇒ (λf.(λt.λf.t)) (λt.λf.f)⇒ λt.λf.f = C_true
- $b_1 \vee b_2$ entspricht:
  if b_1 then True else b_2
- $\neg b_1$ entspricht:
  if b_1 then False else True
- $b_1 \Rightarrow b_2$ entspricht:
  if b_1 then b_2 else True

Kodierung natürlicher Zahlen

Eine natürliche Zahl drückt aus, wie oft etwas geschehen soll. $c_0 = \lambda s.\lambda z.z$; $c_1 = \lambda s.\lambda z.sz$; $c_2 = \lambda s.\lambda z.s(sz)$;...;$c_n = \lambda s.\lambda z.s^n z$
Arithmetische Operationen

- Addition: $plus = \lambda m.\lambda n.\lambda s.\lambda z.ms(nsz)$
- Multiplikation: $times = \lambda m.\lambda n.\lambda s.n(ms) = \lambda m.\lambda n.\lambda s.\lambda z.n(ms)z$
- Exponentiation: $exp = \lambda m.\lambda n.nm = \lambda m.\lambda n.\lambda s.\lambda z.nmsz$
- Vorgänger: $pred = \lambda n.\lambda s.\lambda x.n(\lambda y.\lambda z.z(ys))(Kx)$
- Subtraktion: $sub = \lambda n.\lambda m.mpredn$
- Nullvergleich: $isZero = \lambda n.n(\lambda x.Cfalse)Ctrue$

Nachrechnen: $M\ \overline{n_k} = N\ \overline{n_k}\ c_0$

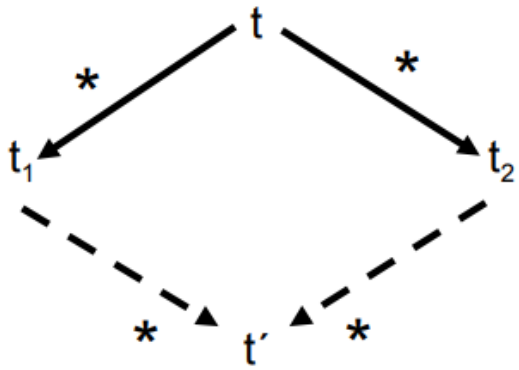
$$= \begin{cases} c_0 & falls\ G\ \overline{n_k}\ c_0 = c_0 \\ N\ \overline{n_k}\ c_1 & sonst \end{cases}$$

$$= \begin{cases} c_1 & falls\ G\ \overline{n_k}\ c_1 = c_0 \\ N\ \overline{n_k}\ c_2 & sonst \end{cases}$$

...

$\text{succ}(c_2) = (\lambda n.\lambda s.\lambda z.s\ (n s\ z))\ (\lambda s.\lambda z.s(sz))$
$\Rightarrow \lambda s.\lambda z.s((\lambda s.\lambda z.\ s\ (sz))\ s\ z)$
$\Rightarrow \lambda s.\lambda z.s((\lambda z.s\ (s\ z))z)$
$\Rightarrow \lambda s.\lambda z.s(s(sz)) = c_3$

Rechnen mit Church - Zahlen




```
1  Runnable task = () -> {
2      String me = Thread.currentThread().getName();
3      System.out.println("Hallo " + me);
4  };
5
6  task.run();
7
8  Thread thread = new Thread(task);
9  thread.start();
```

Idee zu exp:
exp $c_m c_n \Rightarrow c_n c_m \Rightarrow (\lambda.s.\lambda z.s^n z)(\lambda s.\lambda z.s^m z)$
$\Rightarrow \lambda z.(\lambda s.\lambda z.s^m z)^n z$
(per Induktion über n) $\overset{\alpha\beta\eta}{\Rightarrow} \lambda s.\lambda z.\lambda z.s^{m\,n} z = {c_m}^n$

Arithmetische Operationen

Vorgänger:	pred	=	λn.λs.λx. n (λy.λz. z (y s))(K x) I
Subtraktion:	sub	=	λn.λm. m pred n
Nullvergleich:	isZero	=	λn. n (λx. C_{false}) C_{true}

$isZero(c_0) = (\lambda n.n(\lambda x.C_{false})C_{true})(\lambda s.\lambda z.z)$
$\Rightarrow (\lambda s.\lambda z.z)(\lambda x.C_{false})C_{true}$
$\Rightarrow (\lambda z.z)C_{true} \Rightarrow C_{true}$ (Bemerkung: I und K sind die Identitätsfunktion bzw. das Konstanten-Funktional)

pred(c_2) = (λn.λs.λx. n (λy.λz. z (y s))(K x) I) (λs. λz. s (s z))
⇒ λs.λx. (λs´. λz´. s´ (s´ z´)) (λy.λz. z (y s)) (K x) I
⇒ λs.λx. (λz´. (λy.λz. z (y s)) ((λy.λz. z (y s)) z´)) (K x) I
⇒ λs.λx. (λy.λz. z (y s)) ((λy.λz. z (y s)) (K x)) I
⇒ λs.λx. (λz´. z´ (((λy.λz. z (y s)) (K x)) s)) I
⇒ λs.λx. I (((λy.λz. z (y s)) (K x)) s)
⇒ λs.λx. I ((λz. z (K x s)) s)
⇒ λs.λx. I (s (K x s))
⇒ λs.λx. (λx´.x´) (s ((λx´´.λy.x´´) x s))
⇒ λs.λx. s ((λx´´.λy.x´´) x s)
⇒ λs.λx. s ((λy.x) s)
⇒ λs.λx. s x = c_1

Auswertungsstrategien

Wenn es in einem Term mehrere Redexe gibt, welchen reduziert man dann?

$$(\lambda x.x)((\lambda x.x)(\lambda z.(\lambda x.x)z))$$
$$(\lambda x.x)((\lambda x.x)(\lambda z.(\lambda x.x)z))$$
$$(\lambda x.x)((\lambda x.x)(\lambda z.(\lambda x.x)z))$$
$$(\lambda x.x)((\lambda x.x)(\lambda z.(\lambda x.x)z)$$
$$(\lambda x.x)((\lambda x.x)(\lambda z.(\lambda x.x)z))$$

Volle β-Reduktion: Jeder Redex kann jederzeit reduziert werden

$$(\lambda x.x)((\lambda x.x)(\lambda z.(\lambda x.x)z))$$
$$\Rightarrow (\lambda x.x)((\lambda x.x)(\lambda z.z))$$
$$\Rightarrow (\lambda x.x)(\lambda z.z))$$
$$\Rightarrow (\lambda z.z) \nRightarrow$$
$$(\lambda x.x)((\lambda x.x)(\lambda z.(\lambda x.x)z))$$

Volle β-Reduktion: Jeder Redex kann jederzeit reduziert werden
Normalreihenfolge: Immer der linkeste äußerste Redex wird reduziert

$$(\lambda x.x)((\lambda x.x)(\lambda z.(\lambda x.x)z))$$
$$\Rightarrow (\lambda x.x)(\lambda z.(\lambda x.x)z)$$
$$\Rightarrow \lambda z.(\lambda x.x)z$$
$$\Rightarrow (\lambda z.z)) \not\Rightarrow$$

Fixpunktsatz und Rekursion

Divergenz

Bisherige Beispiele werten zu einer Normalform aus. Aber:

$$\omega = (\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx)$$

$\lambda x.xx$ wendet sein Argument auf das Argument selbst an $\Rightarrow$ dadurch reproduziert ω sich selbst.

Divergenz:
Terme, die nicht zu einer Normalform auswerten, divergieren. Diese modellieren unendliche Ausführungen.

Der Fixpunktsatz

Fixpunktsatz
Für alle $F \in \Lambda$ existiert ein $X \in \lambda$ sodass gilt: $FX = X$

- Der Fixpunktsatz besagt, dass im Lambda-Kalkül jeder Term einen Fixpunkt hat, d.h. einen Wert, der auf sich selber abgebildet wird.
- Beweis:
  - Zu jedem beliebigen F sei $W = \lambda x.F(xx)$ und $X = (WW)$
  - Dann gilt: $X \equiv WW \equiv (\lambda x.F(xx))W \equiv F(WW) \equiv FX$
- Bemerkungen:
  - Für einige Lambda-Terme ist die Identifikation eines Fixpunktes einfach, z.B. für den Term $\lambda x.x$ (alle Terme sind Fixpunkte)
  - Für andere Terme, wie $\lambda xy.xy (= \lambda x.\lambda y.xy)$ ist das nicht so klar
  - Der Beweis des Fixpunktsatzes ist konstruiv, d.h. er liefert zu jedem Lambda-Term einen Fixpunkt

Anwendung des Fixpunktsatzes

- Aufgabe: Berechne den Fixpunkt zum Term $\lambda xy.xy$
  - Lösungsansatz: $W \equiv \lambda x.(\lambda xy.xy)(xx) \equiv \lambda x.\lambda y.(xx)y \equiv \lambda xy.(xx)y$
  - Damit ist der gesuchte Fixpunkt $X \equiv ((\lambda xy.(xx)y)(\lambda xy.(xx)y))$
  - Nachrechnen:
    $(\lambda xy.xy)((\lambda xy.(xx)y)(\lambda xy.(xx)y))$
    $\equiv (\lambda x.\lambda y.xy)((\lambda xy.(xx)y)(\lambda xy.(xx)y)$
    $\equiv \lambda y.((\lambda xy.(xx)y))(\lambda xy.(xx)y)y$
    $\equiv (\lambda xy.(xx)y)(\lambda xy.(xx)y)$
    $\equiv$ X
- Bemerkung: Der so für die Identitätsfunktion $\lambda x.x$ konstruierte Fixpunkt ist übrigens $(\lambda x.xx)(\lambda x.xx)$, er spielt die besondere Rolle des Standardterms $\perp$ für nicht-terminierende Ausführungen

Der Fixpunkt-Kombinator

Im Ergebnis unserer Diskussion des Fixpunktsatzes definieren wir den Fixpunkt-Kombinator wie folgt:

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

- Dieser Kombinator spielt eine wichtige Rolle bei der Definition rekursiver Funktionen im Lambda-Kalkül, wie wir im folgenden sehen werden
- Für jeden Lambda-Term M gilt: $YM = M(YM)$
  - Beweisidee: zeige, dass beide Terme auf einen identischen Term reduziert werden können
- Der Term Y ist übrigens nicht der einzige Kombinator, der Fixpunkte zu Lambda-Termen konstruiert
  - A. Turing: $\Theta \equiv (\lambda xy.y(xxy))(\lambda xy.y(xxy))$

Rekursion im Lambda-Kalkül

- Die bisher definierten Funktionen waren alle nicht-rekursiv
- Viele Funktionen kann man aber nur unter Zuhilfenahme von Rekursion (bzw. Iteration) beschreiben
- In üblichen Programmiersprachen werden rekursive Funktionsdefinitionen durch die Verwendung von Namen für Funktionen möglich - man verwendet hierbei einfach den Namen der gerade zu definierenden Funktion im Rumpf der Definition:

```
1
2    \item fun fak(i) -> if (i = 0) then 1 else i * fak(i-1).
```

- Im Lambda-Kalkül gibt es jedoch keine Namen für Funktionen:
  - Daher stellt man eine rekursive Funktion f mittels einer Funktion G dar, die einen zusätzlichen Parameter g hat, an den man dann G selber bildet
  - Schaut kompliziert aus, ist es auch (Q-Q)
  - Warum so kompliziert? Damit die Definition von G im eigenen Rumpf verfügbar ist

Rekursive Funktionen sind Fixpunkte

- Rekursive Funktion von g
  - $g = \lambda n...g...n...$ Rumpf verwendet g
- Daraus gewinnt man das Funktional
  - $G = \lambda g.\lambda n...g...n...$
- Falls G einen Fixpunkt g* hat, d.h. $G(g*) = g*$, so
  - $g* = G(g*) = \lambda n...g*...n$
- Vergleiche: $g = \lambda n...g...n...$

Rekursive Definition $\Leftrightarrow$ Fixpunkt des Funktionals

- Beispiel: Fakultät
  - $g = \lambda n.$ if isZero n then c_1 else (times n g(pred n)) - rekursiv
  - $G = \lambda g.\lambda n.$ if isZero n then c_1 else (times n g(pred n)) - funktional

Der Fixpunktkombinator dient als Rekursionsoperator

Wir berechnen den gesuchten Fixpunkt des Funktionals G mit dem Fixpunktkombinator, der somit als Rekusrsionsoperator dient: Rekursionsoperator

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$
$$Yf = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))f$$
$$\Rightarrow (\lambda x.f(xx))(\lambda x.f(xx))$$
$$\Rightarrow f((\lambda x.f(xx))(\lambda x.f(xx)) \Leftarrow f(Yf)())$$

also $f(Yf) \overset{\beta}{=} Yf$
d.h. Yf ist Fixpunkt von f

```
g   =  λ n. if isZero n then c1 else (times n g (pred n))
G   =  λ g. λ n. if isZero n then c1 else (times n g (pred n))   } // Keine Lambda-Terme
G   =  λ g. λ n. (λ a. a) (isZero n) c1 (times n (g (sub n c1)))
Y   =  λf. (λx. f (x x)) (λx.f (x x))
Fak =  Y G

Fak c2 = Y G c2 ⇒ ((λx. G (x x)) (λx. G (x x))) c2
      ⇒ G ((λx. G (x x)) (λx. G (x x))) c2
      ⇒ (isZero c2) c1 (times c2 ((λx. G (x x)) (λx. G (x x)) (pred c2)))
      ⇒ times c2 ( (λx. G (x x)) (λx. G (x x)) (pred c2 ) )
                          YG ⇒
      ⇒ times c2 ( ((λx. G (x x)) (λx. G (x x))) c1
      ⇒ times c2 ((isZero c1) c1 (times c1 ((λx. G (x x)) (λx. G (x x)) (pred c1 ))))
      ⇒ times c2 ( times c1 ( (λx. G (x x)) (λx. G (x x)) (pred c1) ) )
      ⇒ times c2 (times c1 ((is_zero c0) c1 ...))  ⇒  c2
```

Beispiel: Fakultät im Lambda-Kalkül

Ausdrucksstärke des Lambdakalküls

- Im Folgenden wollen wir zeigen, dass der Lambda-Kalkül genau die rekursiven Funktionen beschreibt
- Eine numerische Funktion ist eine Abbildung $f : \mathbb{N}^k \to \mathbb{N}$ mit $k \in \mathbb{N} \cap \{0\}$
- Wir definieren hierzu:
  - Anfangsfunktionen:
    - Projektion: $U_i^k(n_1, n_2, ..., n_k) = n_i$ für $1 <= i <= k$
    - Nullfunktion: $Z(n) = 0$
    - Nachfolger: $S(n) = n + 1$
  - Minimalisierung:
    - Für eine Relation $P(m)$ bezeichne $\mu m[P(m)]$ die kleinste Zahl m sodass P(m) gilt.
- Bemerkung: im Folgenden notieren wir $n_1, n_2, ..., n_k$ kurz als $\overline{n_k}$
- Eine numerische Funktion ist Lambda-definierbar, wenn es einen Kombinator M gibt, sodass $M\overline{n_k} = f(\overline{n_k})$
- Im folgenden sei C eine Klasse von numerischen Funktionen, und es gelte $g, h, h_1, h_2, ..., h_m \in C$
- Wir definieren nun die folgenden Eigenschaften:
  - C ist abgeschlossen unter Komposition, wenn für jede Funktion f, die über $f(\overline{n_k}) := g(h_1(\overline{n_k}), ..., h_m(\overline{n_k}))$ definiert ist, gilt $f \in C$
  - C ist abgeschlossen unter primitiver Rekursion, wenn für jede Funktion f, die über
    $$f(0, \overline{n_k}) = g(\overline{n_k})$$
    $$f(j + 1, \overline{n_k}) = h(f(j, \overline{n_k}), j, \overline{n_k})$$
    definiert ist, gilt: $f \in C$
  - C ist abgeschlossen unter unbeschränkter Minimalisierung , wenn für jede Funktion f, die über $f(\overline{n_k}) = \mu m[g(\overline{n_k}, m) = 0]$ definiert ist (wobei für alle $\overline{n_k}$ ein m existiere, sodass $g(\overline{n_k}, m) = 0$ ist), gilt $f \in C$

Definition:
Die Klasse der rekursiven Funktionen ist die kleinste Klasse numerischer Funktionen, die alle oben genannten Anfangsfunktionen enthält und abgeschlossen ist unter Komposition, primitiver Rekursion und unbeschränkter Minimalisierung

- Lemma 1: Die Anfangsfunktionen sind Lambda-definierbar
- Beweis:
  - $U_i^k = \lambda x_1 x_2 ... x_k.x_i$
  - $S = \lambda n.\lambda s.\lambda z.s(nsz)$ (siehe succ bei Churchzahlen)
  - $Z = \lambda fx.x$ (siehe c_0 bei Churchzahlen)

- Lemma 2: Die Lambda-definierbaren Funktionen sind abgeschlossen unter primitiver Rekursion
- Beweis: Sei f definiert über
  $$f(0, \overline{n_k}) = g(\overline{n_k})$$
  $$f(j + 1, \overline{n_k}) = h(f(j, \overline{n_k}), j, \overline{n_k})$$
  und seien g und h Funktionen (die per Induktionsvoraussetzung) durch die Lambda-terme G und H berechnet werden
  - Intuitiv kann f berechnet werden, indem man überprüft ob j = 0 ist, und wenn ja $g(\overline{n_k})$, ansonsten $h(f(j, \overline{n_k}), j, \overline{n_k})$
  - Ein Term M hierfür existiert laut Fixpunktsatz und es gilt: $M \equiv Y(\lambda f\, x\, \overline{y_k}.if(isZero\, x)(G\, \overline{y_k})(H(f(pred\, x)\overline{y_k})(pred\, x)\overline{y_k}))$

- Lemma 3: Die Lambda-definierbaren Funktionen sind abgeschlossen unter unbeschränkter Minimalisierung
- Beweis:
  - Sei f über $f(\overline{n_k}) = \mu m[g(\overline{n_k}, m) = 0]$ definiert, wobei g (per Induktionsvoraussetzung) durch den Lambda-Term G berechnet wird
  - Intuitiv kann man f berechnen, indem man bei 0 beginnend für m überprüft, ob $g(\overline{n_k}, m) = 0$ ist, und wenn ja m ausgibt, ansonsten die Überprüfung mit $m + 1$ fortsetzt
  - Ein Term für eine solche Funktion kann laut Fixpunktsatz konstruiert werden und man erhält mit Anwendung des Fixpunktkombinators zunächst:
    $$N \equiv Y(\lambda f\, \overline{x_k}\, y.if(isZero\, (G\, \overline{x_k}\, y))y(f\, \overline{x_k}\, (succ\, y)))$$
  - Nun definiert man die Funktion f durch den folgenden Term M:
    $$M \equiv \lambda \overline{x_k}.N\, \overline{x_k}\, c_0$$

Nachrechnen:

$$
\begin{aligned}
M\ \overline{n_k} &= N\ \overline{n_k}\ c_0 \\
&= \begin{cases} c_0 & falls\ G\ \overline{n_k}\ c_0 = c_0 \\ N\ \overline{n_k}\ c_1 & sonst \end{cases} \\
&= \begin{cases} c_1 & falls\ G\ \overline{n_k}\ c_1 = c_0 \\ N\ \overline{n_k}\ c_2 & sonst \end{cases} \\
&\ldots
\end{aligned}
$$

- Aus den Lemmata 1 bis 3 folgt nun der Satz:

  Alle rekursiven Funktionen sind Lambda-definierbar

Berechnungsreihenfolgen und Konfluenz

Noch einmal Auswertungsstrategien

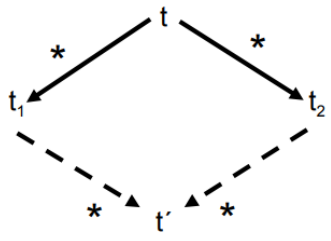
- Bei unserer initialen Betrachtung der Auswertungsstrategien haben wir die volle β-Rekursion und die Normalreihenfolge kennengelernt
- Nun wollen wir unsere Betrachtungen hierzu noch einmal vertiefen und definieren zunächst:
  - Ein Redex wird als "äußerst"(outermost) bezeichnet, wenn er nicht Teil eines anderen Redex ist.
  - Ein Redex wird als ïnnerst"(innermost) bezeichnet, wenn er keinen eigenständigen Redex beinhaltet
- Mit diesen Begriffen können im folgenden die gebräuchlichsten Auswertungsstrategien formuliert werden
  - Normal Order: Evaluiere Argumente so oft, wie sie verwendet werden
  - Applicative Order: Evaluiere Argumente einmal
  - Lazy Evaluation: Evaluiere Argumente höchstens einmal
- Eine zentrale Kernfrage: Welche Auswertungsstrategie führt (möglichst schnell) zu einem nicht weiter reduzierbaren Term?
  - Bei unserer beispielhaften Berechnung des Terms Fak c_2 haben wir nach der initialen Anwendung des Fixpunktkombinators zunächst den Term isZero c_2 reduziert.
  - Ebenso hätten wird den weiter innen stehenden Fixpunktkombinator zuerst erneut anwenden können(bei voller β-Reduktion kann jeder Term jederzeit reduziert werden).
  - Auf diese Weise hätten wir unendlich oft vorgehen, damit einen immer länger werdenden Term ableiten können und somit nicht das gewünschte Resultat c_2 berechnet.
- Eine weitere Kernfrage: Angenommen mehrere unterschiedliche Reduktionsreihenfolgen führen zu einem nicht weiter zu reduzierenden Ergebnis - führen all diese Reihenfolgen zum gleichen Ergebnis?
- Wir definieren zuerst einen zentralen begriff in diesem Zusammenhang:

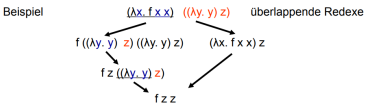
  Ein Transitiosnsystem $(D, \rightarrow *)$ heißt genau dann konfluent, wenn für alle $t, t_1, t_2 \in D$ gilt: wenn $t \rightarrow *t_1$ und $t \rightarrow *t_2$, dann gibt es ein $t' \in D$ mit $t_1 \rightarrow *t'$ und $t_2 \rightarrow *t'$
- Wenn der Lambda-Kalkül konfluent ist, kann hieraus gefolgert werden, dass unterschiedliche Reduktionsreihenfolgen, die zu einer nicht mehr weiter zu reduzierenden Form führen, somit auf den gleichen Term führen müssen.
- Achtung: hieraus kann nicht gefolgert werden, dass alle Reduktionsreihenfolgen auf den gleichen Term führen, da dies ja nur für "terminierende"Reduktionsreihenfolgen gilt!

Church-Rosser-Eigenschaft

Satz (Church-Rosser)

Der untypisierte λ-Kalkül ist konfluent: Wenn $t \overset{*}{\Rightarrow} t_1$ und $t \overset{*}{\Rightarrow} t_2$, dann gibt es ein t' mit $t_1 \overset{*}{\Rightarrow} t'$ und $t_2 \overset{*}{\Rightarrow} t'$





Beweisidee: Definiere $\overset{\rightarrow}{\Rightarrow}$ als "parallele"β-Reduktion.

- Es gilt: $\Rightarrow \subseteq \overset{\rightarrow}{\Rightarrow} \subseteq \overset{*}{\Rightarrow}$
- Zeige Diamant Eigenschaft für $\overset{\rightarrow}{\Rightarrow}$

Eindeutigkeit der Normalform

Korollar (Eindeutigkeit der Normalform)

Die Normalform eines λ-Terms ist - sofern sie existiert - eindeutig.

Beweis:

- t_1 und t_2 Normalformen von t, d.h. $t \overset{*}{\Rightarrow} t_1 \not\Rightarrow$ und $t \overset{*}{\Rightarrow} t_2 \not\Rightarrow$
- Nach Chruch-Rosser gibt es t' mit $t_1 \overset{*}{\Rightarrow} t'$ und $t_2 \overset{*}{\Rightarrow} t'$
- Nach Annahme $t_1 \not\Rightarrow$ und $t_2 \not\Rightarrow$, also $t_1 = t' = t_2$

Bei terminierenden β-Reduktionen ist irrelevant, welchen Redex man zuerst reduziert!

Auswertung von Parametern in Programmiersprachen

Behandlung von Parametern in Programmiersprachen

- Die Art und Weise, wie in einer Programmiersprache Parametter übergeben - d.h. wie die Reihenfolge und die Zeitpunkte ihrer Auswertung gehandhabt - werden, hat Einfluss auf wichtige Eigenschaften der Sprache:
  - Effizienz der Berechnungen
  - Termininerungsverhalten
  - Ausdruckskraft
- Hierbei ist es insbesondere von Interesse, wie Parameter gehandhabt werden, deren Werte undefiniert sind (z.B. 1/0)

  Wir definieren zunächst den zentralen begriff ßtrikt":
  Eine n-stellige Funktion heißt strikt im k-ten Argument ($1 <= k <= n$), wenn gilt: $f(x_1, x_2, ..., x_{k-1}, \perp, x_{k+1}, ..., x_n) = \perp$

- Ein undefiniertes Argument führt hier zu einem undefinierten Resultat
- Grundsätzlich kann man die Auswertungsstrategien von Programmiersprachen in strikte und nicht-strikte Strategien einteilen; sehr gebräuchlich sind dabei insbesondere:
  - Call by Value: Ausdrücke, die Parameter bei einem Funktionsaufruf beschreiben, werden vor der Übergabe an die Funktion vollständig ausgewertet
  - Call by Name: Ausdrücke, die Parameter bei einem Funktionsaufruf beschreiben, werden nicht bei Übergabe, sondern erst dann ausgewertet, wenn sie in der aufgerufenen Funktion tatsächlich benötigt werden
- Beide Varianten haben spezifische Vor- und Nachteile:
  - Call by Value: weniger Berechnungsaufwand, wenn ein Parameter mehr als einmal im Funktionsrumpf vorkommt; weniger Speicheraufwand bei der Übergabe
  - Call by Name: weniger Berechnungsaufwand, wenn ein Argument nicht zum Ergebnis beiträgt; höherer Aufwand bei Übergabe
- Die Programmiersprache Erlang realisiert grundsätzlich eine strikte Handhabung von Parametern, da sie die Strategie Call by Value verwendet
- Allerdings wird bei der Definition einer Funktion der resultierende Wert erst dann berechnet, wenn die Funktion ausgewertet wird
  - Das erlaubt über den Umweg zusätzlicher Funktionsdefinitionen auch die Realisierung einer nicht-strikten Auswertungsstrategie - ermöglicht Nachbildung der sogenannten Lazy-Evaluation
  - hierbei wird ein nicht-strikt zu evaluierendes Argument als Resultat einer anonymen nullstelligen Funktion (ohne Parameter) "verpackt"
  - Im Rumpf der eigentlichen Funktion wird diese Funktion dann ausgewertet (= aufgerufen), wenn feststeht, dass dieses Argument für die Berechnung des Ergebnisses benötigt wird
  - Andere funktionale Sprachen wie Haskell oder Gofer verwenden Call by Name und realisieren damit grundsätzlich Lazy-Evaluation

```
-module(lazy).
-export([test1/3, test2/3]).
test1(P, A, B) ->    % A and B are arbitrary values
  if
    P==true  -> A;
    P==false -> B
 end.
test2(P, A, B) ->    % A and B have to be functions
  if
    P==true  -> A();
    P==false -> B()
 end.
```

```
> lazy:test1(true, 3, 4/0).
** exception error: bad argument in an
     arithmetic expression
     in operator  '/'/2
        called as 4 / 0
> lazy:test2(true, fun() -> 3 end, fun() -> 4/0 end).
3
```

- Erläuterungen:
  - Im zweiten Beispiel wird der Rückgabewert der übergebenen Funktionne nur ausgewertet, wenn sie im Rumpf der auszuführenden Funktion aufgerufen werden
  - Innerhalb von Erlang-Modulen kann man sich mit Hilfe einer Macro-Definition Schreibarbeit sparen:

```
1
2  -define(DELAY(E), fun() -> E end).
3  check() -> test2(true, ?DELAY(3), ?DELAY(4/0)).
```

- Je nachdem, ob und wie häufig ein übergebener Parameter im Funktionsrumpf benötigt wird, können bei Lazy-Evaluation Berechnungen
  - komplett eingespart oder
  - (in identischer Form) wiederholt erforderlich werden
  - Unter Umständen kann man in der betreffenden Funktion durch Einführung einer temporären Variable redundante Mehrfachberechnungen einsparen ($\rightarrow$ Call by Need)
- Die Parameterübergabe ist bei Call by Name in der Regel aufwändiger als bei Call by Value
  - Die meisten Programmiersprachen (Java, C, C++, Pascal etc.) verwenden daher Call by Value ($\rightarrow$ strikte Auswertung)
  - Eine Ausnahme wird oft bei dem IF-Konstrukt gemacht (der auszuführende Code ist hier ja meist auch kein Parameter)
- Zu Ausdrucksstärke: während strikte Funktionen durch die Strategie Call by Value realisiert werden, ist es nicht so, dass Lazy Evaluation es erlaubt, alle nicht-strikten Funktionen zu realisieren
  - Die folgenden Gleichungen definieren eine nicht-strikte Multiplikation $\otimes$ auf der Basis der Multiplikation $\cdot$ für Zahlen:

$$0 \otimes y = 0$$
$$x \otimes 0 = 0$$
$$x \otimes y = x * y$$

  - Wenn ein Arguemnt undefiniert ist, dann liefert $\otimes$ ein Ergebnis, sofern das andere Argument zu 0 evaluiert wird ($\rightarrow fak(-1) \otimes (fak(3) - 6)$)
  - Implementiert werden kann die Funktion nur durch eine Art von paralleler Auswertung mit Abbruch der anderen Berechnung sobald 0 als Resultat berechnet und zurückgegeben wurde
- Wir betrachten nun die Beziehungen zwischen Parameterbehandlung in Programmiersprachen und Reduktion von Lambda-Termen

Auswertungsstrategien & Programmiersprachen

Werte in Programmiersprachen wie Haskell:

- Primitive Werte: 2, True
- Funktionen: $(\backslash x \rightarrow x)$, (&&), $(x \rightarrow (\backslash y \rightarrow y + y)x)$

Werte im λ-Kalkül:

- Abstraktionen: $c_2 = \lambda s.\lambda z.s\ (s\ z)$, $C_{true} = \lambda t - \lambda f.t$, $\lambda x.x$, $\lambda b_1.\lambda b_2.\ b_1\ b_2 (\lambda t.\lambda f.\ f)$, $\lambda x.\ (\lambda y.\ plus\ yy)x$

Auswertungsstrategie: Keine weitere Reduzierung von Werten
Reduziere keine Redexe unter Abstraktionen (umgeben von λ):
$\Rightarrow$ call-by-name, call-by-value

Call-By-Name

Call-By-Name: Reduziere linkesten äußersten Redex

- Aber nicht falls von einem λ umgeben

  $(\lambda y.(\lambda x.y(\lambda z.z)x))((\lambda x.x)(\lambda y.y))$

  $(\lambda x.((\lambda x.x)(\lambda y.y))(\lambda z.z)x)$

Intuition: Reduziere Argumente erst, wenn benötigt

Auswertung in Haskell: Lazy-Evaluation = call-by-name (+sharing)

- Standard-Auswertungsstrategie für Funktionen/Konstruktoren
- listOf x = x : listOf x
- 3: listOf 3 $\nRightarrow$
- (div 1 0) : (6 : [])
- tail ((div 1 0): (6 : [])) $\Rightarrow$ 6 : [] $\nRightarrow$

Call-By-Value

Call-By-Value: Reduziere linkesten Redex

- der nicht einen λ umgibt
- und dessen Argument ein Wert ist

  $(\lambda y.(\lambda x.y(\lambda z.z)x)((\lambda x.x)(\lambda y.y))$

  $\Rightarrow (\lambda y(\lambda x.y(\lambda z.z)x))(\lambda y.y)$

  $\Rightarrow (\lambda x.(\lambda y.y(\lambda z.z)x)) \nRightarrow$
- Intuition: Argumente vor Funktionsaufruf auswerten
- Auswertungsstrategie vieler Sprachen: Java, C, Scheme, ML, ...
- Arithmetik in Haskell: Auswertung by-value
- $prodOf(x) = y * prodOfx$
- $3 * prodOf3 \Rightarrow 3 * (3 * prodOf3) \Rightarrow ...$
- $((div10) * 6) * 0 \Rightarrow \bot$
- $((div22) * 6) * 0 \Rightarrow ((1 * 6) * 0) \Rightarrow 6 * 0 \Rightarrow 0 \nRightarrow$

Vergleich der Auswertungsstrategien

Call-by-name vs. Call-by-value

- Werten nicht immer zur Normalform aus $\lambda x.(\lambda y.y)x$
- Gibt es Normalform, dann darauf β-reduzierbar (Church-Rosser)
- Call-by-name terminiert öfter
- $Y(\lambda y.z) = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))(\lambda y.z)$

  $\Rightarrow \lambda x.(\lambda y.z(x\ x))(\lambda x.(\lambda y.z)(x\ x))$

  $\Rightarrow (\lambda y.z)((\lambda x.(\lambda y.z)(x\ x))(\lambda x.(\lambda y.z(x\ x))) \overset{cbn}{\Rightarrow} z$

  $\overset{cbv}{\Rightarrow} (\lambda y.z)((\lambda x.(\lambda y.z)(x\ x))(\lambda x.(\lambda y.z(x\ x))))$

  $\overset{cbv}{\Rightarrow} (\lambda y.z)((\lambda y.z)((\lambda x.(\lambda y.z)(x\ x))(\lambda x.(\lambda y.z(x\ x)))))$

Standardisierungssatz
Wenn t eine Normalform hat, dann findet Normalreihenfolgenauswertung diese.

Abschließende Bemerkungen

- Der Lambda-Kalkül wurde in den dreißiger Jahren des 20. Jahrhunderts von Alonzo Church erfunden, um damit grundsätzliche Betrachtungen über berechenbare Funktionen anzustellen
- Trotz der Einfachheit der dem Kalkül zugrunde liegenden Regeln, realisiert er ein universelles Berechnungsmodell
- Der Lambda-Kalkül hat die Entwicklung zahlreicher, für die Informatik wichtiger Konzepte beeinflusst
  - Funktionale Programmiersprachen (die minimalen Funktionen von Lisp wurden auf Grundlage des Lambda-Kalküls definiert)
  - Forschund zu Typsystemen für Programmiersprachen
  - Repräsentation von Logik-Termen im Lambda-Kalkül führte zu Theorembeweisen für Logiken höherer Stufen
- Manche Puristen vertreten gelegentlich die Ansicht, dass funktionale Programmiersprachen nicht viel mehr sind, als "Lambda-Kalkül mit etwas syntaktischem Zucker"

Zusammenfassung

- Funktionale Programmierung folgt einem verallgemeinerten Konzept der Funktionsauswertung
- Die Programmiersprache Erlang ist dynamisch typisiert und unterstützt auch Funktionen höherer Ordnung
- Manche Algorithmen lassen sich in Erlang aufgrund der mächtigen Listenkonstrukte und des flexiblen Pattern Matching sehr kompakt formulieren ($\rightarrow$ Potenzmenge, Quicksort)
- Das heißt jedoch nicht, dass sehr kompakter Code auch zu sehr effizientem Laufzeit- und/oder Speicherbedarf führt - teilweise muss der Code relativ geschickt optimiert werden, um einigermaßen effiziente Lösungen zu erhalten ($\rightarrow$ Quicksort)
- Manche Aufgaben, die in imperativen Programmiersprachen sehr effizient und einfach lösbar sind ($\rightarrow$ Teilen einer Liste in gleich große Hälften) sind mittels Listen nur recht umständlich und aufwendig lösbar
- Es gilt in der Praxis also abzuwägen, für welche Aufgaben eine funktionale Sprache eingesetzt werden soll

Multithreading und parallele Programmierung

Grundlagen

Lernziele

- Programmierung paralleler Algorithmen und Verfahren als Paradigma
- Verständnis grundlegender Architekturen und Modelle
- Praktische Erfahrungen mit Erlang, Java und C++

Grundbegriffe

- Prozess := Programm in Ausführung; Ausführungsumgebung für ein Programm; hat eigenen Adressraum; Prozessor kann immer nur einen Prozess ausführen
- Thread ("Faden") := leichtgewichtige Ausführungseinheit oder Kontrollfluss (Folge von Anweisungen) innerhalb eines sich in Ausführung befindlichen Programms; "leichtgewichtigïm Vergleich zu Betriebssystemprozess; Threads eines Prozesses teilen sich Adressraum; Thread kann von CPU oder Core ausgeführt werden
- Shared Memory := Kommunikation (über Variablen im) gemeinsamen Speicher; Prozess kann direkt auf Speicher eines anderen Prozesses zugreifen; erfordert explizite Synchronisation, z.B. über kritische Abschnitte
- Message Passing := Prozesse mit getrennten Adressräumen; Zugriff nur auf eigenen Speicher; Kommunikation durch explizites Senden/Empfangen von Nachrichten; implizite Synchronisation durch Nachrichten
- Parallelisierungsarten
  - Instruktionsparallelität: parallele Ausführung mehrerer Operationen durch eine CPU-Instruktion; explizit: Vektorinstruktionen, SIMD; implizit: Pipelining von Instruktionen
  - Taskparallelität: Ausnutzung inhärenter Parallelität durch simultane Ausführung unabhängiger Aufgaben
  - Datenparallelität: Gemeinsame Operation auf homogener Datenmenge; Zerlegung eines Datensatzes in kleinere Abschnitte

The free launch is over

Taktfrequenz wächst nur noch langsam

- Physikalische Gründe: Wärmeentwicklung, Energiebedarf, Kriechströme,...

Auswege

- Hyperthreading:
  - Abarbeitung mehrerer Threads auf einer CPU (5-15 % Performancegewinn)
  - Einfache Hardwareunterstützung (einige Register)
- Multicore:
  - Mehrere CPUs auf einem Chip
  - Billiger als echte Mehrprozessorsysteme
- Caching:
  - Vergrößerung L1, L2, L3 Cache
  - Speicherzugriff 10-50 · teurer als Cachezugriff

Konsequenzen und Trends

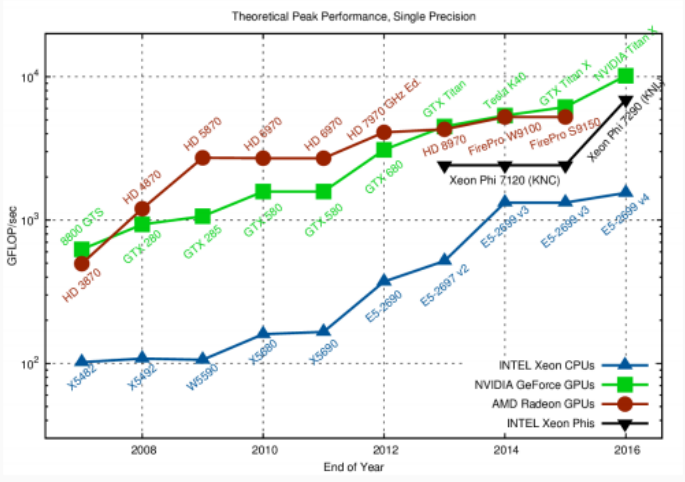
- Applikationen müssen nebenläufig programmiert werden um CPU auszunutzen → Many-Core-Systeme
- CPU-Begrenzung von Applikationen
- Effizienz und Performanceoptimierung werden immer wichtiger
- Unterstützung von Nebenläufigkeit/Parallelität durch Programmiersprachen

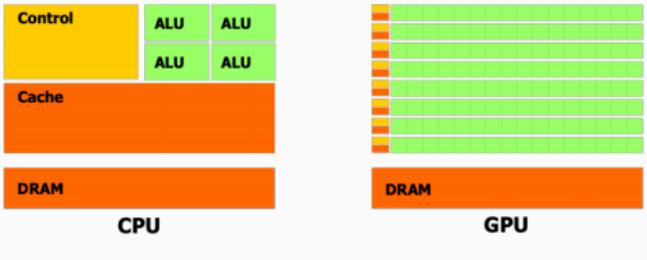
Symmetrisch vs. Nicht-symmetrisch

SMP (Symmetric Multi Processing)	NUMA (Non-Uniform Memory-Access)
Speicherbandbreite begrenzt und von allen Prozessoren gemeinsam genutzt	jedem Prozessor sind Teile des Speichers zugeordnet
Skalierbarkeit begrenzt	lokaler Zugriff ist schneller als entfernter
Single Socket-Lösung	Mehr-Socket-Board

CPU vs. GPU

- GPU = Graphics Processing Units
- Hochparallele Prozessorarchitekturen (nicht nur) für Grafikrendering





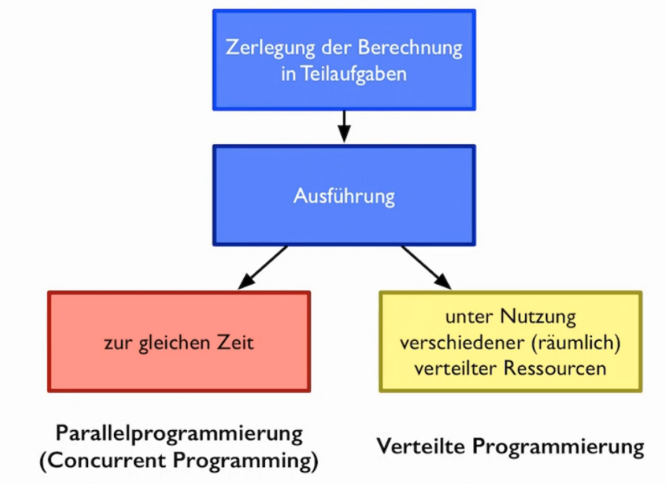


Abbildung 1: Einordnung

Abbildung 2: Architekturen: SIMD, SMP, NUMA, Cluster, Grid

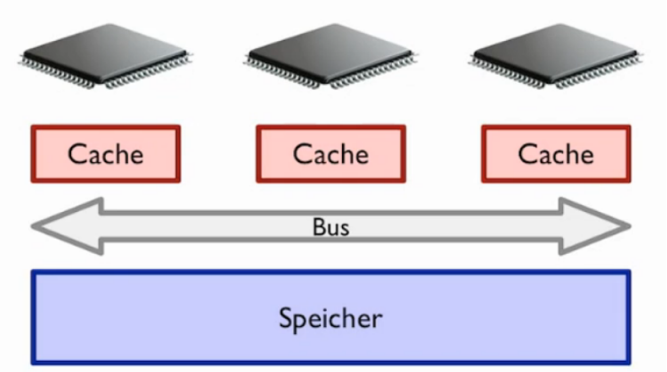


Abbildung 3: Multiprozessorsysteme

- Zugriff über Bus auf gemeinsamen Speicher
- jeder Prozessor mit eigenen Caches

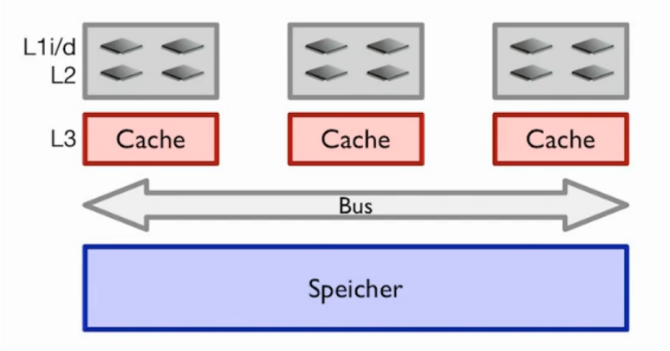


Abbildung 4: Multicore-Systeme

- mehrere Prozessorkerne auf einem Chip
- Kerne typischerweise mit eigenen L1/L2-Caches und gemeinsamen L3-Cache

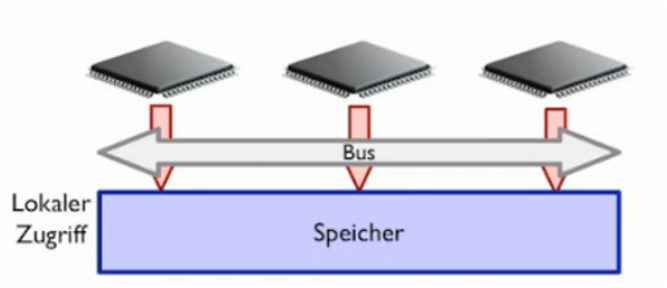


Abbildung 5: SMP (Symmetric Multi Processing)

- Speicherbandbreite begrenzt und von allen Prozessoren gemeinsam genutzt
- Skalierbarkeit begrenzt
- Single Socket Lösung

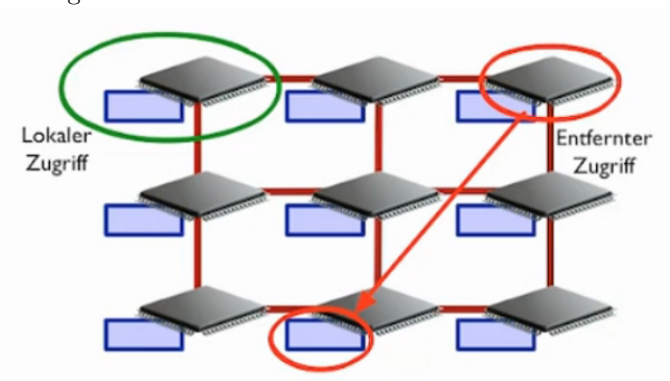
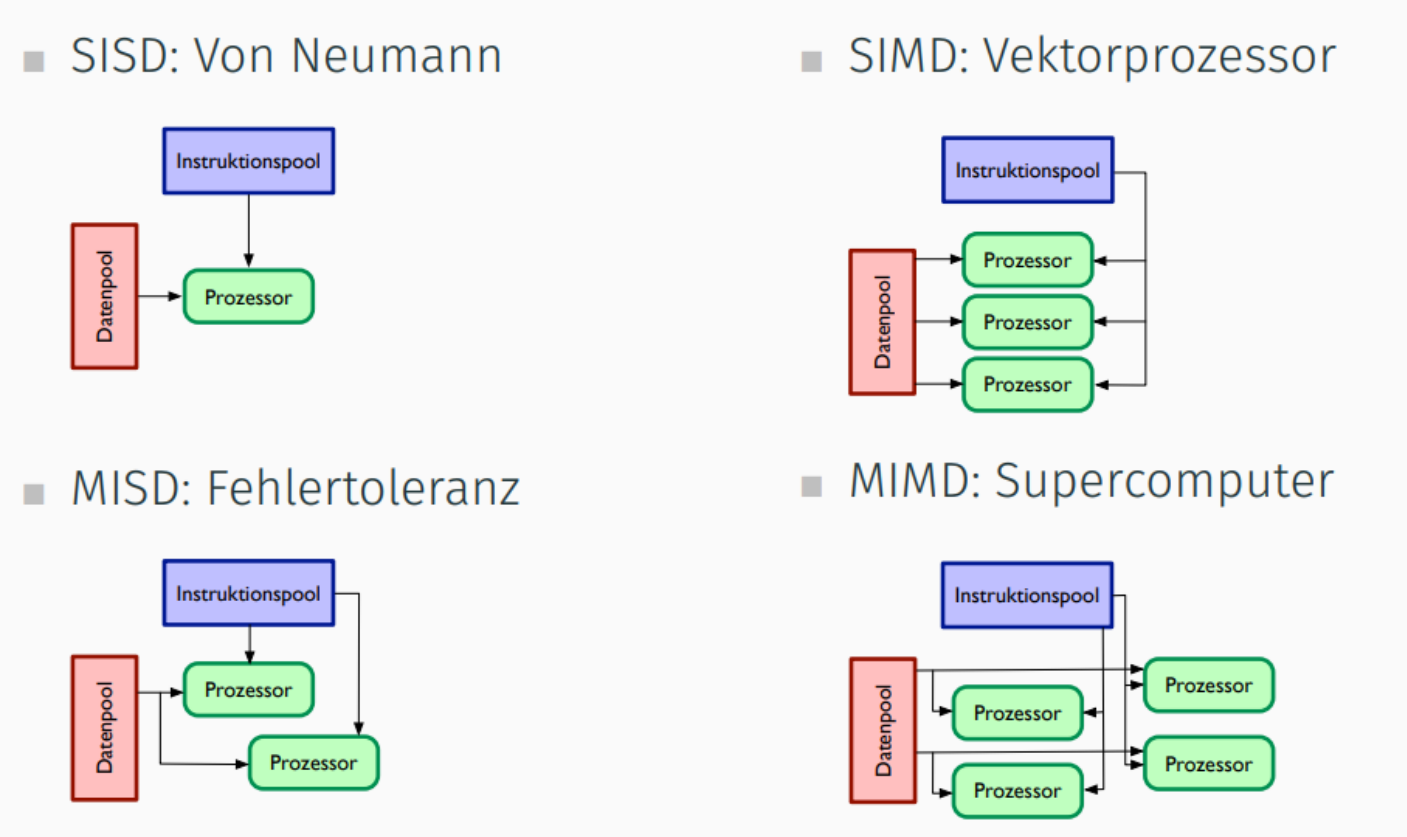


Abbildung 6: NUMA (Non-Uniform Memory Access)

- jedem Prozessor sind Teile des Speichers zugeordnet
- lokaler Zugriff ist schneller als entfernter
- Typisch für Multi-Socket Systeme

Flynn's Architekturklassifikation

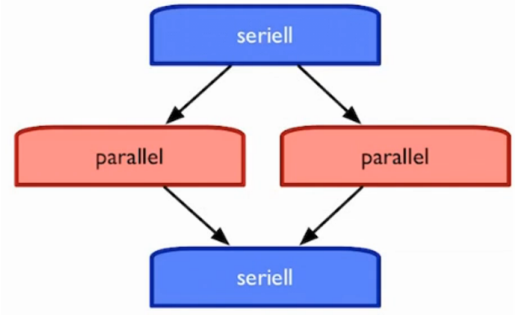


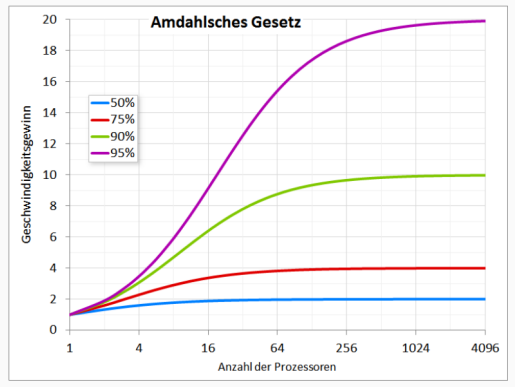
Maße zur Leistungsbewertung

- Maße für Laufzeitgewinn durch Parallelisierung
- T_n = Laufzeit des Programms mit n Prozessoren/Kernen
- Speedup $Speedup = \frac{T_1}{T_n}$
- Effizienz $Effizienz = \frac{Speedup}{n}$

Amdahlsches Gesetz

- Berücksichtigung parallelisierbarer und serieller Anteile im Programmablauf
- p = paralleler Anteil
- s = serieller Anteil
- n Prozessoren
- $p + s = 1$
- Maximaler Speedup $Speedup_{max} = \frac{T_1}{T_n} = \frac{s+p}{s+\frac{p}{n}} = \frac{1}{s+\frac{p}{n}}$





Prozesse und Threads

Prozess := Programm in Ausführung; Ausführungsumgebung für ein Programm

- hat eigenen Adressraum
- Prozessor kann immer nur einen Prozess ausführen

Thread ("Faden") := leichtgewichtige Ausführungsreinheit oder Kontrollfluss (Folge von Anweisungen) innerhalb eines sich in Ausführung befindlichen Programms

- leichtgewichtig im Vergleich zu Betriebssystemprozess
- Threads eines Prozesses teilen sich den Adressraum
- Thread kann von einer CPU oder einem Core ausgeführt werden

Shared Memory vs Message Passing

Art der Kommunikation zwischen Prozessen oder Threads

Shared Memory

- Kommunikation (über Variable im) gemeinsamen Speicher

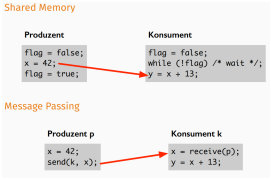


Abbildung 7: Shared Memory vs Message Passing

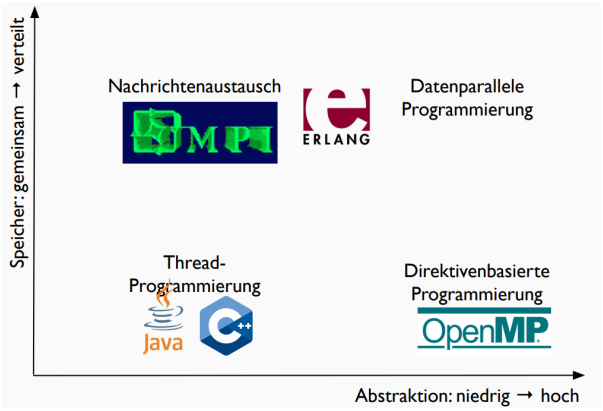


Abbildung 8: Programmiermodelle

```
#include <immintrin.h>

__m256 first = _mm256_setr_ps(10.0, 11.0,
              12.0, 13.0, 14.0, 15.0,
              16.0, 17.0);
__m256 second = _mm256_setr_ps(5.1, 5.1, 5.1,
               5.1, 5.1, 5.1, 5.1, 5.1);
__m256 result = _mm256_add_ps(first, second);
```

Abbildung 9: Instruktionsparallelität: SIMD

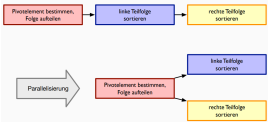


Abbildung 10: Taskparallelität

- Autovektorisierung durch Compiler
- explizite Instruktionen
- Beispiel: Addition zweier Vektoren

- Unabhängigkeit von Teilprozessen → Desequentialisierung
- Beispiel: Quicksort

```
for (i=0; i < Z; i++)
  for (j=0; j < S; j++)
    S[i][j] = A[i][j]+B[i][j];
```

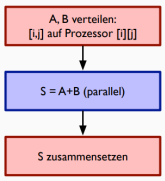


Abbildung 11: Datenparallelität

- homogene Datenmenge: Felder, Listen, Dokumentenmenge,...
- Verteilung der Daten
- alle Prozessoren führen gleiches Programm auf jeweils eigenen Daten aus
- Beispiel: Matrixaddition S = A + B

- Prozess kann direkt auf Speicher eines anderen Prozesses zugreifen
- erfordert explizite Synchronisation, z.B. über zeitkritische Abschnitte

Message Passing

- Prozesse mit getrennten Adressräumen; Zugriff nur auf eigenen Speicher
- Kommunikation durch explizites Senden/Empfangen von Nachrichten

Parallelisierungsarten

Instruktionsparallelität:

parallele Ausführung mehrerer Operationen durch eine CPU-Instruktion
explizit Vektorinstruktionen, SIMD
implizit Pipelining von Instruktionen
Taskparallelität Ausnutzung inhärenter Parallelität durch simultane Ausführung unabhängiger Aufgaben
Datenparalelität

- Gemeinsame Operation auf homogener Datenmenge
- Zerlegung eines Datensatzes in kleinere Abschnitte

Herausforderungen

- Zerlegung eines Problems in parallel verarbeitbare Teile
  - Beispiel: Suche in einer Datenbank mit 1 TB Größe
  - Annahme: 100 MB/Sekunde mit einem Prozessor = 175 Minuten
  - bei paralleler Suche durch 10 Prozessoren = 17.5 Minuten
  - Übertragbar auf andere Probleme, z.B. Sortieren, Suche in Graphen?
- Synchronisation konkurrierender Zugriffe auf gemeinsame Ressourcen

```
1> spawn(fun() -> io:format(["Hallo Erlang!"]) end).
Hallo Erlang!<0.133.0>
```

```
-module(ch4_1).

-export([start/0, say_hello/1]).

say_hello(Msg) ->
  io:format("~n~p", [Msg]).

start() ->
  spawn(ch4_1, say_hello, ["Hallo Erlang!"]).
```

Abbildung 12: Beispiele

```
Erlang/OTP 23 [erts-11.0] [source] [64-bit] [smp:4:4]

Eshell V11.0  (abort with ^G)
1> erlang:system_info(schedulers).
4
```

Abbildung 13: SMP Erlang

  - Beispiel: Produzent-Konsument-Beziehung
  - Annahme: Datenaustausch über gemeinsame Liste
  - Fragestellungen: Benachrichtigung über neues Element in der Liste, Konsument entnimmt Element während Produzent einfügt
  - Wechselseitiger Ausschluss
- außerdem: Fehlersuche, Optimierung

Zusammenfassung

- Parallele Verarbeitung als wichtiges Paradigma moderner Software
- verschiedene parallele
  - Hardwarearchitekturen und
  - Programmiermodelle
- Herausforderungen
  - Problemzerlegung
  - Synchronisation
  - ...
- im Weiteren: konkrete Methoden und Techniken in Erlang und C++

Parallele Programmierung in Erlang

Unterstützung paralleler Programmierung in Erlang

- Leichtgewichtige Prozesse und Message Passing
- SMP-Support (Symmetric Multi Processing)
- Ziele für effiziente Parallelisierung
  - Problem in viele Prozesse zerlegen (aber nicht zu viele ...)
  - Seiteneffekte vermeiden (würde Synchronisation erfordern ...)
  - Sequentiellen Flaschenhals vermeiden (Zugriff auf gemeinsame Ressourcen: IO, Registrierung von Prozessen, ...) -¿ Small Messages, Big Computation!

Prozesse in Erlang

- Erlang VM = Betriebssystemprozess
- Erlang-Prozess = Thread innerhalb der Erlang VM
  - kein Zugriff auf gemeinsame Daten, daher "Prozess"
- jede Erlang-Funktion kann einen Prozess bilden
- Funktion spawn erzeugt einen Prozess, der die Funktion Fun ausführt

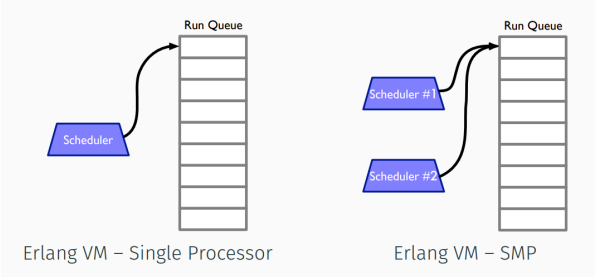
```
    Pid = spawn(fun Fun/0)
```

- Resultat = Prozessidentifikation Pid, mittels der man dem Prozess Nachrichten schicken kann.
- über self() kann man die eigene Pid ermitteln
- Übergabe von Arghumenten an den Prozess bei der Erzeugung

```
    Pid = spawn(fun() $\rightarrow$ any_func(Arg1, Arg2, ...) end)
```

- 4 Betriebssystemthreads (hier 2 Kerne mit Hyperthreading)
- kann mit -smp [disable | enable | auto] beeinflusst werden
- +S [Anzahl] bestimmt Anzahl der Scheduler
  - sollte nicht größer als Anzahl der Kerne/Prozessoren sein

Scheduler in Erlang



Message Passing in Erlang: Senden einer Nachricht

```
Pid ! Message
```

- an Prozess Pid wird die Nachricht Message gesendet
- der Prozess muss eine Empfangsoperation ausführen. damit ihn die Nachricht erreichen kann

```
receive
    Pattern1 [when Guard1] -> Expressions1;
    Pattern2 [when Guard2] -> Expressions2;
    ...
end
```

- trifft eine Nachricht ein, wird versucht, diese mit einem Pattern und ggf. vorhandenen Guard zu "matchen"
- erstes zutreffendes Pattern (inkl. Guard) bestimmt, welcher Ausdruck ausgewertet wird
- trifft kein Pattern zu, wird die Nachricht für spätere Verwendung aufgehoben und Prozess wartet auf die nächste Nachricht (→ selective receive)

Ein einfacher Echo-Server

```
1  -module(ch4_2).
2  -export([run/0]).
3
4  run() -> Pid2 = spawn(fun loop/0),
5    Pid2 ! {self(), hello},
6    receive
7      {Pid2, Msg} -> io:format("P1 ~w~n",[Msg])
8    end,
9    Pid2 ! stop.
10
11 loop() ->
12   receive
13     {From, Msg} -> From ! {self(), Msg}, loop();
14     stop -> true
15   end.
```

Erklärungen

- Funktion loop() realisiert einen (nur bedingt nützlichen) Echo-Dienst, der jede empfangene Nachricht unverändert an den Absender zurückschickt, bis er nach Empfang von stop endet
- Funktion run()
  1. startet den Echoserver (Zeile 4)
  2. schickt ihm als nächstes eine Nachricht (Zeile 5)
  3. wartet auf eine Antwort (Zeile 6)
  4. gibt diese aus (Zeile 7)
  5. schickt dann stop an den Echoserver (Zeile 9)
- Aufruf in der Funktion loop() erfolgt endrekursiv, daher wird kein wachsender Aufrufstapel angelegt (Hinweis: grundsätzlich zu beachten, da sonst der Speicherbedarf stetig wächst)

Ansätze zur Parallelisierung

- Beispiel: Berechnung einer (zufällig generierten) Liste von Fibonaccizahlen
- Sequentielle Lösung über lists:map/2

```
% Berechnung der Fibonacci-Zahl für F
fibo(0) -> 0;
fibo(1) -> 1;
fibo(F) when F > 0 -> fibo(F - 1) + fibo(F - 2).

% Liste von Num Fibonacci-Zahlen
run(Num) ->
  Seq = lists:seq(1, Num),   % Zufallszahlen erzeugen
  Data = lists:map(fun(_) -> random:uniform(20) end, Seq),
  lists:map(fun fibo/1, Data).
```

pmap: Parallele Funktionen höherer Ordnung

- Parallele Variante von lists:map(Fun, list)
- für jedes Listenelement einen Prozess erzeugen
- Ergebnisse einsammeln

```
pmap(F, L) ->
   S = self(), % Berechnung der Fibonacci-Zahl für F
   Pids = lists:map(fun(I) ->
           % Prozess erzeugen
           spawn(fun() -> do_fun(S, F, I) end)
         end, L),    % Ergebnisse einsammeln
         gather(Pids).
```

pmap: Hilfsfunktionen

Eigentliche Verarbeitungsfunktion ausführen

```
do_fun(Parent, F, I) ->
   % Parent ist der Elternprozess
   Parent ! { self(), (catch F(I))}.
```

- Funktion F aufrufen, catch sorgt für korrekte Behandlung von Fehlern in F
- Ergebnis zusammen mit eigener Pid (self()) an Elternprozess senden

Einsammeln der Ergebnisse

```
1  % rekursive Implementierung
2  gather([Pid | T]) ->
3    receive
4    % Ordnung der Ergeb. entspricht Ordnung der Argumente
5      { Pid, Ret } -> [Ret | gather(T)]
6    end;
7  gather([]) -> [].
```

- Zeile 5: Warten bis Paar (Pid, Ergebniswert) eintrifft
- Zeile 7: Tail ist leer → alle Ergebnisse eingetroffen

Parallele Berechnung der Fibonacci-Zahlen

```
1  % Liste von Num Fibonacci-Zahlen
2  run(Num) ->
3    Seq = lists:seq(1, Num),   % Zufallszahlen erzeugen
4    Data = lists:map(fun(_) ->
5                       random:uniform(20) end, Seq),
6    % Berechnung parallel ausführen
7    pmap(fun fibo/1, Data).
```

Diskussion

- Passende Abstraktion wählen
  - Ist Ordnung der Ergebnisse notwendig?
  - Werden Ergebnisse benötigt?
- Anzahl der parallelen Prozesse
  - Abhängig von Berechnungsmodell, Hardware etc.
  - evtl. pmap mit max. Anzahl gleichzeitiger Prozesse
- Berechnungsaufwand der Prozesse
  - Berechnung vs. Daten/Ergebnisse senden

pmap: Alternative Implementierung

- ohne Berücksichtigung der Ordnung der Ergebnismenge
- Zählen für die bereits eingetroffenen Ergebnisse

```
1  pmap(F, L) ->
2    ...
3    gather2(length(L), Ref, []).
4
5  gather2(N, Ref, L) ->
6    receive
7      {Ref, Ret } -> gather2(N-1, Ref, [Ret | L])
8    end;
9  gather2(0,_, L) -> L.
```

Speedup

Bestimmung des Speedups erfordert

- Zeitmessung
- Kontrolle der genutzten Prozessoren/Cores

Welchen Einfluss hat die Zahl der erzeugten Prozesse.

Speedup: Zeitmessung

Nutzung der Funktion timer:tc/3

```
1> timer:tc(ch4_4, run, [30]).
{7900,[233,1,987,610,377,8,144,89,89,3]}
```

Für bessere Aussagekraft: mehrfache Ausführung

```
benchmark(M, Fun, D) ->
  % 100 Funktionsaufrufe
  Runs = [timer:tc(M, Fun, [D]) || _ <- lists:seq(1, 100)],
  % Durchschnitt der Laufzeiten in Millisekunden berechnen
  lists:sum([T || { T, _ } <- Runs]) / (1000 * length(Runs)).
```

Bestimmung: Speedup

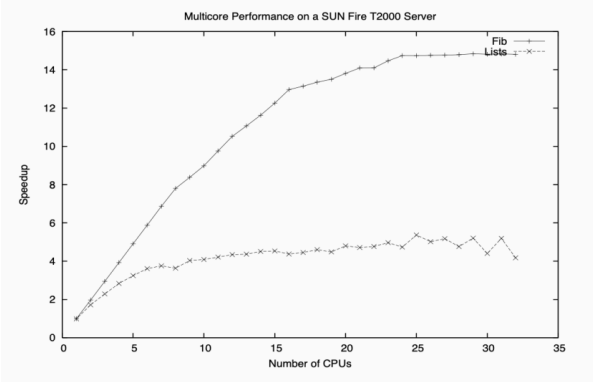
ch4_6:benchmark(ch4_4, run, 1000).

Anzahl Threads	Laufzeit (msecs.)
1	108
2	55
4	47
Sequentiell	108

Achtung:

- Aufwand für Berechnung einer Fibonaccizahl ist nicht konstant
- Zufallszahlen als Eingabe

Diskussion: Speedup



Datenparallelität: Das Map-Reduce-Paradigma

- Parallelisierungsmuster inspiriert von Konzepten funktionaler Programmiersprachen (map,reduce/fold)
- Basis von Big-Data-plattformen wie Hadoop, Spark,...
- Grundidee:
  - map(F, Seq) ? wende Funktion F (als Argument übergeben) auf alle Elemente einer Folge Seq an,
    - Funktion F kann unabhängig (=parallel) auf jedes Element angewendet werden
    - Partitionieren und Verteilen der Elemente der Folge
    - z.B. multipliziere jedes Element mit 2
  - reduce(F, Seq) = wende eine Funktion F schrittweise auf die Elemente einer Folge Seq an und produziere einen einzelnen Wert,
    - prinzipiell ähnlich zu map(F, Seq), d.h. Funktion F kann auf Paare unabhängig angewendet werden
    - z.B. die Summe aller Elemente der Folge

map in Erlang

Definition von `map` (auch als `lists:map/2`)

```
1  map(_, []) -> [];
2  map(F, [H|T]) -> [F(H)|map(F,T)].
```

Definition der Multiplikation

```
1  mult(X) -> X * 2.
```

Anwendung

```
1  1> S=[1,2,3,4].
2  [1,2,3,4]
3  2> mr:map(fun mr:mult/1, S).
4  [2,4,6,8]
```

reduce in Erlang

Definition von `reduce` (auch als `lists:foldl/3` bzw. `lists:foldr/3`)

```
1  reduce(_, Init, []) -> Init;
2  reduce(F, Init, [H|T]) -> reduce(F, F(H,Init), T).
```

Definition der Addition

```
1  add(X, Y) -> X + Y.
```

Anwendung

```
1  1> S=[1,2,3,4].
2  [1,2,3,4]
3  2> mr:reduce(fun mr:add/2, 0, S).
4  10
```

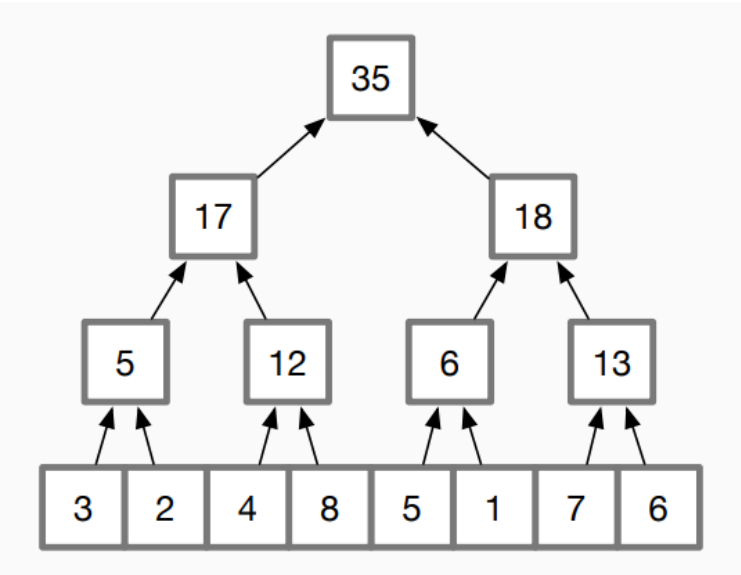
Parallelisierung von map und reduce map

- Funktion F kann unabhängig (=parallel) auf jedes Element angewendet werden
- Partitionieren und Verteilen der Elemente der Folge
- siehe pmap

reduce

- prinzipiell ähnlich, d.h. Funktion F kann auf Paare unabhängig angewandt werden

Parallelisierung von reduce



Taskparallelität: Sortieren

Quicksort in Erlang

```
1 qsort([]) -> [];
2 qsort([H|T]) -> qsort([Y || Y <- T, Y < H]) ++
3         [H] ++
4         qsort([Y || Y <- T, Y >= H]).
```

- typische funktionale Notation von Quicksort mit List Comprehensions
- Zeile 2: H dient als Pivotelement

Idee:

- Prozess für das Sortieren der einen Hälfte starten
- Elternprozess kann andere Hälfte sortieren
- rekursive Zerlegung...

Parallel Quicksort

Version 1 Quicksort in Erlang

```
1 qsort2([]) -> [];
2 qsort2([H|T]) ->
3   Parent = self(),
4   spawn(fun() ->
5     Parent ! qsort2([X || X <- T, X >= H]) end),
6   qsort2([Y || Y <- T, Y < H]) ++
7         [H] ++
8         receive T2 -> T2 end.
```

Erläuterungen

- Zeile 4: Erzeugen eines neuen Prozesses zur Sortierung der öberen"Hälfte
- Zeile 6-7: Wie bisher
- Zeile 8: Warten auf Empfang der sortierten anderen Hälfte

Zeitmessung:

```
1 1> L = ch4_6:rand_list(100000).
2 ...
3 2> ch4_6:benchmark(ch4_10, qsort, L).
4 131.90963
5 3> ch4_6:benchmark(ch4_10, qsort2, L).
6 293.59211
```

Bewertung

- parallele Version 1 ist langsamer!
- mögliche Erklärung: Prozess-Start ist aufwändiger als Sortieren kleiner Teilfolgen
- bessere Variante nach John Hughes: Parallel Programming in Erlang
  - Kontrolle der Granularität für parallele Ausführungen
  - danach Sortieren mit sequenzieller Variante
  - einfache Idee: Anzahl der parallelen Zerlegung begrenzen

```
1 qsort3(L) -> qsort3(4, L). % 4 Rekursionsstufen parallel
2
3 qsort3(0, L) -> qsort(L); % Umschalten
4 ...
```

Version 2

```
qsort3(L) -> qsort3(6, L).

qsort3(0, L) -> qsort(L);
qsort3(_,[]) -> [];
qsort3(N, [H|T]) ->
  Parent = self(), spawn_link(fun() ->
    Parent ! qsort3(N-1, [Y || Y <- T, Y >= H])
  end),
qsort3(N-1, [Y || Y <- T, Y < H]) ++
      [H] ++
      receive T2 -> T2 end.
```

```
4> ch4_6:benchmark(ch4_10, qsort3, L).
87.54315
```

Fazit

- leichtgewichtige Prozesse als Baustein der Parallelisierung in Erlang
- Prozesskommunikation ausschließlich über Message Passing
- funktionaler Charakter (u.a. Vermeidung von Seiteneffekten) vereinfacht Parallelisierung deutlich
- Daten- und Taskparallelität möglich
- hoher Abstraktionsgrad, aber auch wenig Einflussmöglichkeiten

Parallele Programmierung in C++

Threads in C++

Thread (Faden) = leichtgewichtige Ausführungseinheit oder Kontrollfluss (Folge von Anweisungen) innerhalb eines sich in Ausführung befindlichen Programms

- Threads teilen sich den Adressraum ihres Prozesses
- in C++: Instanzen der Klasse std::thread
- führen eine (initiale) Funktion aus

```
#include <thread>
#include <iostream>

void say_hello() {
  std::cout << "Hello Concurrent C++\n";
}

int main() {
  std::thread t(say_hello);
  t.join();
}
```

Alternative Erzeugung von Threads über Lambda-Ausdruck

```
std::thread t([]() { do_something(); });
```

mit Instanzen einer Klasse - erfordert Überladen von operator()

```
struct my_task {
  void operator()() const { do_something(); }
};

my_task tsk;
std::thread t1(tsk); // mit Objekt
std::thread t2{ my_task() }; // über Konstruktor
```

Parameterübergabe bei Threaderzeugung

- über zusätzliche Argumente des thread-Konstruktors
- Vorsicht bei Übergabe von Referenzen, wenn Elternthread vor dem erzeugten Thread beendet wird

```
void fun(int n, const std::string\& s) {
    for (auto i = 0; i < n; i++)
        std::cout << s << " ";
    std::cout << std::endl;
}
std::thread t(fun, 2, "Hello");
t.join();
```

Warten auf Threads

- t.join() wartet auf Beendigung des Threads t
- blockiert aktuellen Thread
- ohne join() keine Garantie, dass t zur Ausführung kommt
- Freigabe der Ressourcen des Threads

```
std::thread t([]() { do_something(); });
t.join();
```

Erscheint die Ausgabe?

```
#include <iostream>
#include <thread>
#include <chrono>

int main() {
    std::thread t([]() {
        std::this_thread::sleep_for(
            std::chrono::seconds(1));
        std::cout << "Hello\n";
    });
}
```

Hintergrundthreads

- Threads können auch im Hintergrund laufen, ohne, dass auf Ende gewartet werden muss
- äbkoppeln"durch detach()
- Thread läuft danach unter Kontrolle des C++-Laufzeitsystems, join nicht mehr möglich

Threadidentifikation

- Threadidentifikator vom Typ std::thread::id
- Ermittlung über Methode get_id()

```
void fun() {
std::cout << "Hello from "
          << std::this_thread::get_id()
          << std::endl;
}
std::thread t(fun);
t.join();

```

Beispiel: Berechnung von Fibonacci-Zahlen in C++

- rekursive und nichtrekursive Variante möglich

```
unsigned int fibonacci(unsigned int n) {
  if (n == 0)
    return 0;

  unsigned int f0 = 0, f1 = 1, f2;
  for (auto i = 1u; i < n; i++) {
     f2 = f0 + f1;
     f0 = f1;
     f1 = f2;
  }
  return f1;
}
```

Parallele Berechnung von Fibonacci-Zahlen

- einfachste Lösung (ähnlich zu Erlang): pro Zahl ein Thread

```
std::vector<std::thread> threads;
unsigned int results[20];

for (auto i = 0u; i < 20; i++) {
    auto f = rand() % 30;
    threads.push_back(std::thread([&]() {
        results[i] = fibonacci(f);
    }));
}
std::for_each(threads.begin(), threads.end(),
    std::mem_fn(&std::thread::join));
```

Erläuterungen

- Zeile 1: Feld der Threads
- Zeile 2: Feld für Ergebniswerte
- Zeile 5: Zufallszahl erzeugen
- Zeilen 6-7 Thread zur Berechnung der Fibonacci-Zahl erzeugen und Ergebnis im Feld speichern
- Zeile 10-11: Warten auf Beendigung der Threads (std::mem_fn = Wrapper für Zeiger auf Member-Funktion)
- aber:
  - Zugriff auf gemeinsame Ressource (results)!
  - Anzahl Fibonaccizahlen = Anzahl Threads

parallel-for in C++

- Unterstützung durch Higher-Level-APIs und Frameworks

OpenMP

```
#pragma omp parallel for
   for (auto i = 0u; i < n; i++) { ... }
```

Parallele Algorithmen in C++17

```
std::for_each(std::execution::par_unseq,
    vec.begin(), vec.end(), [](auto&& item) { ... });
```

Intel TBB

```
tbb::parallel_for(tbb::blocked_range<int>(0,vec.size()),
            [&](tbb::blocked_range<int> r) { ... }
```

Kontrolle der Anzahl der Threads

- Erzeugung von Threads ist mit Kosten verbunden
- begrenzte Anzahl von Hardwarethreads (Anzahl Cores, Hyperthreading)
- Ermittlung der Anzahl der unterstützten Hardwarethreads

```
std::thread::hardware_concurrency()
```

- Nutzung für Implementierung von Threadpools, Task Libraries, ...

Probleme nebenläufiger Ausführung

```
struct jawsmith {
    std::string msg;

    jawsmith(const std::string& m) : msg(m) {}
    void operator()() const {
        for(;;) {
            std::this_thread::sleep_for(
            std::chrono::seconds(1));
            for (auto& c : msg) {
                std::cout << c << std::flush;
            }
        }
    }
};
```

```
std::thread t1 { jawsmith("DASISTEINELANGENACHRICHT") };
std::thread t2 { jawsmith("dieistaberauchnichtkurz") };
```

Ausgabe:

```
dDieistaberauchnichtkASISTEINELANGENACHurzRICHT...
```

Race Conditions (Wettlaufsituationen) := Ergebnis nebenläufiger Ausführung auf gemeinsamen Zustand (hier: Ausgabekanal) hängt vom zeitlichen Verhalten der Einzeloperationen ab

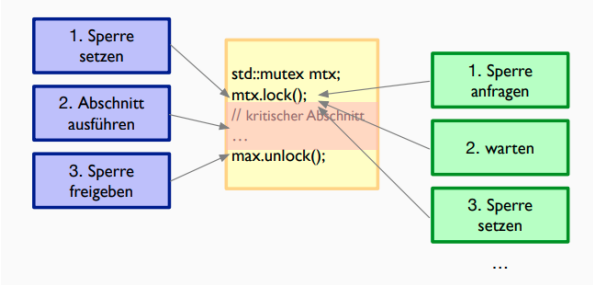
- Race Conditions (Wettlaufsituation) := Ergebnis nebenläufiger Ausführung auf gemeinsamen Zustand (hier: Ausgabekanal) hängt vom zeitlichen Verhalten der Einzeloperationen ab
- kritischer Abschnitt: Programmabschnitt in einem Thread, in dem auf eine gemeinsame Ressource (Speicher etc.) zugegriffen wird und der nicht parallel (oder zeitlich verzahnt) zu einem anderen Thread ausgeführt werden darf
- Lösung durch wechselseitigen Ausschluss (engl. mutual exclusion = mutex)
  - Instanz der Klasse std::mutex
  - Methoden zum Sperren (lock) und Freigeben (unlock)
  - 'mutex' : Standard-Mutex für exklusiven Zugriff
  - 'timed_mutex' : Mutex mit Timeout für Warten (try_lock_for())
  - 'recursive_mutex' : rekursives Mutex - erlaubt mehrfaches Sperren durch einen Thread, z.B. für rekursive Aufrufe
  - 'recursive_timed_mutex' : rekursives Mutex mit Timeout
  - 'shared_mutex' : Mutex, das gemeinsamen Zugriff (lock_shared()) mehrerer Threads oder exklusiven Zugriff (lock()) ermöglicht
  - 'shared_timed_mutex' : Mutex mit Timeout und gemeinsamen Zugriff
- Lock Guards
  - Vereinfachung der Nutzung von Mutexen durch RAII ("Ressourcenbelegung ist Initialisierung")
  - Konstruktor = lock
  - Destruktor = unlock
  - std::unique_lock erweiterte Variante von lock_guard, vermeidet aber sofortiges Sperren
  - std::lock : erlaubt gleichzeitiges deadlock-freies Sperren von 2 Mutexen
  - Sperrstrategien: u.a.
    - std::try_to_lock versucht Sperre ohne Blockierung zu setzen
    - std::adopt_lock versucht nicht, ein zweites Mal zu sperren, wenn bereits durch den aktuellen Thread gesperrt

Wechselseitiger Ausschluss

- kritischer Abschnitt: Programmabschnitt in einem Thread, in dem auf eine gemeinsame Ressource (Speicher etc.) zugegriffen wird und der nicht parallel (oder zeitlich verzahnt) zu einem anderen Thread ausgeführt werden darf
- Lösung durch wechselseitigen Ausschluss (engl. mutual exclusion = mutex)

Mutex in C++

- Instanz der Klasse std::mutex
- Methoden zum Sperren (lock) und Freigeben (unlock)



Mutex-Varianten

- mutex: Standard-Mutex für exklusiven Zugriff
- timed_mutex: Mutex mit Timeout für Warten (try_lock_for())
- recursive_mutex:rekursives Mutex - erlaubt mehrfaches Sperren durch einen Thread, z.B. für rekursive Aufrufe
- recursive_timed_mutex: rekursives Mutex mit Timeout
- shared_mutex: Mutex, das gemeinsamen Zugriff (lock_shared()) mehrerer Threads oder exklusiven Zugriff (lock()) ermöglicht
- shared_timed_mutex: Mutex mit Timeout und gemeinsamen Zugriff

Lock Guards

- Vereinfachung der Nutzung von Mutexen durch RAII ("Ressourcenbelegung ist Initialisierung")
- Konstruktor = lock
- Destruktor = unlock

```
1  std::vector<int> data;
2  std::mutex my_mtx;
3
4  void add(int val) {
5    std::lock_guard<std::mutex> guard(my_mtx);
6    data.push_back(val);
7  }
8
9  int get() {
10   std::lock_guard<std::mutex> guard(my_mtx);
11   return data.front();
12 }
13
```

Lock Gurads und Locks

- std::unique_lock erweiterte Variante von lock_guards, vermeidet aber sofortiges Sperren
- std::lock erlaubt gleichzeitiges deadlock-freies Sperren von 2 Mutexen
- Sperrstrategien: u.a.
  - std::try_to_lock versucht Sperre ohne Blockierung zu setzen
  - std::adopt_lock versucht nicht, ein zweites Mal zu sperren, wenn bereits durch den aktuellen Thread gesperrt

Atomare Datentypen

- std::atomic_flag = sperrfreier, atomarer Datentyp:
  - clear() setzt den Wert auf false
  - test_and_set() setzt den Wert atomar auf true und liefert den vorherigen Wert
- std::atomic¡bool¿ = mächtigere Variante, erlaubt explizites Setzen
  - operator= atomare Wertzuweisung
  - load() liefert den aktuellen Wert
  - read-modify-write-Operation (siehe später)
- std::atomic¡T¿ = generische Variante für weitere Datentypen

Synchronisation über atomare Variable

```
1  std::list<std::string> shared_space;
2  std::atomic<bool> ready{false};
3
4  void consume() {
5      while (!ready.load())
6         std::this_thread::sleep_for(
7              std::chrono::milliseconds(10));
8      std::cout << shared_space.front() << std::endl;
9      shared_space.pop_front();
10 }
11
12 void produce() {
13     shared_space.push_back("Hallo!");
14     ready = true;
15 }
16
17 std::thread t1(consumer);
18 std::thread t2(producer);
19 ...
```

Erläuterungen

- Zeile 1: gemeinsam genutzte Liste - erfordert synchronisierten Zugriff
- Zeile 2: atomare boolsche Variable ready
- Zeile 4/12: Konsument/Produzent-Threads
- Zeile 5: atomares prüfen der ready-Variablen
- Zeile 6-7 kurz warten und neu versuchen
- Zeile 8-9/13 Zugriff auf gemeinsame Liste
- Zeile 14: atomares Setzen der Variablen ready

Taskparallelität: Die 5 speisenden Philosophen

- fünf Philosophen teilen sich eine Schüssel Sphagetti
- fünf Gabeln, je eine zwischen zwei Philosophen
- Philosoph kann nur mit zwei benachbarten Gabeln essen
- Gabeln werden nur nach dem Essen zurückgelegt
- Philosoph durchläuft Zyklus von Zuständen: denken → hungrig → essen → denken → etc.

Das Problem mit den Philosophen

- Jeder greift die linke Gabel
- und wartet auf die rechte Gabel
- ... und wartet ...

Verklemmung!

Lösungsidee

- immer beide Gabeln aufnehmen, dh. wenn nur eine Gabel verfügbar ist: liegen lassen und warten
- synchronisierter Zugriff auf Gablen, dh. in einem kritischen Abschnitt unter gegenseitige Ausschluss
- Wecken von wartenden Philosophen

Verklemmungsfreies Sperren

```
  std::lock(mtx1, mtx2);
  std::lock_guard<std::mutex> lk1(mtx1, std::adopt_lock);
  std::lock_guard<std::mutex> lk2(mtx2, std::adopt_lock);

```

Führt zu Verklemmung; Alternative Lösung

```
  std::unique_lock<std::mutex> lk1(mtx1, std::defer_lock);
  std::unique_lock<std::mutex> lk2(mtx2, std::defer_lock);
  std::lock(lk1, lk2);

```

Gabeln und Spaghetti-Teller

```
class philosopher {
private:
   int               id;
   spaghetti_plate&  plate;
   fork&             left_fork;
   fork&             right_fork;
public:
   philosopher(int n, spaghetti_plate& p) :
      id(n), plate(p),
      left_fork(p.forks[n]), // 1. Gabel
      right_fork(p.forks[(n + 1) % 5]) // 2. Gabel
      {}
   ...
};
```

Die Philosophen-Klasse

```
// Gabel = Mutex
struct fork {
   std::mutex mtx;
};

// 1 Teller mit 5 Gabeln
struct spaghetti_plate {
   // Benachrichtigung der Philosophen
   std::atomic<bool> ready{false};
   std::array<fork, 5> forks;
};
```

Die Philosophen-Klasse: Hilfsmethoden

Textausgabe erfordert synchronisierten Zugriff auf cout über globalen Mutex

```cpp
void say(const std::string& txt) {
    std::lock_guard<std::mutex> lock(out_mtx);
    std::cout << "Philosopher #" << id
              << txt << std::endl;
}
```

Hilfsmethode für zufällige Wartezeit in Millisekunden

```cpp
std::chrono::milliseconds wait() {
    return std::chrono::milliseconds(rand() % 500 + 100);
}
```

Die Philosophen-Klasse: Essen

```cpp
void eating() {
    // Versuche, die Gabeln (verklemmungsfrei)
    // aufzunehmen
    std::lock(left_fork.mtx, right_fork.mtx);
    std::lock_guard<std::mutex>
      left_lock(left_fork.mtx, std::adopt_lock);
    std::lock_guard<std::mutex>
      right_lock(right_fork.mtx, std::adopt_lock);

    // Essen simulieren
    say(" started eating.");
    std::this_thread::sleep_for(wait());
    say(" finished eating.");
}
```

Die Philosophen-Klasse: Denken

```cpp
void thinking() {
    say(" is thinking.");
    // Wenn Philosophen denken ....
    std::this_thread::sleep_for(wait());
}
```

Das Leben eines Philosophen

- Zur Erinnerung: überladener ()-Operator eines Objekts definiert auszuführende Funktion eines Threads

```cpp
void operator()() {
    // Warten bis der Teller bereit ist
    while (!plate.ready);

    do {
        // solange der Teller bereit ist
        thinking();
        eating();
    } while (plate.ready);
   }
```

Das Dinner: Initialisierung

```cpp
// der Teller
spaghetti_plate plate;

// die 5 Philosophen
std::array<philosopher, 5> philosophers {{
    { 0, plate }, { 1, plate },
    { 2, plate }, { 3, plate },
    { 4, plate }
}};

// Thread pro Philosoph erzeugen
std::array<std::thread, 5> threads;
for (auto i = 0u; i < threads.size(); i++) {
    threads[i] = std::thread(philosophers[i]);
}
```

Das Dinner beginnt

- Beginn (und Ende) des Dinners über atomare Variable signalisieren
- Philosophen-Threads arbeiten ihre operator()()-Methode ab

```cpp
// das Essen beginnt und dauert 5 Sekunden ;-)
std::cout << "Starting dinner ..." << std::endl;
plate.ready = true;
std::this_thread::sleep_for(std::chrono::seconds(5));
// Abräumen
plate.ready = false;
// Warten auf Beendigung der Threads
std::for_each(threads.begin(), threads.end(),
    std::mem_fn(&std::thread::join));
std::cout << "Dinner finished!" << std::endl;
```

Fazit

- Philosophenproblem: klassisches Problem der Informatik zur Demonstration von Nebenläufigkeit und Verklemmung
- von Edsger W. Dijkstra formuliert
- betrachte C++ Lösung illustriert
  - Nebenläufigkeit durch Threads
  - Synchronisation über Mutexe

  - verklemmungsfreies Sperren
- moderene C++ Sprachversion vereinfacht Programmierung gegenüber Low-Level-API auf Betriebssystemebene (z.B. pthreads)

Weitere Möglichkeiten der Thread-Interaktion

- bisher:
  - Mutexe und Locks
  - atomare Variablen
- typischer Anwendungsfall: Warten auf Ereignis / Setzen eines Flags

```
1  bool ready;
2  std::mutex mtx;
3
4  // Warten ...
5  std::unique_lock<std::mutex> l(mtx);
6  while (!flag) {
7      l.unlock();
8      std::this_thread::sleep_for(std::chrono::milliseconds(200));
9      l.lock();
10 }
```

Bedingungsvariablen

- Thread wartet, bis Bedingung erfüllt ist
- Erfüllung der Bedingung wird durch anderen Thread angezeigt (notify) ⇀ Äufwecken"des wartenden Threads
- notwendig: synchronisierter Zugriff über Mutex

```
1  std::list<std::string> shared_space;
2  std::mutex mtx;
3  std::condition_variable cond;
4
5  void consume() {
6      while (true) {
7          std::unique_lock<std::mutex> l(mtx);
8          cond.wait(l, [&] {
9              return !shared_space.empty();
10         });
11         auto data = shared_space.front();
12         shared_space.pop_front();
13         l.unlock();
14         // data verarbeiten
15     }
16 }
```

```
1  void produce() {
2      // data erzeugen
3      std::lock_guard lg(mtx);
4      shared_space.push_back(data);
5      cond.notify_one();
6  }
```

Thread-sichere Datenstrukturen

- Thread-Sicherheit := eine Komponente kann gleichzeitig von verschiedenen Programmbereichen (Threads) mehrfach ausgeführt werden, ohne dass diese sich gegenseitig behindern
- verschiedene Varianten:
  - Standard-Datenstruktur + über Mutexe/Sperren synchronisierte Zugriffe
  - Integration der Sperren in die Datenstruktur
  - Sperr-frei Datenstrukturen: nicht-blockierend, Vermeidung von Sperren, z.B. durch Compare/Exchange-Operationen
- async , future und promise
- std::future - Resultat einer asynchronen Berechnung, d.h. einer Berechnung die erst noch stattfindet
- std::async() - asynchrones Starten eines Tasks

```
1      int long_calculation() { ... }
2      std::future<int> result = std::async(long_calculation);
3      // Fortsetzung der Berechnung ...
4      result.wait();
5      std::cout << result.get() << std::endl;
6
```

- std::promise - erlaubt Wert zu setzen, wenn der aktuelle Thread beendet ist; oft in Kombination mit std::future eingesetzt
- future = Ergebnisobjekt, promise = Ergebnisproduzent
  - Warten auf Ende des Tasks (wait(), wait_for())
  - Ergebnis lesen (get())

Anforderungen

- mehrere Threads können gleichzeitig auf die Datenstruktur zugreifen
- kein Thread sieht (Zwischen-)Zustand, bei dem Invarianten der Datenstruktur durch einen anderen Thread (kurzzeitig) verletzt ist
- Vermeidung von Wettlaufsituationen
- Vermeidung von Verklemmungen
- korrekte Behandlung von Ausnahmen (Fehlern)

Thread-sichere Queue

```
1  template <typename T>
2  class ts_queue {
3  private:
4      mutable std::mutex mtx;
5      std::condition_variable cond;
6      std::queue<T> the_queue;
7  public:
8      ts_queue() {}
9      ...
10 };
```

- Zeilen 1,2,6: Kapselung der std::queue-Klasse
- Zeile 4: Mutex für exklusiven Zugriff
- Zeile 5: Bedingungsvariable für Warten

Thread-sichere Queue: Methode push

```
1  void push(T val) {
2      std::lock_guard<std::mutex> l(mtx);
3      the_queue.push(std::move(val));
4      cond.notify_one();
5  }
```

- Zeile 2: Lock Guard sichert exklusiven Zugriff
- Zeile 3: Element an die Queue anhängen
- Zeile 4: Aufwecken von eventuell wartenden Threads

Thread-sichere Queue: Methode waiting_pop

```
1  void waiting_pop(T& val) {
2      std::lock_guard<std::mutex> l(mtx);
3      cond.wait(l, [this] { return !the_queue.empty(); });
4      val = std::move(the_queue.front());
5      the_queue.pop();
6  }
```

- Zeile 2: Lock Guard sichert exklusiven Zugriff
- Zeile 3: Warten bis Queue nicht mehr leer ist
- Zeilen 4,5: erstes Element aus der Queue entnehmen

async, future und promise

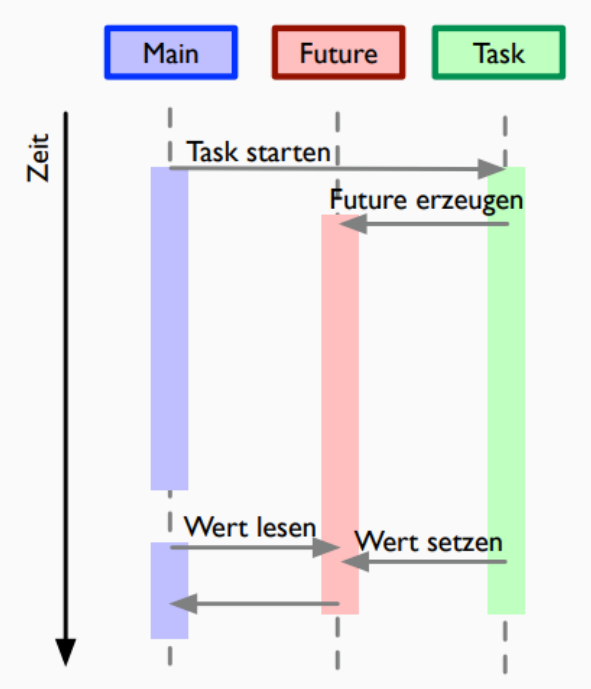
- std::future - Resultat einer asynchronen Berechnung, d.h. einer Berechnung die erst noch stattfindet
- std::async() - asynchrones Starten eines Tasks

```
1  int long_calculation() { ... }
2  std::future<int> result = std::async(long_calculation);
3  // Fortsetzung der Berechnung ...
4  result.wait();
5  std::cout << result.get() << std::endl;
```

- std::promise - erlaubt Wert zu setzen, wenn der aktuelle Thread beendet ist, of in Kombination mit std::future eingesetzt
- future = Ergbenisobjekt, promise = Ergebnisproduzent

Future

- Methoden zum
  - Warten auf das Ende des Tasks (wait(), wait_for())
  - Ergebnis lesen (get())



Beispiel

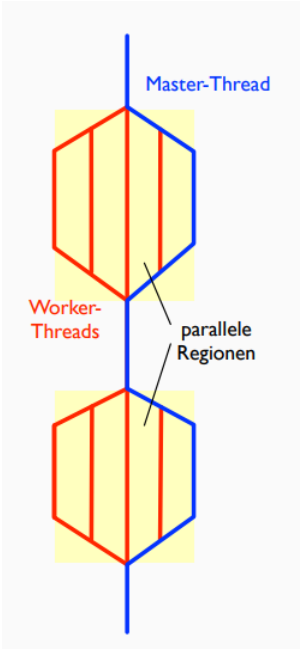
```
1  // Promise für einen String-Wert
2  std::promise<std::string> promise;
3  // zugehöriges Future-Objekt
4  auto res = promise.get_future();
5
6  // Produzenten-Thread
7  auto producer = std::thread([&] {
8      promise.set_value("Hello World!");
9  });
10 // Konsumenten-Thread
11 auto consumer = std::thread([&] {
12     std::cout << res.get() << "\n";
13 });
14 producer.join();
15 consumer.join();
```

Deklarative Parallelisierung mit OpenMP

- Programmierschnittstelle für Parallelisierung in C/C++/Fortran
- Programmiersprachenerweiterung durch Direktiven
- in C/C++: #pragma omp ...
- zusätzliche Bibliotheksfunktionen: #include ¡omp.h¿
- aktuelle Version 5.0
- Unterstützung in gcc und clang
  - vollständig 4.5, partiell 5.0
  - Nutzung über Compilerflag - fopenmp
- beschränkt auf Architekturen mit gemeinsamen Speicher

Programmiermodell

- Master-Thread und mehrere Worker-Threads (Anzahl typischerweise durch OpenMP-Laufzeitsystem bestimmt)
- über parallel-Direktive kann Arbeit in einem Programmabschnitt auf Worker-Threads aufgeteilt werden
- Ende des parallelen Abschnitts ⇀ implizite Synchronisation
- Fortsetzung des Master-Threads



- Master-Thread und mehrere Worker-Threads (Anzahl typischerweise durch OpenMP-Laufzeitsystem bestimmt)
- über parallel -Direktive kann Arbeit in einem Programmabschnitt auf Worker-Threads aufgeteilt werden
- Ende des parallelen Abschnitts → implizite Synchronisation → Fortsetzung des Master-Threads
- der dem 'pragma' folgende Block wird parallel von allen Threads ausgeführt

```
#include <iostream>
#include <omp.h>
int main() {
    #pragma omp parallel
    {
        std::cout << "Hello World from thread #"
        << omp_get_thread_num() << " of "
        << omp_get_num_threads() << "\n";
    }
    std::cout << "Finished!\n";
    return 0;
}

```

- Schleifenparallelisierung: jedem Thread wird ein Teil der Iteration zugewiesen (beeinflusst nur äußere Schleife)

```
...
#pragma omp parallel for
  for (int i = 0; i < 20; i++) {...

```

  - collapse(n) gibt an, dass n Schleifen in einem gemeinsamen Iterationsbereich zusammengefasst und auf die Threads verteilt werden sollen

```
    #pragma omp parallel for collapse(3)

```

- Beeinflussung der Thread Anzahl
  - maximale Anzahl:

```
    #pragma omp parallel for num_threads(8)

```

  - bedingte Parallelisierung:

```
    #pragma omp parallel for if(i>50)

```

- Aufteilung des Iterationsbereichs; Beeinflussung durch schedule -Direktive
  - schedule(auto): Default - implementierungsspezifisch
  - schedule(static, n): statische Round-Robin-Verteilung - Bereiche der Größe n (Angabe von n ist optional)
  - schedule(dynamic, n): dynamische Verteilung nach Bedarf
  - schedule(guided, n): Verteilung nach Bedarf und proportional zur Restarbeit
- Direktiven für parallele Ausführung
  - '#pragma omp single/master' Abschnitt wird nur durch einen/den Master-Thread ausgeführt
  - '#pragma omp critical' kritischer Abschnitt
  - '#pragma omp barrier' Warten auf alle Worker-Threads
  - '#pragma omp atomic' kritischer Abschnitt, Zugriff auf gemeinsame Variable (z.B. Zähler)
- Speicherklauseln für Variablen
  - 'shared' für alle Threads sichtbar/änderbar
  - 'private' jeder Thread hat eigene Kopie der Daten, wird nicht außerhalb initialisiert
  - 'reduction' private Daten, die am Ende des Abschnitts zu globalem Wert zusammengefasst werden
  - 'firstprivate / lastprivate' privat - initialisiert mit letztem Wert vor dem Abschnitt / Wert des letzten Threads der Iteration wird zurückgegeben
- zuweisung von Programmabschnitten zu Threads → statische Parallelität (geeignet für rekursive Abschnitte)

```
    #pragma omp parallel sections
    {
      #pragma omp section
      qsort(data, left, p - 1);
      #pragma omp section
      qsort(data, p + 1, right);
    }

```

- Task Programmierung
  - reihum Threads zugewiesen werden
  - an beliebiger Stelle definiert werden können
  - von beliebigem Thread definiert werden kann

```
    unsigned int f1, f2;
    #pragma omp task shared(f1)
      f1 = fib(f - 1);
    #pragma omp task shared(f2)
      f2 = fib(f - 2);
    #pragma omp taskwait
      return f1 + f2;

```

Hello World! mit OpenMP

- der dem pragma folgende Block wird parallel von allen Threads ausgeführt

```cpp
#include <iostream>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        std::cout << "Hello World from thread #"
                  << omp_get_thread_num() << " of "
                  << omp_get_num_threads() << "\n";
    }
    std::cout << "Finished!\n";
    return 0;
}
```

Schleifenparallelisierung

- parallele Ausführung einer Schleife: jedem Thread wird ein Teil der Iterationen zugewiesen
- für for-Schleifgen mit eingeschränkter Syntax (ganzzahlige Schleifenvariablen, Operatoren auf Schleifenvariablen) und für STL-Iteratoren

```cpp
unsigned int results[20];
#pragma omp parallel for
for (int i = 0; i < 20; i++) {
    auto f = rand() % 30;
    results[i] = fibonacci(f);
}
```

Beeinflussung der Thread-Anzahl

maximale Anzahl

```cpp
unsigned int results[20];
#pragma omp parallel for num_threads(8)
for (int i = 0; i < 20; i++) {
    results[i] = fibonacci(rand() % 30);
}
```

bedingte Parallelisierung

```cpp
unsigned int results[20];
#pragma omp parallel for if (i > 50)
for (int i = 0; i < 20; i++) {
    results[i] = fibonacci(rand() % 30);
}
```

Aufteilung des Iterationsbereichs

- Iterationsbereich kann auf verschiedene Weise auf Threads aufgeteilt werden
- Beeinflussung durch schedule-Direktive
  - schedule(auto): Default - implementierungsspezifisch
  - schedule(static,n): statische Round-Robin-Verteilung - Bereiche der Größe n (Angabe von n ist optional)
  - schedule(dynamic, n): dynamische Verteilung nach Bedarf
  - schedule(guided, n): Verteilung nach Bedarf und proportional zur Restarbeit
  - ...

Geschachtelte Schleifen

- Parallelisierung mit **parallel for** beeinflusst nur äußere Schleife
- collapse(n) gibt an, dass n Schleifen in einem gemeinsamen Iterationsbereich zusammengefasst, und auf die Threads verteilt werden sollen
- Beispiel: Matrizenmultiplikation

```
#pragma omp parallel for collapse(3)
for (int row = 0; row < m; row++)
    for (int col = 0; col < n; col++)
        for (int inner = 0; inner < k; inner++)
            prod[row][col] += A[row][inner] * B[inner][col];
```

Synchronisation

- Direktiven für parallele Ausführung
  - **#pragma omp single/master** Abschnitt wird nur durch einen/den Master-Thread ausgeführt
  - **#pragma omp critical** kritischer Abschnitt
  - **#pragma omp barrier** Warten auf alle Worker-Threads
  - **#pragma omp atomic** kritischer Abschnitt - ZUgriff auf gemeinsame Variable (z.B. Zähler)
- Speicherklauseln für Variablen
  - **shared** für alle Threads sichtbar/änderbar
  - **private** jeder Thread hat eigene Kopie der Daten, wird nicht außerhalb initialisiert
  - **reduction** private Daten, die am Ende des Abschnitts zu globalem Wert zusammengefasst werden
  - **firstprivate/lastprivate** privat - initialisiert mit letztem Wert vor dem Abschnitt / Wert des letzten Threads der Iteration wird zurückgegeben

Parallele Abschnitte

- Zuweisung von Programmabschnitten zu Threads ⇀ statische Parallelität
- geeignet z.B. für rekursive Aufrufe

```
void qsort(int data[], int left, int right) {
    if (left < right) {
        int p = partition(data, left, right);

        #pragma omp parallel sections
        {
            #pragma omp section
            qsort(data, left, p - 1);
            #pragma omp section
            qsort(data, p + 1, right);
        }
    }
}
```

Task-Programmierung mit OpenMP

- seit OpenMP 3.0 Unterstützung von Tasks, die
  - reihum Threads zugewiesen werden
  - an beliebiger Stelle definiert werden können
  - von beliebigem Thread definiert werden kann

```
unsigned int fibonacci(unsigned int f) {
    if (f < 2) return n;
    unsigned int f1, f2;
    #pragma omp task shared(f1)
    f1 = fib(f - 1);
    #pragma omp task shared(f2)
    f2 = fib(f - 2);
    #pragma omp taskwait
    return f1 + f2;
}
```

Fazit

- C++ bietet weitreichende und mächtige Konzepte zur Parallelisierung
  - von Basiskontrolle wie Threads und Synchronisationsprimitiven (u.a. Mutexe)
  - ...über höherwertige Abstraktionen wie async, Features und Promises
  - bis hin zu deklarativen Ansätzen wie OpenMP
- alle Formen von Parallelität (Instruktions-, Daten-, und Taskparallelität) möglich
- aber anspruchsvolle Programmierung
- erleichtert durch zusätzliche Bibliotheken und Frameworks wie Parallel STL, TBB, ...

```
//[Hello.java]
package runnable;
public class Hello {

  public class Heartbeat implements runnable {
      int pulse;
      public Heartbeat(int p) { pulse = p * 1000; }
      public void run() {
          while(true) {
              try { Thread.sleep(pulse); }
              catch(InterruptedException e) {}
              System.out.println("poch");
```

```
            }
        }
    }

    public static void main(String[] args) {
        Thread t = new Thread(new Heartbeat(2)); //Thread Objekt mit runnable erzeugen
        t.start(); //methode start() aufrufen $\rightarrow$ ruft run() auf
    }
}
```

```
//[Hello.cpp]
#include <iostream> // Datei iostream aus System-Includes
#include "X.hpp" // Datei X.hpp aus Projekt-Ordner

#ifdef DEBUG // falls Konstante DEBUG definiert ist
std::cout << "Wichtige Debugausgabe" << std::endl;
#endif

#define DEBUG // Konstante setzen

class Stromfresser {
  public:
      Stromfresser() {
          std::cout << "Mjam" << std::endl;
      }
};
class Roboter : virtual public Stromfresser {};

void fun(int n, const std::string& s) {
  std::lock_guard<std::mutex> guard(my_mtx); //verklemmungsfrei mit unique_lock<std::mutex>
  for (auto i = 0; i < n; i++)
      std::cout << s << " ";
  std::cout << std::endl;
}

std::mutex my_mtx;

int main(int argc, char* argv[]){
  std::thread t(fun, 2, "Hello");
  t.join();
  return 0;
}
```

```
%[Hello.erl]
-module(cheat_sheet).  % end with a period

%% Let these functions be called externally.
-export([countdown/1, countdown/0]).  % number of parameters - it matters!

%% atoms begin with a lower-case letter
%% Variables begin with an upper-case letter

%% Start defining a function with the most specific case first
countdown(0) $\rightarrow$
  io:format("Zero!~n");  % function clauses end with a semicolon

%% Another clause of the same function, with a guard expression
countdown(Bogus) when Bogus < 0 $\rightarrow$
  io:format("Bad value: ~B~n", [Bogus]), % normal lines end with a comma
  error;  % return value (io:format returns 'ok')

%% Last clause matches any single parameter
countdown(Start) $\rightarrow$
  %% case and if statements return values!
  Type = case Start rem 2 of
      1 $\rightarrow$ "odd";  % case and if clauses end with semicolons
      0 $\rightarrow$ "even"  % except the last one
  end,  % end with comma, like a normal line
  io:format("~B is ~s~n", [Start, Type]),
  countdown(Start - 1).  % When the function is all done, end with a period

%% This is a different function because it has a different number of parameters.
countdown() $\rightarrow$
  countdown(10).
```

Parallele Programmierung in Java

Unterstützung durch

- Thread-Konzept
- eingebaute Mechanismen zur Synchronisation nebenläufiger Prozesse
- spezielle High-Level-Klassen im Package **java.util.concurrent**

Threads in Java

- Repräsentiert durch Klasse **java.lang.Thread**
- Implementierung eines eigenen Kontrollflusses
- Eigene Klasse muss Runnable implementieren
  - Implementierung des Interface **java.lang.Runnable**
    - keine weitere Beeinflussung des Threads über zusätzliche Methoden notwendig
    - soll von anderer Klasse als Thread abgeleitet werden
  - Subklasse von **java.lang.Thread**
    - zusätzliche Methoden zur Steuerung des Ablaufs benötigt
    - keine andere Superklasse notwendig

Threads: Runnable-Schnittstelle Eigene Klasse muss **Runnable** implementieren

- Methode **public void run()** - wird beim Start des Threads aufgerufen

```
public class Heartbeat implements Runnable {
    int pulse;
    public Heartbeat(int p) { pulse = p * 1000; }
    public void run() {
        while(true) {
            try { Thread.sleep(pulse); }
            catch(InterruptedException e) {}
            System.out.println("poch");
        }
    }
}

```

Thread-Erzeugung

- Thread-Objekt mit Runnable-Objekt erzeugen
- Methode **start()** aufrufen
  - Ruft **run()** auf

```
public static void main(String[] args) {
    Thread t = new Thread(new Heartbeat(2)); //Thread Objekt mit runnable erzeugen
    t.start(); //methode start() aufrufen $\rightarrow$ ruft run() auf
}

```

Subklasse von Thread

- Klasse muss von Thread abgeleitet werden
- Methode run() muss überschrieben werden

```
public class Heartbeat2 extends Thread {
    int pulse = 1000;
    public Heartbeat2() {}
    public void setPulse(int p) { pulse = p * 1000; }

    public void run() {
        while(true) {
            try { Thread.sleep(pulse); }
            catch(InterruptedException e) {}
            System.out.println("poch");
        }
    }
}
```

- Objekt der eigenen Thread-Klasse erzeugen
- Methode **start()** aufrufen
  - Ruft **run()** auf

```
public static void main(String[] args) {
    Heartbeat2 t = new Heartbeat2(2);
    t.start();
}
```

- Spätere Beeinflussung durch andere Threads möglich

```
...
t.setPulse(2);
```

Threads: Wichtige Methoden

void start() initiiert Ausführung des Threads durch Aufruf der Methode run
void run() die eigentliche Arbeitsmethode
static void sleep(int millis) hält die Ausführung des aktuellen Threads für 'millis' Millisekunden an; Keinen Einfluss auf andere Threads!
void join() blockiert den aufrufenden Thread so lange, bis der aufgerufene Thread beendet ist

Parallele Berechnung von Fibonacci-Zahlen

```
public class Fibonacci implements Runnable {
    int fi;
    public Fibonacci(int f) { fi = f; }
    int fibo(int f) {
        if (f < 2) return 1;
        else return fibo(f-1) + fibo(f-2); }

    public void run() {
        int res = fibo(fi);
        System.out.println("Fibonacci(" + fi
                + ") = " + res);
    }
}
```

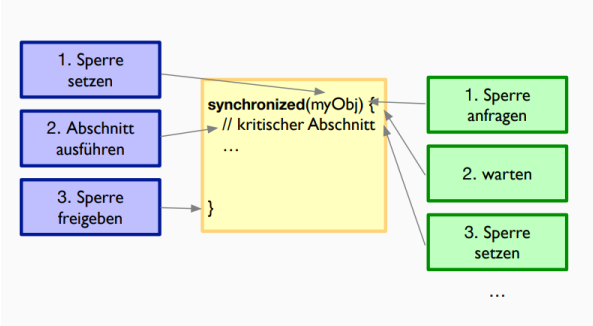
Thread-Erzeugung und Ausführung

```
public static void main(String[] args) {
    Thread[] threads = new Thread[10];
    for (int i = 0; i < 10; i++) {
        threads[i] = new Thread(new Fibonacci(40 + i));
        threads[i].start();
    }
}
```

Wechselseitiger Ausschluss in Java

Schlüsselwort **synchronized**

- Implementierung von sogenannten Monitoren bzw. locks (exklusiven Sperren)
  - nur ein Thread darf den kritischen Abschnitt betreten
  - alle anderen Threads, die darauf zugrifen wollen, müssen auf Freigabe warten
- für Methoden: **public synchronized void doSomething()**
  - nur ein Thread darf diese Methode auf einem Objekt zur gleichen Zeit ausführen
- für Anweisungen: **synchronized(anObject){...}**
  - nur ein Thread darf den Block betreten
  - Sperre wird durch das Objekt **anObject** verwaltet (jedem Java-Objekt ist eine Sperre zugeordnet)



- Schlüsselwort synchronized
  - Implementierung von sogenannten Monitoren bzw. locks (exklusiven Sperren); nur ein Thread darf den kritischen Abschnitt betreten; alle anderen Threads, die darauf zugreifen wollen, müssen auf Freigabe warten
  - für Methoden:

```
1       public synchronized void doSomething()
2
```

    - nur ein Thread darf diese Methode auf einem Objekt zur gleichen Zeit ausführen
  - für Anweisungen: synchronized(anObject) ...
    - nur ein Thread darf den Block betreten
    - Sperre wird durch das Objekt anObject verwaltet (jedem Java-Objekt ist ein Sperre zugeordnet)

wait & notify

Signalisierung zwischen Threads in Java
Basismethoden der Klasse **java.lang.Object**
wait() der aktive Thread wartet an diesem Objekt, Sperren werden ggf. freigegeben.
notify() wekct an diesem Objekt wartenden Thread auf
notifyAll() weckt alle an diesem Objekt wartenden Threads auf
wait() & notify() dürfen nur in einem **synchronized**-Block aufgerufen werden

Java: High-Level-Klassen

- Paket **java.util.concurrent** seit Java Version 1.5
- Abstraktionsschicht versteckt Details über Thread-Erzeugung
- Übernimmt Erstellung und Überwachung von parallelen Tasks, u.a.
  - **ExecutorService** zum erzeugen asynchroner Tasks
  - **Future**: Referenz auf diesen Task bzw. dessen Ergebnis
  - **ForkJoinPool & RecursiveAction**: rekursives Aufteilen eines großen Problems

Tasks und Futures in Java

- Task = logische Ausführungseinheit
- Thread = Mechanismus zur asynchronen/parallelen Ausführung von Tasks

```
1 Runnable task = () -> {
2   String me = Thread.currentThread().getName();
3   System.out.println("Hallo " + me);
4 };
5 task.run();
6 Thread thread = new Thread(task);
7 thread.start();
```

Future & ExecutorService

- **ExecutorService** stellt Methoden zum Starten/Beenden/Steuern von parallelen Aufgaben bereit
- implementiert **Executor**-Interface
  - definiert Methode **void execute(Runnable r)**
- Starten einer Aufgabe mit **submit**
  - *Future* < *T* > **submit(Callable c)**
  - *Future* <? > **submit(Runnable r)**
- Zugriff auf das Ergebnis mit **get**
  - **T get(long timeout, TimeUnit unit)**
  - **T get()**

Future & ExecutorService: Beispiel

```
class App {
    ExecutorService executor = Executors.newFixedThreadPool(4);
    void search(final String w) throws InterruptedException {
        Future<String> future =
            executor.submit(new Callable<String>() {
                public String call() {
                    return searcher.search(target);
                }
            });
        displayOtherThings(); // do other things
        try {
            displayText(future.get()); // get is blocking
        } catch (ExecutionException ex) {
            cleanup();
            return;
        }
    }
}
```

RecursiveAction & Fork/Join

- Rekursives Zerlegen eines großen Problems in kleinere Probleme
- Solange bis Problem klein genug um direkt ausgeführt werden zu können
- Task erstellt zwei oder mehr Teiltasks von sich selbst ⇀ Datenparallelität
- ForkJoinPool zum Ausführen → implementiert Executor Interface
- Fazit
  - Parallelprogrammierung in Java sehr ähnlich zu C++
  - Konzepte: Threads, kritische Abschnitte über synchronized
  - mächtige Abstraktionen in java.util.concurrent
    - Tasks und Futures, Executor und ThreadPool
    - thread-sichere Datenstrukturen
    - Synchronisation: Barrieren, Semaphoren, ...

Beispiel

```
class MyTask extends RecursiveAction {
    String[] source; int start, length;
    public MyTask(String[] src, int s, int l) {
        source = src; start = s; length = l;
    }

    void computeDirectly() { ... }

    @Override
    void compute() {
        if (length < THRESHOLD) computeDirectly();
        else {
            int split = length / 2;
            invokeAll(new MyTask(source, start, split),
                new MyTask(source, start + split, length - split))
        }
    }
}
```

Starten der Verarbeitung:

1. (große) Gesamtaufgabe erstellen
2. ForkJoinPool erstellen
3. Aufgabe vom Pool ausführen lassen

```
String[] src = ...

MyTask t = new MyTask(src, 0, src.length);
ForkJoinPool pool = new ForkJoinPool();
pool.invoke(t);
```

Zusammenfassung

- Parallelprogrammierung als wichtige Technik zur Nutzung moderner Hardware (Multicore, GPU, ...)
- verschiedene Architekturen und Programmiermodelle
- Instruktions-, Daten- und Taskparallelität
- Message Passing vs. gemeinsamer Speicher
- Konzepte in Erlang, C++, Java
- hoher Abstraktionsgrad funktionaler Sprachen
- C++/Java: Thread-Modell und Synchronisation mit vielen weiteren Konzepten
- höherwertige Abstraktion durch zusätzliche Bibliotheken und Programmierschnittstellen

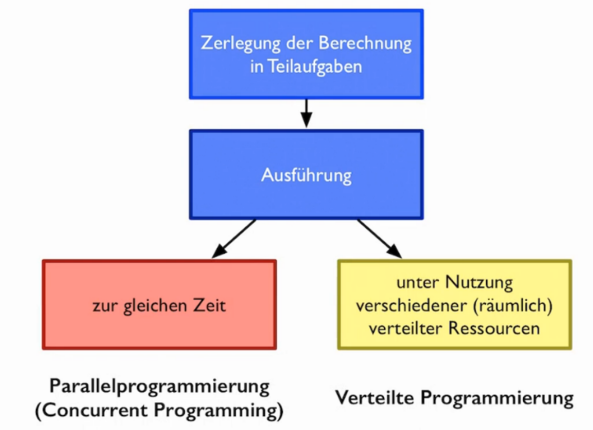
Verteilte Programmierung

Grundlagen

Lernziele

- Verständnis von Techniken verteilter Programmierung als Paradigma
  - Modelle und Konzepte unabhängig von Programmiersprache und Betriebssystem
  - Herausforderungen und Besonderheiten verteilter Programme
- Kennenlernen konkreter Konzepte und Mechanismen
  - praktische Beispiele in Java, Erlang und C++
  - Bewertung und Vergleich verschiedener Plattformen

Einordnung



- mehrere Rechner
- Prozesse auf verschiedenen Rechnern
- Kommunikation über Knotengrenzen hinweg
- Behandlung von Knoten- oder Netzwerkausfällen

Ziele

- Bisher:
  - eine Maschine
  - Prozesse kommunizieren nur innerhalb dieser Maschine (shared Memory vs. Message Passing)
- Jetzt:
  - mehrere Rechner
  - Prozesse auf verschiedenen Rechnern
- Erfordert:
  - Kommunikation über Knotengrenzen hinweg
  - Behandlung von Knoten- oder Netzwerkausfällen

Motivation

- viele verschiedene Systeme (Knoten) zur Verfügung
  - PC, Server, virtuelle Maschinen
  - Lastverteilung, Spezialisierung auf bestimmte Probleme
- Knoten sind über Netzwerke verbunden
  - LAN(Wohnräume, Büros,...): bis zu 10 Gbit/s
  - MAN(Metropolitan Area Network, Behördennetze, dicht besiedelte Regionen): bis zu 10 Gbit/s
  - WAN(Wide Area Network, weltweite Vernetzung): hohe Kapazitäten zwischen den ISPs

Wofür werden verteilte Systeme eingesetzt?

- Gemeinsame Nutzung von Ressourcen
  - Cloud-Umgebungen
  - verteilte Datenbanksysteme
- Teilaufgaben in großen Anwendungen
  - parallele Ausführung
  - getrennte Teilaufgaben (Micro-Services)
- Informationsaustausch
  - Email, Messenger
  - verteilte Algorithmen

Software Architekturen

1. Früher: Hardware, Betriebssystem, Anwendung
   - Virtualisierung von Prozessor, Speicher, E/A Systemen
   - Interprozesskommunikation (IPC)
2. Middlewaresysteme: Hardware, OS, Middleware, Anwendung
   - verteilte Dienste
   - Programmierparadigmen: RPC, Client/Server,...
   - Java, CORBA, ...
3. Heute: Virtualisierung
   - VM Hypervisor: verstecken Hardware vor dem Betriebssystem
   - Docker: eine Anwendung pro Container

Herausforderungen

- viele verschiedene Computer/Server
  - verschiedene Betriebssysteme
  - unterschiedliche Leistungsfähigkeit
- Systemkomponenten müssen miteinander kommunizieren
- verteilte Algorithmen: Nachrichten senden, empfangen, bestätigen, Synchronisation
- Knotenausfälle behandeln

⇒ brauchen Modelle zur Beschreibung der Kommunikation

Anforderungen

Anforderungen an Kommunikationsmodelle in ...
... verteilten Systemen

- Korrektheit
- Sicherheit
- Verfügbarkeit
- Skalierbarkeit
- Heterogenität

...verteilten Verkehrsmanagementsystemen

- Echtzeitfähigkeit
- Offenheit
- Korrektheit, Sicherheit
- Skalierbarkeit, Verfügbarkeit

Anforderungen an den Betrieb eines (großen) verteilten Systems

- (Last-)Skalierbarkeit (Scale-out):
  - viele kleine Server - statt eines großen
  - neue Server nach Bedarf hinzuzufügen
- Funktionssicherheit (Safety) / IT-Sicherheit (Security)
- Fehlertoleranz / Verfügbarkeit
  - Ausfälle von einzelnen Knoten kompensieren
  - Redundante Verarbeitung
- Offenheit / Interoperabilität
  - neue Knoten und Systeme einfach integrieren
- Transparenz
  - verstecke die vielen Server vor den Nutzern

Grundlagen verteilter Programmierung in Java und Erlang

Sockets

- Verteilte Programmierung: Wir müssen einen entfernten Computer ansprechen
- benötigen: Adresse → IP-Adresse
- da mehrere Dienste auf demselben Computer laufen lauscht jeder Dienst auf einem Port (Nummer)
  - Wichtige Ports: 80 WWW, 20 (FTP), 25 (SMTP)
- **Socket** beschreibt einen Endpunkt, dh. Adresse und Port in einem TCP (oder UDP) Netzwerk
- **Server-Socket** wartet auf Verbindungen
- **Client** initiiert Verbindung, ebenfalls über einen (Client-Socket)

Sockets in Java

Socket in dem package "java.net.Socket"

- einen **ServerSocket** auf Port 4242 erstellen

```
1       ServerSocket serverSocket = new ServerSocket(4242);
2
```

- Warte blockierend auf Verbindungen

```
1       Socket client = serverSocket.accept();
2
```

- Client Socket erstellen und zum Server verbinden

```
1       Socket client = new Socket("localhost",4242);
2
```

- Sockets in ähnlicher Form in C++

Sockets in Java - Beispiel Echo Server (Serverseite)

```
1   ServerSocket server = new ServerSocket(4242);
2   while(true){
3     try(Socket client = server.accept(); ){
4       Scanner in = new Scanner(client.getInputStream());
5       PrintWriter out = new PrintWriter(client.getOutputStream(), true);
6       String line = in.readLine();
7       out.println(line);
8     }
9     catch ( Exception e) { e.printStackTrace(); }
10  }
```

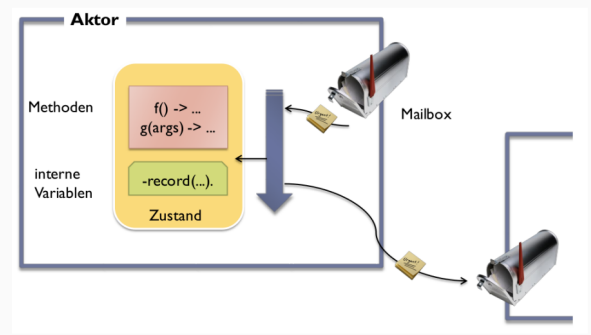
Echo Server (Clientseite)

```
try(Socket server = new Socket("localhost", 4242); ){
  Scanner in = new Scanner(client.getInputStream() );
  PrintWriter out = new PrintWriter(server.getOutputStream(), true);
  out.println("Hello World");
  System.out.println(in.nextLine());
}
catch ( Exception e) { e.printStackTrace(); }
```

Aktormodell in Erlang

- formales Modell für Nebenläufigkeit und Verteilung
- Basis für verschiedene Programmiersprachen/Frameworks: Erlang, Akka (Scala/Java)
- Prinzipien:
  - Aktor kapselt Zustand und Verhalten
  - Aktoren sind aktiv
  - Aktoren kommunizieren durch Nachrichtenaustausch
    - Nichtblockierendes Senden
    - Blockierendes Empfangen

Übersicht



- Aktormodell in Erlang nativ umgesetzt
  - Sende- und Empfangsoperationen schon für parallele Programmierung benutzt
  - bisher aber nur auf einem Knoten
- Programmbestandteile im Aktormodell
  - Verhaltensdefinition ⇒ f() → ... end.
  - Erzeugen neuer Aktoren ⇒ Pid = spwan(fun ...).
  - Empfangen von Nachrichten ⇒ receive ... end.
  - Senden ⇒ Pid ! Request.
- kein globaler Zustand

Kommunikation zwischen Erlangknoten

Erlangknoten starten (sname = short name)

```
#erl -sname node1 -setcookie 1234
 Eshell V11.0 (abort with ^G)
 (node1@localhost)1>
```

Weiteren Erlangknoten starten (selber oder anderer PC)

```
#erl -sname node2 -setcookie 1234
  Eshell V11.0 (abort with ^G)
  (node2@localhost)1>
```

Liste der verbundenen Knoten

```
(node@localhost)1> nodes().
[]
```

Cookie-System

- verteilte Erlangknoten benötigen zur Kommunikation gemeinsames Magic Cookie (Passwort)
- Mehrere Varianten
  - Datei ~/.erlang.cookie
  - Erlang-Funktion

```
      :set_cookie(node(), Cookie).
```

  - Option

```
      erl -setcookie Cookie
```

Verbindungsaufbau zwischen Erlangknoten

Verbindungsaufbau mittels net_adm:ping Funktion

```
(node1@localhost)2> net_adm:ping("node2@localhost").
pong
(node1@localhost)3> nodes().
["node2@localhost"]

(node2@localhost)1> nodes().
["node1@localhost"]
```

Kommunikation zwischen Erlangknoten

Starten eines Prozesses auf einem entfernten Host

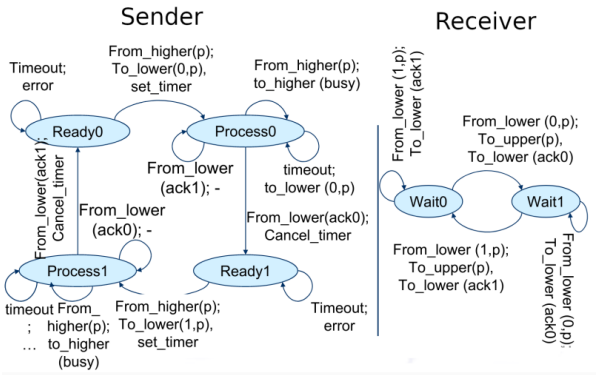
```
complicated() ->
  receive
    {Sender, I} -> Sender ! I*I
  end.

sender(Num, Pid) ->
  Pid ! {self(), Num},
  receive
    Res -> io:format("Result = ~p~n", [Res])
(node1@localhost)4> N2Pid = spawn("node2@localhost", fun ch5_1:complicated/0)
(node1@localhost)5> ch5_1:sender(25, N2Pid).
Result = 625
ok
```

Alternating Bit Protokoll

Übersicht

- ermöglicht es, Nachrichten über einen verlustbehafteten Kommunikationskanal vollständig zu übertragen, sofern Verluste nur gelegentlich auftreten (transiente Fehler)
- Empfänger quittiert jedes erhaltene Paket (Achnowledgement, kurz ACK)
- **Achtung** Kanal kann Nachrichten und ACKs verlieren
  - benötigen je zwei unterschiedliche Sequenznummern und ACKs
- Empfänger liefert eine Nachricht nur beim ersten Empfang aus (keine Duplikate)
- bei Timeout: Nachricht erneut senden
- bei Erhalt eines unerwarteten ACKs: **aktuelle Nachricht erneut senden**

Zustände



Das Alternating Bit Protokoll

Wir implementieren eine Variante, bei welcher:

- der Sender zu Beginn eine Liste mit sämtlichen zu sendenden Nachrichten erhält, und
- der Empfänger die erstmals empfangenen Nachrichten einfach auf dem Bildschirm ausgibt
- alle Aktoren Statusmeldungen ausgeben
- Verluste über einen Zufallszahlengenerator ausgelöst werden

drei Prozesse mit **initialize(ErrorRate, NumberOfMessages, ReceiverPid, SenderPid, ChannelPid)** initialisieren und starten

- Sender hat vier Zustandsfunktionen; Startet mit senderReady0(List): Liste mit Zahlen 1,...,NumberOfMessages
- Kanal: Nachricht "verlieren", wenn Zufallszahl $\not\geq$ ErrorRate
- Empfänger hat zwei Zustandsfunktionen; zu Beginn wird receiverWait0 gestartet
- initialize wartet auf eine ready-Nachricht; sendet danach stop-Nachrichten an alle

```erlang
-module(altbit).
-export([initialize/5]).
-import(rand, [seed/3, uniform/0]).

for(Max, Max, F) -> [F(Max)]; % for convenience
for(I, Max, F) -> [F(I)|for(I+1, Max, F)].
```

```erlang
initialize(ErrorRate, NumberOfMessages, R, S, C) ->
  rand:seed({23, 13, 97}), % initialize RNG
  SendList = for(1,NumberOfMessages,fun(I) -> I end),
  register(initializer, self()), % others may send us "ready"
  Receiver = spawn(R, fun() -> receiverWait0() end),
  register(receiver, Receiver),
  Channel =  spawn(C, fun() -> channelIdle(ErrorRate) end),
  register(channel, Channel),
  Sender = spawn(S, fun() -> senderReady0(SendList) end),
  register(sender, Sender),
  io:format("Started ABP with ~.10B Messages and Error-Rate ~f
  ↪  ~n", [NumberOfMessages, ErrorRate]),
  receive % wait for Signal that all Messages went ok
    ready -> io:format("All ~.10B Messages
    ↪  sent.~n",[NumberOfMessages])
  end, % Now clean up everything:
  Sender ! stop,  Receiver ! stop, Channel ! stop,
  unregister(initializer).
```

```erlang
senderReady0([]) -> initializer ! ready;
senderReady0([M|MS]) ->
  channel ! {receiver, {seq0, M}},
  io:format("Sender: sends Message ~.10B. ~n",[M]),
  senderProcess0([M|MS]).
```

```erlang
senderProcess0([]) -> initializer!ready; % to be safe
senderProcess0([M|MS]) ->
  receive
    ack0 -> io:format("Sender: received expected Ack for
    ↪  Message ~.10B. ~n",[M]),
            senderReady1(MS);
    ack1 -> io:format("Sender: received unexpected Ack; Send
    ↪  Again Message ~.10B.~n",[M]),
            channel ! {receiver, {seq0, M}},
            senderProcess0([M|MS]);
    stop -> true
    after 1000 -> io:format("Sender: Timeout! Repeat Message
    ↪  ~.10B. ~n",[M]),
                  channel ! {receiver, {seq0, M}},
                  senderProcess0([M|MS])
  end.
```

```erlang
channelIdle(ErrorRate) ->
  RN = uniform(), % for determining if msg to be dropped
  receive
    {receiver, {Seq, M}} when RN =< ErrorRate -> % drop
      io:format("Channel: drops Message ~.10B. ~n",[M]),
      channelIdle(ErrorRate);
    {receiver, {Seq, M}} when RN > ErrorRate ->  % deliver
      receiver!{Seq, M},
      channelIdle(ErrorRate);
    {sender, M} when RN =< ErrorRate ->          % drop
      io:format("Channel: drops  ~w ~n",[M]),
      channelIdle(ErrorRate);
    {sender, M} when RN > ErrorRate ->% deliver
      sender ! M,
      channelIdle(ErrorRate);
    stop -> true
  end.
```

```erlang
receiverWait0() ->
  receive
    {seq0, M} ->
      io:format("Receiver: received and delivers Message
      ↪  ~.10B. ~n",[M]),
      channel ! {sender, ack0},
      receiverWait1();
    {seq1, M} ->
      io:format("Receiver: ignores unexpected Message ~.10B.
      ↪  ~n",[M]),
      channel ! {sender, ack1},
      receiverWait0();
    stop -> true
  end.
```

```
> altbit:initialize(0.45, 3, 'receiver@pc1', 'channel@pc2', 'sender@pc3').
Started ABP with 3 Messages and Error-Rate 0.450000
Sender: sends Message 1.
Channel: drops Message 1.
Sender: Timeout! Send Again Message 1.
Channel: drops Message 1.
Sender: Timeout! Send Again Message 1.
Receiver: received and delivers Message 1.
Sender: received expected Ack for Message 1.
Sender: sends Message 2.
Receiver: received and delivers Message 2.
Channel: drops  ack1
Sender: Timeout! Send Again Message 2.
Receiver: ignores unexpected Message 2.
Sender: received expected Ack for Message 2.
Sender: sends Message 3.
Receiver: received and delivers Message 3.
Sender: received expected Ack for Message 3.
All 3 Messages successfully transmitted.
```

Kommunikationsmodelle & Implementierungen

Kommunikationsmodelle

Frage: Wie sollen Knoten miteinander kommunizieren?

- Sprechen die Teilnehmer direkt miteinander oder über einen Vermittler?
- Kann jeder jedem eine Nachricht schicken?
- Wartet ein Teilnehmer darauf, dass seine Nachricht angekommen ist?
- Wartet ein Teilnehmer darauf, dass eine Nachricht ankommt?
- Muss ein Teilnehmer auf eine Nachricht antworten?

⇒ das Verhalten der Teilnehmer ist in Kommunikationsmodellen beschrieben

Arten von Kommunikationsmodellen

Es gibt viele verschiedene Modelle, z.B. für Botschaftenbasierte Modelle

- Auftragsorientierte Modelle
- Funktionsaufrufbasierte Modelle
- Blackboards
- Ereignisbasierte Modelle
- Strombasierte Modelle
- Wissensbasierte Modelle

Kommunikationspartner sind für uns:

- Threads/Prozesse innerhalb verteilter Anwendungen
- Komponenten verteilter Systeme (Browser ⇔ Webserver, DB Client ⇔ DB-Server)

Modellbestandteile

- Rollenmodell:

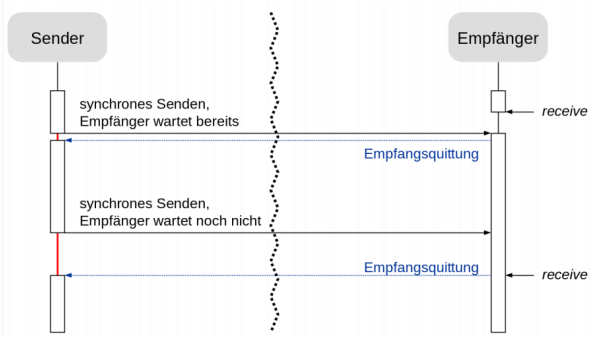
  - gemeinsames Handlungsmuster festlegen
  - z.B. Anrufer/Angerufener, Clinet/Server, Quelle/Senke
- Datenmodell:
  - einheitliche Interpretation der ausgetauschten Daten
  - z.B. Dateiformate (XML/JSON), Kodierungen (MPEG4/H.264)
- Fehlersemantiken
  - Einvernehmen über Wirkungen von Ausfällen
  - Eigenschaften von Kommunikationsoperationen müssen bei Ausfällen garantiert werden
- Terminierungssemantik
  - Einvernehmen über das Ende der Kommunikation
  - Garantien über das Ende von Kommunikationsoperationen (auch bei Ausfällen)

Kommunikationsarten

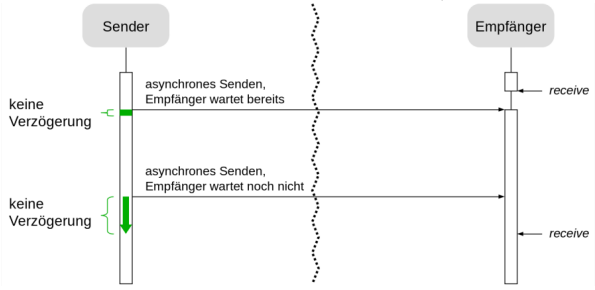
- Wann ist eine Kommunikationsoperation abgeschlossen?
- entspricht Terminierungssemantik
- zwei grundlegende Arten:
  - synchron
    - blockierend
    - Teilnehmer wartet bis die Gegenseite bereit ist
    - kann lange dauern, Sender kann nicht weiter arbeiten
    - Senden: Botschaftenankunft garantiert, einfache Implementierung synchroner Aktivitäten
    - Empfangen: Botschaftenankunft einfach und präzise feststellbar
  - asynchron
    - nicht-blockierend
    - Der Teilnehmer wartet nicht auf die Gegenseite (”fire and forget”)
    - unklar ob Botschaft angekommen
    - Senden: einfache Implementierung von Nebenläufigkeit
    - Empfangen: unklar wann Botschaft ankommt, einfache Implementierung von Nebenläufigkeit
  - gilt sowohl für das Senden als auch das Empfangen

Kommunikationsarten: Senden

synchrones Senden: Der Sender wartet bis der Empfänger die Botschaft annimmt



asynchrones Senden: Der Sender wartet nicht bis der Empfänger die Botschaft annimmt (”fire and forget”Prinzip)

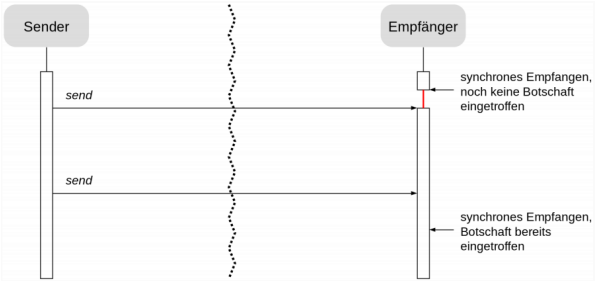


Synchrones vs. asynchrones Senden

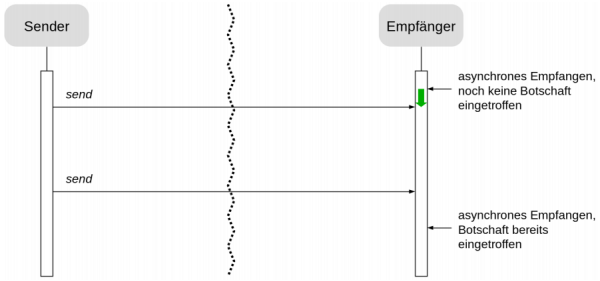
- synchrones Senden
  - kann lange dauern, der Sender kann währenddessen nicht weiterarbeiten
  - die Botschaftenankunft ist garantiert, eine einfache Implementierung synchroner Aktivitäten
- asynchrones Senden
  - unklar ob die Botschaft angekommen ist
  - einfache Implementierung von Nebenläufigkeit

Kommunikationsarten: Empfangen

synchrones Empfangen: Der Empfänger wartet bis die Botschaft eintrifft



asynchrones Empfangen: Der Empfänger macht weiter, falls keine Nachricht eingetroffen ist

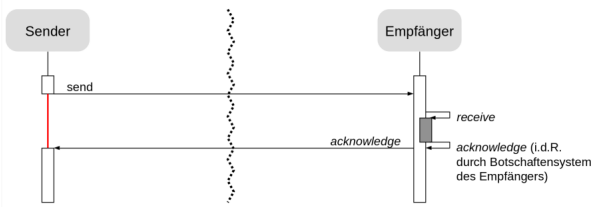


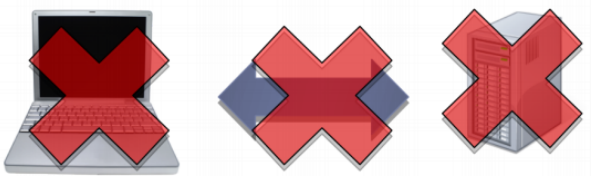
Synchrones vs. asynchrones Empfangen

- synchrones Empfangen:
  - kann lange dauern, der Sender kann nicht weiterarbeiten
  - Botschaftenankunft ist einfach und präzise feststellbar
- asynchrones Empfangen:
  - unklar wann die Botschaft ankommt;
    Benachrichtigungstechniken
    - Nachfragen (Polling)
    - ankommende Botschaft erzeugt neuen Thread beim Empfänger
    - weitere Techniken möglich
  - einfache Implementierung von Nebenläufigkeit

Fehlerbehandlung

- unverlässliches vs. verlässliches Senden
  - "Brief vs. Einschreiben"
- verlässliche Kommunikation erfordert
  - Quittierungen (Acknowledgements) → mehr Daten senden
  - Timeouts → Zeitverwaltung, langes Warten
- vielfältige Fehlermöglichkeiten in verteilten Anwendungen:
  - Kommunikations-/Netzwerkfehler: → Nachricht/Antwort gar nicht oder verzögert zugestellt
  - Serverausfall: Nachricht empfangen? Operation ausgeführt?
  - Clientausfall: Aufruf gültig? Bestätigung erhalten?





vielfältige Fehlermöglichkeiten in verteilten Anwendungen:

- Kommunikations-/Netzwerkfehler: → Nachricht/Antwort gar nicht oder nur verzögert zugestellt
- Serverausfall: Nachricht empfangen? Operation ausgeführt?
- Clientausfall: Aufruf gültig? Bestätigung erhalten?
- Beispiel: Reisebuchung
  - Buchung durchgeführt? Bestätigung erhalten?
  - Bei wiederholter Ausführung: wirklich neue Buchung?

Fehlerbehandlung in Erlang

- Timeout beim Warten auf Nachrichten
- Wenn keine passende Nachricht innerhalb **Time** msecs empfangen wird, dann wird der Rückgabewert des **after**-Ausdrucks verwendet.

```
1  receive
2      {ok, Resp} $\rightarrow$ Resp;
3      {notfound} $\rightarrow$ notfound;
4      after Time $\rightarrow$ timeout
5  end
6
```

Umgang mit Fehlern (Timeouts, Ausfälle):

- Maybe:
  - keine Wiederholung
  - keine Ausführungsgarantie
- At-least-once:
  - wiederholte Ausführung, aber keine Erkennung von Nachrichtduplikaten
  - nur für idempotente Operationen (Lesen)
- At-most-once:
  - garantiert, dass mehrfache Aufrufe nur zu einziger Ausführung führen
  - z.B. durch Sequenznummern (erfordert Protokollierung zur Duplikateliminierung)
  - für nicht-idempotente Operationen (schreibend, z.B. Einfügen, Löschen)

Überwachung von Erlang-Prozessen

- Linking von Prozessen: link(Pid)
- M überwacht S; S bricht durch Fehler ab

- M wartet auf EXIT Nachricht von S → asynchroner Handler nötig

on_exit-Handler

```
on_exit(Pid, Fun) ->
  spawn(fun() ->
    process_flag(trap_exit, true),
    link(Pid),
    receive
      {'EXIT', Pid, Why } -> Fun(Why)
    end
  end).
```

- überwacht den Prozess **Pid** auf Abbruch
- Anwendungsspezifische Reaktionen möglich
  - Fehlermeldung
  - Neustart des Prozesses
- auch über Erlang-Knotengrenzen hinweg!

Anwendung des on_exit-Handlers

```
1> F = fun() -> receive X -> list_to_atom(X) end end.
2> Pid = spawn(F).
3> on_exit(Pid, fun(Why) ->
    io:format("~p died with ~p~n", [Pid, Why]) end).
4> Pid ! ping.
ping
<0.41.0> died with
{badarg,[{erlang,list_to_atom,[ping]}]}
```

- Funktion anlegen (Liste in Atom konvertieren)
- Prozess erzeugen
- **on_exit**-Handler definieren
- Fehler verursachen (Nachricht ist keine Liste)

Fehlersemantiken

Umgang mit Fehlern (Timeouts, Ausfälle)

- Maybe:
  - keine Wiederholung
  - keine Ausführungsgarantie
- At-least-once:
  - wiederholte Ausführung, aber keine Erkennung von Nachrichtenduplikaten
  - nur für idempotente Optionen (Lesen)
- At-most-once:
  - garantiert, dass mehrfache Aufrufe nur zu einziger Ausführung führen
  - z.B. durch Sequenznummern (erfordert Protokollierung zur Duplikatelliminierung)
  - für nicht-idempotente Operationen (schreibend, z.B. Einfügen, Löschen)

Auftragsorientierte Modelle

- klassische Modell serviceorientierten Systemdesigns
- in verteilten Systemen:
  - Menge von Dienstanbietern (Server)
  - Menge von Clients, die diese Dienste nutzen wollen

Typische Anwendungsszenarien

DB-Server verwalten Datenbestände, verarbeiten SQL Anfragen

- Clients: "Gib mir alle Personen, die älter als 18 Jahre alt sind"

Web Webserver stellt HTML Dokumente bereit, Browser ruft URLs für Dokumente auf
E-Mail Mailserver verwalten Postfächer, leiten Mails weiter, Outlook/Thunderbird/...senden/lesen von Emails
Namensdienste (DNS), Fileserver, Zeitserver (NTP)

Auftragsorientierte Modelle: Modellsicht

- Rollenmodell: Clients erteilen Aufträge an Server
- Datenmodell: Notschaften mit vereinbarter Struktur (Protokoll)

```
POST /axis2/services/TimeWS HTTP/1.1
Content-Type: application/soap+xml; charset=UTF-8;
action="urn:getTimeOfDay"

<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope xmlns:soapenv="..." >
  <soapenv:Body/>
</soapenv:Envelope>
```

- Fehlersemantiken: Was ist der Grund, wenn ich keine Antwort erhalte?
  - Auftrag angekommen? Vollständig bearbeitet?
  - Was passiert wenn ein Auftrag wiederholt wird?

- Terminierungssemantiken:
  - Auftragserteilung in der Regel synchron
  - es existieren aber auch asynchrone Aufträge

Auftragsorientierte Modelle: Implementierung

- Implementierung aufbauend auf send/receive

Ein Fileserver in Java

Server

```
1  try(ServerSocket ss = new ServerSocket(4242)) {
2    Socket s = ss.accept(); // warte auf Clients
3    // ...
4
5    String line = null;
6    if((line = socketReader.readLine()) != null) {
7      String command = lines.split(" "); // trenne beim Leerzeichen
8      String operation = command[0]; String path = command[1];
9      switch (operation.trim().toUpperCase()) {
10       case "GET":
11         String content = readFile(path);
12         socketWriter.write(content); break;
13       case "DELETE":
14         boolean ok = deleteFile(path);
15         socketWriter.write(String.valueOf(ok)); break;
16       // ...
17     } /*switch*/ } /*if*/  } /* try */
```

Erläuterungen zum Server

- Zeile 1 & 2: Serversocket erstellen, lauscht auf Port 4242, wartet blockierend bis sich ein Client verbindet
- Zeile 6: lies eine Zeile vom Client
- Zeile 7: unser Nachrichtenformat: **Operation <Leerzeichen> Dateipfad**
- Zeile 8ff: unterscheide Operationen und führe Aktionen aus; antworte dem Client entsprechend

Client

```
1  try(Socket socket = new Socket("localhost", 4242)) {
2    String command = args[0] + " " + args[1];
3
4    socketWriter.write(command);
5
6    String response;
7    while((response = socketReader.readLine()) != null) {
8      System.out.println(response);
9    }
10 }
```

- Zeile 1: erstelle Clientsocket, d.h. Verbindungsaufbau zum Server auf localhost auf Port 4242
- Zeile 2: lese Befehl und Dateipfad
- Zeile 4: sende Befehl als String an den Server
- Zeile 6ff: lese alle Antwortzeilen vom Server; Ausgabe auf dem Bildschirm

Auftragsorientierte Modelle

- Können benutzt werden, um einfache Protokolle zu implementieren
- Binär oder ASCII
  - auch Übertragung komplexer Objekte möglich
- gesendeter Befehl könnte einer Methode/Funktion auf dem Server entsprechen
  - es erfolgt eine Art entfernter Funktionsaufruf
  - RPC wird im nächsten Abschnitt behandelt
- Funktionalität kann über das Internet angeboten werden

  ⇒ Implementierung eines Webservices

Webservices - Allgemein

- WebService: Dienst, der über das Internet/WWW von Clients angesprochen werden kann
- typischerweise über HTTP
- Früher **SOAP**: Simple Object Access Protocol
  - Protokoll zum Austausch von Informationen in XML
  - Verzeichnisdienste zum Finden von Diensten, z.B. UDDI
- Heute **REST**

REST

- Die Grundidee von REST:
  - **REST**: Representional State Transfer
  - oftmals existiert ein HTTP Server / Anwendungsserver schon
  - Idee: Jede Ressource die vom Server angeboten wird, ist durch eine URI beschrieben/identifiziert
    - Datei, ein Eintrag in einer Datenbank, Tweet,...
  - Anlegen, Lesen, Verändern, Löschen (CRUD)
    - Art der Operation über HTTP Request-Typ festlegen (POST, GET, PUT, DELETE)
  - Unabhängigkeit von verwendeter Programmiersprache in Client und Server durch HTTP und Textformate

Anforderungen an Ressourcen

Anforderungen an Ressourcen nach Fielding:

1. Adressierbarkeit: jede Ressource muss über URI adressierbar sein (Achtung: URI != URL, Identifier vs. Locator)
2. Zustandslosigkeit: Kommunikation zwischen Client und Server hat keinen Zustand (Session/Cookie)
   - bei jeder Anfrage werden alle Informationen gesendet
3. Einheitliche Schnittstelle: über HTTP Standardmethoden auf Ressourcen zugreifen
4. Entkopplung von Ressource und Repräsentation: Ressourcen können in verschiedenen Formaten angeboten werden (JSON, XML,...)

HTTP Methoden für REST

- selbe URL mit verschiedenen Methoden aufrufbar
- Methode bestimmt ausgeführte Aktion auf dem Server

GET eine Ressource lese, Daten sollten nicht verändert werden
POST neue Ressource erstellen

- Die URI ist dem Anrufer zunächst unbekannt
- Der Server kann dem Anrufer die erzeugte URI in der Antwort mitteilen

PUT neue Ressource erstellen, oder existierende bearbeiten
DELETE zum Löschen von Ressourcen

REST - Beispiel

Spotify API

- **Authorization**-Header benötigt
- **id**: Spotify-ID eines Künstlers

Artists

Endpoints for retrieving information about one or more artists from the Spotify catalog.

Base URL: `https://api.spotify.com/v1`

METHOD	ENDPOINT	USAGE	RETURNS
GET	/v1/artists/{id}	Get an Artist	artist
GET	/v1/artists/{id}/albums	Get an Artist's Albums	albums
GET	/v1/artists/{id}/top-tracks	Get an Artist's Top Tracks	tracks
GET	/v1/artists/{id}/related-artists	Get an Artist's Related Artists	artists
GET	/v1/artists	Get Several Artists	artists

Implementierung von RESTful Webservices

- manuelle Implementierung recht aufwändig
  - unterscheiden von HTTP Methoden (GET, POST,...)
  - parsen/prüfen von URL Pfaden und Parametern
  - setzen von Antwortheadern & Kodierung in XML/JSON
- REST Frameworks erleichtern die Arbeit deutlich
  - JAX-RS Spezifikation für Java zur Erstellung von RESTful Services
    - Implementierung: Jersey: https://eclipse-ee4j.github.io/jersey/
    - Implementierung: Spring: https://spring.io/guides/gs/rest-service/
  - Microsofts **cpprestsdk** für C++ als Client-Bibliothek: https://github.com/Microsoft/cpprestsdk
- Beispiel: Jersey
- Definition einer einfachen Klasse
  - Einstellungen über Annotationen
  - Klasse muss als Servlet in einem Applicationserver ausgeführt werden

```
1  @Path("/files")
2  public class FileServer {
3    @GET
4    @Path("/{fname}")
5    @Produces(MediaType.APPLICATION_JSON)
6    public FileInfo getDetails(@PathParam("fname") String file) {
7      FileInfo infos = getFileInfos(file);
8      return infos;
9    }
10 }
```

Restful Webservice - Erläuterungen

- Zeile 1: dieser Dienst ist über den Pfad files erreichbar, z.B. http://localhost/files
- Zeile 3: die nachfolgende Methode soll HTTP GET Anfragen verarbeiten
- Zeile 4: die URL enthält den Dateinamen als Pfad-Bestandteil, z.B. http://localhost/files/myfile.txt
- Zeile 5: Hinweis an das Jersey-Framework das Ergebnis automatisch ins JSON Format umzuwandeln
- Zeile 6: normale Definition einer Methode & Mapping des Eingabeparameters auf den URL-Parameter
- Zeile 8: das infos Objekt vom Typ FileInfo wird automatisch als JSON repräsentiert

Aufruf von REST-Services

https://reques.in kostenloser Dienst zum Testen von REST-Clients

- Variante 1: telnet reques.in 80 ...
- Variante 2: Auf der Kommandozeile
  $ curl https://reqres.in/api/users/1

```
{"data":{
  "id":1, "email":"george.bluth@reqres.in",
  "first_name": "George", "last_name":"Bluth", ... }}
```

- Variante 3: Aufruf in einem Programm

HTTP GET Aufrufe in Java

In Java ab Version 11 eingebauter HTTP Client

```java
HttpClient httpClient = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()
  .GET()
  .uri(URI.create("https://reqres.in/api/users/1"))
  .build();

HttpResponse<String> response = httpClient.send(
  request, HttpResponse.BodyHandlers.ofString());

System.out.println(response.body());
```

HTTP POST in Java

```java
String data = "{ \"name\": \"morpheus\", \"job\": \"leader\" }";
HttpRequest postRequest = HttpRequest.newBuilder()
  .uri(URI.create("https://reqres.in/api/users"))
  .POST(HttpRequest.BodyPublishers.ofString(data))
  .header("Content-Type", "application/json").build();

HttpResponse<String> postResp = httpClient.send(
  postRequest, HttpResponse.BodyHandlers.ofString());
System.out.println(postResp.body());
```

Antwort:

```
{ "name": "morpheus", "job": "leader",
"id": "703", "createdAt": "2020-06-24T12:09:22.148Z"}
```

Eigentlich: JSON Ergebnis mit geeigneten Frameworks parsen und weiterverarbeiten

Zusammenfassung

- Auftragsorientierte Modelle nach dem Client-Server Prinzip
- WebServices bieten Dienste über das WWW an
- RESTful WebServices
  - jede Ressource hat eine URI
  - HTTP Methoden für Aktionen auf Ressourcen
  - unabhängig von Programmiersprachen

Funktionsaufrufbasierte Protokolle

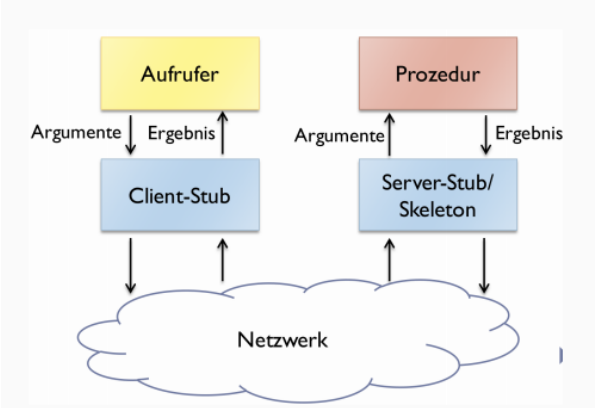
- Grundidee: Adaption von anwendungsnahen und unkomplizierten Kommunikationsparadigmen an Eigenschaften verteilter Systeme
- d.h., aus Aufrufen auf lokalen Prozeduren und Methoden werden Aufrufe entfernter Prozeduren und Methoden
- bekannt als:
  - RPC: Remote Procedure Calls
  - oder Java RMI: Remote Method Invocation
- Erlang und Java haben die Konzepte nativ implementiert, in C++ nur über zusätzliche Bibliotheken

Eigenschaften von Prozedurfernaufrufen

Aufruf und Ausführung in unterschiedlichen Umgebungen/Kontexten

- Programmiersprachen
- Namens- und Adressräume
- Betriebssystemkontext
- Hardwarekontext

Woher kennt der Aufrufer die Signatur der Prozedur auf dem Server? ⇒ Stubs



Remote Procedure Calls (RPC)

Stubs

Ein Stub hat verschiedene Aufgaben:

- wandelt lokalen Prozeduraufruf in Netzwerkfunktion um
- Ein- und Auspacken von Argumenten und Ergebnissen
- Anpassung von Datenrepräsentationen
- implementiert Übertragungsprotokoll über das Netzwerk

Der Server-Stub/Skeleton

- wartet auf Anfragen von Clients
- übernimmt sonst gleiche Aufgaben wie Client-Stub

RPC in Erlang

- Vordefiniertes Erlang-Modul für RPC

```
rpc:call(Node, Module, Func, Args)
rpc:call(Node, Module, Func, Args, Timeout)
```

- führt **Module:Func(Args)** auf **Node** aus
  - weitere Funktionen für asynchrone Aufrufe, Aufrufe von mehreren Servern

```
(node2@localhost)1> node().
node2@localhost
(node2@localhost)2> rpc:call(node1@localhost,erlang,node,[]).
node1@localhost
```

- andere Möglichkeit: eigene Funktionen über **register** anmelden (siehe Alternating Bit Protokoll)
- mit **whereis** PID von registrierten Erlang-Prozessen finden

RMI: Javas RPC Variante

- seit Java 5 nativ in die Sprache eingebaut - keine explizite Generierung von Stubs notwendig
- Aufruf von Objektmethoden:
  - Server: Objekte mit Zuständen
  - Objekte können als Methoden-Argumente und Ergebnisse verwendet werden
- entfernt aufrufbare Methoden in einem Java-interface definieren
  - abgeleitet von **java.rmi.Remote**

RMI - Schnittstelle für entfernte Objekte

```java
package fileserver;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface FileService extends Remote {
    FileInfo getFileInfos(String filename) throws
    ↪ RemoteException;
}

class FileInfo implements java.io.Serializable {
  String name; long size; String owner;
}
```

RMI: Server

Server-Objekt muss:

- Remote-Schnittstelle implementieren
- im RMI-Laufzeitsystem bekannt gemacht werden
- im Namensverzeichnis registriert werden

Server-Objekt anlegen:

```java
package fileserver;
public class FileServer implements FileService {
  @Override
  public FileInfo getFileInfos(String fileName) throws
  ↪ RemoteException {
    return ...
  }
}
```

RMI: Serverobjekt registrieren

```java
public static void main(String args[]) {
  try {
    // Server-Objekt erzeugen
    FileServer srv = new FileServer();
    // .. exportieren
    FileService stub = (FileService)
    ↪ UnicastRemoteObject.exportObject(srv, 0);
    // ... und registrieren
    Registry registry = LocateRegistry.getRegistry();
    registry.bind("MyFileServer", stub);
    } catch (Exception e) { ...}
}
```

RMI - Client

- über Namensdienst Server-Objekt finden
- Stub erzeugen (erfolgt automatisch von JVM)
- Methode auf dem Server-Objekt aufrufen

```java
String host = args[0];
Registry registry = LocateRegistry.getRegistry(host);
FileService stub = (FileService)
↪ registry.lookup("MyFileServer");
FileInfo infos = stub.getFileInfos("myfile.txt");
System.out.println("Details: " + infos.toString());
```

RMI - Ablauf

Starten des Namensdienstes (registry):

```
> rmiregistry &
```

Starten des Servers

```
> java -Djava.rmi.server.codebase=file:classDir/
↪ fileserver.FileServer &
```

Starten des Clients

```
> java -Djava.rmi.server.codebase=file:classDir/
↪ fileserver.Client  hostname
```

Interoperabilität von RPC

Problem von Erlang, Java RMI, etc.:

- an Programmiersprache gebunden → verschiedene Systeme können nicht miteinander verbunden werden

Lösungsansätze:

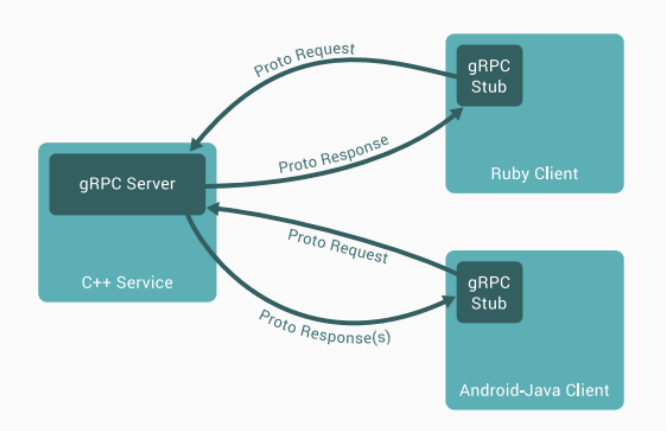
- **XML-RPC, JSON-RPC**: kodiere alle zum Aufruf nötigen Informationen als XML bzw. JSON
  - HTTP zur Übertragung
  -
```
{ "jsonrpc": "2.0", "method": "getFileInfos",
↪  "params": ["myfile.txt"], "id": 1}
```
  - **id** für die Zuordnung von Antworten zu Anfragen
- **gRPC**: Code-Generierung für Server, Stubs und ausgetauschte Daten

gRPC

- initiiert von Google im Jahr 2015
- plattformunabhängige Beschreibung von Daten und Diensten

gRPC

- ProtoBuf Dateien übersetzt in konkrete Programmiersprache
- C/C++, Java, Python, GO, uvm.



gRPC: Dienstbeschreibung

fileservice.proto

```
message FileInfo {
  string name = 1;
  uint64 size = 2;
  string owner = 3;
}

message Request {
  string fname = 1;
}

service FileService {
  rpc GetDetail(Request) returns (FileInfo);
}
```

gRPC: Dienstbeschreibung - Erläuterungen

- Datenklasse **FileInfo** mit drei Attributen, Zahlen geben Reihenfolge bei Serialisierung an
- **Request:** Service darf nur eine Eingabe- und Ausgabe-Message haben
  - extra Typen für Parameter und Ergebnis erlauben einfache Erweiterung ohne Signaturen zu ändern
- **FileService:** Klasse die unseren Dienst darstellt; enthält eine Methode **GetDetail** die von Clients aufgerufen werden kann

gRPC: Dienstbeschreibung

- *.proto Dateien werden mittels protoc Compiler in die Zielsprache übersetzt

```
> protoc -I ../protos --grpc_out=.
↪  --plugin=protoc-gen-grpc=grpc_cpp_plugin
↪  ../protos/fileservice.proto
> protoc -I ../protos --cpp_out=.
↪  ../protos/fileservice.proto
```

- erzuegt C++ Dateien für messages sowie Service-Klassen (FileService)
- Klasse **FileService** enthält generierten Stub und Methoden für den Server zum überschreiben

gRPC: Server erzeugen

```
class FileServiceImpl final : public FileService::Service {
  Status GetDetail(ServerContext* context, const Request* req,
  ↪  FileInfo* result) override {

  // Werte bestimmen ...
  string fName = req->fname();
  string owner = getOwner(fName); // dummy
  uint64 size = getFileSize(fName); // dummy

  // ... und im Ergebnis-Objekt setzen
  result->set_owner(owner);
  result->set_size(size);
  result->set_name(fName);
  return Status::OK;
  }
}
```

gRPC: Server starten

```
void RunServer() {
  std::string server_address("0.0.0.0:50051");
  // Instanz unseres Dienstes anlegen
  FileServiceImpl service();

  ServerBuilder builder;
  // lausche auf gegebenem Port
  builder.AddListeningPort(server_address,
  ↪  grpc::InsecureServerCredentials());
  // unseren Dienst registrieren
  builder.RegisterService(&service);
  // starte RPC Server
  std::unique_ptr<Server> server(builder.BuildAndStart());
  server->Wait();
}
```

gRPC: Client

```
class FileServiceClient {
private:
  std::unique_ptr<FileService::Stub> stub_;
public:
  FileServiceClient(): stub_(FileService::NewStub(
  ↪  grpc::CreateChannel("localhost:50051",
  ↪  grpc::InsecureChannelCredentials()))) {}

  void GetFileInfo(const string& file) {
    ClientContext context; FileInfo* info;
    Request r;  r.set_fname(file);

    Status status = stub_->GetDetail(&context, r, info);
    if (!status.ok()) {
      std::cout << "GetDetail rpc failed." << std::endl;
      return false;
    } else { /* do something with info */ }
  } /* GetFileInfo */
} /* class */
```

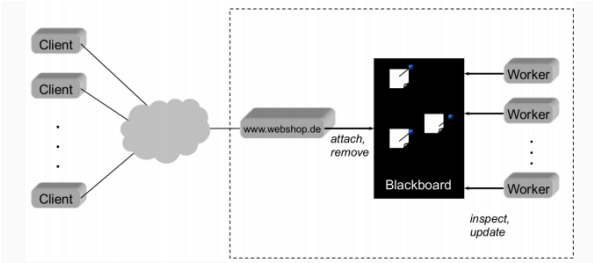
Zusammenfassung

- Funktionsaufrufbasierte Modelle:
  Prozedur/Funktion/Methode auf einem entfernten Host aufrufen
- Stubs kapseln Kommunikationsoperationen von Anwendung
- einheitliches Dateiformat notwendig
- Java RMI
- gRPC für Interoperabilität verschiedener Plattformen/Sprachen

Weitere Kommunikationsmodelle und Cloud-Computing

Blackboards

- das 'schwarze Brett': Teilnehmer hinterlegen
  - Gesuche und Angebote
  - Aufträge und Ergebnisse
- zeitliche und räumliche Entkopplung autonomer und anonymer Komponenten
- implementiert zum Beispiel in JavaSpaces



Blackboards: Modell-Sicht

- Rollenmodell:
  - Spezialist (Worker): aktiver Nutzer (Anbieter, Suchender, Bearbeiter)
  - Moderator: optionale Kontrollkomponente
    - delegiert Arbeit nach bestimmter Strategie an Spezialisten

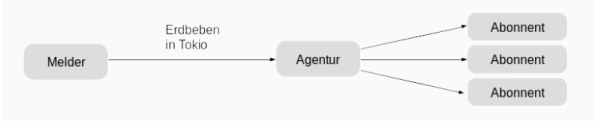
- Datenmodell:
  - globaler virtueller persistenter Speicher
  - allgemein: Tupel $< Typ, Name, Wert >$
  - Methoden zum Lesen, Schreiben, Aktualisieren, Löschen
- Fehler- und Terminierungssemantiken:
  - als verlässliche und unverlässliche Variante umsetzbar
  - Kommunikationsoperationen i.d.R asynchron

Blackboards: Vor- und Nachteile

- Vorteile
  - Offenheit: neue Spezialistentypen möglich
  - gute Lastskalierbarkeit: mehr Spezialisten hinzufügen
  - Interoperabilität durch gemeinsame Tupeldefinition
  - Anonymität + Kommunikationskontrolle
  - Fehlertoleranz: redundante Spezialisten
  - Nutzung von Nebenläufigkeit für Spezialisten
- Nachteile
  - Synchronisation der Schreibzugriffe am Board
  - Moderator potentieller Engpass
  - Board und Moderator sind potentielle Single Point of Failures
  - erschwerte Testbarkeit durch Asynchronität, Nichtdeterminismus

Ereignisbasierte Modelle

- Asynchronität von Blackboards nicht immer hilfreich
  - Niemand weiß, wann etwas an das Board gepinnt wird
  - warten auf bestimmte Tupel (nur Angebote von Mountainbikes) ist umständlich
- Börsen, Nachrichtenagenturen, Replikationssysteme sind an bestimmten Themen/Ereignissen interessiert
- Ereignisbasierte Modelle:
  - nutzen ebenfalls autonome und anonyme Komponenten
  - verfügen zusätzlich über asynchrone Benachrichtigung über Veränderung



Ereignisbasierte Modelle: Modell-Sicht

- Rollenmodell:
  - Herausgeber (Publisher): registriert Abonnenten und ihre Interesse an bestimmten Themen, meldet Ereignisse an Abonnenten
  - Abonnent (Subscriber): abonniert Themen bei Herausgebern, erhält passende Ereignisse zugeteilt
  - Agentur: optionale Abonnementverwaltung, Anonymisierung, zeitliche und räumliche Entkopplung
- Datenmodell:
  - allgemein: Tupel, z.B. $< Ereignistyp, Name, Wert >$
  - aber auch problemspezifischer Ausprägungen
  - Methoden: Melden (notify/publish), Abonnieren (subscribe/unsubscribe)
  - Ankündigen/Aktualisieren von Ereignisreportoires (advertise/unadvertise)
- Fehler- und Terminierungssemantiken:
  - Ankündigungen/Aktualisieren i.d.R. verlässlich, daher synchron
  - Melden unverlässlich, da Abonnenten ausgefallen sein können, daher asynchron

Ereignisbasierte Modelle: Vor- und Nachteile

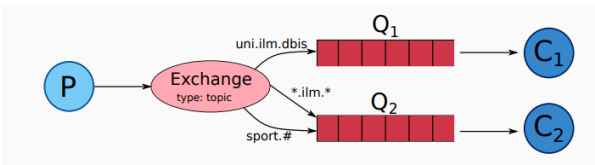
- Vorteile:
  - wie Blackboards, plus
  - direkte Benachrichtigungen bei gesuchten Ereignissen/Themen
- Nachteile:
  - erhebliche Management-Last beim Vermittler
    - potentieller Engpass, SPoF - es sei denn...

  Variante: mehrere Vermittler-Instanzen parallel und verteilt in einer Clusterumgebung
  - RabbitMQ, Apache Kafka, Apache ActiveMQ

Beispiel: RabbitMQ

RabbitMQ

- ist ein Message-Broker, dh. Vermittlersystem für Nachrichten zwischen Publishern und Subscriber
- implementiert in Erlang
  - Topic ist eine Liste von Wörtern, durch Punkte getrennt
  - Wildcards möglich, * ersetzt genau ein Wort, # ersetzt mehrere Wörter



RabbitMQ Publish

```
1  import com.rabbitmq.client.*;
2
3  public class RMQTest {
4    private static final String EXCHANGE_NAME = "rmq_test";
5
6    public static void main(String[] args) throws Exception {
7      ConnectionFactory factory = new ConnectionFactory();
8      factory.setHost("localhost");
9      try (Connection connection = factory.newConnection();
10       Channel channel = connection.createChannel()) {
11       channel.exchangeDeclare(EXCHANGE_NAME, "topic");
12
13       channel.basicPublish(EXCHANGE_NAME, "uni.ilm.dbis",
14                            null, "Hallo Welt".getBytes("UTF-8"));
15     }
16   }
17 }
```

- Zeilen 7,8: **ConnectionFactory** zum Handling von Verbindungen, im Beispiel nur auf dem lokalen Host
- Zeilen 9,10: erstelle neue Verbindungen und einen Channel
- Zeile 11: Nachrichten sollen anhand des Topics zugestellt werden
- Zeile 13: veröffentliche eine Nachricht: sende an den Exchange "rmq_testëine Nachricht mit dem Topic **uni.ilm.dbis** und dem Inhalt "Hallo Welt".
  - **null** hier für eventuelle weitere Einstellungen

RabbitMQ Subscribe

```
1  // wie zuvor Channel erstellen
2  // ...
3  channel.exchangeDeclare(EXCHANGE_NAME, "topic");
4  String queueName = channel.queueDeclare().getQueue();
5  channel.queueBind(queueName, EXCHANGE_NAME, "*.ilm.*");
6  channel.queueBind(queueName, EXCHANGE_NAME, "sport.#");
7
8  DeliverCallback callback = (consumerTag, delivery) -> {
9    String message = new String(delivery.getBody(), "UTF-8");
10   String key = delivery.getEnvelope().getRoutingKey();
11   System.out.println("Topic = "+key);
12   System.out.println("Nachricht: "+message);
13 };
14
15 channel.basicConsume(queueName, true,
16   callback, consumerTag -> { });
```

- Zeilen 1&2: Connection und Channel erstellen, siehe vorherige Bilder
- Zeilen 4-6: Queue erzeugen und auf Topics registrieren
- Zeilen 8-13: Java-Lambda Funktion anlegen, wird für jede eintreffende Nachricht aufgerufen
- Zeilen 15&16: Warte auf der erzeugten Queue auf Nachrichten
  - durch **true** Parameter wird ankommende Nachricht mit **ACK** quittiert
  - zweite anonyme Funktion für Handling von Abbrüchen

Cloud Computing

- Verteilte Systeme benötigen oftmals viele Knoten
- Administration der Systeme erfordert viel Aufwand
- Hardware, Betriebssystem und Anwendungssoftware veralten schnell
- hohe Kosten, aber Systeme oftmals nicht voll ausgelastet
- Grundidee: einmal eingerichtete Hardware, durch Virtualisierung mehreren Kunden zugänglich machen
- Zugriff über das Internet

Arten und Ziele

Arten von Clouds

- Public Cloud: für jeden zugängliche Ressourcen
- Private Cloud: z.B. in Unternehmen im eigenen Netzwerk
- Community Cloud: Cloud-Umgebung wird von mehreren (festgelegten) Gruppen/Organisationen geteilt
- Hybrid Cloud: private Clouds, aber für Skalierbarkeit auch Ressourcen von Public Clouds nutzen

Ziele

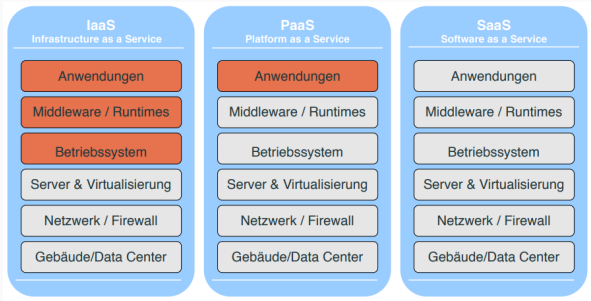
- Auslastung der physischen Hardware durch Virtualisierung $\Rightarrow$ Umsatzsteigerung
- für Kunden
  - hohe Verfügbarkeit, Skalierbarkeit
  - keine Anschaffungs-, geringere Administrationskosten

Geschätsmodell

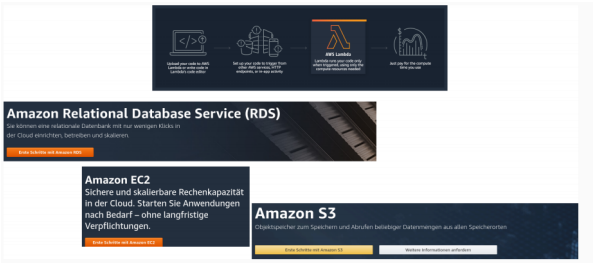
Cloud Computing als Geschäftsmodell:

- Cloudprovider stellen Ressourcen bereit:
  - persistente Speicher (HDDs, SSDs)
  - CPUs/RAM
  - spezialisierte Hardware (FPGAs, GPUs)
  - Software
- verschiedene Maschinenvarianten
  - Datenverarbeitungsoptimiert, Arbeitsspeicheroptimiert, uvm.
- Kunden starten dynamisch Instanzen der Maschinen/Software
  - Bezahlung nur für genutzte Zeit ("pay-as-you-go")

Architekturen



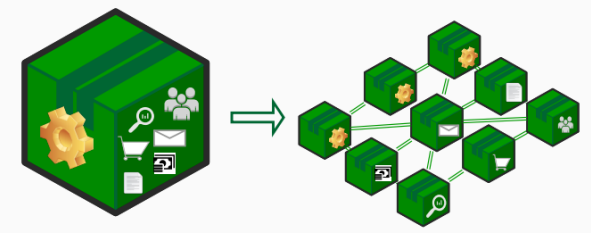
Amazon AWS Produkte



Hinweis: Bei AWS, Google, Azure viele Dienste auch (dauerhaft) kostenlos nutzbar!

Microservices

- typische Anwendung hat verschiedene Komponenten (Suche, Buchungen/Warenkorb, Bezahlsystem, Bewertungssystem)
- monolithisch: alle Komponenten in einer Anwendung
  - bei Bugfix/Update gesamte Anwendung aktualisieren und neu starten
- Ansatz Microservice: jede Komponente oder Funktionalität einer Anwendung als unabhängigen Dienst, läuft 24/7
- Dienste mit unterschiedlichen Sprachen und Technologien umsetzbar
- unterstützt durch Virtualisierung und Cloud-Angebote, z.B. Serverless Computing

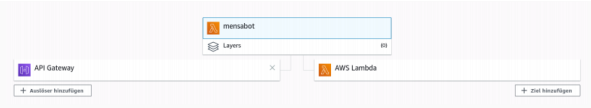


Serverless Computing

- wörtlich: 'ohne Server'
  - aber in Wirklichkeit: 'es ist nicht dein Server'
  - Idee: Entwickler konzentriert sich auf (kleine Funktion)
    - Function-as-a-Service (FaaS)
  - Laufzeitumgebung bzw. Cloud-Anbieter stellen Server und Konfiguration bereit
  - Ausführung der Funktion nach Bedarf
  - verschiedene Programmiersprachen unterstützt

Aufgaben des Entwicklers

- schreibt nur auszuführenden Code
- konfiguriert Triggerereignisse
  - REST Aufruf, neuer Eintrag in DB, Monitoring-Ereignis einer anderen Anwendung
  - ⇒ Funktion läuft nur nach Eintreten des Triggers
  - optional: verknüpft z.B. weitere Funktion, die nach Beendigung ausgeführt wird



AWS Lambda Designer: HTTP Trigger, auszuführende Funktion und Nachfolgefunktion.

Vergleich Microservice vs. Serverless

Micorservice:

- Anwendung wird durch Dienste strukturiert
- Dienst hat einen Service Contract (Schnittstellendefinition)
  - z.B. Portnummer, Antwortzeitgarantien,
  - unterstütze Anfragen und resultierende Antworten
- läuft kontinuierlich in einem eigenen (virtuellen) Knoten

Serverless

- einzelne Funktionen; kleiner als ein ganzer Dienst
- wird nur nach Trigger ausgeführt
- kurze Lebenszeit; oft begrenzt durch den Anbieter

Mittlerweile auch Serverless Microservice: Microservice nicht immer ausführen, sondern nach Trigger

AWS Lambda: Java Beispiel

```java
import com.amazonaws.services.lambda.runtime.RequestHandler;
public class HandlerUserDetails implements
  ↪ RequestHandler<String, UserInfo> {
  Gson gson = new GsonBuilder().setPrettyPrinting().create();
  @Override
  public UserInfo handleRequest(String ev, Context ctx) {
    // ev is JSON string, contains query parameters

    String username = ...; // get from ev
    Userinfo info = getUserInfo(username);
    return info;
  } /* method */ } /* class */
```

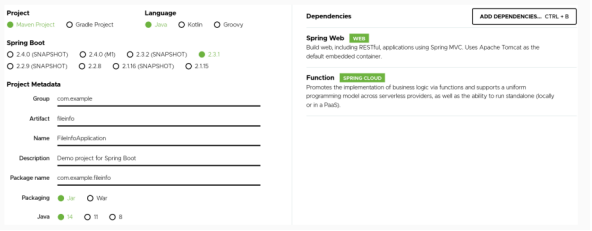
RequestHandler als Einstiegspunkt für Lambda-Funktionen
handleRequest wird von der Lambda-Umgebung aufgerufen

Spring Functions

- neben AWS viele weitere Functions-Angebote
- Anbieter-spezifische API macht Migration schwierig
- zusätzliche Frameworks mit Plugins für konkrete Anbieter
  - serverless.com
  - Spring Cloud Functions

Spring Cloud Function: Projekt anlegen

https://start.spring.io



Projekt generieren, herunterladen und entpacken

Spring Cloud Functions: Beispiel

src/main/java/com/example/fileinfo/FileInfoApplication.java:

```java
@SpringBootApplication
public class FileInfoApplication {
  public static void main(String[] args) {
    SpringApplication.run(FileInfoApplication.class, args);
  }

  @Bean
  public Function<String, FileInfo> info() {
    return name -> {
      FileInfo i = getFileInfo(name); // dummy
      return i;
    };
  }
}
```

- Zeile 1: für Spring Boot, diese Klasse enthält Definitionen für Dienste
- Zeilen 3-5: Ausführen der Klasse als Spring Applikation
- Zeile 7: Ergebnis der Methode als Bean behandeln
  - Bean Objekt wird von Spring verwaltet, als serverless Funktion
  - durch **Spring Web** Funktionsname = REST Pfad
- Zeile 8ff: Methode **info** gibt eine Java Lambda-Funktion zurück
  - Eingabe vom Typ **String**
  - Ergebnis vom Typ **FileInfo**

Aufruf der Spring Cloud Funktion

1. Projekt kompilieren

```
./mvnw clean install
```

2. Starten

```
java - jar target/fileinfo-0.0.1-SNAPSHOT.jar
```

3. Aufruf als normaler REST call

```
\newline curl localhost:8080/info -d myfile.txt
```

4. Antwort

```
\newline \{"name":"myfile.txt","size":1234\}
```

Zusammenfassung

- Cloud-Angebote auf verschiedenen Ebenen (IaaS, PaaS, SaaS)
- Cloud & Virtualisierung verringern Anschaffungs- und Administrationskosten
- Microservices: monolithische Anwendung in kleinere Dienste zerlegen
- Serverless: nur noch Funktion implementieren (FaaS)