Programmierparadigmen

Kapitel 3a

Einführung in Funktionale Programmierung

Einleitung

- Bisher Java & C++: objektorientiert, imperativ, Elemente von applikativ
- Sprachen enthalten Mechanismen zur Fehlervermeidung
  - Unit-Tests, Exceptions, Starke Typisierung...
- Aber Leitfrage: Wie kann komplexere Software nachweisbar korrekt konstruiert werden?
  - Stärker!
- Unterschiedliche Teilfragestellungen, z.B.:
  - Keine Laufzeitfehler: Zeiger ok? Array-Grenzen eingehalten? Keine Division durch 0? Keine Wurzel aus negativen Zahlen?
  - Keine Fehler durch parallele Ausführung: Protokoll ok?
  - Model Checking: Entspricht das Programmverhalten einem formalen Modell

Einleitung – Schwierigkeiten bei Java und C++

- In C++ & Java sehr schwierig zu beantworten!
  - Bei C++ kein Speicherschutz
  - Komplexe Sprachelemente
  - Möglichkeit Threads komplex interagieren zu lassen
  - Schwierigkeiten durch imperatives Vorgehen
- Beispiele für Probleme durch imperatives Vorgehen
  - Variablen können über die Zeit unterschiedliche Werte annehmen
    - Worauf beziehen in Beweis?
    - Benötigt Überführen in Static Single Assingment Form
  - Funktionen können Variablen in verschiedenen Teilen des Programms ändern
    - Wie Zustandsänderungen in Beweis gut fassen?
  - Schleifeninvarianten teils schwer zu formulieren

Einleitung – Kurze Erinnerung

- Beispiel Summe der Quadratzahlen: Erfordert Vor- und Nachbedingung & Schleifeninvariante

```
int sumSq(int k) {
    int s = 0;
    // pre: s == 0 && k >= 0
    // inv: s == summe der i ersten quadratzahlen
    for(int i = 1; i < k; ++i) {
        s += i * i;
    }
    // post: s == summe der k ersten quadratzahlen
    return s;
}
```

- Je nach Schleife möglicherweise sehr komplex aufzuschreiben
- Mehrere Beweisschritte pre → inv, inv → inv‘, inv → post
- Möglicherweise rekursiv besser? → Braucht andere Sprachen

Applikative Algorithmen

Grundidee:

- Definition zusammengesetzter Funktionen durch Terme:

  $f\,(x) = 5x + 1$

- Unbestimmte:
  - $x, y, z, \ldots$ vom Typ **`int`**
  - $q, p, r, \ldots$ vom Typ **`bool`**
- Terme mit Unbestimmten:
  - Terme vom Typ **`int`**

    $x,\ x - 2,\ 2x + 1,\ (x + 1)(y - 1)$
- Terme vom Typ **`bool`**

  $p,\ p \wedge \texttt{true},\ (p \vee \texttt{true}) \Rightarrow (q \vee \texttt{false})$

Funktionsdefinitionen

Sind $v_1, \ldots, v_n$ Unbestimmte vom Typ $T_1, .., T_n$ (**`bool`** oder **`int`**) und ist $t\,(v_1, \ldots, v_n)$ ein Term, so heißt

$$f\,(v_1, \ldots, v_n) = t\,(v_1, \ldots, v_n)$$

eine **Funktionsdefinition** vom Typ T . T ist dabei der *Typ* des Terms.

- f : *Funktionsname*
- $v_1, \ldots, v_n$: *formale Parameter*
- $t\,(v_1, \ldots, v_n)$: *Funktionsausdruck*

Erweiterung der Funktionsdefinition

- Erweiterung der Definition von Termen
- Neu: **Aufrufe** definierter Funktionen sind Terme
- Beispiel für Einsatz in Funktionsdefinitionen:

```
f(x, y)  =  if g(x, y)  then h(x + y)
                        else h(x - y) fi
g(x, y)  =  (x = y) V odd(y)
h(x)     =  j(x + 1) · j(x - 1)
j(x)     =  2x - 3
```

Beispiel für Auswertung

```
f (1, 2) ↦   if g(1, 2) then h(1 + 2) else h(1 - 2) fi
         ↦   if 1 = 2 V odd(2) then h(1 + 2) else h(1 - 2) fi
         ↦   if false then h(1 + 2) else h(1 - 2) fi
         ↦*  h(1 - 2)
         ↦   h( - 1)
         ↦   j( - 1 + 1) · j( - 1 - 1)
         ↦   j(0) · j( - 1 - 1)
         ↦   j(0) · j( - 2)
         ↦   (2 · 0 - 3) · j( - 2)
         ↦*  ( - 3) · ( - 7)
         ↦   21
```

Rekursive Funktionsdefinitionen

```
f (x, y) =  if x = 0 then y else (
               if x > 0 then f (x - 1, y) + 1
               else - f ( - x, - y) fi) fi
```

Auswertungen

$f\,(0, y)$	$\mapsto y$ für alle y
$f\,(1, y)$	$\mapsto f\,(0, y) + 1 \mapsto y + 1$
$f\,(2, y)$	$\mapsto f\,(1, y) + 1 \mapsto (y + 1) + 1 \mapsto y + 2$
...	
$f\,(n, y)$	$\mapsto n + y$ für alle $n \in$ `int`, $n > 0$
$f\,(-1, y)$	$\mapsto - f\,(1, - y) \mapsto - (1 - y) \mapsto y - 1$
...	
$f\,(x, y)$	$\mapsto x + y$ für alle x, $y \in$ `int`

Applikative Algorithmen

Ein **applikativer Algorithmus** ist eine Menge von Funktionsdefinitionen

$$
\begin{aligned}
f_1(v_{1,1}, \ldots, v_{1,n_1}) &= t_1(v_{1,1}, \ldots, v_{1,n_1}),\\
&\vdots\\
f_m(v_{m,1}, \ldots, v_{m,n_m}) &= t_m(v_{m,1}, \ldots, v_{m,n_m}).
\end{aligned}
$$

Die erste Funktion f_1 wird wie beschrieben ausgewertet und ist die Bedeutung (Semantik) des Algorithmus.

Soweit unser Wissen aus „Algorithmen und Programmierung“. In diesem Kapitel werden wir das Thema Funktionale Programmierung erheblich vertiefen.

Kategorien der funktionalen Sprachen (1)

- Kategorisierung nach unterschiedlichen Kriterien
- 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

(Danksagung: Diese und einige folgende Folien gehen zurück auf Material von Günter Hübel, der eine gleichnamige Vorlesung bis 2010 gehalten hat.)

Kategorien der funktionalen Sprachen (2)

- 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 „Sharing“ (im Unterschied zur Normalform-Reduktion – dort werden gleiche Ausdrücke immer wieder erneut berechnet).

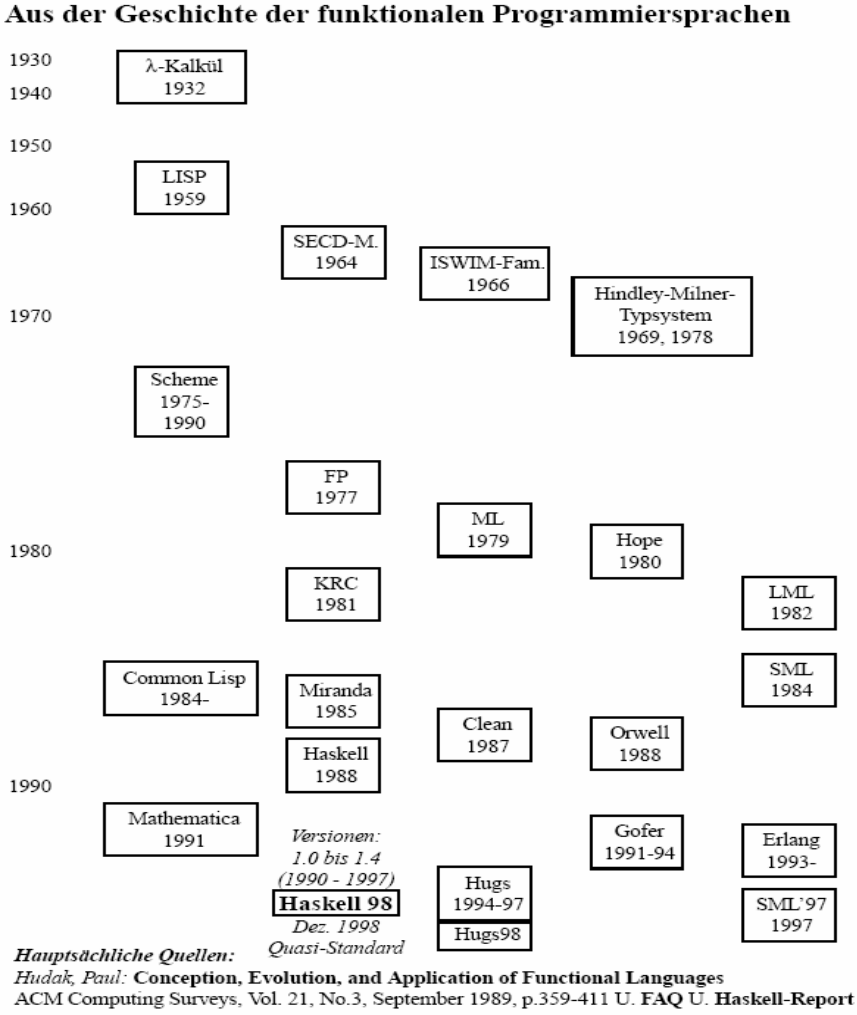
Kategorien der funktionalen Sprachen (3)

- 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)

Klassifikation funktionaler Sprachen

		Strikte Auswertung	Bedarfsauswertung
Dynamisch typisiert	Erster Ordnung	Mathematica, Erlang (?)	
	Höherer Ordnung	LISP, Scheme, APL, FP	SASL
Statisch typisiert	Erster Ordnung	SISAL	Id
	Höherer Ordnung	ML, SML, Hope	Miranda, Haskell 98

(Quelle: Odersky, M.: Funktionale Programmierung. In: Informatik-Handbuch, 1997, 1999, 2002)



Die funktionale Programmiersprache Erlang

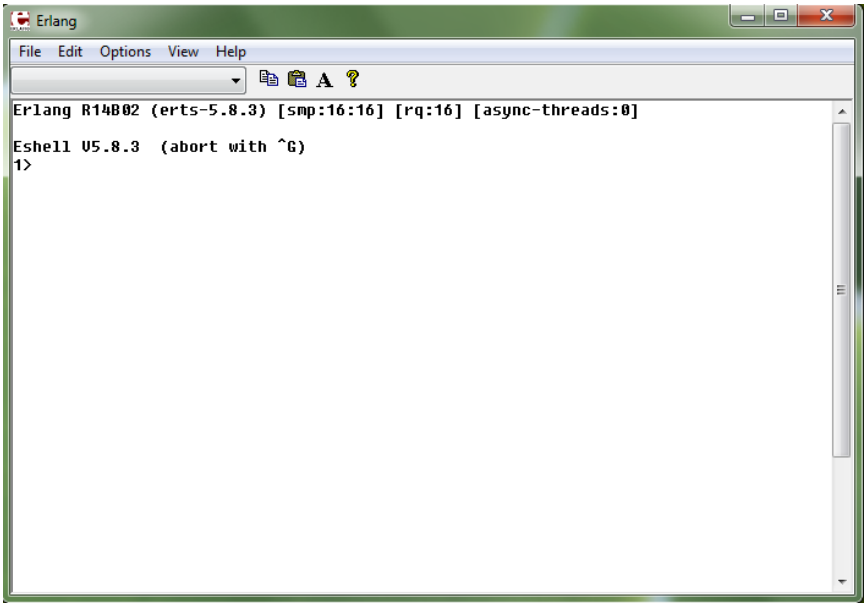
- Entwickelt ab der zweiten Hälfte der 1980er Jahre im Ericsson Computer Science Laboratory (CSLab, Schweden)
- Ziel war, eine einfache, effiziente und nicht zu umfangreiche Sprache, die sich gut zur Programmierung robuster, großer und nebenläufiger Anwendungen für den industriellen Einsatz eignet.
- Erste Version einer Erlang-Umgebung entstand 1987 auf der Grundlage von Prolog
- Später wurden Bytecode-Übersetzer und abstrakte Maschinen geschaffen.
- Größtes industrielles in Erlang programmiertes Produkt: Ericsson AXD301-Switching-System (beruht auf Open Telecom Platform, OTP, einer Weiterentwicklung der bisherigen Erlang-Bibliotheken)
- 1998 zog sich Ericsson aus der Entwicklung der Sprache zurück und Erlang wurde ein Open Source Projekt

Literatur zu Erlang

- Joe Armstrong. *Programming Erlang – Software for a Concurrent World.* http://pragprog.com/titles/jaerlang/programming-erlang

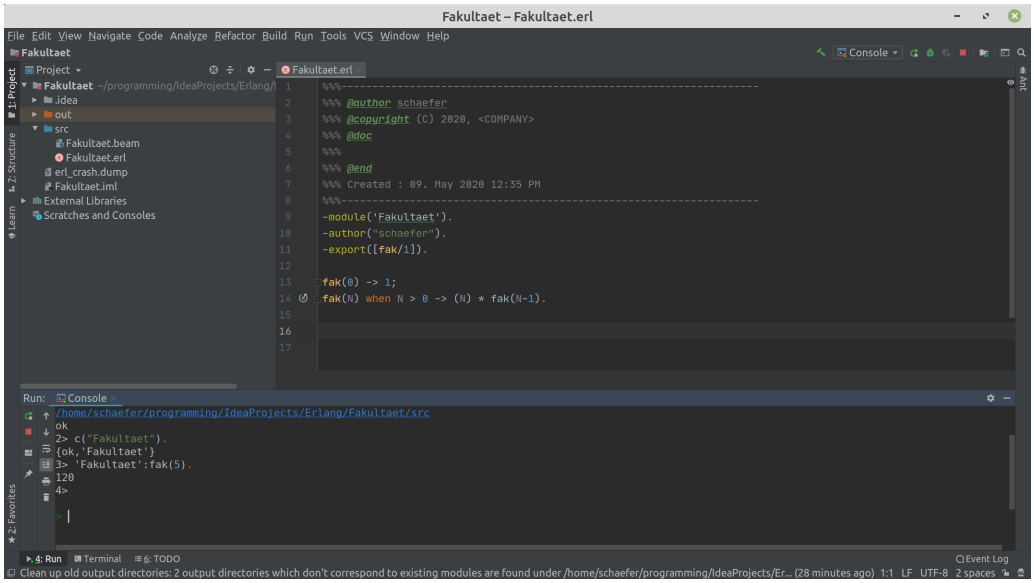
Start des Erlang Interpreters (1)

- Download bei: http://www.erlang.org/download.html
- Unter Windows gibt es eine komfortable Version der Shell: werl.exe
- Unter UNIX einfach „erl“ ausführen



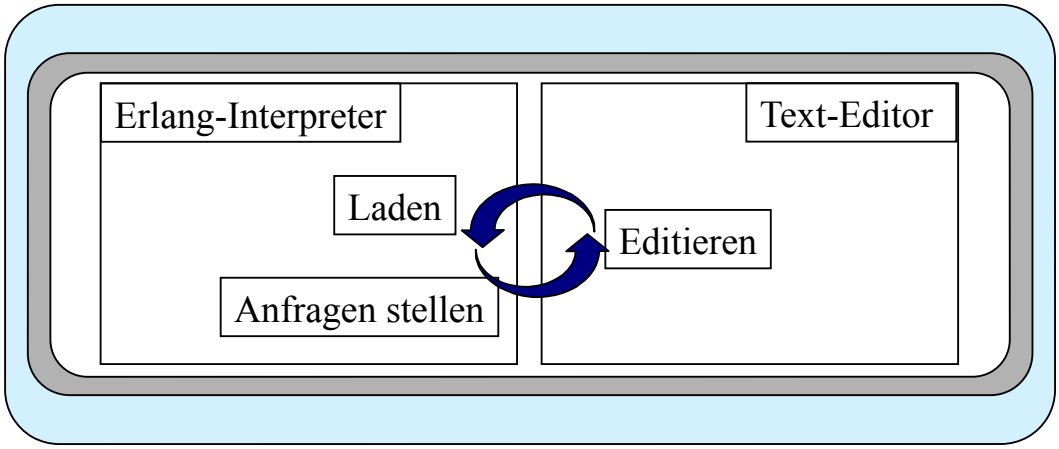
Start des Erlang Interpreters (2)

- Alternativ kann auch das Erlang-Plugin in IntelliJ Idea installiert werden (unter Settings – Plugins – im Suchfeld Erlang eingeben)
- Anlegen einer Run-Configuration mit einer Console (incl. Start im Ordner „src“ eines Projekts) ermöglicht flexibles Übersetzen und Anfragen



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



Ein erstes Beispiel: Die Fakultätsfunktion (1)

- Modul fakultaet.erl:

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

- Laden in den Interpreter mittels:

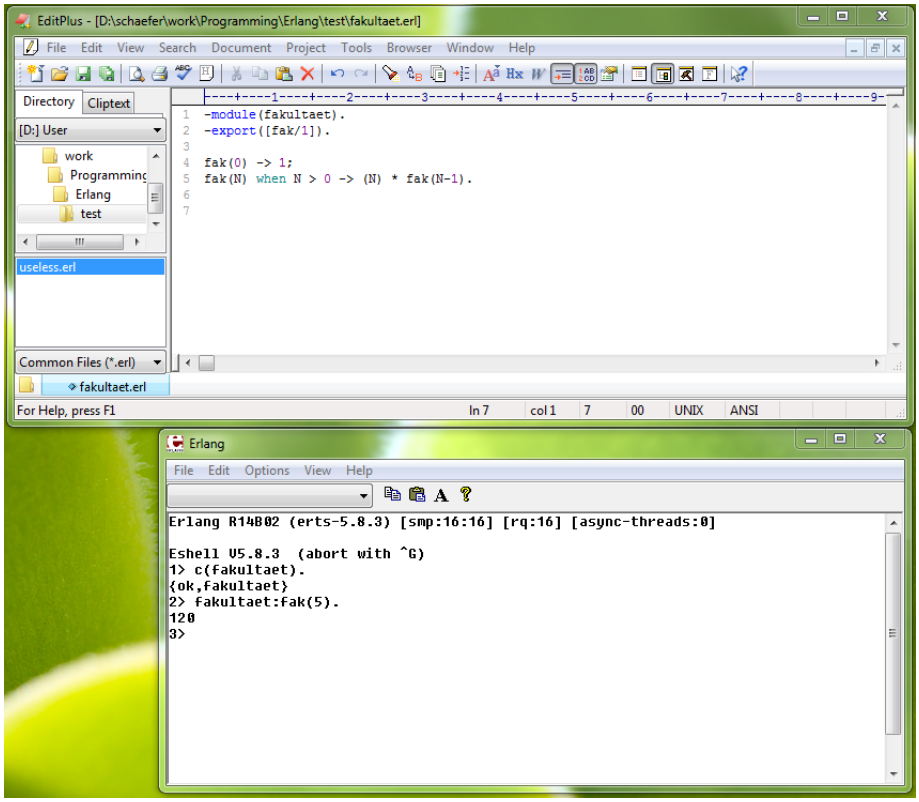
```
c(fakultaet).
```

- Testen der Funktion, z.B. mit:

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

(ergibt 120)

Ein erstes Beispiel: Die Fakultätsfunktion (2)



Elemente von Erlang (1)

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

- Gleitkommazahlen (Floats):
  - `17.368`
  - `-56.654`
  - `12.34E-10`.

(Die folgende Einführung folgt: http://www.erlang.org/course/course.html)

Elemente von Erlang (2)

- Atome (Atoms):
  - `abcef`
  - `start_with_a_lower_case_letter`
  - `'Blanks can be quoted'`
  - `'Anything inside quotes \n\012'`
- 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`

Elemente von Erlang (3)

- 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 “Dingen” speichern
  - Tupel beliebiger Größe sind zulässig
  - Kommentare werden in Erlang mit `%` eingeleitet und erstrecken sich dann bis zum Zeilenende

Elemente von Erlang (4)

- Listen:
  - `[123, xyz]`
  - `[123, def, abc]`
  - ```
    [{person, 'Joe', 'Armstrong'},
     {person, 'Robert', 'Virding'},
     {person, 'Mike', 'Williams'}
    ]
    ```
  - `"abcdefgh"` wird 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
  - `"..."` ist eine Kurzform für die Liste der Ganzzahlen, die die ASCII-Codes der Zeichen innerhalb der Anführungszeichen repräsentieren

    ```

## Elemente von Erlang (5)

- 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).`

## Elemente von Erlang (6)

- Komplexe Datenstrukturen:

```
[{{person,'Joe', 'Armstrong'},
  {telephoneNumber, [3,5,9,7]},
  {shoeSize, 42},
  {pets, [{cat, tubby},{cat, tiger}]},
  {children,[{thomas, 5},{claire,1}]}},
 {{person,'Mike','Williams'},
  {shoeSize,41},
  {likes,[boats, beer]},
  ... }]
```

- Erläuterungen:
  - Beliebig komplexe Strukturen können erzeugt werden
  - Datenstrukturen können durch einfaches Hinschreiben erzeugt werden (keine explizite Speicherbelegung oder -freigabe)
  - Datenstrukturen können gebundene Variablen enthalten

Elemente von Erlang (7)

- 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
- Erläuterungen:
  - „Pattern Matching“, zu Deutsch „Mustervergleich“, spielt eine zentrale Rolle bei der Auswahl der „richtigen“ Anweisungsfolge für einen konkreten Funktionsaufruf und dem Binden der Variablen für die Funktionsparameter (siehe spätere Erklärungen)

Elemente von Erlang (8)

- Pattern Matching (Fortsetzung):
  - `[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:
  - 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)

Elemente von Erlang (9)

- Funktionsaufrufe:
  - `module:func(Arg1, Arg2, ... Argn)`
  - `func(Arg1, Arg2, .. Argn)`
- Erläuterungen:
  - Arg_1 .. Arg_n 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)

Elemente von Erlang (10)

- Modul-Deklaration:
  - ```
    -module(demo).
    -export([double/1]).

    double(X) -> times(X, 2).

    times(X, N) -> 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

    ```

## Elemente von Erlang (11)

- 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

## Elemente von Erlang (12)

- Definition von Funktionen:
  - ```
    func(Pattern1, Pattern2, ...) ->
        ... ;   % Vor dem ; steht der Rumpf
    func(Pattern1, Pattern2, ...) ->
        ... ;   % Das ; kündigt weitere Alternativen an
        ...     % Beliebig viele Alternativen möglich
    func(Pattern1, Pattern2, ...) ->
        ... .   % 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

    ```

## Elemente von Erlang (13)

- Bei imperativen Sprachen schreibt man:
  - ```
    function(Args)
    if X then
       Expression
    else if Y then
       Expression
    else
       Expression
    ```
- Bei funktionalen Sprachen schreibt man stattdessen:
  - ```
    function(X) ->
       Expression;
    function(Y) ->
       Expression;
    function(_) ->
       Expression.
    ```

Elemente von Erlang (14)

- Beispiel:
  - ```
    -module(mathStuff).
    -export([factorial/1, area/1]).

    factorial(0) -> 1;
    factorial(N) -> N * factorial(N-1).

    area({square, Side}) ->
       Side * Side;
    area({circle, Radius}) ->
       3 * Radius * Radius;        % almost :-)
    area({triangle, A, B, C}) ->
       S = (A + B + C)/2,
       math:sqrt(S*(S-A)*(S-B)*(S-C));
    area(Other) ->
       {invalid_object, Other}.
    ```

    ```

## Elemente von Erlang (15)

- Einige Beispielaufrufe:
  - `6> c(mathstuff).`
  - `{ok,mathstuff}`
  - `7> mathstuff:area(circle, 2).`
  - `** exception error: undefined function mathstuff:area/2`
  - `8> mathstuff:area({circle, 2}).`
  - `12`
  - `9> mathstuff:area({square, 2}).`
  - `4`
  - `10> mathstuff:area({triangle, 2, 3, 4}).`
  - `2.9047375096555625`
  - `11> mathstuff:area({2}).`
  - `{invalid_object,{2}}`
  - `12>`

## Elemente von Erlang (16)

- Was passiert wenn wir `mathstuff:factorial()` mit einem negativen Argument aufrufen?
  - Unsere Funktionsdefinition fängt diesen Fall nicht ab…
  - Der Interpreter reagiert nicht mehr… :o(
- Erste Reaktion: rette das Laufzeitsystem durch Eingabe von CTRL-G
  - ```
    User switch command
    02. --> h
    03.  c [nn]            - connect to job
    04.  i [nn]            - interrupt job
    05.  k [nn]            - kill job
    06.  j                 - list all jobs
    07.  s [shell]         - start local shell
    08.  r [node [shell]]  - start remote shell
    09.  q                 - quit erlang
    10.  ? | h             - this message
    11. -->
    ```

    ```

## Elemente von Erlang (17)

- Erste Reaktion (Fortsetzung):
  - 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

## Elemente von Erlang (18)

- Zweite Reaktion: Ergänze factorial() um zusätzliche Bedingung
  - „Beschütze“ die Funktion vor Endlosrekursion durch Ergänzung eines sogenannten Wächters (Guards) bei dem entsprechenden Fallmuster (Pattern)
  - ```
    factorial(0) -> 1;
    factorial(N) when N > 0 ->
                    N * factorial(N - 1).
    ```
- Erläuterungen:
  - Der Guard wird durch das Atom `when` und eine Bedingung vor dem Pfeil `->` formuliert
  - Vollständig „beschützte“ Klauseln können in beliebiger Reihenfolge angeordnet werden
  - ```
    factorial(N) when N > 0 ->
                    N * factorial(N - 1);
    factorial(0) -> 1.
    ```
  - Achtung: Ohne Guard führt diese Reihenfolge zu Endlosschleifen

Elemente von Erlang (19)

- Beispiele für Guards:
  - ```
    is_number(X)      % X is a number
    is_nteger(X)      % X is an integer
    is_float(X)       % X is a float
    is_atom(X)        % X is an atom
    is_tuple(X)       % X is a tuple
    is_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

Elemente von Erlang (20)

- Traversieren (“Ablaufen”) 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
    ```

    ```

## Elemente von Erlang (21)

- 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 “endrekursiv” ist (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)

Elemente von Erlang (22)

- „Identisch“ 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

Elemente von Erlang (23)

- 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)

Elemente von Erlang (24)

- 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

Elemente von Erlang (25)

- 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**

Elemente von Erlang (26)

- Standardbeispiel für Funktion höherer Ordnung:
  - ```
    -module(higher).
    -export([map/2, addOne/1]).
    map(_, [])-> [];
    map(F, [H|T]) -> [F(H)|map(F, T)].
    addOne(N)->N+1.
    ```
  - ```
    > higher:map(fun higher:addOne/1, [1,2,3,4]).
    [2,3,4,5]
    ```
- Die Funktion `map` akzeptiert als erstes Argument eine Funktion, die auf alle Elemente der als zweites Argument gegebenen Liste angewendet wird
- Beim Übergeben einer “normalen” Funktion an eine solche Funktion, wird diese zuerst mittels `fun` in eine anonyme Funktion gewandelt

    ```

## Elemente von Erlang (27)

- Beispiel für Funktion als Rückgabewert:
  - ```
    > MakeTest = fun(L) ->
        (fun(X) -> lists:member(X, L) end) end.
    ```
  - ```
    > Fruit = [apple,pear,orange].
    > IsFruit = MakeTest(Fruit).
    ```
  - ```
    > IsFruit(apple).
    true
    > IsFruit(dog).
    false
    ```
- Erläuterung:
  - MakeTest ist eine Variable, die auf eine anonyme Funktion verweist, die auf Eingabe einer Liste eine Funktion zurückgibt, welche testet, ob ein übergebener Wert in der Liste gespeichert ist (die Liste ist nun fest in die Testfunktion “eingebaut”)

Elemente von Erlang (28)

- Definition eigener Kontrollstrukturen:
  - ```
    for(Max, Max, F) -> [F(Max)];
    for(I, Max, F) -> [F(I)|for(I+1, Max, F)].
    ```
  - ```
    > for(1,10,F).
    [F(1), F(2), ..., F(10)]
    > for(1,10,fun(I) -> I end).
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    ```
- Erläuterung:
  - Die Erlang-Sprachdefinition umfasst keine speziellen Kommandos für übliche Kontrollstrukturen, da diese in ihrer allgemeinen Form im Kontext einer funktionalen Sprache wenig Sinn machen
  - Für einige übliche Aufgaben kann man jedoch mittels Funktionen höherer Ordnung leicht eigene Kontrollstrukturen definieren

    ```

## Elemente von Erlang (29)

- List Comprehensions (zu Deutsch: „Listen Begriffe/Verständnis“):
  - `> L = [1,2,3,4,5].`
    `[1,2,3,4,5]`
  - `> [2*X || X <- L ].`
    `[2,4,6,8,10]`
- Erläuterungen:
  - Mittels List Comprehensions können viele Aufgaben sehr kompakt formuliert werden (noch kompakter als mit map etc.)
  - Die Notation `[ F(X) || X <- L]` bedeutet “die Liste mit Werten `F(X)` wobei `X` jeweils der Liste `L` entnommen werden soll”
  - Mit dieser Notation kann die Funktion `map` noch kompakter formuliert werden: `map(F, L) -> [F(X) || X <- L].`

## Elemente von Erlang (30)

- Die allgemeine Form einer List Comprehension ist:
  - `[X || Qualifier1, Qualifier2, ...]`
- X ist ein beliebiger Ausdruck und jeder Qualifier ist entweder ein Generator oder ein Filter:
  - Generatoren werden mit `Pattern <- ListExpr` notiert, wobei `ListExpr` ein Ausdruck sein muss, der zu einer Liste von Termen evaluiert
  - Filters sind entweder Prädikate (Funktionen, die true oder false zurückgeben) or bool’sche Ausdrücke
- Beachte, dass der Generator-Teil einer List Comprehension durch Pattern Matching wie ein Filter wirken kann:
  - `> [ X || {a, X} <- [{a,1},{b,2},{c,3},{a,4},hello,"wow"]].`
    `[1,4]`

## Beispiel: Pythagoräische Tripel

- Ein pythagoräisches Tripel ist ein Tripel von Ganzzahlen {A, B, C}, so dass gilt: $A^2 + B^2 = C^2$
- Die folgende Funktion erzeugt alle Pythagoräischen Tripel, bei denen die Summe A + B + C ≤ N ist, wobei N der Funktion als Parameter übergeben wird:

```
pythag(N) ->
[{A,B,C} ||
A <- lists:seq(1,N),
B <- lists:seq(1,N),
C <- lists:seq(1,N),
A+B+C =< N,
A*A+B*B =:= C*C].
```

- Erläuterung: Die Funktion `lists:seq(1,N)` gibt eine Liste mit allen ganzen Zahlen zwischen 1 und N zurück

Elemente von Erlang (31)

- Case-Ausdrücke:

```
case Expression of
Pattern1 [when Guard1] -> Expr_seq1;
Pattern2 [when Guard2] -> Expr_seq2;
...
end
```

- Erläuterung:
  - Eigentlich kann man durch entsprechende Pattern bei der Funktionsdefinition (und entsprechend viele Deklarationen) ohne Case-Anweisung in Erlang auskommen
  - Andererseits führt Verwendung eines Case-Ausdrucks oft zu besser lesbaren Programmen
  - Mindestens eines der Pattern muss zutreffen, sonst wird eine Exception ausgelöst, da kein Wert ermittelt werden kann

Beispiel: Implementierung einer Filter-Funktion

```
filter(P, [H|T]) ->
case P(H) of
   true -> [H|filter(P, T)];
   false -> filter(P, T)
end;
filter(P, []) -> [].
```

- Die Filterfunktion wendet das Prädikat P (eine übergebene anonyme Funktion) auf alle Elemente der Liste an und gibt eine Ergebnisliste zurück, bei der nur die Elemente enthalten sind, für die das Prädikat den Wert true ergibt
- Das ist erheblich lesbarer als:

```
filter(P, [H|T]) -> filter1(P(H), H, P, T);
filter(P, []) -> [].

filter1(true, H, P, T) -> [H|filter(P, T)];
filter1(false, H, P, T) -> filter(P, T).
```

Elemente von Erlang (32)

- If-Ausdrücke:

```
if
Guard1 -> Expr_seq1;
Guard2 -> Expr_seq2;
...
end
```

- Erläuterung:
  - Auch die Verwendung eines If-Ausdrucks oft zu besser lesbaren Programmen
  - Der erste zu true evaluierte Guard führt zur Rückgabe des entsprechenden Ausdrucks
  - Mindestens einer der Guards muss zutreffen, sonst wird eine Exception ausgelöst, da kein Wert ermittelt werden kann

Effizienter Aufbau von Listen

- Bei Listen sollte grundsätzlich am Anfang eingefügt werden, da das Einfügen am Ende sehr ineffizient ist
- Der folgende Code führt zu sehr ineffizienten Berechnungen:
  - `List ++ [H] % uneffizient wenn List lang ist!!!`
- Es ist daher besser, H am Anfang einzufügen, und den Konkatenationsoperator „++“ möglichst nicht zu verwenden:
  - `[H | List]  % effizient, aber andere Reihenfolge`
- Sollte das bei einer rekursiven Funktionsbearbeitung zu einer „umgedrehten“ Ergebnisliste führen, sollte auf das Ergebnis die Funktion `lists:reverse/1` angewendet werden
- Die Funktion `lists:reverse/1` wird durch den Compiler durch sehr effizienten Code ersetzt (nicht durch den im Modul `lists` deklarierten Code, der nur zur Erläuterung dient)

Berechnung von Funktionen über Listen (1)

- Mit Hilfe der beiden Funktionen `foldl` und `foldr` können Funktionen über Listen berechnet werden
- Hierbei ist zu beachten, daß im Unterschied zu den Funktionen `filter` und `map`, bei welchen die auszuwertende Funktion jeweils auf jedes einzelne Element angewendet wurde, bei `foldl` und `foldr` ein Funktionswert als Ergebnis einer rekursiven Funktionsanwendung über die gesamte Liste berechnet wird
- Die übergebenen Funktionen sind bei `map` und `filter` einstellig
- Die übergebenen Funktionen sind bei `foldl` und `foldr` zweistellig
- Der Unterschied zwischen `foldl` und `foldr` besteht darin, daß die Funktion bei `foldl` linksassoziativ und bei `foldr` rechtsassoziativ ausgewertet wird

Berechnung von Funktionen über Listen (2)

- `foldl` und `foldr` werden rekursiv über die gesamte Liste angewendet. Daher muß ihnen als zusätzlicher Parameter ein Wert übergeben werden, welcher für den Basisfall der Rekursion benötigt wird.
  - Anwendungsbeispiele:

```
> lists:foldl(fun(X, Sum) -> X + Sum end, 0,
              [1,2,3,4,5]).
15
> lists:foldl(fun(X, Prod) -> X * Prod end, 1,
              [1,2,3,4,5]).
120
```

- Der zweite Parameter der anzuwendenden Funktion enthält den bisher akkumulierten Wert:
  - Im Basisfall der Rekursion von `foldl` bzw. `foldr` wird der zu akkumulierende Wert auf den o.g. zusätzlichen Wert gesetzt
  - Er sollte daher ideal ein Neutralelement bzgl. der Funktion sein

Berechnung von Funktionen über Listen (3)

- Definition von `foldl`:
  -
```
foldl(F, Accu, [Hd|Tail]) ->
  foldl(F, F(Hd, Accu), Tail);
foldl(F, Accu, []) when is_function(F, 2) -> Accu.
```
- Typsignatur der Funktion `foldl`:
  -
```
@spec foldl(fun((T, term()) -> term()),
            term(), [T]) -> term().
```
- Erläuterungen:
  - Im Basisfall der Funktion `foldl` wird mit dem Guard `is_function` geprüft, ob `F` eine Funktion der Stelligkeit 2 ist, d.h. eine Funktion, die 2 Argumente akzeptiert
  - Die Angabe einer Typsignatur mittels `@spec` erfolgt rein zu Dokumentationszwecken und wird vom Erlang-Compiler ignoriert
  - Der Erlang-Compiler ermittelt Typen grundsätzlich selbst

Berechnung von Funktionen über Listen (4)

- Definition von `foldr`:
  - ```
    foldr(F, Accu, [Hd|Tail]) ->
      F(Hd, foldr(F, Accu, Tail));
    foldr(F, Accu, []) when is_function(F, 2) -> Accu.
    ```
- Typsignatur der Funktion foldr:
  - ```
    @spec foldr(fun((T, term()) -> term()),
                term(), [T]) -> term().
    ```
- Erläuterungen:
  - Während `foldl` linksassoziativ arbeitet, wertet `foldr` die übergebene Funktion rechtsassoziativ über die Elemente der Liste aus, d.h.:
    - Bei `foldl` wird zuerst das erste Elemente der Liste mit dem Akkumulator verknüpft, dann das Ergebnis mit dem zweiten Element, etc.
    - Bei `foldr` wird zuerst das letzte Element mit dem Akkumulator verknüpft, dann das Ergebnis mit dem vorletzten Element, etc.

Berechnung von Funktionen über Listen (5)

- Zur besseren Lesbarkeit der folgenden Beipiele wird angenommen, dass `(-)` die Subtraktionsfunktion und `(^)` die Exponentiationsfunktion bezeichne (das ist kein gültiger Erlang-Code):
  - Beispiel 1: Die Subtraktion `(-)` ist linksassoziativ:
    - Soll also der Term `lists:foldl( (-), 10, [4, 2, 1])` berechnet werden, ist vollständig geklammert die folgende Berechnung durchzuführen: `((10 - 4) - 2) - 1`
    - Beachte: Bei diesem Beispiel ist die 10 kein neutrales Element
  - Beipiel 2: Die Exponentation `(^)` ist rechtsassoziativ, daher muß sie mittels `foldr` berechnet werden
    - Soll daher der Term `lists:foldr( (^), 1, [4, 3, 2])` berechnet werden, ist die folgende Berechnung durchzuführen: `4 ^ (3 ^ (2 ^ 1))`

Berechnung von Funktionen über Listen (6)

- Weitere Bemerkungen:
  - Bei `foldl` stellt der Rückgabewert der zu berechnenden Funktion den **ersten Parameter** für die nächste (Teil-)Berechnung dar
  - Bei `foldr` stellt der Rückgabewert der zu berechnenden Funktion den **zweiten Parameter** für die nächste (Teil-)Berechnung dar
  - Ist die auszuwertende Funktion assoziativ (also sowohl links- als auch rechts-assoziativ) so kann wahlweise `foldl` oder `foldr` verwendet werden
    - Da `foldl` endrekursiv ist, sollte man in diesem Fall aus Effizienzgründen immer `foldl` verwenden
    - Endrekursive Funktionen benötigen bei ihrer Ausführung weniger Speicherplatz, da ihr Ergebnis nach Rückkehr aus der Rekursion feststeht und das Laufzeitsystem somit keinen Aufrufstack verwalten muss (automatische Optimierung durch das Laufzeitsystem)

Einfache Listenfunktionen in Erlang (1)

- Die Funktion `elem` testet, ob ein als erster Parameter übergebener Wert ein Element der als zweiter Parameter übergebenen Liste ist:

```
elem(_, [])  -> false;
elem(E, [X|XS]) when E == X -> true;
elem(E, [X|XS]) -> elem(E, XS).
```

- Alternative Implementierung mit `if`:

```
elem2(_, [])  -> false;
elem2(E, [X|XS]) ->
if
 E == X -> true;
 true -> elem2(E, XS)
end.
```

Einfache Listenfunktionen in Erlang (2)

- Maximum einer Liste von Werten:
  - ```
    maxlist([X]) -> X;
    maxlist([X|XS]) ->
     M = maxlist(XS), % aus Effizienzgründen
     if
      X > M -> X;
      true -> M       % true entspricht else-zweig
     end.
    ```
- Erläuterungen:
  - Die Speicherung des Maximums der Restliste in der Variablen M führt zu einer besseren Laufzeiteffizienz, als wenn dieser Wert mehrfach berechnet würde (was natürlich auch möglich wäre)

Einfache Listenfunktionen in Erlang (3)

- Berechnen der Länge einer Liste:
  - ```
    len([]) -> 0;
    len([X|XS]) -> 1 + len(XS).
    ```
- Löschen des i.ten Elements einer Liste:
  - ```
    loesche([X|XS], 1) -> XS;
    loesche([X|XS], N) -> [X|loesche(XS, N-1)].
    ```
- Löschen des letzten Elements einer Liste:
  - ```
    loescheLetztes(XS) -> loesche(XS, len(XS)).
    ```
- Ist die folgende Implementierung evtl. besser?
  - ```
    loescheLetztes2([X]) -> [];
    loescheLetztes2([X|XS]) ->
      [X|loescheLetztes2(XS)].
    ```
- Falls ja, warum?

    ```

## Einfache Listenfunktionen in Erlang (4)

- Die folgende Funktion entfernt Duplikate in einer Liste:
  - `duplweg(L) -> lists:reverse(dupl([], L)).`

```
dupl(KS, []) -> KS;
dupl(KS, [X|XS]) ->
 B = elem2(X,KS),   % siehe vorige Folien
 if
  B==true -> dupl(KS, XS);
  true -> dupl([X|KS], XS)
 end.
```

  - Beispiel:

```
> dupl_weg [3, 5, 1, 3, 1, 2]
[3, 5, 1, 2]
```

Berechnung der Potenzmenge einer Menge (1)

- Definition:
  - Die Potenzmenge *PM* einer Menge *M* ist definiert als die Menge aller Teilmengen von *M*: $PM(M) := \{ K \mid K \subseteq M \}$
- Implementierung:
  - Mengen können beliebig viele Elemente enthalten und sind somit dynamische Strukturen
  - Daher bietet es sich an, Mengen als Listen zu implementieren
  - Mengen enthalten jedes Element nur einmal
  - Listen können Duplikate von Elementen enthalten
  - Im Folgenden ignorieren wir dieses Detail zunächst
  - Die Potenzmenge kann als Liste von Listen implementiert werden

Berechnung der Potenzmenge einer Menge (2)

- Aufgabe: Schreiben sie eine Erlang-Funktion *pm*, welche die Potenzmenge PM einer Menge *M* berechnet.
- Ansatz: Entwicklung der Implementierung mittels vollständiger Induktion - schon wieder? ;o)
  - **Induktionsanfang:** Die Potenzmenge der leeren Menge [ ] ist [[ ]]
  - **Induktionsvoraussetzung:** Die Potenzmenge aller Mengen mit *n* Elementen kann berechnet werden
  - **Induktionsschritt:** Die Potenzmenge aller Mengen mit *n* + *1* Elementen kann berechnet werden

    Beweis:
    - Sei *M* eine Menge mit *n* + *1* Elementen. Dann erhält man durch Wegnehmen eines Elementes $e \in M$ eine Menge *M'* mit *n* Elementen
    - Die Potenzmenge *PM'* := *PM(M')* kann berechnet werden.
    - Es gilt: $PM := PM' \cup \left\{ \{e\} \cup i \mid i \in PM' \right\}$

Berechnung der Potenzmenge einer Menge (3)

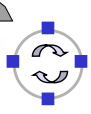
- Die folgende Funktion `pm` berechnet die Potenzmenge PM einer Menge M:

```
pm([]) -> [[]];
pm([X|XS]) ->
 P = pm(XS),        % laut Induktionsvoraussetzung
 lists:append(lists:map(fun(YS) -> [X|YS] end, P),
              P).  % laut Induktionsschritt
```

- Beispiele:

```
> pm([1,2]).
[[1,2],[1],[2],[]]

> pm [1,2,3]
[[1,2,3],[1,2],[1,3],[1],
[2,3],[2],[3],[]]
```



Vergleich: Berechnung der Potenzmenge in C++ (1)

```
// use vector instead of set, avoids operator<
// std::vector<int> v = { 1, 2, 3 }; auto a = potMenge(v);
template<typename T> std::vector<T> potMenge(const T& v) {
    assert(v.size() < sizeof(uint64_t));
    std::vector<T> r;
    for(uint64_t i = 0; i < (1ULL << v.size()); ++i) {
        r.push_back(T());
        T& z = r.back();
        for(std::size_t j = 0; j < v.size(); ++j)
            if(i & (1ULL << j))
                z.push_back(v[j]);
    }
    return r;
}
```

Vergleich: Berechnung der Potenzmenge in C++ (2)

```
template<typename T> std::vector<T> potMenge(const T& v) {
    std::vector<T> r;
    //  I : r = (), i = b_0 b_1 b_2 ... b_{|v|-1} = 00...0
    //  C : r = (z_0 ... z_i), ∀z_l = (v_{a_0}, ... v_{a_n}), ∀a_k ∈ z_l ⟺ b_k = 1, a_k < a_{k+1}, 0 ≤ a_k < |v| - 1
    for(uint64_t i = 0; i < (1ULL << v.size()); ++i) {
        r.push_back(T());
        T& z = r.back();
        //  I' : z = (), j = 0
        //  C' = z = (v_{a_0}, ... v_{a_n}), ∀a_k ∈ z ⟺ b_k = 1, a_k < a_{k+1}, 0 ≤ a_k < j
        for(std::size_t j = 0; j < v.size(); ++j)
            if(i & (1ULL << j))
                z.push_back(v[j]);
        //  → z = (v_{a_0}, ... v_{a_n}), ∀a_k ∈ z ⟺ b_k = 1, a_k < a_{k+1}, 0 ≤ a_k < |v| - 1
    }
    //  → r = (z_0 ... z_{2^i - 1}) [...]
    //  → r = Pot(v)
    return r;
}
```

Berechnung der Schnittmenge zweier Mengen (1)

- **Induktionsanfang:** Die Schnittmenge der leeren Menge mit einer beliebigen Menge ist die leere Menge
- **Induktionsvoraussetzung:** Für alle Mengen mit n Elementen kann die Schnittmenge mit jeder beliebigen Menge berechnet werden
- **Induktionsschritt:** Für alle Mengen mit n +1 Elementen kann die Schnittmenge mit jeder beliebigen Menge berechnet werden
  - **Beweis:** Sei M1 eine Menge mit n + 1 Elementen und M2 eine beliebige Menge. Dann erhält man durch Wegnehmen eines Elements h aus der Menge M1 eine Menge M1' mit n Elementen. Nun kann eine der folgenden Situationen vorliegen:
    - h ist Element von M2: In diesem Fall muss zunächst die Schnittmenge von M1' und M2 berechnet werden und anschließend h zu der berechneten Schnittmenge hinzugefügt werden.
    - h ist nicht Element von M2: In diesem Fall muss lediglich die Schnittmenge von M1' und M2 berechnet werden.

Berechnung der Schnittmenge zweier Mengen (2)

- Implementierung in Erlang:

```
cs([],_) -> [];
cs([X|XS],YS) ->
 B = elem2(X,YS),
 if
  B==true -> [X|cs(XS,YS)]; % laut IV
  true -> cs(XS,YS)         % laut IV
 end.
```

- Erläuterungen:
  - Die Implementierung folgt genau dem Beweis
  - Tatsächlich ist sie kürzer als der Beweis
  - Aber: wäre sie uns ohne den Beweis eingefallen – und selbst wenn, wären wir dann von ihrer Korrektheit überzeugt?
  - Wie wäre das bei der Berechnung der Potenzmenge?

Noch einmal Mergesort (1)

- Idee (→ Alg. & Prog.):
  1. Teile die zu sortierende Liste in zwei gleich große Teillisten
  2. Sortiere diese (rekursives Verfahren!)
  3. Mische die Ergebnisse

Induktionsannahme: Felder F der Länge *n/2* können sortiert werden.

Induktionsanfang: *n = 1*: trivial

Induktionsschritt: *n/2* ⇝ *n* :

- Teile das (Teil-)Feld *F* der Länge *n* in zwei gleich große Hälften
- Sortiere beide Teilfelder mit *n/2* Elementen nach Induktionsannahme
- Füge die beiden sortierten Teilfelder zu einem Feld zusammen, in dem jeweils das kleinste Element der beiden Teilfelder entfernt und in das zusammengefügte Feld aufgenommen wird (erfordert *n* Schritte)

- Rekurrenz: $T(n) = 2\,T(n/2) + n;$ also $a = 2; b = 2; f(n) = n;$

$$\Rightarrow \quad n^{\log_b a} = n^{\log_2 2} = n^1 = n \quad \Rightarrow \quad f(n) = \Theta(n^{\log_b a}) = \Theta(n)$$

$$\Rightarrow \quad T(n) = \Theta(n^{\log_2 2} \log n) = \Theta(n \log n)$$

Noch einmal Mergesort (2)

- Aufteilen einer Liste in zwei gleich große Teillisten:
  - ```
    split(Ls) -> split3(Ls, Ls, []).

    split3([], Ls1, Ls2) -> {reverse(Ls2) , Ls1};
    split3([_], Ls1, Ls2) ->{reverse(Ls2) , Ls1};
    split3([_,_|TT], [Ls1_H | Ls1_T], Ls2) ->
      split3(TT, Ls1_T, [Ls1_H | Ls2]).
    ```
- Erläuterungen:
  - Die Funktion split3 baut in jedem rekursiven Schritt zwei Elemente in der ersten übergeben Liste ab, und fügt dafür ein Element von der zweiten Liste in die dritte Liste ein
  - Ist die erste Liste leer bzw. hat sie nur noch ein Element, befinden sich n div 2 Elemente in der dritten Liste in falscher Reihenfolge
  - Diese werden nun umgedreht und als erste Hälfte zurückgegeben, sowie die zweite Liste mit den verbliebenen Elementen als zweite Hälfte
  - Im Vergleich zur Halbierung eines Feldes in Java (O(1)), wird hierbei ein Aufwand von O(n) erforderlich (plus zusätzlicher Speicheraufwand!)

    ```

## Noch einmal Mergesort (3)

- Mischen zweier sortierter Listen zu einer gemeinsamen Liste:
  - ```
    merge(_, [], Ls2) -> Ls2;
    merge(_, Ls1, []) -> Ls1;
    merge(Rel, [H1|T1], [H2|T2]) ->
     case Rel(H1, H2) of
      true  -> [H1 | merge(Rel, T1, [H2|T2])];
      false -> [H2 | merge(Rel, [H1|T1], T2)]
     end.
    ```
- Erläuterungen:
  - Die Funktion `Rel` realisiert das Ordnungskriterium (s.u.)
  - Somit ist `merge` eine Funktion höherer Ordnung

Noch einmal Mergesort (4)

- Die eigentliche Mergesort-Funktion:
  - ```
    msort(_, [])   -> [];
    msort(_, [H])  -> [H];
    msort(Rel, Ls) -> {Half1 , Half2} = split(Ls),
     merge(Rel, msort(Rel, Half1),
                msort(Rel, Half2)).
    ```
- Parametrisierung mit Ordnungsrelationen:
  - ```
    lte(X, Y) -> (X < Y) or (X == Y).
    gte(X, Y) -> (X > Y) or (X == Y).
    msort_lte(Ls) -> msort(fun lte/2, Ls).
    msort_gte(Ls) -> msort(fun gte/2, Ls).
    ```
- Laufzeitkomplexität:
  - Wie bei der Java Implementierung O(n log n), jedoch ist die Java-Variante aufgrund des einfacheren Aufteilens der Liste effizienter

    ```

## Noch einmal QuickSort (1)

- Idee (→ Alg. & Prog.):
  - Ähnlich wie MergeSort durch rekursive Aufteilung
  - Vermeidung des Mischvorgangs durch Aufteilung der Teillisten in zwei Hälften bezüglich eines Pivot-Elementes, wobei
    - In einer Liste alle Elemente größer als das Pivot-Element sind
    - In der anderen Liste alle Elemente kleiner sind

**Induktionsannahme:** Felder F der Länge < *n* können sortiert werden.

**Induktionsanfang:** *n* = *1*: trivial

**Induktionsschritt:** < *n* ⇝ *n* :

- Wähle ein beliebiges Pivot-Element
- Verschiebe nun alle Elemente, die kleiner als das Pivot-Element sind, in ein erstes Teilfeld *L* und alle Elemente, die größer als das Pivot-Element sind, in ein zweites Teilfeld *R*
- Sortiere beide Teilfelder mit < n Elementen nach Induktionsannahme
- Füge die sortierten Teilfelder zusammen: *L‘ + Pivot-Element + R‘*

## Noch einmal QuickSort (2)

- Quicksort in Erlang:

```
qsort([]) -> []; % empty lists are already sorted
qsort([Pivot|Rest]) ->
   qsort([Front || Front <- Rest, Front < Pivot])
   ++ [Pivot] ++
   qsort([Back || Back <- Rest, Back >= Pivot]).
```

- Erläuterungen:
  - Man beachte, dass im Vergleich zur Java-Implementierung keine eigene Partition-Funktion benötigt wird, da ihre Aufgabe von den beiden List Comprehensions übernommen wird
  - Die erste Comprehension erzeugt eine Liste mit allen Elementen von `Rest` die kleiner als `Pivot` sind; die zweite eine entsprechende Liste mit allen Elementen von `Rest` die größer gleich `Pivot` sind
  - Diese Erlang-Implementierung ist somit erheblich lesbarer! ☺
  - Leider ist sie nicht sehr effizient… 😩 Warum?

Noch einmal QuickSort (3)

- 1. Beide List Comprehensions müssen jeweils die volle Liste durchgehen
  - Wir schreiben daher doch eine eigene Partition-Funktion, welche die Liste nur einmal durchgeht:
  - ```
    part(_, [], {L, E, G}) -> {L, E, G};
    part(X, [H | T], {L, E, G}) ->
     if
       H < X -> part(X, T, {[H | L], E, G});
       H > X -> part(X, T, {L, E, [H | G]});
       true  -> part(X, T, {L, [H | E], G})
     end.
    ```
  - ```
    qsort2([]) -> [];
    qsort2([H | T]) ->
     {Less, Equal, Greater} = part(H, T, {[], [H], []}),
     qsort2(Less) ++ Equal ++ qsort2(Greater).
    ```

Noch einmal QuickSort (4)

- 2. Die Listen-Konkatenation „++“ ist recht uneffizient wenn die vordere Liste lang ist, und es müssen während der Berechnung zahlreiche Teillisten im Speicher angelegt und gespeichert werden
  - Wir werden daher die Ergebnisliste effizienter „aufsammeln“ durch Verwenden des „Akkumulators“-Konzepts
  - ```
    qsort3([]) -> [];
    qsort3([H | T]) -> qsort3_acc([H | T], []).
    ```
  - Die Arbeit wird also hauptsächlich von `qsort2_acc` übernommen, und `qsort3` stellt nur eine einfache Schnittstelle zur Verfügung
  - ```
    qsort3_acc([], Acc) -> Acc;
    qsort3_acc([H | T], Acc) ->
      part_acc(H, T, {[], [H], []}, Acc).
    ```
  - Es mag zunächst verwirren, dass hier keine rekursiven Aufrufe von `qsort3_acc` erfolgen, jedoch erfolgen diese von `part_acc` aus

Noch einmal QuickSort (5)

- Die hauptsächliche Logik des Algorithmus findet sich nun in `part_acc`
  - ```
    part_acc(_, [], {L, E, G}, Acc) ->
      qsort3_acc(L, (E ++ qsort3_acc(G, Acc)));
    part_acc(X, [H | T], {L, E, G}, Acc) ->
     if
      H < X -> part_acc(X, T, {[H | L], E, G}, Acc);
      H > X -> part_acc(X, T, {L, E, [H | G]}, Acc);
      true  -> part_acc(X, T, {L, [H | E], G}, Acc)
     end.
    ```
  - Solange die Liste des zweiten Parameters noch nicht vollständig bearbeitet wurde, arbeitet `part_acc` weitgehend wie `part`
  - Im Basisfall jedoch wird im zweiten rekursiven Aufruf von `qsort3_acc` der Akkumulator mit der sortierten Restliste gefüllt und
  - Die Liste `E` der Werte, die gleich dem Pivotwert sind, ist meist kurz

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 (→ 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 (→ Quicksort)
- Manche Aufgaben, die in imperativen Programmiersprachen sehr effizient und einfach lösbar sind (→ 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

    ```