MPGI1 CHEATSHEET - OPAL

Funktionale Programmierung: Keine Variabeln, nur **Funktionen**.
Funktionen haben dadurch keine Seiteneffekte (**zeitlos**, bei festen Parametern immer die selben Ergebnisse → großer Vorteil bei Mehrkernsystemen (siehe *Scala*))

Eine **Funktion (f : A -> B)** ist eine eindeutige **Zuordnung.**
A : Definitionsmenge - B : Zielmenge/Wertemenge
Jedem Element des **Definitionsbereich** (Teil von A) wird ein Element des **Wertebereichs/Bildmenge** (Teil von B) zugeordnet.

Deklaration: `FUN name : definitionsmenge -> wertemenge`
Definition: `DEF name(Argument) == 2*Argument`
Beispiel: `FUN doubleAdd : real ** real -> real`
`DEF doubleAdd(a,b) == 2*(a+b)`

Aufbau:
Jede Klasse besteht aus zwei Dateien: **Signature**(*.sign) und **Implementierung** (*.impl)
-> sie müssen den selben Namen haben, der auch nach SIGNATURE bzw IMPLEMENTATION stehen muss
-> Die Deklarationen gehören in die Signatur, die Definition in die Implementierung

Alle Deklarationen im **Signaturteil** sind **außerhalb** des Moduls **sichtbar**.
Alle Deklarationen im **Implementierungsteil** sind nur **innerhalb** des Moduls **sichtbar.** **(Hilfsfunktionen)**
Die Implementierung ist durch diese Trennung *versteckt* (**Opak**), sie muss für die Verwendung der Funktionen nicht bekannt sein. Der Signaturteil (+ Kommentare) würde in der Dokumentation stehen. Siehe auch **abstrakter Datentyp.**

Debug/ausführen
erfolgt mittels **oasys**, wobei die Datei zuerst geladen werden muss (>a *Dateiname*), fokusiert (>f *Dateiname.impl oder .sign*) und dann die Funktion ausgewählt werden kann(>e *Funktion*). Polymorphen-Strukturen lassen sich **nicht** laden! -> Testdatei erstellen
Mittels Sysdefs-Datei kann eine .exe erzeugt werden. (Befehle: ocs)

Kontrollstrukturen dienen in Opal zur Fallunterscheidung: **(FI NICHT VERGESSEN)**
`IF p THEN pTrue ELSE pFalse FI`
Dijkstra-IF:
Folge von IF-THEN Konstrukten, beendet mit FI **:** das **ELSE** wird gespart, Code wird leserlicher
IF p **THEN** pTrue
IF ~(p) **THEN** pFalse
FI

Rekursion
Eine Funktion heißt **rekursiv** (lat.*recurrere* „zurücklaufen“), wenn sie sich selbst (auch indirekt – über andere Funktionen) aufruft. Besteht aus **Rekursionsanfang(anker)**,**Rekursionsschritt**. Terminierung ist die Eigenschaft einer Funktion: ihre **Inkarnationskette** beim Aufrufen ist endlich.Die Rekursion sollte nicht offen sein, sondern monton in Richtung des Ankers verlaufen!
Rekursionsarten:

→ **Repetitive Rekursion**:	*f(x) = f(x-1)*	ein rekursiver Aufruf, Ergebnis steht innerhalb der Funktion fest: Endrekursiv „tail recursion“ Entspricht einer **Schleife**
→ **Lineare Rekursion**:	*f(x) = 1+f(x-1)*	höchstens **ein** rekursiver Aufruf, Ergebnis beim **Rausgehen** ermitteln
→ **Baumartige Rekursion**:	*f(x) = f(x-1) + f(x-2)*	mehrere rekursive Aufrufe Verzweigte Rekursion "fat recursion" Bsp: Standard Fibonnaci
→ **Geschachtelte Rekursion**:	f(x) = f(f(x*2))	Argumente der rekursiven Aufrufen sind weitere Aufrufe "compound recursion"
→ **Verschränkte Rekursion**:	*f(x) = g(x-1)* *g(x) = f(x-1)*	zwei oder mehr Funktionen rufen sich gegenseitig auf

Problem: Hoher Heapspeicherverbrauch, da jedesmal die Rücksprungaddressen gespeichert werden müssen - Iterationen sind bei gleicher funktionalität effektiver!

Datentypen:

SORT *DateType*	**definiert** einen **Name/Typ/Sorte.**
TYPE *DateType* == DateType	**deklariert** die **induzierte Signatur**.
DATA *DateType* == DateType	**definiert** und **deklariert induzierte Signatur**.

TYPE gehört in die SIGNATURE - **DATA** in die IMPLEMENTATION
SORT gibt nur den namen, keine weiteren Operationen

Abstrakte Datentypen In der Schnittstelle (Signatur) ist nur sichtbar was zur Benutzung verwendet wird, keine Hilfsfunktionen. Die interne realisierung ist Unbekannt und kann bei einem Update verändert werden ohne andere Klasse zu verändern.

Induzierte Signatur (wird vom Compiler erstellt)
besteht aus: **Typnamen, Konstruktoren, Selektoren** und **Diskriminatoren (bei allen 3!)**
Beispiel: **TYPE** Tier **==** Affe(größe : nat)
Tiger(größe: nat, hatHunger : bool)
Induziert:

SORT Tier	(Typnamen)
FUN Affe : nat → Tier	(Konstruktoren)
FUN Tiger : nat ** bool → Tier	
FUN hatHunger : Tier → bool	(Selektoren)
FUN größe : Tier → nat	
FUN Affe? : Tier → bool	(Diskriminatoren)
FUN Tiger? : Tier → bool	

alle Funktionen die auf auf **jedem** Element des Datentypes funktionieren heißen **total**
ansonsten **partiell:** **FUN** *größe* wäre in diesem Fall total, **FUN** *hatHunger* nicht!

Arten von Datentypen:

→ **Produkttyp**: Anzahl der **Konstruktoren** ist gleich **eins.**
Spezialfall des Summentypes
TYPE point == point(x : nat, y : nat, value: denotation)

→ **Aufzählunstyp**: Anzahl der **Selektoren** ist gleich **null.**
Grenzfall des Produkttypes
DATA Gemüse == Apfel Birne Elefant

→ **Varianten/Summentyp**: Anzahl der **Konstruktoren** ist **größer** als eins.
TYPE Tier == Ente
Tiger(hatHunger : bool)

Rekursiver Datentyp ist ein Datentyp dessen Definition auf sich selbst bezieht.
Beispiele: *Sequenzen*, *Bäume:* **TYPE Baum** == nil
Knoten(inhalt: data, links : **Baum**, rechts : **Baum**)

Polymorphen Typ, Parametrische/generische Struktur: d.h. eine Datenstruktur, die einen anderen Typ als Parameter übernimmt. Dadurch kann eine Datenstruktur für mehrere Typen **instanziiert** werden und somit leichter wiederverwendet werden können.
Beispiel: **SIGNATURE** seq[alpha] **SORT** alpha **TYPE** seq == .... Verwendung: seq[nat]

Pattern matching verbessert die lesbarkeit und nachvollziehbarkeit des Programms. Es gibt mehrere Definitionen einer Funktion, bei deren Konstruktoren als Argumente vorkommen.
FUN Gefährlichkeit : Tier → nat
DEF Gefährlichkeit(Affe(s)) == s*2
DEF Gefährlichkeit(Tiger(s, h)) == **IF** h **THEN** s*10000 **ELSE** s*3 **FI**
geht auch für einzelne Zahlen. der Unterstrich *f (_)* wird verwendet, wenn die Variabel egal ist.

Bubblesort:	O(n)	O(n^2)	O(n^2)	stabil	„leichtere" nach Oben
Selection Sort:	O(n^2)	O(n^2)	O(n^2)	(in)stabil	sucht das kleinste und tauscht
Insertion Sort:	O(n)	O(n^2)	O(n^2)	stabil	setze an der richtigen Stelle
Quicksort:	O(n log n)	O(n log n)	O(n^2)	(in)stabil	teile bzgl. Pivot und verkettet
Mergesort:	O(n log n)	O(n log n)	O(n log n)	stabil	sortiert beim zusammenfügen
Heapsort:	O(n log n)	O(n log n)	O(n log n)	instabil	sortiere mit einem Heap
Radixsort:	O(n * d)	Speicherbedarf: O(n)		stabil	geht nur auf zahlen

Stabile Sortierverfahren sind solche, die die relative Reihenfolge der Daten deren Sortierschlüssel gleich sind nicht verändern
In-situ: ohne Zusatzspeicher , **Ex-situ**: mit Zusatzspeicher

Suchverfahren

lineare Suche: liste wird komplett durchlaufen: O(n), geht immer
binäre Suche: Teilt in der Mitte und vergleicht, geht nur bei sortierten listen mit wahlfreien Zugriff: O(log n)
Interpolationssuche: schätzt die Position des gesuchten Elementes ab O(log(log n)) funktioniert nur gut auf gleichverteilten Daten ansonsten ist die Laufzeit O(n)
*neue Pos = left + (right – left) / (array ! right – array ! left) * (suchwert - array ! left)*
Hashing: Ohne Kollision O(1), im schlimssten Fall linear
Grover-Algorithmus: funktioniert auf unsortierten Daten mit O(sqrt(n)) geht nur auf Quantencomputern

High-Order-Function(HOF)/Funktional ist eine Funktion, die andere Funktionen als Parameter akzeptiert und/oder eine Funktion zurückliefert. „functions as first-class citizens“

Listenfunktionale:

→ **filter**: schmeißt die ungewünschten Elemente aus der Sequenz weg
FUN filter : (in -> bool) -> seq[in] -> seq[in]
`filter(even?)(1::2::3::4::<>) ↝ 2::4::<>`

→ **map**: wandelt alle Elemente aus der Sequenz um
FUN map : (a -> b) -> seq[a] -> seq[b]
`map(odd?)(1::2::<>) ↝ true::false::<>`

→ **reduce**: reduziert eine Sequenz auf ein Wert, rechts geklammer : a o (b o (...))
FUN reduce : (a ** b → b) ** b → seq[a] → seq[b]
`reduce((\\a,b. a + b),0)(1::2::3::<>) ↝ 6`

→ **zip**: fügt zwei Sequenzen nach dem Reißverschlussprinzip zusammen
FUN zip : (a ** b -> c) -> seq[a] ** seq[b] -> seq[c]
`zip(&)(1::2::<>)("one"::"two"::<>) ↝ &(1,"one")::&(2,"two")::<>`

*gibt es genauso für Tree: bei TreeReduce: (a ** b ** b -> b) ** b -> ....*

ProTip: Reduce ist mächtig genug, um andere Listenfunktionale zu simmulieren
reduce((\\E,Acc.**IF** P(E) **THEN** E::Acc **ELSE** Acc **FI**),<>) ≡ filter(P)
reduce((\\E,Acc.T(E)::Acc),<>) ≡ map(T)

Aufwandklassen

$\lim_{n\to\infty}\frac{f(n)}{g(n)}=c\in(\mathbb{R}\setminus\{0\})\Leftrightarrow f\in\Theta(g)\Leftrightarrow g\in\Theta(f)$ **f** wächst **gleich schnell** wie **g**

$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0\Leftrightarrow f\in O(g)\Leftrightarrow g\in\Omega(f)$ **f** wächst **höchstens so schnell** wie **g**

$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty\Leftrightarrow f\in\Omega(g)\Leftrightarrow g\in O(f)$ **f** wächst **mindestens so schnell** wie **g**

$f\in\Theta(g)\Leftrightarrow f\in\Omega(g)\wedge f\in O(g)$

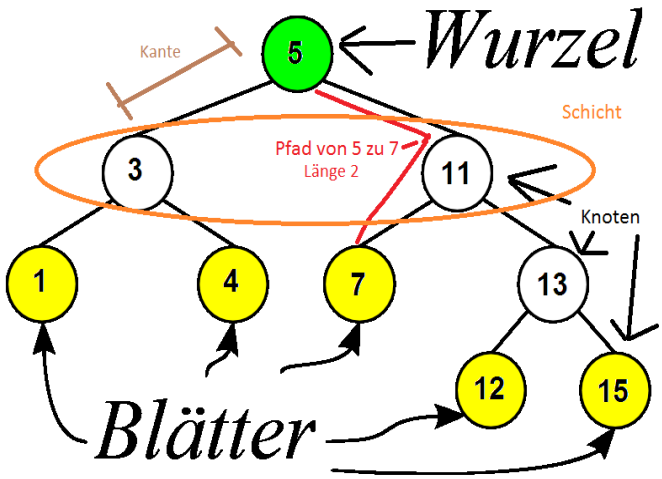
→ **Best-case: Ω(f)** = {**h** | ∀n.∃c,n₀.((c∈ℝ ∧ c≥0 ∧ n∈ℕ ∧ n≥n₀) ⇒ **h(n) ≥ c·f(n)**) }
→ **Average-case**: **Θ(f)** = {**h** | ∀n.∃c₁,c₂,n₀.
((c₁∈ℝ ∧ c₂∈ℝ ∧ c₁≥0 ∧ c₁≥0 ∧ n∈ℕ ∧ n≥n₀) ⇒ **c₁·f(n) ≥ h(n) ≥ c₂·f(n)**) }
→ **Worst-case: O(f)** = {**h** | ∀n.∃c,n₀.((c∈ℝ ∧ c≥0 ∧ n∈ℕ ∧ n≥n₀) ⇒ **h(n) ≤ c·f(n)**) }

Kostenfunktion und Aufwandklassen

Rekurenzfunktion aufstellen und vergleichen:

Gleichung für $\mathcal{K}(n)$		Ordnung von $\mathcal{K}(n)$
$\mathcal{K}(n) = \mathcal{K}(n-1) + bn^k$		$\mathcal{O}(n^{k+1})$
$\mathcal{K}(n) = c \cdot \mathcal{K}(n-1) + bn^k$	mit $c > 1$	$\mathcal{O}(c^n)$
$\mathcal{K}(n) = c \cdot \mathcal{K}(n/d) + bn^k$	mit $c > d^k$	$\mathcal{O}(n^{\log_d c})$
$\mathcal{K}(n) = c \cdot \mathcal{K}(n/d) + bn^k$	mit $c = d^k$	$\mathcal{O}(n^k \log n)$
$\mathcal{K}(n) = c \cdot \mathcal{K}(n/d) + bn^k$	mit $c < d^k$	$\mathcal{O}(n^k)$

Ein **Baum** besteht aus **Knoten** (Bsp. *11*,*13*) und **Kanten**. **Tiefe** ist die Entfernung vom **Wurzel** (Bsp. Für 3: *1*).
Schicht ist die Liste von Knoten mit bestimmter Entfernung vom Wurzel (Bsp. Für 1: *3*,*11*).
Pfadlänge: Anzahl der Kanten
Höhe ist die Länge des längsten Pfades
Größe ist die Anzahl der Knoten (Bsp. *9*).

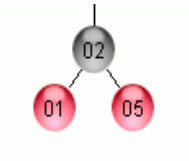


Binärbäume sind Bäume mit **höchstens zwei** Nachfolger.

```
DATA binTree == nil
                binTree(left:binTree,
                        val: type,
                        right:binTree)
```

Für binäre **Suchbäume** gilt: alle Nachfolger des linken Teilbaums < Wurzel < alle Nachfolger des rechten Teilbaums.
Probleme : Ungewichtet kann beim einfügen zu **unbalancierten** Bäumen entarten
Lösung: gewichtete Bäume -> AVL, B-Baum : (Rot/Schwarz, 2/3/4-Bäume)
balancierte Bäume garantieren O(log n) für einfügen, suchen, löschen und aufbau O(n log n)

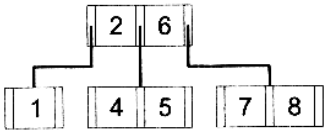
Rot-Schwarz-Baum:



annähernd Balanciert, der längste Pfad ist nie mehr als doppelt so lang wie der des kürzesten von der Wurzel aus

1. Jeder Knoten im Baum ist entweder rot oder schwarz.
2. Die Wurzel des Baums ist schwarz.
3. Alle Blatt-Knoten (NIL) sind schwarz.
4. Ist ein Knoten rot, so sind beide Kinder schwarz.
5. Jeder Pfad von einem gegebenen Knoten k zu seinen Blattknoten enthält die gleiche Anzahl schwarzer Knoten , k nicht mitgezählt (*Schwarzhöhe/Schwarztiefe*).

B-Bäume: - kein Binärbaum! -



B-Bäume sind keine Binärbaume, sondern ausgeglichene Mehrwegbäume. D.h. sie sind Bäume, die in einem Knoten mehrere Elemente speichern können.
Ein B-Baum der Ordnung m kann m-1 Elemente in einem Knoten speichern und hat folgende Eigenschaften:

1. Jeder Knoten hat höchstens m Kinder.
2. Jeder Knoten mit Ausnahme der Wurzel und der Blattknoten hat mindestens m/2 Kinder.
3. Die Wurzel hat mindestens 2 Kinder (oder ist ein Blattknote n).
4. Alle Blattknoten sind auf der gleichen Ebene, und tragen keine weiteren Informationen
5. Ein innerer Knoten mit k Kindern besitzt k-1 Schlüssel.

2-3-Bäume bzw. 2-4-Bäume sind B-Bäume der Ordnung 3 bzw. 4. Die 2 gibt die minimale Anzahl der Kinder pro Knoten an.

Für **AVL-Bäume(**Adelson-Velskii und Landis**)** gilt: binärer Suchbaum; **Höhedifferenz** aller Teilbäume **höchstens 1**.
Beim Einfügen/Löschen wird der Teilbaum **rotiert** falls die Eingenschaft verletzt wird (einfach oder doppelt) – am stärksten balanciert, suche ist am schnellsten

Für **Heaps** gilt: binäre **linksvolle** (nur im untersten Schicht dürfen die recht-äußere Elemente fehlen) Suchbäume;alle **Nachfolger** sing **großer** (bzw. kleiner) **als der Wurzel** der jeweiligen (Unter-)Baums.
Beim Einfügen/Löschen wird **heapify** aufgerufen falls die Eingenschaft verletzt wird.

Traversierungen (Binärbäume):

→ **preorder**: Wert, linker Unterbaum, rechter Unterbaum
→ **inorder**: linker Unterbaum, Wert, rechter Unterbaum
→ **postorder**: linker Unterbaum, rechter Unterbaum, Wert

Stack/Kellerspeicher/Stapel: Last-in first-out - Operationen: top, push, pop, empty?
Queue/Schlange: First-in first-out - Operationen: top, push, pop, empty?
Set: Menge, wobei jedes Element nur einmal vorkommen darf
Map/Assoziatives Array: Verknüpfung zwischen zwei Elementen

Ein- und Ausgabe

→ muss **sequentiel** erfolgen
→ Rückdaten müssen in `com[ ]` gepackt und dann mit **λ-Terme** (**Typisierung**!) ausgepackt werden
→ **Monade** können mit **&** verkettet werden
→ aufgrund der Rechtsassoziativität mussen viele **Klammen** gesetzt werden
→ **Dateien** sind wie Milchkarton: müssen **geöffnet** und **geschlossen** werden.

Wichtige Funktionen:

→ **ask**	gibt eine Denotation auf den Bildschirm aus und liest *bool/nat/int/real/char/denotation* von der Tastatur ein.
→ **writeLine**	gibt *bool/nat/int/real/char/denotation* auf den Bildschirm aus.
→ **&**	verknüpft zwei Kommandos
→ **open**	öffnet eine Datei
→ **readLine**	liest eine Zeile aus der Datei ein (ohne Zeilenumbruch)
→ **writeLine**	schreibt eine Zeile (plus Zeilenumbruch) in eine Datei
→ **close**	schließt eine Datei
→ **succeed**	packt das zurückgeliefene Wert in `com[ ]` ein

Beispiele:

```
ask("Gib eine Zahl ein: ") &
      \\x:nat. write("Du hast: " ++ `(x) ++ "eingegeben" ++ `(newline))

FUN firstLine : com[void]
DEF firstLine ==
ask("Dateiname: ") & (
      \\Filename. open(Filename, "r") & (
            \\File. readLine(File) & (
                  \\Line. writeLine("Erste Zeile: " ++ `(Line)) & close(File)
            )
      )
)
```

λ-Kalkül ist die Grundlage vom Opal – **Nicht behandelt im WS2011**

→ **α-Konversion**: Umbennenung von Parameter (Vorsicht: Freivariablen).
$\lambda x.t \equiv_\alpha \lambda y.t[y/x]$

→ **β-Reduktion**: Funktionsanwendung.
$(\lambda x.t)t' \equiv_\beta t[t'/x]$

→ **Substitution**: t[y/x] ersezt frei vorkommende x durch y in dem Term t.
$(\lambda x.(\lambda y.(y\ w)))[w/y] = (\lambda x.(\lambda y.(y\ w))[x/y]) = (\lambda x.(\lambda y.(y\ w)))$

→ **η-Reduktion**: Reduktion der Funktion (Vorsicht: Freivariablen).
$\lambda x.t \equiv_\eta t$

```
true  == λx.λy.x
false == λx.λy.y
λif.λthen.λelse.( if then else )

0 == λf.λx.x
1 == λf.λx.(f x)
n == λf.λx.(fⁿ x)
```

Tipps:
Wenn der compiler nicht versteht was die Funktion tut:
Urpsrungs-Annotation: `DEF xyz (a,b) == value(a) <'Nat b`
Typ-Annotation: `DEF foo(x:nat) == x*x` *und* `DEF foo(x:real) == x+x`

Tomasz Przedmojski © 2010; Changed by Georg Hieronimus 2011 Licensed under CC-NC-BY-SA